

역할 부여

당신은 전문적인 기술 블로그 작성가입니다. 본인이 했던 기술을 기록용으로도 작성하지만 이 기술이 필요로 한 다른 사람들에게 효과적으로 전달해야 하기도 합니다.

작성 가이드

- 사용자 입력 데이터를 참고하여 새로운 글을 생성하세요.
- 제목(`title`)은 사용자가 제공한 값을 그대로 사용하세요.
- 내용(`content`)은 사용자가 입력한 `memo`를 바탕으로 500자 이상 자연스럽게 확장하세요.
- `memo`에 해당하는 명령어나 코드가 있으면 바로 아래에 추가하세요.
- 기본 명령어와 함께 응용하는 예시도 같이 추가하세요.
- 이전 글(`latest_posts`)을 참고하여 스타일과 일관성을 유지하세요.
- ****반드시 Markdown 형식으로 출력하세요.****
- 문장은 간결하고, 독자가 쉽게 이해할 수 있도록 작성하세요.

예제

입력 데이터:

```
{
  "title": "인공지능의 미래",
  "memo": "AI가 미래 산업에 미치는 영향",
  "latest_posts": [
    { "title": "데이터 동기화를 위한 KAFKA활용(1)", "link":
      "https://velog.io/@inssg/KAFKA1", "content": "<h2 id=W\"kafka-brokerW\">Kafka
      Broker</h2>Wn<ul>Wn <li>실행된 Kafka 애플리케이션 서버 개념</li>Wn <li>3대 이상
      의 Broker Cluster 구성</li>Wn</ul>Wn<h2
      id=W\"zookeeperW\">Zookeeper</h2>Wn<ul>Wn <li>이러한 브로커들을 컨트롤해주는 역할Wn
      <ul>Wn <li>즉, 클라이언트가 서로 공유하는 데이터를 관리해주는 역할</li>Wn
      </ul></li>Wn <li>분산되어 있는 각 애플리케이션의 정보를 중앙에 집중하고 구성관리,
```

그룹관리 네이밍, 동기화 등을 제공

- Kafka와 데이터를 주고 받기 위해 사용하는 자바 라이브러리
- Producer, Consumer, Admin, Stream 등 각종 API 제공
- 다양한 Third Party Library 존재

Kafka 실행해보기

Zookeeper를 먼저 실행한 다음, Kafka 서버를 기동하자

```
# window 기준, zookeeper 실행
```

```
$KAFKA_HOME/bin/windows/zookeeper-server-start.bat
```

```
$KAFKA_HOME/config/zookeeper.properties
```

```
# kafka 서버 구동
```

```
$KAFKA_HOME/bin/windows/kafka-server-start.bat
```

```
$KAFKA_HOME/config/server.properties
```

topic 생성, 목록

확인, 정보확인

```
# 생성 / kafka서버는 기본적으로 9092
```

포트를 사용하며 토픽의 파티션은 1개로 가정

```
$KAFKA_HOME/bin/windows/kafka-
```

```
topics.bat --create --topic 토픽명 --bootstrap-server localhost:9092 --partitions 1
```

```
목록 확인$KAFKA_HOME/bin/windows/kafka-topics.bat --bootstrap-server localhost:9092
```

```
--list
```

```
# 정보 확인$KAFKA_HOME/bin/windows/kafka-topics.bat --describe --topic 토픽명 --bootstrap-server localhost:9092
```

메세지를 생산하

는 producer, 소비하는 consumer 기동하기

```
# 생산
```

```
$KAFKA_HOME/bin/windows/kafka-console-producer.bat --broker-list localhost:9092 --
```

```
topic 앞에서설정한토픽명
```

```
# 소비, from-beginning 해당 토픽의 처음부터 메세지를
```

```
받아오기 위한 옵션$KAFKA_HOME/bin/windows/kafka-console-consumer.bat --
```

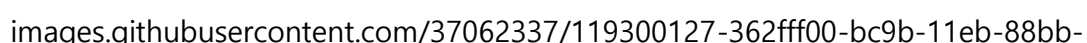
```
bootstrap-server localhost:9092 --topic 앞에서설정한토픽명 --from-
```

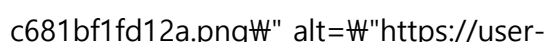
```
beginning
```

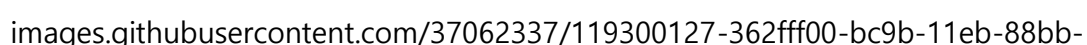
Kafka Connect

- Kafka Connect를 통해 Data를 import/export 가능하다
- 코드 없이

- Configuration으로 데이터를 이동
- Restful api를 통해 지원
- Stream
- or Batch 형태로 데이터 전송 가능
- 커스텀 Connector를 통한 다양한 plugin
- 제공 (DB를 적용해 볼 예정)









- Kafka Connect Source : 특정 리소스에서 데이

- 터를 가져와 카프카 클러스터에 가져오는걸 개입한다.(import)
- Kafka Connect
- Sink : Cluster에 저장되어 있는 데이터를 다른 쪽으로 보내는데 개입한
- 다.(export)

