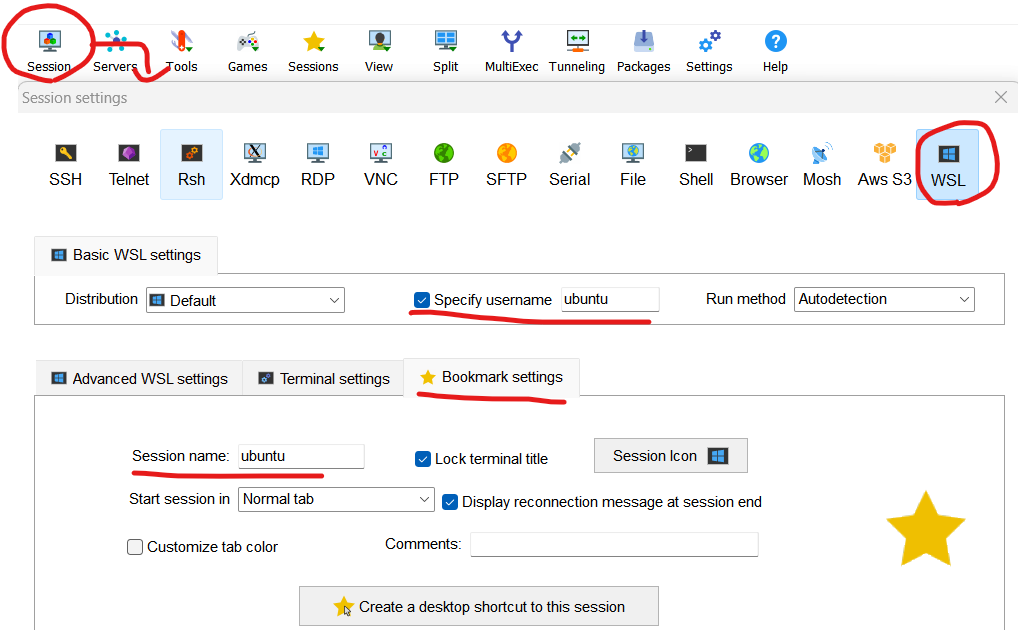
# CI/CD 가이드

# I. 사전 작업

## 1) 로컬에 ubuntu 설치

<https://github.com/cna-bootcamp/cna-handson/blob/main/prepare/%EB%A1%9C%EC%BB%AC%EA%B0%9C%EB%B0%9C%ED%99%98%EA%B2%BD%EA%B5%AC%EC%84%B1.md#%EB%A1%9C%EC%BB%AC-ubuntu-%EC%82%AC%EC%9A%A9-%EC%84%A4%EC%A0%95>

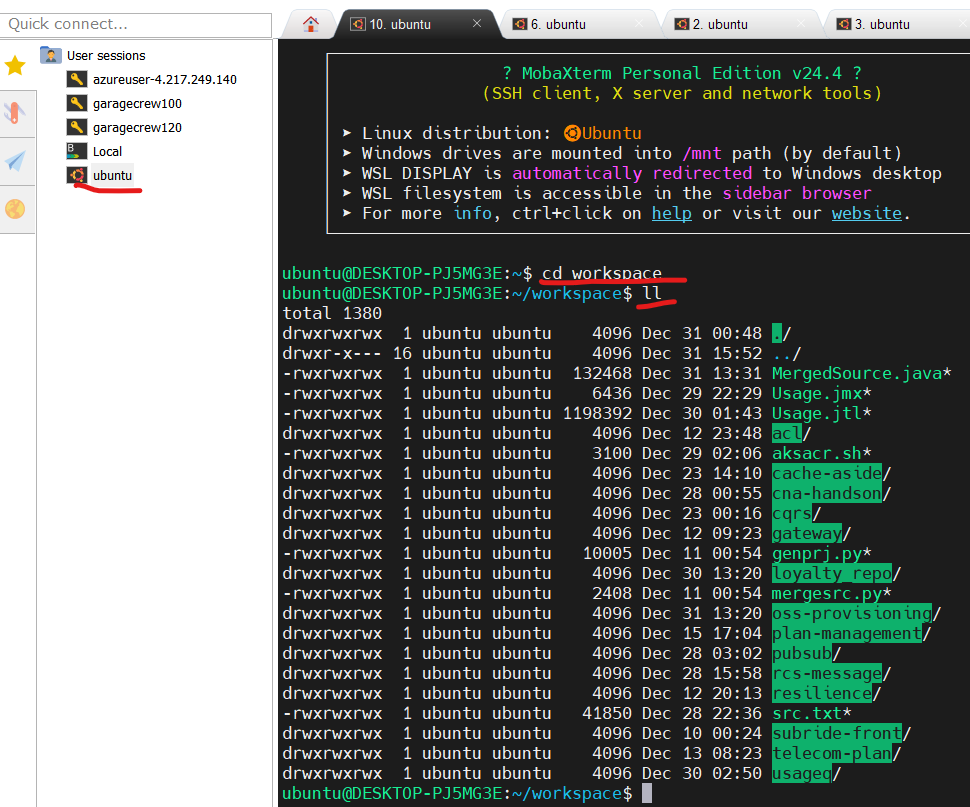
설치 완료 후 WSL 세션을 만듭니다.



그리고, 그 세션으로 접근합니다.

workspace로 이동하여 c:\home\workspace의 디렉토리가 보이는지 확인합니다.

안 보인다면 mount가 안된거니 위 가이드를 보고 mount를 수행하십시오.



**이후 작업은 이 로컬 ubuntu에서 작업합니다.**

## 2) Jenkins, SonarQube, ArgoCD 설치

아래 링크를 참조하여 설치합니다.

<https://github.com/cna-bootcamp/lifesub-guide/blob/main/5.CICD/guide_cicd.md#%ED%88%B4-%EC%84%A4%EC%B9%98-jenkins-sonarqube-argocd>

## 3) Backing서비스와 애플리케이션 수동 배포

CI/CD Pipeline을 간결하게 하기 위해 Backing서비스를 미리 배포합니다.

또한 Application배포를 위한 manifest파일을 만들어 정상 동작하는지 점검 합니다.

앞으로의 작업들은 특별한 언급이 없으면 로컬 ubuntu에서 수행합니다.

### - 사전작업

0) 기존 배포 오브젝트 삭제

- Database 삭제

| helm ls helm delete {db release명} |
| --- |

삭제 후 pvc도 삭제합니다.

| k get pvc  K delete pvc --all |
| --- |

- MQ, Blob Storage 등: Azure portal에서 찾아서 삭제합니다.

1) Dockerfile 업로드

Backend 프로젝트에 deployment/container디렉토리를 만들고 ‘Dockerfile’이라는 이름으로 생성

| # Build stage FROM openjdk:23-oraclelinux8 AS builder ARG BUILD\_LIB\_DIR ARG ARTIFACTORY\_FILE COPY ${BUILD\_LIB\_DIR}/${ARTIFACTORY\_FILE} app.jar  # Run stage FROM openjdk:23-slim ENV USERNAME k8s ENV ARTIFACTORY\_HOME /home/${USERNAME} ENV JAVA\_OPTS=""  # Add a non-root user RUN adduser --system --group ${USERNAME} && \  mkdir -p ${ARTIFACTORY\_HOME} && \  chown ${USERNAME}:${USERNAME} ${ARTIFACTORY\_HOME}  WORKDIR ${ARTIFACTORY\_HOME} COPY --from=builder app.jar app.jar RUN chown ${USERNAME}:${USERNAME} app.jar  USER ${USERNAME}  ENTRYPOINT [ "sh", "-c" ] CMD ["java ${JAVA\_OPTS} -jar app.jar"] |
| --- |

프론트엔드 프로젝트의 최상위에 ‘deployment/container’ 디렉토리를 만들고 아래 내용으로 ‘Dockerfile-{frontend service}’라는 이름으로 생성. 예) Dockerfile-lifesub-web

| # Build stage FROM node:20-slim AS builder ARG PROJECT\_FOLDER  ENV NODE\_ENV=production  WORKDIR /app  # Install dependencies COPY ${PROJECT\_FOLDER}/package\*.json ./ RUN npm ci --only=production  # Build application COPY ${PROJECT\_FOLDER} . RUN npm run build  # Run stage FROM nginx:stable-alpine  ARG BUILD\_FOLDER ARG EXPORT\_PORT  # Create nginx user if it doesn't exist RUN adduser -S nginx || true  # Copy build files COPY --from=builder /app/build /usr/share/nginx/html  # Copy and process nginx configuration COPY ${BUILD\_FOLDER}/nginx.conf /etc/nginx/templates/default.conf.template  # Add custom nginx settings RUN echo "client\_max\_body\_size 100M;" > /etc/nginx/conf.d/client\_max\_body\_size.conf RUN echo "proxy\_buffer\_size 128k;" > /etc/nginx/conf.d/proxy\_buffer\_size.conf RUN echo "proxy\_buffers 4 256k;" > /etc/nginx/conf.d/proxy\_buffers.conf RUN echo "proxy\_busy\_buffers\_size 256k;" > /etc/nginx/conf.d/proxy\_busy\_buffers\_size.conf  # Set permissions RUN chown -R nginx:nginx /usr/share/nginx/html && \  chmod -R 755 /usr/share/nginx/html && \  chown -R nginx:nginx /var/cache/nginx && \  chown -R nginx:nginx /var/log/nginx && \  chown -R nginx:nginx /etc/nginx/conf.d && \  touch /var/run/nginx.pid && \  chown -R nginx:nginx /var/run/nginx.pid  USER nginx  EXPOSE ${EXPORT\_PORT}  CMD ["nginx", "-g", "daemon off;"] |
| --- |

2) nginx.conf 파일 업로드

frontend 의 deployment/container디렉토리 밑에 ‘nginx.conf’라는 파일을 아래 내용을 생성

| server {  listen 18080;  server\_name localhost;    location / {  root /usr/share/nginx/html;  index index.html index.htm;  try\_files $uri $uri/ /index.html;    # Cache static files  location ~\* \.(jpg|jpeg|png|gif|ico|css|js)$ {  expires 1y;  add\_header Cache-Control "public, no-transform";  }  }    # Health check endpoint  location /health {  access\_log off;  return 200 'healthy\n';  add\_header Content-Type text/plain;  }    # Error pages  error\_page 500 502 503 504 /50x.html;  location = /50x.html {  root /usr/share/nginx/html;  } } |
| --- |

3) Backend 확인 및 수정

최상위의 build.gradle에 아래 Actuator 라이브러리 없으면 추가

| dependencies {  // Spring Boot Starters  implementation 'org.springframework.boot:spring-boot-starter'  …  **implementation 'org.springframework.boot:spring-boot-starter-actuator'**  } |
| --- |

각 서비스의 application.yml에 아래 추가 (common은 필요 없음)

| # Actuator 설정 management:  endpoints:  web:  exposure:  include: health,info,metrics,prometheus  endpoint:  health:  show-details: always  probes:  enabled: true  health:  livenessState:  enabled: true  readinessState:  enabled: true |
| --- |

4) Ingress Controller 설치

<https://github.com/cna-bootcamp/lifesub-guide/blob/main/3.Kubernetes/guide_k8s.md#ingress-controller-%EC%B6%94%EA%B0%80>

### - ‘개발가이드’ 업로드

아래 링크의 파일을 다운로드한 후 Claude Project의 Knowledge에 업로드

<https://docs.google.com/document/d/15q4cLdK7quBlm8nZhm6vNCKKtNPH1SfYb_yTLwaQSyQ/edit?usp=drive_link>

### - Instructions에 추가

아래 [실행 가이드]를 Claude Project의 instruction에 추가합니다.

| 'r:'로 시작하면 실행하기 위한 가이드를 요청하는 것임. {요청사항} 1.'최적안 가이드'에 따라 Knowledge의 'MergedSource'를 분석하여 필요한 Backing서비스를 정확히 찾음. 2.Backing서비스의 설치 방법은 Knowledge의 '개발가이드' 참조. 3.네임스페이스 안내: kubens를 이용한 네임스페이스 변경 안내 4.Backend는 Database 설치 방법 안내: '개발가이드'와 동일하게 안내.  - '데이터 설계서'를 참조하여 마이크로서비스별로 별도의 DB 배포  - helm chart로 설치하는 방법 안내.  - helm chart release명: "{db종류}-{system}"  - values.yaml은 '개발가이드'의 샘플을 그대로 사용. values.yaml 직접 작성하면 안됨.  - 개발가이드에 있는 values.yaml를 사용하도록 안내.  - ID는 {Team ID}, PW는 'Hi5Jessica!'. 5.Application 빌드 방법 안내 1) 공통요청  - '최적안 가이드'에 따라 정확하게 안내.  - 모든 배포 관련 파일은 'deployment'디렉토리 하위에 생성.  - 'az acr'명령 대신 'docker login' 명령 사용 방법 제공. username과 password 찾는법도 제공.  - ACR 로그인, 빌드, 푸시 방법 안내   - Image Pull Secret 객체 생성 방법 안내  2) Backend  - 멀티 프로젝트이므로 {backend root} 하위에 각 {backend service}디렉토리로 구성됨.   - build & push image 명령:  - Application build는 아래 예제 참조.   클린 아키텍처 프로젝트는 {backend service}-infra"임.  ```  ./gradlew :${backend service}:clean :${backend service}:build -x test  ```  - Dockerfile 관련 작업:  - 새 Dockerfile을 생성하지 말고 이미 존재하는 파일(deployment/container/Dockerfile-{frontend service})만 참조  - 주의: Dockerfile 또는 nginx.conf 파일을 직접 생성하는 코드를 가이드 안 함  - 클린 아키텍처 프로젝트는 MergedSource.java를 보고 BUILD\_LIB\_DIR를 정확히 파악  "{backend service}-infra/build/libs"이거나 "{backend service}/build/libs"중 하나임.   - 이미지 빌드 명령은 {backend root}디렉토리에서 수행하고 아래 예시를 참조  ```  docker build \  --build-arg BUILD\_LIB\_DIR="{backend service}/build/libs" \  --build-arg ARTIFACTORY\_FILE="{backend service}.jar" \  -f deployment/container/Dockerfile \  -t ${ACR}/{Team ID}/{backend service}:{version} .  ```  - ALLOWED\_ORIGINS는 {frontend host}로 함  - Service 오브젝트의 포트는 "80"으로 함  - 'az acr'명령 대신 'docker login' 명령 사용 방법 제공. username과 password 찾는법도 제공.  - {backend root} 디렉토리에서 각 서비스 별 수행. 3) Frontend  - 단일 프로젝트이므로 {frontend service}가 최상위 디렉토리임.  - build & push image 명령:  - Dockerfile은 기존 파일인 deployment/container/Dockerfile-{frontend service}을 사용(생성하면 안됨)  - Dockerfile 관련 작업:  - 새 Dockerfile을 생성하지 말고 이미 존재하는 파일(deployment/container/Dockerfile)만 참조  - 주의: Dockerfile 파일을 직접 생성하는 코드를 가이드 안 함  - 이미지 빌드 명령은 {frontend service}디렉토리로 이동하여 수행하도록 하고 아래 예시를 참조  ```  docker build \  --build-arg PROJECT\_FOLDER="." \  --build-arg BUILD\_FOLDER="deployment/container" \  --build-arg EXPORT\_PORT="18080" \  -f deployment/container/Dockerfile-lifesub-web \  -t ${ACR}/{Team ID}/{frontend service}:{version} .  ```  - {frontend service} 디렉토리에서 수행. 6.k8s 배포를 위한 manifest 생성 방법 안내  - Backend 서비스는 '{backend root}/deploymant/manifest' 하위에 k8s 리소스 유형별로 디렉토리를 만들고 작성   - Frontend 서비스는 '{frontend service}/deploymant/manifest' 하위에 k8s 리소스 유형별로 디렉토리를 만들고 작성   - Frontend 서비스의 k8s 서비스 오브젝트 타입은 LoadBalancer로 지정  - manifest 파일은 생략하지 말고 모든 {backend service}와 {frontend service}의 yaml 내용을 제공  - Backend 서비스는 각 서비스의 application.yml 분석하여 ConfigMap과 Secret객체 생성에 참조  - 일반 환경변수는 ConfigMap으로 생성. 공통 환경변수와 서비스별로 환경변수로 작성   - ConfigMap 작성 시 Database연결 Host값은 배포된 Database의 서비스 오브젝트명을 확인하여 작성  - 보안이 필요한 환경변수는 Secret으로 생성. 공통 환경변수와 서비스별로 환경변수로 작성  - Secret 변수 값은 'stringData'를 사용하여 평문으로 지정.  - env만 사용하지 말고 envFrom도 사용  - JWT Secret Key는 openssl명령으로 생성  - ImagePullPolicy는 Always로 함.  - Service yaml에 targetPort 누락하지 않기. {frontend service}는 'EXPORT\_PORT'와 동일함.  - 동적으로 manifest 파일이 생성되게 가이드.   - Pod 수는 {replicas}로 함  - Resource Request/Limit지정: {resource.requests.cpu}, {resource.requests.memory}, {resource.limits.cpu}, {resource.limits.memory}값을 참조  - Backend 서비스는 Startup Probe, Liveness Probe, ReadinessProbe 지정   - Startup Probe: Database 연결이 되어야 함   - Readiness Probe: Actuator '/actuator/health'로 지정   - Liveness Probe: Actuator '/actuator/health'로 지정   - Frontend 서비스는 {ingress host} 값을 이용하여 아래 예와 같이 각 backend서비스의 주소를 nginx public폴더 하위에 runtime-env.js파일로 마운트 되도록 함  - {backend service}\_URL: 'http://{ingress host}/{api그룹경로}'형식   ```  window.\_\_runtime\_config\_\_ = {  AUTH\_URL: 'http://{ingress host}/api/auth',  CONTENT\_URL: 'http://{ingress host}/api/contents',  TOKEN\_URL: 'http://{ingress host}/api/tokens'  }  ```  - Ingress manifest는 Backend 서비스에만 작성. 한 파일로 작성함.  - host: {ingress host}  - path: /{api그룹경로}  - pathType: Prefix  - annotation에 rewrite와 use-regex는 사용 안함  7.manifest 점검 및 실행 방법 안내  - Database의 k8s Service 명을 확인하는 방법  - 명령어: kubectl get svc  - ConfigMap의 Database Host 수정 방법 안내  - manifest 실행 방법을 안내: root 디렉토리에서 수행.  {참고자료} - MergedSource: Backend 소스 - MergedFront: Frontend 소스 - 개발가이드  {응답순서} 1.네임스페이스 생성 2.(Backend배포시에만)Database 설치: '{요청사항}의 4. Database 설치 방법 안내' 준수 3.(Backend배포시에만)기타 Backing 서비스 설치: MergedSource 분석하여 제공 4.Application 빌드: '{요청사항}의 5. Application 빌드 방법 안내' 준수  - 주의: 새 Dockerfile을 생성하지 말고 이미 존재하는 파일만 참조  - 프로젝트의 상위 디렉토리(예: ~/workspace)로 이동해야함을 강조하여 알림 5.Image Pull Secret 객체 생성  6.Kubernetes Manifest 생성: '{요청사항}의 6. k8s 배포를 위한 manifest 생성 방법 안내' 준수  - ConfigMap: 일반 환경변수  - Secret  - Database 접속 정보  - 그외 보안 정보   - Deployment: 컨테이너 설정  - Service: 서비스 설정  - (Backend 배포시에만) Ingress 7.manifest 점검 및 실행: '{요청사항} > 7. manifest 점검 및 실행 방법 안내' 준수 8.정상 배포 확인 방법   {응답형식} - 문단형식 |
| --- |

