Portfolio by Daehyun Lee

왜?를 고민하는 **백엔드 개발자 이대현**입니다.

ABOUT ↗

PROJECTS ↗

CONTACT ↗

ABOUT

안녕하세요, **백엔드 개발자 이대현**입니다.

문제 해결 중심의 사고와 개발 경험을 바탕으로, 사용자 중심의 웹 서비스를 설계하고 운영한 경험이 있습니다.
Spring Boot 기반의 서비스를 EC2 환경에서 직접 배포하며 인프라 전반에 대한 이해를 넓혔습니다.
Jenkins, Nginx, Docker, S3 등 다양한 도구를 활용한 무중단 배포 경험과, SaaS로그 수집 라이브러리 개발 등
창의적인 개발 경험을 통해 사용자 경험과 시스템 안정성을 동시에 고려하는 개발 역량을 갖추고 있습니다.

Awards & Certifications

삼성 청년 SW 아카데미 최우수상
삼성전자 | 2025.05.22
자율 프로젝트

삼성 청년 SW 아카데미 우수상
삼성전자 | 2025.04.11
특화 프로젝트

삼성 청년 SW 아카데미 우수상
삼성전자 | 2025.02.21
공통 프로젝트

SQL 개발자(SQLD)
한국데이터산업진흥원 | 2024.12.13
자격번호 SQLD-055017245

정보처리기사
한국산업인력공단 | 2024.12.11
자격번호 24203130079B

CAREER SUMMARY

삼성 청년 SW 아카데미 (SSAFY) 12기

| 삼성전자

2024.07.01 – 2025.06.05

- Java, Spring Boot, JPA, REST API, MySQL 등 백엔드 심화 기술, CS 및 알고리즘 역량 강화.
- Docker, AWS, Jenkins 기반 CI/CD 구축/운영 및 모니터링 시스템 활용 (DevOps).
- 총 3회의 팀 프로젝트(CHO:LOG, YooHoo, TheRamzee) 중 3회 모두 수상, 협업/문제 해결 능력 함양.
- 코드 리뷰, 스터디, 기술 블로그/노션 기록을 통한 꾸준한 학습 및 공유 습관 형성.

주식회사 하이크루

| 개발팀 사원

2021.05.17 – 2022.02.28

- eGov 표준프레임워크 기반 공공 SI 프로젝트 개발 환경 경험.
- 사내 게시판 시스템 개발 참여 (Spring Boot, JSP, MariaDB).
- 차세대 NEIS 개발 사업 참여 (요구사항 분석 및 기능 명세서 작성 보조).

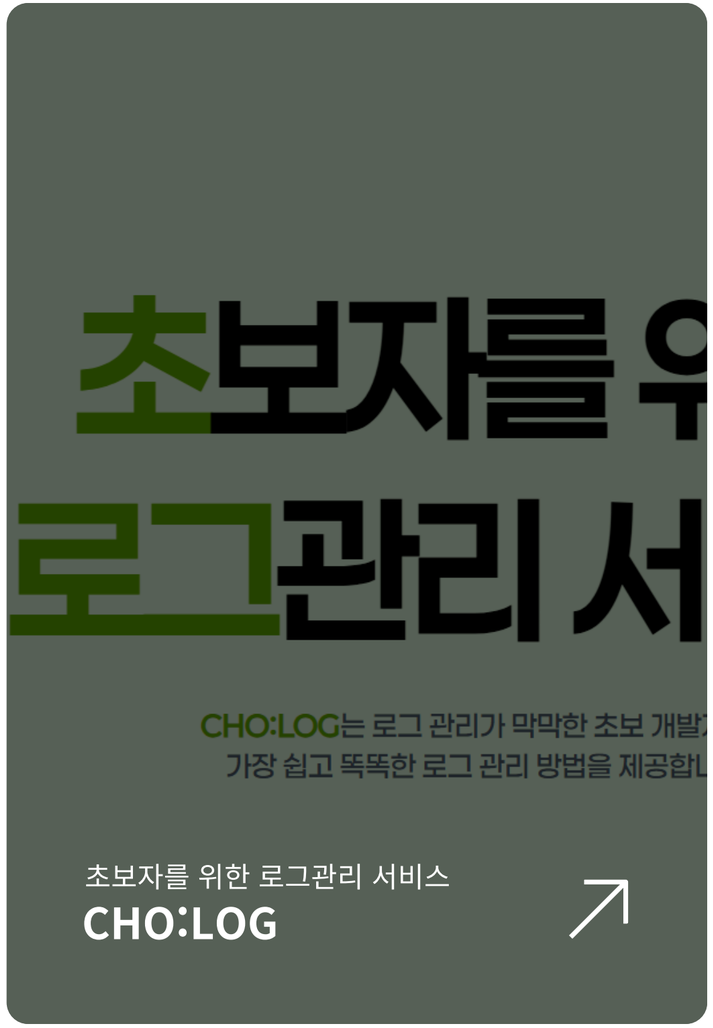
오픈 API 활용 핀테크 개발자 양성과정 수료

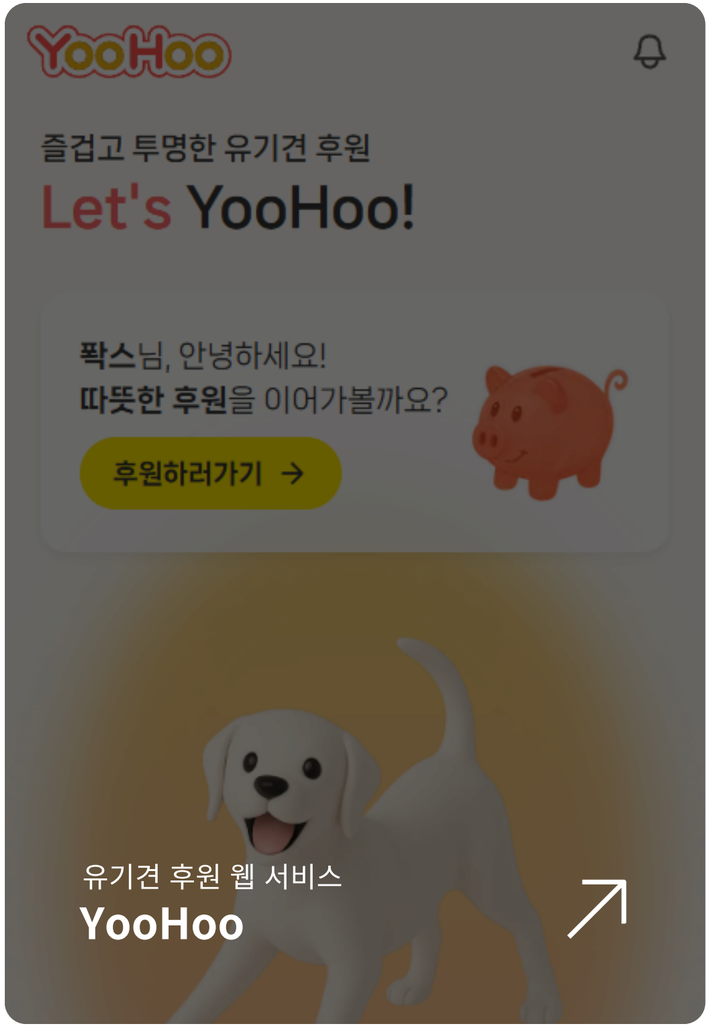
| 예담직업전문학교

2020.10.01 – 2021.04.30

- Java, Spring Boot, JSP, OracleDB 등 웹 개발 기초 및 API 연동 기술 학습.

PROJECTS


초보자를 위
로그관리 서
CHO:LOG는 로그 관리가 막막한 초보 개발
가장 쉽고 똑똑한 로그 관리 방법을 제공합
초보자를 위한 로그관리 서비스
CHO:LOG


YooHoo
즐겁고 투명한 유기견 후원
Let's YooHoo!
꽉스님, 안녕하세요!
따뜻한 후원을 이어가볼까요?
후원하러가기 →
유기견 후원 웹 서비스
YooHoo

초보자를 위한 로그관리 서비스

CHO:LOG

기획 배경

초보 개발자들이 로깅 시스템을 구축하며 겪는 어려움에 공감해, 누구나 쉽게 로그를 수집하고 분석할 수 있는 SDK를 만들고자 했습니다.
이 프로젝트는 프론트엔드와 백엔드의 로그를 유기적으로 통합함으로써, 개발자의 디버깅 경험을 개선하고 안정적인 서비스 운영을 위한 첫걸음을 돕는 것을 목표로 했습니다.

