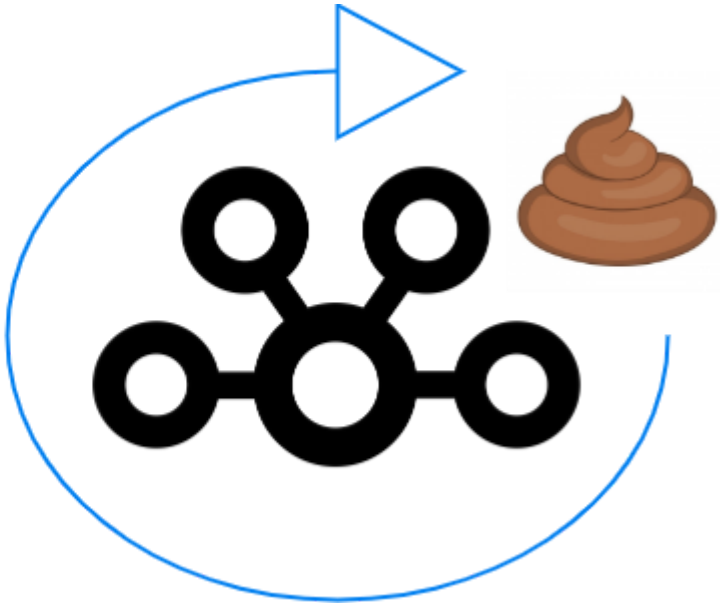
Exceptions and Retry Policy in Kafka

Victor Alekseev Follow
Feb 10 · 26 min read

> *"Whatever can go wrong, will go wrong all the time. If you think things are going well, then you're missing something."*



Once upon a time, when programs were small, and computer monitors delighted cats, we mostly dealt with monolithic applications, representing from the user's point of view one significant failure point. The application either worked, or it didn't. At most, the user could reload the entire page.Often this helped; in other cases, it was self-evident that nothing would help anytime soon. The replay policy was straightforward and was implemented on the user side. Errors were a problem caused by something breaking unexpectedly and needing to be repaired as soon as possible.

Today we deal mostly with distributed applications, where errors are not annoying accidents but typical and even expected events. Practically everything can go wrong: the

request and even the response can be lost, the synchronously called service can suddenly restart or move, the data needed to fulfill the request may not yet be available, and so on. This leads to two fundamental conclusions: the calling party must be able to repeat calls if necessary, and the receiving party must be able to provide idempotent processing. Let's talk today in more detail about the first aspect in the context of Kafka and Spring Cloud infrastructure and transactional processing messages.

General exception management

This chapter will look at the exceptions in terms of message retry, the different strategies for retry execution, and the possible architecture for configuring this aspect of the application.

Classification of exceptions

First, we need to divide possible errors into several groups, each of which requires its specific approach to retry. The meaning of the terms stateful and stateless applied to the exception handling process should be clarified beforehand.

- **"Stateless"** in this case means that it is up to the application to implement a process of sequential processing attempts. We read the message from the topic once and try to process it several times. The state of the datastore (the group's offset inside the Kafka partition) is not changed in this case.
- In contrast, during **"Stateful"** processing, the implementation of iterations is left to the data store. Over and over, we "discard" a message we have already received, putting it back and forcing the consumer to reread it. Such discarding can be implemented by manipulating the group's offset in the Kafka partition.

Possible types of exceptions:

- **Not retryable exceptions (NRE)** — are errors that will most likely always be handled with an error. In this case, trying to re-execute the request will not allow us to fix the situation. In the vast majority of cases, we can only log the fact that an exception has been raised, indicating a bug in our application, and send the message into Dead Letter Queue (DLQ) for future analysis. Typical examples of such

exceptions are **NullPointerException, ClassCastException, NoSuchMethodException, DeserializationException, MessageConversionException, MethodArgumentResolutionException**, and so on. This class of exceptions also includes errors in non-idempotent business operations that should not be automatically repeated and cannot be automatically reversed.

- **Stateless retryable exceptions (SLE)** — indicate problems that can be fixed by a re-call and will most likely be fixed soon. Most business level exceptions are of this type. In this case, the critical factor is the expected fixing speed, allowing a sequence of retries to be made within the timeout allocated to process a single message. Until the entire list of message processing attempts is complete, the next request will not be pulled from the partition by the service. After a series of several not successful attempts, such messages are usually sent to DLQ too.

- **Stateful blocking retryable exceptions (SBE)** — for which we don't know how many attempts it will take before the message is appropriately handled for this type of exception. Consequently, we cannot iterate in memory and must periodically reread the message repeatedly from the partition for the next attempt. Simultaneously, the identified problem is quite cardinal for the application and is likely to block the execution of all subsequent requests. Therefore, we have to suspend the flow of incoming messages until we have safely processed this one. Typical examples of such exceptions are: **SocketTimeoutException, ConnectException, TransactionTimedOutException, UnknownHostException**. In this case, we can never send failed messages to DLQ since the application is undoubtedly inoperable and must stop processing messages in general.

- **Stateful not blocking retryable exception (SNBE)** — are similar to the previous one, with the difference that the problem detected affects only the processing of this message. Subsequent messages can most likely be handled by the application safely. Usually, this type of situation is related to the fact that the data required to execute the request is not yet available. We have to wait a little while for it to be delivered to the appropriate repository. Whether or not to send messages whose processing causes exceptions of this type to DLQ depends on business requirements and, above all, on the SLA and the specified incident handling policy.

The enum **ExceptionType** can express this classification with the next members: **NRE, SLE, SBE, SNBE**.

As a rule, it is not the exception itself used to classify an exception, but the presence of some specific exception (or the successor of some specific exception) in the chain of its causes. Spring provides us with an extremely convenient template for our classifier in the form of the **BinaryExceptionClassifier** class. It is recommended for convenience to add the following methods to the successor: **addException(class, defaultValue)** and **addException(className, defaultValue)**. The last one is used when the exception list is externally configured or the type of the necessary exception is available only at runtime.

In some cases, the same class's exceptions may differ only in some other aspects, such as message text, for example. As an illustration, we can consider working with PostgreSQL when we meet with only one type of exception — **PSQLException**, containing different text messages. SBE exceptions include only those containing the following message fragments: "canceling statement due to statement timeout", "canceling statement due to user request", or "The connection attempt failed". Therefore, it is also recommended to implement a programmatic exception classification strategy based on an interface **ExceptionTypeResolver** too. This interface can contain only a single method **ExceptionType classify(consumerRecord, consumer, throwable)**.

We also need to be able to mask some exceptions with others. For example, in some cases, **NullPointerException** is used by legacy code when an object is not found in some repository. Thus, in fact, we deal not with NRE, but with SNBE. Fortunately, **BinaryExceptionClassifier** examines the chain of causes and stops at the first exception it knows of. Therefore, if we wrap **NullPointerException** in **StatefulBlockingRetryException**, our exception will be classified correctly. Accordingly, it is necessary to have one type of wrapper for each of the above exception types.

