Kafka Client Types and Exception Handling

- Java producer/consumer clients with minimal dependencies.
- A Spring Boot application with Kafka consumer endpoints, internal storage and web interfaces. Demonstrates setup of batch processing and batch error handling.

Purpose

- Get quickly up and running with Kafka using the standard Java Kafka clients.
- Experiment with the console clients to learn about communication patterns possible with Kafka, how topic partitions and consumer groups work in practice, and how error conditions affect the clients and the communication.
- Experiment with the settings to learn and understand behaviour, easily modify and re-run code in the experimentation process.
- Learn and experiment with setup of multiple Kafka consumers in a Spring Boot application.
- Learn about batch error handling strategies in Spring Kafka.
- Contains code and examples of tests that use a local temporary Kafka environment to execute.

Requirements

- JDK 11+
- Maven 3.6.X (must be able to handle modular Java project)

Lab Solution

Complete lab solution is available at following path:

```
cd ~/kafka-training/labs/lab-pattern-exception
chmod +x *.sh

./install-mvn.sh

source ~/.bashrc

mvn --version
```

Make sure to install maven before proceeding.

Overview

Getting started

Building

The project consists of three Maven modules:

1. messages
2. clients
3. clients-spring

The `messages` module is used by both the Spring application and regular command line clients and contains various messages types and handling of them.

The build process is standard:

```
mvn install
```

If all goes well, an executable über-jar is built in `clients/target/clients-<version>.jar` for the basic Java clients. The automated tests actually spin up Kafka on localhost, and so take a while to complete. To skip the tests during development iterations, use `mvn install -DskipTests` instead.

The jar-file can be executed simply by running `./clients.sh` from the project top directory, or alternatively using `java -jar clients/target/clients*.jar`.

Running a Kafka environment on localhost

Make sure that Zookeeper and Kafka are already running. Start them by running following script incase they are not running:

```
~/kafka-training/run-zookeeper.sh
```

Wait about 30 seconds or so for ZooKeeper to startup.

```
~/kafka-training/run-kafka.sh
```

Running the kafka-sandbox command line clients

To get started:

```
$ chmod +x clients.sh
$ ./clients.sh --help
Use: 'producer [TOPIC [P]]' or 'consumer [TOPIC [GROUP]]'
```

```
Use: 'console-message-producer [TOPIC [P]]' or 'console-message-consumer [TOPIC
[GROUP]]'
Use: 'sequence-producer [TOPIC [P]]' or 'sequence-consumer [TOPIC [GROUP]]'
Use: 'null-producer [TOPIC [P]]' to produce a single message with null value
Use: 'string-producer STRING [TOPIC [P]]' to produce a single UTF-8 encoded string
message
Use: 'newtopic TOPIC [N]' to create a topic with N partitions (default 1).
Use: 'deltopic TOPIC' to delete a topic.
Use: 'showtopics' to list topics/partitions available.
Default topic is chosen according to consumer/producer type.
Default consumer group is 'console'
Kafka broker is localhost:9092
```

(The run script will trigger a Maven build if no JAR file exists in `clients/target/` .)

The producer and consumer modes are paired according to the type of messages they can exchange. The default 'producer' creates synthentic "temperature measurement" events automatically after starting up, hence the naming of the corresponding default Kafka topic. The default 'consumer' is able to read these messages and display them as console output.

The 'sequence-producer' creates records with an ever increasing sequence number. The corresponding consumer does a simple validation of received messages, checking that the received sequence number is the expected one. This can be used to detect if messages are lost or reordered in various situations. The consumer keeps an account of the number of errors detected and writes status to stdout upon message reception.

The 'console-message-producer' is an interactive producer that reads messages you type on the command line and ships them off to a Kafka topic. The 'console-message-consumer' is able to read these messages and display them as console output. These can be used to get a more controlled message production where sending is driven by user input.

The 'null-producer' produces a single record with a null value to a topic. Used for testing error handling of poison pill messages. The 'string-producer' can produce records with UTF-8 encoded strings as payload. It can be used to trigger deserialization errors for consumers expecting JSON payload.

The commands 'newtopic', 'deltopic' and 'showtopics' allow simple administration of Kafka topics for testing purposes.

