

|                                     |
|-------------------------------------|
| 목차                                  |
| 개요                                  |
| 환경                                  |
| Project Architecture                |
| GitHub Actions                      |
| - GitHub Secrets                    |
| - github-actions.yml                |
| AWS (Amazon Web Services)           |
| - EC2 (Elastic Compute Cloud)       |
| - Route 53                          |
| - RDS (Relational Database Service) |
| Docker                              |
| Nginx                               |
| 트러블 슈팅                              |
| 1. 깃허브 액션 에러                        |
| 2. 컨테이너 생성 에러                       |
| 3. EC2 무반응                          |
| 4. SSL 인증 오류                        |
| 후기 및 앞으로 해야 하는 것들                   |

## 개요

프로젝트를 배포해야 하는데

배포에 대해 알아보던 중 CI/CD를 알게 되었고

AWS나 Docker 공부도 할 겸

GitHub Actions로 CI/CD를 구축해서 자동 배포를 해보기로 했다.

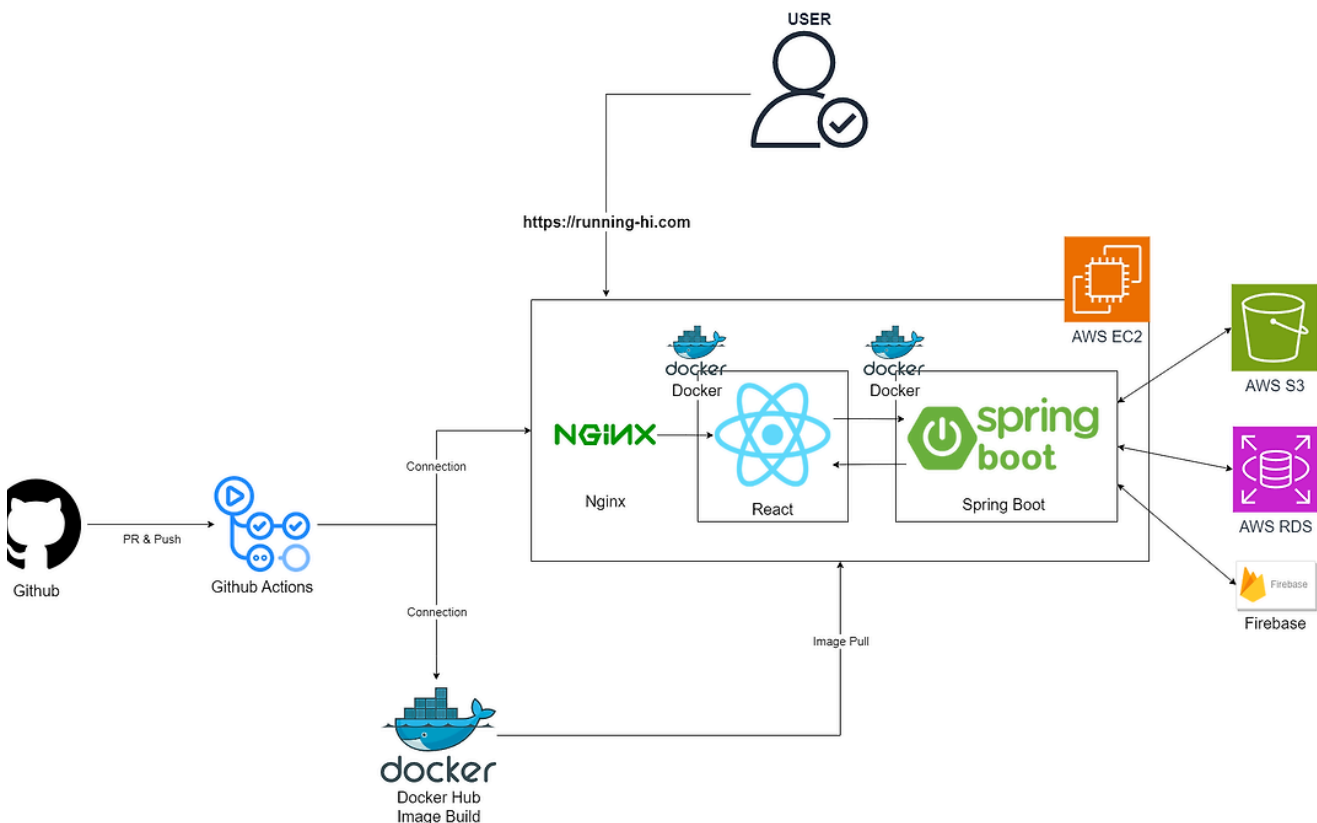
CI(Continuous Integration)는 코드의 빌드나 테스트, Merge 같은 지속적 통합을 의미하고

CD(Continuous Delivery & Continuous Deployment)는 지속적 서비스 제공 혹은 배포를 의미한다.

## 환경

- Java 17
- Spring Boot 3.0.6
- GitHub Actions
- AWS EC2(t2.micro, t3a.small)
- AWS RDS(db.t3.small)
- Docker 24.0.5
- Docker Compose v2.23.3
- Node 20 (React)
- Nginx 1.18.0

## Project Architecture



배포할 프로젝트의 웹 구조이다.

