**Tìm hiểu Kafka trong Spring**

1. **Cấu hình producer**

* Các thuộc tính của producer

@Bean  
 public Map<String, Object> producerConfigs() {  
 Map<String, Object> props = new HashMap<>();  
// props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, bootstrapServers);  
// System.out.println("bootstrapServers " + bootstrapServers);  
// props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class);  
// props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, JsonSerializer.class);  
  
 // Địa chỉ của Kafka broker  
 props.put(ProducerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*, bootstrapServers);  
 // Lớp serializer cho key  
 props.put(ProducerConfig.*KEY\_SERIALIZER\_CLASS\_CONFIG*, StringSerializer.class);  
 // Lớp serializer cho value  
 props.put(ProducerConfig.*VALUE\_SERIALIZER\_CLASS\_CONFIG*, JsonSerializer.class);  
 // Kích thước batch gửi dữ liệu (0 = không sử dụng batch)  
 props.put(ProducerConfig.*BATCH\_SIZE\_CONFIG*, 16384);  
 // Thời gian chờ đợi tối đa trước khi gửi batch (ms)  
 props.put(ProducerConfig.*LINGER\_MS\_CONFIG*, 1);  
 // Bộ đệm trước khi gửi dữ liệu tới server (bytes)  
 props.put(ProducerConfig.*BUFFER\_MEMORY\_CONFIG*, 33554432);  
 // Số lần thử gửi lại dữ liệu khi gặp lỗi  
 props.put(ProducerConfig.*RETRIES\_CONFIG*, 10);  
 // Thời gian chờ giữa các lần thử gửi lại (ms)  
 props.put(ProducerConfig.*RETRY\_BACKOFF\_MS\_CONFIG*, 100);  
 // Thời gian tối đa chờ đợi phản hồi từ server (ms)  
 props.put(ProducerConfig.*REQUEST\_TIMEOUT\_MS\_CONFIG*, 30000);  
 // Kích thước tối đa của dữ liệu mà broker có thể nhận (bytes)  
 props.put(ProducerConfig.*MAX\_REQUEST\_SIZE\_CONFIG*, 1048576);  
 // Thời gian chờ đợi tối đa cho các yêu cầu gửi (ms)  
 props.put(ProducerConfig.*DELIVERY\_TIMEOUT\_MS\_CONFIG*, 120000);  
 // Có nên bật cơ chế nén dữ liệu hay không (none, gzip, snappy, lz4, zstd)  
 props.put(ProducerConfig.*COMPRESSION\_TYPE\_CONFIG*, "none");  
 // Đảm bảo an toàn dữ liệu (acks = 0, 1, all)  
 props.put(ProducerConfig.*ACKS\_CONFIG*, "all");  
 return props;  
 }

* Cấu hình Producer :
* @Bean  
  public ProducerFactory<String, Object> producerFactory() {  
   return new DefaultKafkaProducerFactory<>(producerConfigs());  
  }  
    
  @Bean  
  public KafkaTemplate<String, Object> retryableTopicKafkaTemplate() {  
   return new KafkaTemplate<>(producerFactory());  
  }  
    
  @Bean  
  public KafkaTemplate<String, Object> kafkaTemplate() {  
   return new KafkaTemplate<>(producerFactory());  
  }

1. **Cấu hình Consumer**

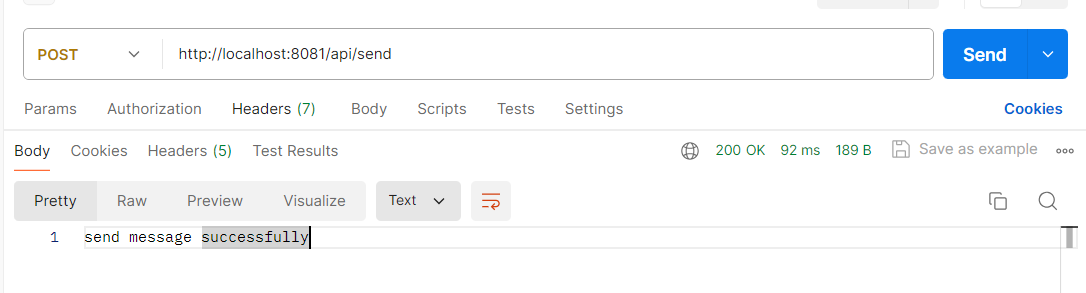
* Các thuộc tính của consumer
* @Bean  
   public Map<String, Object> consumerConfigs() {  
   Map<String, Object> props = new HashMap<>();  
  // props.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, bootstrapServers);  
  // props.put(ConsumerConfig.GROUP\_ID\_CONFIG, groupId);  
  // props.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class);  
  // props.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, JsonDeserializer.class);  
  // props.put(JsonDeserializer.TRUSTED\_PACKAGES, "\*");  
    
