**Spring Boot-Kafka Exception Handling**

This document will discuss exception handling in Spring boot Kafka based message processing applications.

Consider an application that consumes messages from a Kafka topic and processes those messages to update the information in the system. If the application fails to process a message, information will not be updated in the system and that message will be lost.

By default, the error handler retries a delivery 10 times, when a Kafka consumer fails to process a message. Failures are simply logged after the deliveries are exhausted. While re-trying, consumer thread is suspended and there are no Consumer.poll () requests to the assigned partition. That means any messages received by the partition will not be received by the consumer until the retry attempts are completed. Also, the consumer group’s offset is not committed until the re-tries are over.

We can, however, change this behavior. In the example below, we will implement a non-blocking logic where a failed record will be sent to a separate retry-topic for processing. This will keep our main consumer thread available to process the subsequent messages.

@KafkaListener(topics = "#{'${app.consumer.subscribed-to.topics}'.split(',')}", containerFactory="kafkaListenerContainerFactory", groupId = "${app.consumer.group-id}")

**public** **void** consume(EventMessage eventMessage) **throws** Exception {

**try** {

logger.info(String.*format*("Consumed Message: %s", eventMessage));

//Process message

EventMessage msg= **new** EventMessage(eventMessage.getDescription()+"");

eventMessageService.process(msg);

} **catch** (Exception e) {

**throw** e;

}

}

Our service will throw an Runtime exception, if the message starts with pattern “error”.

We will create a separate thread to listen to our re-try topic.

We configure two different Kafka listener container factories to be used by the main topic consumer and the retry topic consumer.

@KafkaListener(topics = "${app.retry.topic}", containerFactory="kafkaRetryListenerContainerFactory", groupId = "${app.consumer.group-id}")

**public** **void** retry(EventMessage eventMessage) **throws** Exception {

**try** {

logger.info(String.*format*("Recieved Message in retry: %s" , eventMessage));

//Do something to re-process the message

EventMessage msg= **new** EventMessage(eventMessage.getDescription()+"");

eventMessageService.process(msg);

} **catch** (Exception e) {

**throw** e;

}

}

Here is the configuration of the Kafka listener container factory, used by the main topic consumer. Failed message is sent directly to the retry topic without any retry attempts.

@Bean

**public** ConcurrentKafkaListenerContainerFactory<Object, Object> kafkaListenerContainerFactory(

ConcurrentKafkaListenerContainerFactoryConfigurer configurer,

ConsumerFactory<Object, Object> kafkaConsumerFactory,

KafkaTemplate<Object, Object> template

, ObjectMapper objectMapper

)

{

ConcurrentKafkaListenerContainerFactory<Object, Object>

factory = **new** ConcurrentKafkaListenerContainerFactory<>();

configurer.configure(factory, kafkaConsumerFactory);

factory.setErrorHandler(**new** SeekToCurrentErrorHandler((record, exception) -> {

**try** {

//Send message to the retry topic

template.send(retryTopic, objectMapper.readValue(record.value().toString(), Object.**class**));

} **catch** (JsonMappingException e) {

logger.error(e.getMessage());

} **catch** (JsonProcessingException e) {

logger.error(e.getMessage());

}

}, **new** FixedBackOff(0, 0)));//A simple BackOff implementation with No retries.

**return** factory;

}

Below is the configuration of the Kafka listener container factory, used by the retry topic consumer:

@Bean

**public** ConcurrentKafkaListenerContainerFactory<Object, Object> kafkaRetryListenerContainerFactory(

ConcurrentKafkaListenerContainerFactoryConfigurer configurer,

ConsumerFactory<Object, Object> kafkaConsumerFactory,

KafkaTemplate<Object, Object> template

,ObjectMapper objectMapper

)

{

ConcurrentKafkaListenerContainerFactory<Object, Object>

factory = **new** ConcurrentKafkaListenerContainerFactory<>();

configurer.configure(factory, kafkaConsumerFactory);

factory.setErrorHandler(**new** SeekToCurrentErrorHandler((record, exception) -> {

**try** {

template.send(dltTopic, objectMapper.readValue(record.value().toString(), Object.**class**));

} **catch** (JsonMappingException e) {

logger.error(e.getMessage());

} **catch** (JsonProcessingException e) {

logger.error(e.getMessage());

}

}, **new** FixedBackOff(retryInterval, retryAttempts)));//A simple BackOff implementation that provides a configured interval between two attempts and a configured number of retries.

**return** factory;

}

We used SeekToCurrentErrorHandler, that seeks all unprocessed topics and partitions so that the current record (and the others remaining) are retrieved by the next poll.

We used FixedBackOff to configure our retry policy. Delivery will be attempted for a maximum number of attempts configured in the property file. Retry interval is also configured in the property file. We send message to the dead letter topic, after exhausting our retry attempts.

