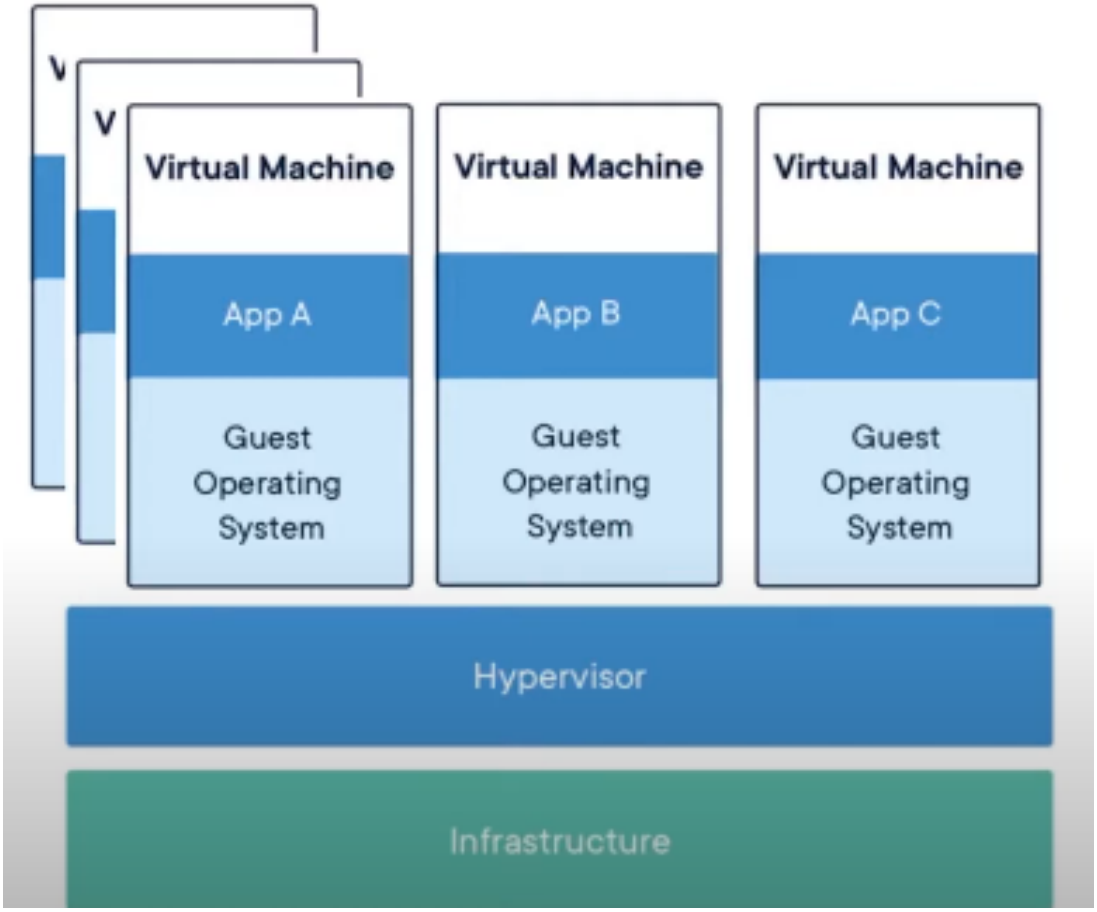
INFRA & DATA

INFRA

▼ 쿠버네티스

▼ 쿠버네티스가 도대체 무엇인가?

가상머신 vs 컨테이너?



만약 내일이 sale이다 그럼 해당 VM을 여러대 생성 하여 하나가 다운 되어도 다른 걸로 쓸 수 있게 해준다.

컨테이너가 가상머신보다 훨씬 더 가볍다

ex) Nginx 실제 사이즈 100MB 가 돌아가기 위해서 WEB서버는 1GB에서 2GB는 필요하다.

