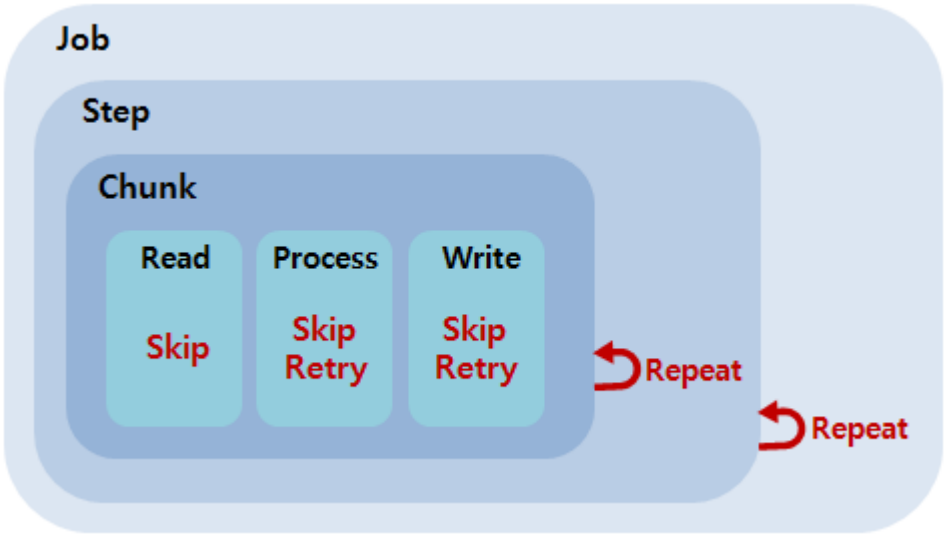
http://www.egovframe.go.kr/wiki/doku.php?id=egovframework:rte2:brte:batch_core:skip_repeat_retry

Skip/Retry/Repeat

Outline

Skip, Retry and Repeat are required to work the batch in an efficient manner. Securing iterated execution of step and chunk, Repeat is followed by reading, processing and writing of data where Skip and Retry can wisely be used for efficiency's sake. See the following figure for how and where these work:



Description

Skip

With Skip, the data is skipped unprocessed when the pre-determined exception is thrown to prevent recognition of small-time failures.

Configuring Skip Logic

Skip is configured in <chunk> of the configuration file of Job. With Skip, 'skip-limit' and (optionally) <skippable-exception-classes> are added to the chunk configuration (reader, processor, writer, commit-interval). See the following table for more details:

Item	Description
skip-limit	The maximum count of Skip. Must input value other than 0 (being default value).
<skippable-exception-classes> <include>	Designates the scope of exception to be skipped.
<skippable-exception-classes> <exclude>	Designates the exception that won't actuate Skip among sub-exceptions.

✔ Make sure the expert to the concerned data works configuration of Skip for <skippable-exception-classes>. You need to keep in mind that skipping the data processing of a vendor company is quite acceptable, while data processing in a banking institute turns tragic.

Refer to the following example for how data is read, via FlatFileItemReader, and skipped when

FlatFileParseException (<include>) is thrown to the maximum 10 times. FlatFileParseException thrown for the eleventh time would thus be recognized as a failure.

```
<step id="step1">
  <tasklet>
    <chunk reader="flatFileItemReader" writer="itemWriter" commit-interval="10" skip-limit="10">
      <skippable-exception-classes>
        <include class="org.springframework.batch.item.file.FlatFileParseException"/>
      </skippable-exception-classes>
    </chunk>
  </tasklet>
</step>
```

✔ A separate counter is available to count skips for read, process and write, with skip-limit counting sum of those skips.

Note that the exception not being FlatFileItemReader is left uncounted. If you want to count such an exception, wisely configure <exclude> to call error when the concerned exception or any sub-exception thereof is thrown:

```
<step id="step1">
  <tasklet>
    <chunk reader="flatFileItemReader" writer="itemWriter" commit-interval="10" skip-limit="10">
      <skippable-exception-classes>
        <include class="java.lang.Exception"/>
        <exclude class="java.io.FileNotFoundException"/>
      </skippable-exception-classes>
    </chunk>
  </tasklet>
</step>
```

Examples

Example of Skip

Retry

With Retry, the processing of data is retried when the pre-determined exception is thrown while data is processed and written, to prevent recognition of small-time failures.

