A WEB DEVELOPER'S PORTFOLIO

PRESENTED BY JAESUK LEE

안녕하세요 웹 개발자 이재석입니다.

Portfolio 2

https://github.com/jaesuk95

Personal

이재석 / Jaesuk Lee
1995.11.27 (29세 / 만 27세)
jaesuk95@outlook.com
010-9657-4511
(21023) 인천 계양구 서부간선로

Personality

MBTI : ISTJ
#현실주의자
#논리적
#계획적
#워커홀릭

Graduation

2021.07 코리아 IT 아카데미
안드로이드 및 웹 개발 수료

2017.11 Australian Maritime College
호주 해양대학교 (조선공학과) 졸업

2013.11 Massey High School
뉴질랜드 고등학교 졸업

Experience

2022.04 – 2023.02 (주)모두디자이너
담당업무 : 백엔드 / DevOps
직급 : 개발팀 / 사원 (팀원)

2021.04 – 2021.06 (주)해강기술
담당업무 : R&D 빅데이터 해양자료 분석, 3D 수치모델링
직급 : 해양사업팀 / 사원 (팀원)

2020.03 – 2021.01 (주)오트로닉스
담당업무 : 해양시스템 운영관리
직급 : 해양사업팀 / 사원 (팀원)

2016.12 – 2017.02 (주)빛과울
담당업무 : 건축도면 설계
직급 : 인턴

프로젝트 소개

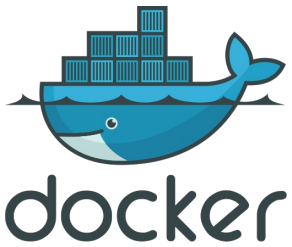
이번 프로젝트의 주제는 전자상거래(e-commerce) 플랫폼으로, 사용자들이 제품을 등록하고 판매할 수 있는 플랫폼을 구축하는 것입니다. 이에 따라 Spring Boot 마이크로서비스 아키텍처를 선택한 이유는:

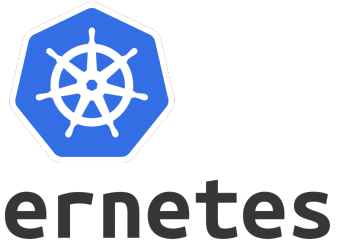
1. Spring Boot 마이크로서비스 아키텍처는 각각의 기능을 독립적으로 개발하고 배포할 수 있는 모듈화된 구조를 제공합니다. 이는 개발자들이 효율적으로 작업할 수 있고, 유지보수와 확장성 측면에서 이점을 제공합니다.

2. Spring Boot는 내장된 강력한 기능을 갖춘 프레임워크로, 빠른 개발을 가능하게 해줍니다. 또한, Spring Boot는 서비스 디스커버리, 로드 밸런싱, 분산 추적과 같은 중요한 마이크로서비스 패턴들을 내장하고 있어 안정성과 확장성을 보장합니다.

따라서, 이러한 이유로 Spring Boot 마이크로서비스 아키텍처를 선택하게 되었습니다.

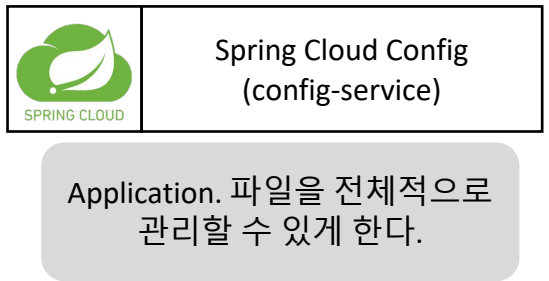
배포 단계에서는 Docker 와 Kubernetes 를 선택했습니다. 이번 프로젝트는 마치 제가 스스로 사업을 시작하고 웹사이트를 호스팅하는 것처럼 진행했습니다. 이러한 마인드셋으로 (Mindset) 빠르게 학습할 수 있었습니다.

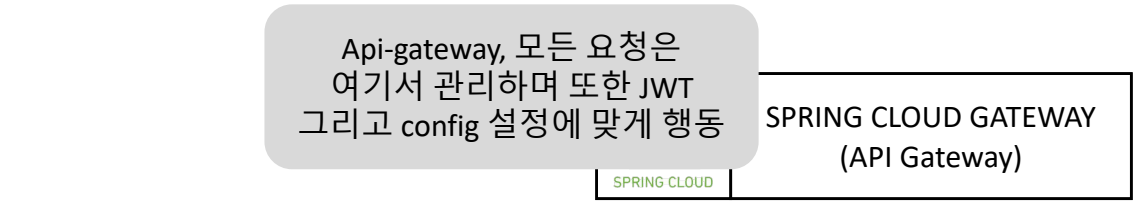


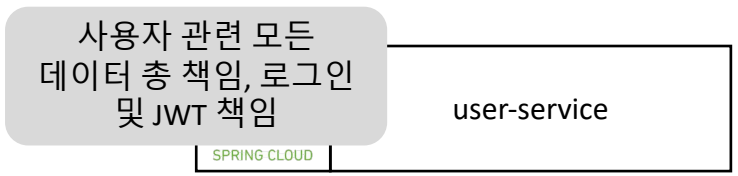


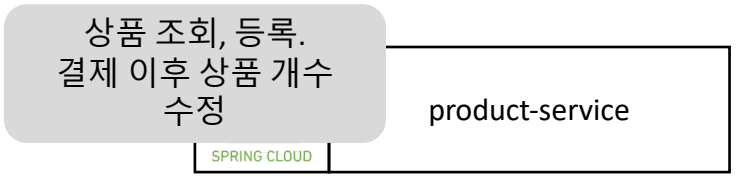
SPRING CLOUD ARCHITECTURE

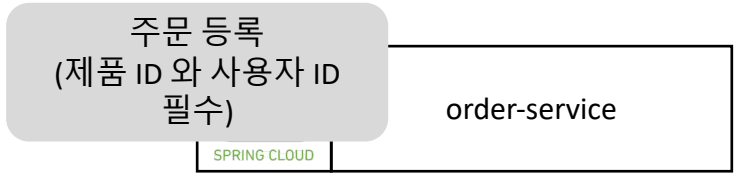
각각의 Springboot Microservices 들은 해당 프로젝트를 구현하기 위해 필수적으로 요구되는 Microservice 이며, 각 서비스마다 주요한 역할을 담당하고 있습니다.