Retry strategies

To implement a sequence of execution attempts, Spring offers the standard **spring-retry** module with the following main utils and annotations:

- **@EnableRetry** on the configuration

- **@Retryable** on the methods, which has to be called several times automatically
- **RetryTemplate** for repeating execution some codes fragment
- **BackOffPolicy** for definition parameters of iterations — count of attempts, pauses, and so on

Kafka Spring uses all these tools, but there is one problem: they are all too smart and imply an iteration externally to the code being executed. For NRE and SLE processing, this is enough, but we need to calculate for the last two types of exceptions how long we should delay the next attempt to handle the message. We also need a convenient way to configure this strategy globally or at a specific topic level. It's best if this would be possible based on a single string value of some configuration parameter.

So we need an interface **RetryPolicy** with a roughly similar set of methods:

- **static RetryPolicy parsePolicy(String strArgs)**
- **int getNext(int current)**
- **int getMaxInterval()**
- **void parse(String strArgs)**

As practice shows, the following strategies are enough to work:

- An unlimited exponential retry policy with parameters: **initialDelay, maxDelay, multiplier**.
- A limited exponential retry policy with parameters: **initialDelay, maxDelay, multiplier, maxDelayCount**.
- An arbitrary strategy with direct enumeration of delays: **5000x3, 10000x5, 30000x10, 60000x20**.

Configuration

As far as we can see from two different chapters, both aspects of handling messages from various topics can be the subject of the configuration: classification of the possible exceptions and definition retry policy for each of them.

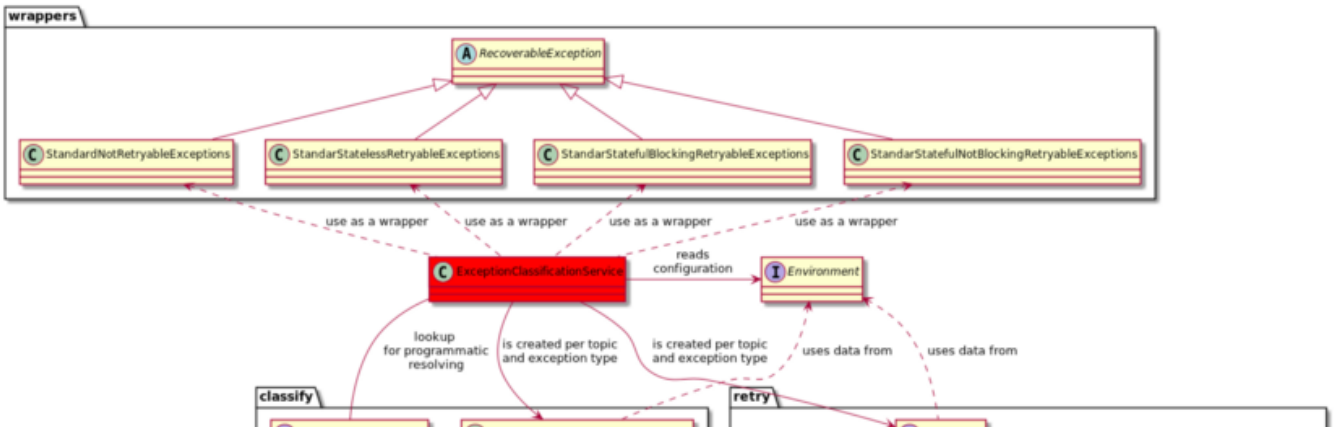
So, the next fragment of the configuration can be applied on the two levels: per topic level and on a global level as a default configuration:

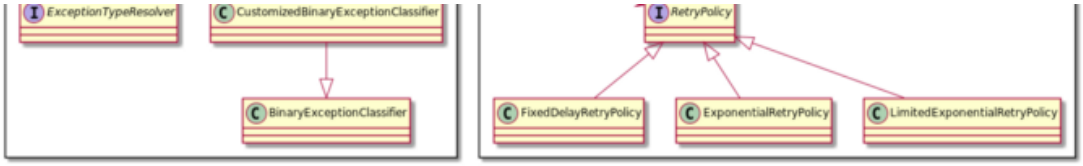
```yaml
kafka:
  failedMessagesRetryConfiguration:
    notRetryableExceptions:
      # in this case retryPolicy configuration property is not applicable at all
      list:
        - java.lang.NullPointerException
    statelessRetryableExceptions:
      # in this case retryPolicy configuration is already defined as a standard consumer BackOff
      # only in global level. Let's keep it as it is for simplicity of description.
      list:
        - org.someCompany.someSystem.SomeBusinessException
    statefulBlockingRetryableExceptions:
      retryPolicy: LimitedExponentialRetryPolicy(1,60,2,120)
      list:
        - java.net.SocketTimeoutException
    statefulNotBlockingRetryableExceptions:
      retryPolicy: FixedDelayRetryPolicy(5000x3, 15000x5, 60000x10, 120000x15)
      list:
        - java.net.SocketTimeoutException
    allOtherExceptions: SLE
```

kafka_retry_policy_config.yaml hosted with ♡ by GitHub view raw

We can hide all aspects of execution related to the configuration behind an **ExceptionClassificationService** service. It provides an only operation — **Pair<ExceptionType,RetryPolicy> classify(records, consumer, exception)**.

This service can be implemented with the next principal architecture:





Infrastructure provided by Spring

When using the simple standard integration between Kafka and Spring, everything is transparent and available for customization.

When we implement non-transactional processing, we can provide customized **ConcurrentKafkaListenerContainerFactory** as a bean with only two additional options.

- First, lets to add **SeekToCurrentErrorHandler**, which implements a retry policy for each message, which was processed with an error.
- Second, we need to customize it by **DeadLetterPublishingRecoverer**, which after several not successful attempts, sends the wrapped original message into the DLQ.

```
1       @Bean
2       ConcurrentKafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory(
3               ConcurrentKafkaListenerContainerFactoryConfigurer configurer,
4               ObjectProvider<ConsumerFactory<Object, Object>> kafkaConsumerFactory
5               KafkaTemplate kafkaTemplate) {
6           ConcurrentKafkaListenerContainerFactory<Object, Object> factory =
7                                   new ConcurrentKafkaListenerContainerFactory<>();
8           configurer.configure(factory, kafkaConsumerFactory);
9           factory.setErrorHandler(
10                  new SeekToCurrentErrorHandler(
11                          new DeadLetterPublishingRecoverer(kafkaTemplate), 3
12                  )
13          );
14          return factory;
15      }
```

kafka-retry-policy-spring-std.java hosted with ♡ by GitHub view raw

Following the code above, Kafka consumer retries processing of message three times in case of SLE error and sends it into **<originalTopicName>.DLT** topic.