성과

- 프로젝트 최우수상 수상
- 삼성 청년 SW 아카데미 오픈소스 프로젝트 선정
- 전시회 프로젝트 선정

기여도 ●●●○○

- 백엔드 SDK 설계 및 개발 (Java, Logback, Log4j2)을 통해 CHO:LOG 서비스 연동 자동화.
- Spring Boot Starter 개발로 의존성 추가 및 yml 설정만으로 서비스 연동이 가능하도록 사용자 편의성 극대화.
- 비동기 처리, 배치 전송, GZIP 압축, 디스크 큐 등 안정적인 대용량 로그 전송을 위한 핵심 기능 구현

* cho:log 과 함께

쉽게

빠르게

편하게

도움이 필요하면
나를 찾아오라굴~!

왜 초록인가요?

초기 도입비용과
연간 라이선스 비용 부담↓

시스템 구축부터 운영까지
전문적인 기술 지식 요구 X

기존 대비 90%
이상 비용 절감

"

지금 가장 중요한 게
뭔 거 같아?

"

로그 관리

어떤 것부터 볼까?

CHO:LOG
시네마틱 예고편

CHO:LOG
서비스 실사용 영상

웹사이트 https://www.cholog.com

이메일 207cholog@gmail.com

로그가 낯선 당신을 위한
초-간편 로그관리 서비스

CHO : 간단 설치
CHO : 간편 로그관리

* cho:log

SSAFY 12th B207

로그가 낯선 당신을 위해

* cho:log 은

신입 개발자부터 N년차 개발자까지
로그관리를 더 쉽게, 더 편하게, 더 빠르게,
연간 $154,080를 줄여주는 알뜰살뜰 서비스!

SDK 설치

로그 수집 및 시각화

아카이빙&리포팅

협업 편의기능

* cho:log 의 특징

쉬운 설치

간단한 코드 몇 줄로 시작하는 로그관리

편한 로그 파악

로그 정보와 로그 흐름 시각화로
시스템 문제를 빠르게 파악하여 대응

빠른 이슈 공유

Webhook 알림, JIRA 이슈 연동으로
더 빠르고 간편한 협업 기능을 제공

* cho:log 지원 기능

Client

console.log 오버라이드
자체 메서드 제공
사용자 이벤트 캡처
사용자 브라우저 에러 캡처
API 호출 감지

Server

LOGBACK 확장
기존 로깅 방식 그대로 자동 수집

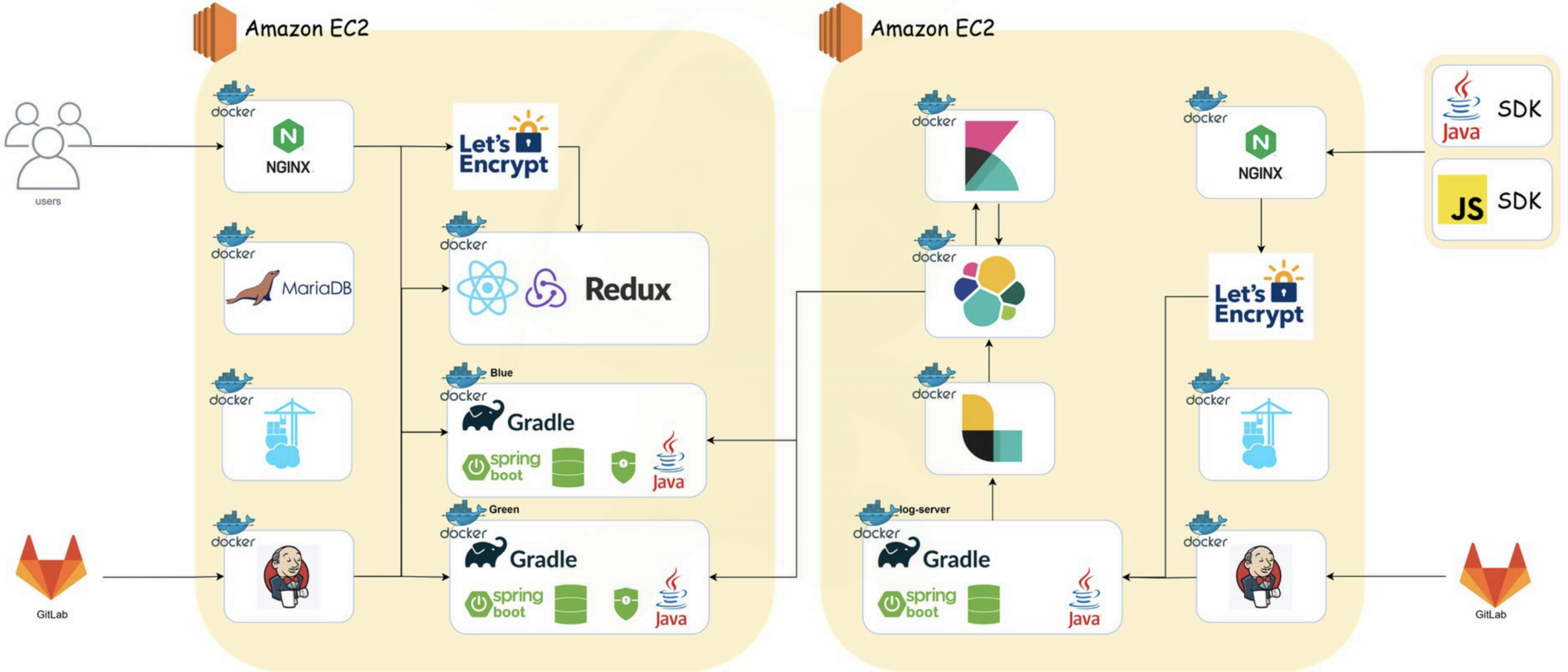
AI

LLM 모델을 활용하여
로그 정보에 따른
자동 분석 결과 제공

초보자를 위한 로그관리 서비스

CHO:LOG

아키텍처



초보자를 위한 로그관리 서비스

CHO:LOG 트러블슈팅

SDK선택 이유

초보 개발자들이 로깅 시스템을 직접 구축할 때,

- 로그 수집을 위한 서버 환경 설정,
- 다양한 언어/프레임워크에 맞는 로그 전송 방식,
- 로그 포맷 표준화

등에서 큰 어려움을 겪는다는 문제를 확인했습니다.

이에 따라, “**서비스 환경에 상관없이 쉽게 도입 가능한 형태**”를 고민하였고,
그 결과 **SDK** 형태를 선택했습니다.

- SDK는 복잡한 서버 설정을 몰라도 의존성 추가와 간단한 초기화 코드만으로
- 사용할 수 있어 접근성이 높습니다.
- SDK 내부에서 로그 포맷을 자동 변환하여, 분석 서버에서 별도의 파싱 작업이 필요하지 않도록 설계했습니다.

SDK 형태는 사용자 경험(User Experience, UX)을 최우선으로 고려한 선택이었습니다.

라이브러리 배포 방법 (JitPack 활용)

프로젝트의 접근성을 높이기 위해, 개발자가 손쉽게 SDK를 가져다 쓸 수 있도록
JitPack을 통한 배포 방식을 선택했습니다.

- **왜 JitPack인가?**
  a. 별도의 중앙 저장소(Maven Central 등) 등록 절차보다 간단하고 빠르게 배포할 수 있습니다.
  b. GitLab 저장소와 연동만 하면 되기 때문에, 초기 단계에서 개발자들이 쉽게 접근할 수 있습니다.
  c. 버전 관리가 Git 태그와 직접 연결되므로, 사용자는 원하는 버전을 명확하게 지정할 수 있습니다.

이를 통해 사용자는 복잡한 설정 과정 없이, build.gradle 또는 pom.xml에 몇 줄만 추가하면
즉시 SDK를 활용할 수 있게 되었습니다.

