Table of Contents

Chapter 1. JSR-352 Support

As of Spring Batch 3.0 support for JSR-352 has been fully implemented. This section is not a replacement for the spec itself and instead, intends to explain how the JSR-352 specific concepts apply to Spring Batch. Additional information on JSR-352 can be found via the JCP here: https://jcp.org/en/jsr/detail?id=352

1.1. General Notes about Spring Batch and JSR-352

Spring Batch and JSR-352 are structurally the same. They both have jobs that are made up of steps. They both have readers, processors, writers, and listeners. However, their interactions are subtly different. For example, the `org.springframework.batch.core.SkipListener#onSkipInWrite(S item, Throwable t)` within Spring Batch receives two parameters: the item that was skipped and the Exception that caused the skip. The JSR-352 version of the same method (`javax.batch.api.chunk.listener.SkipWriteListener#onSkipWriteItem(List<Object> items, Exception ex)`) also receives two parameters. However the first one is a `List` of all the items within the current chunk with the second being the `Exception` that caused the skip. Because of these differences, it is important to note that there are two paths to execute a job within Spring Batch: either a traditional Spring Batch job or a JSR-352 based job. While the use of Spring Batch artifacts (readers, writers, etc) will work within a job configured via JSR-352's JSL and executed via the `JsrJobOperator`, they will behave according to the rules of JSR-352. It is also important to note that batch artifacts that have been developed against the JSR-352 interfaces will not work within a traditional Spring Batch job.

1.2. Setup

1.2.1. Application Contexts

All JSR-352 based jobs within Spring Batch consist of two application contexts. A parent context, that contains beans related to the infrastructure of Spring Batch such as the `JobRepository`, `PlatformTransactionManager`, etc and a child context that consists of the configuration of the job to be run. The parent context is defined via the `jsrBaseContext.xml` provided by the framework. This context may be overridden via the `JSR-352-BASE-CONTEXT` system property.

The base context is not processed by the JSR-352 processors for things like property injection so no components requiring that additional processing should be configured there.

1.2.2. Launching a JSR-352 based job

JSR-352 requires a very simple path to executing a batch job. The following code is all that is needed to execute your first batch job:

```
JobOperator operator = BatchRuntime.getJobOperator();
jobOperator.start("myJob", new Properties());
```

While that is convenient for developers, the devil is in the details. Spring Batch bootstraps a bit of infrastructure behind the scenes that a developer may want to override. The following is bootstrapped the first time `BatchRuntime.getJobOperator()` is called:

Bean Name	*Default Configuration*	*Notes*
dataSource	Apache DBCP BasicDataSource with configured values.	By default, HSQLDB is bootstrapped.
`transactionManager`	`org.springframework.jdbc.datasource.DataSourceTransactionManager`	References the dataSource bean defined above.
A Datasource initializer		This is configured to execute the scripts configured via the `batch.drop.script` and `batch.schema.script` properties. By default, the schema scripts for HSQLDB are executed. This behavior can be disabled via `batch.data.source.init` property.
jobRepository	A JDBC based `SimpleJobRepository`.	This `JobRepository` uses the previously mentioned data source and transaction manager. The schema's table prefix is configurable (defaults to BATCH_) via the `batch.table.prefix` property.
jobLauncher	`org.springframework.batch.core.launch.support.SimpleJobLauncher`	Used to launch jobs.
batchJobOperator	`org.springframework.batch.core.launch.support.SimpleJobOperator`	The `JsrJobOperator` wraps this to provide most of it's functionality.
jobExplorer	`org.springframework.batch.core.explore.support.JobExplorerFactoryBean`	Used to address lookup functionality provided by the `JsrJobOperator`.
jobParametersConverter	`org.springframework.batch.core.jsr.JsrJobParametersConverter`	JSR-352 specific implementation of the `JobParametersConverter`.
jobRegistry	`org.springframework.batch.core.configuration.support.MapJobRegistry`	Used by the `SimpleJobOperator`.
placeholderProperties	`org.springframework.beans.factory.config.PropertyPlaceholderConfigure`	Loads the properties file `batch-${ENVIRONMENT:hsql}.properties` to configure the properties mentioned above. ENVIRONMENT is a System property (defaults to hsql) that can be used to specify any of the supported databases Spring Batch currently supports.