Configuring Retry Logic

Retry is configured in <chunk> of the configuration file of Job. With Retry, 'retry-limit' and (optionally) <retryable-exception-classes> are added to the chunk configuration (reader, processor, writer, commit-interval). See the following table for more details:

Item	Description
retry-limit	Designates the maximum retry count
<retryable-exception-classes> <include>	Designates the scope of exception to be retried.
<retryable-exception-classes> <exclude>	Designates the exception that won't actuate Retry among sub-exceptions.

✔ Note Retry takes place only in Item Processing and Item Writing.

FlatFileParseException thrown in data processing is rather skipped, instead of being retried. However, you can wisely retry data processing and writing to resolve DeadlockLoserDataAccessException thrown. This comes in handy when

an error is thrown to approach of a new process to the locked data being processed.

✔ Successfully read data is reposited in the cache. You may load the cache data to get back to the process where you have first encountered trouble.

✔ Retryable exception gives rise to rollback that affects performance and brings about latency.

Refer to the following example for how Retry takes place when DeadlockLoserDataAccessException (<include>) is thrown to the maximum 3 times. DeadlockLoserDataAccessException thrown for the eleventh time would thus be recognized as a failure. In the following example you can note a pair of retry attempts are left:

```
<step id="step1">
  <tasklet>
    <chunk reader="itemReader" writer="itemWriter" commit-interval="2" retry-limit="3">
      <retryable-exception-classes>
        <include class="org.springframework.dao.DeadlockLoserDataAccessException"/>
      </retryable-exception-classes>
    </chunk>
  </tasklet>
</step>
```

Other Configurations

- retry-policy : The customized retry policy can be applied, ignoring retry-limit defined in <chunk>

```
<job id="retryPolicyJob" xmlns="http://www.springframework.org/schema/batch">
  <step id="retryPolicyStep">
    <tasklet>
       <chunk reader="reader" writer="writer" commit-interval="100" retry-policy="retryPolicy" />
    </tasklet>
  </step>
</job>

<bean id="retryPolicy" class="org.springframework.batch.retrypolicy.SimpleRetryPolicy">
  <property name="maxAttempts" value="3" />
</bean>
```

- backOffPolicy : Customized interval of data processing (unit: ms) can be applied

```
<job id="job1" job-repository="jobRepository">
  <step id="step1" parent="stepParent">
        ...
  </step>
</job>

<bean id="stepParent" class="org.springframework.batch.core.step.item.FaultTolerantStepFactoryBean"
abstract="true">
  <property name="backOffPolicy">
    <bean class="org.springframework.batch.retry.backoff.FixedBackOffPolicy"
      <property name="backOffPolicy" value="2000" />
    </bean>
  </property>
</bean>
```

Retry Template

You may from time to time need that a failure is immediately followed by retry, in the event of web services or ROM services with troubled network and DeadLockLoserException thrown. In Spring, you may establish a retry template to do so.

```
public interface RetryOperations {

    <T> T execute(RetryCallback<T> retryCallback) throws Exception;

    <T> T execute(RetryCallback<T> retryCallback, RecoveryCallback<T> recoveryCallback)
        throws Exception;

    <T> T execute(RetryCallback<T> retryCallback, RetryState retryState)
        throws Exception, ExhaustedRetryException;

    <T> T execute(RetryCallback<T> retryCallback, RecoveryCallback<T> recoveryCallback,
        RetryState retryState) throws Exception;

}
```

Callback is a simple interface where business logics are contained.

```
public interface RetryCallback<T> {

    T doWithRetry(RetryContext context) throws Throwable;

}
```

When an exception is thrown immediately after callback is executed, you can have it retried until success or terminate on an immediate basis. To do so, you can implement Retry Template as follows:

```
RetryTemplate template = new RetryTemplate();

template.setRetryPolicy(new TimeoutRetryPolicy(30000L));

Foo result = template.execute(new RetryCallback<Foo>() {

    public Foo doWithRetry(RetryContext context) {
        // Do stuff that might fail, e.g. webservice operation
        return result;
    }

});
```

Note that the foregoing example calls the web service for result to be returned to the user. In the event of failure of calls, the service is retried until it gets timed out.