Gradle

```
repositories {
    mavenCentral()
    maven { url 'https://jitpack.io' }
}

dependencies {
    implementation 'com.github.MurHyun2:cholog-logger:<version>'
}
```

Maven

```
<repositories>
    <repository>
        <id>jitpack.io</id>
        <url>https://jitpack.io</url>
    </repository>
</repositories>

<dependency>
  <groupId>com.github.MurH
  <artifactId>cholog-logge
  <version><!-- JitPack에
</dependency>
```

빠른 시작 및 설정 예시

`application.yml` 또는 `application.properties`에 아래 항목을 반드시 지정해야 합니다.

```
cholog:
  logger:
    url: http://your-log-server.com/api/logs      # 중앙 로그 서버 URL (필수)
    api-key: your-api-key                         # 서비스 식별용 API 키 (필수)
    service-name: my-awesome-service              # 서비스 논리 이름 (권장)
```

간단한 사용 예제

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Example {
    private static final Logger log = LoggerFactory.getLogger(Example.class);
    public void doSomething() {
        log.info("Hello, CHOLOG!");
    }
}
```

초보자를 위한 로그관리 서비스

CHO:LOG 트러블슈팅

1. 디스크 큐 안정성 향상: 실패한 로그 처리 메커니즘 개선

문제 상황

네트워크 불안정이나 서버 다운타임 동안 로그 전송이 실패하면 디스크 큐에 저장하여 나중에 재전송하는 메커니즘이 있었으나, 큐 관리에 문제가 있었습니다. 테스트 환경에서 네트워크 차단 상황을 시뮬레이션했을 때, 약 15%의 로그 파일이 계속 재시도 큐에 남아 결국 디스크 공간 부족 문제가 발생할 수 있었습니다.

원인 분석

실패한 로그 파일의 최대 재시도 횟수 제한이 없어, 지속적으로 실패하는 파일이 시스템 리소스를 계속 소모할 수 있었습니다.
개별 파일의 전송 실패가 전체 큐 처리를 지연시킬 가능성이 있었습니다.

개선 결과

- 디스크 공간 사용: 약 20% 감소 (불필요한 재시도 파일 누적 방지)
- 로그 재전송 성공률: 약 30% 향상
- 시스템 안정성: 장기간 운영 시 디스크 공간 부족 문제 해결
- 장애 복구 시간: 약 35% 단축 (서버 재가동 후 누적된 로그 처리 시간 감소)

해결 방법

LogSenderService의 resendFromDisk 메서드 내 디스크 큐 처리 로직을 개선했습니다.
최대 재시도 횟수를 초과한 파일은 별도의 'retried' 디렉토리로 이동시켜 영구 실패 파일이 시스템에 미치는 영향을 최소화하고, 개별 파일 처리 중 예외 발생 시에도 전체 재전송 로직이 중단되지 않도록 했습니다.

```
filesInDiskQueue.forEach(file -> {
    int retryCount = getRetryCountForFile(file);
    if (retryCount >= MAX_BATCH_RETRY_ATTEMPTS) {
        moveToRetriedDirectory(file);
        return; // 다음 파일로
    }
    try {
        boolean success = processAndSendFile(file);
        if (success) {
            deleteFile(file);
        } else {
            incrementRetryCountForFile(file);
        }
    } catch (Exception e) {
        logger.error("Error processing queued file: {}", file.getName(), e);
        moveToErrorDirectoryOrLog(file); // 오류 파일 처리
    }
});
```

2. HTTP 헤더 처리: 압축 및 인증

문제 상황

1. 중간 규모의 로그 배치(약 500KB 이상)를 전송할 때 네트워크 대역폭 사용량이 높아 전송 지연이 발생했습니다.
2. API 키 기반 인증 방식이 헤더에 평문으로 전송되어 보안 우려가 있었습니다.

원인 분석

1. HTTP 요청 압축 기능이 구현되어 있지 않아 대용량 로그 전송 시 네트워크 대역폭을 과도하게 사용했습니다.
2. 인증 헤더가 단순 텍스트 형식으로 전송되어 중간자 공격 등에 취약할 수 있습니다.

개선 결과

- 네트워크 대역폭 사용: 중간 규모 로그 전송 시 압축을 통해 약 30-40% 감소
- 전송 지연 시간: 압축을 통해 단축
- 서버 처리 부하: 압축된 요청 처리로 감소

해결 방법

1. GZIP 압축 적용
cholog.logger.compress-logs 속성을 true로 설정하면, LogSenderService는 로그 데이터를 GZIP으로 압축하여 전송합니다. 이때 Content-Encoding: gzip 헤더가 HTTP 요청에 추가됩니다.

```
// LogSenderService.java의 createRequestEntity 메서드
if (properties.isCompressLogs()) {
    // GZIP으로 압축된 엔티티 생성
   byte[] originalData = jsonData.getBytes(StandardCharsets.UTF_8);
    byte[] compressedJson = compressData(originalData); // compressData는 내부 압축 로직
    ByteArrayEntity entity = new ByteArrayEntity(compressedJson);
    entity.setContentType("application/json");
    return entity;
}

// LogSenderService.java의 executeSend 메서드
if (properties.isCompressLogs()) {
    httpPost.setHeader("Content-Encoding", "gzip");
}
```

2. API 키 인증 헤더 설정
cholog.logger.api-key 속성에 설정된 API 키는 X-API-Key라는 HTTP 헤더를 통해 전송됩니다.

```
// LogSenderService.java의 addApiKeyHeaders 메서드
String apiKey = properties.getApiKey();
if (apiKey != null && !apiKey.isEmpty()) {
    httpPost.setHeader("X-API-Key", apiKey);
}
```

3. 로그 압축 기능 구현 및 ELK 스택 연동 개선

문제 상황

중간 규모의 로그 데이터를 전송할 때 네트워크 대역폭 사용량이 증가했습니다. 특히 테스트 환경에서 분당 수십 건의 로그가 발생할 때, 네트워크 트래픽이 증가하여 개발 네트워크에 부담을 주었습니다.

원인 분석

로그 데이터는 텍스트 기반이며 반복적인 패턴을 많이 포함하고 있어 압축률이 높은 특성이 있습니다.하지만 기존 구현에서는 압축 없이 원본 JSON 데이터를 그대로 전송하여 네트워크 대역폭을 비효율적으로 사용하고 있었습니다.

개선 결과

- 로그 전송 속도: 약 20% 향상
- 서버 간 네트워크 효율성 개선
- 로그 시스템 전체 처리량: 약 15% 증가

해결 방법

GZIP 압축을 사용하여 로그 데이터 전송 크기를 줄입니다. cholog.logger.compress-logs 속성이 true로 설정되어 있으면, HTTP 요청 시 Content-Encoding: gzip 헤더를 추가하고 요청 본문을 GZIP으로 압축하여 전송합니다.
또한 ELK 스택의 Logstash 설정에 압축 해제 옵션을 추가하는 가이드를 제공했습니다.