Communication patterns with Kafka

These examples assume that you have a local Kafka broker up and running on `localhost:9092` *, see [relevant section](#).*

Example: one to one

This example is possibly the simplest case and can be easily demonstrated using the command line clients in kafka-sandbox.

We will use the default topic with a single partition:

In terminal 1:

```
./clients.sh producer
```

The producer will immediately start sending messages to the Kafka topic 'measurements'. Since this default topic only has one partition, the exact place where the messages will be stored can be denoted as 'measurements-0',

meaning partition 0 for the topic.

In terminal 2:

```
./clients.sh consumer
```

The consumer will connect to Kafka and starting polling for messages. It will display the messages in the console as they arrive. The consumer subscribes to the topic 'measurements', but does not specify any partition in particular. So it will be assigned a partition automatically.

The consumer uses the default consumer group 'console'. The consumer group concept is important to understand:

1. The consumer group is simply a named identifier chosen by the clients.
2. There can only be *one* consumer client instance in a particular consumer group assigned to a single topic-partition at any given time.
3. Consumed partition offsets for a topic is stored per *consumer group*. In other words, Kafka stores the progress on a *per consumer group* basis, for a particular topic and its partitions. (Consumer clients are responsible for comitting the progress back to Kafka.)
4. When a new consumer group name is established, the consumers which are part of that group will typically start receiving only new messages sent to the topic. This is however configurable, and the consumers in kafka-sandbox are by default setup to start at the very beginning of a topic, if Kafka has no stored offset data for the group to begin with.
5. When the constellation of consumers in the same consumer group connected to a topic changes, Kafka will rebalance the consumers and possibly reassign partitions within the group.

To observe what happens when a consumer disconnects and reconnects to the same topic:

1. Stop the running consumer in terminal 2 by hitting `CTRL+C` . (You may notice in the Kafka broker log that the consumer instance left the topic.)

2. Start the consumer again. Notice that it does not start at the beginning of the Kafka topic log, but continues from the offset where it left off. This is because the consumer group offset is stored server side.

3. Kill the consumer again, and restart with a different (new) consumer group:

   ```
   ./clients.sh consumer measurements othergroup
   ```

   Notice how it now starts displaying messages from the very beginning of the topic (offset 0). This is because no previous offset has been stored for the 'othergroup' group in Kafka and the client is configured to start at the beginning of the topic in that situation.

   What happens when a second consumer joins ? Start a second consumer in a new terminal window:

   ./clients.sh consumer measurements othergroup

   You will now notice that one of the two running consumers will stop receiving messages, and in that case the following message will appear:

> *Rebalance: no longer assigned to topic measurements, partition 0*

This is because the topic only has one partition, and only one consumer in a single consumer group can be associated with a single topic partition at a time.

If you now kill the consumer that currently has the assignment (and shows received messages), you will notice that Kafka does a new rebalancing, and the previously idle consumer gets assigned back to the partition and starts receiving messages where the other one left off.

Example: one to many

One to many means that a single message produced on a topic is typically processed by any number of different consumer groups.

Initialize a new topic with 1 partition and start a producer:

```
./clients.sh newtopic one_to_many 1

./clients.sh producer one_to_many
```

And fire up as many consumers as desired in new terminal windows, but increment the group number N for each one:

```
./clients.sh consumer one_to_many group-N
```

You will notice that all the consumer instances report the same messages and offsets after a short while. Because they are all in different consumer groups, they all see the messages that the single producer sends.

Example: one time message processing with parallel consumer group

In this scenario, it is only desirable to process a message once, but it can be processed by any consumer in a consumer group.

Create a topic with 3 partitions:

```
./clients.sh newtopic any_once 3
```

Start three producers in three terminals, one for each partition:

```
./clients.sh producer any_once 0

./clients.sh producer any_once 1

./clients.sh producer any_once 2
```

Here we are explicitly specifying which partition each producer should write to, so that we ensure an even distribution of messages for the purpose of this example. If partition is left unspecified, the producer will select a partition based on the Kafka record keys. The producer of "measurement" messages in the demo code uses a fixed "sensor device id" based on the PID as key, and so the messages become fixed to a random partition. See the Apache code for class `org.apache.kafka.clients.producer.internals.DefaultPartitioner` - it is not complicated and explains it in detail. The partitioner class [strategy] to use is part of the Kafka producer config.

Next, we are going to start consumer processes.

