http://www.egovframe.go.kr/wiki/doku.php?id=egovframework:rte2:brte:batch_core:listener

Listener

Outline

You can add composition of jobs by setting event for each step (Job, Step, Chunk, Read, Process, Write). You can define such events in listener.

Description

JobListener(Intercepting Job Execution)

While execution of Job is underway, the user can take the best advantage of the events occurred to keep the user alerted. In eGovFramework, SimpleJob calls JobListener to do so.

```
public interface JobExecutionListener {
        void beforeJob(JobExecution jobExecution);
        void afterJob(JobExecution jobExecution);
}
```

JobListeners can be added to SimpleJob by way of the listeners of Job, as follows:

```
<job id="footballJob">
        <step id="playerload" parent="s1" next="gameLoad"/>
        <step id="gameLoad" parent="s2" next="playerSummarization"/>
        <step id="playerSummarization" parent="s3"/>
        <listeners>
                <listener ref="sampleListener"/>
        </listeners>
</job>
```

Note that afterJob is to be called regardless of success or failure of job. If needed, Jobexecution should define success and/or failure of suc job.

```
public void afterJob(JobExecution jobExecution){
        if( jobExecution.getStatus() == BatchStatus.COMPLETED ){
                //job success
        }
        else if(jobExecution.getStatus() == BatchStatus.FAILED){
                //job failure
        }
}
```

✔ Annotations falling under the interface JobExecutionListener

Annotations	Description
@BeforeJob	Called before Job
@AfterJob	Called after Job

StepListener(Intercepting Step Execution)

StepExecutionListener

StepExecutionListener is one of the most typical form of listener in execution of Step and keeps the user notified of the Step information before and after Step.

```
public interface StepExecutionListener extends StepListener {
        void beforeStep(StepExecution stepExecution);
        ExitStatus afterStep(StepExecution stepExecution);
}
```

By way of ExitStatus, the return type of afterStep, the user is granted an opportunity to edit exit-code returned.

Annotations for the foregoing interface are a sfollows:

✔ Annotations for the interface StepExecutionListener:

Annotations	Description
@BeforeStep	Called before Step
@AfterStep	Called after Step

ChunkListener

Chunk processes the items within the scope of transaction. While committing transaction, ChunkListener executes the implementation logic before or after Chunk processing.

```
public interface ChunkListener extends StepListener {
        void beforeChunk();
        void afterChunk();
}
```

The method beforeChunk is called before ItemReader implements reading, whereas afterChunk is called before rollback and after Chunk commitment.

✔ Annotations for the interface ChunkListener:

Annotations	Description
@BeforeChunk	Called before Chunk
@AfterChunk	Called after Chunk

✔ You may not apply ChunkListeners unless chunk is declared, such as TaskletStep.

ItemReadListener

You can keep the skip log to have some logics processed in the future. In case read error is thrown, ItemReaderListener processes skip.

```
public interface ItemReadListener<T> extends StepListener {
        void beforeRead();
        void afterRead(T item);
        void onReadError(Exception ex);
}
```

The method beforeRead is called before ItemReader implements reading.
The method afterRead is called to transfer the items read when the implementation of read is successful.

The method onReadError is called when error is thrown in implementation of reading. Exception information is thus thrown.

✔ Annotations for the interface ItemReadListener:

Annotations	Description
@BeforeRead	Called before Read
@AfterRead	Called after Read
@OnReadError	Called when error is thrown

ItemProcessListener

Item processing has its own listener like ItemReadListener does.

```
public interface ItemProcessListener<T, S> extends StepListener {
		void beforeProcess(T item);
		void afterProcess(T item, S result);
		void onProcessError(T item, Exception e);
}
```

The method beforeProcess is called before ItemProcessor executes processing.
The method afterProcess is called upon successful processing of items.
You can keep the error log as onProcessError is thrown when relevant.

✔ Annotations for the interface ItemProcessListener:

Annotations	Description
@BeforeProcess	Called before Process
@AfterProcess	Called after Process
@OnProcessError	Called when error is thrown in process

ItemWriteListener

You can have the listener called while ItemWriteListener writes items.

```
public interface ItemWriteListener<S> extends StepListener {
		void beforeWrite(List<? extends S> items);
		void afterWrite(List<? extends S> items);
		void onWriteError(Exception exception, List<? extends S> items);
}
```

The method beforeWrite is called before ItemWrite implements writing.
The method afterWrite is called to transfer the items written when the implementation of writing is successful.
The method onWriteError is called when error is thrown in implementation of writing. Exception information and items, aggregated in the form of a list, are thrown.