### - Application 소스 업로드

1) 백엔드 어플리케이션

| cd ~/workspace python3 mergesrc.py |
| --- |

애플리케이션 root 디렉토리를 입력하여 Mergedsource.java로 소스를 합칩니다.

Claude Project의 knowledge에 업로드 합니다.

참고) mergesrc.py 사용법

* python mergesrc.py:

‘cqrs’입력 : CQRS 프로젝트 전체 합침

* python mergesrc.py :   
  cqrs/command  
  CQRS의 command 서비스 소스만 합침
* python mergesrc.py :   
  cqrs/command cqrs/query  
  CQRS의 command, query 서비스 소스만 합침

- Jar파일명 지정

각 서비스의 build.gradle에 Jar파일명 지정.

| bootJar {  archiveFileName = "{서비스명}.jar" }  예시)  bootJar {  archiveFileName = "command.jar" } |
| --- |

※ 기존 {서비스 디렉토리}/build/libs 하위에 생성된 jar파일은 모두 삭제 하세요.

2) 프론트엔드 어플리케이션

| cd ~/workspace python3 merge-node.py {frontend 디렉토리명} |
| --- |

예) python3 merge-node.py lifesub-web

애플리케이션 root 디렉토리를 입력하여 MergedFront.txt로 소스를 합칩니다.

Claude Project의 knowledge에 업로드 합니다.

### - 설치 가이드 생성 및 배포

아래 프롬프트를 입력하여 설치 가이드를 생성합니다.

* {Team ID}: 본인 팀의 id를 입력
* {system}: 전체 서비스를 대표하는 명칭
* {backend root}: 백엔드 애플리케이션 root 프로젝트명
* {frontend service}: 프론트엔드 서비스명. Frontend 최상위 디렉토리명이어야 함
* {backend service}: 백엔드 프로젝트명을 쉼표로 구분하여 입력

아래는 프롬프트 예제입니다.

프론트엔드부터 배포합니다.

사용하는 API G/W(SCG, ingress, istio)에 따라 service object의 External IP를 구합니다.

그 값을 {ingress host}에 지정합니다.

Ingress 이용 시에는 아래와 같이 구합니다.

| k get svc -n ingress-basic |
| --- |

SCG는 SCG의 서비스 오브젝트의 IP를 지정합니다.

| r: 프론트엔드 실행 가이드를 작성해 주세요. - instruction의 '[실행가이드]'의 지침을 준수해서 만들어 주세요. - Azure 서비스  - AKS: aks-digitalgarage-01  - ACR: acrdigitalgarage01  - Resource Group: rg-digitalgarage-01  - Location: koreacentral - 환경변수   - {Team ID}: unicorn  - {system}: lifesub  - 네임스페이스: {Team ID}-{system}-ns   - {frontend service}: lifesub-web  - {version}: 1.0.0  - {replicas}: 1  - {resource.requests.cpu}: 256m, {resource.requests.memory}:256Mi  - {resource.limits.cpu}:1024m, {resource.limits.memory}:1024Mi  - {ingress host}: {Team ID}.20.249.205.95.nip.io |
| --- |

백엔드를 배포합니다.

Frontend service 오브젝트의 External IP를 구해 {frontend host}에 지정합니다.

이 값은 ALLOWED\_ORIGINS값을 위한 ConfigMap에서 사용됩니다.

Frontend를 로컬에서 실행하여 연결하려면 아래와 같이 ‘[http://localhost:3000’번이](about:blank) 포함 되어야 합니다.

프론트엔드 배포후에는 프론트엔드 서비스 오브젝트의 IP도 포함되어야 합니다.

| r: 백엔드 실행 가이드를 작성해 주세요.  - instruction의 '[실행가이드]'의 지침을 준수해서 만들어 주세요. - 데이터베이스는 각 서비스마다 독립적으로 구성 - 데이터베이스 설치 시 architecture를 standalone으로 하여 1개만 배포 - Azure 서비스  - AKS: aks-digitalgarage-01  - ACR: acrdigitalgarage01  - Resource Group: rg-digitalgarage-01  - Location: koreacentral - 환경변수   - {Team ID}: unicorn  - {system}: lifesub  - 네임스페이스: {Team ID}-{system}-ns   - {backend root}: lifesub  - {backend service}: member, mysub, recommend  - {version}: 1.0.0  - {replicas}: 2  - {resource.requests.cpu}: 256m, {resource.requests.memory}:256Mi  - {resource.limits.cpu}:1024m, {resource.limits.memory}:1024Mi  - {frontend host}: [http://localhost:3000,http://4.230.147.248](about:blank)  - {ingress host}: {Team ID}.20.249.205.95.nip.io |
| --- |

### - 설치가이드에 따라 배포

설치 가이드에 따라 Database, MQ, Application을 배포 합니다.

### - Git repository에 업로드

설치 가이드 수행하면서 만들어진 파일들을 Git repository에 업로드 합니다.

프론트엔드와 백엔드 모두 업로드 합니다.

| cd ~/workspace/{project} git add . && git commit -m "add deployment files" && git push |
| --- |

개발자에게는 git pull 명령으로 최신 소스로 동기화하고 작업하도록 하십시오.

### - DB의 Load Balancer 서비스 Object 만들기

로컬에서 DB 접속을 위해 필요 합니다.

a) DB가 배포된 namespace로 이동. 작업 디렉토리도 만들고 이동동

| kubens {namespace}  mkdir -p ~/work && cd ~/work |
| --- |

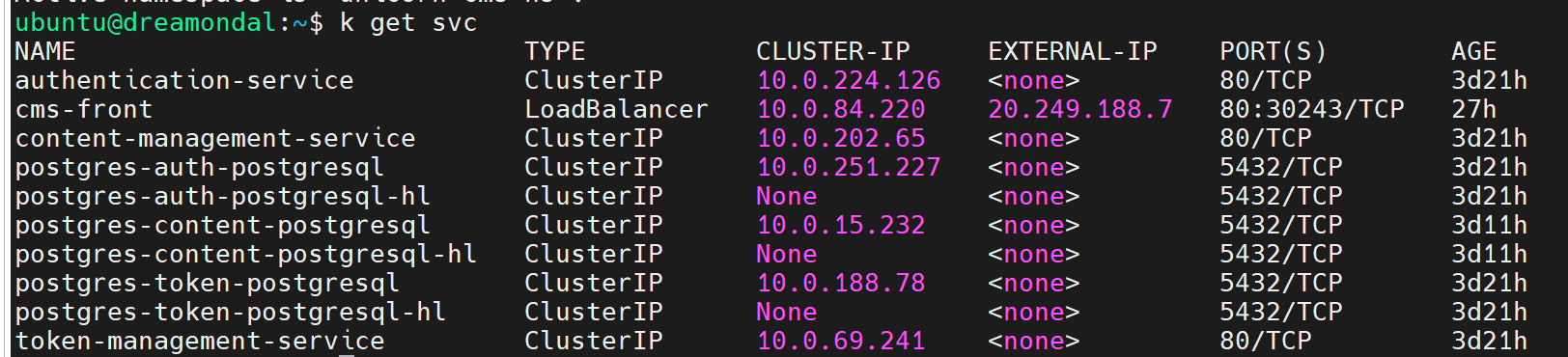
b) DB서비스 오브젝트 리스트 보기

아래 예와 같이 각 서비스별로 DB가 배포되어 있습니다.

이 중에 끝에 ‘-h1’으로 끝나는 것이 headless 서비스 입니다.

우리는 이 서비스 오브젝트를 이용합니다.

Headless 서비스는 Pod 로드밸런싱 목적이 아니라 Stateful Pod를 직접 연결하기 위한 목적입니다.



c) 배포된 DB의 서비스 오브젝트를 이용하여 yaml 파일을 만듭니다.

k get svc {headless service}-h1 -o yaml > {DB유형}-{서비스명}-external.yaml

예) k get svc postgres-auth-postgresql-hl -o yaml > postgres-auth-external.yaml

d) 에디터로 yaml 파일을 엽니다.

e) 아래 빨강색 부분을 지웁니다.



아래 예와 같이 남기면 됩니다.

| apiVersion: v1 kind: Service metadata:  name: postgres-auth-postgresql-hl  namespace: unicorn-cms-ns spec:  ports:  - name: tcp-postgresql  port: 5432  protocol: TCP  targetPort: tcp-postgresql  selector:  app.kubernetes.io/instance: postgres-auth  app.kubernetes.io/name: postgresql  type: ClusterIP |
| --- |

f) 서비스 오브젝트 이름을 바꿉니다.

{DB유형}-{서비스명}-external

예) postgres-auth-external

g) 서비스 type을 LoadBalancer로 변경

최종 결과 예시는 아래와 같습니다.

| apiVersion: v1 kind: Service metadata:  name: postgres-auth-external  namespace: unicorn-cms-ns spec:  ports:  - name: tcp-postgresql  port: 5432  protocol: TCP  targetPort: tcp-postgresql  selector:  app.kubernetes.io/instance: postgres-auth  app.kubernetes.io/name: postgresql  type: LoadBalancer |
| --- |

h) 파일을 Apply하여 서비스 오브젝트를 생성합니다.

| k apply -f {yaml 명} |
| --- |

i) 서비스 오브젝트가 생성되고 External IP가 부여 되는지 확인

| k get svc |
| --- |

j) End Point를 확인합니다. End point에 연결할 Pod IP가 나와야 합니다.

| k get ep |
| --- |

**참고)**

※ Database의 Service명을 잘 확인하고 manifest.yaml의 ConfigMap의 DB Host를 동일하게 맞춤

- mongo DB는 primary pod를 연결하는 ‘mongo-external’ 서비스명 사용  
 (Service object 생성 하는 yaml에서 pod name이 master Pod를 가르치도록 해야 함)

- postgres는 primary pod를 연결하는 service명 지정(예: postgres-cqrs-postgresql-primary-hl)

- kubectl get ep 로 Master Pod로 연결되는 End Point가 생성되는지 확인

※ Event Grid Subscribe는 bastion VM에서 수행해야 정상 설치됨

※ Event Hub는 Namespace가 Basic버전이라 아래 예와 같이 partition-count, cleanup-policy, retention-time을 지정해야 정상 생성됨.

| az eventhubs eventhub create \ --name ${EVENTHUB\_NAME} \ --namespace-name ${EVENTHUB\_NS} \ --resource-group ${RESOURCE\_GROUP} \ --partition-count 1 \ --cleanup-policy Delete \ --retention-time 24 |
| --- |

※ 각 어플리케이션의 service의 targetPort와 application.yaml의 server.port가 일치하는 검사

### 

### - EventHub 관련 객체 생성 체크

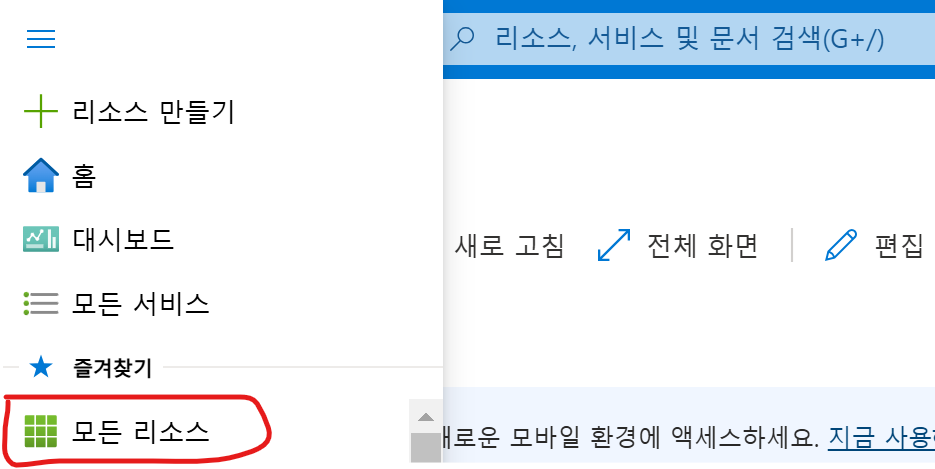
아래 Naming Rule에 따라 객체명을 계산

| - StorageID: ${TEAMID}storage{root project} - BLOB CONTAINER: ${TEAMID}-checkpoints-{root project} - EventHub Namespace: ${TEAMID}-ns-{root project} - EventHub Name: ${TEAMID}-name-{root project} |
| --- |

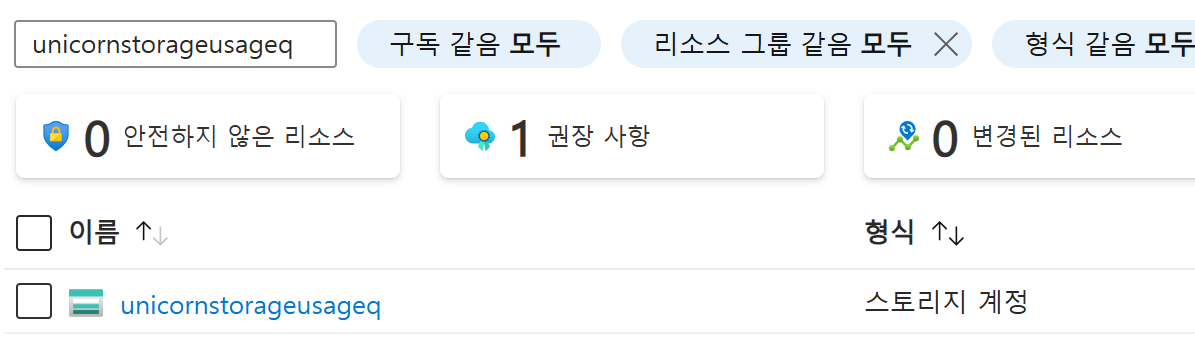
예시)

| - StorageID: unicornstorageusageq - BLOB CONTAINER: unicorn-checkpoints-usageq - EventHub Namespace: unicorn-ns-usageq - EventHub Name: unicorn-name-usageq |
| --- |

