Application Integration With Kafka - Part 2.

05 MAY 2021 // MARTIN HOLT

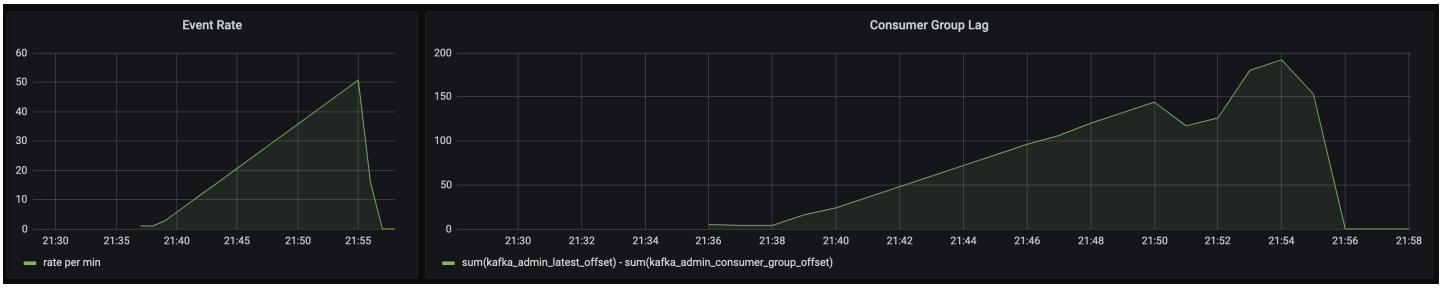
Carrying on from part 1 it's time to look at the role of the Consumer in application integration with Kafka. Watching a group of Consumers chew their way through a backlog of messages is extremely satisfying but there is peace of mind knowing that they are doing what they are supposed to.

THE SIMULATION

As before we will use a simulation based on a Micronaut microservice that both produces and consumes messages from a Kafka cluster. Code for the simulations can be found here.

CONSUMER GROUPS AND LAG

Part 1 simulated a throttled Producer. Let's take the brakes off and remove the throttling from the Kafka cluster. As the simulation runs the Producer ramps up but the single threaded Consumer cannot keep up as can be seen in the image below where the `lag` grows. Once the Producer stops producing the Consumer quickly catches up and the `lag` returns to zero:



Before we discuss `lag` in detail we need to understand Consumer Groups. Consumer Groups provide a way for a group of Consumers to traverse a topic in a coordinated manner. The Consumer Group keeps track of which offsets have been committed (i.e. which messages have been read) and distributes work between Consumers entering and leaving the group. There is a guarantee that only one Consumer will consume from a partition at any one time.

In the simulation the Consumers will join a Consumer Group called `endlessloop_group` which is created when the application starts. This simulation starts with an empty topic so it makes sense to start reading from the beginning. This is achieved by setting the `auto.offset.reset` value to `EARLIEST`. Had we chosen the `LATEST` value then the Consumer Group would start from the offset at the time the group is created, potentially missing messages.

The easiest way to speed up consumption is to add more Consumers - in Micronaut this is as simple as increasing the `threads` parameter in the `KafkaListener` (although adding more containers is also an option):

```
@KafkaListener(
        groupId = "${kafka.topic.consumer_group}",
        threads = 12,
        offsetReset = OffsetReset.EARLIEST
)
public class Consumer {
```

Once the application starts we can see below that there are 12 Consumers of which only 9 have been assigned a partition.

Time	__name__	client_id	instance	job	Value #A
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-11	endless:8080	endless	1
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-12	endless:8080	endless	1
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-13	endless:8080	endless	1
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-14	endless:8080	endless	1
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-15	endless:8080	endless	1
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-16	endless:8080	endless	1
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-17	endless:8080	endless	1
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-18	endless:8080	endless	1
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-19	endless:8080	endless	1
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-20	endless:8080	endless	0
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-21	endless:8080	endless	0
2021-05-03 22:26:12	kafka_consumer_assigned_partitions_...	endlessloop-consumer-22	endless:8080	endless	0

The guarantee of one Consumer per partition means scaling beyond the number of partitions is pointless. Fortunately it is relatively simple to add extra partitions to a topic so if volumes start to rise then you can scale to meet the challenge.