✔ Annotations for the interface ItemWriteListener:

Annotations	Description

@BeforeWrite	Called before Write
@AfterWrite	Called after Write
@OnWriteError	Called when error is thrown in writing

SkipListener

Despite itemReadListener, ItemProcessListener and ItemWriteListner keeping the user notified of errors thrown, Skip is always left unnoticed. Refer to the following interface that traces the item skipped:

```
public interface SkipListener<T,S> extends StepListener {
        void onSkipInRead(Throwable t);
        void onSkipInProcess(T item, Throwable t);
        void onSkipInWrite(S item, Throwable t);
}
```

onSkipInRead is called when skip takes place while reading. Note, however, that rollback must be kept notified as Skip has taken place more than once in that case.
The method onSkipInWrite is called wiehn Skip takes place in writing, in which case the concerned item, successfully read, provides itself as a factor.

✔ Annotations for the interface SkipListener:

Annotations	Description
@OnSkipInRead	Called when Skip takes place in reading
@OnSkipInWrite	Called when Skip takes place in writing
@OnSkipInProcess	Called when Skip takes place in Process

SkipListeners and Transactions

One of the most common options of using SkipListener is to keep the skipped item recorded for future use in manual processing or uniform processing.

✔ With transaction rollback highly likely, Spring guarantees you as follows:

1. The proper skip method is called once for each item (deemed appropriate for the error occurred).
2. SkipListener is always called before transaction commit, in which case the transaction resource called by listener cannot be rolled back as fallen through in ItemWriter.

Example

- Using Annotations

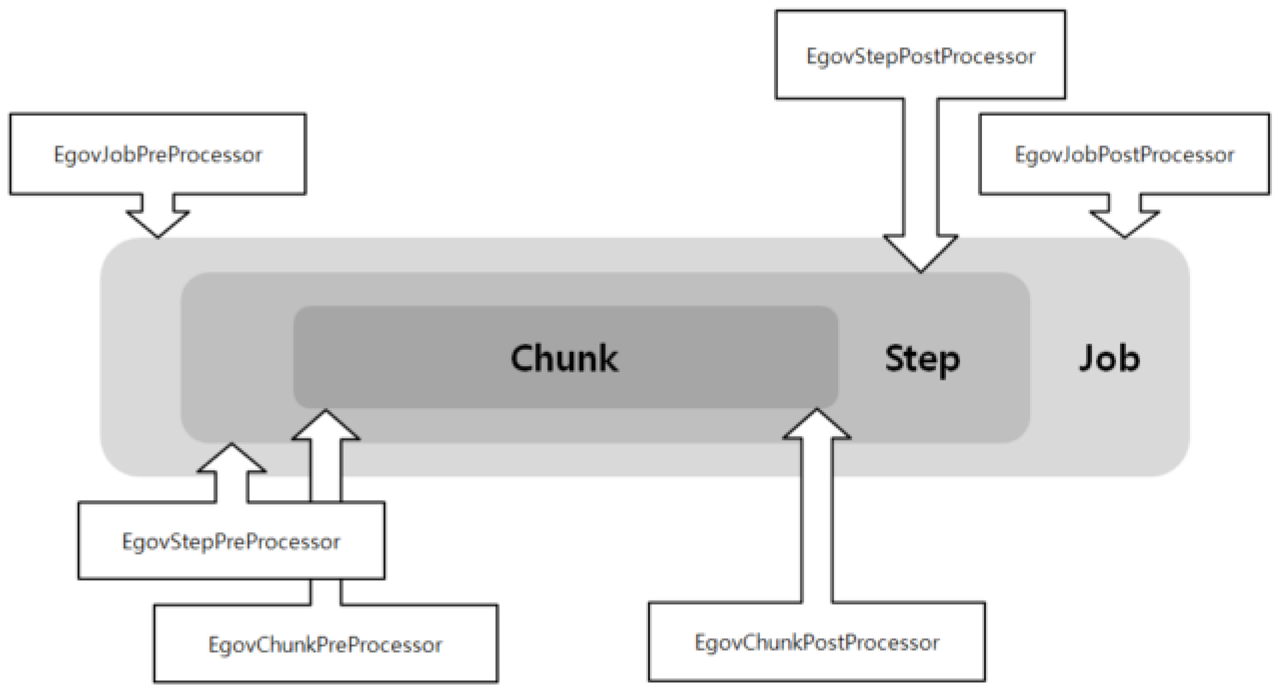
```
public class EventNoticeListener {
        @Autowired
        EgovEmailEventNoticeTrigger egovEmailEventNoticeTrigger;


        // Implemented upon completion of Job
        @AfterJob
        public ExitStatus sendJobNotice(JobExecution jobExecution) {

                egovEmailEventNoticeTrigger.invoke(jobExecution);
```

```
                    ...
            }

             // Implemented upon completion of Step
            @AfterStep
            public ExitStatus sendStepNotice(StepExecution stepExecution) {

                        egovEmailEventNoticeTrigger.invoke(stepExecution);
                        ...
            }

            // Implemented when Error takes place on Read
            @OnReadError
            public void sendErrorNotice(Exception e) {

                        egovEmailEventNoticeTrigger.invoke(e);
                        ...
            }
}
```