None of the above beans are optional for executing JSR-352 based jobs. All may be overriden to provide customized functionality as needed.

1.3. Dependency Injection

JSR-352 is based heavily on the Spring Batch programming model. As such, while not explicitly requiring a formal dependency injection implementation, DI of some kind implied. Spring Batch supports all three methods for loading batch artifacts defined by JSR-352:

- Implementation Specific Loader - Spring Batch is built upon Spring and so supports Spring dependency injection within JSR-352 batch jobs.
- Archive Loader - JSR-352 defines the existing of a batch.xml file that provides mappings between a logical name and a class name. This file must be found within the /META-INF/ directory if it is used.
- Thread Context Class Loader - JSR-352 allows configurations to specify batch artifact implementations in their JSL by providing the fully qualified class name inline. Spring Batch supports this as well in JSR-352 configured jobs.

To use Spring dependency injection within a JSR-352 based batch job consists of configuring batch artifacts using a Spring application context as beans. Once the beans have been defined, a job can refer to them as it would any bean defined within the batch.xml.

XML Configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.springframework.org/schema/beans
                              http://www.springframework.org/schema/beans/spring-
beans.xsd
                              http://xmlns.jcp.org/xml/ns/javaee
                              http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd">

    <!-- javax.batch.api.Batchlet implementation -->
    <bean id="fooBatchlet" class="io.spring.FooBatchlet">
            <property name="prop" value="bar"/>
    </bean>

    <!-- Job is defined using the JSL schema provided in JSR-352 -->
    <job id="fooJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
        <step id="step1">
            <batchlet ref="fooBatchlet"/>
        </step>
    </job>
</beans>
```

Java Configuration

```
@Configuration
public class BatchConfiguration {

    @Bean
    public Batchlet fooBatchlet() {
        FooBatchlet batchlet = new FooBatchlet();
        batchlet.setProp("bar");
        return batchlet;
       }
}


<?xml version="1.0" encoding="UTF-8"?>
<job id="fooJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
    <step id="step1" >
        <batchlet ref="fooBatchlet" />
    </step>
</job>
```

The assembly of Spring contexts (imports, etc) works with JSR-352 jobs just as it would with any other Spring based application. The only difference with a JSR-352 based job is that the entry point for the context definition will be the job definition found in /META-INF/batch-jobs/.

To use the thread context class loader approach, all you need to do is provide the fully qualified class name as the ref. It is important to note that when using this approach or the batch.xml approach, the class referenced requires a no argument constructor which will be used to create the bean.

```
<?xml version="1.0" encoding="UTF-8"?>
<job id="fooJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
    <step id="step1" >
        <batchlet ref="io.spring.FooBatchlet" />
    </step>
</job>
```

1.4. Batch Properties

1.4.1. Property Support

JSR-352 allows for properties to be defined at the Job, Step and batch artifact level by way of configuration in the JSL. Batch properties are configured at each level in the following way:

```
<properties>
    <property name="propertyName1" value="propertyValue1"/>
    <property name="propertyName2" value="propertyValue2"/>
</properties>
```

`Properties` may be configured on any batch artifact.

1.4.2. @BatchProperty annotation

`Properties` are referenced in batch artifacts by annotating class fields with the `@BatchProperty` and `@Inject` annotations (both annotations are required by the spec). As defined by JSR-352, fields for properties must be String typed. Any type conversion is up to the implementing developer to perform.

An `javax.batch.api.chunk.ItemReader` artifact could be configured with a properties block such as the one described above and accessed as such:

```
public class MyItemReader extends AbstractItemReader {
    @Inject
    @BatchProperty
    private String propertyName1;

    ...
}
```

The value of the field "propertyName1" will be "propertyValue1"

1.4.3. Property Substitution

Property substitution is provided by way of operators and simple conditional expressions. The general usage is `#{operator['key']}`.

Supported operators:

- jobParameters - access job parameter values that the job was started/restarted with.
- jobProperties - access properties configured at the job level of the JSL.
- systemProperties - access named system properties.
- partitionPlan - access named property from the partition plan of a partitioned step.