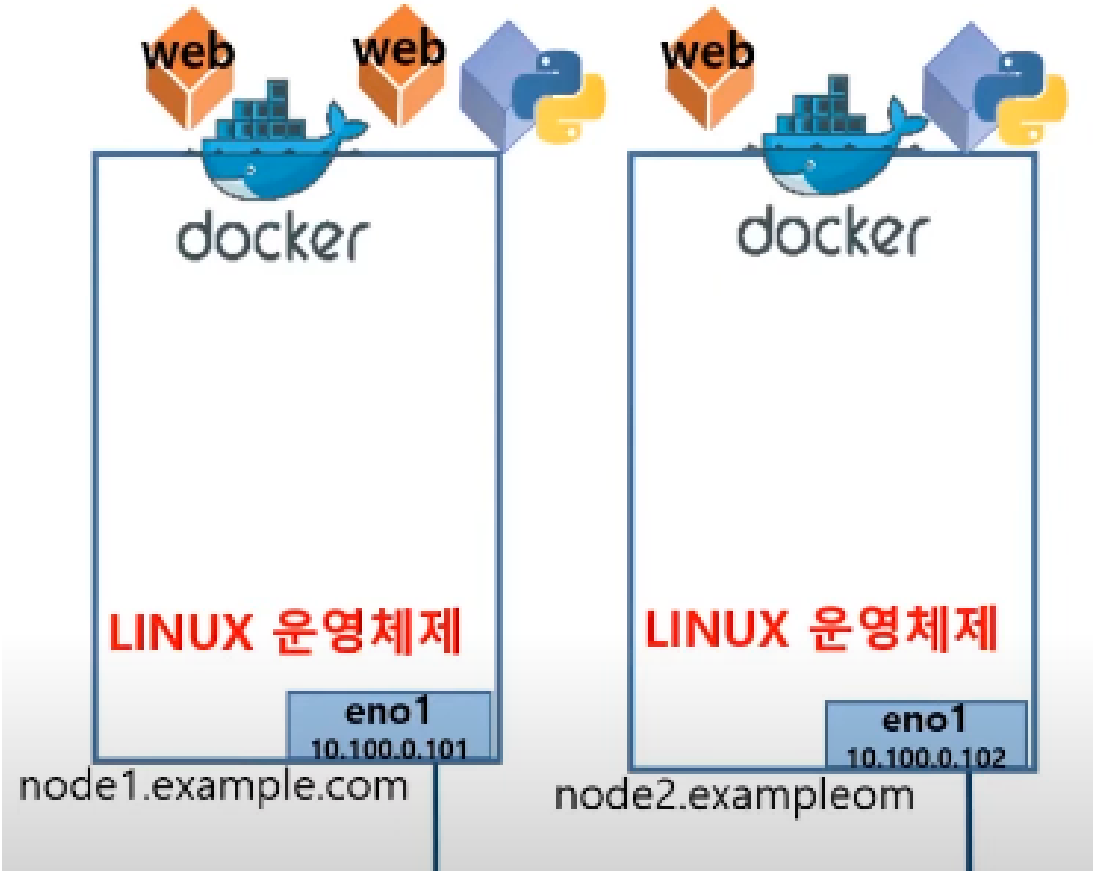
따라서 VM을 복제하게 되면 100MB +1~2GB로 엄청 무거워지게 된다.

하지만 컨테이너는 웹 서버와 웹 컨텐츠만 들어가있기에 굉장히 가벼워 진다. 따라서 확장성이 컨테이너가 좋다. → **배포에 엄청난 장점이 있다.**

따라서 도커 컨테이너에 웹서버를 컨테이너로 올려 여러개로 실행 할 수 있다.