EgovPre/PostProcessor

eGovFramework categorizes various listeners by composition of batch (Job, Step, Chunk) and pre/post processing to provide the processor separately identifiable. Processors are called by <listener> of the configuration file for Job, calling configuration as follows:

✔ Note that the following EgovSampleXXXProcessor assumes inheritance of eGovFramework Procesor.



Job Processor

Category

Class	Method	Parameter	Description
EgovJobPreProcessor	beforeJob()	JobExecution	Called before Job

EgovJobPostProcessor	afterJob()	JobExecution	Called after Job

```
public class EgovJobPreProcessor extends JobExecutionListenerSupport {
		/**
		  * Parts called before execution of Job
		  */
		public void beforeJob(JobExecution jobExecution) {

		}
}
public class EgovJobPostProcessor extends JobExecutionListenerSupport {
		/**
		  * Parts called after execution of Job
		  */
		public void afterJob(JobExecution jobExecution) {

		}
}
```

Settings

The Job Processor defined by the user that inherited the foregoing class configures, using <listeners>, as follows:

```
<job id="ProcessorJob" xmlns="http://www.springframework.org/schema/batch">
		<listeners>
				<listener ref="jobListener" />
		</listeners>
		<step id="ProcessorStep">
				<tasklet>
						<chunk reader="itemReader" writer="itemWriter" commit-interval="2"/>
				</tasklet>
		</step>
</job>

<bean id="jobListener" class="User-defined job processor class" />
```

Step Processor

Category

Class	Method	Parameter	Description
EgovStepPreProcessor	beforeStep()	StepExecution	Called before Step
EgovStepPostProcessor	afterStep()	StepExecution	Called after Step

```
public class EgovStepPreProcessor<T, S> extends StepListenerSupport<T, S> {
		/**
		  * Parts called before execution of Step
		  */
		public void beforeStep(StepExecution stepExecution) {

		}
}
public class EgovStepPostProcessor<T, S> extends StepListenerSupport<T, S> {
		/**
		  * Parts called after execution of Step
		  */
		public ExitStatus afterStep(StepExecution stepExecution) {
				return null;
```

```
        }
}
```

Settings

The Step Processor defined by the user that inherited the foregoing class configures, using <listeners>, as follows:

```
<job id="ProcessorJob" xmlns="http://www.springframework.org/schema/batch">
        <step id="ProcessorStep">
                <tasklet>
                        <chunk reader="itemReader" writer="itemWriter" commit-interval="2"/>
                </tasklet>
                <listeners>
                        <listener ref="stepListener" />
                </listeners>
        </step>
</job>

<bean id="stepListener" class="User-defined step processor class" />
```

Chunk Processor

Category

Class	Method	Parameter	Description
EgovChunkPreProcessor	beforeChunk()	N/A	Called before Chunk
EgovChunkPostProcessor	afterChunk()	N/A	Called after Chunk

```
public class EgovChunkPreProcessor extends ChunkListenerSupport {
        /**
          * Parts called before execution of Chunk
          */
        public void beforeChunk() {

        }
}
public class EgovChunkPostProcessor extends ChunkListenerSupport {
        /**
          * Parts called after execution of Chunk
          */
        public void afterChunk() {

        }
}
```

Settings

The Chunk Processor defined by the user that inherited the foregoing class configures, using <listeners>, as follows:

```
<job id="ProcessorJob" xmlns="http://www.springframework.org/schema/batch">
        <step id="ProcessorStep">
                <tasklet>
                        <chunk reader="itemReader" writer="itemWriter" commit-interval="2">
                                <listeners>
                                        <listener ref="chunkListener" />
                                </listeners>
                        </chunk>
                </tasklet>
        </step>
</job>

<bean id="chunkListener" class="User-defined chunk processor class" />
```

Example

Pre/Post Process Examples

References

http://static.springsource.org/spring-batch/reference/html/configureJob.html#interceptingJobExecution
http://static.springsource.org/spring-batch/reference/html/configureStep.html#interceptingStepExecution