Kafka Connect 실행해보기

Kafka Connect 실행

```
$KAFKA_CONNECT_HOME/bin/windows/connect-
```

distributed.bat ./etc/kafka/connect-distributed.properties

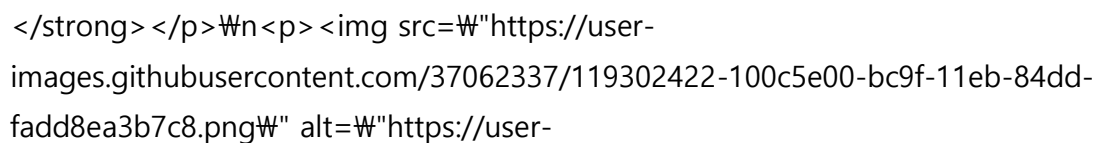
설정 정보 수정

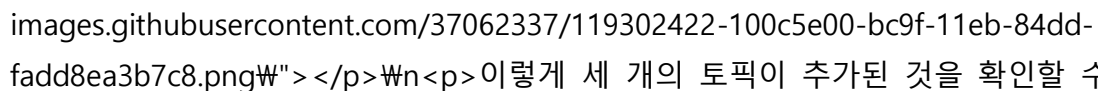
삽입

```
# rem classpath addition for core 위에  
rem classpath addition for LSB style path  
if exist %BASE_DIR%\share\java\kafka (n  
call:concat %BASE_DIR%\share\java\kafka  
</pre>  
<pre>  
<code>  
# mariadb 연동을 위해 jdbc 추가# 위치 : \etc\kafka\connect-
```

distributed.propertiesplugin.path=[confluentinc-kafka-connect-jdbc-10.0.1 폴더]
이후 kafka connect jdbc maridb 드라이버 파일을 confluent-6.1.0/share/java/kafka 에 복사

connect 연결 이후 토픽을 확인하면





이렇게 세 개의 토픽이 추가된 것을 확인할 수 있다.

커넥트가 소스에서 읽어왔던 데이터를 저장하고 관리하기 위한 토픽들이

다.

Soure Connect 추가

```
# connect의  
기본 포트는 8083  
# 커넥트 추가, POST로 전송  
{  
  "name" : "my-source-  
connect",  
  "config" : {  
    "connector.class" :
```

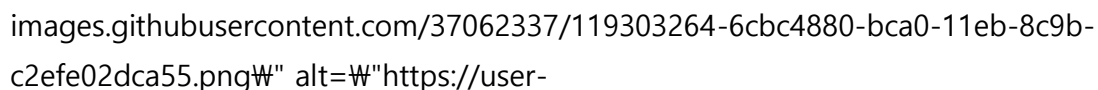
```
"io.confluent.connect.jdbc.JdbcSourceConnector",  
  "connection.url" : "jdbc:mysql://localhost:3306/mydb",  
  "connection.user" : "root",  
  "connection.password" : "test1357",  
  "mode" : "incrementing",  
  "incrementing.column.name" : "id",  
  "table.whitelist" : "users", # 감지할 테이블명  
  "topic.prefix" : "my_topic_", #  
  저장될 토픽의 prefix 최종으로 만들어진 토픽 : my_topic_users  
  "tasks.max" :  
  "1"  
}
```

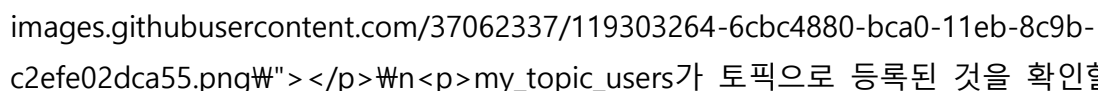
```
"io.confluent.connect.jdbc.JdbcSourceConnector",  
  "connection.url" : "jdbc:mysql://localhost:3306/mydb",  
  "connection.user" : "root",  
  "connection.password" : "test1357",  
  "mode" : "incrementing",  
  "incrementing.column.name" : "id",  
  "table.whitelist" : "users", # 감지할 테이블명  
  "topic.prefix" : "my_topic_", #  
  저장될 토픽의 prefix 최종으로 만들어진 토픽 : my_topic_users  
  "tasks.max" :  
  "1"  
}
```

```
"io.confluent.connect.jdbc.JdbcSourceConnector",  
  "connection.url" : "jdbc:mysql://localhost:3306/mydb",  
  "connection.user" : "root",  
  "connection.password" : "test1357",  
  "mode" : "incrementing",  
  "incrementing.column.name" : "id",  
  "table.whitelist" : "users", # 감지할 테이블명  
  "topic.prefix" : "my_topic_", #  
  저장될 토픽의 prefix 최종으로 만들어진 토픽 : my_topic_users  
  "tasks.max" :  
  "1"  
}
```

```
"io.confluent.connect.jdbc.JdbcSourceConnector",  
  "connection.url" : "jdbc:mysql://localhost:3306/mydb",  
  "connection.user" : "root",  
  "connection.password" : "test1357",  
  "mode" : "incrementing",  
  "incrementing.column.name" : "id",  
  "table.whitelist" : "users", # 감지할 테이블명  
  "topic.prefix" : "my_topic_", #  
  저장될 토픽의 prefix 최종으로 만들어진 토픽 : my_topic_users  
  "tasks.max" :  
  "1"  
}
```

```
"io.confluent.connect.jdbc.JdbcSourceConnector",  
  "connection.url" : "jdbc:mysql://localhost:3306/mydb",  
  "connection.user" : "root",  
  "connection.password" : "test1357",  
  "mode" : "incrementing",  
  "incrementing.column.name" : "id",  
  "table.whitelist" : "users", # 감지할 테이블명  
  "topic.prefix" : "my_topic_", #  
  저장될 토픽의 prefix 최종으로 만들어진 토픽 : my_topic_users  
  "tasks.max" :  
  "1"  
}
```