When transactions are being used, no error handlers are configured, by default, so that the exception will roll back the transaction. Error handling for transactional containers must be handled by implementing the **AfterRollbackProcessor** interface, instead of **SeekToCurrentErrorHandler**. Its main method is **void process(consumerRecords, consumer, exception, recoverable)**, about which we can read in the source code the following:

> "process the remaining records.Recoverable will be true if the container is processing individual records; this allows the processor to recover (skip) the failed record rather than re-seeking it. This is not possible with a batch listener since only the listener itself knows which record in the batch keeps failing. IMPORTANT: If invoked in a transaction when the listener was invoked with a single record, the transaction id will be based on the container group.id and the topic/partition of the failed record to avoid issues with zombie fencing. So, generally, only its offset should be sent to the transaction. For other behavior, the process method should manage its own transaction."

The default implementation of this interface is **DefaultAfterRollbackProcessor** and the rest of this article will be about how we can customize its behavior in different situations.

Unfortunately, if we use Spring Kafka Cloud integration, our situation is much sadder. Of course, we can provide our implementation of the **ConcurrentKafkaListenerContainerFactory**, but it will not be taken into account. The keystone for Spring Kafka Cloud integration class **KafkaMessageChannelBinder** creates its own instances of this class by method **createKafkaConsumerFactory()** and configures all containers by method **createConsumerEndpoint()** (it contains about 150 lines of a hard structured code). Both methods are closed for legal customization.

The only, what we can do is customize already created instances of **ConcurrentMessageListenerContainer**. It can be achieved by publishing the bean with type **ListenerContainerCustomizer**, which constructor accepts only one parameter **BinderFactory**. In the method **void configure(container, destinationName, group)** we get access to the "ready to use" instance of **ConcurrentMessageListenerContainer**,

which property **afterRollbackProcessor** is initialised with the instance of **DefaultAfterRollbackProcessor**. This instance of a rollback processor, in turn, uses two other components:

- **BiConsumer<ConsumerRecord, Exception>** recover lambda for sending the finally a no-go message into DLQ (if it is configured, of course)
- **BackOff** for processing messages resulting in SLE exceptions. The situation is made even more complicated by the fact that both dependencies are not available through getters.

So this instance of **DefaultAfterRollbackProcessor** is closed to customization through inheritance too — we can't provide the constructor with the necessary dependencies initialized inside **KafkaMessageChannelBinder**. The situation is a blind alley, so you have to perform the wonders of dirty hacking.

- By means of Apache Jakarta **FieldUtils.readField** we can read a private property **failureTracker** with type **FailedRecordTracker** from an available instance of **DefaultAfterRollbackProcessor**.
- The recover we need is available as **failedRecordTracker.getRecoverer()**.
- To get **BackOff** strategy, we need to repeat this dirty hack and to read a private property **backOff** of **failureTracker**.

After that we are free to inherit from the standard **DefaultAfterRollbackProcessor** class and substitute its standard implementation by the method **ConcurrentMessageListenerContainer.setAfterRollbackProcessor()**. The door to further improvements becomes open!

```
1       @Bean
2       public ListenerContainerCustomizer listenerContainerCustomizer(BinderFactory binders) {
3           return new ListenerContainerCustomizer() {
4               @Override
5               public void configure(Object container, String destinationName, String group) {
6                   ConcurrentMessageListenerContainer cmlc = (ConcurrentMessageListenerContainer) c
7                   cmlc.setAfterRollbackProcessor(new CustomizedAfterRollbackProcessor(cmlc));
8               }
9           };
```

```
10      }
11
12      private static class CustomizedAfterRollbackProcessor extends DefaultAfterRollbackProcessor
13          public CustomizedAfterRollbackProcessor(ConcurrentMessageListenerContainer container) {
14              super(getInstalledRecover(container), getInstalledBackOff(container));
15          }
16      }
```

kafka-retry-policy-spring-cloud.java hosted with ♡ by GitHub view raw

The last tricky detail — class **FailedRecordTracker** is package-private, so functions **getInstalledRecover** and **getInstalledBackOff** has to be implemented inside the package **org.springframework.kafka.listener**.

Basic setup and customization

As a minimum, before we have to configure DLQ processing properly. It is not complicated, and the next sample of YAML demonstrates it:

```yaml
spring.cloud.stream.kafka.bindings:
  some-topic:
    consumer:
      enableDlq: true
      dlqName: some-topic-error
      dlqProducerProperties: # not required, but can be useful
```

kafka-retry-policy-dlq-config.yaml hosted with ♡ by GitHub view raw

The count of partitions in the DLQ

However, one detail is not self-evident — by default, the dead-letter record is sent to the same partition as the original record. It is an entirely reasonable default restriction: if we want to continue somehow processing pending into DLQ messages, keeping their original partitioning helps us keep their mutual processing order.

Therefore, the dead-letter topic must have at least as many partitions as the original topic. If it was by mistake not performed, we could see in the log the next not quite clear at first glance messages:

- _ exception thrown when sending a message with key='{...}' and payload='{...}' to topic some-topic-error and partition <X>_
- *TimeoutException: Topic some-topic-error not present in metadata after 60000 ms.*

As a result, the consuming thread is blocked during 60000 ms. After this, the execution will be continued without the throwing of any exception. What is incredibly annoying is that it is challenging to associate these exceptions with the original ones that attempted to forward the messages to DLQ. It feels like somewhere in Kafka's depths, the timeout exception is just swallowed, and execution continues without trying to continue sending the message.

By the way, Kafka is generally very fond of swallowing exceptions that she thinks can't be handled by the user or that she hopes to somehow compensate for. A good example is exceptions inside **ConsumerInterceptor / ProducerInterceptor**, which are only logged but cannot be caught by an application code.

Thus, the original message is solely lost and does not get into the DLQ without any apparent symptoms. Unfortunately, I have not found any way to set a timeout to get the topic's metadata. The best way to correct such an error is to use the correct configuration of topics or manually manage this process by providing **DlqPartitionFunction** bean.

Also, it is possible to implement a wrapper around recover lambda — **CustomizedRecover**.

- This advanced Recover, by the timer, will interrupt the sending process with **sendingIntoDlqThread.interrupt()**.
- The sending process after return from the wrapped method can validate **Thread.interrupted()** and itself throws a timeout exception.
- After receiving this exception, **DefaultAfterRollbackProcessor** will discard the current message's reception, which will be accepted again by the next **poll()** call.

The method is quite a dirty hack because it is impossible to ensure no race condition, and there is always the danger of interrupting something wrong. But it's easy to

implement, works, and prevents from losing the messages in case of a wrong accidentally incorrect configuration (see epigraph).