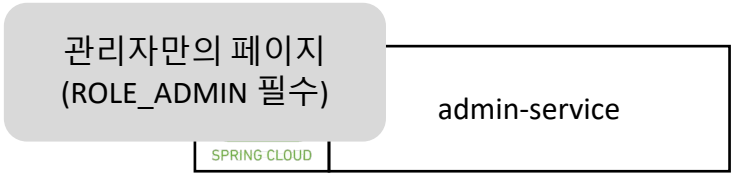


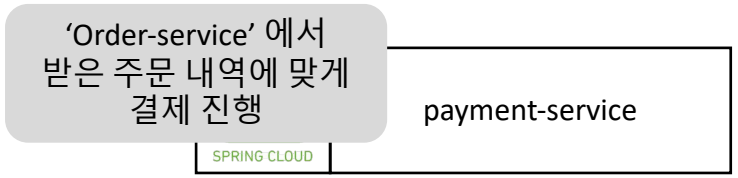






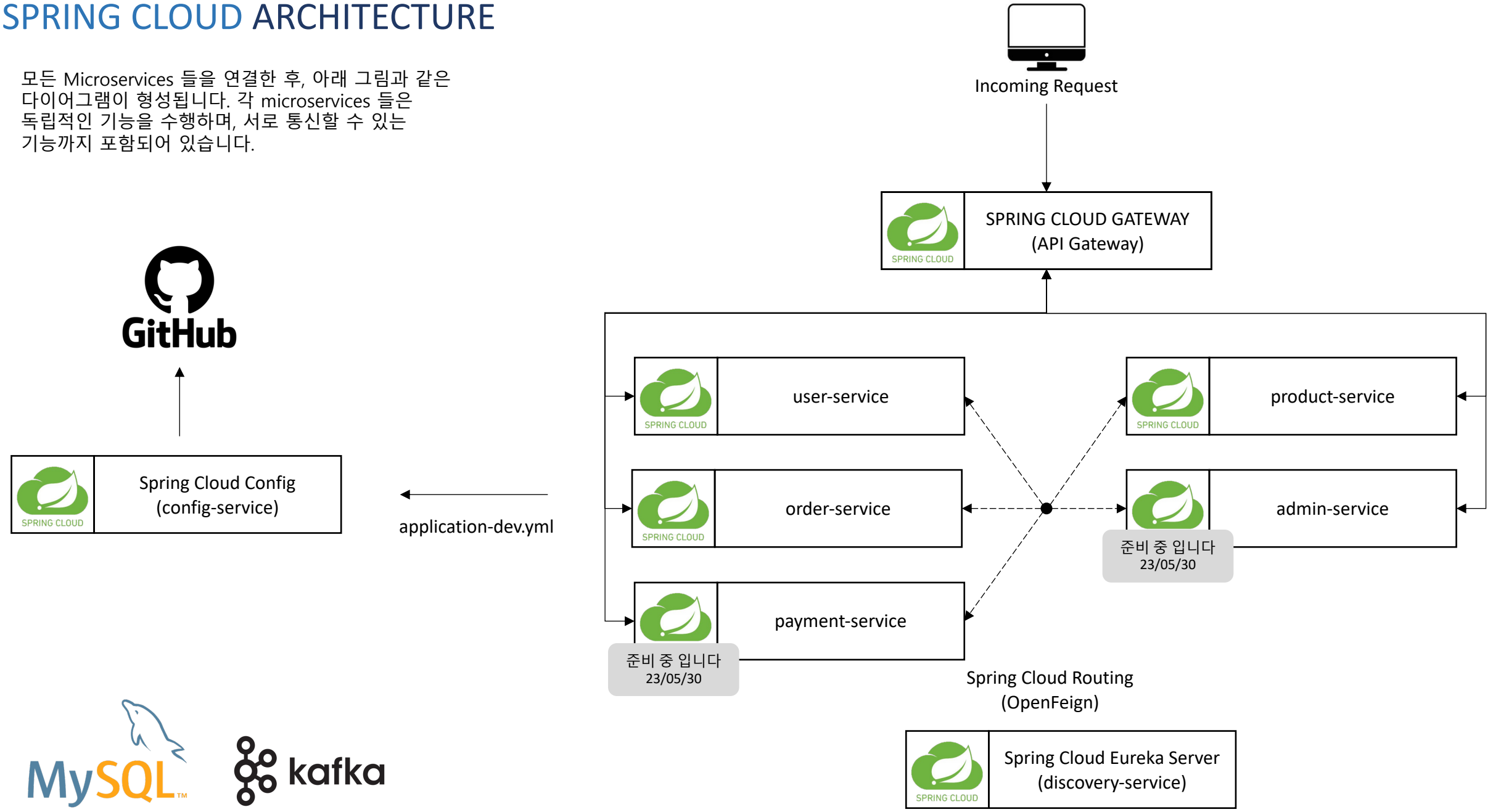






SPRING CLOUD ARCHITECTURE

모든 Microservices 들을 연결한 후, 아래 그림과 같은 다이어그램이 형성됩니다. 각 microservices 들은 독립적인 기능을 수행하며, 서로 통신할 수 있는 기능까지 포함되어 있습니다.



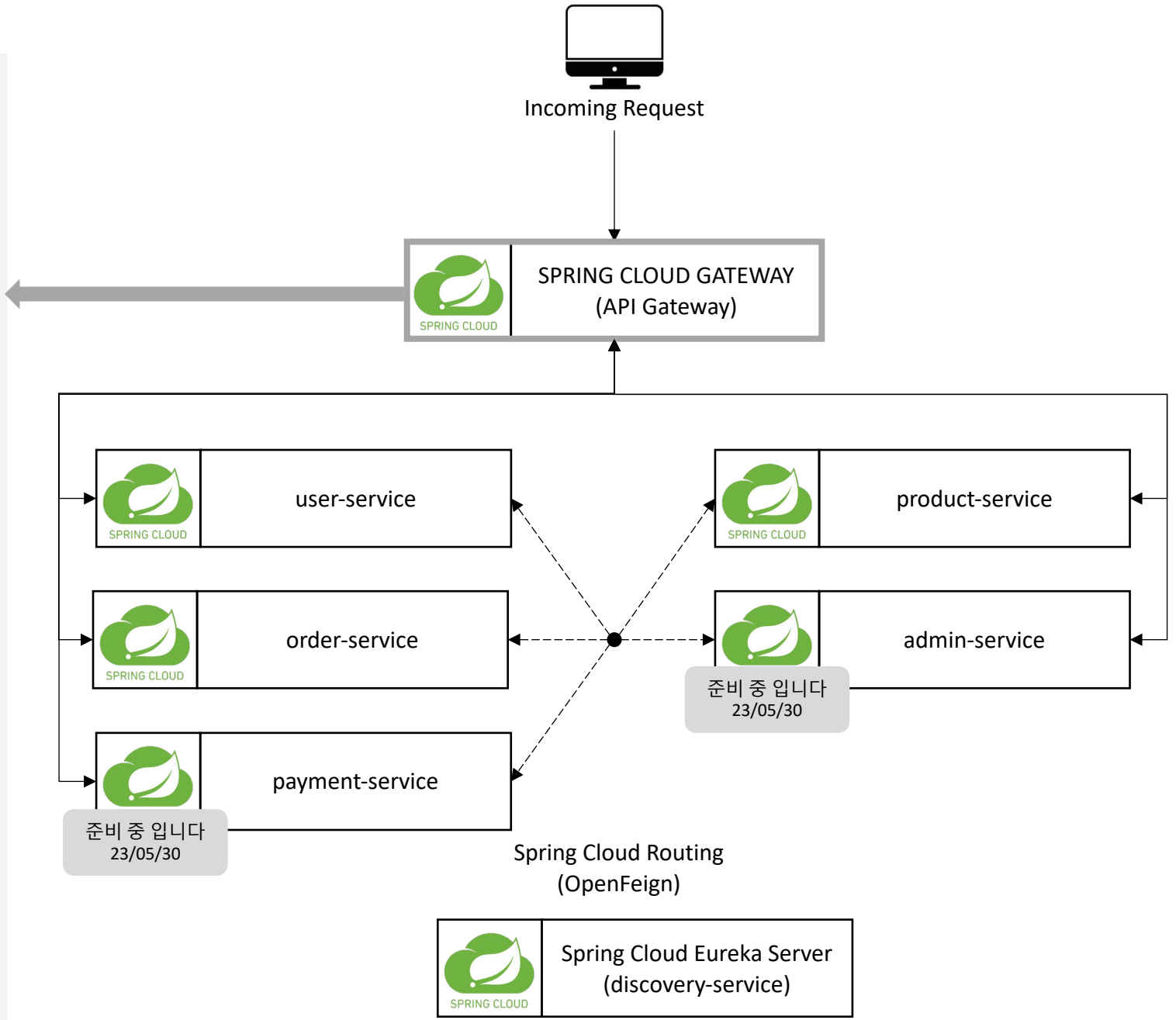
SPRING CLOUD ARCHITECTURE

API gateway 의 추가적인 주 역할은 Client 의 JWT token 을 확인합니다. 클라이언트의 요청에 따라 토큰이 반드시 필요한 요청이 있습니다. 예를 들어 클라이언트의 권한 USER_ROLE 를 반드시 확인해야 하는 과정이 있습니다.