my_topic_users가 토픽으로 등록된 것을 확인할 수 있다. 이후, 마리아 db에 users 테이블을 만들어 데이터를 삽입 후 확인해보면

```
{  
  "schema" : {  
    "type" : "struct",  
    "fields" : [  
      {  
        "type" : "int32",  
        "optional" : false,  
        "field" : "id"  
      },  
      {  
        "type" : "string",  
        "optional" : true,  
        "field" : "user_id"  
      },  
      {  
        "type" : "string",  
        "optional" : true,  
        "field" : "pwd"  
      },  
      {  
        "type" : "string",  
        "optional" : true,  
        "field" : "name"  
      }  
    ]  
  }  
}
```

```

W"typeW": W"int64W",Wn W"optionalW": true,Wn W"nameW":
W"org.apache.kafka.connect.data.TimestampW",Wn W"versionW": 1,Wn W"fieldW":
W"created_atW" }Wn ],Wn W"optionalW": false,Wn W"nameW": W"usersW" }Wn },Wn
W"payloadW": {Wn W"idW": 1,Wn W"user_idW": W"helW",Wn W"pwdW": W"1234W",Wn
W"nameW": W"koW",Wn W"created_atW":
1621788132000Wn }Wn }</code></pre>Wn<p>Schema에 fields는 각 컬럼 정보들이 저장
되어 있고, payload에 컬럼별 저장된 데이터 값들이 있다.</p>Wn<p><strong>Sink
Connect 추가</strong></p>Wn<pre><code>{Wn W"nameW":W"my-sink-connectW",Wn
W"configW": {Wn
W"connector.classW":W"io.confluent.connect.jdbc.JdbcSinkConnectorW",Wn
W"connection.urlW":W"jdbc:mysql://localhost:3306/mydbW",Wn
W"connection.userW":W"rootW",Wn W"connection.passwordW":W"test1357W",Wn
W"auto.createW":W"trueW", # DB를 자동으로 만들기 설정, 토픽과 같은 이름의 테이블 생
성Wn W"auto.evolveW":W"trueW",Wn W"delete.enabledW":W"falseW",Wn
W"tasks.maxW":W"1W",Wn W"topicsW":W"my_topic_usersW" # 정보를 받을 토픽
Wn }Wn}</code></pre>Wn<p>본 내용은 이도원님의<a
href=W"https://www.infllearn.com/course/%EC%8A%A4%ED%94%84%EB%A7%81-%ED%8
1%B4%EB%9D%BC%EC%9A%B0%EB%93%9C-%EB%A7%88%EC%9D%B4%ED%81%AC%EB
%A1%9C%EC%84%9C%EB%B9%84%EC%8A%A4/dashboardW">Spring Cloud로 개발하는
마이크로서비스 애플리케이션</a>을 수강하고 정리한 내용입니다.</p>,"
"publishedDate": "2023-03-23T04:52:12.000+00:00" },

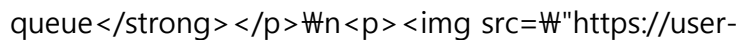
```

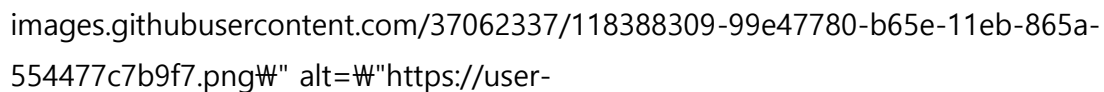
```

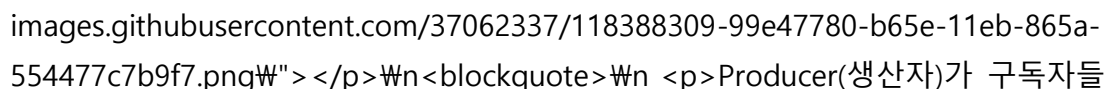
{ "title": "MSA with SPRING CLOUD(5) WnSpring CloudBus & 설정정보 암호화",
"link": "https://velog.io/@inssg/SPRING-CLOUD-2", "content": "<h1
id=W"W"></h1>Wn<h2 id=W"기존의-spring-cloud-config가-변경되었을-경우-해당-마이크
로-서비스에-적용하는-방법W">기존의 Spring Cloud Config가 변경되었을 경우 해당 마이
크로 서비스에 적용하는 방법</h2>Wn<p><strong>각 마이크로서비스 서버와 Config
서버를 재기동하는 방법</strong></p>Wn<blockquote>Wn <p>서버를 재기동 하는 것
자체가 굉장히 불편하다.</p>Wn</blockquote>Wn<p><strong>Actuator를 활용하여
POST refresh를 요청해 애플리케이션을 재기동 하지 않고 적용하는 방법
</strong></p>Wn<blockquote>Wn <p>http로 call하여 변경을 감지해 적용하는 것은 좋
으나, 마이크로서비스가 많을 경우 이것도 불편하다. 각각의 마이크로서비스 마다
refresh를 호출해야 하기 때문이다.</p>Wn</blockquote>Wn<h2 id=W"spring-cloud-
busW">Spring Cloud Bus</h2>Wn<blockquote>Wn <p>분산 시스템의 노드(마이크로서
비스)를 경량 메세지 브로커와 연결</p>Wn <p>각각의 시스템 상태 및 구성에 대한 변
경사항을 연결된 노드(마이크로서비스)에게 전달</p>Wn <p>순수 Actuator만 적용하면

```

각 마이크로서비스 마다 refresh를 호출해야 했지만, Spring Cloud Bus를 사용할 경우, Spring Cloud Bus와 연결된 마이크로서비스 아무 장소에서나 busrefresh 요청을 할 경우, 다른 서비스들에게 전파된다.