    
   // Basic configurations  
   // Địa chỉ của Kafka broker  
   props.put(ConsumerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*, bootstrapServers);  
   // ID của nhóm consumer  
   props.put(ConsumerConfig.*GROUP\_ID\_CONFIG*, groupId);  
   // Lớp deserializer cho key  
   props.put(ConsumerConfig.*KEY\_DESERIALIZER\_CLASS\_CONFIG*, StringDeserializer.class);  
   // Tin tưởng tất cả các package để deserialization  
   props.put(JsonDeserializer.*TRUSTED\_PACKAGES*, "\*");  
   // Optional configurations  
   // Xác định nơi bắt đầu tiêu thụ khi không có offset hiện tại (earliest, latest, none)  
   props.put(ConsumerConfig.*AUTO\_OFFSET\_RESET\_CONFIG*, "earliest");  
   // Tự động commit offset hay không (false = không tự động commit)  
   props.put(ConsumerConfig.*ENABLE\_AUTO\_COMMIT\_CONFIG*, false);  
   // Khoảng thời gian commit offset nếu ENABLE\_AUTO\_COMMIT\_CONFIG là true  
   props.put(ConsumerConfig.*AUTO\_COMMIT\_INTERVAL\_MS\_CONFIG*, "1000");  
   // Thời gian timeout của phiên  
   props.put(ConsumerConfig.*SESSION\_TIMEOUT\_MS\_CONFIG*, "10000");  
   // Khoảng thời gian giữa các heartbeat gửi tới Kafka broker  
   props.put(ConsumerConfig.*HEARTBEAT\_INTERVAL\_MS\_CONFIG*, "3000");  
   // Số lượng bản ghi tối đa mà consumer có thể lấy trong một lần poll  
   props.put(ConsumerConfig.*MAX\_POLL\_RECORDS\_CONFIG*, "500");  
   // Thời gian tối đa giữa các lần poll  
   props.put(ConsumerConfig.*MAX\_POLL\_INTERVAL\_MS\_CONFIG*, "300000");  
   // Kích thước tối thiểu của dữ liệu mà consumer sẽ cố gắng lấy trong một lần fetch  
   props.put(ConsumerConfig.*FETCH\_MIN\_BYTES\_CONFIG*, "1");  
   // Thời gian tối đa mà broker sẽ chờ đợi dữ liệu để gửi về consumer nếu chưa đạt được kích thước tối thiểu  
   props.put(ConsumerConfig.*FETCH\_MAX\_WAIT\_MS\_CONFIG*, "500");  
   // ID của client consumer  
   props.put(ConsumerConfig.*CLIENT\_ID\_CONFIG*, "consumer-client-id");  
   // Thời gian chờ giữa các lần thử kết nối lại  
   props.put(ConsumerConfig.*RECONNECT\_BACKOFF\_MS\_CONFIG*, "50");  
   // Thời gian chờ trước khi retry sau một lỗi  
   props.put(ConsumerConfig.*RETRY\_BACKOFF\_MS\_CONFIG*, "100");  
   // Thời gian tối đa mà metadata được cache trước khi refresh  
   props.put(ConsumerConfig.*METADATA\_MAX\_AGE\_CONFIG*, "300000");  
   // Thời gian tối đa mà consumer sẽ chờ cho một request tới broker  
   props.put(ConsumerConfig.*REQUEST\_TIMEOUT\_MS\_CONFIG*, "40000");  
   // Kích thước tối đa của dữ liệu mà consumer có thể fetch từ mỗi phân vùng  
   props.put(ConsumerConfig.*MAX\_PARTITION\_FETCH\_BYTES\_CONFIG*, "1048576");  
   // Kích thước buffer nhận dữ liệu của socket  
   props.put(ConsumerConfig.*RECEIVE\_BUFFER\_CONFIG*, "65536");  
   // Kích thước buffer gửi dữ liệu của socket  
   props.put(ConsumerConfig.*SEND\_BUFFER\_CONFIG*, "131072");  
   // Thời gian tối đa mà một kết nối có thể không hoạt động trước khi đóng  
   props.put(ConsumerConfig.*CONNECTIONS\_MAX\_IDLE\_MS\_CONFIG*, "540000");  
   // Có bỏ qua các topic nội bộ hay không  
   props.put(ConsumerConfig.*EXCLUDE\_INTERNAL\_TOPICS\_CONFIG*, "true");  
   // Xác định mức độ cách ly của transaction (read\_uncommitted, read\_committed)  
   props.put(ConsumerConfig.*ISOLATION\_LEVEL\_CONFIG*, "read\_uncommitted");  
   return props;  
   }
* Cấu hình consumer:
* @Bean  
  public ConsumerFactory<String, Object> consumerFactory() {  
   return new DefaultKafkaConsumerFactory<>(consumerConfigs());  
  }  
    