밑에 있는 코드를 보시면, 'api/user-service/profile/**' 는 필터가 추가되어 있습니다. 여기서 Client 의 토큰을 검증하는 역할을 합니다.

반면, 'api/user-service/auth/**' 에서는 필터가 없어 토큰을 확인하지 않는 차이가 있습니다.

```yaml
cloud:
  gateway:
    default-filters:
      - name: GlobalFilter
        args:
          baseMessage: Spring Cloud Gateway Global Filter
          preLogger: true
          postLogger: true
    routes:
      - id: user-service
        uri: lb://USER-SERVICE
        predicates:
          - Path=/api/user-service/auth/**

      - id: user-service
        uri: lb://USER-SERVICE
        predicates:
          - Path=/api/user-service/profile/**
        filters:
          - AuthorizationHeaderFilter
```



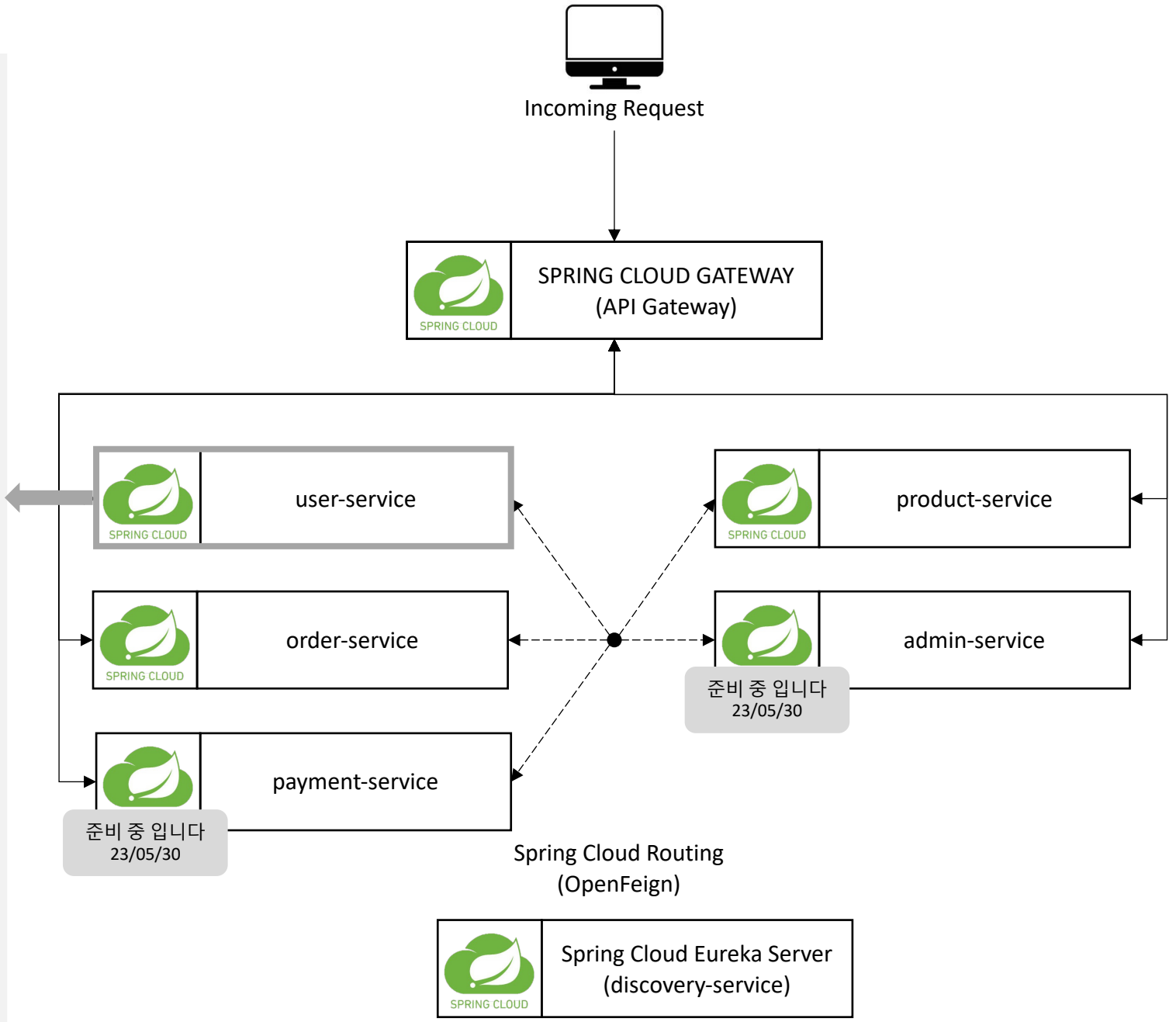
SPRING CLOUD ARCHITECTURE

클라이언트가 로그인을 시도하게 되면
'AuthenticationFilter' 의 'successfulAuthentication'
메소드를 통과하게 됩니다.
이전 단계에서 클라이언트의 정보가 검증 되어 있어
서버에서는 토큰을 생성하여 헤더에 반환합니다.

토큰 Payload 안에는 클라이언트의 ID, 토큰 만료시간,
권한까지 포함되어 있어 특정 Microservice 에서 권한이
필요할 경우 토큰을 API gateway 를 통해 검증할 수
있습니다.

이로 인해, 각 microservices 들은 부담을 줄일 수 있는
장점입니다.

```
@Override
protected void successfulAuthentication(HttpServletRequest request, HttpSer

    log.debug(((User)authResult.getPrincipal()).getUsername());
    String username = ((User) authResult.getPrincipal()).getUsername();
    UserDto userDetails = userService.getUserDetailsByEmail(username);

    String token = Jwts.builder()
            .setSubject(userDetails.getUserId())
            .claim( name: "user_role",userDetails.getRole().toString())
            .setExpiration(new Date(System.currentTimeMillis() + Long.parse
            .signWith(SignatureAlgorithm.HS512, env.getProperty("token.secr
            .compact();

    response.addHeader( name: "token", token);
    response.addHeader( name: "userId", userDetails.getUserId());

}
```



SPRING CLOUD ARCHITECTURE

주문서를 생성하기 위해서는 'order-service' microservice 에서 요청을 받아야합니다. 백단에서 진행되는 과정은:

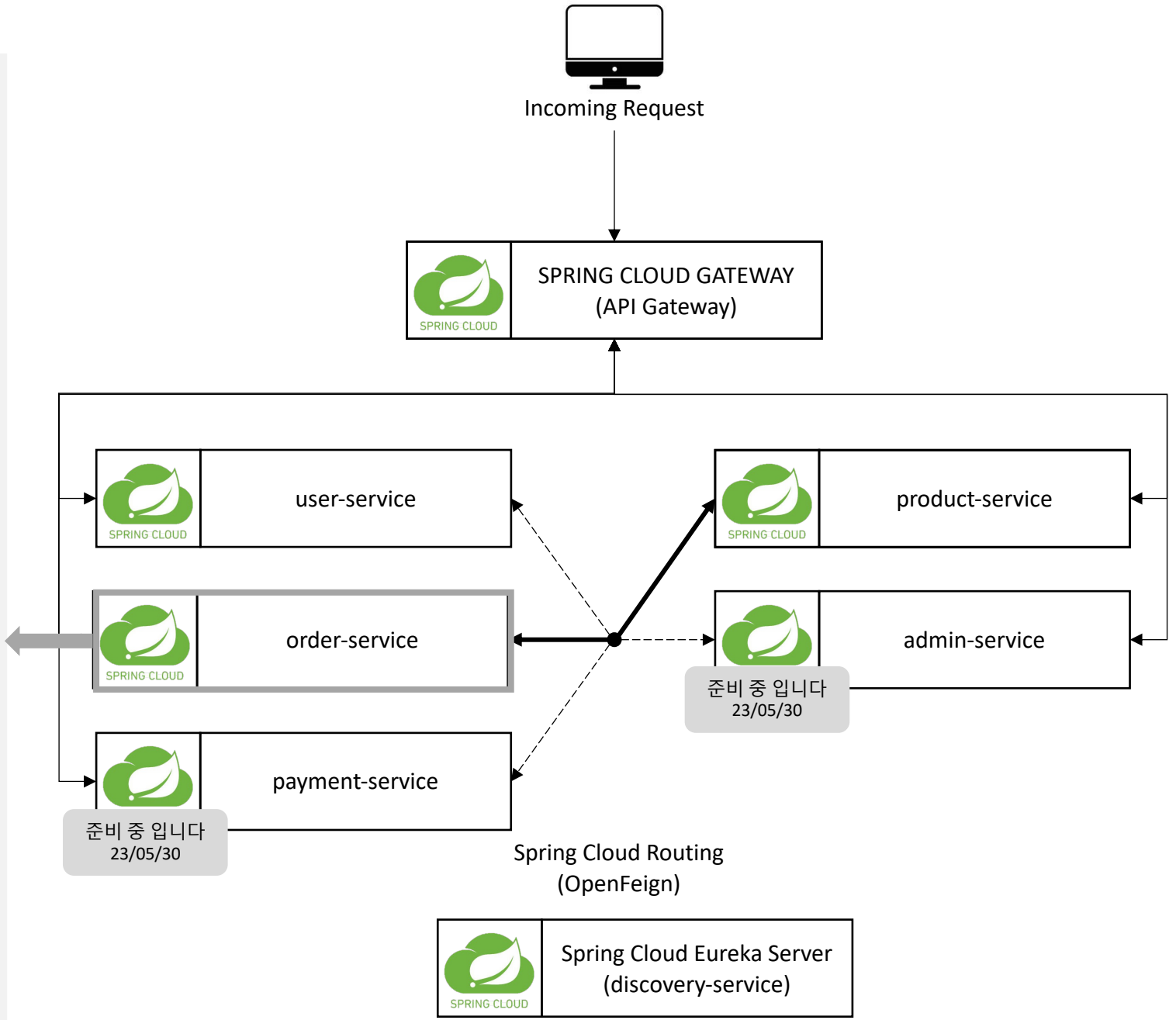
1. 마이크로서비스 'order-service' 는 토큰 인증을 필수적으로 해야하므로, API-Gateway 에서 검증을 1차 적으로 해야합니다.

2. 이후, order-service 에서 상품 ID 와 제품의 제고를 확인을해 구매 가능한 상품인지 'product-service' 와 소통을 합니다.

3. 주문서를 생성하기 위해서는 Kafka Message Queuing 에 메시지를 전송합니다.

4. Kafka Message Queuing 서버에 전송하는 이유는 데이터를 한 곳에 모으기 위해서 사용됩니다. 자세한 내용은 다음 페이지에서 확인하실 수 있습니다.

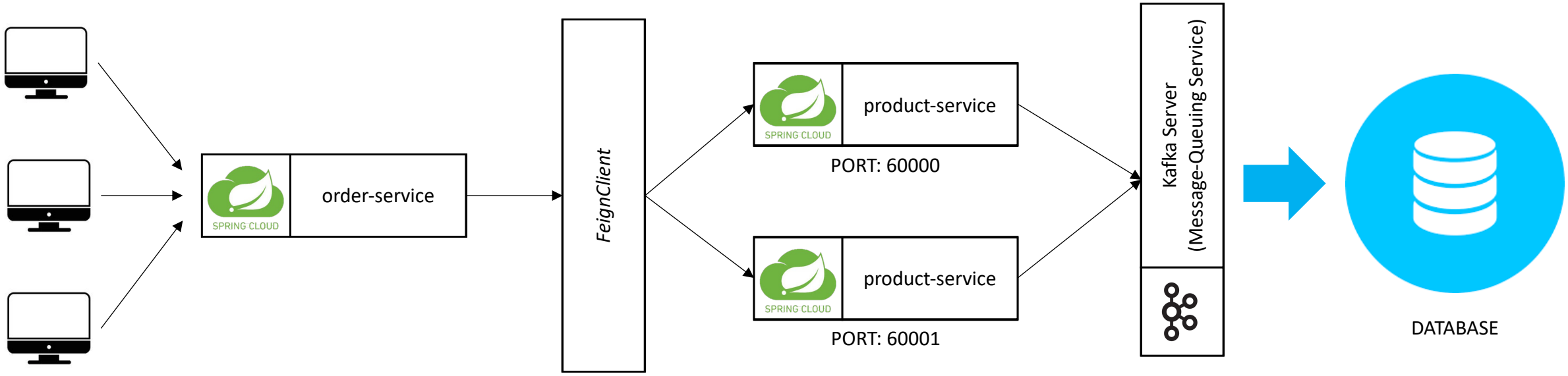
```java
@Override
public ResponseOrder registerUserOrder(RequestOrder requestOrder) {
    String user_id = requestOrder.getUser_id();
    String productId = requestOrder.getProductId();
    ResponseOrder responseOrder = new ResponseOrder();

    log.info("Before call orders microservice");
    CircuitBreaker circuitbreaker = circuitBreakerFactory.create("circuitBreaker");
    ResponseProduct responseProduct = (ResponseProduct) circuitbreaker.run(() ->
                    productServiceClient.productAvailable(productId),
            throwable -> new ArrayList<>());// throwable -> new ArrayList<>() 이 코드
    log.info("After called orders microservice");

    if (responseProduct == null) {
        responseOrder.setMessage("Server Error");
```



SPRING CLOUD ARCHITECTURE

Kafka Sink Connect 를 사용하여 데이터베이스에 등록하는 방식을 활용하고 있습니다.

이로 인해, 인스턴스가 늘어나도 단 하나의 데이터 베이스에서 관리하며, 또한 동시성 문제를 해결해주는 역할을 수행합니다



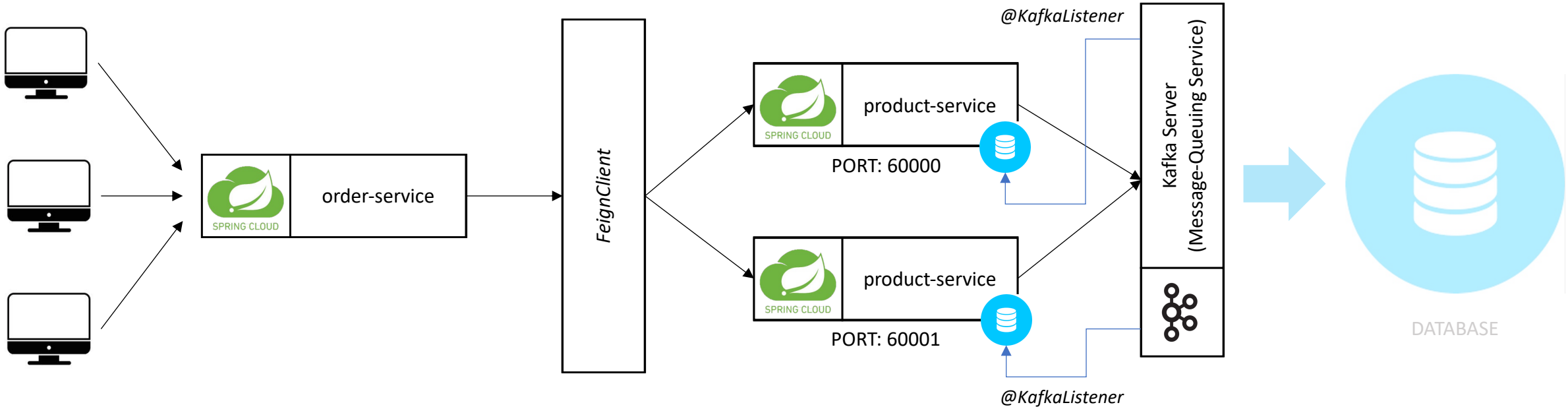

```
1 usage
List<Field> fields = Arrays.asList(new Field( type: "string",  optional: true,  field: "order_id"),
        new Field( type: "string",  optional: true,  field: "user_id"),
        new Field( type: "string",  optional: true,  field: "product_id"),
        new Field( type: "int32",  optional: true,  field: "qty"),
        new Field( type: "int32",  optional: true,  field: "total_price"),
        new Field( type: "int32",  optional: true,  field: "unit_price"));
1 usage
Schema schema = Schema.builder()
        .type("struct")
        .fields(fields)
        .optional(false)
        .name("orders")
        .build();
```