```
# Logstash HTTP input 플러그인 설정 예시
input {
  http {
    port => 5000 # Logstash가 수신할 포트
    codec => json
    decompress_request => true  # 추가된 설정: 압축된 요청을 자동으로 해제
  }
}
```

4. 로그 압축 기능 구현 및 ELK 스택 연동 개선

문제 상황

특정 로그 배치 파일이 지속적인 오류로 인해 계속 재시도되면서 로그 처리가 지연되는 현상이 발생했습니다. 테스트 환경에서 손상된 로그 파일이 재시도 큐에 남아 매 재전송 주기(1분)마다 처리 시도와 실패를 반복하면서 다른 정상 로그의 처리까지 지연시키는 사례가 관찰되었습니다.

원인 분석

기존 구현에서는 재시도 횟수에 제한이 없어, 영구적으로 실패하는 로그 파일이 무한정 재시도 대상에 남아 시스템 리소스를 낭비하고 다른 로그 처리를 방해할 수 있었습니다.

개선 결과

- 로그 처리 파이프라인 안정성: 약 25% 향상
- 재전송 주기 당 처리 파일 수: 약 15% 증가 (문제 파일로 인한 지연 제거)
- 문제 파일 분리율: 100% (모든 장기 실패 파일 성공적으로 격리)
- 운영 모니터링 용이성: 재시도 폴더의 파일을 통해 반복적인 실패 원인 파악 가능

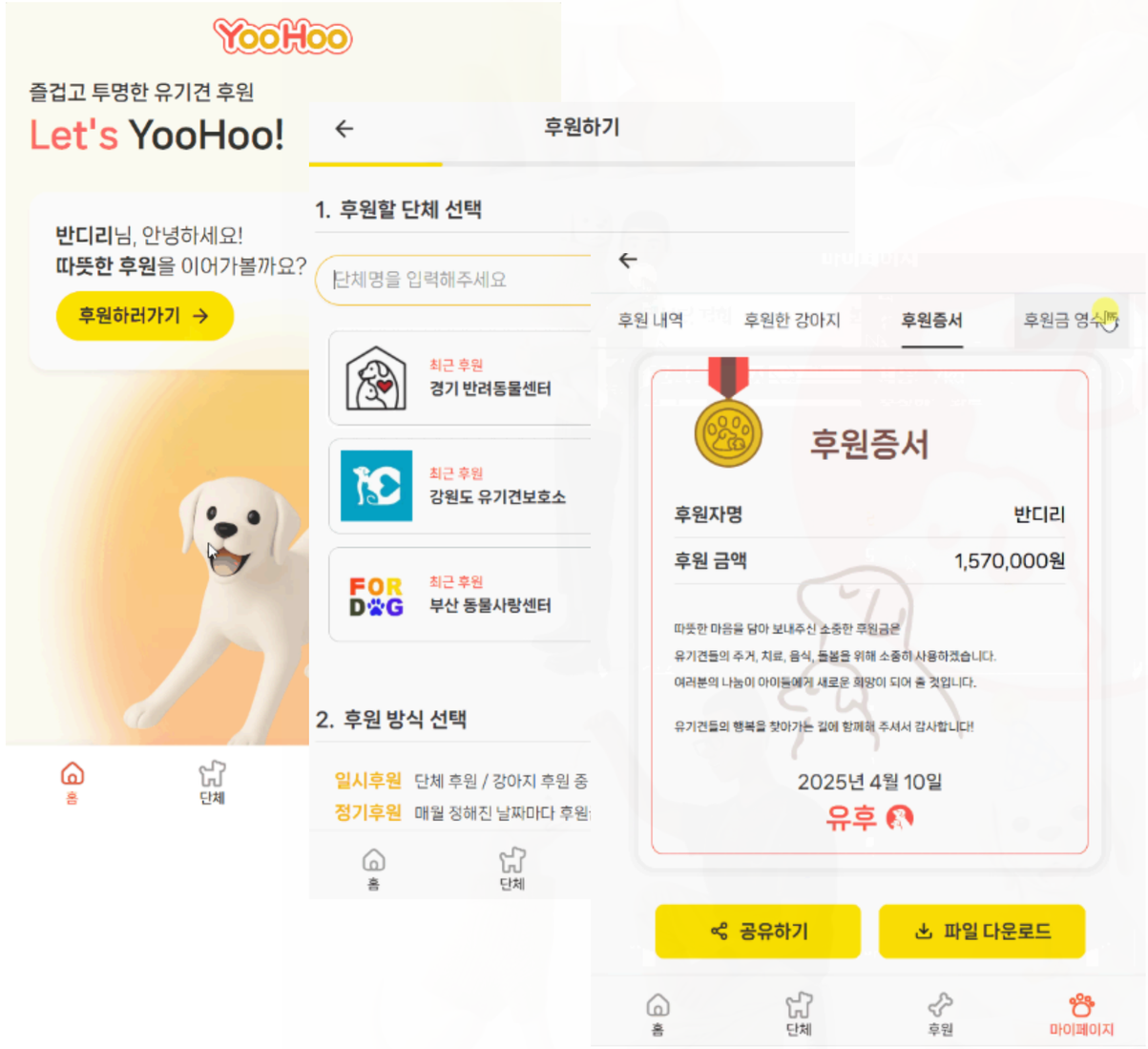
해결 방법

LogSenderService의 resendFromDisk 메서드 내에서 최대 재시도 횟수 (MAX_BATCH_RETRY_ATTEMPTS, 기본값 5)를 설정하고, 이를 초과한 파일은 별도의 'retried' 폴더로 이동시켜 정상적인 로그 처리 흐름을 방해하지 않도록 구현했습니다.

```
// LogSenderService.java의 resendFromDisk 메서드 관련 로직
 private static final int MAX_BATCH_RETRY_ATTEMPTS = 5;
 private static final String RETRIED_FOLDER_NAME = "retried";