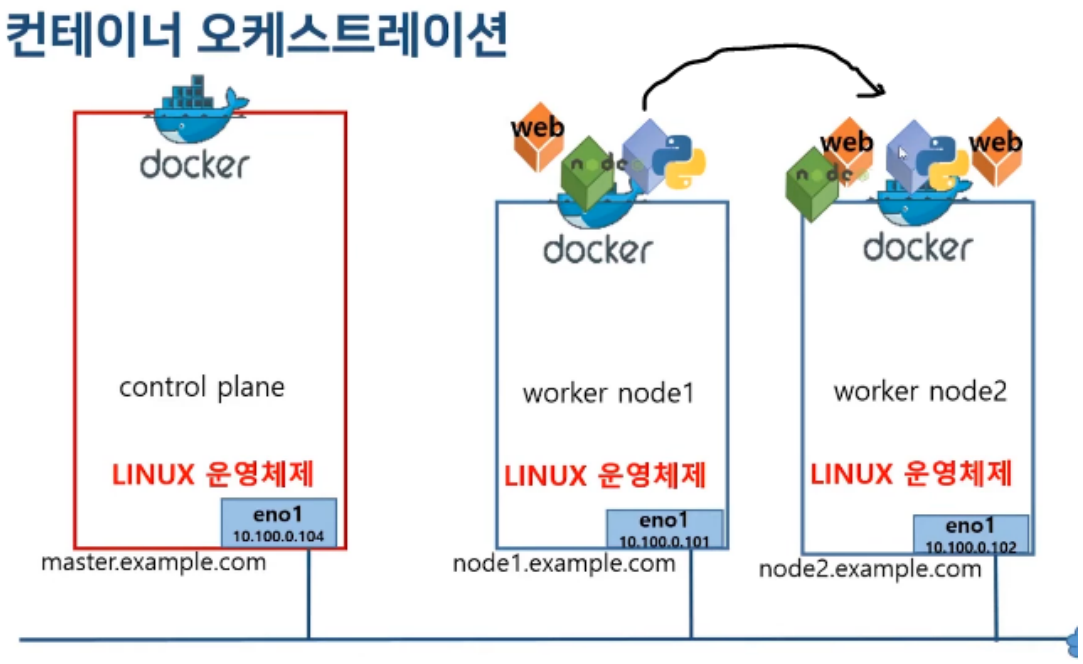
if 시스템 자체가 다운되어 버린다면?

멀티호스트 도커 플랫폼을 운영하자.

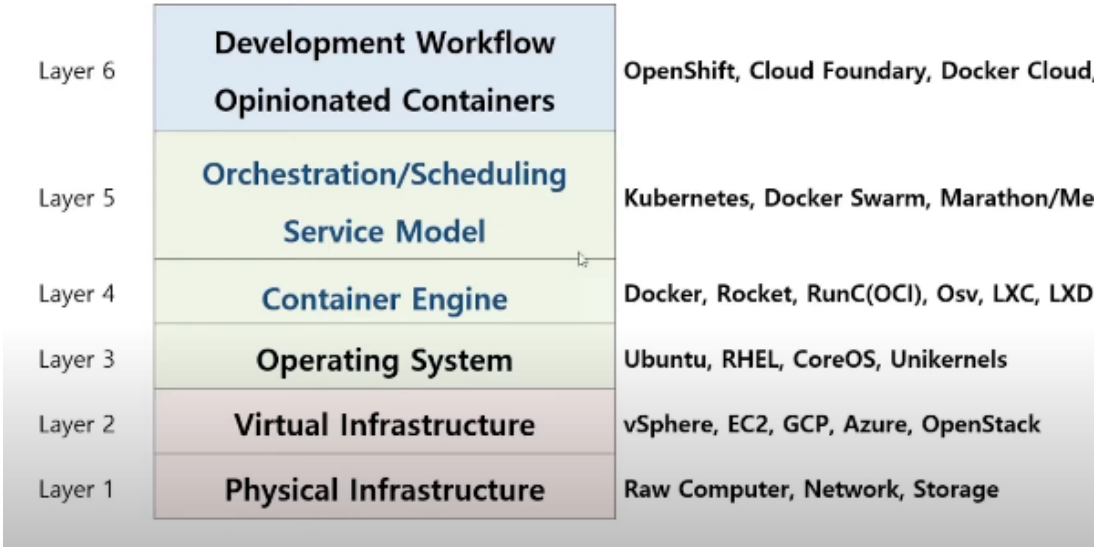


→ 이렇게 되면 컨테이너 하나하나를 개발자가 관리하게 되면 정말 힘들다.

→ 따라서 오케스트레이션 시스템이 필요하다.