### GitHub Actions



깃허브 액션은 깃허브에서 제공하는 빌드, 배포 파이프라인 자동화를 해 주는 CI/CD 플랫폼이다.

Jenkins나 기타 파이프라인 프로그램보다

깃허브 액션을 쓴 이유는

































원래 깃허브 액션으로 CI를 하고 있었고

GitHub Secrets로 민감 정보 관리하기가 수월하기 때문이다.

#### - GitHub Secrets

깃허브에서 제공하는 보안 서비스이다.

환경변수들을 key-value방식으로 저장할 수 있다.

|          |              |   |   |
|----------|--------------|---|---|
|          | last week    |  |  |
| DEV      | last week    |  |  |
| PROD     | last week    |  |  |
| ORD_DEV  | 3 days ago   |  |  |
| ORD_PROD | 2 days ago   |  |  |
| AME_DEV  | 3 days ago   |  |  |
| AME_PROD | 2 days ago   |  |  |
|          | 2 weeks ago  |  |  |
|          | last week    |  |  |
|          | last week    |  |  |
|          | 3 months ago |  |  |
|          | last month   |  |  |
| RD       | 3 months ago |  |  |
| ME       | 3 months ago |  |  |
|          | last week    |  |  |
|          | last week    |  |  |

GitHub Secrets

백엔드 레포지토리에서 관리되고 있는 민감 정보들이다.

프론트엔드 레포지토리도 이런 식으로 민감 정보들이 관리되고 있다.

#### - github-actions.yml

name: Java CI/CD with Gradle and Docker

*# event trigger*

on:

push:

branches:

- dev
- master

pull\_request:

branches:

- dev

permissions:

contents: read

jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

*## jdk setting*

- name: Set up JDK 17

uses: actions/setup-java@v3

with:

java-version: '17'

distribution: 'temurin'

*## application.yml 파일을 생성합니다.*

- name: make application.yml

run: |

cd ./src/main/resources

touch ./application.yml

echo "\${{ secrets.APPLICATION }}" > ./application.yml

shell: bash

*## application-dev.yml 파일을 생성합니다.*

- name: make application-dev.yml

if: contains(github.ref, 'dev')

run: |

cd ./src/main/resources

touch ./application-dev.yml

echo "\${{ secrets.APPLICATION\_DEV }}" > ./application-dev.yml

shell: bash

*## application-prod.yml 파일을 생성합니다.*

- name: make application-prod.yml

if: contains(github.ref, 'master')

run: |

cd ./src/main/resources

touch ./application-prod.yml

echo "\${{ secrets.APPLICATION\_PROD }}" > ./application-prod.yml

shell: bash

*## jwt.yml 파일을 생성합니다.*

- name: make jwt.yml

run: |

cd ./src/main/resources

touch ./jwt.yml

echo "\${{ secrets.JWT }}" > ./jwt.yml

shell: bash

*## log4j2.yml 파일을 생성합니다.*

- name: make log4j2.yml

run: |

cd ./src/main/resources

touch ./log4j2.yml

echo "\${{ secrets.LOG4J2 }}" > ./log4j2.yml

shell: bash

*## runningHiFirebaseKey.json 파일을 생성합니다.*

- name: create-json

id: create-json

uses: jsdaniell/create-json@1.1.2

with:

name: "runningHiFirebaseKey.json"

json: "\${{ secrets.FIREBASE\_KEY }}"