**Azure Portal에서 확인**

****

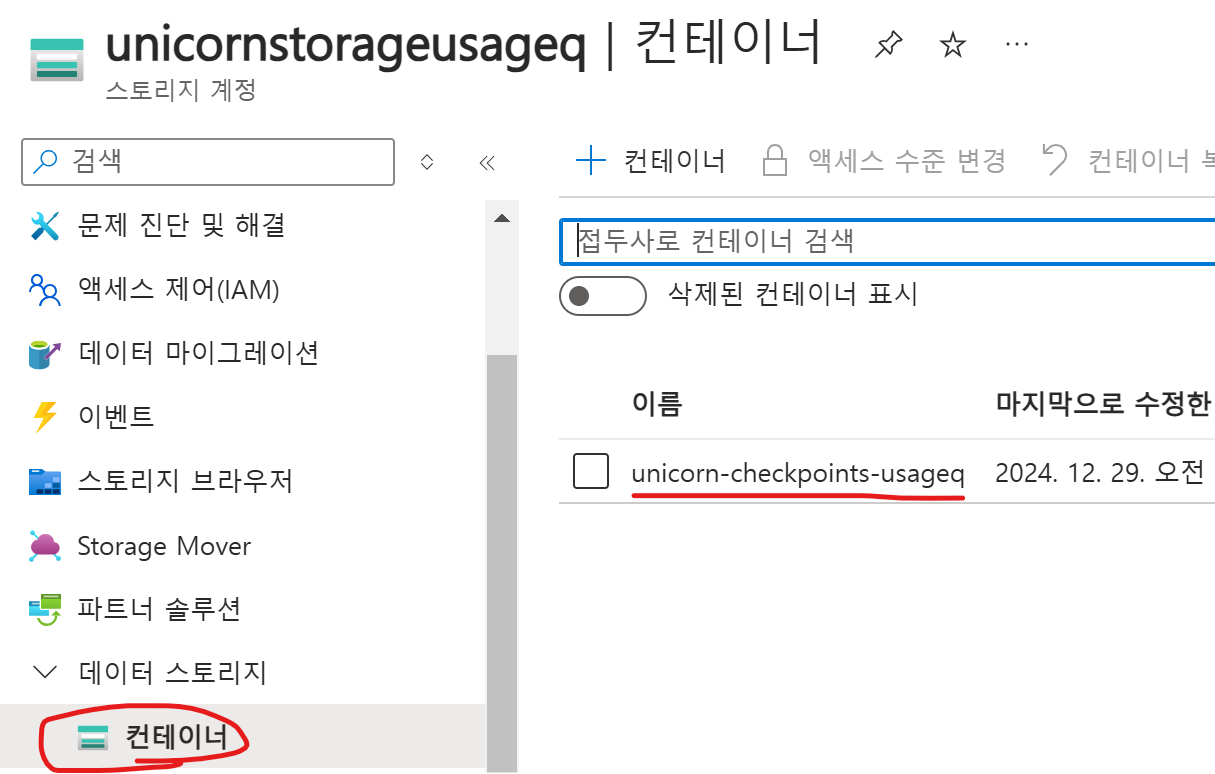
**Storage ID를 필터링하여 찾음**



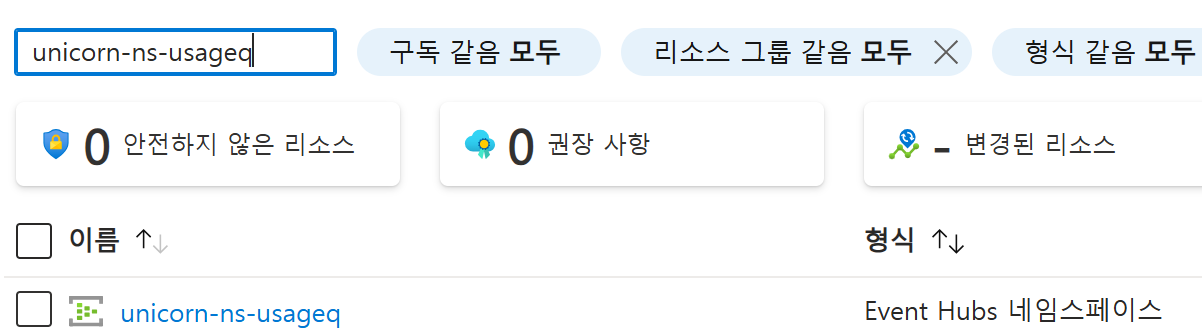
**BLOB Container 생성 확인: Storage객체 클릭하여 확인**

**※ EventHub의 Dead Letter Queue**

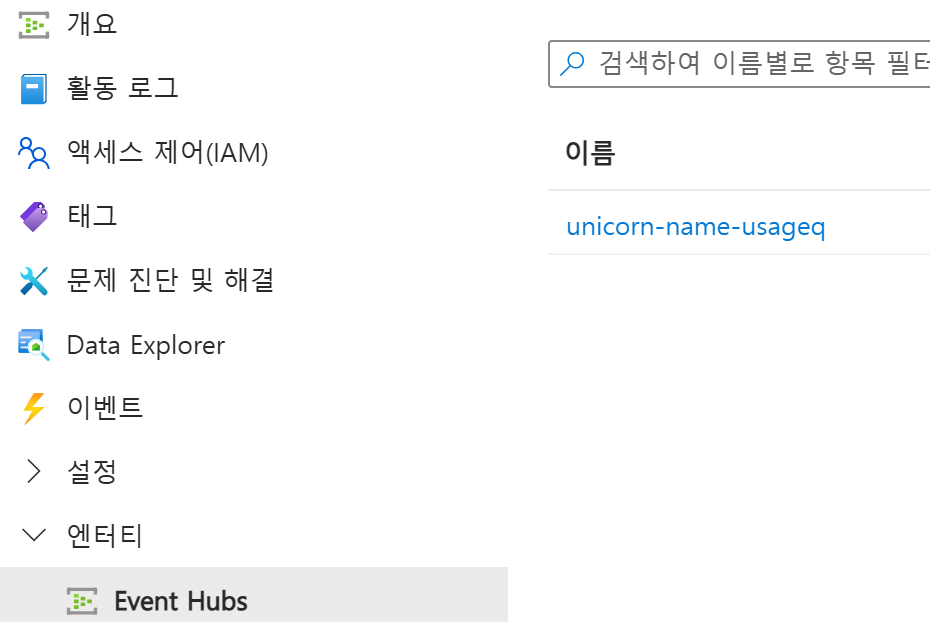
Azure Event Hub의 경우 Dead Letter Queue는 Checkpoint Store(BLOB Container)에서 관리합니다.



EventHub Namespace 확인



EventHub 생성 확인: Namespace클릭하여 확인



### - EventGrid 관련 객체 생성 체크

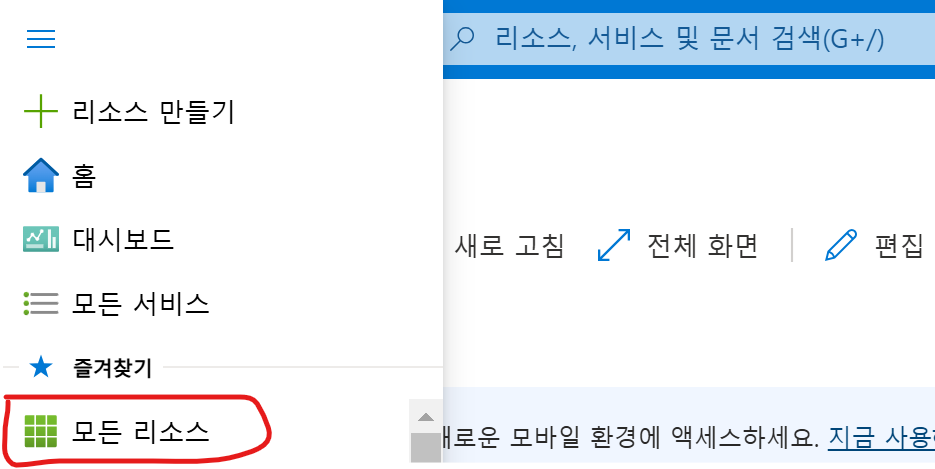
아래 Naming Rule에 따라 객체명을 계산하므로 Azure Portal에서 찾을 때 참고하세요.

| - StorageID: ${TEAMID}storage{root project} - Topic: ${TEAMID}-topic-{root project} - Dead Letter: ${TEAMID}deadletter{root project} - Event Grid Subscriber: ${TEAMID}-sub-{root project} |
| --- |

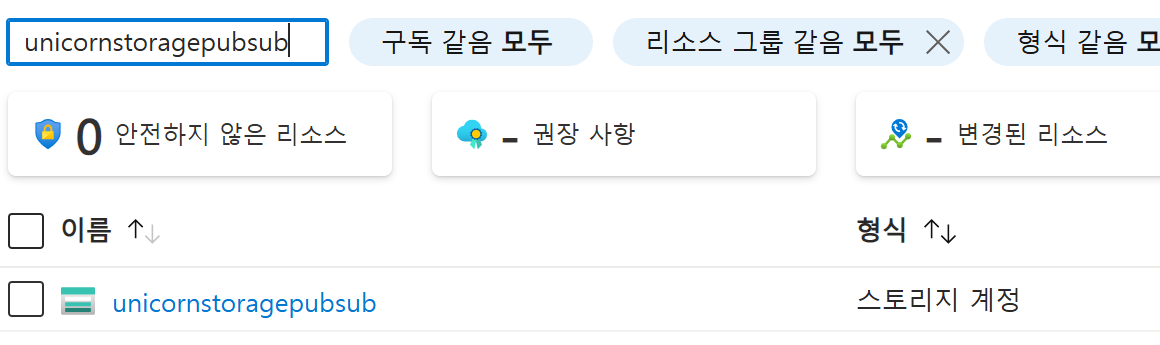
예시)

| - StorageID: unicornstoragepubsub - Topic: unicorn-topic-pubsub - Dead Letter: unicorndeadletterpubsub - Event Grid Subscriber: unicorn-sub-pubsub |
| --- |

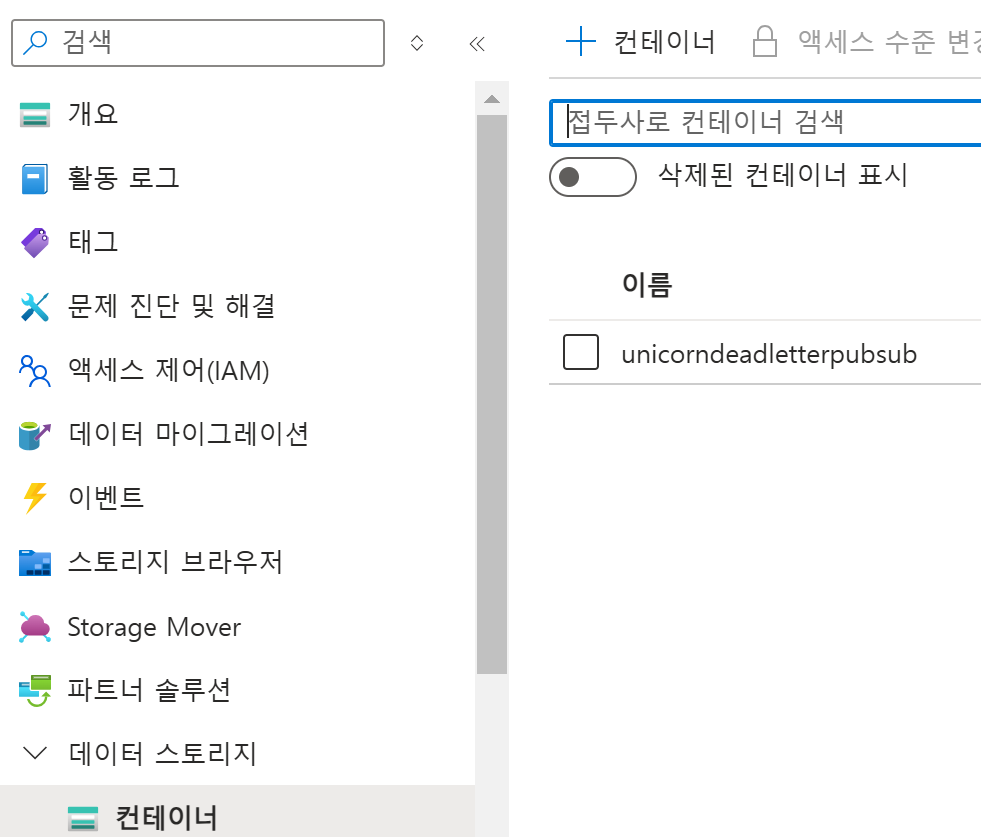
**Azure Portal에서 확인**

****

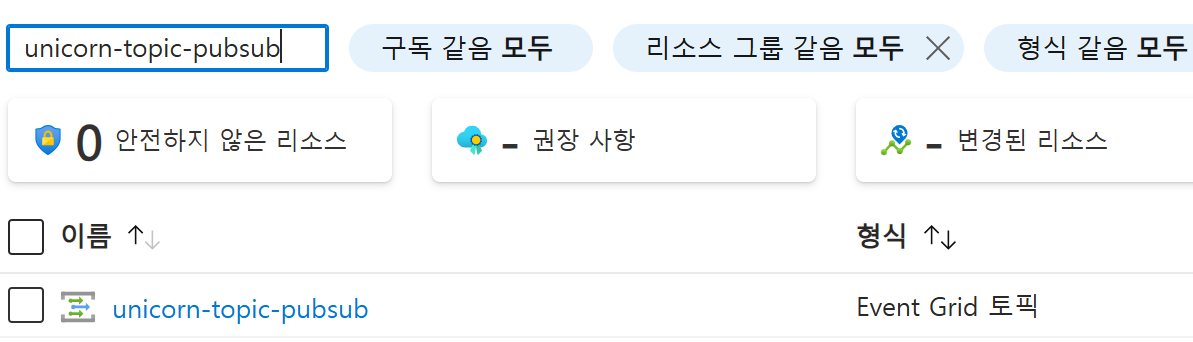
**Storage ID를 필터링하여 찾음**



Dead Letter Queue가 생성되었는지 확인

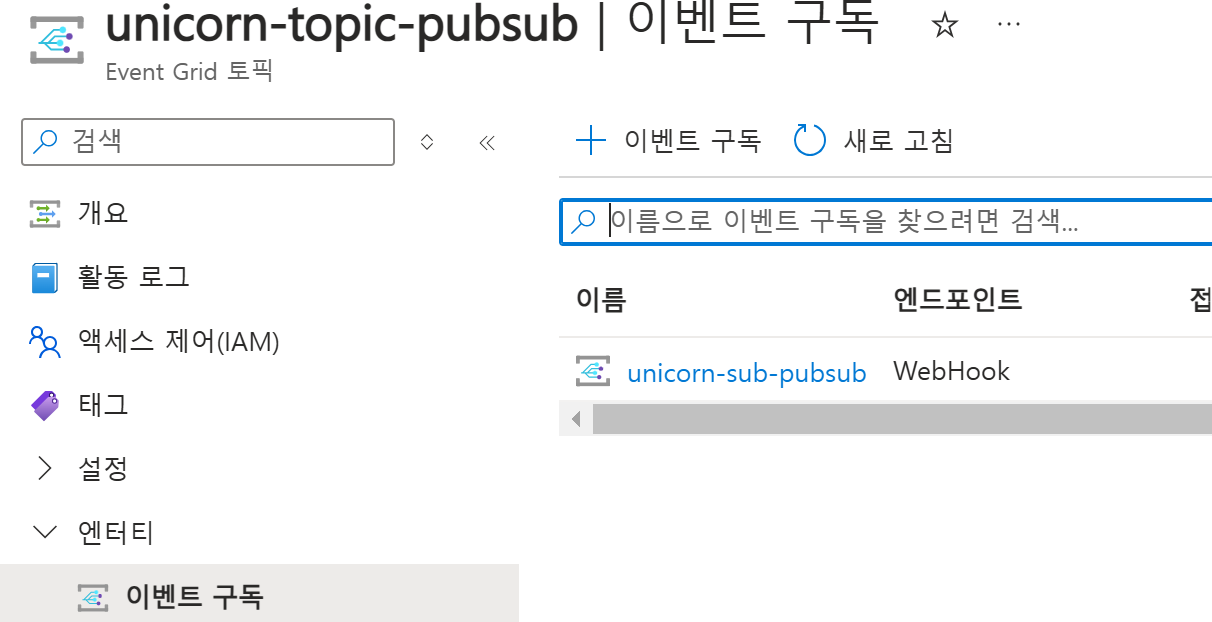


**EventGrid Topic확인**



**EventGrid Subscriber 생성 확인**

EventGrid Topic를 클릭한 후 아래 메뉴에서 확인.