Begin with a single consumer:

```
./clients.sh consumer any_once group
```

You will notice that this first consumer gets assigned all three partitions on the topic and starts displaying received messages.

Let's scale up to another consumer. Run in a new terminal:

```
./clients.sh consumer any_once group
```

When this consumer joins, you can see rebalancing messages, and it will be assigned one or two partitions from the topic, while the first is removed from the corresponding number of partitions. Now the load is divided betweeen the two running consumers.

Scale further by starting a third consumer in a new terminal:

```
./clients.sh consumer any_once group
```

After the third one joins, a new rebalancing will occur and they will each have one partition assigned. Now the load is divided evenly and messages are processed by three parallel processes.

Try to start another fourth consumer (same topic/group) and see what happens. (Hint: you will not gain anything wrt. message processing capacity.)

Many to many

The previous example can also be considered a many to many example if more consumers are started in several active consumer groups. In that case, all the messages produced will be handled in parallel by several different groups (but only once per group).

Consumer group rebalancing

You will notice log messages from the consumers whenever a consumer group rebalancing occurs. This typically happens when a consumer leaves or a new consumer arrives. It will provide insight into how Kafka distributes messages amongst consumers in a group.

Many to one

This example demonstrates a many-to-one case, where there are lots of producers collecting "temperature sensor events" and sending it to a common topic, while a single consumer is responsible for processing the messages.

Start a single consumer for topic 'manydevices':

```
./clients.sh consumer manydevices
```

Start 10 producers by executing the following command 10 times:

```
$ ./clients.sh producer manydevices 1>/dev/null 2>&1 &
[x10..]
```

The producers will be started in the background by the shell and the output is hidden. (You can examine running background jobs with the command `jobs`.)

After a short while, you should see the consumer receiving and processing messages from many different producers ("sensor devices"). Depending on the number of running producers, you may see the consumer receiving multiple records per poll call to Kafka. This is simply due to the increased rate of messages being written to the topic.

To kill all producers running in the background, execute command:

```
kill $(jobs -p)
```

Error handling in general

The demo clients in this app are "unsafe" with regard to message sending and reception. The producer does not care about failed sends, but merely logs it as unfortunate events. Depending on business requirements, you will likely

need to take proper care of exception handling and retry policies, to ensure no loss of events at either the producing or consuming end.

Error handling: broker goes down

What happens to a producer/consumer when the broker suddenly stops responding ? In particular, what happens to the messages that are being sent ? Are they lost or can they be accidentally reordered ?

Here is a recipe to experiment with such scenarios.

Run a producer and a consumer in two windows:

```
$ ./clients.sh producer
[...]

$ ./clients.sh consumer
[...]
```

Then pause the docker container with the broker to simulate that it stops responding:

Stop the run-kafka script and quickly run again

Now watch the error messages from the producer that will eventually appear. A prolonged pause will actually cause messages to be lost with the current kafka-sandbox code. It keeps trying to send new messages without really caring what happens to already dispatched ones. Depending on use case, this may not be desirable, and one may need to develop code that always retries failed sends to avoid losing events.

The producer recovers and sends its internal buffer of messages that have not yet expired due to timeouts.

You may also stop the broker entirely for some time , which causes it to lose its runtime state, and see what happens with the clients:

Stop the run-kafka script, wait for some time and run the script again to start Kafka

You'll notice that the clients recover eventually, but if it is down for too long, messages will be lost. Also, you will notice rebalance notifications from consumers once they are able to reconnect to the broker.

Behaviour can be adjusted by the many config options that the Kafka clients support. You can experiment and modify config by editing the code in `no.nav.kafka.sandbox.KafkaConfig` , see `#kafkaProducerProps()` and `#kafkaConsumerProps(String)` .

Error handling: detecting message loss with sequence-producer/consumer

The 'sequence-producer' and corresponding 'sequence-consumer' commands can be used for simple detection of message loss or reordering. The producer will send messages containing an ever increasing sequence number, and the consumer validates that the messages it receives have the expected next number in the sequence. When validation fails it logs errors and increases an error counter, so that it is easy to spot.

Start the producer:

```
./clients.sh sequence-producer
```

It will start at sequence number 0. If you restart, it will continue from where was last stopped, since the next sequence number is persisted to a temporary file. (To reset this, stop the sequence-producer and remove the file `target/sequence-producer.state` .)