- name: move-json-file

run: |

mv runningHiFirebaseKey.json ./src/main/resources/

*## gradlew 실행 권한 부여*

- name: Grant execute permission for gradlew

run: chmod +x gradlew

- name: Build with Gradle (without Test)

```
run: ./gradlew clean build -x test --stacktrace

##### CD(Continuous Delivery) #####
## docker hub에 로그인하고 runninghi-dev에 이미지를 빌드 & push 합니다.
- name: Docker build & push to dev repo
  if: contains(github.ref, 'dev')
  run: |
    docker login -u "${{ secrets.DOCKER_USERNAME_DEV }}" -p "${{ secrets.DOCKER_PASSWORD_DEV }}"
    docker build -f Dockerfile-dev -t "${{ secrets.DOCKER_USERNAME_DEV }}/runninghi-dev .
    docker push "${{ secrets.DOCKER_USERNAME_DEV }}/runninghi-dev

## docker hub에 로그인하고 runninghi-prod에 이미지를 빌드 & push 합니다.
- name: Docker build & push to prod repo
  if: contains(github.ref, 'master')
  run: |
    docker login -u "${{ secrets.DOCKER_USERNAME_PROD }}" -p "${{ secrets.DOCKER_PASSWORD_PROD }}"
    docker build -f Dockerfile-prod -t "${{ secrets.DOCKER_USERNAME_PROD }}/runninghi-prod .
    docker push "${{ secrets.DOCKER_USERNAME_PROD }}/runninghi-prod

## AWS EC2에 접속하고 develop을 배포합니다.
- name: Deploy to Dev
  uses: appleboy/ssh-action@master
  id: deploy-dev
  if: contains(github.ref, 'dev')
  with:
    host: ${{ secrets.HOST_DEV }} # EC2 인스턴스 퍼블릭 DNS
    username: ${{ secrets.USERNAME }} # 인스턴스를 시작하는 데 사용되는 AMI에 정의된 사용자 이름을 입력합니다. 사용자 지정 사용자 이름을 정의하지 않은 경우 기본 사용자 이름인
    key: ${{ secrets.PRIVATE_KEY }} # 키 페어의 pem 키
    #      sudo docker run -d --log-driver=syslog -p 8081:8081 -e SPRING_PROFILES_ACTIVE=dev "${{ secrets.DOCKER_USERNAME_DEV }}/runninghi-dev
  script: |
    sudo docker login -u "${{ secrets.DOCKER_USERNAME_DEV }}" -p "${{ secrets.DOCKER_PASSWORD_DEV }}"
    sudo docker stop runninghi_spring_boot_dev
    sudo docker rm -f $(sudo docker ps --filter 'status=exited' -a -q)
    sudo docker pull "${{ secrets.DOCKER_USERNAME_DEV }}/runninghi-dev
    sudo docker-compose up -d
    sudo docker image prune -a -f

## AWS EC2에 접속하고 production을 배포합니다.
- name: Deploy to Prod
  uses: appleboy/ssh-action@master
  id: deploy-prod
  if: contains(github.ref, 'master')
  with:
    host: ${{ secrets.HOST_PROD }} # EC2 인스턴스 퍼블릭 DNS
    username: ${{ secrets.USERNAME }} # 인스턴스를 시작하는 데 사용되는 AMI에 정의된 사용자 이름을 입력합니다. 사용자 지정 사용자 이름을 정의하지 않은 경우 기본 사용자 이름인
    key: ${{ secrets.PRIVATE_KEY }} # 키 페어의 pem 키
    #      sudo docker run -d --log-driver=syslog -p 8082:8082 -e SPRING_PROFILES_ACTIVE=prod "${{ secrets.DOCKER_USERNAME_PROD }}/runninghi-prod
  script: |
    sudo docker login -u "${{ secrets.DOCKER_USERNAME_PROD }}" -p "${{ secrets.DOCKER_PASSWORD_PROD }}"
    sudo docker stop runninghi_spring_boot_prod
    sudo docker rm -f $(sudo docker ps --filter 'status=exited' -a -q)
    sudo docker pull "${{ secrets.DOCKER_USERNAME_PROD }}/runninghi-prod
    sudo docker-compose up -d
    sudo docker image prune -a -f
```

내가 작성한 workflows의 gradlew.yml 파일이다.  
파일 이름은 github-actions.yml로 직관적이게 바꿨다.

우선 CI 과정은  
각자의 코드들을 하나의 브랜치로 통합하는 과정이다.  
PR 또는 Push 이벤트가 발생하면  
깃허브 액션 파일이 이벤트를 감지해 트리거가 실행된다.  
그리고 프로젝트 build를 위해 필요한 설정 파일들(yml)을 만들고 build 한다.  
여기서 깃허브 시크릿츠에 있는 값들을 사용한다.

다음은 CD 과정이다.  
깃허브 액션은 도커 허브에 로그인을 하고  
앞선 과정에서 빌드된 jar파일로 image를 만든다.  
image를 만든 뒤 EC2 인스턴스에 접속해  
해당 이미지를 도커 허브로부터 풀 받아서 실행시킨다.  
이렇게 하면 우리 개발자들은 늘 하던 대로 코드만 Merge해도 자동 배포가 된다.

그리고 실제 사용자들을 위한 안정적인 서비스 제공과  
프론트엔드 개발자들이 편하게 api 테스트하기 위해  
프로젝트 환경을 develop과 production으로 분리하기로 결정했다.

하지만 현재 계획한 페이지들 완성이 덜 되어서,  
환경 분리는 사이트가 최소한으로 완성이 되었을 때 비로소 효과를 볼 것 같다.

AWS (Amazon Web Services)

아카데미에서 AWS 금액 지원을 해줘서  
다양한 서비스들을 이용하기로 하였다.

- EC2 (Elastic Compute Cloud)

EC2는 클라우드에서 온디맨드 확장 가능 컴퓨팅 용량을 제공한다.  
한마디로 아마존에서 컴퓨터를 빌려주는 서비스다.  
인스턴스 1개 = 컴퓨터 1대라고 생각하면 된다.

우리 프로젝트의 서버를 24시간 365일 띄워놓고 싶어서 EC2를 사용하기로 했다.

AWS EC2 인스턴스

아카데미에서 AWS 지원하는 계정을 원생들이 모두 받았고  
전체 금액이 한정되어 있는 상황이라  
중요한 인스턴스(prod)만 유료 유형으로 사용 중이다.

- Route 53

AWS Route 53은 DNS(도메인 이름 시스템) 웹 서비스다.  
Route 53은 DNS 서버가 포트 53에서 쿼리에 반응하며  
최종 사용자를 인터넷상의 애플리케이션으로 경로를 지정해 준다는 의미라고 한다.