```
#{jobParameters['unresolving.prop']}?:#{systemProperties['file.separator']}
```

The left hand side of the assignment is the expected value, the right hand side is the default value. In this example, the result will resolve to a value of the system property `file.separator` as `#{jobParameters['unresolving.prop']}` is assumed to not be resolvable. If neither expressions can be resolved, an empty String will be returned. Multiple conditions can be used, which are separated by a ';'.

1.5. Processing Models

JSR-352 provides the same two basic processing models that Spring Batch does:

- Item based processing - Using an `javax.batch.api.chunk.ItemReader`, an optional `javax.batch.api.chunk.ItemProcessor`, and an `javax.batch.api.chunk.ItemWriter`.
- Task based processing - Using a `javax.batch.api.Batchlet` implementation. This processing model is the same as the `org.springframework.batch.core.step.tasklet.Tasklet` based processing currently available.

1.5.1. Item based processing

Item based processing in this context is a chunk size being set by the number of items read by an `ItemReader`. To configure a step this way, specify the `item-count` (which defaults to 10) and optionally configure the `checkpoint-policy` as item (this is the default).

```
...
<step id="step1">
    <chunk checkpoint-policy="item" item-count="3">
        <reader ref="fooReader"/>
        <processor ref="fooProcessor"/>
        <writer ref="fooWriter"/>
    </chunk>
</step>
...
```

If item based checkpointing is chosen, an additional attribute `time-limit` is supported. This sets a time limit for how long the number of items specified has to be processed. If the timeout is reached, the chunk will complete with however many items have been read by then regardless of what the `item-count` is configured to be.

1.5.2. Custom checkpointing

JSR-352 calls the process around the commit interval within a step "checkpointing". Item based checkpointing is one approach as mentioned above. However, this will not be robust enough in many cases. Because of this, the spec allows for the implementation of a custom checkpointing algorithm by implementing the `javax.batch.api.chunk.CheckpointAlgorithm` interface. This functionality is functionally the same as Spring Batch's custom completion policy. To use an implementation of `CheckpointAlgorithm`, configure your step with the custom `checkpoint-policy` as shown below where `fooCheckpointer` refers to an implementation of `CheckpointAlgorithm`.

```
...
<step id="step1">
    <chunk checkpoint-policy="custom">
        <checkpoint-algorithm ref="fooCheckpointer"/>
        <reader ref="fooReader"/>
        <processor ref="fooProcessor"/>
        <writer ref="fooWriter"/>
    </chunk>
</step>
...
```

1.6. Running a job

The entrance to executing a JSR-352 based job is through the `javax.batch.operations.JobOperator`. Spring Batch provides its own implementation of this interface (`org.springframework.batch.core.jsr.launch.JsrJobOperator`). This implementation is loaded via the `javax.batch.runtime.BatchRuntime`. Launching a JSR-352 based batch job is implemented as follows:

```
JobOperator jobOperator = BatchRuntime.getJobOperator();
long jobExecutionId = jobOperator.start("fooJob", new Properties());
```

The above code does the following:

- Bootstraps a base `ApplicationContext` - In order to provide batch functionality, the framework needs some infrastructure bootstrapped. This occurs once per JVM. The components that are bootstrapped are similar to those provided by `@EnableBatchProcessing`. Specific details can be found in the javadoc for the `JsrJobOperator`.
- Loads an `ApplicationContext` for the job requested - In the example above, the framework will look in /META-INF/batch-jobs for a file named fooJob.xml and load a context that is a child of the shared context mentioned previously.
- Launch the job - The job defined within the context will be executed asynchronously. The `JobExecution’s` id will be returned.

All JSR-352 based batch jobs are executed asynchronously.

When `JobOperator#start` is called using `SimpleJobOperator`, Spring Batch determines if the call is an initial run or a retry of a previously executed run. Using the JSR-352 based `JobOperator#start(String jobXMLName, Properties jobParameters)`, the framework will always create a new JobInstance (JSR-352 job parameters are non-identifying). In order to restart a job, a call to `JobOperator#restart(long executionId, Properties restartParameters)` is required.

1.7. Contexts

JSR-352 defines two context objects that are used to interact with the meta-data of a job or step from within a batch artifact: `javax.batch.runtime.context.JobContext` and `javax.batch.runtime.context.StepContext`. Both of these are available in any step level artifact (`Batchlet`, `ItemReader`, etc) with the `JobContext` being available to job level artifacts as well (`JobListener` for example).

To obtain a reference to the `JobContext` or `StepContext` within the current scope, simply use the `@Inject` annotation:

```
@Inject
JobContext jobContext;
```

@Autowire for JSR-352 contexts

Using Spring's @Autowire is not supported for the injection of these contexts.

In Spring Batch, the `JobContext` and `StepContext` wrap their corresponding execution objects (`JobExecution` and `StepExecution` respectively). Data stored via `StepContext#setPersistentUserData(Serializable data)` is stored in the Spring Batch `StepExecution#executionContext`.

1.8. Step Flow

Within a JSR-352 based job, the flow of steps works similarly as it does within Spring Batch. However, there are a few subtle differences:

- Decision's are steps - In a regular Spring Batch job, a decision is a state that does not have an independent `StepExecution` or any of the rights and responsibilities that go along with being a full step.. However, with JSR-352, a decision is a step just like any other and will behave just as any other steps (transactionality, it gets a `StepExecution`, etc). This means that they are treated the same as any other step on restarts as well.
- `next` attribute and step transitions - In a regular job, these are allowed to appear together in the same step. JSR-352 allows them to both be used in the same step with the next attribute taking precedence in evaluation.
- Transition element ordering - In a standard Spring Batch job, transition elements are sorted from most specific to least specific and evaluated in that order. JSR-352 jobs evaluate transition elements in the order they are specified in the XML.

1.9. Scaling a JSR-352 batch job

Traditional Spring Batch jobs have four ways of scaling (the last two capable of being executed across multiple JVMs):

- Split - Running multiple steps in parallel.
- Multiple threads - Executing a single step via multiple threads.
- Partitioning - Dividing the data up for parallel processing (master/slave).
- Remote Chunking - Executing the processor piece of logic remotely.

JSR-352 provides two options for scaling batch jobs. Both options support only a single JVM:

- Split - Same as Spring Batch
- Partitioning - Conceptually the same as Spring Batch however implemented slightly different.

1.9.1. Partitioning

Conceptually, partitioning in JSR-352 is the same as it is in Spring Batch. Meta-data is provided to each slave to identify the input to be processed with the slaves reporting back to the master the results upon completion. However, there are some important differences:

- Partitioned `Batchlet` - This will run multiple instances of the configured `Batchlet` on multiple threads. Each instance will have it's own set of properties as provided by the JSL or the `PartitionPlan`
- `PartitionPlan` - With Spring Batch's partitioning, an `ExecutionContext` is provided for each partition. With JSR-352, a single `javax.batch.api.partition.PartitionPlan` is provided with an array of `Properties` providing the meta-data for each partition.
- `PartitionMapper` - JSR-352 provides two ways to generate partition meta-data. One is via the JSL (partition properties). The second is via an implementation of the `javax.batch.api.partition.PartitionMapper` interface. Functionally, this interface is similar to the `org.springframework.batch.core.partition.support.Partitioner` interface provided by Spring Batch in that it provides a way to programmatically generate meta-data for partitioning.
- `StepExecutions` - In Spring Batch, partitioned steps are run as master/slave. Within JSR-352, the same configuration occurs. However, the slave steps do not get official `StepExecutions`. Because of that, calls to `JsrJobOperator#getStepExecutions(long jobExecutionId)` will only return the `StepExecution` for the master.

  The child `StepExecutions` still exist in the job repository and are available via the `JobExplorer` and Spring Batch Admin.

- Compensating logic - Since Spring Batch implements the master/slave logic of partitioning using steps, `StepExecutionListeners` can be used to handle compensating logic if something goes wrong. However, since the slaves JSR-352 provides a collection of other components for the ability to provide compensating logic when errors occur and to dynamically set the exit status. These components include the following:

Artifact Interface	*Description*
`javax.batch.api.partition.PartitionCollector`	Provides a way for slave steps to send information back to the master. There is one instance per slave thread.
`javax.batch.api.partition.PartitionAnalyzer`	End point that receives the information collected by the `PartitionCollector` as well as the resulting statuses from a completed partition.
`javax.batch.api.partition.PartitionReducer`	Provides the ability to provide compensating logic for a partitioned step.

1.10. Testing

Since all JSR-352 based jobs are executed asynchronously, it can be difficult to determine when a job has completed. To help with testing, Spring Batch provides the `org.springframework.batch.test.JsrTestUtils`. This utility class provides the ability to start a job and restart a job and wait for it to complete. Once the job completes, the associated `JobExecution` is returned.