RetryContext

Being the parameter for RetryCallback, RetryContext is ignored in callback. Where deemed necessary, however, you can use this as an attribute bag while repetition is underway.

RecoveryCallback

When Retry is used up, you can hand off a right to control callback to RecoveryCallback, like the way you've worked for RetryContext.

```
Foo foo = template.execute(new RetryCallback<Foo>() {
    public Foo doWithRetry(RetryContext context) {
        // business logic here
    },
```

```
   new RecoveryCallback<Foo>() {
      Foo recover(RetryContext context) throws Exception {
                // recover logic here
      }
});
```

When the business logic gets failed before the decision for stopping template is made, the client is granted an opportunity to work on other processes by way of RecoveryCall.

Stateless Retry

Stateless Retry is an endless loop of retries until success. Contrary to RetryContext where the status for retry and cancellation is defined, you do not need to secure global access for Stateless Retry as the state information is reposited in the stack. There's a big difference between Stateless Retry and Stateful Retry, described below, in that Stateless Retry is unique RetryPolicy to it (when using RetryTemplate, you can controll both Stateless Retry and Stateful Retry). When Retry falls through in Stateless Retry, callback is implemented in the same thread.

Stateful Retry

There are a handful of considerations to failures that turns the resources to be transacted invalid. (For most of the cases,) Stateful Retry is applied to simple remote call that involves no transaction and renewal of database using hibernate, where transaction is rolled back to commence a valid transaction again and throw exception as soon as the failure takes place.

Rethrowing an exception to rollback renders the method RetryOperations.execute and the context in stack lost, in which case Stateless Retry shall not be considered. In avoidance of such a loss, you need to get the context out of the stack for reposition in the hip repository, for which Spring offers RetryContextCache. Involving simple reposition process using Map, RetryContextCache is worth a consideration in cluster environment where a multitude of processes (of course RetryContextCache does not fit some cluster environments).

RetryOperations is responsible for, among others, recognition of failed tasks returned in the form of a brand-new execution (and, in general, in a brand-new transaction). To do so, Spring offers abstraction of Retry State to better identify the various statuses of calls made for retry. In order to confirm such statuses, the user is required to provide RetryState objects that are designed to return the unique key that identifies Item. Note the identifier is used as a key in RetryContextCache.

✔ Caution

Special attention is required when attempting ti implement Object.equals() and Object.hashCode() for keys returned by RertyState. One another way recommended is to use the business key so that messageID can be used for JMS messaging.

Retry Policies

RetryPolicy defines retry and failure of the method Execute in RetryTemplate. Serving as a factory of Retry Context, RetryTemplate is responsible for using the current policies with priority and sending such policies to RetryCallback. When callback falls through, RetryTemplate calls RetryPolicy to renew the status (and reposit the status in RetryContext), and inquires the policy by calling Retry Policy. When the follow-up attempt is not made (in the event of restriction or timeout), RetryPolicy will then be responsible for administration of the statuses used up, whereby RetryExhastedException is thrown and the associated transaction is rolled back. To be specific, RetryPolicy from time to time attempt to recover when it deems the transaction is maintainable.

✔ Tip

Where there is a failure, the feasibility of retry must be determined. For such exceptions thrown in the form of business logic, retry is not a good option. While you can keep the business logic intact by aggressively attempting retry, you would not retry something that will have been failed.

Spring thus offers Stateless RetryPolicy such as SimpleRetryPolicy and TimeoutRetryPolicy exemplified as follows. SimpleRetryPolicy permits retries provided for in the exception list, to the extent not in excess of the maximum count. In SimpleRetryPolicy there is “Critical” exception list which takes precedence over the general exception list.

```
SimpleRetryPolicy policy = new SimpleRetryPolicy(5);
// Retry on all exceptions (this is the default)
policy.setRetryableExceptions(new Class[] {Exception.class});
// ... but never retry IllegalStateException
policy.setFatalExceptions(new Class[] {IllegalStateException.class});
// Use the policy...
RetryTemplate template = new RetryTemplate();
template.setRetryPolicy(policy);
template.execute(new RetryCallback<Foo>() {
    public Foo doWithRetry(RetryContext context) {
        // business logic here
}
});
```

Keep in mind the user should customize the retry policy to re-defined decisions.