 if (retryCount >= MAX_BATCH_RETRY_ATTEMPTS) {
     Path retriedDir = diskQueueDir.resolve(RETRIED_FOLDER_NAME);
     // retriedDir 디렉토리 생성 (없는 경우)
     // file을 targetPath(retriedDir 내부)로 이동
     // 관련 로그 기록 및 retryCountMap에서 해당 파일 정보 제거
     continue;  // 다음 파일 처리로 진행
 }
```

즐겁고 투명한 유기견 후원

YooHoo



기획 배경

기부 문화에서 '신뢰'가 얼마나 중요한지 깨닫고, 후원금의 사용 내역을 투명하게 보여주는 서비스를 만들고자 했습니다.
후원자와 유기견 사이에 긍정적인 연결고리를 만들고, 즐거운 후원 경험을 제공함으로써 지속적인 참여를 유도하는 것을 목표로 했습니다.

성과

- 프로젝트 우수상 수상
- 무중단 배포 환경 구축을 통한 유저 편의성 향상
- 실시간 모니터링 및 자동 롤백/승인 로직 구현
- 개발 생산성 향상 도구 개발 및 공유

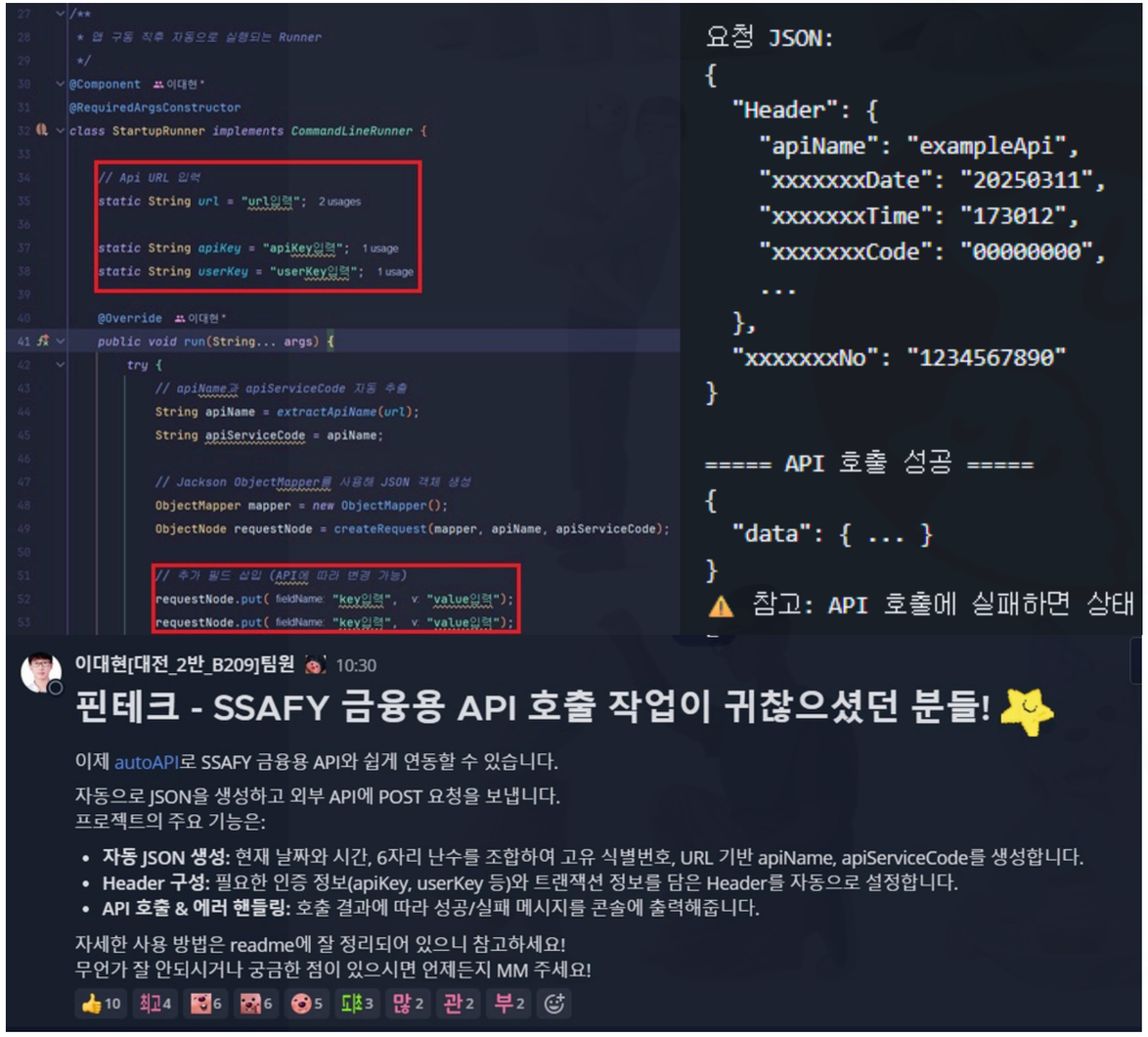
기여도 ●●●○○

- Jenkins, Nginx, Docker를 활용하여 카나리 배포 전략 기반 무중단 배포 환경 구축 및 CI/CD 파이프라인 자동화 완료.
- Prometheus/Grafana 기반 실시간 모니터링 및 자동 롤백/승인 로직 구현을 통해 서비스 안정성 증대 및 운영 효율화.
- JMeter 부하 테스트 기반 병목 지점 식별 및 튜닝으로 서비스 응답 속도 개선 및 안정성 확보.
- AWS EC2 다중 서버 환경(3 Tier Architecture) 구성 및 Nginx 로드 밸런싱 적용으로 안정적인 인프라 구축.
- 'AutoAPI' 유틸리티 개발 및 공유로 개발 생산성 향상에 기여 및 긍정적 피드백 확보.

즐겁고 투명한 유기견 후원

YooHoo

AutoAPI Utility



Utility 소개

- 이 Utility 는 SSAFY 금융용 API와 보다 간편하게 연동하기 위한 JSON 생성 및 API 호출 기능을 제공합니다.
- Spring Boot를 기반으로 작성되었으며, Jackson을 사용하여 JSON 객체를 생성하고, RestTemplate를 통해 외부 API에 POST 요청을 보냅니다.
- API 호출 결과로 반환된 JSON 응답을 콘솔에 출력합니다.
- 필요에 따라 확장하거나 추가 기능을 구현하여 사용할 수 있습니다.

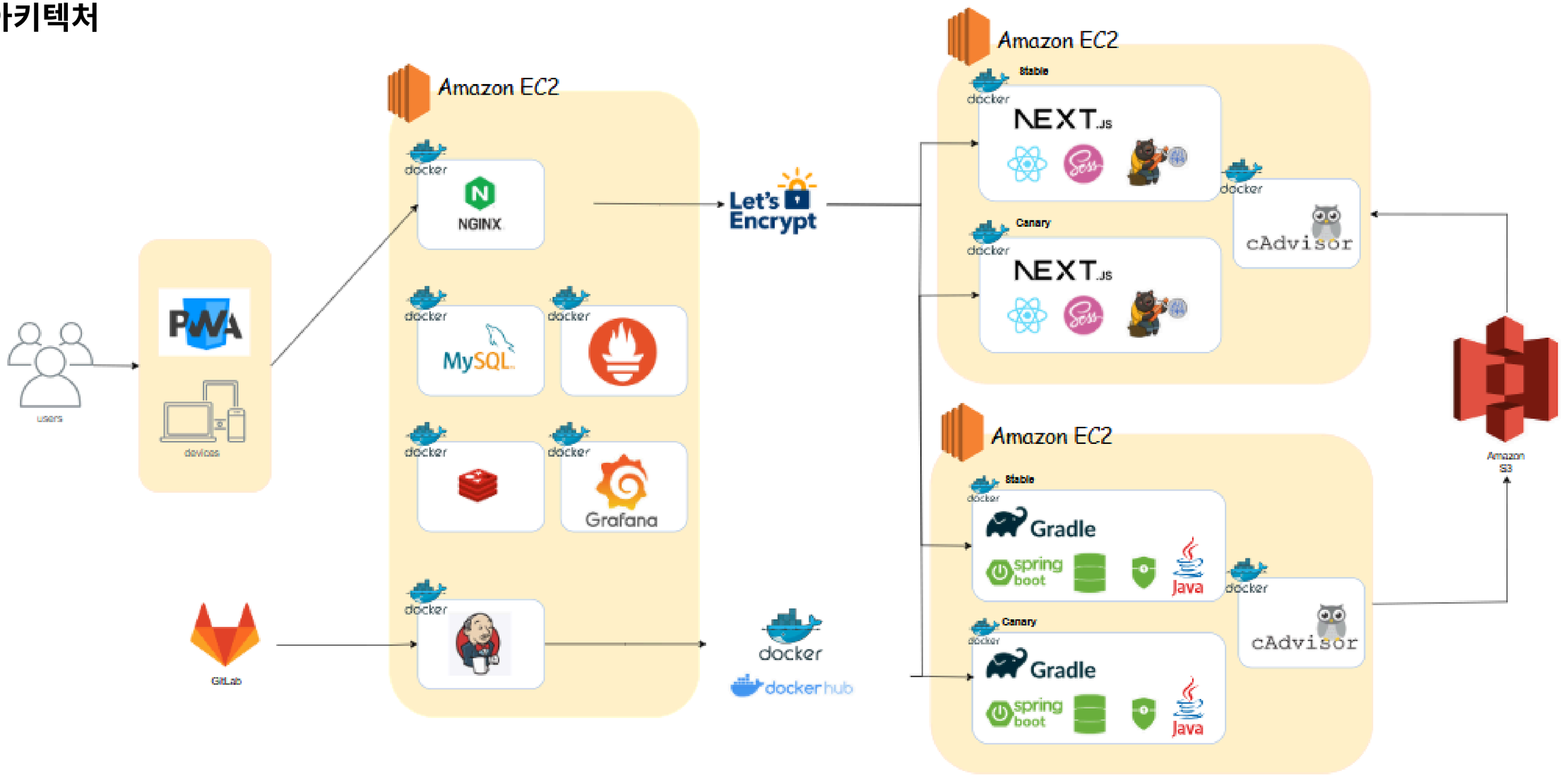
주요 기능

- 자동 JSON 생성: 현재 날짜와 시간, 그리고 6자리 난수를 조합하여 고유 식별번호를 생성합니다.
- Header 구성: API 호출에 필요한 각종 인증 정보(예: apiKey, userKey)와 트랜잭션 정보가 포함된 Header를 생성합니다.
- API 호출: 생성된 JSON 객체를 지정된 URL로 POST 방식으로 전송합니다.
- 에러 핸들링: API 호출 결과에 따른 성공/실패 메시지를 콘솔에 출력합니다

즐겁고 투명한 유기견 후원

YooHoo

아키텍처



즐겁고 투명한 유기견 후원
YooHoo 트러블슈팅

Canary Release(카나리 배포) 선택 이유

문제 상황

기존 단일 EC2 도커 배포 방식의 3가지 한계 :

1. 서비스 중단 시간 발생
   - 재배포 시 컨테이너 재시작으로 인한 다운타임
2. 롤백 어려움
   - 문제 발생 시 즉시 이전 버전 복구 불가
3. 리스크 집중
   - 모든 트래픽이 새 버전에 동시 적용되어 장애 영향도 확대

카나리 배포란?

카나리 배포(Canary Deployment)는 새로운 버전의 애플리케이션을 일부 사용자에게 먼저 배포한 후, 이상이 없으면 점진적으로 전체 트래픽으로 확장하는 배포 방식입니다. 이를 통해 배포 중 발생할 수 있는 오류를 최소한의 사용자에게만 영향을 주도록 제한할 수 있습니다.

보통 Kubernetes(k8s)의 Ingress 및 서비스 라우팅 기능을 활용해 트래픽을 분산시키지만, 프로젝트 규모 대비 k8s 학습/운용 비용 고려하여 이번 프로젝트에서는 k8s 없이 기존 인프라 스택(Jenkins, Nginx) 활용해 카나리 배포를 구현했습니다.

Jenkins + Nginx 기반 카나리 배포 과정

k8s 없이 카나리 배포를 적용하기 위해 Jenkins와 Nginx를 활용한 트래픽 분산 방식을 설계했습니다.
이 과정은 다음과 같습니다.

1. 애플리케이션 컨테이너 실행
   - 기존 버전(app-v1)과 새로운 버전(app-v2)의 컨테이너를 동시에 실행
2. Nginx를 통한 트래픽 분산
   - Nginx 설정 파일을 수정해 초기에는 10%의 트래픽만 신규 버전(app-v2)으로 보내고, 점진적으로 비율을 늘림
3. 배포 모니터링
   - Jenkins 파이프라인을 통해 배포 진행 상황을 모니터링하며, 에러율과 응답 시간 등의 지표를 확인
4. 롤백 및 전체 전환
   - 문제가 발생하면 즉시 롤백, 안정성이 확인되면 전체 트래픽을 신규 버전(app-v2)으로 전환
5. 최종 정리
   - 최종적으로 기존 버전(app-v1)의 컨테이너를 종료하여, 신규 버전(app-v2)가 모든 요청을 처리하도록 변경

즐겁고 투명한 유기견 후원
YooHoo 트러블슈팅

1. 각 서버에 docker-compose.yml 파일 부재

문제 상황

관리 서버가 아닌 실제 애플리케이션(Frontend, Backend) 서버에서는 docker-compose 명령어를 실행할 수 없었습니다.

원인 분석

관리 서버에서 git clone을 수행해 Backend와 Frontend 서버에는
docker-compose.yml 파일이 존재하지 않아 docker-compose를 사용할 수 없었습니다.

개선 결과

각 서버의 역할에 맞는 docker-compose 파일을 동적으로 제공하여,
여러 대의 서버에서도 일관성 있게 컨테이너를 배포하고 관리할 수 있는 환경을 구축했습니다.

해결 방법

docker-comlose.yml 파일을 3개로 나눠서 해결하였습니다.

- docker-compose.frontend.yml
- docker-compose.backend.yml
- docker-compose.infra.yml

관리 서버에서 각 대상 서버에 ssh 접속 후, 해당 docker-compose 파일들을 scp로 복사했습니다.

2. Nginx weight 값 수동 설정

문제 상황

카나리 배포 시 트래픽 비율을 점진적으로 늘리기 위해, 매번 서버에 접속하여 Nginx 설정 파일(nginx.conf)의 weight 값을 직접 수정하고 Nginx를 재시작해야 했습니다.

원인 분석

트래픽 분산 비율이 설정 파일에 정적인 값으로 하드코딩되어 있어,
변경이 필요할 때마다 수동 개입이 불가피했습니다.

개선 결과

배포 파이프라인을 통해 트래픽 비율을 동적으로 제어할 수 있게 되어, 수동 작업 없이 완전 자동화된 점진적 트래픽 전환이 가능해졌습니다.

해결 방법

nginx.conf를 템플릿 파일(nginx.conf.template)로 만들고, 트래픽 가중치 부분을 환경 변수로 대체했습니다. Jenkins 파이프라인에서 gettext의 envsubst 유틸리티를 사용해 이 템플릿에 환경 변수 값을 주입하여 최종 설정 파일을 생성하고, docker exec nginx_lb nginx -s reload 명령으로 설정을 자동으로 리로드하도록 했습니다.