Producer(생산자)가 구독자들에게 전달할 Message를 Queue에 넣어두면, Consumer가 Message를 처리하는 방식. 이 과정들은 비동기 방식이다!

이 때 메세지의 형태는 단순 String 뿐만 아니라, 객체, 리소스등 다양한 형태일 수 있다.

중간에 Queue라는 미들웨어를 둬으로써 연결된 상대를 개의치 않고 효율적으로 메세지를 보낼 수 있

다.

Cloud Config Server는 각 마이크로서비스에 연결되어 있을 것이다.

설정이 변경되었을 경우 연결된 마이크로서비스 마다 Spring Cloud Bus가 직접 변경사항의 Message를 Push하는 기능을 담당한

다.

AMQP (Advanced Message Queuing Protocol)

메세지 지향 미들웨어를 위한 개방형 표준 응용 계층 프로토콜

ex) Erlang, RabbitMQ

메세지 지향, 큐잉, 라우팅(P2P, Pub / Sub 구조), 신뢰성, 보안

Kafka

Scalar 언어로 개발한 오픈 소스 메세지 브로커 프로젝트

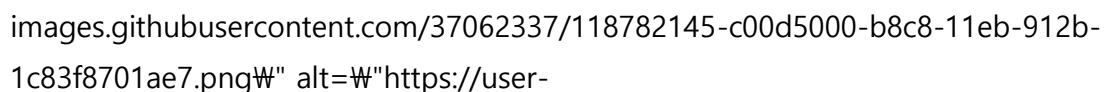
분산형 스트리밍 플랫폼

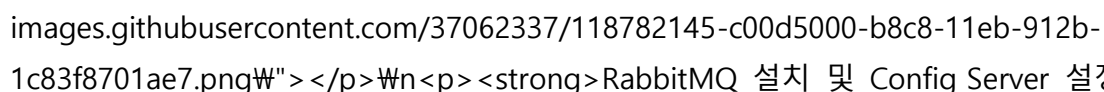
대용량의 데이터를 처리 가능한 메세징 시스템

RabbitMQ를 통한 설정정보 갱신하기-with-java-keytool

먼저, 지금까지 강의를 들으면서 Message Queue까지 적용된 프로젝트 구조는 다음과 같

다.





RabbitMQ 설치 및 Config Server 설정

```
# Config Serverserver:
port: {Config Server 포트 별도 지정}
spring:
  rabbitmq:
    host: {RabbitMQ의 호스트}
    port: {RabbitMQ의 포트 default 5672}
    username: guest# defaultpassword: guest# defaultapplication:
name: config-server
cloud:
  config:
    server:
      git:
        uri: {설정들을 저장할 git 주소}
```

```

username: {private 저장소의 경우 git username}\n password: {private 저장소의 경우 git
password}\n\n# Actuator 설정management:\n endpoints:\n web:\n exposure:\n
include: health, busrefresh</code></pre>\n<p><strong>Java Keytool을 활용한
bootstrap.yml 설정</strong></p>\n<pre><code># Config Serverencrypt:\n key-
store:\n location: {Key 저장 위치}\n password: {Key의 비밀번호}\n alias: {Key를 만들때
제작한 alias}</code></pre>\n<p><strong>Config Server에서 [POST] /encrypt, /decrypt
를 이용해 암호화된 값, 복호화된 값을 알 수 있다.</strong></p>\n<p><strong>Git
Repository에서 관리되는 설정 정보들 (내거 기준)</strong></p>\n<p><img
src=W"https://user-images.githubusercontent.com/37062337/118783941-83daef00-b8ca-
11eb-81c2-37c6858c2a3d.pngW" alt=W"https://user-
images.githubusercontent.com/37062337/118783941-83daef00-b8ca-11eb-81c2-
37c6858c2a3d.pngW"></p>\n<p>이 중에서 user-service.yml 을 보면</p>\n<p><img
src=W"https://user-images.githubusercontent.com/37062337/118784149-b8e74180-b8ca-
11eb-8e8a-7a9281c1d8fe.pngW" alt=W"https://user-
images.githubusercontent.com/37062337/118784149-b8e74180-b8ca-11eb-8e8a-
7a9281c1d8fe.pngW"></p>\n<p>Java Keytool을 활용해 암호화된 값들을 Config Git
Repository에 기입하여 반영한 다음 push를 하게 되면 RabbitMQ에 연결된 모든 마이크
로서비스들의 설정 정보를 전파하게 되고, Keytool로 암호화 된 것이 [GET] config-server
host/user-service/default로 접속하면 암호화된 값이 변경된 값으로 복호화해서 보여준
다.</p> ", "publishedDate": "2023-03-23T04:51:37.000+00:00" }

]

}

