**Создание механизма связи Consumer-а и Producer-а через Kafka:**

[**https://habr.com/ru/articles/738874/**](https://habr.com/ru/articles/738874/), а сам проект лежит на:

**https://github.com/MarMaksk/kafka-cluster**

**Для основного кода программы:**

1. Consumer:

- над классом пишем или @Service или @Component,

- над методом пишем @KafkaListener (topics=\*\*\* - обязательно!!!, и др.параметры id/containerFactory/properties и т.д. по необходимости).

2. Producer:

- над классом пишем или @Service или @Component,

- в методе реализовываем метод “send” класса “KafkaTemplate”, чаще всего заполняя его названием topic-а, ключом, и значением(информацией которую передаем).

**Для тестирования Consumer-а и Producer-а: *(смотри пример в файле “Docker-compose Kafka Zookeeper KafkaUI Tests Pgadmin.doc”)***

1. Для Consumer-а и для Producer-а создаем специальные (для тестов) application.yml.
2. для Consumer-а:

- создаем класс/сы с данными для получаемой сущности,

- над классом пишем @Testcontainers и @SpringBootTest,

- создаем и заполняем начальные параметры контейнеров для БД и Kafka из готовых образов Docker-а, при этом над каждым контейнером пишем @Container,

- объединяем и запускаем созданые контейнеры используя “Startables.deepStart()”,

- заполняем динамические свойства нашего интеграционного теста в отдельном методе над которым пишем @DynamicPropertySource\*,

- делаем автозагрузку нашего интерфейса для работы с БД,

- в самом тестовом методе:

а).иммитируем работу Producer-а, для чего формируем данные для отправки, настраиваем ProducerFactory и оборачиваем KafkaTemplate-ом (*для получения дополнительных методов*) после чего отправляем сформированные данные на Kafka,

б). удостовериваемся в том, что данные отправленные Producer-ом на Kafka получены нашим Consumer-ом, после чего с дополнением доп.полями (если это необходимо) переведены в Entity и после правильно записаны в БД.

3. для Producer -а:

- создаем класс с данными для отправляемой сущности,

- над классом пишем @SpringBootTest, @DirtiesContext и @EmbeddedKafka() с параметрами,

- в самом тестовом методе:

а). создаем и отправляем на Kafka сущность,

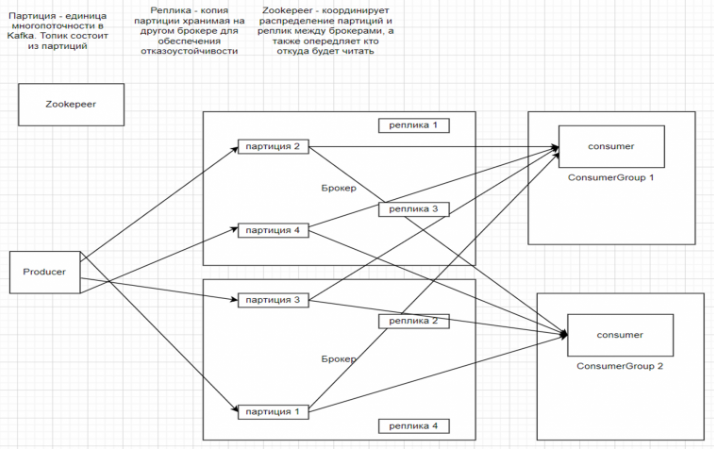
б). создаем Consumer-а, который должен получить от Kafka отправленную нами(Producer-ом) сущность,

в). сверяем то, что отправили с тем, что получил Consumer

**Пример, где Producer будет отправлять в Kafkу сообщения “**"message number " + номер сообщения”**, а 2 Consumer-а будут просто выводить в log.info(), что** "Прочитано сообщение с номером: {} в потоке: {}" **и все это будет запущено в Docker (Docker-compose).**

У нас есть 2 брокера, 1 продюсер, 4 партиции, 2 консюмер группы и 2 консюмера распределённых по консюмер группам.