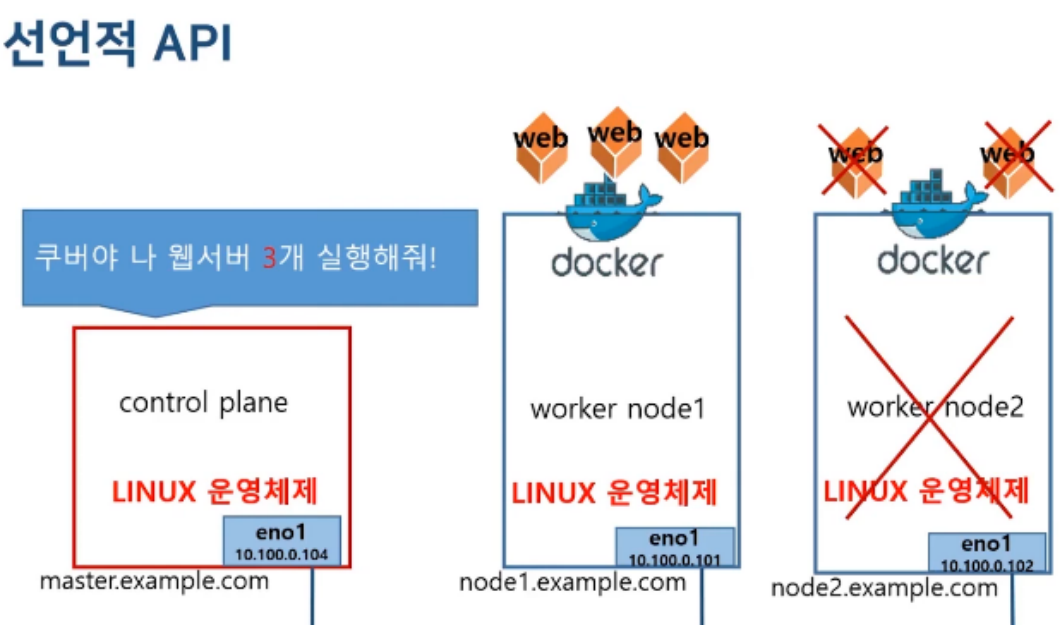
컨테이너 계층 구조



ex) 네트워크에 EC2를 얹고 EC2 위에 Ubuntu라는 Operating System을 얹는다. 그리고 Docker를 얹고 Kubernates를 통해 관리한다.

쿠버네티스의 특징

- 선언적 API



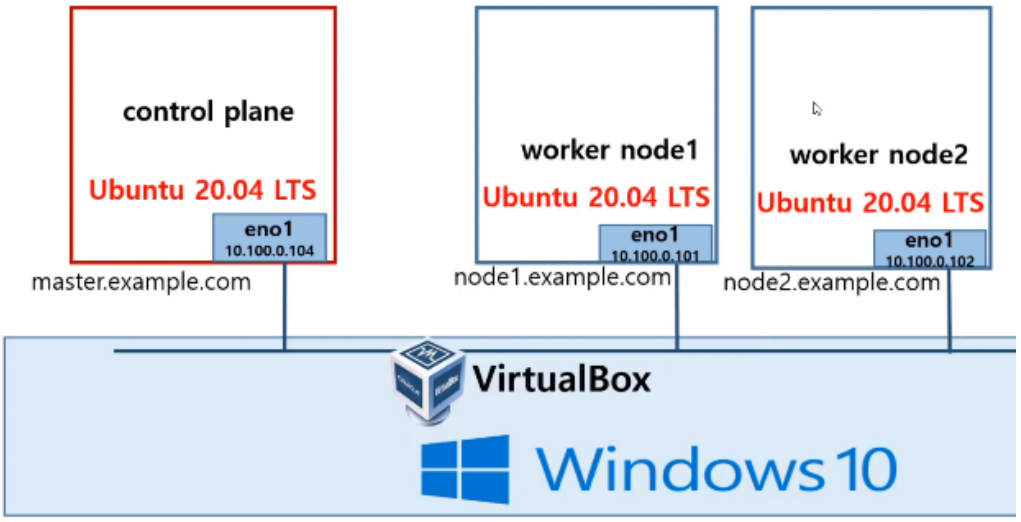
▼ 쿠버네티스 설치하기 전 개념

설치 안하고 쓰는 것 보다 직적 설치하고 사용하는 것을 추천한다.

- 쿠버네티스를 쓸려면 무조건 Container들 사이를 연결 하는 **Container Network Interface**가 있어야 한다.
  - 다양한 종류의 플러그인이 존재한다.
  - 여기에서는 위브넷을 사용할 것이다. 종류 다양하니 뭐든 쓰면 된다.