MEASURING AND MONITORING LAG

The term `lag` is used here as an indication of how much work the Consumers in a Consumer Group have left to do. In the application I have added some custom metrics in the `KafkaMetrics` class that use the AdminClient to publish:

- The offsets on each partition of the Consumer Group - metric `kafka.admin.consumer_group.offset`
- The latest offset on each partition of the topic - `kafka.admin.latest.offset`

Thus the total `lag` for the Consumer Group can be easily computed as:

```
sum(kafka.admin.latest.offset) - sum(kafka.admin.consumer_group.offset)
```

`Lag` is a great metric to monitor but I would recommend you use it with caution when setting alarms. A constantly growing `lag` indicates that your application is not keeping up and is cause for action. However `lag` may be cyclical and in this case `lag` may be acceptable if your Consumers are coping and catching up with the work to be done.

What may be more interesting is the `rate of consumption` . With some mental gymnastics you can compare the rate that the Consumer Group offset is moving to the `lag` to work out how long until the `lag` is removed. If this time period indicates that `lag` will adversely affect your customers you may want to be notified.

I would also recommend that you monitor the `rate of consumption per partition` as there are situations where consumption may stop on one partition but not all - more on that later…

To illustrate I once worked with an application where the health check said `green` but the Consumer threads had crashed and no work was being done. It took us a week to notice which left a lot of annoyed customers. It may also be a good idea to monitor the `number of assigned partitions` in the Consumer Group to ensure your Consumers are up and running.

LAG AND TOPIC RETENTION

It is a good idea to be aware of the topic retention policy especially the retention.ms property which…

> *represents an SLA on how soon consumers must read their data.*

The default value is 7 days but if you have a shorter value you may want to monitor your `lag` . The following cautionary tale illustrates why…

There were once two Postgres databases replicated using a `change data capture` pattern via Kafka. Replication was relatively fast and retention time on the topic was set to 4 hours. The exporting database used a schema that was related to the current year. At midnight on the 1st of January the import stopped working as the new year's schema was missing in the importing database. At 2am on the 1st of January I received the support call. Making schema changes in a production database on the 1st of January with 2 hours to spare can only be described as sub optimal. Be aware of the retention time on your topics (and give your Consumers some slack if you can)!

MESSAGE METADATA

Consumers themselves pass no judgement on the messages they consume so it is up to the application to make a decision on whether an action should be performed. Due to the loose coupling between Producer and Consumer it is difficult for the Consumer to know when a message will be received and the quality of that message.

For example suppose that an upstream service performs a data migration of historical data which results, eventually and maybe unintentionally, in a lot of historic data being written to a topic. Your customers may be unimpressed if your application starts congratulating them on purchases that they made a long time ago.

There is a good case for negotiating some form of metadata in the contract you have with your Producer, as with any other api. This can be acheived in the message itself or as Kafka headers as exemplified here where both a `trace id` and an `event timestamp` header have been added:

```
var record = new ProducerRecord<>(config.topic, key, message);
var headers = record.headers();
// Add a trace id
headers.add(HEADER_TRACE_ID, UUID.randomUUID().toString().getBytes(StandardCharsets.UTF_8));
// Add a iso date time representing the business event time up to two minutes back in time
headers.add(HEADER_EVENT_ISO_DATE_TIME, ...).getBytes(StandardCharsets.UTF_8));
```

On the other side the Consumer can make a decision based on the metadata - in this case messages older than 60 seconds can be ignored:

```
var now = ZonedDateTime.now();
  var eventTs = ...
  if (eventTs.isAfter(now.minusSeconds(60l))) {
      LOG.info("Consuming message with key " + key);
  } else {
      LOG.info("Ignoring message with key " + key);
  }
```