Продюсер отправляет 10 000 сообщений и для каждого сообщения используя ключ определяет в какую партицию оно попадёт. В нашем случае у нас 4 партиции, которые распределены по двум брокерам. Из‑за наличия двух брокеров мы смогли установить replica‑set равным двум, и теперь у каждой партиции есть копия, которая ждёт пока отвалится мастер, чтобы сменить её. Копия получает все изменения с мастера. Консюмер группы две, в каждой по одному консюмеру, следовательно каждый консюмер получает все сообщения.



**Consumer**

**Кастомный сериалайзер *(***для сообщений отправляемых в DLT очередь(очередь куда будут отправлены сообщения которые небыли доставлены)***)***

package ru.home.kafka.config;

public class DltMessageSerializer<T> implements Serializer<T> {

private final ObjectMapper objectMapper = new ObjectMapper();

@Override

public byte[] serialize(String topic, T data) {

try {

return objectMapper.writeValueAsBytes(data);

} catch (JsonProcessingException e) { throw new SerializationException("Error when serializing to JSON", e); }

}

@Override

public void configure(Map<String, ?> configs, boolean isKey) { // Дополнительная конфигурация не требуется. }

@Override

public void close() { // Дополнительная конфигурация не требуется. }

}

**Конфигурация Kafka**

package ru.home.kafka.config;

@Configuration

@EnableKafka

@RequiredArgsConstructor

public class KafkaConfiguration {

private static final String DLT\_TOPIC\_SUFFIX = ".dlt";

private final ProducerFactory<Object, Object> producerFactory;

private final ConsumerFactory<Object, Object> consumerFactory;

@Bean

public KafkaTemplate<Object, Object> kafkaTemplate() { return new KafkaTemplate<>(producerFactory); }

@Bean

public ConcurrentKafkaListenerContainerFactory<Object, Object> kafkaListenerContainerFactory(

DefaultErrorHandler errorHandler){

// Позволяет создавать консюмеров, которые могут обрабатывать сообщения из нескольких партиций Kafka одновременно, а также настраивать параметры такие как количество потоков, хэндлинг и т.д.

ConcurrentKafkaListenerContainerFactory<Object, Object> kafkaListenerContainerFactory = new ConcurrentKafkaListenerContainerFactory<>();

// Настройка фабрики для создания консьюмера Kafka

kafkaListenerContainerFactory.setConsumerFactory(consumerFactory);

// Возврат сообщений в DLT очередь

kafkaListenerContainerFactory.setCommonErrorHandler(errorHandler);

// Обработка сообщений в 4 потока – Число потоков должно равняться числу подтем(партиций)

kafkaListenerContainerFactory.setConcurrency(4);

return kafkaListenerContainerFactory;

}

//Обработчик исключений при получении сообщений из kafka по умолчанию.

@Bean

public DefaultErrorHandler errorHandler(DeadLetterPublishingRecoverer deadLetterPublishingRecoverer) {

final var handler = new DefaultErrorHandler(deadLetterPublishingRecoverer);

// Обрабатываем любые исключения и отправляем в DLT

handler.addNotRetryableExceptions(Exception.class);

return handler;

}

}

//Публикатор в dead-letter topic. DeadLetterPublishingRecoverer имплементирует интерфейс ConsumerRecordRecoverer, который публикует неудавшуюся запись в теме недоставленного сообщения.

@Bean

public DeadLetterPublishingRecoverer publisher(KafkaTemplate<Object, Object> bytesTemplate) {

// Определяем логику выбора партиции для отправки сообщения в DLT. В данном случае, создаём новый объект TopicPartition, используя имя топика (consumerRecord.topic()) и добавляя суффикс (константа DLT\_TOPIC\_SUFFIX), а также номер партиции (consumerRecord.partition()). Следовательно в DLT топике должно быть столько партиций сколько и в топике откуда читаем.