Backoff Policies

Some transient failures are not repeated when reattempted. In this connection, when RetryCallback is failed:

```
public interface BackoffPolicy {

    BackOffContext start(RetryContext context);

    void backOff(BackOffContext backOffContext)
        throws BackOffInterruptedException;

}
```

Discretion is on you to determine the method of BackoffPolicy, provided you abide by the policy Object.waite(). In avoidance of a couple retry locked and fallen through, you can work Backoff Policies by way of ExponentialBackoffPolicy.

Listeners

You may from time to time need to receive additional callbacks while a multitude of iterations are underway. To do so, Spring offers RetryListener along with RetryTemplate by which the user can register RetryListener, transferred along with the RetryContext and Throwable to callback.

```
public interface RetryListener {

    void open(RetryContext context, RetryCallback<T> callback);

    void onError(RetryContext context, RetryCallback<T> callback, Throwable e);

    void close(RetryContext context, RetryCallback<T> callback, Throwable e);
}
```

Both open callback and close callback are called before and after retry, while onError() is applied to independent RetryCallback call. Also, the method Close receives Throwable when the last error is thrown by RetryCallback. Several listeners are enlisted in a certain sequence. As for the method Open, the same sequence is applied for call, with onError() and close() called in reverse order.

Declarative Retry

You may from time to time encounter business processes that you want to restart every time it occurs, such as remote service calls. Spring offers AOP Interceptor that covers RetryOperations. RetryOperationsInterceptor executes the method intercepted, while the RetryPolicy in RetryTemplate works to reattempt the failure.
Refer to the following example of declarative retry that involves use of Spring AOP Namespace when service call is re-attempted:

```
<aop:config>
    <aop:pointcut id="transactional" expression="execution(* com..*Service.remoteCall(..))" />
    <aop:advisor pointcut-ref="transactional" advice-ref="retryAdvice" order="-1"/>
</aop:config>

<bean id="retryAdvice" class="org.springframework.batch.retry.interceptor.RetryOperationsInterceptor"/>
```

In the foregoing example the basic RetryTemplate in teh interceptor is used. Keep in mind that RetryTemplate should be applied to change either listener or policy.

Examples

Retry Example

Repeat

A batch comprises iterated execution of steps and chunks underneath them, where Repeat Policy defines such execution in implementation of batches.

RepeatTemplate

Batch processing is simple optimization or iterated repetition of job elements. Spring strategically generalizes such repetition and features RepeatOperations to provide the iterator framework. Refer to the following for how RepeatOperations are made of:

```
 public interface RepeatOperations {
    RepeatStatus iterate(RepeatCallback callback) throws RepeatException;
}
```

Callback is a simple interface that helps interated business logics to be added.

```
public interface RepeatCallback {
    RepeatStatus doInIteration(RepeatContext context) throws Exception;
}
```

Execution of callback is iterated until the repetition is deemed complete, while the relevant interface returns either RepeatStatus.CONTINUABLE or RepeatStatus.FINISHED. RepeatStatus transfers the call information for repetition. In general, implementation of RepeatOperations is made by confirming RepeatStatus that defines either repetition or conclusion. When intending to send the signal FINISHED to the caller, callback returns ExistStatus.FINISHED. As for implementation of RepeatTemplate, refer to the following example:

```
RepeatTemplate template = new RepeatTemplate();
template.setCompletionPolicy(new FixedChunkSizeCompletionPolicy(2));
template.iterate(new RepeatCallback() {
                        public ExitStatus doInIteration(RepeatContext context) {
                        // Do stuff in batch...
                        return ExitStatus.CONTINUABLE;
                        }
});
```

In the foregoing example, the interface returns ExitStatus.CONTINUABLE to keep the task continued. The interface rather returns ExitStatus.FINISHED if intended to get the task finished.

- RepeatContext

RepeatContext is the method factor of RepeatCallback. While the greater part of callbacks ignores contexts, callbacks function as attribute bags for reposition of data for transient use. When the method Iterate returns the result, the context ceases to exist. With the RepeatContext containing parent contexts when nested repetition is required, the parent contexts are useful to reposit the data that needs to be shared between the iterative calls.

- RepeatStatus

ExitStatus is used to define the processing is successful or not, or transmit the textual information of the status of batches or iterations. ExitStatus describes how the exit codes and free-form codes can be established.