```

기대하는 출력 형식:

```

{
  "title": "데이터 동기화를 위한 KAFKA 활용(2)",
  "content": "이번에 적용해볼 것들
  • Order Service에 요청된 주문의 수량 정보를 Catalog Service에 반영
  • Order Service에서 Kafka Topic으로 메세지 전송 → Producer
  • Catalog Service에서 Kafka Topic에 전송된 메세지 취득 → Consumer

```

kafka 의존성 추가

```
<dependency>

    <groupId>org.springframework.kafka</groupId>

    <artifactId>spring-kafka</artifactId>

</dependency>
```

Catalog Service

@EnableKafka

@Configuration

```
public class KafkaConsumerConfig {
```

```
    // Consumer 빈 설정 및 등록
```

```
    @Bean
```

```
    public ConsumerFactory<String, String> consumerFactory() {
```

```
        Map<String, Object> properties = new HashMap<>();
```

```
        // kafka 서버의 host, port, 컨슈머는 데이터를 받아오기 때문에 역직렬화 설정
```

```
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
```

```
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "consumerGroupId");
```

```
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
```

```
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
```

```
        return new DefaultKafkaConsumerFactory<>(properties);
```

```
    }
```

```

// 카프카에 들어온 정보를 감지하기 위해 리스너 빈 등록

@Bean

public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory() {

    ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory

        = new ConcurrentKafkaListenerContainerFactory<>();

    kafkaListenerContainerFactory.setConsumerFactory(consumerFactory());

    return kafkaListenerContainerFactory;

}
}

```

catalog service 수정

```

@Service

@RequiredArgsConstructor

@Slf4j

@Transactional

public class KafkaConsumer {

    private final CatalogRepository catalogRepository;

    private final String kafkaTopic = "example-catalog-topic";

    // Listen할 토픽 설정

    @KafkaListener(topics = kafkaTopic)

    public void updateQuantity(String kafkaMessage) {

        log.info("kafka Message = " + kafkaMessage);
    }
}

```



```

// 역직렬화
Map<Object, Object> map = new HashMap<>();

ObjectMapper objectMapper = new ObjectMapper();

try {

    map = objectMapper.readValue(kafkaMessage, new
TypeReference<Map<Object, Object>>() {});

    } catch(JsonProcessingException e) {

        e.printStackTrace();

    }

    CatalogEntity entity = catalogRepository.findByProductId((String)
map.get("productId"));

    // 상품이 존재할 경우 상품의 수량 수정
    if (entity != null) {

        entity.setStock(entity.getStock() - (Integer) map.get("quantity"));

    }

}
}

```

Order Service

@Configuration

@EnableKafka

public class KafkaProducerConfig {

// kafka로 메시지를 보내야하기 때문에 직렬화 해주어야함

@Bean

```

public ProducerFactory<String, String> producerFactory() {
    Map<String, Object> properties = new HashMap<>();
    properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
    properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
    properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
    return new DefaultKafkaProducerFactory<>(properties);
}

// 데이터 전달 인스턴스
@Bean
public KafkaTemplate<String, String> kafkaTemplate() {
    return new KafkaTemplate<>(producerFactory());
}
}

```

Order Controller 수정

```

@PostMapping("/{userId}/orders")
public ResponseEntity<ResponseOrder> createOrder(@PathVariable("userId") String
userId,
                                                    @RequestBody RequestOrder
request)
                                                    throws JsonProcessingException {

    ModelMapper modelMapper = new ModelMapper();
    modelMapper.getConfiguration().setMatchingStrategy(MatchingStrategies.STRICT);
}

```

```
OrderDto orderDto = modelMapper.map(request, OrderDto.class);

orderDto.setUserId(userId);

OrderDto createdOrder = orderService.createOrder(orderDto);

// kafkaTopic에 주문 데이터 전달
kafkaProducer.send(kafkaTopic, orderDto);

ResponseOrder responseOrder = modelMapper.map(orderDto, ResponseOrder.class);
return ResponseEntity.status(HttpStatus.CREATED)
    .body(modelMapper.map(createdOrder, ResponseOrder.class));
}
```

Order Service에서 주문해보기

먼저 상품의 재고 확인

[GET] /catalog-service/catalogs

```
{
  {
    "productId": "CATALOG-001",
    "productName": "Berlin",
    "unitPrice": 1500,
    "stock": 99,
    "createdAt": "2021-05-24T16:19:54.62"
  },
  ...
}
```

```
}
```

주문

```
# [POST] /order-service/{userId}/orders
```

Request

```
{
```

```
  "productId" : "CATALOG-001",
```

```
  "quantity" : 95,
```

```
  "unitPrice" : 2000
```

```
}
```

Response

```
{
```

```
  "productId": "CATALOG-001",
```

```
  "quantity": 95,
```

```
  "unitPrice": 2000,
```

```
  "totalPrice": 190000,
```

```
  "orderId": "b0b2d4be-e484-44c8-a42d-258bc79ec1bc"
```

```
}
```

다시 상품의 재고 확인

```
# [GET] /catalog-service/catalogs
```

```
{
```

```
{
```

```
  "productId": "CATALOG-001",
```

```
  "productName": "Berlin",
```

```
  "unitPrice": 1500,
```

```
  "stock": 4,
```