In general, working with DLQ requires the same careful configuration as working with the main incoming topic. For example, if we want to handle large messages (more than 1048576 bytes by default), we have to extend this limit for both topics simultaneously through **max.request.size** configuration properties. Otherwise, we will be faced with a "poisonous" message, and the service will stop.

Uncommitted by default offset

The next small problem is a strange "by default" configuration of **DefaultAfterRollbackProcessor**.

After unsuccessful processing of the message and sending it into DLQ, the processor does not commit Kafka's next offset.

- Usually, it is not a problem because after reading and successfully processing the following message, the next offset will be committed, and the problematic message will not be received twice.
- But if we stop and start the service immediately after, or if re-balancing takes place, this message will be received and processed again.

Retrieving and processing a single message in this way is certainly not a problem. But if there are hundreds of thousands of such messages, and after another random re-balancing, we start processing them all over again, it may already become a problem.

This issue can be fixed by calling the next methods of the **DefaultAfterRollbackProcessor**: **setCommitRecovered(true)** and **setKafkaOperations(kafkaTemplate)**. The only problem is where to get the instance of **KafkaTemplate**.

Usually, we create it from the **ProducerFactory** instance, but it is not available with Spring Kafka Cloud integration (surprise!!!). Fortunately, we can get it without reflection as **KafkaMessageChannelBinder) binders.getBinder(null, MessageChannel.class.getTransactionalProducerFactory()**.

Handling NRE, SLE and not recognized exceptions

By default, **DefaultAfterRollbackProcessor** can properly recognize NRE exceptions by embedded instance of **BinaryExceptionClassifier**. But since we want to get complete control over this process, we can clear this list. From that moment, all exceptions, from the processor's point of view, will be stateless retryable. So, we only have to filter NRE exceptions and send them into DLQ manually.

Dealing with SLE is even easier. All exceptions, recognized as SLE, may be passed to the default implementations. After that, three standard attempts of execution will be performed, and after the third failure, the failed message will be automatically sent into DLQ.

So our implementation of the **CustomizedAfterRollbackProcessor** class looks something like this:

```
1   public class CustomizedAfterRollbackProcessor extends DefaultAfterRollbackProcessor {
2       public CustomizedAfterRollbackProcessor(ConcurrentMessageListenerContainer container) {
3           super(customizeRecover(getInstalledRecover(container)), getInstalledBackOff(container));
4           ((ExtendedBinaryExceptionClassifier) getClassifier()).getClassified().clear();
5       }
6
7       @Override
8       public void process(List records, Consumer consumer, Exception exception, boolean recoverabl
9           Pair<ExceptionType, RetryPolicy> retry =
10              exceptionClassificationService.classify(records, consumer, exception);
11
12          switch (retry.getKey()) {
13              case NRE:   sendIntoDlqDirectly(records, consumer, exception);
14                          break;
15              case SLE:   super.process(records, consumer, exception, recoverable);
16                          break;
17              case SBE:   processWithStatefulBlockingRetries(records, consumer, exception);
18                          break;
19              case SNBE:  processWithStatefulNotBlockingRetries(records, consumer, exception);
20                          break;
21          }
22      }
23
```

```
24      // Remember this function, we will need it again
25      private void sendIntoDlqDirectly(List records, Consumer consumer, Exception exception) {
26          ConsumerRecord consumerRecord = (ConsumerRecord) records.get(0);
27          // usage preconfigured Spring Cloud DLQ implementation
28          getCuztomizedRecover().accept(consumerRecord, exception);
29          // commit next offset, to avoid getting error again after re-balancing
30          kafkaTemplate.sendOffsetsToTransaction(
31                  Collections.singletonMap(
32                          consumerRecord.getPartition(),
33                          new OffsetAndMetadata(skipped.offset() + 1)
34                  )
35          );
36      }
37  }
```

In this section (and the next ones too), the above the source code is slightly simplified, assuming that the “records” list contains only one record or records belonging to only one partition of some topic. Such a rare situation may happen if the listener serves only one topic with one partition or if the number of dedicated threads is equivalent to the number of partitions. Experience shows that in this case the allocation is made according to the principle of “dedicated partition per a thread” and there are no problems.

But normally the situation is more complicated and this list will contain records belonging to different partitions and topics. We will have to significantly complicate the processing algorithm because we need to commit a new offset for only one partition. On the contrary, for the other partitions, we have to shift back to the current offset to process the record again.

Handling Stateful Blocking Retryable Exception

Handling stateful blocking retryable exception is slightly more complicated. We have to handle the message over and over again. If it fails, we move the offset back and wait for a while before trying again.

We repeat this process indefinitely until the message is finally handled successfully. All other messages in the partition are, of course, blocked waiting to be received. The message to be processed is, in fact, a poisonous message or somehow close to it.

The main problem is storing the process context:

- Normally, we want to delay the message exponentially.
- Accordingly, we need to keep the iteration number somewhere so that we can use it to calculate the next delay.
- Since we can't change the message already stored in Kafka's topic, we have to store the message's context in the service's memory.

We can turn to the following tradeoff method, which looks quite correct:

- The retry context of the message consists only of two fields: **offset** and **currentIterationIdx**
- To store context, you can use a static structure **ContextHolder**, implemented as **Map<String, Context>**, where something like **"<message.topic>_<message.partition>"** is used as a key. The count of such a combination for each service is small and limited, so we don't need to use a complex cache implementation — **ConcurrentHashMap** is enough.
- When we query the context from the cache, we can get either **null** or an obsolete context. For the obsolete context **message.offset != context.offset**. In both cases, we assume this is the first iteration for the current message and recreate the context with corresponding **offset** and **currentIterationIdx = 0**.
- The context may become obsolete in the following two cases. It is the first attempt to process this message, and the context is left over from the previous, already processed SBE message. Or re-partitioning occurred, and the message processing moved to another cluster node. In both cases, "start from scratch" is an acceptable strategy.

As a result, we know the next iteration for the current message and can calculate the delay for the next iteration by implementing **RetryPolicy**, configured globally or on the

specific topic level.

Now, all we have to do is to decide what to do with this knowledge.