### - 정상 실행될때까지 리팩토링

테스트 하면서 에러가 나면 claude와 협업하여 수정합니다.

수정후에는,

아래 예와 같이 수정된 애플리케이션을 빌드, 이미지 빌드/배포를 한 후 Pod를 재시작합니다.

백엔드 예시

| cd ~/workspace/lifesub  SERVICE=member  ACRNAME=unicorncr.azurecr.io TAG=latest  ./gradlew $SERVICE:build -x test  docker build -f deployment/container/Dockerfile -t $ACRNAME/telecom/$SERVICE:$TAG .  docker push $ACRNAME/telecom/$SERVICE:$TAG  k rollout restart deployment/$SERVICE  k logs -f deploy/$SERVICE |
| --- |

프론트엔드 예시

| cd ~/workspace/lifesub-web    SERVICE=lifesub-web  ACRNAME=unicorncr.azurecr.io TAG=latest  docker build -f deployment/container/Dockerfile-lifesub-web -t $ACRNAME/telecom/$SERVICE:$TAG .  docker push $ACRNAME/telecom/$SERVICE:$TAG  k rollout restart deployment/$SERVICE  k logs -f deploy/$SERVICE |
| --- |

모든 서비스가 정상 동작하면 다음 단계로 진행합니다.

# II. Jenkins

## 1. CI/CD 파일 작성

### 1) Sample CI/CD파일 업로드

Claude Project의 Knowledge에 업로드

1.1) 백엔드 배포를 위한 샘플 업로드

- sample\_deploy.yaml.template

| # ConfigMap  apiVersion: v1  kind: ConfigMap  metadata:  name: common-config  namespace: ${namespace}  data:  ALLOWED\_ORIGINS: ${allowed\_origins}  JPA\_DDL\_AUTO: update  JPA\_SHOW\_SQL: 'true'  ---  apiVersion: v1  kind: ConfigMap  metadata:  name: member-config  namespace: ${namespace}  data:  POSTGRES\_DB: member  POSTGRES\_HOST: member-postgresql  POSTGRES\_PORT: '5432'  SERVER\_PORT: '8081'  ---  apiVersion: v1  kind: ConfigMap  metadata:  name: mysub-config  namespace: ${namespace}  data:  FEE\_LEVEL\_ADDICT: '100000'  FEE\_LEVEL\_COLLECTOR: '50000'  POSTGRES\_DB: mysub  POSTGRES\_HOST: mysub-postgresql  POSTGRES\_PORT: '5432'  SERVER\_PORT: '8082'  ---  apiVersion: v1  kind: ConfigMap  metadata:  name: recommend-config  namespace: ${namespace}  data:  POSTGRES\_DB: recommend  POSTGRES\_HOST: recommend-postgresql  POSTGRES\_PORT: '5432'  SERVER\_PORT: '8083'  ---  # Secrets  apiVersion: v1  kind: Secret  metadata:  name: common-secret  namespace: ${namespace}  stringData:  JWT\_SECRET\_KEY: ${jwt\_secret\_key}  type: Opaque  ---  apiVersion: v1  kind: Secret  metadata:  name: member-secret  namespace: ${namespace}  stringData:  JWT\_ACCESS\_TOKEN\_VALIDITY: '3600000'  JWT\_REFRESH\_TOKEN\_VALIDITY: '86400000'  POSTGRES\_PASSWORD: ${postgres\_password}  POSTGRES\_USER: ${postgres\_user}  type: Opaque  ---  apiVersion: v1  kind: Secret  metadata:  name: mysub-secret  namespace: ${namespace}  stringData:  POSTGRES\_PASSWORD: ${postgres\_password}  POSTGRES\_USER: ${postgres\_user}  type: Opaque  ---  apiVersion: v1  kind: Secret  metadata:  name: recommend-secret  namespace: ${namespace}  stringData:  POSTGRES\_PASSWORD: ${postgres\_password}  POSTGRES\_USER: ${postgres\_user}  type: Opaque  ---  # Deployments  apiVersion: apps/v1  kind: Deployment  metadata:  name: member  namespace: ${namespace}  spec:  replicas: ${replicas}  selector:  matchLabels:  app: member  template:  metadata:  labels:  app: member  spec:  containers:  - name: member  image: ${member\_image\_path}  imagePullPolicy: Always  envFrom:  - configMapRef:  name: common-config  - configMapRef:  name: member-config  - secretRef:  name: common-secret  - secretRef:  name: member-secret  resources:  requests:  cpu: ${resources\_requests\_cpu}  memory: ${resources\_requests\_memory}  limits:  cpu: ${resources\_limits\_cpu}  memory: ${resources\_limits\_memory}  ports:  - containerPort: 8081  startupProbe:  httpGet:  path: /actuator/health  port: 8081  failureThreshold: 30  periodSeconds: 10  livenessProbe:  httpGet:  path: /actuator/health  port: 8081  initialDelaySeconds: 60  periodSeconds: 15  readinessProbe:  httpGet:  path: /actuator/health/readiness  port: 8081  initialDelaySeconds: 10  periodSeconds: 5  ---  apiVersion: apps/v1  kind: Deployment  metadata:  name: mysub  namespace: ${namespace}  spec:  replicas: ${replicas}  selector:  matchLabels:  app: mysub  template:  metadata:  labels:  app: mysub  spec:  containers:  - name: mysub  image: ${mysub\_image\_path}  imagePullPolicy: Always  envFrom:  - configMapRef:  name: common-config  - configMapRef:  name: mysub-config  - secretRef:  name: common-secret  - secretRef:  name: mysub-secret  resources:  requests:  cpu: ${resources\_requests\_cpu}  memory: ${resources\_requests\_memory}  limits:  cpu: ${resources\_limits\_cpu}  memory: ${resources\_limits\_memory}  ports:  - containerPort: 8082  startupProbe:  httpGet:  path: /actuator/health  port: 8082  failureThreshold: 30  periodSeconds: 10  livenessProbe:  httpGet:  path: /actuator/health  port: 8082  initialDelaySeconds: 60  periodSeconds: 15  readinessProbe:  httpGet:  path: /actuator/health/readiness  port: 8082  initialDelaySeconds: 10  periodSeconds: 5  ---  apiVersion: apps/v1  kind: Deployment  metadata:  name: recommend  namespace: ${namespace}  spec:  replicas: ${replicas}  selector:  matchLabels:  app: recommend  template:  metadata:  labels:  app: recommend  spec:  containers:  - name: recommend  image: ${recommend\_image\_path}  imagePullPolicy: Always  envFrom:  - configMapRef:  name: common-config  - configMapRef:  name: recommend-config  - secretRef:  name: common-secret  - secretRef:  name: recommend-secret  resources:  requests:  cpu: ${resources\_requests\_cpu}  memory: ${resources\_requests\_memory}  limits:  cpu: ${resources\_limits\_cpu}  memory: ${resources\_limits\_memory}  ports:  - containerPort: 8083  startupProbe:  httpGet:  path: /actuator/health  port: 8083  failureThreshold: 30  periodSeconds: 10  livenessProbe:  httpGet:  path: /actuator/health  port: 8083  initialDelaySeconds: 60  periodSeconds: 15  readinessProbe:  httpGet:  path: /actuator/health/readiness  port: 8083  initialDelaySeconds: 10  periodSeconds: 5  ---  # Services  apiVersion: v1  kind: Service  metadata:  name: member  namespace: ${namespace}  spec:  selector:  app: member  ports:  - port: 80  targetPort: 8081  type: ClusterIP  ---  apiVersion: v1  kind: Service  metadata:  name: mysub  namespace: ${namespace}  spec:  selector:  app: mysub  ports:  - port: 80  targetPort: 8082  type: ClusterIP  ---  apiVersion: v1  kind: Service  metadata:  name: recommend  namespace: ${namespace}  spec:  selector:  app: recommend  ports:  - port: 80  targetPort: 8083  type: ClusterIP  ---  apiVersion: networking.k8s.io/v1  kind: Ingress  metadata:  name: lifesub  namespace: ${namespace}  annotations:  kubernetes.io/ingress.class: nginx  nginx.ingress.kubernetes.io/rewrite-target: /$2  nginx.ingress.kubernetes.io/use-regex: "true"  spec:  ingressClassName: nginx  rules:  - http:  paths:  - path: /member(/|$)(.\*)  pathType: ImplementationSpecific  backend:  service:  name: member  port:  number: 80  - path: /mysub(/|$)(.\*)  pathType: ImplementationSpecific  backend:  service:  name: mysub  port:  number: 80  - path: /recommend(/|$)(.\*)  pathType: ImplementationSpecific  backend:  service:  name: recommend  port:  number: 80 |
| --- |

- sample\_deploy\_env\_vars

| # Team Settings  teamid=unicorn  root\_project=lifesub  namespace=unicorn-lifesub-ns  # Container Registry Settings  registry=unicorncr.azurecr.io  image\_org=lifesub  # Application Settings  replicas=1  allowed\_origins=http://4.230.147.248  # Security Settings  jwt\_secret\_key=8O2HQ13etL2BWZvYOiWsJ5uWFoLi6NBUG8divYVoCgtHVvlk3dqRksMl16toztDUeBTSIuOOPvHIrYq11G2BwQ  postgres\_user=admin  postgres\_password=Hi5Jessica!  # Resource Settings  resources\_requests\_cpu=256m  resources\_requests\_memory=256Mi  resources\_limits\_cpu=1024m  resources\_limits\_memory=1024Mi |
| --- |

- sample\_Jenkinsfile

| def PIPELINE\_ID = "${env.BUILD\_NUMBER}"  def getImageTag() {  def dateFormat = new java.text.SimpleDateFormat('yyyyMMddHHmmss')  def currentDate = new Date()  return dateFormat.format(currentDate)  }  podTemplate(  label: "${PIPELINE\_ID}",  serviceAccount: 'jenkins',  containers: [  containerTemplate(name: 'podman', image: "mgoltzsche/podman", ttyEnabled: true, command: 'cat', privileged: true),  containerTemplate(name: 'gradle',  image: 'gradle:jdk17',  ttyEnabled: true,  command: 'cat',  envVars: [  envVar(key: 'DOCKER\_HOST', value: 'unix:///run/podman/podman.sock'),  envVar(key: 'TESTCONTAINERS\_DOCKER\_SOCKET\_OVERRIDE', value: '/run/podman/podman.sock'),  envVar(key: 'TESTCONTAINERS\_RYUK\_DISABLED', value: 'true')  ]),  containerTemplate(name: 'azure-cli', image: 'hiondal/azure-kubectl:latest', command: 'cat', ttyEnabled: true),  containerTemplate(name: 'envsubst', image: "hiondal/envsubst", command: 'sleep', args: '1h')  ],  volumes: [  emptyDirVolume(mountPath: '/home/gradle/.gradle', memory: false),  emptyDirVolume(mountPath: '/root/.azure', memory: false),  emptyDirVolume(mountPath: '/run/podman', memory: false)  ]  ) {  node(PIPELINE\_ID) {  def props  def imageTag = getImageTag()  def manifest = "deploy.yaml"  def namespace  def services = ['member', 'mysub', 'recommend']  stage("Get Source") {  checkout scm  props = readProperties file: "deployment/deploy\_env\_vars"  namespace = "${props.namespace}"  }  stage("Setup AKS") {  container('azure-cli') {  withCredentials([azureServicePrincipal('azure-credentials')]) {  sh """  az login --service-principal -u \$AZURE\_CLIENT\_ID -p \$AZURE\_CLIENT\_SECRET -t \$AZURE\_TENANT\_ID  az aks get-credentials --resource-group ictcoe-edu --name ${props.teamid}-aks --overwrite-existing  kubectl create namespace ${namespace} --dry-run=client -o yaml | kubectl apply -f -  """  }  }  }  stage('Build Applications & SonarQube Analysis') {  container('podman') {  sh 'podman system service -t 0 unix:///run/podman/podman.sock & sleep 2'  }  container('gradle') {  def testContainersConfig = '''docker.client.strategy=org.testcontainers.dockerclient.UnixSocketClientProviderStrategy  docker.host=unix:///run/podman/podman.sock  ryuk.container.privileged=true  testcontainers.reuse.enable=true'''  sh """  # TestContainers 설정  mkdir -p member/src/test/resources mysub-infra/src/test/resources recommend/src/test/resources  echo '${testContainersConfig}' > member/src/test/resources/testcontainers.properties  echo '${testContainersConfig}' > mysub-infra/src/test/resources/testcontainers.properties  echo '${testContainersConfig}' > recommend/src/test/resources/testcontainers.properties  """  // Member 서비스 빌드 및 SonarQube 분석  withSonarQubeEnv('SonarQube') {  sh """  chmod +x gradlew  # 빌드 실행  ./gradlew :member:build :mysub-infra:build :recommend:build -x test  # Member 서비스  ./gradlew :member:test :member:jacocoTestReport :member:sonar \  -Dsonar.projectKey=lifesub-member \  -Dsonar.projectName=lifesub-member \  -Dsonar.java.binaries=build/classes/java/main \  -Dsonar.coverage.jacoco.xmlReportPaths=build/reports/jacoco/test/jacocoTestReport.xml \  -Dsonar.exclusions=\*\*/config/\*\*,\*\*/entity/\*\*,\*\*/dto/\*\*,\*\*/\*Application.class,\*\*/exception/\*\*  # Recommend 서비스  ./gradlew :recommend:test :recommend:jacocoTestReport :recommend:sonar \  -Dsonar.projectKey=lifesub-recommend \  -Dsonar.projectName=lifesub-recommend \  -Dsonar.java.binaries=build/classes/java/main \  -Dsonar.coverage.jacoco.xmlReportPaths=build/reports/jacoco/test/jacocoTestReport.xml \  -Dsonar.exclusions=\*\*/config/\*\*,\*\*/entity/\*\*,\*\*/dto/\*\*,\*\*/\*Application.class,\*\*/exception/\*\*  # Mysub 서비스 (biz & infra 구조)  ./gradlew :mysub-infra:test :mysub-infra:jacocoTestReport :mysub-infra:sonar \  -Dsonar.projectKey=lifesub-mysub \  -Dsonar.projectName=lifesub-mysub \  -Dsonar.java.binaries=build/classes/java/main \  -Dsonar.coverage.jacoco.xmlReportPaths=build/reports/jacoco/test/jacocoTestReport.xml \  -Dsonar.exclusions=\*\*/config/\*\*,\*\*/entity/\*\*,\*\*/dto/\*\*,\*\*/\*Application.class,\*\*/exception/\*\*  """  }  }  }  stage('Quality Gate') {  timeout(time: 10, unit: 'MINUTES') {  def qg = waitForQualityGate()  if (qg.status != 'OK') {  error "Pipeline aborted due to quality gate failure: ${qg.status}"  }  }  }  stage('Build & Push Images') {  container('podman') {  withCredentials([usernamePassword(  credentialsId: 'acr-credentials',  usernameVariable: 'USERNAME',  passwordVariable: 'PASSWORD'  )]) {  sh "podman login ${props.registry} --username \$USERNAME --password \$PASSWORD"  services.each { service ->  def buildDir = service == 'mysub' ? 'mysub-infra' : service  def jarFile = service == 'mysub' ? 'mysub.jar' : "${service}.jar"  sh """  podman build \  --build-arg BUILD\_LIB\_DIR="${buildDir}/build/libs" \  --build-arg ARTIFACTORY\_FILE="${jarFile}" \  -f deployment/container/Dockerfile \  -t ${props.registry}/${props.image\_org}/${service}:${imageTag} .  podman push ${props.registry}/${props.image\_org}/${service}:${imageTag}  """  }  }  }  }  stage('Generate & Apply Manifest') {  container('envsubst') {  sh """  export namespace=${namespace}  export allowed\_origins=${props.allowed\_origins}  export jwt\_secret\_key=${props.jwt\_secret\_key}  export postgres\_user=${props.postgres\_user}  export postgres\_password=${props.postgres\_password}  export replicas=${props.replicas}  export resources\_requests\_cpu=${props.resources\_requests\_cpu}  export resources\_requests\_memory=${props.resources\_requests\_memory}  export resources\_limits\_cpu=${props.resources\_limits\_cpu}  export resources\_limits\_memory=${props.resources\_limits\_memory}  # 이미지 경로 환경변수 설정  export member\_image\_path=${props.registry}/${props.image\_org}/member:${imageTag}  export mysub\_image\_path=${props.registry}/${props.image\_org}/mysub:${imageTag}  export recommend\_image\_path=${props.registry}/${props.image\_org}/recommend:${imageTag}  # manifest 생성  envsubst < deployment/${manifest}.template > deployment/${manifest}  cat deployment/${manifest}  """  }  container('azure-cli') {  sh """  kubectl apply -f deployment/${manifest}  echo "Waiting for deployments to be ready..."  kubectl -n ${namespace} wait --for=condition=available deployment/member --timeout=300s  kubectl -n ${namespace} wait --for=condition=available deployment/mysub --timeout=300s  kubectl -n ${namespace} wait --for=condition=available deployment/recommend --timeout=300s  """  }  }  }  } |
| --- |