And as we can see from the logs the Consumer can now choose to ignore any messages older than 60 seconds. As a bonus I have added the `trace id` to the logs (in red below) which would aid in any discussions with the Producer.

```
  __  __ _                                  _
 |  \/  (_) ___ _ __ ___  _ __   __ _ _   _| |_
 | |\/| | |/ __| '__/ _ \| '_ \ / _` | | | | __|
 | |  | | | (__| | | (_) | | | | (_| | |_| | |_
 |_|  |_|_|\___|_|  \___/|_| |_|\__,_|\__,_|\__|
  Micronaut (v2.4.1)

21:10:38.464 [3e56c085-56dd-4c3b-afa0-7b9713d8bf2c] [pool-1-thread-9] INFO  s.m.e.c.Consumer - Ignoring message with key c95d1712-311e-4b8e-b9c7-960f64a59c83
21:10:39.373 [b9a6d70d-c19f-4494-8ac9-616cf848ee01] [pool-1-thread-7] INFO  s.m.e.c.Consumer - Ignoring message with key b6b3cb74-4bef-45dd-85f8-7a8569b67838
21:10:40.380 [3f121921-8ef8-4937-8fda-46953e5e4681] [pool-1-thread-5] INFO  s.m.e.c.Consumer - Ignoring message with key 5f74dbba-607a-4a78-adff-dcba60064800
21:10:41.340 [efee064e-a0da-45bb-b7d5-f78e0105f5e5] [pool-1-thread-7] INFO  s.m.e.c.Consumer - Ignoring message with key 68fc2c73-1cd3-4f5d-9b36-46edb5f42ea2
21:10:42.341 [8547fdc9-27e5-4ae3-b434-047976d5dc79] [pool-1-thread-2] INFO  s.m.e.c.Consumer - Consuming message with key a24eb015-07d5-4e02-99cc-a22ddbf25bdd
21:10:43.345 [7e5c7608-7cb2-4137-86af-9780d91dacd3] [pool-1-thread-6] INFO  s.m.e.c.Consumer - Consuming message with key e067015b-1875-454f-afd2-0d4f8a1bc286
21:10:44.341 [66757ba3-ca33-4aa1-9105-7d0226e35be6] [pool-1-thread-6] INFO  s.m.e.c.Consumer - Consuming message with key 197395b5-cfdf-4e31-a71a-98d0768492ab
21:10:45.342 [b105d53d-4d06-40d3-b6ae-74d03a24d6bb] [pool-1-thread-4] INFO  s.m.e.c.Consumer - Ignoring message with key 9a654657-5e11-47b5-bc62-467b3e209a80
21:10:46.349 [4b0dda33-0552-44e3-acd9-d3d39891c2b7] [pool-1-thread-4] INFO  s.m.e.c.Consumer - Ignoring message with key 9f9dff0b-b06c-46f7-b350-26fd59a0f40f
21:10:47.346 [19722524-580d-4fcb-a72e-16ae4e9f1b96] [pool-1-thread-7] INFO  s.m.e.c.Consumer - Consuming message with key cb6d3075-025e-4a6c-a7e6-0de22587f31f
21:10:48.352 [ea5ca3b9-e28e-4c87-8bf8-04484f5adb63] [pool-1-thread-8] INFO  s.m.e.c.Consumer - Ignoring message with key a593f6f9-441c-4d80-81f9-efb1a8222ebd
21:10:49.349 [541c76e5-a954-4604-a470-7bfa74993136] [pool-1-thread-9] INFO  s.m.e.c.Consumer - Consuming message with key ef7fe616-d3ce-439f-bb50-90fa74210b26
21:10:50.362 [5dbd7333-d3c9-4df9-a0e8-87dbbe7ece8c] [pool-1-thread-1] INFO  s.m.e.c.Consumer - Consuming message with key 12447fee-c922-47db-becb-40c4ffb9f234
21:10:51.364 [40af547d-3c53-4ae3-bda7-caf40070b45e] [pool-1-thread-7] INFO  s.m.e.c.Consumer - Consuming message with key a4926660-c184-43e2-bb9b-15f1b58d27e9
```