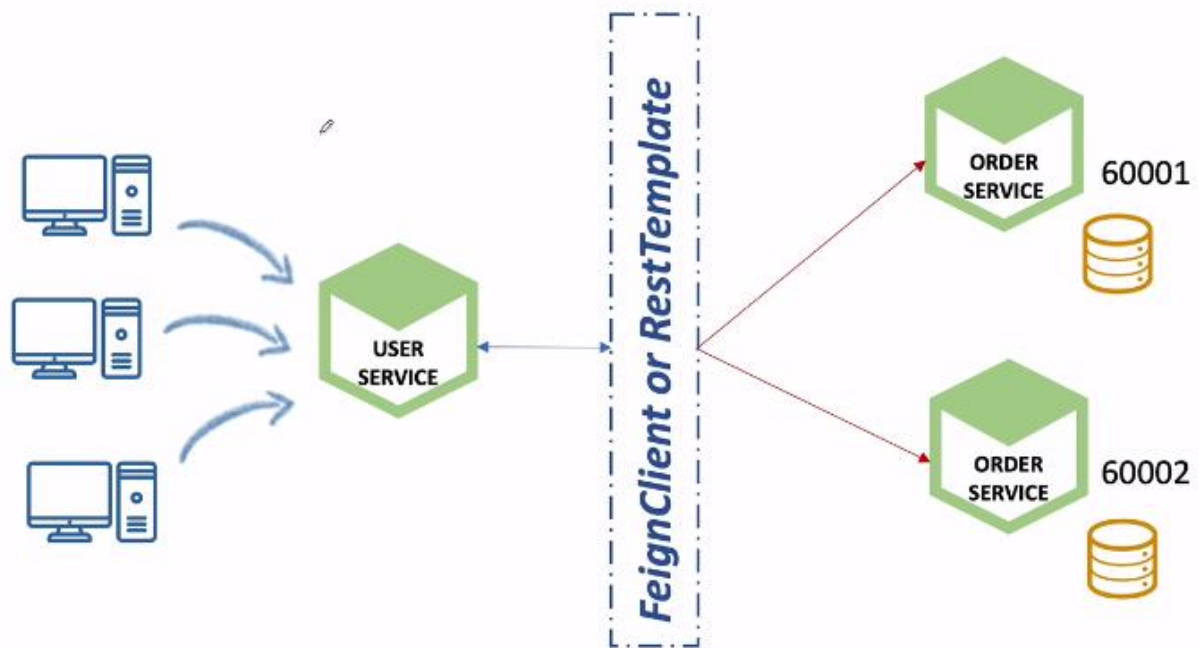
```
"createdAt": "2021-05-24T16:19:54.62"
},
...
}
```

그리고 내 정보에서 주문 확인

```
{
  "email": "kobumssh@naver.com",
  "name": "고범석",
  "userId": "b169e54c-c563-449a-9fe4-2b77fa4df2fd",
  "orders": [
    {
      "productId": "CATALOG-001",
      "quantity": 95,
      "unitPrice": 2000,
      "totalPrice": 190000,
      "createdAt": "2021-05-24T16:39:24",
      "orderId": "b0b2d4be-e484-44c8-a42d-258bc79ec1bc"
    }
  ]
}
```

Order Service Instance가 복수일 경우 동기화 문제를 해결해보기

만약 OrderService 인스턴스를 하나 더 띄우게 되면 프로젝트에는 각각의 인스턴스에 H2 내장 DB가 붙는다. 그리고 라운드 로빈 방식으로 각 서비스가 호출되기 때문에 주문을 여러번 하고 내 주문 조회를 하게 되면 조회를 할 때 마다 결과가 다르게 나온다.



이 문제를 해결하기 위해 Order Service에 요청된 주문 정보를 DB가 아니라 Kafka의 Topic으로 전송하고, Topic에 설정된 Kafka Sink Connect를 사용해 단일 DB(Maria DB)에 저장해보자

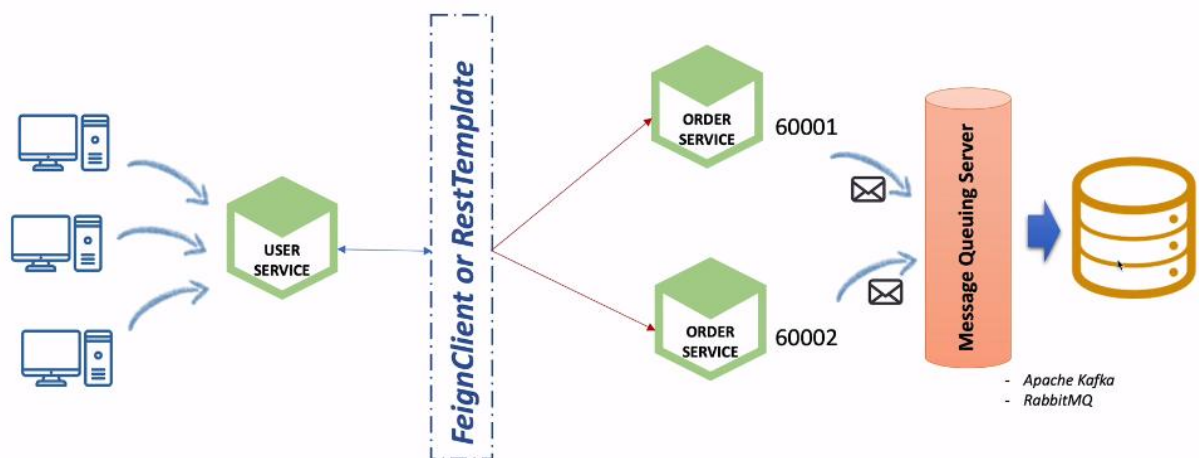


Table 생성

```
CREATE TABLE orders (
  id int auto_increment primary key,
  product_id varchar(20) not null,
  quantity int default 0,
  unit_price int default 0,
```

```
total_price int default 0,  
user_id varchar(50) not null,  
order_id varchar(50) not null,  
created_at datetime default NOW()  
);
```