  @Bean  
  public ConcurrentKafkaListenerContainerFactory<String, Object> kafkaListenerContainerFactory() {  
   ConcurrentKafkaListenerContainerFactory<String, Object> factory =  
   new ConcurrentKafkaListenerContainerFactory<>();  
   factory.setConsumerFactory(consumerFactory());  
   return factory;  
  }

**Ví dụ:**

* Gửi message đến topic: hnheoxtopic
* @PostMapping("/send")  
  public ResponseEntity<String> placeOrder(){  
   kafkaService.sendMessage();;  
   return ResponseEntity.*ok*("send message successfully") ;  
  }
* public void sendMessage() {  
   kafkaTemplate.send("hnheoxtopic", "test"  
   );  
  }
* Consumer nhận và xử lý message
* @KafkaListener(topics = "hnheoxtopic", groupId = "hnheoxid")  
  public void handleNotification(String message){  
   System.*out*.println("receive message : " + message);  
   try {  
   Thread.*sleep*(10000);  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   System.*out*.println("10s receive message : " + message);  
    
  }

**Kết quả nhận được:**

Postman:



Log:



1. **Dead Letter Queue**

Dead Letter Queue (DLQ) là một cơ chế trong các hệ thống message queuing, dùng để xử lý các thông điệp không thể được xử lý đúng cách bởi consumer. Khi một thông điệp không thể được tiêu thụ thành công sau một số lần thử nhất định hoặc gặp phải lỗi không thể khắc phục, nó sẽ được chuyển đến DLQ để đảm bảo rằng hệ thống không bị ngừng hoạt động hoặc gặp phải các vấn đề liên quan đến việc xử lý các thông điệp lỗi

Ví dụ:

* Payment Model :

public class Payment {  
 private String reference;  
 private BigDecimal amount;  
 private Currency currency;  
}

* Gửi message đến topic payment

public void pay() {  
 kafkaTemplate.send("payments", Payment.*createPayment*("heox")  
 );  
}

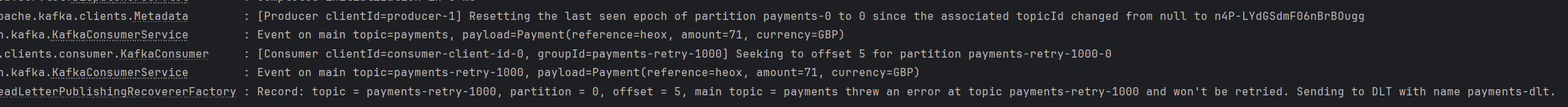
* Consumer nhận và xử lý message và luôn trả ra lỗi

@RetryableTopic(attempts = "2", kafkaTemplate = "retryableTopicKafkaTemplate")  
@KafkaListener(topics = { "payments" }, groupId = "payments")  
public void handlePayment(  
 Payment payment, @Header(KafkaHeaders.*RECEIVED\_TOPIC*) String topic) {  
 *log*.info("Event on main topic={}, payload={}", topic, payment);  
 throw new RuntimeException("Simulated error");  
}

* Cấu hình Dead Letter Topic

@DltHandler  
public void handleDltPayment(  
 Payment payment, @Header(KafkaHeaders.*RECEIVED\_TOPIC*) String topic) {  
 *log*.info("Event on dlt topic={}, payload={}", topic, payment);  
}

**Kết quả** : khỉ producer gửi message và consumer nhận và xử lý sau khi retry 2 lần và vẫn xảy ra lỗi thì consumer đã nhảy đến Dead letter topic



**Log:**

2024-08-07 15:29:13.209 INFO 12980 --- [ntainer#1-0-C-1] com.hn.kafka.KafkaConsumerService : Event on main topic=payments, payload=Payment(reference=heox, amount=71, currency=GBP)

2024-08-07 15:29:13.728 INFO 12980 --- [etry-1000-0-C-1] o.a.k.clients.consumer.KafkaConsumer : [Consumer clientId=consumer-client-id-0, groupId=payments-retry-1000] Seeking to offset 5 for partition payments-retry-1000-0

2024-08-07 15:29:14.273 INFO 12980 --- [etry-1000-0-C-1] com.hn.kafka.KafkaConsumerService : Event on main topic=payments-retry-1000, payload=Payment(reference=heox, amount=71, currency=GBP)

2024-08-07 15:29:14.274 ERROR 12980 --- [etry-1000-0-C-1] k.r.DeadLetterPublishingRecovererFactory : Record: topic = payments-retry-1000, partition = 0, offset = 5, main topic = payments threw an error at topic payments-retry-1000 and won't be retried. Sending to DLT with name payments-dlt.

Và ở **Dead Letter Topic** đã log ra :” Event on **dlt** topic=payments-dlt, payload=Payment(reference=heox, amount=71, currency=GBP)”