TIMING

Consumers are active processes that will chug on 24/7 acting on messages. In certain cases timing can be an issue. For example say your application sends out push notifications and your application acts on the message at 3am - you may not be popular. You may want to delay processing of certain types of message.

This is not as easy as it seems - if you choose to not process one message on a partition you will not be able to read past that message. This requires some careful thought - you probably cannot guarantee that subsequent messages warrant the same sort of delay, and you probably do not want to try to reset the offset in your consumer group to come back to a single message later. There are a few patterns you can use here:

- Offload the message to a database or in-memory cache and let another process pick up the message when the time is right
- Fan-out the messages to seperate topics where the Consumers are active only during certain times

As described in Part 1 offloading will add complexity, requires careful monitoring and can fail.

DEALING WITH FAILURE

A Consumer may read a message but fail to act on it. It is a good idea to have a plan for how to cope with this.

To simulate this let's add a 1% chance of an error where this error will only ever occur on a single message. The error is signalled using a header called `this-is-a-problem`:

```
// 1% chance of this occuring
if (ThreadLocalRandom.current().nextInt(100) == 50) {
    // Only set this header once
    if (!haveHadAProblem.compareAndExchange(false, true)) {
        headers.add(HEADER_IS_A_PROBLEM, "true".getBytes(StandardCharsets.UTF_8));
    }
}
```

If a message has this header then the Consumer should fail to consume the message:

```
if (headers.contains(HEADER_IS_A_PROBLEM)) {
    throw new IllegalStateException("There is a problem");
}
```

The Consumer fails to process the message and moves on to the next message on the partition as can be seen in the logs:

```
11:50:15.256 [pool-1-thread-2] ... key 0c33fcb8-0059-43e7-95c2-b171c0fa0c82 partition 1, offset 8
11:50:18.258 [pool-1-thread-2] ... key 28f38bc6-c7ca-4b07-975d-0fe0a2f17b9f partition 1, offset 9
11:50:20.259 [pool-1-thread-2] ... key 50c84db8-973a-4127-ad70-7d2d20dd0fa2 partition 1, offset 10
11:50:22.263 [pool-1-thread-2] ... Error processing record...  There is a problem
11:50:29.243 [pool-1-thread-2] ... key d545b7e7-1428-40ff-b8f3-52a45095fe3d partition 1, offset 12
```

In this example we are implementing the Consumer using the Micronaut KafkaListener. The default Micronaut exception handling uses the DefaultKafkaListenerExceptionHandler where the strategy is `log and proceed` - the Consumer will continue to consume from the topic partition.

If you are happy that your application fails to act on a message then this strategy is fine. However this problem may have been `transient` - a network glitch as you try to fetch information from another system say - in which case there is a chance for remedy. The problem may be `persistent` - an upstream service makes a schema change that isn't backward compatible causing this and all subsequent messages to fail without a code change - in which case you may want to stop the application.

SHUTTING DOWN THE CONSUMERS

Let's assume that the error is `persistent` and that the Consumer should give up. This can be achieved by replacing the exception handler:

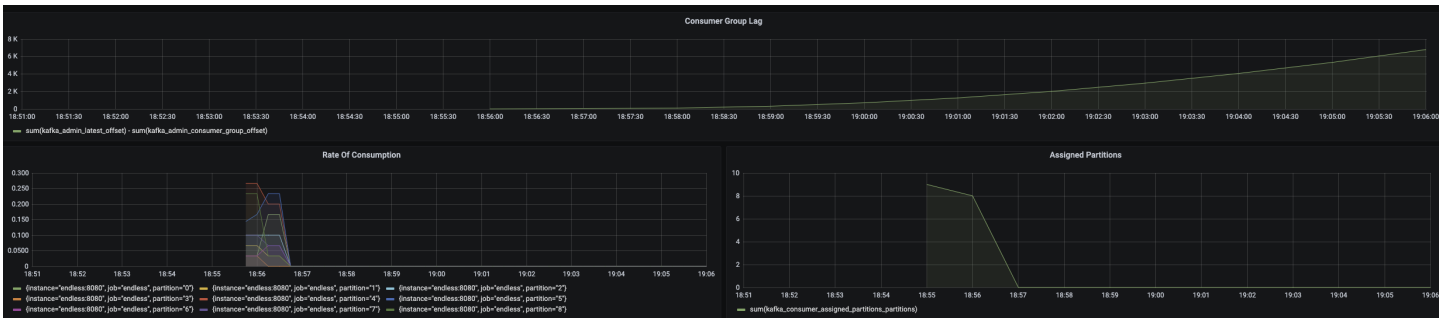
```
@Replaces(DefaultKafkaListenerExceptionHandler.class)
@Singleton
public class CustomKafkaListenerExceptionHandler implements KafkaListenerExceptionHandler {

    private static final Logger LOG = LoggerFactory.getLogger(CustomKafkaListenerExceptionHandler

    @Override
    public void handle(KafkaListenerException exception) {
        throw exception;
    }
}
```

This is pretty crude but has the effect of stopping the thread that is consuming the offending message. However as our Consumers are part of a Consumer Group the partition with the offending message will be re-assigned to another thread, which will crash, and so on and so on until no threads are consuming - this is a `poison message`. The image below

shows how the `lag` is building but the `consumption rate per partition` is zero and no partions are assigned to a Consumer.



This can be a pretty nasty type of problem (and was the cause of the week long outage mentioned above). Monitor `rate of consumption per partition` and `number of assigned partitions` to catch these cases.

You may want to shut down the consumers if a series of messages show a persistent error by using the Circuit Breaker pattern. Ignoring a persistent message when running at scale can cause headaches as all subsequent messages may have the same problem (schema compatibility issues for example). A simulation of this is out of scope for this blog post.

RETRYING TRANSIENT ERRORS

Let's try to deal with `transient` errors by once again using the Resilience4j Retry module and adding a `Retry` with a maximum of 5 retries at a fixed interval of 1 seconds.

```
private final RetryConfig retryConfig =
    RetryConfig.custom()
            .retryExceptions(IllegalStateException.class)
            .maxAttempts(5)
            .waitDuration(Duration.ofSeconds(1l))
            .build();
```