Order Service의 datasource 설정 변경

spring:

```
datasource:  
  
    url: jdbc:mariadb://localhost:포트/스키마  
  
    driver-class-name: org.mariadb.jdbc.Driver  
  
    username: root  
  
    password: 비밀번호입력
```

Kafka Sink Connect 추가

connect의 기본 포트는 8083

커넥트 추가, POST로 전송

```
{  
  
    "name": "my-order-sink-connect",  
  
    "config": {  
  
        "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",  
  
        "connection.url": "jdbc:mysql://localhost:3306/mydb",  
  
        "connection.user": "root",  
  
        "connection.password": "test1357",  
  
        "auto.create": "true", # DB를 자동으로 만들기 설정, 토픽과 같은 이름의 테이블 생성  
  
        "auto.evolve": "true",
```

```

        "delete.enabled":"false",

        "tasks.max":"1",

        "topics":"orders"      # 정보를 받을 토픽
    }
}

```

Order Service Controller 수정

```

@PostMapping("/{userId}/orders")

public ResponseEntity<ResponseOrder> createOrder(@PathVariable("userId") String
userId,

                                                    @RequestBody RequestOrder
request)

                                                    throws JsonProcessingException {

```

```

    ModelMapper modelMapper = new ModelMapper();

    modelMapper.getConfiguration().setMatchingStrategy(MatchingStrategies.STRICT);

```

```

    OrderDto orderDto = modelMapper.map(request, OrderDto.class);

    orderDto.setUserId(userId);

    orderDto.setOrderId(UUID.randomUUID().toString());

    orderDto.setTotalPrice(request.getUnitPrice() * request.getQuantity());

```

```

// catalog-service와 데이터 동기화

kafkaProducer.send(catalogTopic, orderDto);

// order-service의 데이터 동기화(단일 DB에 저장)

orderProducer.send(orderTopic, orderDto);

```



```

        ResponseOrder responseOrder = modelMapper.map(orderDto, ResponseOrder.class);

        return ResponseEntity.status(HttpStatus.CREATED)

                                .body(modelMapper.map(responseOrder,
ResponseOrder.class));
    }

```

각각의 DTO들과 OrderProducer 설정

// Kafka에 넣을 DTO, 직렬화 해주기 잊지말기

```
public class KafkaOrderDto implements Serializable {
```

```
    private Schema schema;
```

```
    private Payload payload;
```

```
}
```

...

// 스키마 정보 클래스

```
public class Schema {
```

```
    private String type;
```

```
    private List<Field> fields;
```

```
    private boolean optional;
```

```
    private String name;
```

```
}
```

...

// Schema 내부의 필드 정의 클래스

public class Field {

private String type;

private boolean optional;

private String field;

}

...

// 실제 데이터

public class Payload {

private String order_id;

private String user_id;

private String product_id;

private int quantity;

private int unit_price;

private int total_price;

}

@Service

@Slf4j

@RequiredArgsConstructor

```

public class KafkaProducer {

    // 카프카에 데이터를 보낼 템플릿
    private final KafkaTemplate<String, String> kafkaTemplate;

    public OrderDto send(String topic, OrderDto orderDto) throws
    JsonProcessingException {

        ObjectMapper objectMapper = new ObjectMapper();

        // dto를 json으로 변환
        String jsonInString = objectMapper.writeValueAsString(orderDto);

        kafkaTemplate.send(topic, jsonInString);

        log.info("Kafka Producer sent data from the Order micro service : " + orderDto);

        return orderDto;
    }
}

@Service
@RequiredArgsConstructor
@Slf4j
@Transactional
public class OrderProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    // 필드, 스키마는 이미 DB 테이블이 존재하기 때문에 그에 맞게 바로 생성

```

```
private final List<Field> fields = Arrays.asList(
    Field.builder().type("string").optional(true).field("order_id").build(),
    Field.builder().type("string").optional(true).field("user_id").build(),
    Field.builder().type("string").optional(true).field("product_id").build(),
    Field.builder().type("int32").optional(true).field("quantity").build(),
    Field.builder().type("int32").optional(true).field("unit_price").build(),
    Field.builder().type("int32").optional(true).field("total_price").build()
);
```

```
private final Schema schema = Schema.builder()
    .type("struct")
    .fields(fields)
    .optional(false)
    .name("orders")
    .build();
```

```
public OrderDto send(String topic, OrderDto orderDto) throws
JsonProcessingException {
```

```
    Payload payload = Payload.builder()
        .order_id(orderDto.getOrderId())
        .user_id(orderDto.getUserId())
        .product_id(orderDto.getProductId())
        .quantity(orderDto.getQuantity())
        .unit_price(orderDto.getUnitPrice())
```

```

        .total_price(orderDto.getTotalPrice())

        .build();

KafkaOrderDto kafkaOrderDto = KafkaOrderDto.builder()

    .schema(schema)

    .payload(payload)

    .build();

ObjectMapper objectMapper = new ObjectMapper();

// json으로 변환

String jsonInString = objectMapper.writeValueAsString(kafkaOrderDto);

kafkaTemplate.send(topic, jsonInString);

log.info("Order Producer sent data from the Order micro-service : " +
kafkaOrderDto);

return orderDto;

}

}

```

테스트

현재 내 주문 정보와 상품 재고는 다음과 같이 나온다.