- sample\_deploy\_env\_vars-front

| # Team Settings  teamid=unicorn  root\_project=lifesub-web  namespace=unicorn-lifesub-ns  # Container Registry Settings  registry=unicorncr.azurecr.io  image\_org=lifesub  # Application Settings  replicas=1  export\_port=18080  # Resource Settings  resources\_requests\_cpu=256m  resources\_requests\_memory=256Mi  resources\_limits\_cpu=1024m  resources\_limits\_memory=1024Mi |
| --- |

- sample\_Jenkinsfile-front

| def PIPELINE\_ID = "${env.BUILD\_NUMBER}"  def getImageTag() {  def dateFormat = new java.text.SimpleDateFormat('yyyyMMddHHmmss')  def currentDate = new Date()  return dateFormat.format(currentDate)  }  podTemplate(  label: "${PIPELINE\_ID}",  serviceAccount: 'jenkins',  containers: [  containerTemplate(name: 'node', image: 'node:20-slim', ttyEnabled: true, command: 'cat'),  containerTemplate(name: 'podman', image: "mgoltzsche/podman", ttyEnabled: true, command: 'cat', privileged: true),  containerTemplate(name: 'azure-cli', image: 'hiondal/azure-kubectl:latest', command: 'cat', ttyEnabled: true),  containerTemplate(name: 'envsubst', image: "hiondal/envsubst", command: 'sleep', args: '1h'),  containerTemplate(name: 'sonar-scanner', image: 'sonarsource/sonar-scanner-cli:latest', command: 'cat', ttyEnabled: true)  ],  volumes: [  emptyDirVolume(mountPath: '/root/.azure', memory: false),  emptyDirVolume(mountPath: '/opt/sonar-scanner/.sonar/cache', memory: false)  ]  ) {  node(PIPELINE\_ID) {  def props  def imageTag = getImageTag()  def manifest = "deploy.yaml"  def namespace  def sonarScannerHome = '/opt/sonar-scanner'  stage("Get Source") {  checkout scm  props = readProperties file: "deployment/deploy\_env\_vars"  namespace = "${props.namespace}"  }  stage('Code Analysis & Quality Gate') {  container('node') {  sh "npm install"  sh "npm test -- --coverage --passWithNoTests" //test code 없어도 통과되게 함  }    container('sonar-scanner') {  withSonarQubeEnv('SonarQube') {  sh """  ${sonarScannerHome}/bin/sonar-scanner \  -Dsonar.projectKey=lifesub-web \  -Dsonar.sources=src \  -Dsonar.tests=src \  -Dsonar.test.inclusions=src/\*\*/\*.test.js,src/\*\*/\*.test.jsx \  -Dsonar.javascript.lcov.reportPaths=coverage/lcov.info  """  }  }    timeout(time: 10, unit: 'MINUTES') {  def qg = waitForQualityGate()  if (qg.status != 'OK') {  error "Pipeline aborted due to quality gate failure: ${qg.status}"  }  }  }  stage("Setup AKS") {  container('azure-cli') {  withCredentials([azureServicePrincipal('azure-credentials')]) {  sh """  az login --service-principal -u \$AZURE\_CLIENT\_ID -p \$AZURE\_CLIENT\_SECRET -t \$AZURE\_TENANT\_ID  az aks get-credentials --resource-group ictcoe-edu --name ${props.teamid}-aks --overwrite-existing  kubectl create namespace ${namespace} --dry-run=client -o yaml | kubectl apply -f -  """  }  }  }  stage('Build & Push Image') {  container('podman') {  withCredentials([usernamePassword(  credentialsId: 'acr-credentials',  usernameVariable: 'USERNAME',  passwordVariable: 'PASSWORD'  )]) {  def imagePath = "${props.registry}/${props.image\_org}/lifesub-web:${imageTag}"    sh """  podman login ${props.registry} --username \$USERNAME --password \$PASSWORD  podman build \  --build-arg PROJECT\_FOLDER="." \  --build-arg BUILD\_FOLDER="deployment/container" \  --build-arg EXPORT\_PORT="${props.export\_port}" \  -f deployment/container/Dockerfile-lifesub-web \  -t ${imagePath} .    podman push ${imagePath}  """  }  }  }  stage('Generate & Apply Manifest') {  container('envsubst') {  sh """  export namespace=${namespace}  export lifesub\_web\_image\_path=${props.registry}/${props.image\_org}/lifesub-web:${imageTag}  export replicas=${props.replicas}  export export\_port=${props.export\_port}  export resources\_requests\_cpu=${props.resources\_requests\_cpu}  export resources\_requests\_memory=${props.resources\_requests\_memory}  export resources\_limits\_cpu=${props.resources\_limits\_cpu}  export resources\_limits\_memory=${props.resources\_limits\_memory}    envsubst < deployment/${manifest}.template > deployment/${manifest}  cat deployment/${manifest}  """  }  container('azure-cli') {  sh """  kubectl apply -f deployment/${manifest}    echo "Waiting for deployment to be ready..."  kubectl -n ${namespace} wait --for=condition=available deployment/lifesub-web --timeout=300s    echo "Waiting for service external IP..."  while [[ -z \$(kubectl -n ${namespace} get svc lifesub-web -o jsonpath='{.status.loadBalancer.ingress[0].ip}') ]]; do  sleep 5  done  echo "Service external IP: \$(kubectl -n ${namespace} get svc lifesub-web -o jsonpath='{.status.loadBalancer.ingress[0].ip}')"  """  }  }  }  } |
| --- |

### 2) Dockerfile 업로드

2.1) 백엔드 애플리케이션

사전작업에서 생성된 deployment/container/Dockerfile 파일을 Claude Project의 knowledge에 업로드

2.2) 프론트엔드 애플리케이션

사전작업에서 생성된 deployment/container/Dockerfile-{서비스명} 파일을 Knowledge에 ‘Dockerfile-{서비스명}’이라는 이름으로 업로드

예) Dockerfile-lifesub-web

### 3) manifest.yaml 업로드

3.1) 백엔드 애플리케이션

사전작업에서 생성된 deployment/manifest.yaml파일을 Claude Project의 knowledge에 업로드 하십시오.

만약, manifest파일이 리소스 유형별로 분리되어 있다면 아래와 같이 merge-manifest.py파일로 합쳐서

올리시면 됩니다.

merge-manifest.py 파일은 ‘교재 > Claude Instruction’에서 다운로드 하세요.

다운로드한 파일을 ‘C:\home\workspace’에 다운로드 하시고 아래 예제처럼 수행하세요.

| cd ~/workspace  python3 merge-manifest.py Manifest파일들이 있는 최상위 디렉토리를 입력: lifesub/deployment/manifest  결과 파일명 입력 (기본값: manifest.yaml): |
| --- |

3.2) 프론트엔드 애플리케이션

사전작업에서 생성된 deployment/manifest.yaml파일을 Claude Project의 knowledge에

‘manifest-front.yaml’이라는 이름으로 업로드 하십시오.

만약, manifest파일이 리소스 유형별로 분리되어 있다면 아래와 같이 merge-manifest.py파일로 합쳐서

올리시면 됩니다.

merge-manifest.py 파일은 ‘교재 > Claude Instruction’에서 다운로드 하세요.

다운로드한 파일을 ‘C:\home\workspace’에 다운로드 하시고 아래 예제처럼 수행하세요.

| cd ~/workspace  python3 merge-manifest.py Manifest파일들이 있는 최상위 디렉토리를 입력: lifesub-web/deployment/manifest  결과 파일명 입력 (기본값: manifest.yaml): manifest-front.yaml |
| --- |

### 4) Instruction에 CI/CD 가이드 추가

| [CI/CD 가이드]  'p:'로 시작하면 {애플리케이션 유형}을 위한 CI/CD 가이드를 요청하는 것임.  {애플리케이션 유형}은 ‘backend’ 또는 ‘frontend’임  {요청사항}  1. Database와 MQ는 이미 설치되어 있음  2. Dockerfile은 Knowledge의 Dockerfile-{서비스명}을 사용. 만약 없으면 ‘Dockerfile’ 사용  3. deploy.yaml.template  - backend는 manifest.yaml의 ConfigMap과 Secret을 동일하게 이용.  - frontend는 manifest-front.yaml의 ConfigMap과 Secret을 동일하게 이용.  - backend는 manifest.yaml과 동일한 object 이름을 사용.  - frontend는 manifest-front.yaml과 동일한 object 이름을 사용  4. deploy\_env\_vars 파일에 환경변수(예: ${TEAMID}-cms-ns)를 사용하면 안됨  5. Jenkins 스크립트에서 SonarQube서버와 연동하여 소스품질검사와 테스트 코드 커버리지를 측정.  sonar.projectKey는 {root project}-{서비스명}-{Team ID}임.  6. Jenkins 스크립트에서 디버깅을 위해 생성된 manifest 파일 출력  7. 각 응답은 별도의 코드블록으로 제공  {참고자료}  - Dockerfile-{서비스명}. 만약 없으면 Dockerfile 참조  - backend  - manifest.yaml  - sample\_deploy.yaml.template  - sample\_deploy\_env\_vars  - sample\_Jenkinsfile  - frontend  - manifest-front.yaml  - sample\_deploy.yaml.template-front  - sample\_deploy\_env\_vars-front  - sample\_Jenkinsfile-front  {응답순서}  - deploy.yaml.template  - deploy\_env\_vars  - Jenkinsfile |
| --- |

### 5) CI/CD 파일 생성

- CI/CD파일 생성

아래 예와 같이 프롬프트를 입력하여 CI/CD파일 생성

* {애플리케이션 유형}: backend 또는 frontend로 지정
* {Team ID}: 본인 팀의 id를 입력
* {root project}: 애플리케이션 root 프로젝트명
* {서비스명}: 서브 프로젝트명을 쉼표로 구분하여 입력

프론트엔드 프롬프트 샘플입니다.

deploy\_env\_var에서 backend service url은 deployment/manifests/configmaps에 있는 yaml 내용에서 runtime-env.js파일의 내용을 참고하세요.

| p: CI/CD 파일을 작성해 주세요.  [요구사항]  - 네임스페이스: {Team ID}-lifesub-ns  - deploy\_env\_var에서 export\_port는 '18080'으로 해주세요.  - 배포 후 Service Object의 External IP가 부여될때까지 기다리고 완료되면 표시해 주세요.  [환경변수]  - {애플리케이션 유형}: frontend  - {Team ID}: unicorn  - {root project}: lifesub-web  - {서비스명}: lifesub-web |
| --- |

백엔드 프롬프트에 샘플입니다.

- ‘mysub’서비스에 대한 내용은 구독추천 서비스에 특화된 내용입니다. Clean 아키텍처 패턴이 적용되어 프로젝트가 2개로 나뉘어져 있기 때문에 언급한 겁니다. 본인의 서비스에 맞게 수정하거나 지워야 합니다.

- allowed\_origins는 Frontend 서비스 오브젝트의 External IP를 지정하면 됩니다.

- {서비스명}은 settings.gradle에 정의된 마이크로서비스 명을 지정하면 됩니다.(common은 제외)

| p: CI/CD 파일을 작성해 주세요.  [요구사항]  - 네임스페이스: {Team ID}-lifesub-ns  - mysub 서비스는 gradel build경로를 'mysub-infra'로 하고 jar파일은 'mysub.jar"으로 해주세요.  - ConfigMap 'common-config'의 'ALLOWED\_ORIGINS'는 변수 처리 해주세요.  - deploy\_env\_var에서 allowed\_origins는 'http://4.127.33.33’으로 해주세요.  [환경변수]  - {애플리케이션 유형}: backend  - {Team ID}: unicorn  - {root project}: lifesub  - {서비스명}: member, mysub, recommend |
| --- |

- 제공 받은 파일들을 프로젝트의 deployment디렉토리 밑에 생성합니다.

- 원격 Git repository에 업로드

| git add . && git commit -m "Add CI/CD" && git push |
| --- |

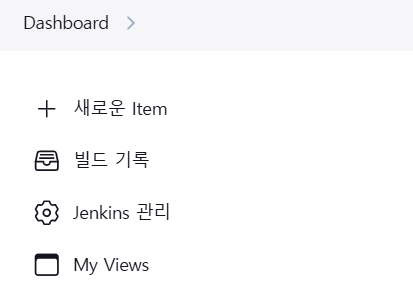
※ deploy.yaml.template에 보안에 위반되는 credential값 때문에 업로드가 안될 수 있음.

이때는 Push 시 메시지를 잘 보고 업로드 허용을 해 줘야 함.