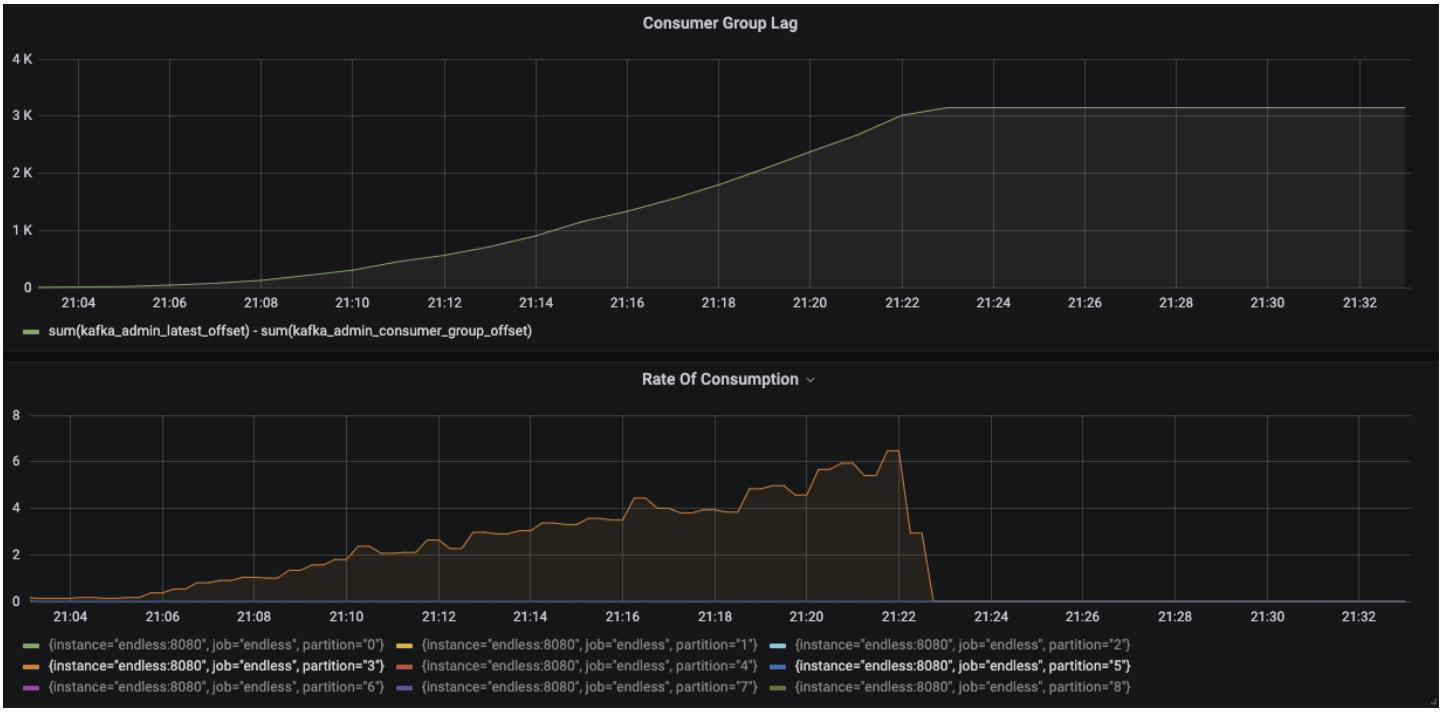
Stopping the Consumers is undesirable so we will adopt a `best effort` strategy - retry the action until the retries are exhausted then log the error and carry on. To do this we revert to the `DefaultKafkaListenerExceptionHandler`. Running this simulation will show that the application behaves as expected and the Consumers remain active.

Let's make things a bit trickier by changing the Consumer configuration max.poll.interval.ms to 3000ms (from the default of 5 minutes). If we look at the logs from the simulation we can see a pattern where a `WARN` represents a retry and an `ERROR` is where we give up:

```
20:04:11.314 [pool-1-thread-3] WARN ...partition 5, offset 34
20:04:14.317 [pool-1-thread-3] ERROR ... partition 5, offset 34
20:04:14.422 [pool-1-thread-4] WARN  ...partition 5, offset 34
20:04:15.423 [pool-1-thread-4] WARN  ...partition 5, offset 34
20:04:16.423 [pool-1-thread-4] WARN  ...partition 5, offset 34
20:04:16.460 [pool-1-thread-3] WARN  ...partition 5, offset 34
20:04:17.402 [pool-1-thread-4] WARN  ...partition 5, offset 34
20:04:17.403 [pool-1-thread-4] ERROR ... partition 5, offset 34
20:04:17.439 [pool-1-thread-3] WARN  ...partition 5, offset 34
20:04:18.440 [pool-1-thread-3] WARN  ...partition 5, offset 34
20:04:19.440 [pool-1-thread-3] WARN  ...partition 5, offset 34
20:04:19.545 [pool-1-thread-4] WARN  ...partition 5, offset 34
20:04:20.441 [pool-1-thread-3] WARN  ...partition 5, offset 34
20:04:20.442 [pool-1-thread-3] ERROR ... partition 5, offset 34
```

We can see that `thread 3` tries to deal with a message on `partition 5` but fails, `thread 4` takes over processing the same message but at the same time `thread 3` tries again. The image below shows how the `lag` grows until the Producer stops, flattens out, but never returns to zero. At the same time we can see active consumption from

`partition 3` but `partition 5` stops early on.