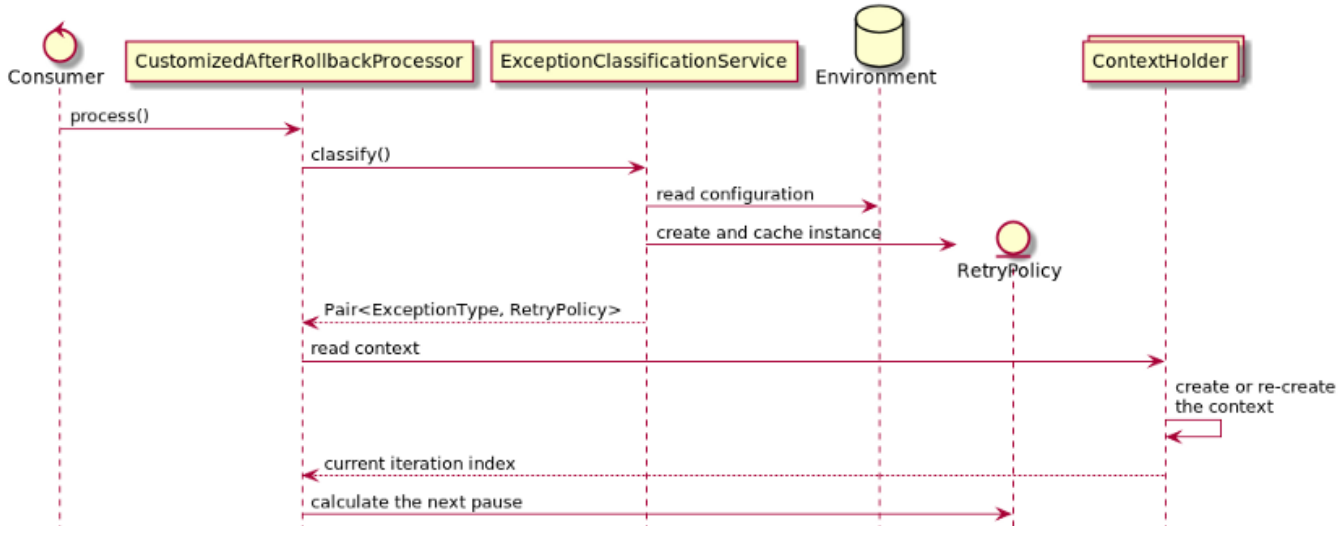
If it is not the last iteration, we have to apply the delay

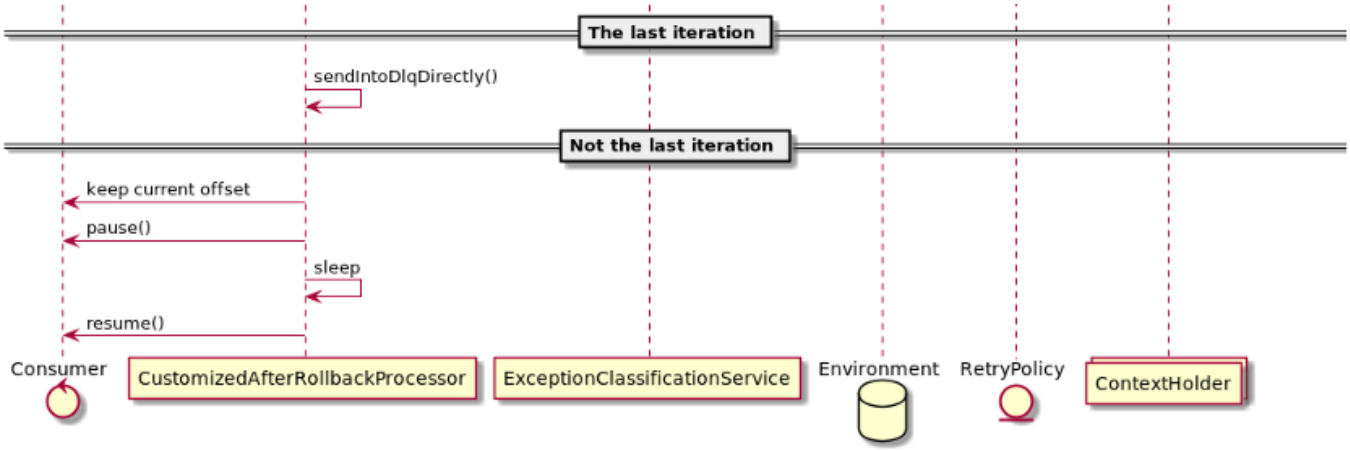
- keep the current offset by **consumer.seek(record.partition(), record.offset())**.
- put the consumer on pause by **consumer.pause(Collections.singletonList(record.partition))**
- wait for the previously calculated delay. As the consumer is paused, we don't risk re-partitioning, caused by **max.poll.interval.ms** timeout. Also for applying the pause can be used methods **container.pause() / container.resume()**. It is extremely doubtful that we can process messages from one partition and unable to do so from another one.
- restore the state by **consumer.resume(Collections.singletonList(partition))**

If it is the last iteration, we have

- directly send the failed message into DLQ as we had also previously performed for NRE exceptions
- it is worth noting that stopping an iteration and redirecting the message to DLQ for this type of exception is an infrequent event.

The next diagram can illustrate the described approach:





Handling Stateful Not Blocking Retryable Exception

In this case, we are not dealing with problems global for the application, but with the inability to process only some specific message. But, we expect that the situation will be fixed by the developers in the future, and this message may be successfully processed.

Typical reasons for such problems:

- The data required for processing is not yet available.
- The consuming process has not reached the corresponding step when it can consume this message
- Processing the message requires some quite specific functionality, which is not available at this moment.
- In the case of a multifunctional topic (quite not recommended), processing of only this type of messages is impossible at the moment

The solution's idea is obvious — temporarily defer the message to some storage and process it later after some time. The main question is what type of storage to use.

DB-based storage

The first thing that comes to mind is a relational database-based repository into which the deferred message is written.

Along with the message, the necessary metadata is published: the reason, the original topic /key, the value of the delay. Based on the **“select for update skip locked”** request, the scheduler process reads the pending messages as they are ready and sends them back into services topics to continue standard processing. Since the procedure for sending Kafka messages is fast enough, we can expect high sufficient for an enterprise architecture throughput — about 1000 messages per second.

To ensure that repeating messages do not compete with the original ones and they are picked up by the service as quickly as possible, it is recommended to organize two input queues — for standard and for repeated messages.