ConsumerRecord используется для получения записей из кластера Kafka. Состоит из имени темы, номера раздела (из которого принимается запись), смещения (которое указывает на запись в разделе Kafka), и пар <ключ, значение>: public ConsumerRecord(string topic,int partition, long offset,K key, V value)

return new DeadLetterPublishingRecoverer(bytesTemplate, (consumerRecord, exception) ->

new TopicPartition(consumerRecord.topic() + DLT\_TOPIC\_SUFFIX, consumerRecord.partition()));

}

**DTO**

package ru.home.kafka.dto;

@Data

@NoArgsConstructor

@AllArgsConstructor

@Builder

public class JsonMessage {

private long number;

private String message;

}

**Сам Consumer**, где у нас будет 2 получателя, которых мы разнесём в разные консюмер группы используя параметр id у аннотации KafkaListener. Также в случае, если номер сообщения кратный 100, мы будем выбрасывать исключение и отправлять сообщение в DLT очередь**:**

package ru.home.kafka.consumer;

import ru.home.kafka.dto.JsonMessage;

@Component

@RequiredArgsConstructor

@Slf4j

public class KafkaConsumerListeners {

@KafkaListener(

topics = "${kafka.topics.test-topic}", // определяет топик откуда читаем, достаем значение из ‘application.yml’

id = "consumer-group-**1**", // определяет группу консюмера

containerFactory = "kafkaListenerContainerFactory") // ВАЖНО: определяет фабрику, которую мы используем.

Иначе используется фабрика по умолчанию и многопоточность не работает

public void handle(@Payload JsonMessage message) { //@Payload - чтобы Spring понял, что этот аргумент

нужно брать из значения сообщения и

readMessage(message);

}

@KafkaListener(

topics = "${kafka.topics.test-topic}",

id = "consumer-group-**2**",

containerFactory = "kafkaListenerContainerFactory")

public void handle2(@Payload JsonMessage message) { readMessage(message); }

public void readMessage(JsonMessage message) {

long number = message.getNumber();

String currentThreadName = Thread.currentThread().getName();

log.info("Прочитано сообщение с номером: {} в потоке: {}", number, currentThreadName);

if (number % 100 == 0) {

log.info("Сообщение кратно 100");

throw new RuntimeException("Получено сообщение с номером кратным 100");

} } }

**ConsumerApplication.java**

package ru.home.kafka;

@SpringBootApplication

public class ConsumerApplication {

public static void main(String[] args) { SpringApplication.run(ConsumerApplication.class, args); }

}

**application.yml**

spring:

main:

allow-bean-definition-overriding: true

application:

name: kafka-example-consumer

kafka:

bootstrap-servers: host.docker.internal:29092,host.docker.internal:29093 # Адреса всех брокеров кластера

listener:

ack-mode: record # Получение каждой записи должно быть подтверждено

producer:

client-id: ${spring.application.name}

key-serializer: org.apache.kafka.common.serialization.StringSerializer

value-serializer: ru.home.kafka.config.DltMessageSerializer # Очередь для исключений

retries: 3

consumer:

group-id: ${spring.application.name}

autoOffsetReset: earliest

# Сериализаторы для всех типов

keyDeserializer: org.springframework.kafka.support.serializer.ErrorHandlingDeserializer

valueDeserializer: org.springframework.kafka.support.serializer.ErrorHandlingDeserializer

properties:

# Тип, для которого определяем конкретные сериализаторы

spring.json.value.default.type: ru.home.kafka.dto.JsonMessage

spring.deserializer.key.delegate.class: org.apache.kafka.common.serialization.StringDeserializer

spring.deserializer.value.delegate.class: org.springframework.kafka.support.serializer.JsonDeserializer

kafka:

topics:

test-topic: topic1

**Dockerfile**

FROM openjdk:17-jdk-alpine

WORKDIR /app

COPY consumer/build/libs/consumer-0.0.1-SNAPSHOT.jar consumer.jar # Копирование JAR-файла в контейнер