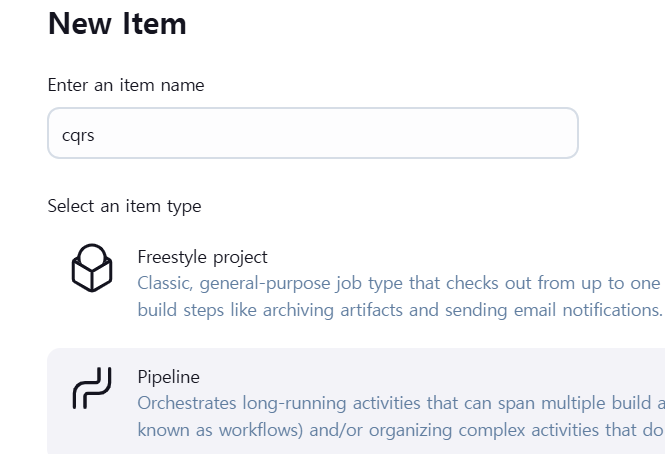
## 2. Pipeline 작성 및 구동

### 1) Jenkins Pipeline 작성

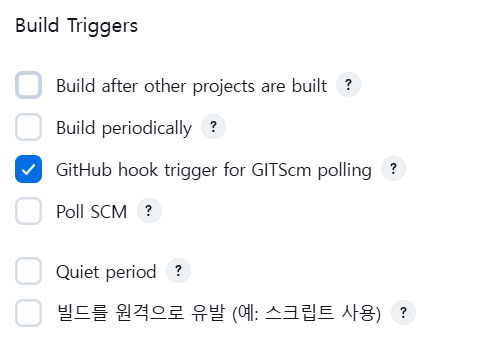
- ‘새로운 Item’ 클릭



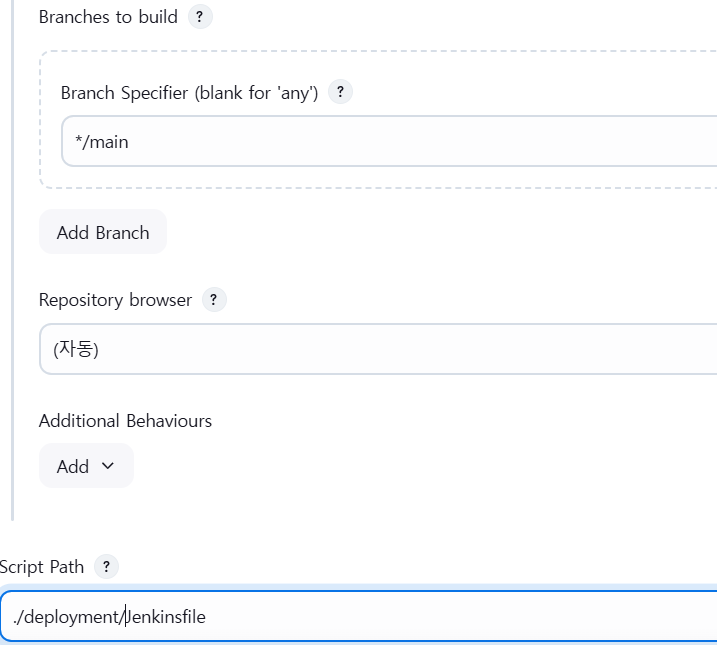
- 이름 입력 후 ‘Pipeline’ 선택



- 아래 예제 참고하여 설정



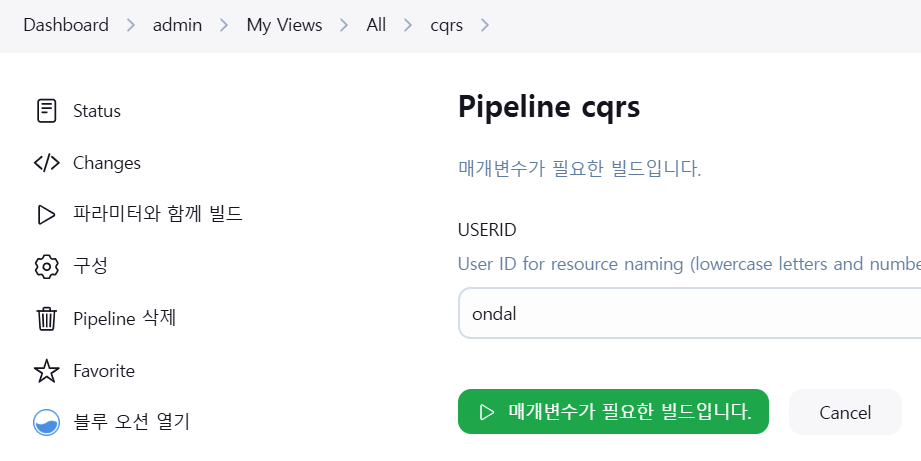




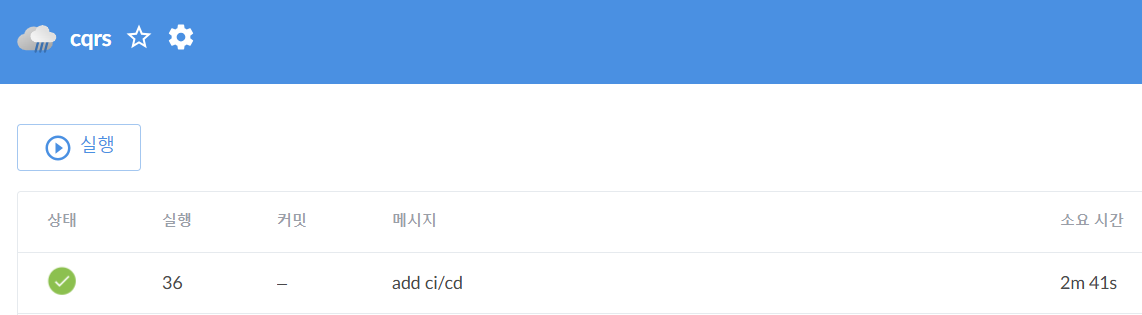
### 2) Pipeline 구동

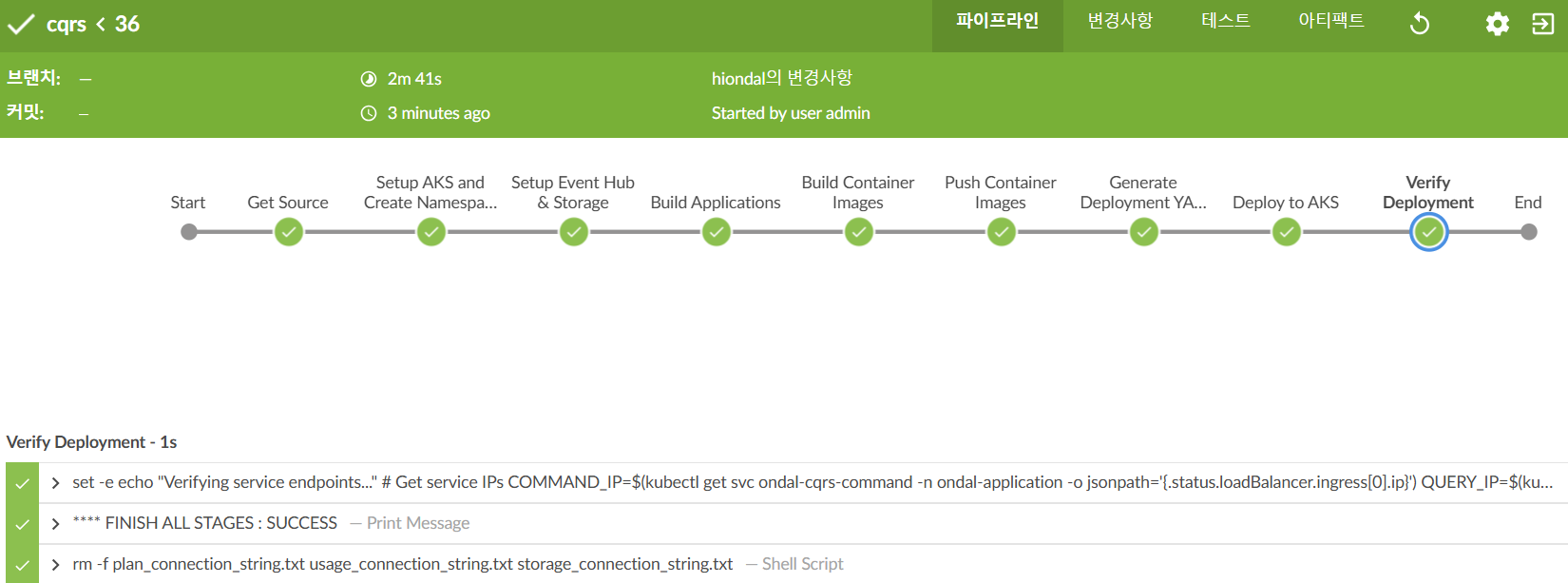
Jenkins파일에서 파라미터를 지정한 경우는 ‘파라미터와 함께 빌드’를 클릭

아니면 ‘지금 빌드’를 클릭합니다.



‘블루오션열기’를 클릭하면 더 직관적인 UI로 진행상황을 볼 수 있습니다.





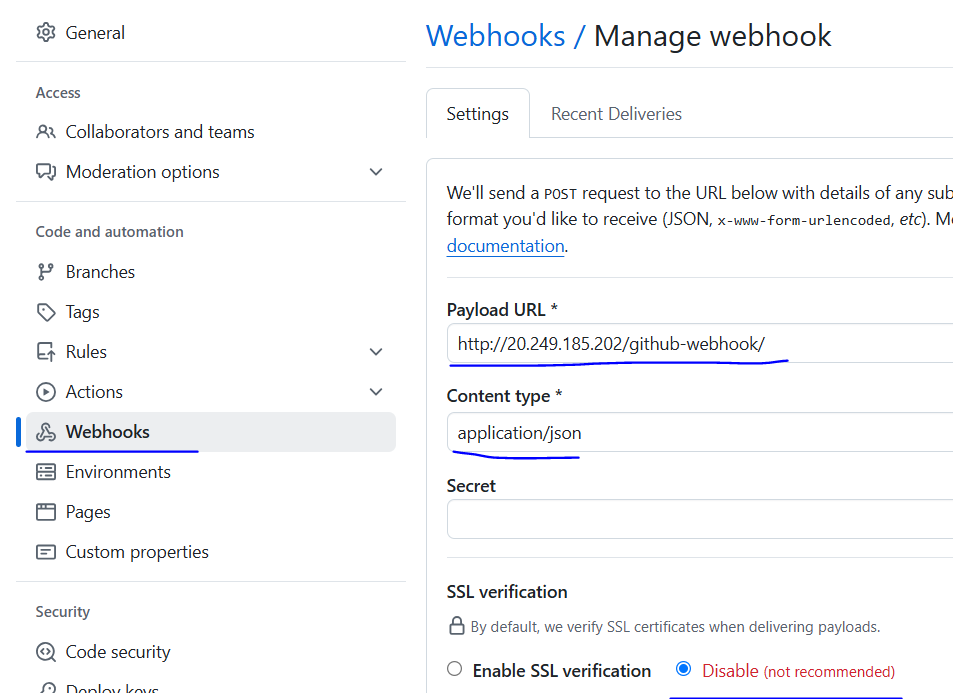
## 3. Webhook 설정

Git push 시 자동으로 pipeline이 구동되게 하려면 아래와 같이 github repository에 webhook 설정을 합니다.

Git repo의 Settings를 누르고 아래 예와 같이 셋팅하세요.

Payload URL의 IP는 Jenkins service의 L/B IP를 지정하셔야 합니다.

Payload URL 마지막에 반드시 ‘/’를 붙여야 합니다.



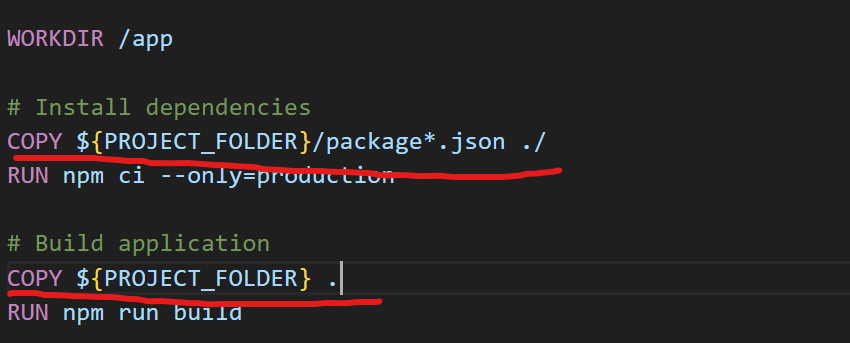
## 4. 프론트엔드 이미지 빌드 성능 향상 TIP

**1) .dockerignore 파일 만들기**

최상위 디렉토리 밑에 아래 내용으로 만듭니다.

| images node\_modules npm-debug.log build .git .github coverage .env\* .cache dist logs \*\*/\*.log \*\*/.DS\_Store |
| --- |

Dockerfile에서 아래 COPY명령 수행 시 .dockerignore에 있는 파일은 복사하지 않습니다. 또한 파일 scan에서도 제외하기 때문에 속도가 올라갑니다.



**2) Library가 이미 설치된 Base image 이용하기**

- 라이브러리가 설치된 Base Image 만들기

deployment/container/Dockerfile-{frontend service}파일의 내용에서 ‘npm ci …’부분까지만 복사합니다.

| # Build stage FROM node:20-slim AS builder ARG PROJECT\_FOLDER  ENV NODE\_ENV=production  WORKDIR /app  # Install dependencies COPY ${PROJECT\_FOLDER}/package\*.json ./ RUN npm ci --only=production |
| --- |

deployment/container/Dockerfile-base라는 이름으로 파일을 만들고 붙여넣기 합니다. 그리고 아래와 같이 수정합니다.

${PROJECT\_FOLDER}를 사용하지 않게 변경하는 부분만 바꾸면 됩니다.

| # Build stage FROM node:20-slim AS builder  ENV NODE\_ENV=production  WORKDIR /app  # Install dependencies COPY ./package\*.json ./ RUN npm ci --only=production |
| --- |

이 Dockerfile을 이용하여 라이브러리가 설치된 base 이미지를 만듭니다.

환경변수는 본인 팀 것으로 변경해야 합니다.

프론트엔드 디렉토리로 이동하셔서 수행하세요.

| export ACR\_NAME="unicorncr"  export IMG\_ORG="cms"  export IMG\_NAME="cms-front-base"  docker build -f deployment/container/Dockerfile-base -t ${ACR\_NAME}.azurecr.io/${IMG\_ORG}/${IMG\_NAME}:latest . |
| --- |

Base 이미지를 ACR에 푸시합니다.

| docker push ${ACR\_NAME}.azurecr.io/${IMG\_ORG}/${IMG\_NAME}:latest |
| --- |

- Base Image를 이용한 Dockerfile 만들기

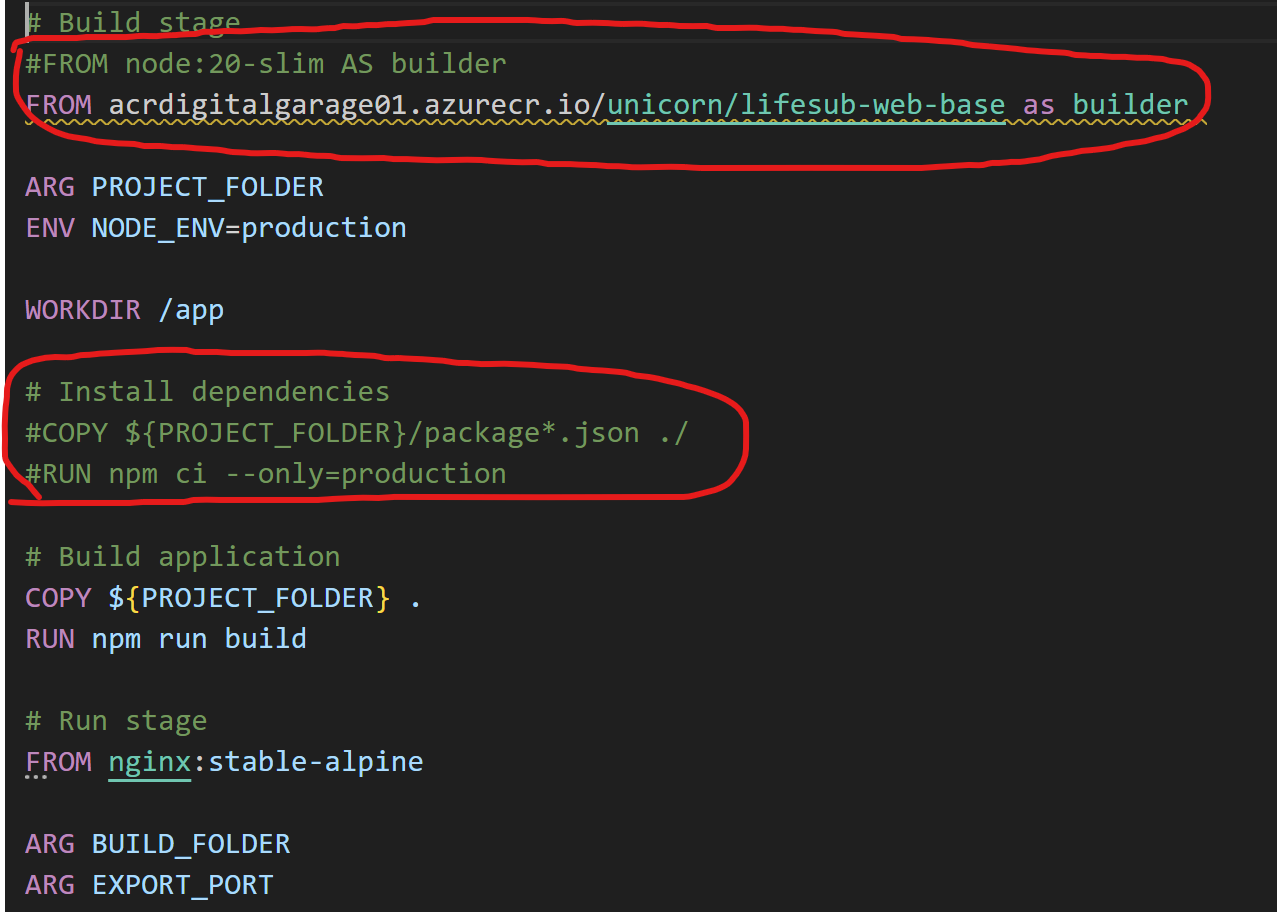
기존 ‘Dockerfile-{frontend service}’파일은 백업하거나 지우고,

deployment/container/Dockerfile-{frontend service} 파일의 내용을 변경합니다.

- 변경할 부분은 아래와 같습니다.

- FROM 뒤의 base image: 위에서 만든 image 경로와 일치해야 합니다.(:latest 태그는 생략해도 됩니다.)