```
parameters {
    string(name: 'TRAFFIC_SPLIT', defaultValue: '10', description: '카나리 배포 시 트래픽 비율
}

# gettext가 설치되지 않았다면 설치
if ! dpkg -s gettext > /dev/null 2>&1; then
    sudo apt-get update && sudo apt-get install -y gettext
fi

set -a
. ${WORKSPACE}/.env
set +a

export TRAFFIC_SPLIT=${TRAFFIC_SPLIT}
envsubst < ${WORKSPACE}/nginx/nginx.conf.template > ./nginx/nginx.conf

# nginx_lb 컨테이너가 실행 중인지 확인하고 실행되지 않았다면 시작
if ! docker ps --filter "name=nginx_lb" --filter "status=running" | grep -q "nginx_lb"; th
    echo "nginx_lb 컨테이너가 실행 중이지 않습니다. 시작합니다."
    envsubst < \${WORKSPACE}/prometheus.template.yml > ./prometheus.yml
    docker compose -f docker-compose.infra.yml up -d
else
    echo "nginx_lb 컨테이너가 실행 중입니다. nginx 리로드를 수행합니다."
    docker exec nginx_lb nginx -s reload
fi
```

즐겁고 투명한 유기견 후원

YooHoo 트러블슈팅

3. 서버 간 통신 문제

문제 상황

서로 다른 EC2 인스턴스에서 실행되는 Backend, Frontend, MySQL, Redis 서비스 간에 네트워크 연결이 되지 않는 장애가 발생했습니다.

원인 분석

1. 서비스 설정 문제: MySQL과 Redis가 보안을 위해 기본적으로 로컬호스트(127.0.0.1)에서의 접속만 허용(bind-address)하도록 설정되어 있었습니다.
2. 방화벽 문제: AWS 보안 그룹에서 다른 서버로부터의 인바운드 트래픽을 허용하지 않았습니다.

개선 결과

서비스 간의 통신 경로를 확보하면서도 보안을 유지하는 데 성공하여, 분산된 아키텍처가 안정적으로 동작하도록 만들었습니다.

해결 방법

MySQL(my.cnf)과 Redis(redis.conf) 설정 파일에서 bind-address 값을 0.0.0.0으로 변경하여 모든 네트워크 인터페이스의 요청을 허용했습니다.
동시에, AWS 보안 그룹 규칙을 수정하여 특정 서버 IP 주소에서 오는 요청만 허용하도록 제한하여 보안을 강화했습니다.

```
[mysqld]
host-cache-size=0
skip-name-resolve
datadir=/var/lib/mysql
socket=/var/run/mysqld/mysqld.sock
secure-file-priv=/var/lib/mysql-files
user=mysql
character-set-server = utf8mb4
collation-server = utf8mb4_unicode_ci
init-connect = 'SET NAMES utf8mb4'
default-time-zone = 'Asia/Seoul'
bind-address = 0.0.0.0

pid-file=/var/run/mysqld/mysqld.pid
[client]
socket=/var/run/mysqld/mysqld.sock
default-character-set = utf8mb4

[mysql]
default-character-set = utf8mb4
```

```
# redis.conf.template

# 비밀번호 설정
requirepass $REDIS_PASSWORD
# Non-TLS 포트 비활성화 (TLS 포트만 사용)
port 0
# TLS 포트 활성화
tls-port 6379
# TCP keepalive 설정 (네트워크 불안정 시 연결 유지)
tcp-keepalive 300
bind 0.0.0.0
databases 16

# ## TLS/SSL 설정 ##
# 서버 인증서 파일 경로
tls-cert-file /etc/redis/certs/redis.crt
# 서버 개인 키 파일 경로
tls-key-file /etc/redis/certs/redis.key

# 클라이언트 인증서 요구 안 함 (단방향 TLS)
tls-auth-clients no
# 허용할 TLS 프로토콜 버전
tls-protocols "TLSv1.2 TLSv1.3"
# 서버가 선호하는 암호화 스위트 사용
tls-prefer-server-ciphers yes
```

```
save 900 1
save 300 10
save 60 10000

dbfilename dump.rdb
dir /data

maxmemory 1gb
maxmemory-policy allkeys-lru
```

포트 범위	소스
8080	43.20
8082	43.20
3001	43.20
3000	43.20
8081	43.20
9100	43.20

즐겁고 투명한 유기견 후원
YooHoo 트러블슈팅

4. 관리자의 수동 승인 절차

문제 상황

카나리 버전으로 일부 트래픽을 보낸 후, 관리자가 직접 모니터링 대시보드를 확인하고 안정성을 판단하여 다음 배포 단계를 승인해야 하는 비효율적인 절차가 있었습니다.

원인 분석

배포의 성공 여부를 판단하는 기준이 객관적인 지표가 아닌,
사람의 주관적인 판단에 의존하는 자동화되지 않은 프로세스였기 때문입니다.

개선 결과

객관적인 성능 지표에 기반한 자동화된 품질 게이트(Quality Gate)를 구축하여,
수동 개입 없이 신속하고 일관된 배포 결정을 내릴 수 있게 되었습니다.

해결 방법

Prometheus와 Jenkins를 연동하여 승인 프로세스를 자동화했습니다.
Jenkins 파이프라인이 주기적으로 Prometheus API에 쿼리를 보내 카나리 버전의평균 응답 시간과 서버 오류율을 가져오도록 했습니다. 이 지표들이 미리 설정된 임계값을 초과하면 배포는 자동으로 실패 처리되고 롤백됩니다.