CMD ["java", "-jar", "consumer.jar"] # Команда для запуска JAR-файла

**Producer**

**DTO**

package ru.home.kafka.dto;

@Data

@NoArgsConstructor

@AllArgsConstructor

@Builder

public class JsonMessage {

private long number;

private String message;

}

**Сам Producer**

package ru.home.kafka.producer;

import ru.home.kafka.dto.JsonMessage;

@Component

@RequiredArgsConstructor

@Slf4j

public class KafkaProducer {

@Value("${kafka.topics.test-topic}") // Определяет топик куда пишем, доставая значиние из 'application.yml'

private String topic;

private int messageNumber = 0;

private final KafkaTemplate<Object, Object> kafkaTemplate; // KafkaTemplate – клас-обертка предоставляющий методы для отправки сообщений в темы Kafka

public void sendMessages() {

while (messageNumber != 10\_000) {

messageNumber++;

JsonMessage jsonMessage = JsonMessage.builder()

.number(messageNumber)

.message("message number " + messageNumber)

.build();

kafkaTemplate.send(topic, String.valueOf(ThreadLocalRandom.current().nextLong()), jsonMessage);

log.info("Отправлено сообщение номер {}", messageNumber);

} } }

**ProducerApplication.java**

package ru.home.kafka;

import ru.home.kafka.producer.KafkaProducer;

@SpringBootApplication

public class ProducerApplication {

public static void main(String[] args) {

ConfigurableApplicationContext context = SpringApplication.run(ProducerApplication.class, args);

KafkaProducer producer = context.getBean(KafkaProducer.class);

producer.sendMessages();

}

}

**application.yml**

spring:

kafka:

bootstrap-servers: host.docker.internal:29092,host.docker.internal:29093 # перечисляем всех наших брокеров

producer:

# настраиваем сериализацию сообщений

key-serializer: org.apache.kafka.common.serialization.StringSerializer

value-serializer: org.springframework.kafka.support.serializer.JsonSerializer

kafka:

topics:

test-topic: topic1

**Dockerfile**

FROM openjdk:17-jdk-alpine

WORKDIR /app

COPY producer/build/libs/producer-0.0.1-SNAPSHOT.jar producer.jar

CMD ["java", "-jar", "producer.jar"]

**Docker-compose.yml** запускающий все

version: "3.9"

services:

zookeeper:

image: confluentinc/cp-zookeeper:latest

environment:

ZOOKEEPER\_CLIENT\_PORT: 2181

ZOOKEEPER\_TICK\_TIME: 2000

ports:

- 22181:2181

kafka:

image: confluentinc/cp-kafka:latest

depends\_on:

- zookeeper

ports:

- 29092:29092

hostname: kafka

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092 *(для обращения контейнеров Docker-а, если нужен доступ извне контейнеров Docker-а см.файл D:\Java\Литература\Docker Kafka\ Доступ к Kafka Из или Вне Docker контейнеров.docx)*

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 2

kafka2:

image: confluentinc/cp-kafka:latest

depends\_on:

- zookeeper

ports:

- 29093:29092

hostname: kafka2

environment:

KAFKA\_BROKER\_ID: 2

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka2:29093 *(для обращения контейнеров Docker-а)*

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 2

kafka-ui:

image: provectuslabs/kafka-ui

container\_name: kafka-ui

ports:

- 8090:8080

restart: always

environment:

- KAFKA\_CLUSTERS\_0\_NAME=local

- KAFKA\_CLUSTERS\_0\_BOOTSTRAPSERVERS=kafka:29092,kafka2:29093

- KAFKA\_CLUSTERS\_0\_ZOOKEEPER=zookeeper:2181

links:

- kafka

- kafka2

- zookeeper

producer:

build:

context: .

dockerfile: producer/Dockerfile

ports:

- 8080:8080

links:

- kafka-ui

consumer:

build:

context: .

dockerfile: consumer/Dockerfile

ports:

- 8081:8080

links:

- kafka-ui