Why has this happened? The `max.poll.interval.ms` is used by the Consumer Group to determine the liveliness of a Consumer. If a Consumer has not polled the Consumer Group in this interval it is considered to have stopped, is ejected from the group, and its partitions are assigned to another Consumer - a process known as `Rebalancing` (more on this shortly). Five seconds of retries exceed the three second limit for `max.poll.interval.ms` so by the time the retries are exhausted the Consumer has been ejected and another thread has picked up the work.

The consequence of this that the Consumer Group can make no forward progress past the failing message on one partition. This is a great reason to monitor the `consumption rate per partition` and not just the consumption rate of the whole Consumer Group (but remember there may be no work to do on that partition, i.e. no `lag`). This situation can occur when a problem is believed to be `transient` but is actually `persistent` - for example picking an item from a data warehouse that should be there but is not - and you have been too generous with your retries.

DEAD LETTER QUEUES

One way of dealing with failure is to adopt a `deal with it later` strategy and here the `dead letter queue` pattern can help. The concept is inherited from message brokers such as IBM MQ whereby a message that cannot be handled now is placed to one side for later processing.

A similar concept could be used to handle both `persistent` and `transient` errors whereby a message is written to a seperate topic. This topic can then be consumed at a later date maybe by another version of the application. This is a relatively easy pattern to engineer but adds a lot of complexity and will require good monitoring and probably a good chunk of time to deal with. In most implementations I have experienced the dead letter queue is where messages go to die and they rarely are processed before the retention time expires - it may be better to take an active decision to drop unhandleable messages.

If you do go down this route it may be worth considering a Circuit Breaker for the case when all messages start to head to the `dead letter queue`.

TO REPUBLISH OR NOT TO REPUBLISH

It is not unusual to need to repeat the actions that are triggered by a message - send a missing email, correct an account balance, etc. It is tempting to republish the same message to the topic to trigger the action. However before you do that remember that you may not be the only consumer of the topic - republishing will affect all consuming applications, not just yours!

Republishing may not be a simple process - for example you may need to serialize the message using an Avro schema. I would strongly recommend looking into the Hexagonal Architecture pattern, also known as `ports and adapters`

which seperates the business logic from the entry points (ports). If an action needs to be performed by your application it may be easier to create a `retry topic` where you can publish the message in another format, say plain Json. Even better you may choose to expose a REST endpoint for republishing.

MESSAGE CONSUMPTION GUARANTEES

In the `Retry` example above it was noted that different Consumer threads in the same Consumer Group were handling the same message. In part 1 we discussed how Producer configuration determines a guarantee level and affects resilience and buffering. The same is true for the Consumer as described in the configuration documentation.

The Kafka Consumer `polls` the broker regularly and at some point `commits` the `offset` of the last message read on the `partition` being consumed. The `commit` moves the `offset` forward for that `partition` in the Consumer Group effectively marking which messages have been read. There can be multiple `polls` within each `commit`.

A deep dive into the `commit` cycle is out of scope for this blog and I would once again recommend O'Reilly's Kafka: The Definitive Guide for a description of controlling the `commit` cycle.

In the simulation we have delegated configuration of the Consumer to the Micronaut KafkaListener with default settings. This can however be tailored to meet our needs by adjusting the offset commit strategy. The default configuration is equivalent to:

```
enable.auto.commit: true                # auto commit enabled
auto.commit.interval.ms: 5000           # auto commit every 5 seconds
```

The KafkaListener is configured to poll every 100ms and will fetch one message at a time (`batch` value defaults to 1) so every `commit` cycle could cover up to 50 messages.