Now start the corresponding consumer:

```
./clients.sh sequence-consumer
```

It will read the sequence numbers already on the topic and log its state upon every message reception. You should see that the sequence is "in sync" and that the error count is 0.

While they are running, restart the Kafka broker:

Kill the run-kafka script and quickly run again.

You should see the producer keeps sending messages, but does not receive acknowledgements. Eventually it will log errors about expired messages. The consumer may also start logging errors about connectivity, depending on how long the broker is down, which depends on how fast the host machine is. (If the broker restart is too quick to cause any errors, use "stop/start" instead, and wait a little while before starting.)

Normally, with the current code in kafka-sandbox, you can observe that some messages are lost in this process, and the consumer increases the error count due to receiving an unexpected sequence number.

There is a challenge here: modify the kafka-sandbox code or config make it more resilient. Ensure that no sequence messages are lost if Kafka stops responding for about 60 seconds, for whatever reason. Test by re-running the procedure described in this section. (Hint: see the various producer timeout config parameters.)

To only display output related to the sequence number producer/consumer, you can pipe the output of the start commands to `...|grep SEQ`, which will filter out the other log messages.

Error handling: consumer dies

What happens within a consumer group when an active consumer suddenly becomes unavailable ?

Start a producer and two consumers with a simple 1 partition topic:

```
./clients.sh producer sometopic
```

Then two consumers in other terminal windows:

```
./clients.sh consumer sometopic group
```

You will notice that one of the consumers is idle (no "untaken" partitions in consumer group), and the other one is assigned the active partition and is processing messages. Figure out the PID of the *active* consumer and kill it with `kill -9`. (The PID is printed to the console right after the consumer is started.)

```
kill -9 <PID>
```

This causes a sudden death of the consumer process and it will take a short while until Kafka notices that the consumer is gone. Watch the broker log and what eventually happens with the currently idle consumer.

The Spring Boot application

The Spring Boot application is in Maven module `clients-spring/`.

The application requires that you have a local Kafka broker up and running on `localhost:9092`, see [relevant section](#).

Running

```
mvn install               # in top project directory to ensure module 'messages' is
installed
cd clients-spring
mvn spring-boot:run
```

.. or use the convenience script `spring-boot.sh` from project top level directory, which is used in all examples:
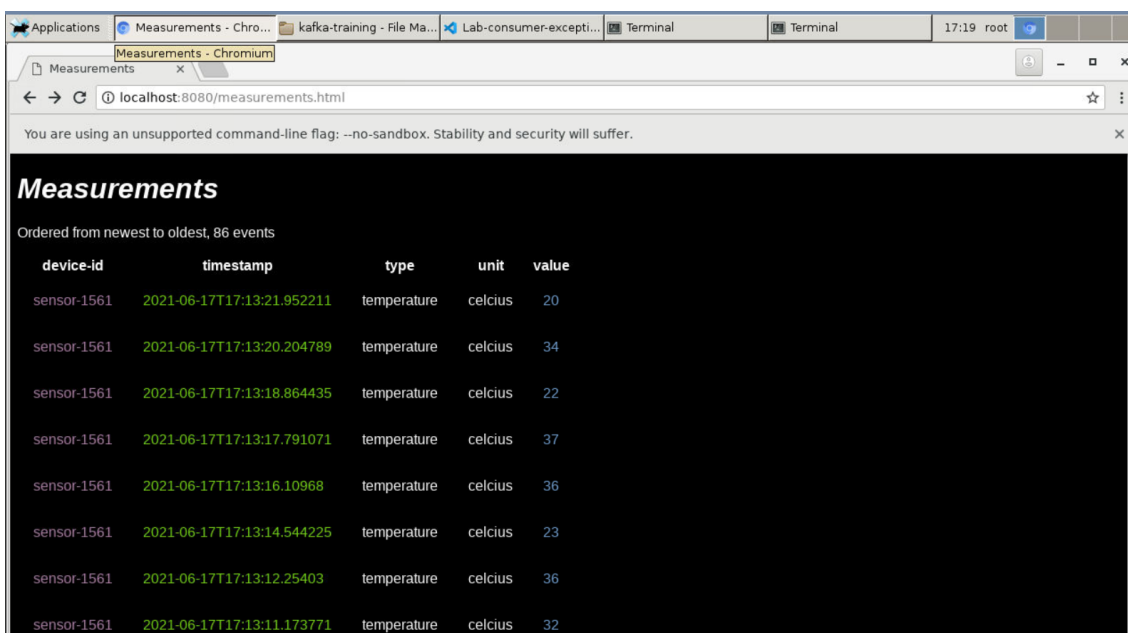
```
chmod +x spring-boot.sh
./spring-boot.sh
```

The application will automatically subscribe to and start consuming messages from the topics `measurements` (the standard producer in previous examples) and `messages` (for messages created by `console-message-producer` client). The consumed messages are stored in-memory in a fixed size event store.