- Install dependencies 부분 리마크



예시)

| # Build stage FROM unicorncr.azurecr.io/cms/cms-front-base:latest AS builder  ARG PROJECT\_FOLDER  ENV NODE\_ENV=production WORKDIR /app  # Build application COPY ${PROJECT\_FOLDER} . RUN npm run build  # Run stage FROM nginx:stable-alpine  ARG BUILD\_FOLDER ARG EXPORT\_PORT  # Create nginx user if it doesn't exist RUN adduser -S nginx || true  # Copy build files COPY --from=builder /app/build /usr/share/nginx/html  # Copy and process nginx configuration COPY ${BUILD\_FOLDER}/nginx.conf /etc/nginx/templates/default.conf.template  # Add custom nginx settings RUN echo "client\_max\_body\_size 100M;" > /etc/nginx/conf.d/client\_max\_body\_size.conf RUN echo "proxy\_buffer\_size 128k;" > /etc/nginx/conf.d/proxy\_buffer\_size.conf RUN echo "proxy\_buffers 4 256k;" > /etc/nginx/conf.d/proxy\_buffers.conf RUN echo "proxy\_busy\_buffers\_size 256k;" > /etc/nginx/conf.d/proxy\_busy\_buffers\_size.conf  # Set permissions RUN chown -R nginx:nginx /usr/share/nginx/html && \  chmod -R 755 /usr/share/nginx/html && \  chown -R nginx:nginx /var/cache/nginx && \  chown -R nginx:nginx /var/log/nginx && \  chown -R nginx:nginx /etc/nginx/conf.d && \  touch /var/run/nginx.pid && \  chown -R nginx:nginx /var/run/nginx.pid  USER nginx  EXPOSE ${EXPORT\_PORT}  CMD ["nginx", "-g", "daemon off;"] |
| --- |

CI/CD Pipeline에서 이제 이 파일을 이용하면 됩니다.

- 수동으로 이미지 빌드/푸시 하기

새로 만든 Dockerfile을 이용하여 이미지를 빌드합니다.

환경변수는 본인 팀 것으로 변경해야 합니다.

‘BUILD\_FOLDER’는 nginx.conf가 있는 디렉토리를 지정해야 합니다.

IMG\_TAG는 ‘월일시분’형식으로 했습니다. 본인에 맞게 바꾸셔도 됩니다.

프론트엔드 디렉토리로 이동하셔서 수행하세요.

| export ACR\_NAME="unicorncr"  export IMG\_ORG="cms"  export IMG\_NAME="cms-front"  export IMG\_TAG="03201052"  docker build -f deployment/container/Dockerfile-base \  --build-arg PROJECT\_FOLDER=. \  --build-arg BUILD\_FOLDER=deployment/container \  --build-arg EXPORT\_PORT=18080 \  -t ${ACR\_NAME}.azurecr.io/${IMG\_ORG}/${IMG\_NAME}:${IMG\_TAG} . |
| --- |

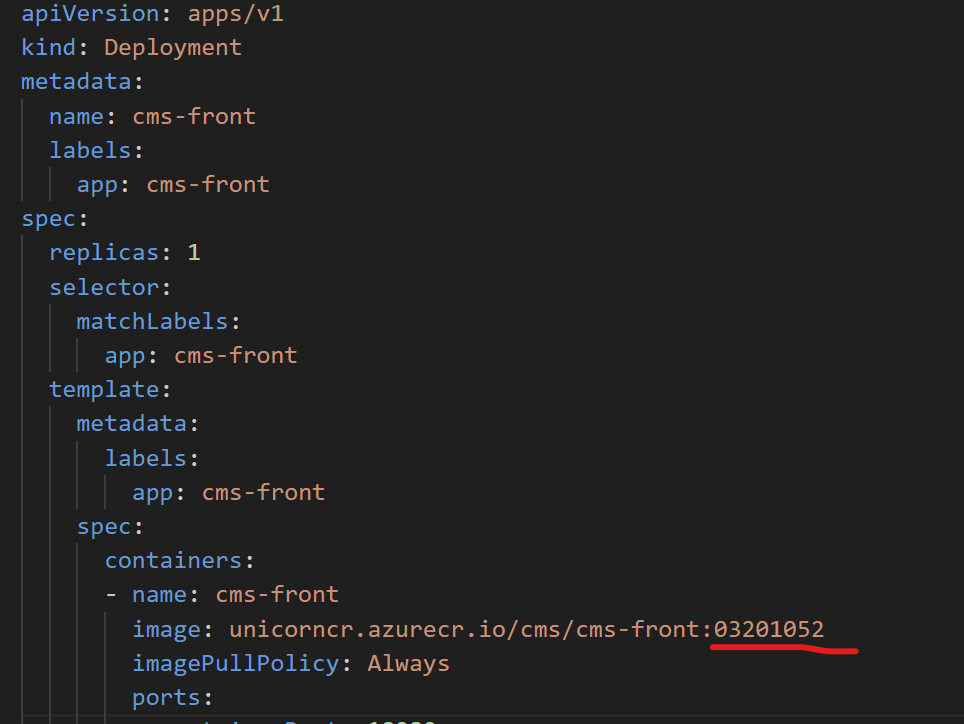
이미지를 ACR에 푸시합니다.

| docker push ${ACR\_NAME}.azurecr.io/${IMG\_ORG}/${IMG\_NAME}:${IMG\_TAG} |
| --- |

- 파드 재 배포하기

Manifest Repository에서 프론트엔드 배포를 위한 deployment yaml파일에서 Image Tag를 변경합니다.

위에서 지정한 IMG\_TAG값과 동일해야 합니다.



ArgoCD에서 동기화를 수행하여 재배포합니다.

※ 수동배포할 때는 Jenkins를 이용한 CI 파이프라인은 중지해야 합니다.

아래와 같이 파이프라인 프로파일에서 ‘GitHub hook’을 uncheck합니다.



# III. GitHub Action

## 1. 사전작업

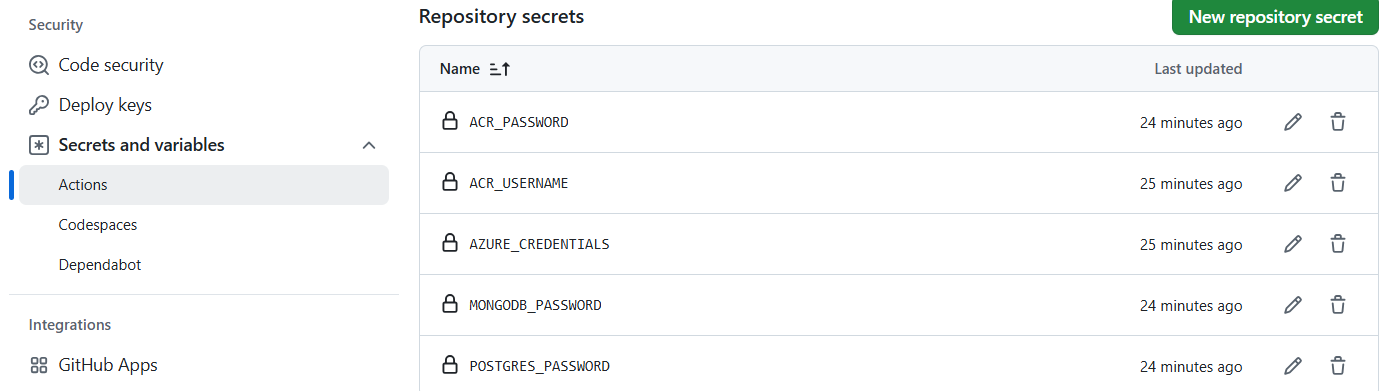
### 1) ‘I. 사전작업’ 수행

### 2) ‘II. Jenkins > 1. CI/CD 파일 작성’ 수행

## 2. Secret 등록

실행에 필요한 Secret을 등록합니다.

GitHub Repository의 Settings > Secrets and variables > Actions에 등록합니다.



### 1) AZURE\_CREDENTIALS 추가

‘Jenkins > 2. Pipeline 사전작업 > 1) azure 리소스 접근 권한 부여 받기’ 참조

| {  "clientId": "5e4b5b41-7208-48b7-b821-d6d5acf50ecf",  "clientSecret": "ldu8Q~GQEzFYU.dJX7\_QsahR7n7C2xqkIM6hqbV8",  "subscriptionId": "2513dd36-7978-48e3-9a7c-b221d4874f66",  "tenantId": "4f0a3bfd-1156-4cce-8dc2-a049a13dba23",  "activeDirectoryEndpointUrl": "https://login.microsoftonline.com",  "resourceManagerEndpointUrl": "https://management.azure.com/",  "activeDirectoryGraphResourceId": "https://graph.windows.net/",  "sqlManagementEndpointUrl": "https://management.core.windows.net:8443/",  "galleryEndpointUrl": "https://gallery.azure.com/",  "managementEndpointUrl": "https://management.core.windows.net/" } |
| --- |

### 2) 나머지 변수 생성

아래 명령으로 ACR credential 정보 구함함

| az acr credential show --name {acr 이름} |
| --- |

아래 변수를 등록합니다.

* ACR\_USERNAME: 맨 아래 username입력
* ACR\_PASSWORD: 첫번째 암호를 입력

## 3. CI/CD yaml 작성

### 1) 참고자료 업로드

사전수행 결과로 생성된 아래 파일들을 Claude project Knowledge에 등록합니다.

1.1) 백엔드 애플리케이션

- Dockerfile-{서비스명}. 한 파일 사용하면 Dockerfile로 등록

- deploy.yaml.template

- deploy\_env\_vars

- Jenkinsfile

1.2) 프론트엔드 애플리케이션

- Dockerfile-{서비스명}

- deploy.yaml.template-front

- deploy\_env\_vars-front

- Jenkinsfile-front

### 2) Instruction에 추가

| [GitHub Action 가이드]  'g:'로 시작하면 {애플리케이션 유형}을 위한 GitHub Action CI/CD 가이드를 요청하는 것임.  {애플리케이션 유형}은 ‘backend’ 또는 ‘frontend’임  {요청사항}  - 참고자료를 바탕으로 github Action파일인 'cicd.yaml'을 작성  - actions/upload-artifact 액션은 최신 버전을 사용해 주세요.  - Build, Release, Run 단계를 분리  - 환경설정은 deployment/deploy\_env\_vars에서 읽어야 함  - deploy\_env\_vars파일 내용에는 '#'으로 주석이 있음에 주의  - Image Tag 처리  - release job에 outputs 섹션을 추가하여 이미지 태그를 다음 job으로 전달  - timestamp 생성 방식은 GITHUB\_OUTPUT 사용  - deploy job의 manifest 생성 단계에서 이미지 태그를 명시적으로 환경변수로 설정  - SonarQube와의 연동하여 코드품질과 테스트코드 커버리지를 측정  - GiuHub Secret이용: AZURE\_CREDENTIALS, ACR\_USERNAME, ACR\_PASSWORD, SONAR\_HOST\_URL, SONAR\_TOKEN  - 디버깅을 위해 생성된 manifest 파일 출력  {참고자료}  - Dockerfile-{서비스명}. 만약 없으면 Dockerfile 참조  - backend  - deploy.yaml.template  - deploy\_env\_vars  - Jenkinsfile  - frontend  - deploy.yaml.template-front  - deploy\_env\_vars-front  - Jenkinsfile-front  [응답형식]  코드블록 |
| --- |

### 3) CI/CD yaml 작성

- cicd.yaml 작성 요청

{서비스명}은 본인의 서비스들을 쉼표로 구분하여 작성.

on.push.branches는 푸시 대상 원격 레포지토리 branch를 지정하세요.

백엔드 애플케이션 예제

| g: CI/CD 파일을 작성해 주세요.  - instruction의 '[GitHub Action 가이드]'의 지침을 준수해서 만들어 주세요.  - on.push.branches: k8s  - 환경변수  - {애플리케이션 유형}: backend  - {서비스명}: alert, usage |
| --- |

프론트엔드 애플리케이션 예제

| g: CI/CD 파일을 작성해 주세요.  - instruction의 '[GitHub Action 가이드]'의 지침을 준수해서 만들어 주세요.  - on.push.branches: main  - 환경변수  - {애플리케이션 유형}: frontend  - {서비스명}: lifesub-web |
| --- |

- cicd.yaml 생성

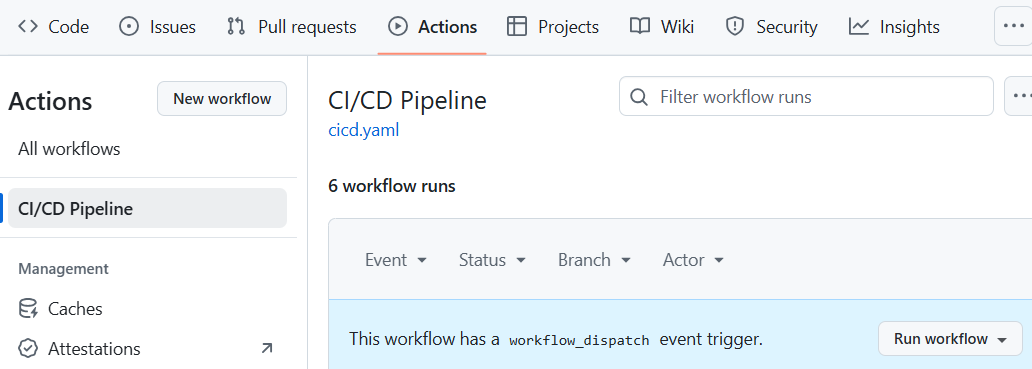
위 프롬프트의 결과 내용을 복사하여, ‘.github/workflows’ 디렉토리를 생성하고, cicd.yaml파일명으로 생성.

## 3. 실행

Push하면 자동으로 실행됩니다.

수동으로 실행하려면,

Actions > CI/CD Pipeline 클릭 후 ‘Run workflow’ 실행



# IV ArgoCD

## 1. Manifest repository 만들기

Manifest 를 관리하는 repository를 만듭니다.

deployment/manifest 디렉토리 하위 폴더와 yaml 파일을 이용하여 만듭니다.

## 2. ArgoCD Project와 Application 작성

아래 링크를 참고하여 만드십시오.

[ArgoCD 설치](https://happycloud-lee.tistory.com/293)

## 3. Jenkinsfile\_ArgoCD 파일 작성

아래 예제 프롬프트를 참고하여 작성합니다.

Backend deployment yaml과 frontend deployment yaml의 경로는 manifest repository의 구조를 보고

정확히 동일하게 맞춰야 합니다.

예1) 리소스 유형별 관리인 경우

| ArgoCD로 배포관리를 하려고 합니다.  Image 명을 변경하여 재배포가 되도록 수정하여 주십시오.  Knowledge의 Jenkinsfile과 Jenkinsfile-front를 변경하여 주십시오.  [manifest repository]  - Jenkins Credential: github-credentials  - 주소: https://github.com/cna-bootcamp/lifesub-manifest.git  - backend deployment yaml: lifesub/deployments/{backend service}-deployment.yaml  - {backend service}: member, mysub, recommend  - frontend deployment yaml: lifesub-web/deployments/lifesub-web-deployment.yaml |
| --- |

예2) 서비스별 관리인 경우

| ArgoCD로 배포관리를 하려고 합니다.  Image 명을 변경하여 재배포가 되도록 수정하여 주십시오.  Knowledge의 Jenkinsfile과 Jenkinsfile-front를 변경하여 주십시오.  [manifest repository]  - Jenkins Credential: github-credentials  - 주소: https://github.com/ktdsgarage/cms-manifest.git  - backend deployment yaml: cms-backend/{backend service}/{backend service}-deployment.yaml  - {backend service}: authentication, content-management, token-management  - frontend deployment yaml: cms-front/cms-front-deployment.yaml |
| --- |

## 4. Git 인증 Jenkins Credential 작성

‘github-credentials’라는 이름으로 만듭니다.

암호는 GitHub로그인 암호가 아니라 Git Access Token을 입력해야 합니다.

## 5. GitHub Action 정지

배포 중복을 방지하기 위해 GitHub Action은 중지합니다.  
백엔드와 프론트엔드의 .github/workflow/cicd.yaml에서 on.push.branches를 주석처리 합니다.