**Serialization, Deserialization, and Message Conversion**

Here is our code to publish messages to a Kafka topic: -

**public** **void** sendEventMessage(String topic, String input) {

logger.info(String.*format*("Producing message: %s", input));

kafkaTemplate.send(topic,**new** EventMessageTypeTwo(input));

}

Rest Controller: -

@RestController

@RequestMapping(value = "/kafka")

**public** **class** Controller {

@Value("${app.consumer.subscribed-to.topics}")

**private** String topicToPublish;

**private** **final** EventProcessor eventProcessor;

@Autowired

Controller(EventProcessor eventProcessor) {

**this**.eventProcessor = eventProcessor;

}

@PostMapping(path = "/send/message/{event}")

**public** **void** sendEventMessage(@PathVariable String event) {

**this**.eventProcessor.sendEventMessage(topicToPublish, event);

}

}

Notice that we are sending messages of type “EventMessageTypeTwo”, and our consumer is consuming messages of type “EventMessage”. We used MessageConverter implementation StringJsonMessageConverter. SpringBoot automatically injects the converter to the listener container factory.

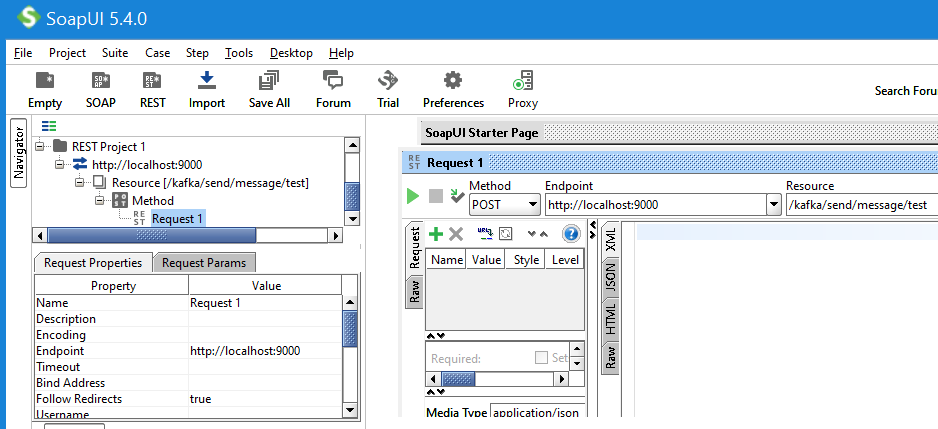
@Bean

**public** RecordMessageConverter converter() {

**return** **new** StringJsonMessageConverter();

}

When we use @KafkaListener annotation, the parameter type is provided to the message converter to assist with the conversion. Message is expected to be compatible.

If we run our Spring Boot application, we can invoke rest resource http://localhost:9000/ kafka/send/message/{event} using the Post method.

When a deserializer fails to deserialize a message, Spring cannot handle the exception as it occurs before the poll() returns. Spring Kafka introduced ErrorHandlingDeserializer that wraps a delegate deserializer. If the delegate fails to deserialize the record content, ErrorHandlingDeserializer returns a DeserializationException, containing the cause and raw bytes. These are passed on to the listener container which invokes ErrorHandler with the failed record.

In our example code, if we change the consumers of the main and retry topics to expect a message of type EventMessageTypeThree, which is not compatible with the message sent by the sender, the message will be delivered to the dead letter topic.

@KafkaListener(topics = "#{'${app.consumer.subscribed-to.topics}'.split(',')}", containerFactory="kafkaListenerContainerFactory", groupId = "${app.consumer.group-id}")

**public** **void** consume(EventMessageTypeThree eventMessage) **throws** Exception {

**try** {

logger.info(String.*format*("Consumed Message: %s", eventMessage));

//Process the message

EventMessage msg= **new** EventMessage(eventMessage.getDescription()+"");

eventMessageService.process(msg);

} **catch** (Exception e) {

**throw** e;

}

}

@KafkaListener(topics = "${app.retry.topic}", containerFactory="kafkaRetryListenerContainerFactory", groupId = "${app.consumer.group-id}")

**public** **void** retry(EventMessageTypeThree eventMessage) **throws** Exception {

**try** {

logger.info(String.*format*("Recieved Message in retry: %s" , eventMessage));

//Do something to re-process the message

EventMessage msg= **new** EventMessage(eventMessage.getDescription()+"");

eventMessageService.process(msg);

} **catch** (Exception e) {

**throw** e;

}

}

Output:

INFO 7128 --- [ntainer#2-0-C-1] c.e.s.event.processor.EventProcessor : Recieved Message in DLT: ConsumerRecord(topic = test.DLT, partition = 0, leaderEpoch = 0, offset = 87, CreateTime = 1592911296830, serialized key size = -1, serialized value size = 44, headers = RecordHeaders(headers = [RecordHeader(key = \_\_TypeId\_\_, value = [106, 97, 118, 97, 46, 117, 116, 105, 108, 46, 76, 105, 110, 107, 101, 100, 72, 97, 115, 104, 77, 97, 112]), RecordHeader(key = \_\_ContentTypeId\_\_, value = [106, 97, 118, 97, 46, 108, 97, 110, 103, 46, 79, 98, 106, 101, 99, 116]), RecordHeader(key = \_\_KeyTypeId\_\_, value = [106, 97, 118, 97, 46, 108, 97, 110, 103, 46, 79, 98, 106, 101, 99, 116])], isReadOnly = false), key = null, value = {"id":0,"description":"incompatibleMessage"})

Source code of this demo.