Web interfaces

http://localhost:8080/measurements.html

A web page showing measurements/"sensor event" messages from Kafka. It uses an API endpoint available at http://localhost:8080/measurements/api

http://localhost:8080/messages.html

A web page showing "console message" events from Kafka. It uses an API endpoint available at http://localhost:8080/messages/api
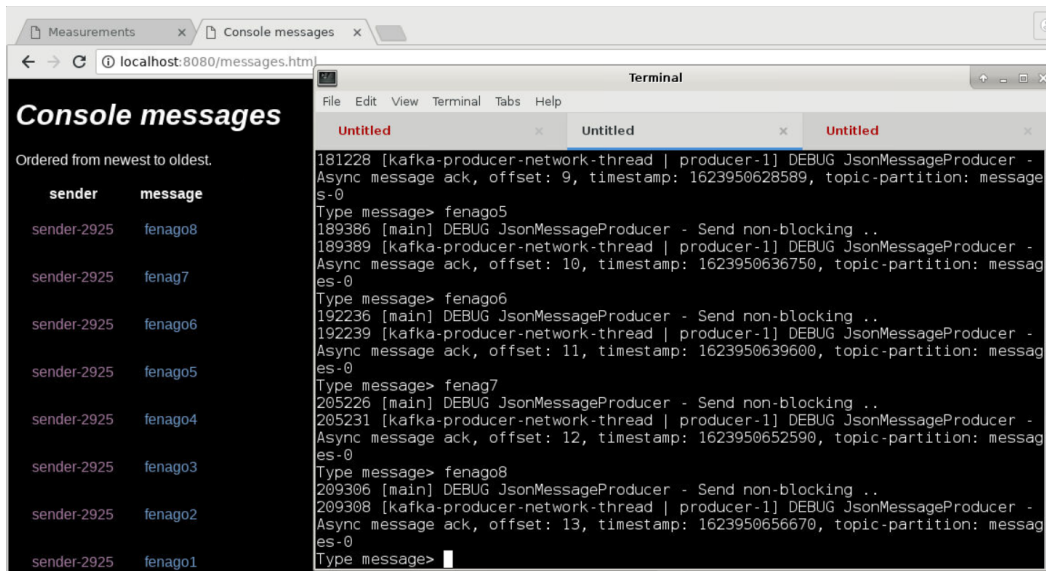
Talk to Spring boot application (or yourself) using Kafka

1. In one terminal, start the Spring boot application as described earlier.

2. In another terminal, from project root dir, start the command line console message producer:

```
$ ./clients.sh console-message-producer
36 [main] INFO Bootstrap - New producer with PID 19445
```

```
191 [main] INFO JsonMessageProducer - Start producer loop
Send messages to Kafka, use CTRL+D to exit gracefully.
Type message>
```

3. Navigate your web browser to http://localhost:8080/messages.html

4. Type a message into the terminal. As soon as the Spring application has consumed the message from Kafka, the web page will display it.

Show live view of measurements as they are consumed by Spring application

1. In one terminal, start the Spring boot application as described earlier.

2. In another terminal, start a measurement producer:

```
./clients.sh producer
```

3. Navigate your web browser to http://localhost:8080/measurements.html

4. Observe live as new measurement events are consumed by the Spring application and displayed on the page. New events are highlighted for a brief period to make them visually easier to distinguish.

Experiment with Spring application

In the previous scenario, try to artificially slow down the Spring application consumer and see what happens to the size of the batches that it consumes. To slow it down, start with the following arguments:

```
./spring-boot.sh --measurements.consumer.slowdown=5000
```

This will make the Kakfa listener endpoint in
`no.nav.kafka.sandbox.measurements.MeasurementsConsumer#receive` halt for 5 seconds every time Spring invokes the method with incoming messages. This endpoint is setup with batching enabled, so you should see larger batches being processed, depending on amount of messages produced, and how slow the consumer is.