There is a risk that the Consumer crashes during a `commit` cycle and that no `commit` is performed even though messages have been actioned. As we have seen above if the client becomes inactive then the Consumer Group is `rebalanced` and the partitions are reassigned - another Consumer will start to consume that `partition` from the `offset` of the last `commit`. `Rebalancing` is one of the key resilience features of Kafka Consumers - for more details I again recommend O'Reilly's Kafka: The Definitive Guide. What is important to understand here is that some messages may be read and actioned more than once.

If there are no consequences to actioning a message twice (it is irrelevant or better still `idempotent`) then this behaviour is fine. In most cases it will not be and you will likely want a finer level of control. For our simulation let's `commit` after every message read - this is done by changing the `offsetStrategy` in the KafkaListener. In this case we will disable the auto commit and perform a manual `ack` (the offset commit strategy can be tailored to meet your needs):

```
@KafkaListener(
        groupId = "${kafka.topic.consumer_group}",
        threads = 12,
        offsetReset = OffsetReset.EARLIEST,
        offsetStrategy = OffsetStrategy.DISABLED
)
public class Consumer {

    @Topic("endless")
    public void consume(@KafkaKey UUID key, byte[] value, MessageHeaders headers, int partition,

        // do stuff
        ...
        ack.ack();
```

This configuration reduces the likelihood that a message is consumed more than once but it is all but impossible to totally remove the risk. I would argue that you still need to be ready for the same action to be applied more than once. Why?

- Your Producer may implement an `at least once` guarantee and one message may be published to two different partition offsets
- Your producing application may not be idempotent - a "double click" in a UI might unintentionally result in two messages that are semantically identical

- Someone may republish an earlier message leading to the same message on two different partitions

It can be tempting to try and identify which messages have already been processed but it is not sufficient to rely on `partition` and `offset`. Adding a unique id as metadata - say a `business-event-id` - is a way to identify unique messages but beware that this is likely to require a distributed data store which can fail and may incurr a significant cost (especially if you have a long retention on a busy topic).

One final extreme real-life example… Under a period of pressure a bug caused the offsets on all Consumer Groups to reset. Most teams didn't notice as they used an offset strategy of `LATEST` but my application used `EARLIEST` and started to eat it's way through three weeks worth of messages. We took the decision not to try to reset the offsets manually and let the application catch up. Fortunately the actions performed by our application were `idempotent` and no customer was affected apart from a short delay during catch up. We even had the last laugh as those teams with the `LATEST` offset strategy had skipped a number of messages during the reset (which effectively removed the `lag`) and had to find a way to re-consume.

This was an extreme occurrence but the key message is idempotence is your friend and if you can have it in place before you start to consume you can save yourself some heartache later on.

SUMMARY

In this blog post we have considered the Consumer and discussed how to prepare your applications.

KEY TAKEAWAYS:

- Use metrics to make sure your Consumers are working across all topics when there is work to be done
- Plan for failure and have a strategy to handle errors.
- Watch out for poisoned messages and take care when adding resilience
- Your Consumers don't care if it is 3a.m. but your customers might
- You will almost certainly have to deal with duplicates. Idempotence is your best bet

Watching a dashboard that shows a group of Consumers efficiently chewing through a backlog of work is truly a thing of beauty. I have mentioned metrics here for monitoring Consumers but let's not forget that the actions performed are often highly relevant to your business - orders purchased, communications sent, items shipped, etc. It is a short step to add metrics to these actions that can be displayed on wallboards for your business that can give you insights into how your products are being used. A dashboard of this nature, for a product launch for example, can give a big boost to a team.

For further reading I can strongly recommend O'Reilly's Kafka: The Definitive Guide that will soon be out in its second edition, and for structuring your applications I recommend Designing Data-Intensive Applications by Martin Kleppmann.

Best of luck with your Kafka journey!