주문 정보

```

{
    "email": "kobumssh@naver.com",
    "name": "고범석",

```

```
"userId": "b169e54c-c563-449a-9fe4-2b77fa4df2fd",  
  
"orders": [  
  {  
    "productId": "CATALOG-001",  
    "quantity": 95,  
    "unitPrice": 2000,  
    "totalPrice": 190000,  
    "createdAt": "2021-05-24T16:39:24",  
    "orderId": "b0b2d4be-e484-44c8-a42d-258bc79ec1bc"  
  }  
]  
}
```

상품 재고

```
[  
  {  
    "productId": "CATALOG-001",  
    "productName": "Berlin",  
    "unitPrice": 1500,  
    "stock": 4,  
    "createdAt": "2021-05-24T16:19:54.62"  
  },  
  {  
    "productId": "CATALOG-002",  
    "productName": "Tokyo",
```

```
"unitPrice": 1000,  
"stock": 108,  
"createdAt": "2021-05-24T16:19:54.625"  
},  
{  
  "productId": "CATALOG-003",  
  "productName": "Stockholm",  
  "unitPrice": 2000,  
  "stock": 105,  
  "createdAt": "2021-05-24T16:19:54.625"  
}  
]
```

이제 주문을 두번 더 해보자.

```
{  
  "productId" : "CATALOG-002",  
  "quantity" : 18,  
  "unitPrice" : 1000  
}  
  
{  
  "productId" : "CATALOG-003",  
  "quantity" : 15,  
  "unitPrice" : 2000  
}
```

그리고 내 주문 조회 및 상품 재고를 조회해보면?

주문 조회

```
{
  "email": "kobumssh@naver.com",
  "name": "고범석",
  "userId": "b169e54c-c563-449a-9fe4-2b77fa4df2fd",
  "orders": [
    {
      "productId": "CATALOG-001",
      "quantity": 95,
      "unitPrice": 2000,
      "totalPrice": 190000,
      "createdAt": "2021-05-24T16:39:24",
      "orderId": "b0b2d4be-e484-44c8-a42d-258bc79ec1bc"
    },
    {
      "productId": "CATALOG-002",
      "quantity": 18,
      "unitPrice": 1000,
      "totalPrice": 18000,
      "createdAt": "2021-05-24T17:30:50",
      "orderId": "c4b5a3f1-6914-4e67-bf12-0b7461da87c8"
    },
    {
      "productId": "CATALOG-003",
      "quantity": 15,
      "unitPrice": 2000,
```



```
    "totalPrice": 30000,  
    "createdAt": "2021-05-24T17:31:46",  
    "orderId": "9ba95fa6-a61b-43fe-a092-5a5834882b53"  
  }  
]  
}
```

재고

```
[  
  {  
    "productId": "CATALOG-001",  
    "productName": "Berlin",  
    "unitPrice": 1500,  
    "stock": 4,  
    "createdAt": "2021-05-24T16:19:54.62"  
  },  
  {  
    "productId": "CATALOG-002",  
    "productName": "Tokyo",  
    "unitPrice": 1000,  
    "stock": 90,  
    "createdAt": "2021-05-24T16:19:54.625"  
  },  
  {  
    "productId": "CATALOG-003",
```

```

    "productName": "Stockholm",

    "unitPrice": 2000,

    "stock": 90,

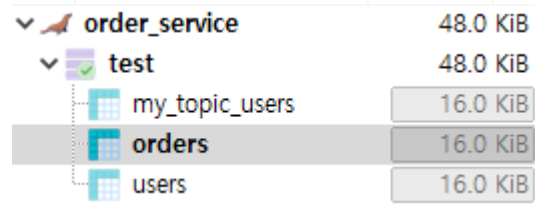
    "createdAt": "2021-05-24T16:19:54.625"

  }
]

```

잘 반영된 것을 확인할 수 있다. Maria DB에서도 확인해보자! HeidiSQL로 접속해서 확인해보았다.

8	b169e54c-c563-449a-9fe4-2b77fa4df2fd	CATALOG-001	b0b2d4be-e484-44c8-a42d-258bc79ec1bc	95	2,000	190,000	2021-05-24 16:39:24
9	b169e54c-c563-449a-9fe4-2b77fa4df2fd	CATALOG-002	c4b5a3f1-6914-4e67-bf12-0b7461da87c8	18	1,000	18,000	2021-05-24 17:30:50
10	b169e54c-c563-449a-9fe4-2b77fa4df2fd	CATALOG-003	9ba95fa6-a61b-43fe-a092-5a5834882b53	15	2,000	30,000	2021-05-24 17:31:46



테이블명도 잘 있다!

본 내용은 이도원님의 [Spring Cloud로 개발하는 마이크로서비스 애플리케이션](#)을 수강하고 정리한 내용입니다.

```

"
}

```