Advantages of DB-based storage

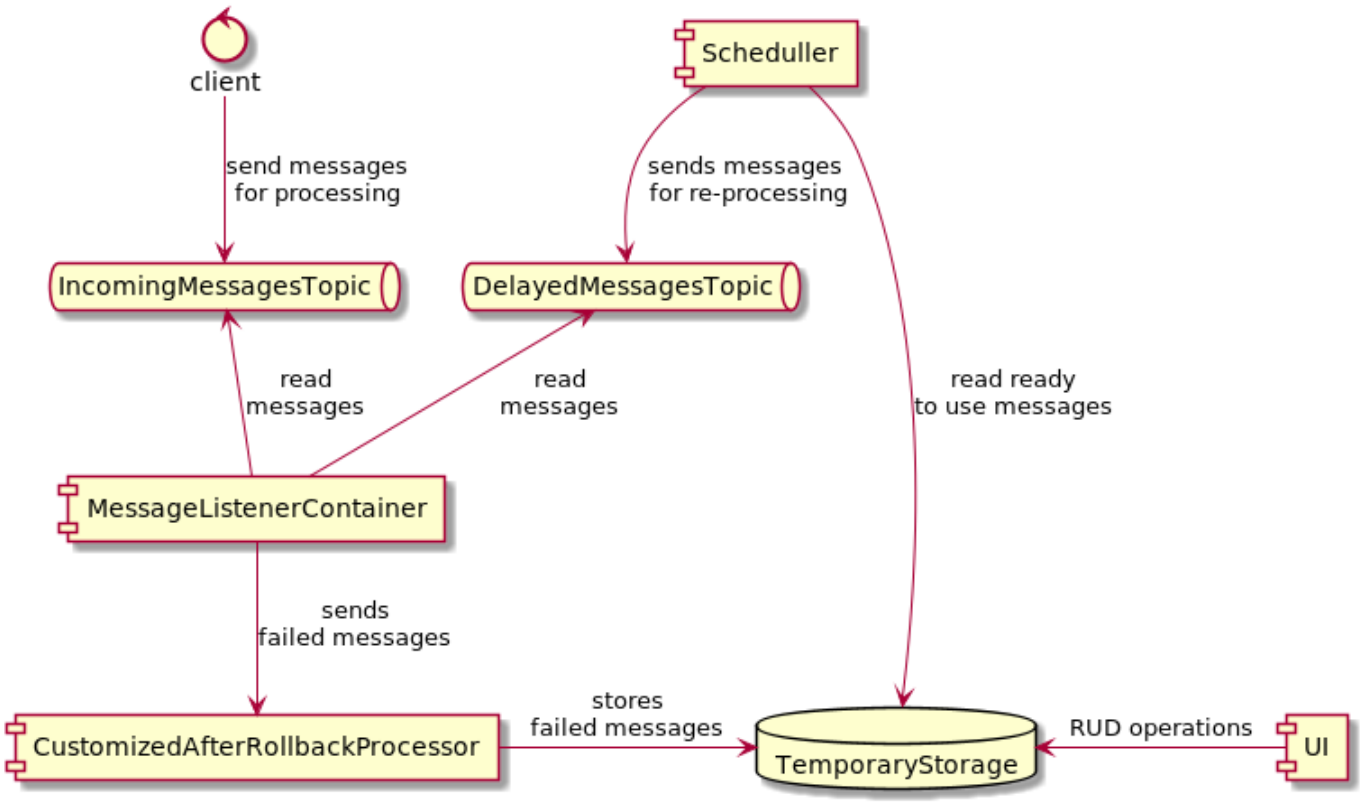
- This repository is highly observable and auditable. At any time, we can find a pending message, find out when it is going to be processed, manually start the re-processing immediately, and so on.
- Implementing a user interface is very trivial.
- This storage is also an excellent candidate to use as a gateway for guaranteed transactional messaging.

Disadvantages of DB-based storage

- If the business process involves simultaneously working with millions of pending messages, the database performance is not enough.
- Even if there are no pending messages in the repository, we still have a process that continually polls and burdened the base.
- Self-implementation of such a repository is not always trivial. For example, several messages will likely be read and passed to the thread pool for sending at a time. If the order of messages between them is critical, that partitioning should be supported by such a scheduler’s logic.

I can suggest using something like **Yandex db-queue** as a basis for such repository implementation. An impressive solution is **PgQ** — an extremely high-performance queue implemented by Skype.

The following diagram can briefly explain the above procedure:



Kafka based storage

If we want to build a really event-driven system, we can use the Leaky bucket pattern. This approach's basic idea is straightforward: if all messages in the same topic have the same value of "delay" property, they can all be processed sequentially.

So the basic architecture of the solution will look something like this:

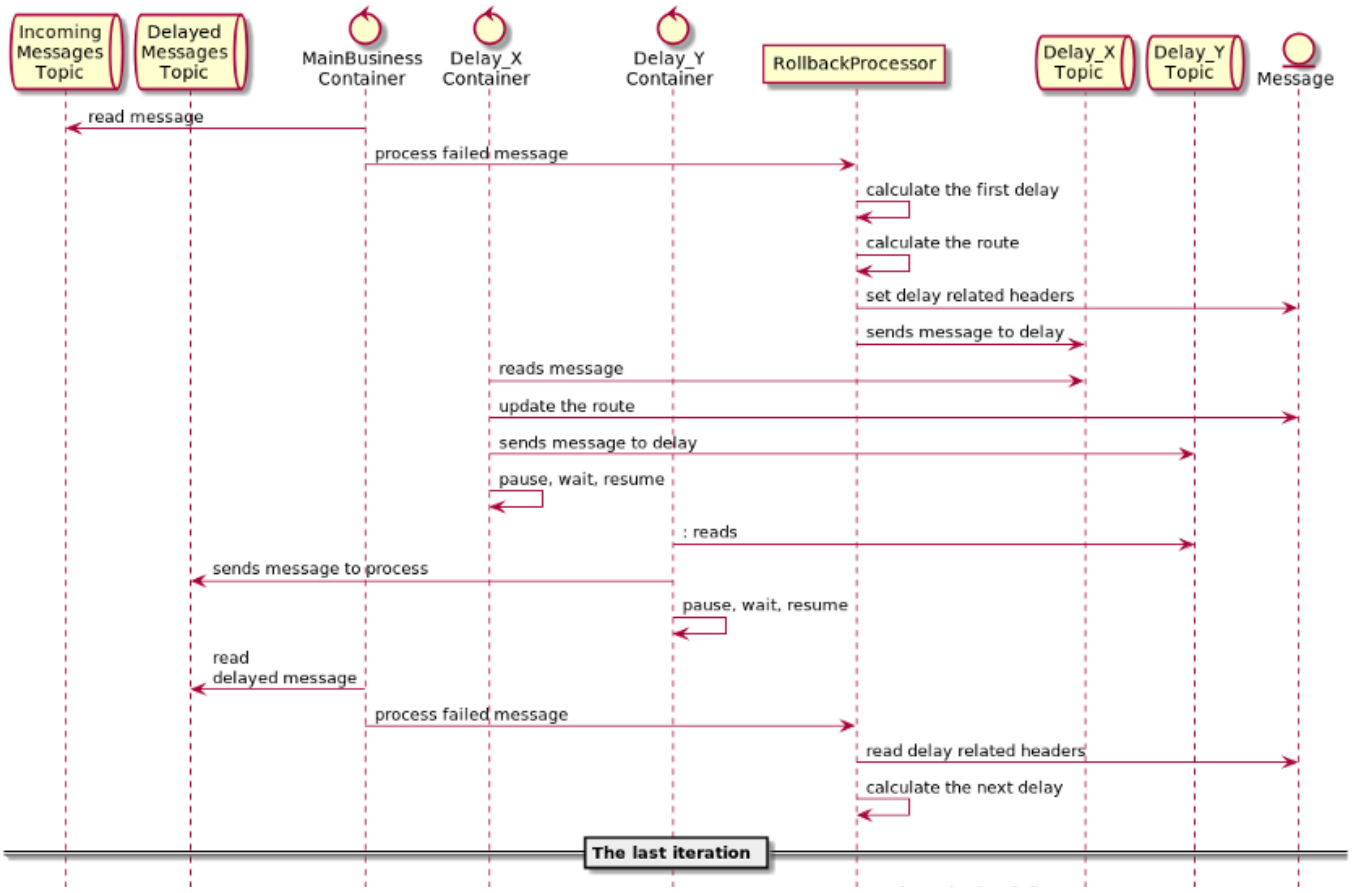
- There is a set of standard topics, each responsible for implementing a delay in message processing of some specific length. For example, one minute, ten minutes, half an hour, an hour, and so on.
- For each topic, we have a standard personal consumer that performs the following relatively simple operations. It reads the next message and looks if the expected delay has expired, relative to when the message was added to the topic.
- If no, it gets a pause and waits to satisfy the required delay (we have already implemented this operation in the context of SBE handling). After that, it sends the