The listener endpoint logs the size of batches it processes, so you should be able to see it in the application log. By default, the listener endpoint is invoked by at most two Spring-kafka managed threads (each with their own

`KafkaConsumer`). This is setup in `no.nav.kafka.sandbox.measurements.MeasurementsConfig` , locate line with `factory.setConcurrency(2);` . Do you think concurrency above 1 has any effect when the topic only has one partition `measurements-0` ? Inspect the thread-ids in the application log as it consumes messages, to determine if there is actually more than one thread invoking the listener method.

To produce more messages in parallel, you can start more producers in the background:

```
./clients.sh producer &>/dev/null &
```

As more producers start, you should notice the logged batch sizes increase, since volume of messages increases and the consumer is slowed down. (Note: to clean up producers running in the background, you can kill them with `kill $(jobs -p)` .)

Going further, you can test true parallel messages consumption in Spring, by changing the number of partitions on the `measurements` topic:

1. Stop Spring Boot application and any running command line producer/consumer clients.

2. Delete measurements topic: `./clients.sh deltopic measurements` .

3. Create new measurements topic with 4 partitions: `./clients.sh newtopic measurements 4`

4. Start Spring boot application as described earlier.

5. Start several producers in the background, as described earlier.

6. Watch Spring application log and notice that there are now two different thread ids invoking the listener endpoint in `no.nav.kafka.sandbox.measurements.MeasurementsConsumer#receive` .

Experiment: batch consumer error handling in Spring Kafka

In general, for the following experiments, you should ensure the Spring Boot app is stopped before following the instructions.

Error handling is important in a distributed asynchronous world. It is often difficult to get right, both because the various error situations can be complex, but also hard to reproduce or picture in advance. The typical result of poor or ignored error handling is growing inconsistencies between data in various systems. In others words, things will eventually become inconsistent intead of consistent !

You can simulate failure to store events by adjusting the configuration property `measurements.event-store.failure-rate` . It is a floating point number between 0 and 1 which determines how often the store that the consumer saves events to should fail with an exception, triggering Spring Kafka error handling.

You can also simulate bad messages by using `./clients.sh null-producer` or `./clients.sh string-producer "{bad-json"` (will cause deserialization failure in Spring Boot app).

Try this:

1. Ensure a producer for 'measurements' topic is running:

   ```
   ./clients.sh producer
   ```

2. Start Spring Boot app in another terminal with:

```
./spring-boot.sh --measurements.event-store.failure-rate=1
```

Now all (100%) store operations will fail with `IOException`. See what happens in the application log. Error handling is all Spring Kafka defaults. Does Spring commit offsets when exceptions occur in consumer ? In other words, does it ever progress beyond the point of failure ?

Now switch to a custom error handler which ignores errors, but still logs them:

```
./spring-boot.sh --measurements.event-store.failure-rate=1 --
measurements.consumer.error-handler=ignore
```

Does Spring Kafka commit offsets and progress through the topic in this case ?

Experiment: batch consumer error handling in Spring Kafka: infinite retries

Try this:

1. Start Spring Boot app with default error handling:

   ```
   ./spring-boot.sh --measurements.consumer.error-handler=spring-default
   ```

2. In another terminal, send a single null-message to the "measurements" topic:

   ```
   ./clients.sh null-producer measurements
   ```

This message will fail in the consumer with a `NullPointerException`, since messages with a `null` value are not accepted by the consumer (although they are allowed by Kafka, and do not fail on JSON deserialization).

Does Spring ever give up retrying the failing batch ? Watch the log from the Spring Boot app.

Experiment: batch consumer error handling in Spring Kafka: limited retries ?

Note that if you have messages on a topic that cause failures and you want to start fresh for a new experiment, you can just delete the topics first:

```
./clients.sh deltopic measurements
./clients.sh deltopic messages
```

Now try this:

1. Start Spring Boot app with error handling that should give up after 2 retries:

   ```
   ./spring-boot.sh --measurements.consumer.error-handler=seek-to-current-with-
   backoff
   ```

2. In another terminal, send a single null-message to the 'measurements' topic:

   ```
   ./clients.sh null-producer measurements
   ```

Does Spring ever give up retrying the failing batch ? What is the delay between retry attempts ? Watch the log from the Spring Boot app.