Title of Property	**Description**
CONTINUABLE	Jobs continuable
FINISHED	Iteration finished

RepeatStatus can be accompanied by logical AND implementation, using the method AND. This signifies that the status FINISHED is ended up the result of FINISHED as well.

Completion Policies

CompletionPolicy defines completion of loops in the method Iterate, within RepeatTemplate and serves as a factory of RepeatContext. RepeatTemplate is responsible for transmission to RepeatCallback using the policy generating RepeatCOntext. Upon completion of callback, CompletionPolicy is asked whether to renew the status of doInIteration of RepeatTemplate (in other words, to reposit the data in RepeatContext), followed by request for policies upon completion of iteration.

Spring provides the simple CompletionPolicy implementor used for general purposes, such as SimpleCompletionPolicy that permits fixed-time execution. (You can use ExistStatus.FINISHED for early finish)

Exception Handling

When exception is thrown out of RepeatCallback, RepeatTemplate asks ExceptionHandler whether or not it should throw exception over again.

```
public interface ExceptionHandler {
        void handleException(RepeatContext context, Throwable throwable)
                throws RuntimeException;
}
```

In general, the exception is counted for the given type and failure is thrown when the limit is reached. In this connection, Spring provides not only SimpleLimitExceptionHandler, but also RethrowOnThresholdExceptionHandler for flexibility. Meanwhile, SimpleLimitExceptionHandler has exception types that compare the property Limit and the current exception thrown, inclusive of the sub-classes for the given type. The exceptions for the given type is ignored until reaching the limit, after which they are thrown. One of the most important properties that SimpleLimitExceptionHandler may choose is “useParent boolean”, with default value being false. The limit is described in RepeatContext. When configured 'true’ the limit is retained within the nested iteration.

Listeners

You may from time to time need to receive additional callbacks while a multitude of iterations are underway. To do so, Spring offers RepeatListener along with RepeatTemplate by which the user can register RepeatListener, transferred along with the RepeatContext and RepeatStatus while callback iterations are underway.

```
public interface RepeatListener {
        void before(RepeatContext context);
        void after(RepeatContext context, RepeatStatus result);
        void open(RepeatContext context);
        void onError(RepeatContext context, Throwable e);
        void close(RepeatContext context);
}
```

Both open callback and close callback are applied to the individual RepeatCallback calls, to be called before and after onError. While a multitude listeners are enlisted, the sequences of call for Open and Before are same with each other, with after(), onError() and close() called in reverse order.

Parallel Processing

Implementation of RepeatOperations may not restrict sequential execution of callback. It is very much important to secure the concurrent execution of multiple callbacks, where Spring provides TaskExecutorRepeatTemplate for TaskExecutor strategy in using RepeatCallback. In general, (just like the general RepeatTemplate) SynchronousTaskExecutor is used for total repetitions within the same thread.

Declarative Iteration

You might from time to time have certain business processing that is needed to e iterated, such as pipeline optimization, in which case the messages are preferably to be treated in the form of batch. In this connection, Spring provides AOP Interceptor that covers the method called in RepeatOperations. By executing the intercepted methods, RepeatOperationsInterceptor causes them iterated under CompletionPolicy of RepeatTemplate.
Refer to the following example for how processMessage methods are iteratively called using AOP Namespace:

```
 <aop:config>
     <aop:pointcut id="transactional"
          expression="execution(* com..*Service.processMessage(..))" />
     <aop:advisor pointcut-ref="transactional"
          advice-ref="retryAdvice" order="-1"/>
</aop:config>

<bean id="retryAdvice" class="org.spr...RepeatOperationsInterceptor"/>
```

In the foregoing example the default RetryTemplate is used to change either listener or policy. If th intercepted method is void return type, the interceptor with no exception returns ExistStatus.CONTINUABLE. (Unlimited repetition might take place if the end point is not defined in CompletionPolicy), or else ExitStatus.CONTINUABLE is iteratively returned until null is returned. In this connection, the business logic in the concerned method returns null or rethrows exception by way of ExceptionHandler provided by RepeatTemplate to finish.

References

Skip : http://static.springsource.org/spring-batch/reference/html/configureStep.html#configuringSkip
Retry : http://static.springsource.org/spring-batch/reference/html/configureStep.html#retryLogic
http://static.springsource.org/spring-batch/reference/html/retry.html
Repeat : http://static.springsource.org/spring-batch/reference/html/repeat.html