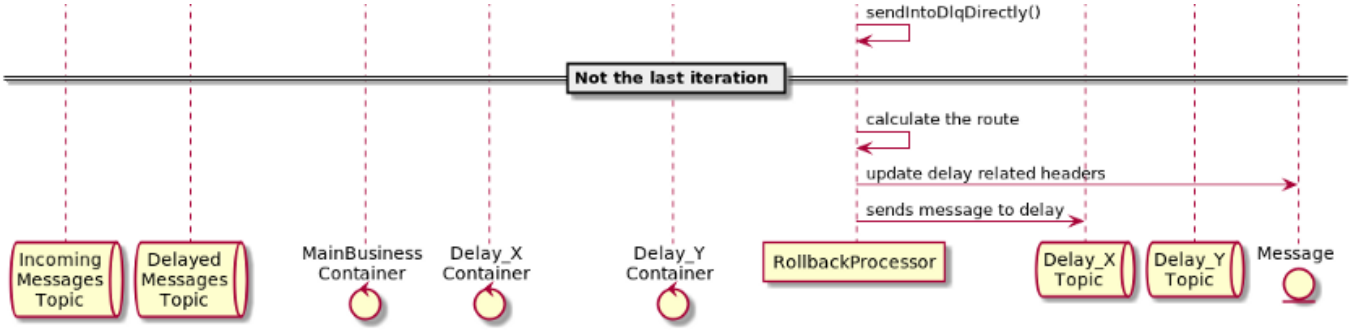
message in the next topic, defined by the standard header, and resumes consumption.

The whole message processing delay for an arbitrary period is realized by decomposing the value into a sequence of fixed delays. Then, the topics' route is added as some message header, which is updated with each subsequent transfer. For example, if we want to arrange execution delay for 17 minutes, the route will look like "delay_10, delay_5, delay_1, delay_1, DelayedMessagesTopic". Finally, the message is sent to the first queue that implements the fixed delay; in our case, it is "delay_10". In the end, the dedicated consumer of "delay_1" sends the message into "DelayedMessagesTopic" for re-processing by an application.

As the application can enrich the message with the set of delay related headers, we don't have to store the iteration process context in the memory and can pass it along with the message. As a minimum, we need for the next headers: "X_DELAY_ROUTE" (list of the delaying topics), "X_DELAY_COUNTER" (the number of the current iteration), "X_DELAY_REASON" — for the logging, and so on.

The next diagram can illustrate this consequence of operations:





The important note: since we need to preserve the mutual order of messages, we must use partitions with the same number as the partition in the original topic when passing from topic to topic.

If messages contain a lot of data, we can significantly reduce traffic passing the postponement message. Such reduction is possible if we do not include the whole original message in the " delayed" message, but only a pointer to it in the form of <topic name / partition number / offset>. In this case, the scheduler at the last stage, instead of forwarding the message itself to the incoming service topic, must: extract a reference to the original message from it, read the original message, clone it and send the resulting copy to the service for processing. In this case, the original topic's retention setting should allow the message to survive to the end of the delaying process. This technique will be discussed in more detail a little later in the "What to do with DLQ" section.

This technology is slightly more complicated in terms of implementation but has several fundamental advantages: completely event-driven architecture and Unlimited horizontal scalability. The main problem is manageability: while a pending message travels between topics, we have no single point to get information about it or re-process it immediately.

Deployment: a local storage VS a dedicated service

In the case of using a relational database as storage, a dedicated service is not an option at all since using only one database instance will significantly reduce throughput.

- Besides, if one of the services starts generating too many "delayed" messages, it will slow down the similar functionality for other services.

- At best, the services will not be runtime independent; at worst, a negative feedback cycle can be achieved.

Although it is an attractive idea — it allows you to have a central point of control and monitor all re-processing messages.

In Kafka's case, we have a real choice between two possibilities. Each has its advantages and disadvantages.

A local storage:

- The number of topics in the system increases significantly because we have to create a different set of "delaying" topics for each new service.
- The services remain entirely independent from each other.
- Each service must have a significant amount of code and threads that implement the scheduler's logic.

A dedicated service

- The total number of topics in the system is much less.
- Security problem — this service becomes a high vulnerability point, allowing attack of all other services. This vulnerability can be fixed by having each service sign its messages sent to the scheduler service. In this case, the signature/encryption key can be symmetric, random, periodically regenerated by each service, and stored within its database.
- The scheduling logic remaining on each of the services can be extremely slim — forward the message to the scheduler's standard queue, adding standard headers to it.

The problem of interdependent messages

As long as the messages are independent of each other, everything works fine as is. But unfortunately, in many cases, we are faced with a situation where the successful processing of one message depends on previous messages' processing. A troubling question arises: if during the processing of some message, we get an SNBE exception and send this message to the dedicated topic for non-blocking re-execution, what we have to do with the subsequent related messages?

Sometimes the processing of the subsequent messages is merely impossible without processing the previous one. For example, when the business process has not yet reached the right point. In this case, the situation is not so bad — we can receive an exception, based on which we can send a follow-up message for delayed re-processing too. We can also neglect this dependence when a subsequent message is sent only in response to a notification of processing the previous one.

Unfortunately, there are situations where follow-up messages can technically be applied, but it is entirely incorrect from a business point of view. For example, in the CQRS pattern, "skipping" one message does not lead to technical problems. But as a result of such swallowing, the state of the data aggregate at the source and the destination sides are different.