Kafka sink connect 를 활용하기 데이터 베이스에 등록하기 위해서는 kafka 의 정해진 format 으로 전송해야 한다.

좌측에 있는 코드처럼 'fields' 에는 테이블 형식에 맞게 구조를 등록하고, payload 에는 저장하고자 하는 데이터를 입력한다.

SPRING CLOUD ARCHITECTURE

@KafkaListener 도 해당 프로젝트에서 사용되고 있습니다.
데이터 등록이 아닌 그 외에 작업들을 하고 있으며 예를 들어, 제품의 재고 업데이트, 삭제등 Kafka sink connect 가 아닌 방식도 활용되고 있습니다.




```java
@Service
public class KafkaProductConsumer {

    2 usages
    private final ProductRepository productRepository;

    Jaesuk *
    @KafkaListener(topics = "product_purchase_update")
    public void processProductUpdate(String kafkaMessage) {
        log.info("Kafka message: => {}", kafkaMessage);

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

Java 코드 예시.

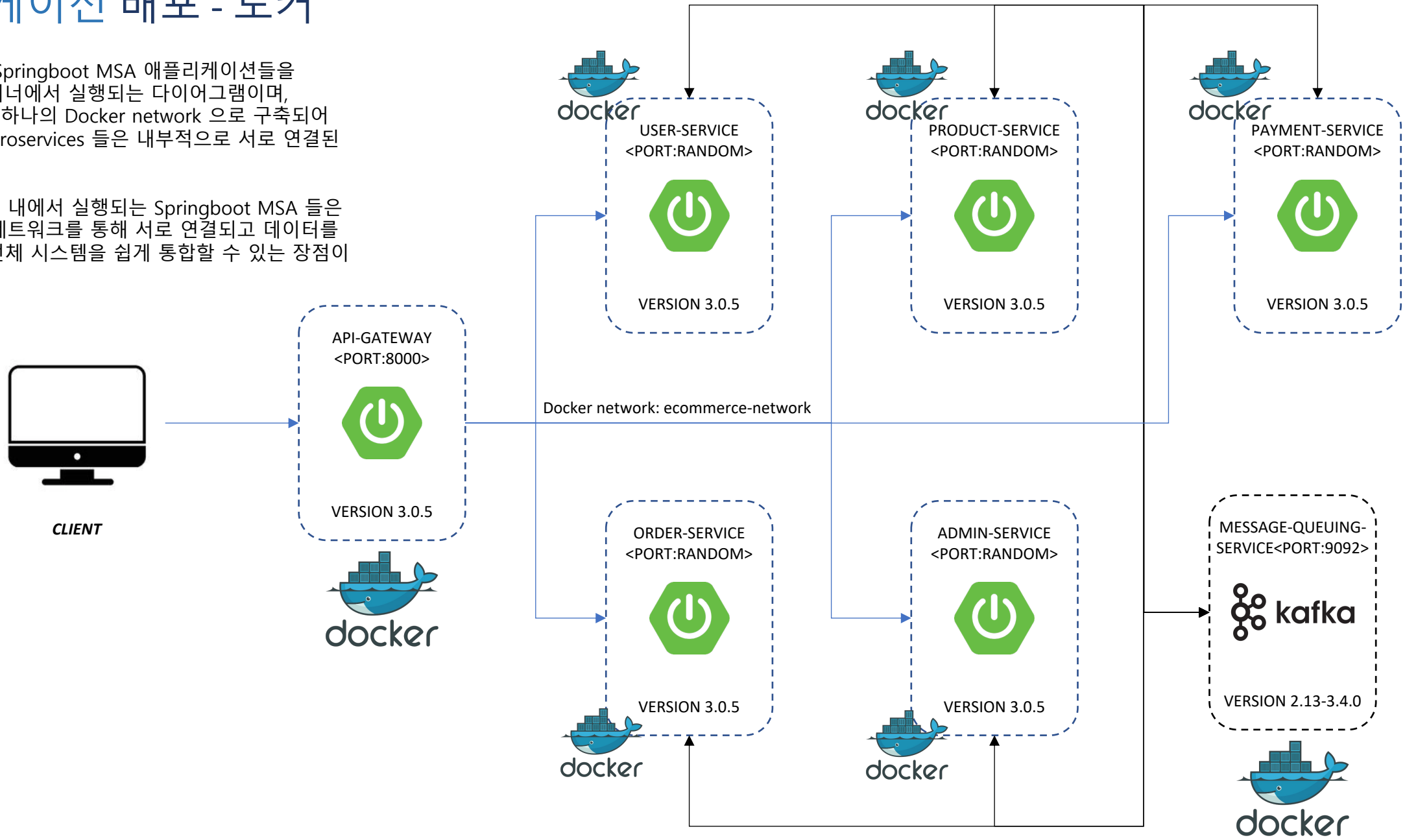
Kafka 메시지에는 업데이트할 주요 내용이 담겨있다.

JPA-Repository 를 활용하여 업데이트할 내용을 간편하게 업데이트 합니다.

애플리케이션 배포 - 도커

아래 그림은, Springboot MSA 애플리케이션들을 Docker 컨테이너에서 실행되는 다이어그램이며, 컨테이너들은 하나의 Docker network 으로 구축되어 있어 모든 microservices 들은 내부적으로 서로 연결된 모습입니다.