The configuration for this error handler can be found in `MeasurementsConfig` . Notice a subtlety here: even though the backoff is configured to retry at most 2 times, *it still continues forever*. The reason for this is explained in the docs.

Experiment: batch consumer error handling in Spring Kafka: really limited retries

To actually limit number of retries for failed messages when using batch consumer, there are a few other options. We will test RetryingBatchErrorHandler here.

Try this:

1. Clear topic measurements with `./clients.sh deltopic measurements` .

2. Start measurements producer and let it send 2-3 valid messages to the topic:

   ```
   ./clients.sh producer
   ```

   Stop it with `CTRL+C` .

3. Send a poison pill `null` message to the measurements topic:

   ```
   ./clients.sh null-producer
   ```

4. Send 2-3 more valid measurement events:

   ```
   ./clients.sh producer
   ```

   After quitting there will be somewhere between 5-10 messages present on the topic, including the `null` message in the middle.

5. Start Spring Boot app with retrying error handler that should give up after 2 retries:

   ```
   ./spring-boot.sh --measurements.consumer.error-handler=retry-with-backoff
   ```

Watch the logs. The batch is processed multiple times by `MeasurementsConsumer` , however it gives up after two retries, since the `null` message will cause a failure in the middle of the batch processing. Note that only the events processed before the poison pill in the batch are actually stored. We lost the messages coming after the poison `null` message in the batch entirely ! See if you can spot the end result by navigating to https://localhost:8080/measurements.html. You should only see the messages produced in step 2.

After giving up, the error handler logs an error and a list of discarded messages. But this error handler also allows us to configure a `ConsumerRecordRecoverer` which will be given the opportunity to recover messages, one by one, after all retry attempts have been exhausted. You can redo this experiment with the recovery option by running Spring boot app with error handler `retry-with-backoff-recovery` instead of `retry-with-backoff` . Are messages after the poision pill still lost ? See the code for this in `RetryingErrorHandler` .

(In general, if you can ensure your storage is idempotent, you will be saving yourself some trouble in these situations, so that multiple writes of the same data is not a problem.)

Experiment: batch consumer error handling in Spring Kafka: limited retries and recovery

A more sophisticated error handler called [RecoveringBatchErrorHandler](#) is also available. Notice in the code for `RetryingErrorHandler` that it has to take care of storing valid events to the event store, which is also the consumer code's main job. So there is a slight duplication of efforts there. Constrast to code in `RecoveringErrorHandler` which only needs to bother with exactly those messages that caused failure in a batch. Those cannot be stored anyway, so it does not require access to the event store.

Try this:

1. Go through steps 1-4 in the previous section, so that you end up with a poison pill null message in between other valid messages on the topic.

2. Start Spring Boot app with the recovering error handler:

   ```
   ./spring-boot.sh --measurements.consumer.error-handler=recovering
   ```

   Notice in the the logs as the consumer first reports receiving the batch of messages. An exception is thrown because of the `null` message. The next time the consumer is invoked, it receives fewer messages in the batch, because the Spring error handler has automatically committed all messages up to, but not including, the failing message. So those previous messages need not be attempted again.

It then tries to run the rest of the batch up until retries are exhausted, then it invokes the custom recovery handler. This recovery handler just logs that the null message is discarded. After that, the last messages in the batch are handed over to the consumer, which stores those successfully.

End result: all valid messages that could be stored, have been stored, and the poison pill null message was simply skipped. Also, there is a performance benefit when comparing to the `RetryingErrorHandler`, since the valid messages are not written multiple times to the store during retries/recovery.

What about transient failures when storing events ?

If a valid message fails to be written into the event store, an `IOException` of some kind is typically thrown. This may be just a temporary condition, so it often makes sense to just retry until all messages in a batch are successfully written. The recovering error handler deals with this situation in exactly that way, by simply throwing an exception from the record recovery code.

For this to work, the consumer listener must wrap exceptions in the Spring exception `BatchListenerFailedException` to communicate to the error handler which record in the batch failed. The error handler will take care of the rest.