Possible solution

- We can add to each message's metadata a particular field — "unique business process identifier". It may be the UID of the actual BPMN process or the ID of the data aggregate.
- A dedicated table for storing such identifiers and related details is created in the service's data storage. If we have sent a message along some route for delayed execution, the business process identifier with the calculated route is added into this table.
- After that, all other messages with this identifier are automatically sent along the specified route without an attempt to process. In this case, the principle is that we should not calculate a new route for these subsequent messages (since the time elapsed since the first message, the configuration may change) but use the original one.

- It must be taken into account that all these messages will be normally processed at some future time. Therefore, in addition to each UID route, you must also store the number of already delayed messages. If the deferred message was processed properly, this counter is decreased, or the record is removed from the storage.

This algorithm is not thread-safe, but while we keep the partition number when forwarding, all messages in the context of one business process will be processed strictly sequentially.

We encounter a similar problem when forwarding a message to a DLQ topic. It can happen for almost all types of exceptions, apart from maybe SBE. Moreover, there may be mixed cases. For example, we start to handle an SNBE exception, get an NRE at the next retry, and send a message to DLQ. Subsequent messages, respectively, must be sent to DLQ as well.

The task of identifying mutually dependent messages can only be effectively solved at the business level. Accordingly, we need mechanisms to notify the application that some message has been sent to DLQ. We also need an easy way to redirect the currently processed message to DLQ without processing it again. Due to this reason, it is recommended to introduce the following two components:

- An interface **CustomizedRecoverListener** with the following two or so methods: **void before(consumerRecord, initialEx, route)** and **void after(consumerRecord, initialEx, route, recoverEx)**. In the case of sending into DLQ, parameter **route** is null. In case of an error happens during recovering, parameter **recoverEx** is not null.
- An exception **PreviousMessageWasRecovered** with the next params: **message** and **route**. In the previous message that was sent into DLQ, parameter **route** is null.

What to do with DLQ?

DLQ is some analog of limbo, the messages from which must be selected by the administrator and analyzed manually. If the application does not provide a UI to perform this operation, then the messages are, in fact, just thrown away.

A minimum set of necessary functionality:

- Getting information about an issue: a timestamp, a service name, a class name, an exception related info (class, message, stack trace), an origin, a location (topic/partition/offset), a payload type and data (or some essential fragments from it), unique request and business process identifiers, and so on.
- Getting a fragment of the log of the corresponding service around the event.
- Getting a list of other messages that may be associated with the failed one.
- Getting a list of messages that were sent to DLQ only because this one is in it.
- Manual sending the message (and its followers) to re-processing. It can only be done after the problems that prevented proper message processing have been resolved on the processing services side. As a fundamental principle, we don't want to edit failed message before re-sending them since the original message stream could theoretically be processed again by the same service. In this case, we don't want the original message to end up in DLQ again.
- Getting statistics about the different types of messages sent to DLQ

If we don't want to write a dedicated application to do that, we can get almost everything we need from a combination of Elasticsearch and Kibana.

- A special "collector" reads messages from all DLQ queues, enriches them with metadata / some business data, and sends them as documents to Elasticsearch. If you are not going to publish any specific properties, you can use the standard Kafka Elasticsearch Connector, but it is closed for the customization, described in the following points.
- It is fundamentally, that we do not include the payload itself in the document. Instead, we save only the topics name, partition, and offset as a payload reference. This way, the saved document takes no more space than the log line, and we can save not only disk resources but also server throughput.
- Through "string field formatters" provided by Kibana, we can transform some documents properties into HTTP links that implement the functionality we need. For

example, we can present the payload reference property, which contains a message's location in the Kafka cluster, as a GET link to the collector's endpoint. The collector has only to select the location from the request, read the Kafka topic's message, archive it and return it to the user as a file. Other links can point back to Kibana / Jaeger and switch UI to different modes that provide the necessary related information "around" the issue.

The doubtless advantage of the proposed solution is the low cost of implementation and the exceptional flexibility in terms of methods of analysis of failed messages. Theoretically, this approach can be extended to some other application topics, and we can use Kibana as an application internal data traffic analyzer. For example, in this way, we can search for delayed messages in the scheduler's topics and send them for re-processing immediately.

The following code snippet illustrates a key point: reading a message from a cluster using a pointer:

```
1       public Message readMessage(String topic, int partition, long offset) {
2           TopicPartition topicPartition = new TopicPartition(topic, partition);
3           MessagingMessageConverter messageConverter = new MessagingMessageConverter();
4           messageConverter.setHeaderMapper(new DefaultKafkaHeaderMapper());
5
6           try (Consumer consumer = consumerFactory.createConsumer()) {
7               consumer.assign(Collections.singletonList(topicPartition));
8               consumer.seek(topicPartition, offset);
9               records = consumer.poll(Duration.of(10, ChronoUnit.SECONDS));
10              return messageConverter.toMessage(records.iterator().next(), null, consumer, null);
11          }
12      }
```

kafka-retry-policy-read-kafka-message.java hosted with ♡ by GitHub view raw

Instead of Elasticsearch, we can also use a traditional relational database. But we must consider that the performance of many of them degrades significantly when we have to deal with hundreds of millions of records within a single table. That's why we need sharding and, best of all, automatic sharding. In this context, automation means that new tables are added to the schema automatically, and SQL queries are automatically rewritten to select data from many such tables at once. In the PostgreSQL ecosystem, for

example, such functionality is provided by the **TimescaleDB** extension. However, if we only want to analyze and manage messages from DLQ topics, we will most likely don't have to deal with a massive amount of messages, and a regular database will do just fine.

Conclusion

I hope I demonstrated that exception handling in a distributed system is quite a complex challenge. An even more difficult task is to test the code that should implement this functionality.

I have deliberately left many interesting questions outside the scope of this article. For example, I did not consider the interaction between the retry policies defined for processing Kafka messages and the BPMN engine tasks.

Of course, such complex mechanisms are themselves sources of remarkable bugs that developers will have to spend weeks and months of their lives researching. A lot of additional complexity is added to the project by magical tools like **KafkaMessageChannelBinder**, designed to make life as easy as possible for the developer and make the technology accessible to every beginner. As a result, only a very experienced specialist with a bared debugger and a set of dirty hacks can make these magic tools work not just any way, but exactly according to the requirements.

Years ago, when I had to work a lot with JMS, I hand-built all the necessary infrastructure around the **DefaultMessageListenerContainer**. Admittedly, I extraordinarily regret that I gave in to the temptation to try an "out-of-the-box" solution in Kafka's case. In terms of reliability and flexibility, it is truly as good as the admirable enterprise-level ready solutions from Oracle, IBM, and other extremely respected vendors.

Therefore, when developing a distributed system, it is crucial to find an optimal balance between the project start-up price, the correctness of its functioning and the cost of bringing it to an ideal state. Don't forget the well-known political figure's definition: *"Pain is the way of existence of protein bodies"*.