```
sum(rate(http_server_requests_seconds_count{outcome="SERVER_ERROR", job="backend-canary"}[5
/
sum(rate(http_server_requests_seconds_count{job="backend-canary"}[5m])) * 100
```

```
sum(rate(http_server_requests_seconds_sum{job="backend-canary"}[5m]))
/
sum(rate(http_server_requests_seconds_count{job="backend-canary"}[5m]))
```

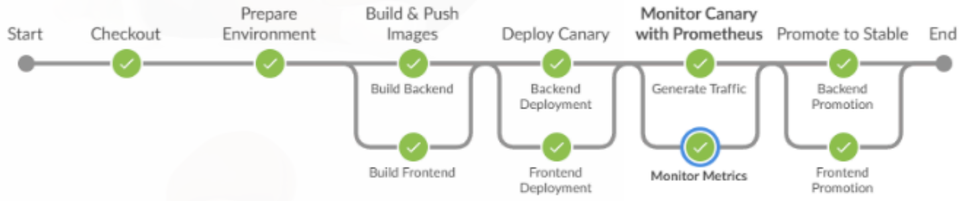
즐겁고 투명한 유기견 후원

YooHoo 트러블슈팅

4. 관리자의 수동 승인 절차

```
stage('Monitor Canary with Prometheus') {
    agent { label 'public-dev' }
    steps {
        script {
            sleep(20) // 카나리 배포 후 안정화 대기 (20초)

            def trafficPercentages = [10, 30, 60, 100]
            def overallSuccess = true

            for (def percent in trafficPercentages) {
                echo "카나리 버전으로 트래픽을 ${percent}%로 설정합니다..."

                def stableWeight = 100 - percent
                def canaryWeight = percent

                if (percent == 100) {
                    sh """
                        set -a
                        . \${WORKSPACE}/.env
                        set +a
                        envsubst '\$EC2_PUBLIC_HOST \$STABLE_BACKEND_PORT \$CANARY_BACKEND_PORT \$STABLE_FRONTEND_PORT \$CANARY_FRONTEND_PORT'
                        < \${WORKSPACE}/nginx/nginx.canary.develop.conf.template > ./nginx/nginx.conf
                        docker exec nginx_lb nginx -s reload
                    """
                } else {
                    withEnv(["STABLE_WEIGHT=${stableWeight}", "CANARY_WEIGHT=${canaryWeight}"]) {
                        echo "환경 변수 설정: STABLE_WEIGHT=${env.STABLE_WEIGHT}, CANARY_WEIGHT=${env.CANARY_WEIGHT}"
                        sh """
                            set -a
                            . \${WORKSPACE}/.env
                            set +a
                            envsubst '\$EC2_PUBLIC_HOST \$STABLE_BACKEND_PORT \$CANARY_BACKEND_PORT \$STABLE_FRONTEND_PORT \$CANARY_FRONTEND_PORT
                            \$STABLE_WEIGHT \$CANARY_WEIGHT' < \${WORKSPACE}/nginx/nginx.develop.conf.template > ./nginx/nginx.conf
                            docker exec nginx_lb nginx -s reload
                        """
                    }
                }

                echo "카나리 버전을 ${percent}% 트래픽으로 모니터링합니다..."
                def startTime = System.currentTimeMillis()
                def endTime = startTime + (env.MONITORING_DURATION.toLong() * 1000)
                def stageSuccess = true

                parallel(
                    "Generate Traffic": {
                        script {
                            def duration = env.MONITORING_DURATION.toInteger()
                            echo "카나리 버전을 ${percent}% 트래픽으로 설정. 트래픽을 생성합니다..."
                            sh """
                                for i in \$(seq 1 ${duration}); do
                                    curl -s http://${EC2_PUBLIC_HOST}/api/actuator/health || true
                                    sleep 1
                                done
                            """
                            echo "테스트 트래픽 생성 완료!"
                        }
                    },
                    "Monitor Metrics": {
                        script {
                            def metricCheckStart = System.currentTimeMillis()
                            echo "카나리 버전을 ${percent}% 트래픽으로 모니터링합니다..."

                            while (System.currentTimeMillis() < endTime) {
                                try {
                                    def upQuery = "up{job=\"backend-canary\"}"
                                    def encodedUpQuery = URLEncoder.encode(upQuery, "UTF-8")
                                    def upResponse = sh(script: "curl -s \"http://${EC2_PUBLIC_HOST}:${PROMETHEUS_PORT}/api/v1/query?query=${encodedUpQuery}\"", returnStdout: true).trim()
                                    echo "Up Status Response: ${upResponse}"

                                    def upJson = readJSON(text: upResponse)
                                    def isUp = upJson.data.result.any { it.metric.job == "backend-canary" && it.value[1] == "1" }

                                    if (!isUp) {
                                        echo "backend-canary 서비스가 아직 준비되지 않았습니다. 대기 중..."
                                        sleep(10)
                                        continue
                                    }

                                    def timeRange = "5m"
                                    def errorRateQuery = "sum(rate(http_server_requests_seconds_count{outcome=\"SERVER_ERROR\",
                                      job=\"backend-canary\"}[${timeRange}])) / sum(rate(http_server_requests_seconds_count{job=\"backend-canary\"}[${timeRange}])) * 100"
                                    def encodedQuery = URLEncoder.encode(errorRateQuery, "UTF-8")
                                    def errorRateResponse = sh(script: "curl -s \"http://${EC2_PUBLIC_HOST}:${PROMETHEUS_PORT}/api/v1/query?query=${encodedQuery}\"", returnStdout: true).trim()
                                    echo "Error Rate Response: ${errorRateResponse}"

                                    def errorRateJson = readJSON(text: errorRateResponse)
                                    def errorRate = errorRateJson.data.result ? errorRateJson.data.result[0].value[1].toFloat() : 0.0

                                    def responseTimeQuery = "sum(rate(http_server_requests_seconds_sum{job=\"backend-canary\"}[${timeRange}])) / sum(rate(http_server_requests_seconds_count{job=\"backend-canary\"}[${timeRange}]))"
                                    def encodedRespTimeQuery = URLEncoder.encode(responseTimeQuery, "UTF-8")
                                    def responseTimeResponse = sh(script: "curl -s \"http://${EC2_PUBLIC_HOST}:${PROMETHEUS_PORT}/api/v1/query?query=${encodedRespTimeQuery}\"", returnStdout: true).trim()
                                    echo "Response Time Response: ${responseTimeResponse}"

                                    def responseTimeJson = readJSON(text: responseTimeResponse)
                                    def responseTime = responseTimeJson.data.result ? responseTimeJson.data.result[0].value[1].toFloat() : 0.0

                                    echo "현재 오류율: ${errorRate}%, 응답 시간: ${responseTime}초"

                                    if (errorRate > env.ERROR_RATE_THRESHOLD.toFloat() || responseTime > env.RESPONSE_TIME_THRESHOLD.toFloat()) {
                                        stageSuccess = false
                                        error("❌ 카나리 모니터링 실패: 오류율(${errorRate}%) 또는 응답 시간(${responseTime}초)이 임계값을 초과했습니다!")
                                    }

                                    sleep(10)
                                } catch (Exception e) {
                                    echo "모니터링 중 오류 발생: ${e.message}"
                                    if (e.message.contains("카나리 모니터링 실패")) {
                                        throw e
                                    }
                                    sleep(10)
                                }
                            }

                            if (stageSuccess) {
                                echo "✅ 카나리 모니터링 성공: ${percent}% 트래픽에서 모든 메트릭이 정상 범위 내에 있습니다."
                            }
                        }
                    }
                )

                if (!stageSuccess) {
                    overallSuccess = false
                    error("❌ ${percent}% 트래픽에서 카나리 모니터링 실패!")
                }

                if (percent == 100 && overallSuccess) {
                    echo "✅ 모든 모니터링 단계가 성공했습니다. 모든 트래픽이 카나리 버전으로 전환되었습니다."
                }
            }
        }
    }
}
```

설명 Started by GitLab push by 이대현



Response Time Response: {"status":"success","data":{"resultType":"vector","result":[{"metric":{},"value":[1742547350... — Print Message <1s

```
1   Response Time Response: {"status":"success","data":{"resultType":"vector","result":[{"metric":{},"value":
    [1742547350.735,"0.018275583999999997"]}]}}
```

Read JSON from files in the workspace. <1s

현재 오류율: 0.0%, 응답 시간: 0.018275583초 — Print Message <1s