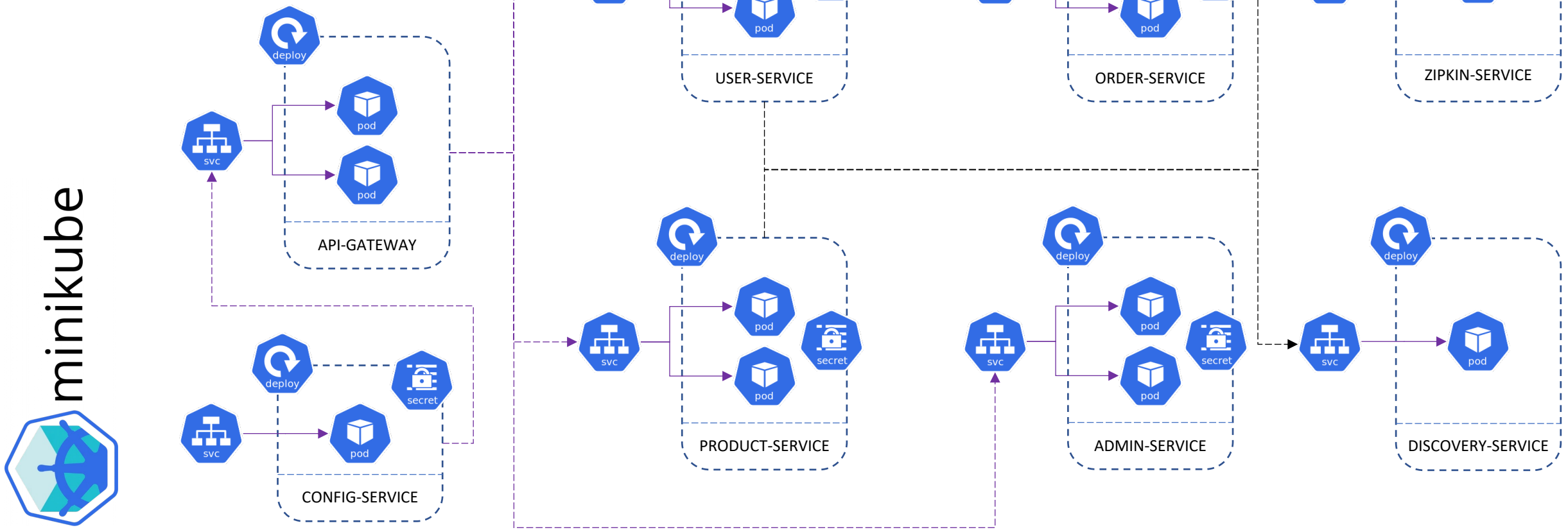
도커 컨테이너 내에서 실행되는 Springboot MSA 들은 이러한 도커 네트워크를 통해 서로 연결되고 데이터를 주고받으며, 전체 시스템을 쉽게 통합할 수 있는 장점이 있습니다.



애플리케이션 배포 - K8S

이번 프로젝트의 마지막 단계로 Kubernetes 를 사용했습니다. Kubernetes 를 선택한 이유는 다음과 같습니다:

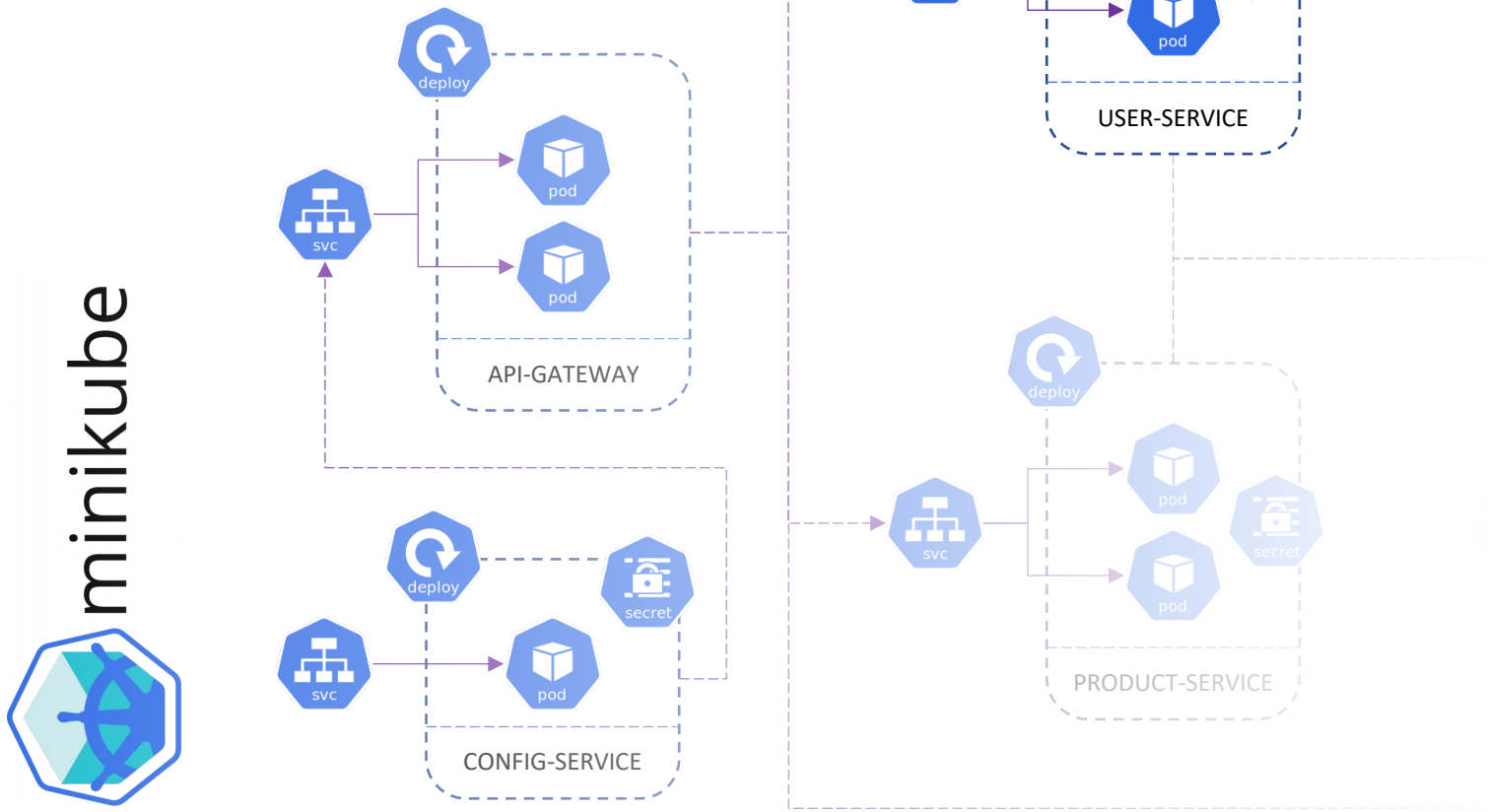
1. 대규모 애플리케이션을 관리하기 매우 효율적입니다.
2. 확장 가능한 Architecture 와 유연성을 가지고 있습니다.
3. 다양한 클라우드 제공 업체(AWS,Azure,GCP)와의 통합이 가능 하다는 장점이 있습니다.



애플리케이션 배포 - K8S

이번 프로젝트의 마지막 단계로 Kubernetes 를 사용했습니다. Kubernetes 를 선택한 이유는 다음과 같습니다:

1. 대규모 애플리케이션을 관리하기 매우 효율적입니다.
2. 확장 가능한 Architecture 와 유연성을 가지고 있습니다.
3. 다양한 클라우드 제공 업체(AWS,Azure,GCP)와의 통합이 가능 하다는 장점이 있습니다.



Kubernetes 의 'Secret' 을 활용하여 데이터베이스 비밀번호와 같은 중요한 정보를 안전하게 저장하기 위해 사용되었습니다.

'Literal' 시크릿 메소드를 사용하여 시크릿을 생성할 수 있었습니다.

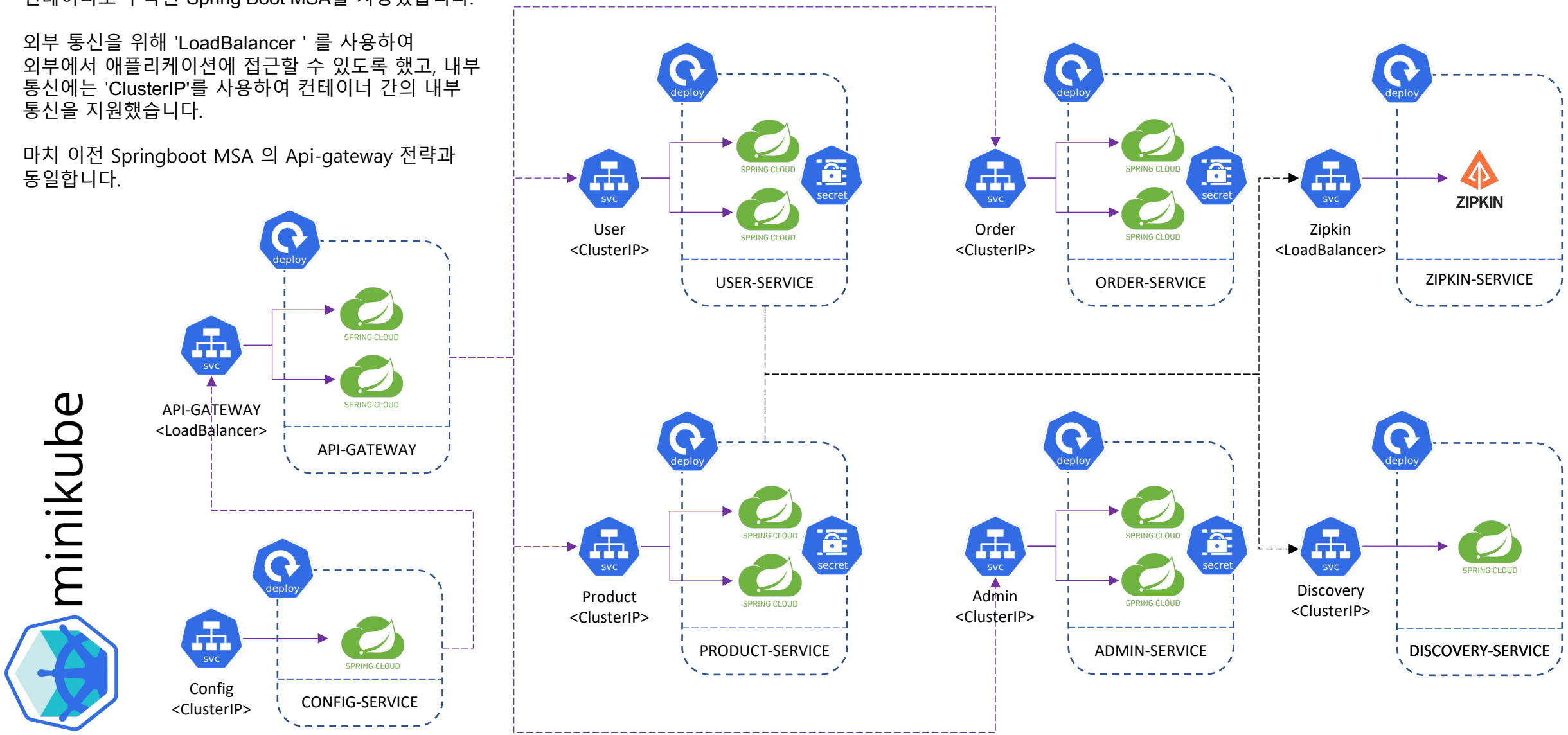
```
        - name: SPRING_DATASOURCE_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: dbsecret
      ports:
      - containerPort: 8080