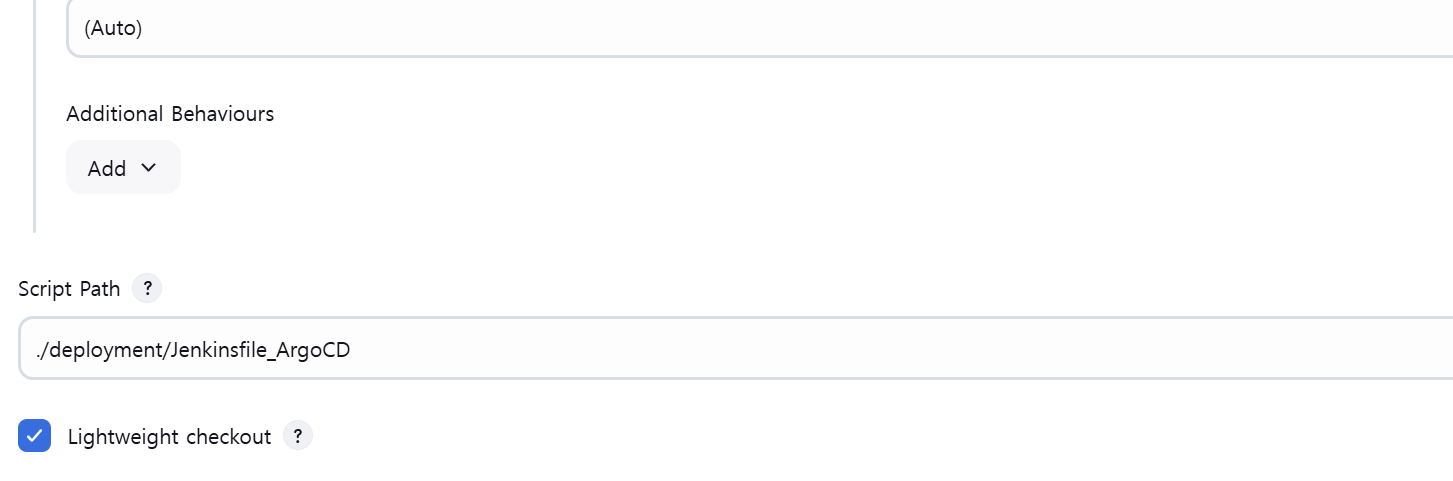
on:

push:

#branches: [ cicd ]

paths:

## 6. Pipeline 수정

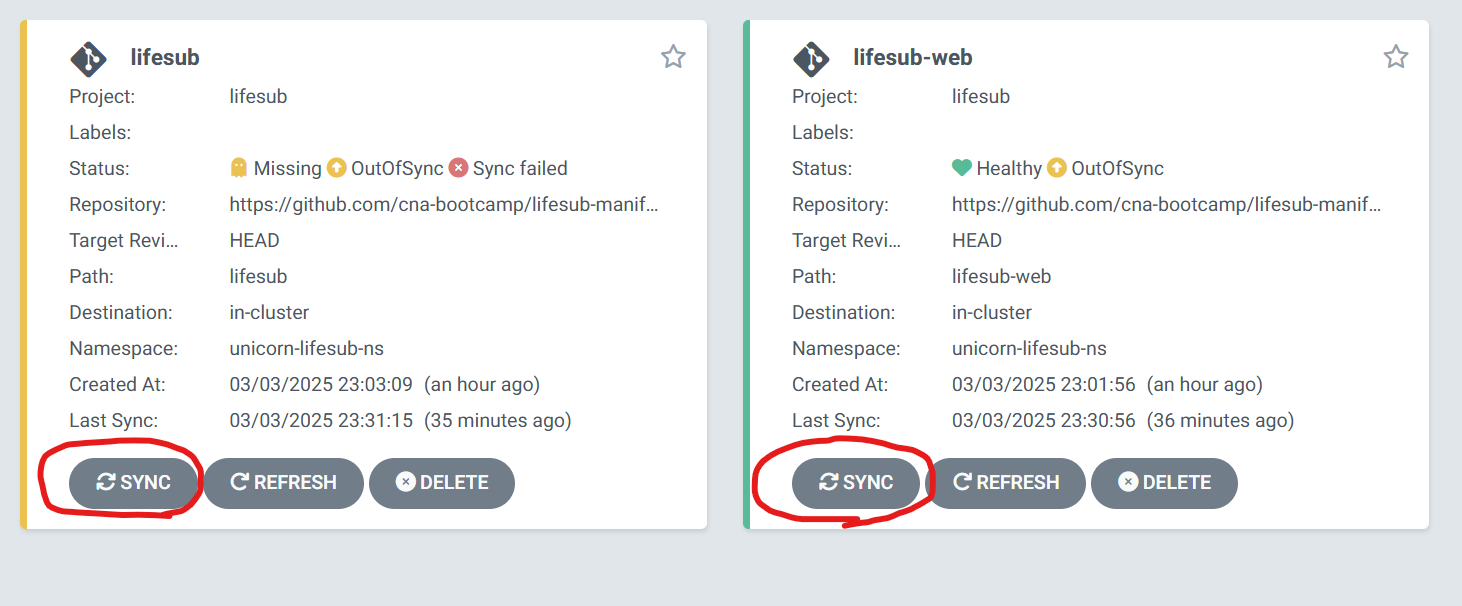
기존 Jenkins pipeline에서 Jenkinsfile의 이름을 Jenkinsfile\_ArgoCD로 변경합니다.  


## 7. 테스트

소스를 Push합니다.

GitHub webhook에 의해 자동으로 Jenkins 파이프라인이 구동됩니다.

완료 후 lifesub-manifest의 deployments디렉토리 하위의 manifest에서 image명이 변경되었는지 확인합니다.

ArgoCD에서 'SYNC'를 눌러 동기화 합니다.  


배포가 제대로 되었는지 확인합니다.

# V. Frontend와 Backend 연결

## 1. Ingress 확인

| k get ing |
| --- |

아래 예와 같이 1개가 나와야 합니다.

2개가 나오면 정확한 것만 남기고 삭제 합니다.

Manifest repository의 ingress yaml을 확인하여 사용중인 ingress가 어떤것인지 확인합니다.

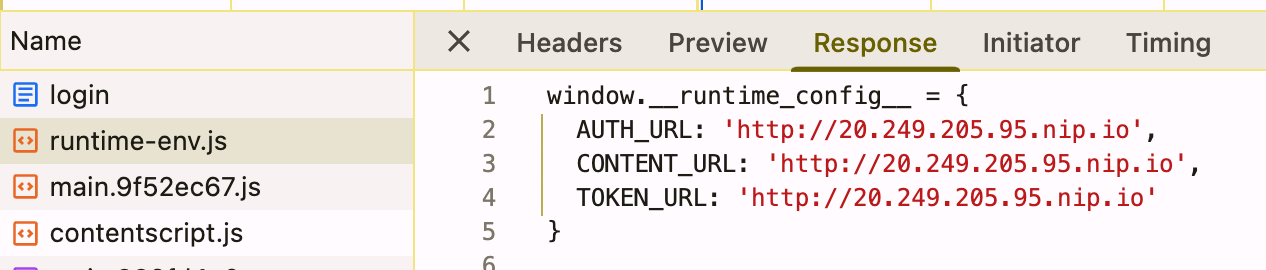
| k get ing NAME CLASS HOSTS ADDRESS PORTS AGE cms-ingress <none> 20.249.205.95.nip.io 20.249.205.95 80 4h53m |
| --- |

HOSTS에 있는 값이 frontend에서 호출하는 주소와 일치해야 합니다.

만약 ‘\*’로 되어 있다면 ingress external ip로 호출하면 됩니다.

브라우저에서 프론트엔드를 주소로 접근하여 디버그 창을 여십시오.

그리고 ‘Network’탭을 클릭하여 runtime-env.js의 내용을 확인하십시오.



만약 틀리다면 manifest repository의 frontend의 ConfigMap을 고치십시오.

예)

| apiVersion: v1 kind: ConfigMap metadata:  name: cms-front-config data:  runtime-env.js: |  window.\_\_runtime\_config\_\_ = {  AUTH\_URL: 'http://20.249.205.95.nip.io',  CONTENT\_URL: 'http://20.249.205.95.nip.io',  TOKEN\_URL: 'http://20.249.205.95.nip.io'  } |
| --- |

정확한 값을 찾으려면 아래와 같이 하십시오.

- 아래 값을 사용하는 프론트엔드의 backend api를 호출하는 소스 확인

- ingress yaml 확인

예)

- api 호출 확인

| // API 기본 URL 설정 const API\_URL = window.\_\_runtime\_config\_\_?.AUTH\_URL || 'http://localhost:8080';  // 로그인 API export const login = async (username, password) => {  try {  console.log('로그인 시도:', { username });    const response = await fetch(`${API\_URL}/api/auth/login`, { |
| --- |

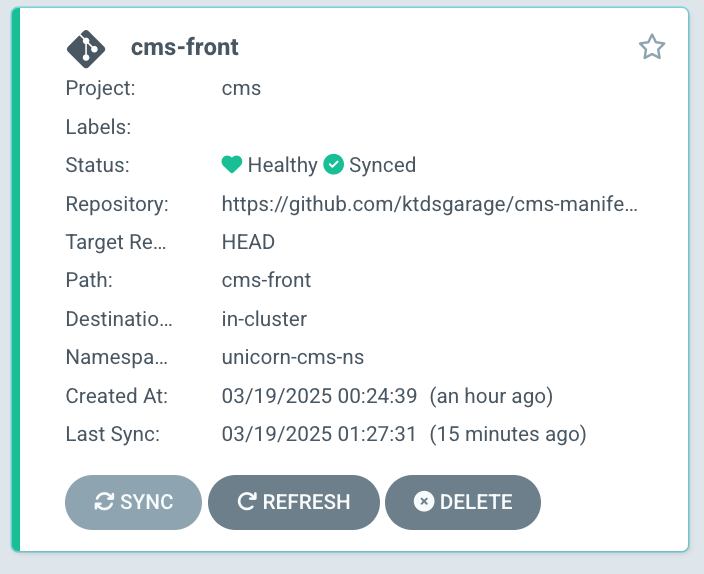
- Ingress yaml 확인: manifest repository에서 확인

| apiVersion: networking.k8s.io/v1 kind: Ingress metadata:  name: cms-ingress  annotations:  kubernetes.io/ingress.class: "nginx" spec:  rules:  - host: 20.249.205.95.nip.io  http:  paths:  - path: /api/auth  pathType: Prefix  backend:  service:  name: authentication-service  port:  number: 80 |
| --- |

Manifest repository로 수정 내용을 git push 하세요.

| git add . && git commit -m "update runtime-env.js" && git push |
| --- |

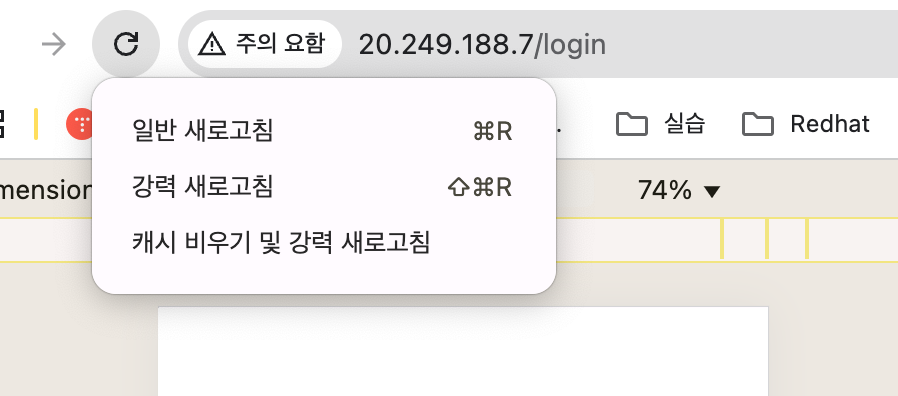
ArgoCD에서 ‘SYNC’ 버튼을 눌러 ConfigMap을 업데이트 합니다.



Frontend Pod를 삭제하여 재시작합니다. ConfigMap 변경 내용이 Pod실행 시 주입되어 반영됩니다.

디버그 창에서 runtime-env.js의 내용이 변경 되었는지 확인 합니다.

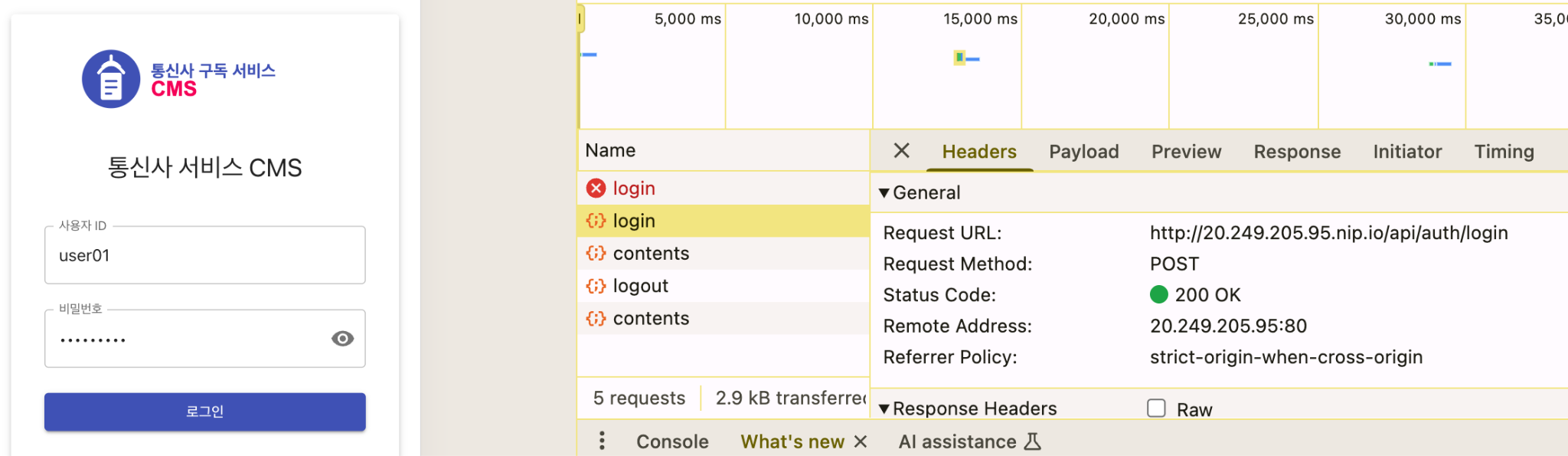
만약 변경이 안되었다면 아래 그림과 같이 리프레시 아이콘을 꾹 누르고 ‘캐시 비우기 및 강력 새로고침’을 수행하여 캐시를 삭제 합니다.



## 2. 로그인이 안될 때

로그인 시 URL을 디버그 창에서 확인합니다.

[http://{ingress](about:blank) host}/{api 경로}가 정확한지 확인합니다.



그래도 안되면 manifest repository에서

backend 서버의 ConfigMap에 지정된 ALLOWED\_ORIGINS값에 지정된 frondend IP가 정확한지 확인합니다.

예)

| apiVersion: v1 kind: ConfigMap metadata:  name: authentication-config data:  SPRING\_APPLICATION\_NAME: "authentication-service"  POSTGRES\_HOST: "postgres-auth-postgresql"  POSTGRES\_PORT: "5432"  POSTGRES\_DB: "authdb"  ALLOWED\_ORIGINS: "http://localhost:3000,http://20.249.188.7" |
| --- |

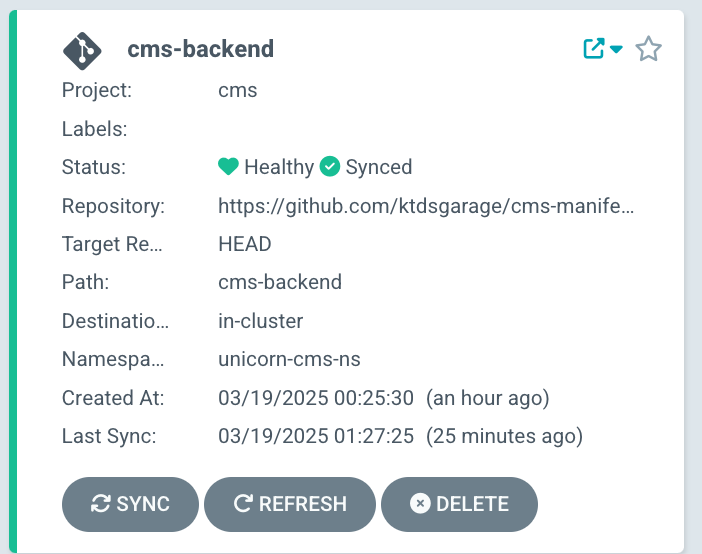
만약 틀리다면 변경하십시오.

각 서비스의 ConfigMap을 모두 확인하여 정확하게 맞추십시오.

Manifest repository로 수정 내용을 git push 하세요.

| git add . && git commit -m "update runtime-env.js" && git push |
| --- |

변경 사항 반영을 위해 ArgoCD에서 백엔드 서버의 동기화를 수행합니다. ‘SYNC’버튼을 클릭하십시오.



ConfigMap의 내용이 파드에 적용되기 위해 백엔드 파드를 모두 삭제하여 재시작 합니다.

그리고 다시한번 브라우저에서 테스트 합니다.

## 3. 불필요한 파일 삭제

ArgoCD와 연동하면 이제 소스 레포지토리에서 아래 파일들은 필요 없습니다.

안지워도 되지만 혼선을 방지하기 위해 삭제 하십시오.

* deployment/manifest 디렉토리 전체 삭제
* deploy.yaml.template 파일
* Jenkinsfile