You can try this of course:

1. Ensure our test topic is empty with `./clients.sh deltopic measurements`.

2. Start Spring boot app with an event store that sometimes fail and using the recovering error handler:

   ```
   ./spring-boot.sh --measurements.consumer.error-handler=recovering --measurements.event-store.failure-rate=0.5
   ```

3. Start a producer in another terminal:

   ```
   ./clients.sh producer
   ```

4. Let it run for a little while and watch Spring Boot app logs. You will see errors when event store fails.

5. Stop producer with `CTRL+C` , noting how many messages it sent to the Kafka topic (it is logged when it quits). Notice in Spring boot logs that it continues working on batches, which is also getting smaller each time, as more messages are *eventually written successfully* to the event store.

6. Navigate to http://localhost:8080/measurements.html and look at how many events have been successfully written to the event store. There should be *none missing*, even though the store failed half of the write attempts !

Handling failure to deserialize messages in batch consumer

Since the deserialization step happens before our consumer listener gets the message, it needs to be handled in a special way. Spring has designed the `ErrorHandlingDeserializer` Kafka deserializer for this purpose. It catches failed attempts at deserializing from a delegated deserializer class.

By default, our Spring Boot app is setup to handle failures with deserializing values.

Test it out:

1. Start Spring boot app:

```
./spring-boot.sh
```

2. Then send badly formatted data to the 'measurements' topic:

```
./clients.sh string-producer '}badjson' measurements
```

Watch logs in Spring boot app. You'll see an error logged with information about the record that failed. This is accomplished by using Spring-Kafka `ErrorHandlingDeserializer` which delegates to the `JsonDeserializer` . Code for setting this up is found in `MesurementsConfig` . Code for extracting the error in the batch listener can be found in `MeasurementsConsumer` .

Now try disabling handling of deserialization errors:

1. Start Spring boot app:

```
./spring-boot.sh --measurements.consumer.handle-deserialization-error=false
```

2. Send badly formatted data to the 'measurements' topic:

```
./clients.sh string-producer '}badjson' measurements
```

You will notice that the Spring boot app now behaves in an undesirable way, both because it will never progress past the bad record (unless it is unassigned from the topic-partition and another consumer in the same group picks up the bad record), and because it does no backoff delaying with the default error handler, so the listener container keeps failing over and over rapidly.

To improve things slightly, you can select an error handler that does backoff-delaying, like `recovering` :

```
./spring-boot.sh --measurements.consumer.handle-deserialization-error=false --
measurements.consumer.error-handler=recovering
```

It will not be able to progress beyond the error, but at least it does delay between attempts to avoid flodding logs and using up a lot of resources.

The best practice is to actually handle deserialization errors, like demonstrated earlier.

Stopping consumer on fatal errors

Sometimes an error is definitively not recoverable and some form of manual intervention or application restart is required. An example may be that the consumer is not authorized to write data to the target data store due to invalid credentials. In such cases you can look to `ContainerStoppingBatchErrorHandler` in Spring Kafka, which by default simply shuts down the consumer on any error thrown from the batch processing in the listener. By extending this class, you can test the type of error and delegate to the super class if the error is deemed fatal.

To see how an *unmodified* `ContainerStoppingBatchErrorHandler` works, you can do the following:

1. Ensure a poison pill message is present on topic:

   ```
   ./clients.sh null-producer
   ```

2. Start a regular producer and let it run:

   ```
   ./clients.sh producer
   ```

3. Start Spring boot app with `stop-container` error handler:

   ```
   ./spring-boot.sh --measurements.consumer.error-handler=stop-container
   ```

Check logs. You will see that as soon as the posion pill message is encountered the error handler kicks into action and stops the Spring message listener container. Then all activity stops and the topic-partition is unassigned from the consumer. As this app does not provide any way of re-starting the consumer, the app itself must be restarted to try again.

Further reading and experimentation

You can investigate and modify code in `MeasurementsConfig` and in package `no.nav.kafka.sandbox.measurements.errorhandlers` to experiment further. Spring has a large number of options and customizability with regard to error handling in Kafka consumers.

Also see https://docs.spring.io/spring-kafka/reference/html/#error-handlers

Tuning logging to get more details

If you would like to see the many technical details that the Kafka clients emit, you can set the log level of the Apache Kafka clients in the file `src/main/resources/simplelogger.properties`. It is by default `WARN`, but `INFO` will output much more information. For the Spring Boot application, logging setup is very standard and can be adjusted according to Spring documentation.