---
```

애플리케이션 배포 - K8S

Kubernetes Minikube 에서는 이전에 이미 Docker 컨테이너로 구축된 Spring Boot MSA를 사용했습니다.

외부 통신을 위해 'LoadBalancer ' 를 사용하여 외부에서 애플리케이션에 접근할 수 있도록 했고, 내부 통신에는 'ClusterIP'를 사용하여 컨테이너 간의 내부 통신을 지원했습니다.

마치 이전 Springboot MSA 의 Api-gateway 전략과 동일합니다.



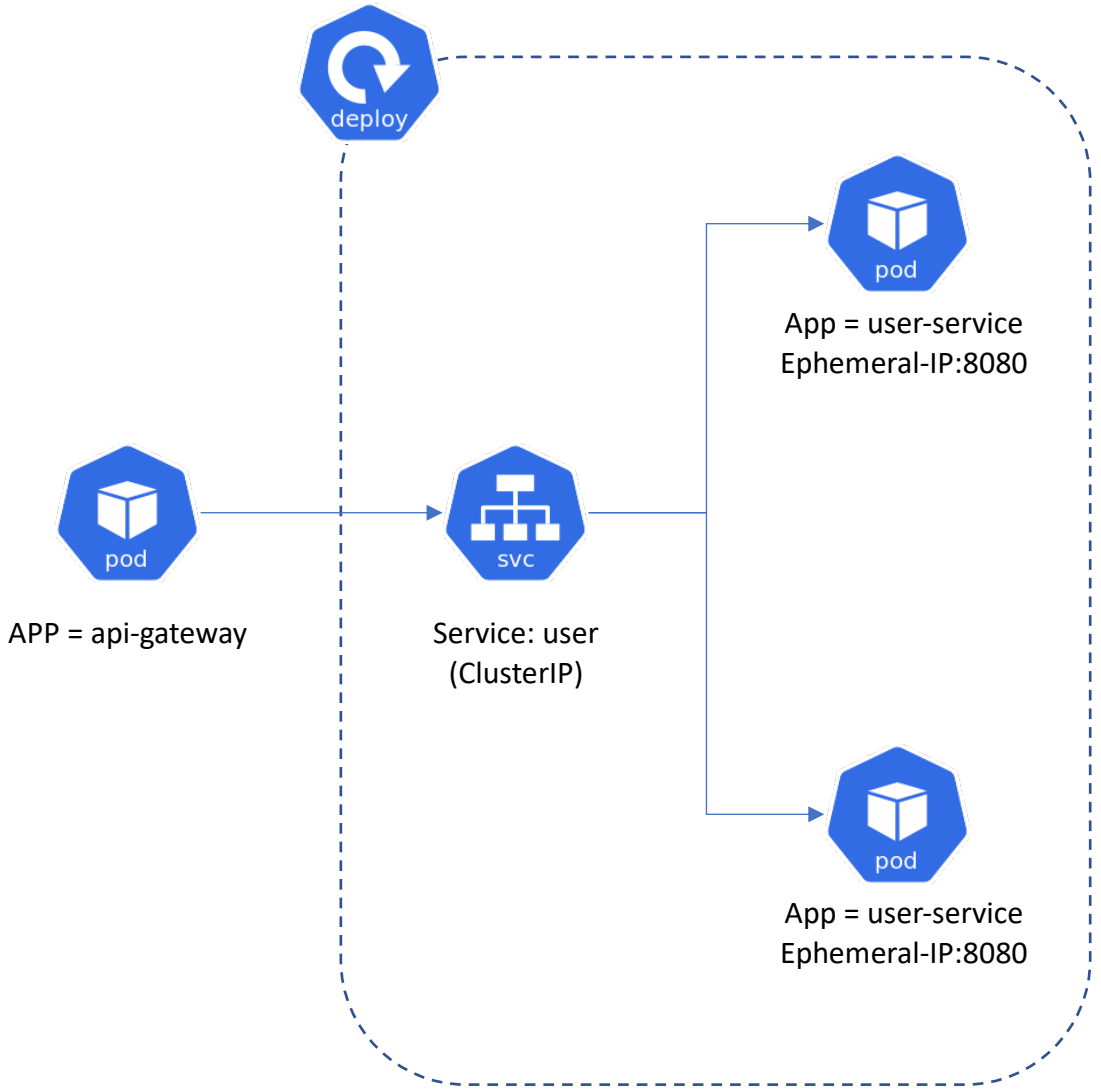
애플리케이션 배포 - K8S

Kubernetes 를 사용하기 전에는 Docker-network 를 사용하여 redirection URI 를 IP 주소 대신 Docker Container name 이름을 사용할 수 있었습니다.

Kubernetes 도 Docker-network 와 비슷한 개념을 가진 방식을 'Kubernetes Discovery Client' 을 제공하며, POD 또는 SERVICE IP 주소 대신 서비스의 이름을 대신 입력해주면 됩니다.

이로 인해, 새로운 서비스가 추가, 제거, 리부팅 되는 경우에도 수동으로 IP 주소를 변경하지 않아도 되는 장점을 활용하고 있습니다.