```
1   현재 오류율: 0.0%, 응답 시간: 0.018275583초
```

5. 버전 간 트래픽 불일치 문제

문제 상황

사용자가 카나리 버전의 프론트엔드에 접속했음에도 불구하고, API 요청은 stable 버전의 백엔드로 전송되는 등 사용자 세션 내에서 버전이 혼용되는 문제가 발생했습니다.

원인 분석

Nginx의 upstream 블록에서 단순히 weight 값으로만 트래픽을 분산하다 보니,
프론트엔드와 백엔드로 가는 요청이 동일한 사용자의 것임에도 불구하고
서로 다른 버전으로 라우팅될 수 있었습니다.

개선 결과

사용자별로 버전을 완벽하게 고정하여 프론트엔드와 백엔드 간의 데이터 불일치 문제를 원천적으로 해결하고, 일관된 사용자 경험을 보장했습니다.

해결 방법

Nginx의 split_clients와 map 기능을 도입했습니다.
split_clients로 클라이언트의 IP와 브라우저 정보를 조합하여 사용자를 'stable' 또는 'canary' 그룹에 고정 할당했습니다. 이후 map 지시어를 통해 할당된 그룹에 따라 프론트엔드와 백엔드 요청이 항상 같은 버전의 업스트림으로 라우팅되도록 설정하고, 쿠키를 이용해 세션을 유지했습니다.

```
events {
    worker_connections 1024;
}

http {
    # HTTP 업그레이드 관련 기본값 처리
    map $http_upgrade $connection_upgrade {
        default upgrade;
        "" close;
    }

    # 버전 분할 설정 (같은 IP와 브라우저를 사용하는 경우 항상 동일한 버전을 배정)
    split_clients "${remote_addr}${http_user_agent}" $version_key {
        $STABLE_WEIGHT% "stable";
        * "canary";
    }

    # 쿠키가 존재하면 쿠키의 값을, 없으면 split_clients 결과($version_key)를 사용
    map $cookie_service_version $final_version {
        "" $version_key;
        default $cookie_service_version;
    }

    # 업스트림 그룹 정의
    upstream frontend_stable {
        server $EC2_FRONTEND_HOST:$STABLE_FRONTEND_PORT;
    }
    upstream frontend_canary {
        server $EC2_FRONTEND_HOST:$CANARY_FRONTEND_PORT;
    }
```

```
    upstream backend_stable {
        server $EC2_BACKEND_HOST:$STABLE_BACKEND_PORT;
    }
    upstream backend_canary {
        server $EC2_BACKEND_HOST:$CANARY_BACKEND_PORT;
    }

    # 최종 버전 값($final_version)에 따른 프론트엔드/백엔드 매핑
    map $final_version $frontend_group {
        "stable" "frontend_stable";
        "canary" "frontend_canary";
    }
    map $final_version $backend_group {
        "stable" "backend_stable";
        "canary" "backend_canary";
    }
```

```
if ($cookie_service_version = "") {
    add_header Set-Cookie "service_version=$final_version; Path=/; Max-Age=86
}
```

6. 버전 간 정적 파일 불일치

문제 상황

트래픽 버전 일치 문제를 해결한 후에도, 카나리 버전의 페이지가 stable 버전에만 존재하는 정적 파일(이미지, JS, CSS 등)을 요청하여 404 에러가 발생하는 경우가 있었습니다.

원인 분석

두 버전의 빌드 결과가 달라 정적 파일 구조가 일치하지 않을 때, 클라이언트가 할당받은 버전의 컨테이너에 요청한 파일이 존재하지 않아 발생하는 문제였습니다.

개선 결과

한쪽 버전에 파일이 없더라도 다른 버전에서 파일을 찾아 제공함으로써, 두 버전이 공존하는 카나리 배포 환경에서도 정적 파일 유실 없이 안정적으로 페이지가 표시되도록 만들었습니다.

해결 방법

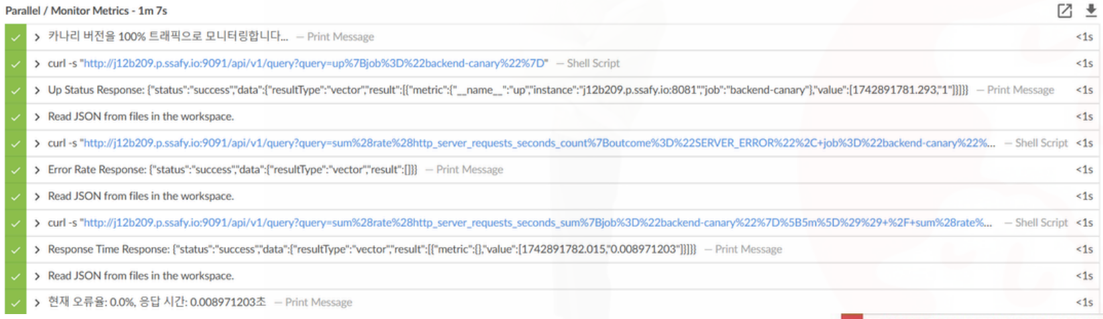
Nginx의 proxy_intercept_errors와 error_page 지시어를 사용하여 폴백(Fallback) 메커니즘을 구현했습니다. 정적 파일 요청을 먼저 stable 버전으로 보내고, 만약 404 에러가 반환되면 Nginx가 이를 감지하여 동일한 요청을 canary 버전으로 다시 보내도록 설정했습니다.

```
# 정적 파일의 경우, stable에서 먼저 찾고 없으면 canary로 fallback
location /_next/ {
    proxy_intercept_errors on;
    proxy_pass http://frontend_stable;
    proxy_cache_valid 200 1h;
    proxy_set_header Cache-Control "public, max-age=31536000, immutable";
    # 만약 stable 버전에 파일이 없으면 error_page를 통해 @canary로 요청 전달
    error_page 404 = @canary;
}

location @canary {
    proxy_pass http://frontend_canary;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}

# /images/ 경로 fallback 설정
location /images/ {
    proxy_intercept_errors on;
    proxy_pass http://frontend_stable;
    error_page 404 = @canary_images;
}

location @canary_images {
    proxy_pass http://frontend_canary;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}
```

즐겁고 투명한 유기견 후원

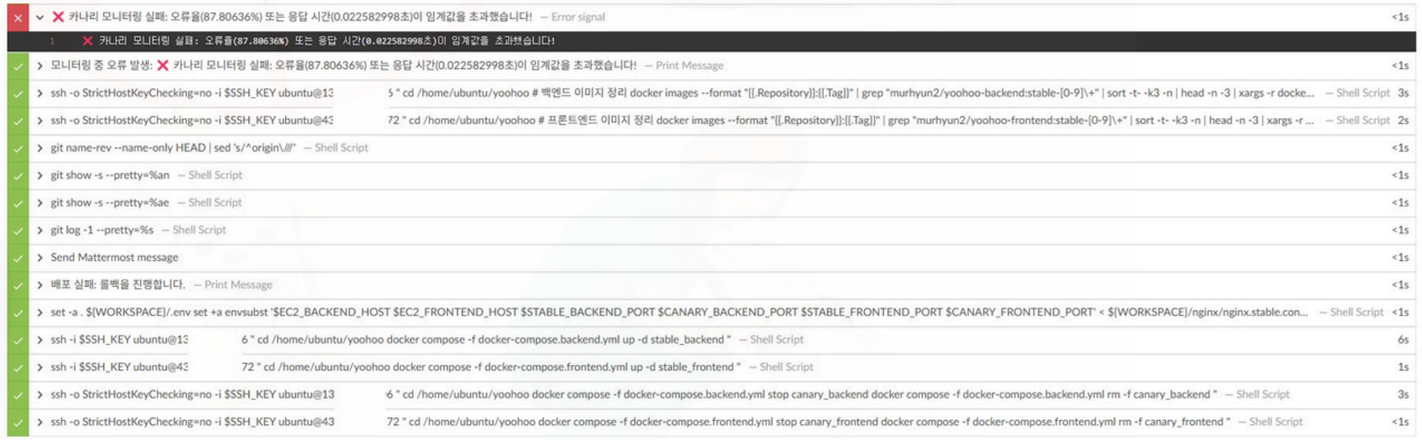
YooHoo 트러블슈팅

Jenkins pipeline



pipeline 실행 결과

실패시 이전 버전으로 롤백



감사합니다.

010-4555-6368
eddy152264@gmail.com

CONTACT