```yaml
      - name: user
        image: jaesuk95/portfolio-user-service:0.0.11
        imagePullPolicy: Always
        env:
          - name: SPRING_CLOUD_CONFIG_URI
            value: "http://config-service:8888"
          - name: EUREKA_CLIENT_ENABLED
            value: "true"
          - name: EUREKA_CLIENT_SERVICE-URL_DEFAULTZONE
            value: "http://discovery-service:8761/eureka"
          - name: SPRING_RABBITMQ_HOST
            value: "rabbitmq"
          - name: SPRING_DATASOURCE_URL
            value: "jdbc:mysql://192.168.0.4:3306/portfolio2_user"
          - name: SPRING_DATASOURCE_PASSWORD
            valueFrom:
              secretKeyRef:
                name: mysecret
                key: dbsecret
        ports:
        - containerPort: 8080

---

apiVersion: v1
kind: Service
metadata:
  name: user
spec:
  selector:
    app: user
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```



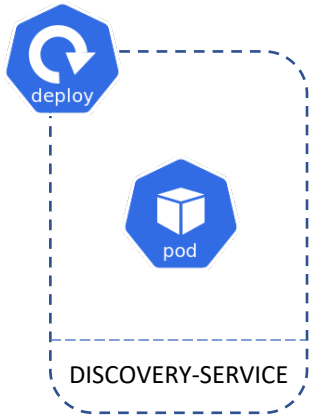
애플리케이션 배포 - K8S

Kubernetes 에도 Microservices 와 마찬가지로 내부 통신은 FeignClient 을 사용되었으며, Redirection URI 를 변경하는 것 외에는 Springboot 에서 특별한 변경 사항이 없었습니다.

Kubernetes 에서 FeignClient 가 정상적으로 동작할 수 있었던 이유는 Eureka Discovery-Service 를 그대로 가져왔기 때문입니다.

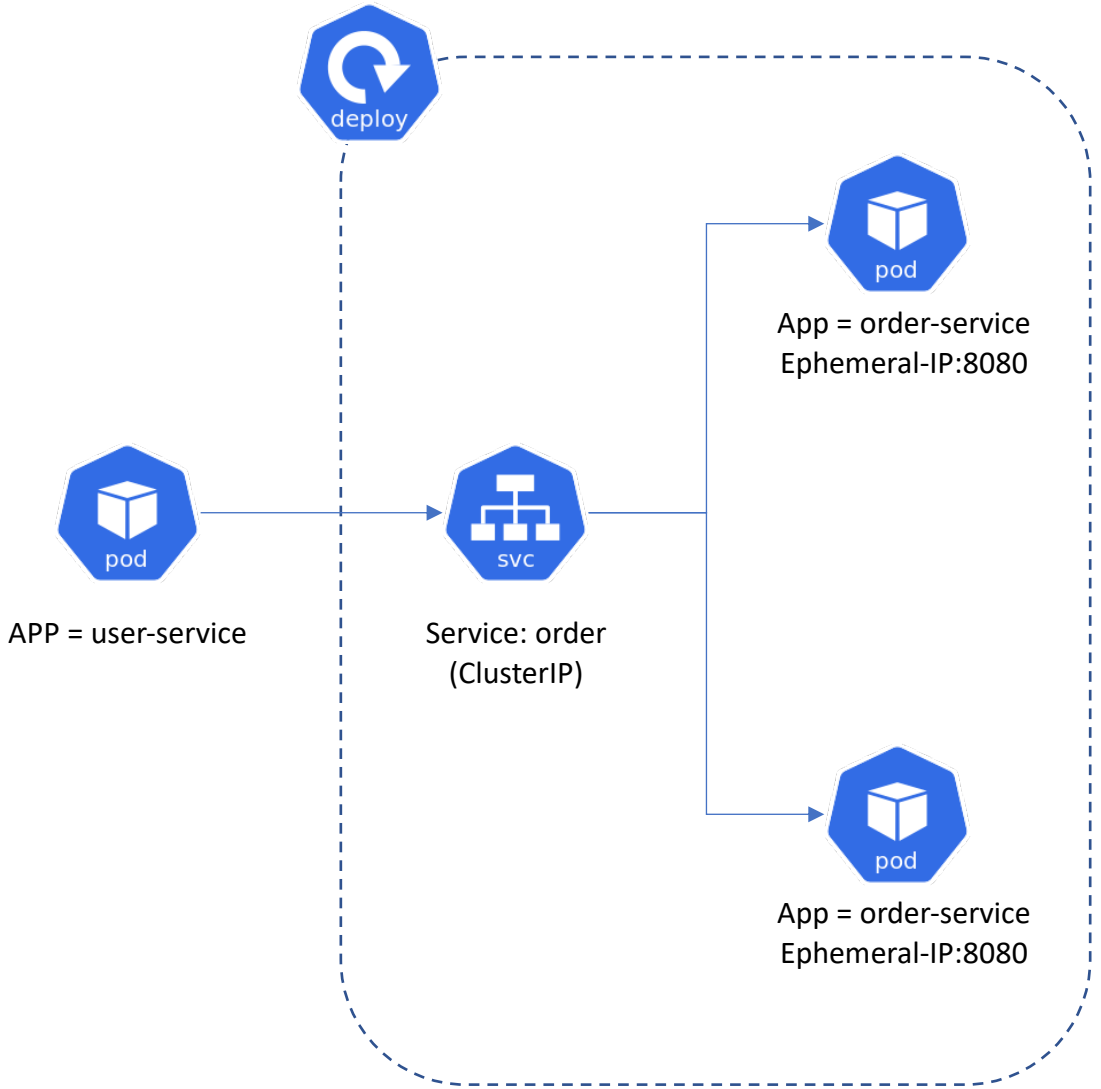
```
10
    2 usages  Jaesuk *
11  @FeignClient("order-service")
12  public interface OrderServiceClient {
        1 usage  Jaesuk
13      @GetMapping("/order-service/{userId}/orders")
14      List<ResponseOrder> getOrders(@PathVariable String userId);
15
```

- Microservices 간의 통신은 FeignClient 사용



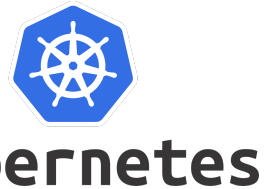
현재 Eureka Discovery-service 가 필요한 상황입니다. Kubernetes Discovery Client 인지하고 있지만 연구할 시간이 조금 더 필요합니다.

앞으로 Springboot 의 'discovery-service' 마이크로 서비스를 중단하고 Kubernetes Discovery 로 대체할 계획입니다.

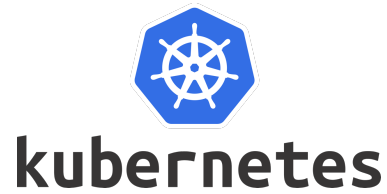


스프링 마이크로서비스 깃 주소:
https://github.com/jaesuk95/portfolio2-msa

스프링 마이크로서비스 Notion 자가노트 기록:
https://www.notion.so/Inflearn-MSA-bc2e8cac4a8a4f13bf4d52c350a88d7c?pvs=4



쿠버네티스 작업 관련 깃 주소:
https://github.com/jaesuk95/portfolio2-msa/tree/master/k8s/minikube



쿠버네티스 Notion 자가노트 기록:
https://www.notion.so/Amigos-Kubernetes-281b18b75bcb481c87e4910c6765068b?pvs=4

감사합니다

끝까지 봐주셔서 대단히 감사드립니다.