- control plane(mater node)
  - 워커 노드들의 상태를 관리 제어
  - single master

    - multi master도 있긴하다.
- worker node
    - 도커 플랫폼을 통해 컨테이너를 동작하며 실제 서비스 제공



당연히

▼ VM에 ubuntu 올리기

1. ubuntu desktop 설치
2. 우분투 연결
3. network 내부에서설정 (Nat Network)
    a. master
    b. node1
    c. node2 이런 식으로
4. ssh server 설치 및 서비스 동장
    a. su -
    b. apt-get update
    c. apt-get install -y openssh-server curl vim tree
5. 양방향 연결 하고, 이미지 내보내기
6. root 비밀번호 설정
7. systemctl default multi-user.target

8. systemctl isolate muti-user.target
9. Xshell을 이용해 연결
10. 참고
    a.

```
- master

- 가상 머신 복제 했을 때 MAC 주소 정책을
모든 네트워크 어댑터의 새 MAC 주소 생성으로 지정

다음 vi /etc/localhost/

// 그래픽 모드로 변환
systemctl isolate grahical.target

// systemctl isolatel multi-user.target

이제 스냅샷을 따서 보관해 두자. 나중에 이 시점으로 돌아가 뭔가 ㄷ
```

▼ 쿠버네티스 설치 1.31 (최신 버전 기준)

- docker대신 contained를 컨테이너 런타임으로 쓴다..
- https://jbground.tistory.com/107 참고
-

```
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.31/deb/
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-k
sudo apt-get update

여기에서 버전을 1.31로 하니까 완료 되었음.
쿠바네티스 gpg 키 같은거 받는 명령어인듯 함.

// 아니 ㅅㅂ 이런게 어딧냐
CPU 부족:

시스템에 사용 가능한 CPU 코어가 1개뿐입니다. 쿠버네티스는 최소 2개
메모리 부족:
```

```
시스템의 RAM이 957MB로, 쿠버네티스에서 요구하는 최소 1700MB보다 작
```

▼ 서버 최적화

기본

1. df -h

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        29G  5.7G   23G  21% /
tmpfs           479M     0  479M   0% /dev/shm
tmpfs           192M  1.9M  190M   1% /run
tmpfs           5.0M     0  5.0M   0% /run/lock
/dev/xvda16     881M   76M  744M  10% /boot
/dev/xvda15     105M  6.1M   99M   6% /boot/efi
tmpfs            96M   16K   96M   1% /run/user/1000
shm              64M     0   64M   0% /run/k3s/containerd/io.containerd.grpc.v1.cri/sandboxes/eeb89a0ff3b28fe053be81c65f1a832577dc364e577da5fd95839824f201089d/shm
shm              64M     0   64M   0% /run/k3s/containerd/io.containerd.grpc.v1.cri/sandboxes/2140ff3f6234044e5a0cb89944131cc32955e740750e9bb602d157e6148e5dc6/shm
shm              64M     0   64M   0% /run/k3s/containerd/io.containerd.grpc.v1.cri/sandboxes/992c5a84a89a9ee140545d1d2be39f4fb044a17ca64cd7cfbfe4e6b745d1d193/shm
shm              64M     0   64M   0% /run/k3s/containerd/io.containerd.grpc.v1.cri/sandboxes/0eb9082da28dfb27414214d6c59e2fd5ce8bd309ee85ed6098ca622a4e23559d/shm
shm              64M     0   64M   0% /run/k3s/containerd/io.containerd.grpc.v1.cri/sandboxes/575571f4573ad2a6881c3e6f0c2adc2a8912a8300884b21239f33d838d1a7806/shm
```

2. htop

메모리등을 전체적으로 확인!

▼ **SWAP MEMORY** 사용 방법

```
### 문제 상황

1. 메모리 문제
    1. ./gradlew build에 시간이 너무 오래 걸리는 문제 발생 → t2.micro의 램이 1GB 밖에 안된다는 점.
    2. SWAP 메모리를 사용하여 해결
        1. RAM이 부족할 경우 HDD의 일정 공간을 RAM 처럼 사용하는 것
        2. 하드디스크를 가상 메모리로 사용 한것.
        3. 특히 리눅스에서 가상메모리 효율이 좋음
```

SWAP 메모리란 무엇이냐하면, RAM이 부족할 경우가 있으므로 HDD의 일정공간을 마치 RAM처럼 사용하는 것이다. 그래서 우리는 반 강제적으로 RAM을 증설한 듯한 효과를 만들수 있다.

```
dd 명령어로 swap 메모리에 할당
sudo dd if=/dev/zero of=/swapfile bs=128M count=16

128씩 16개의 공간을 만드는 것이여서 우리의 경우 count를 16으로 할
스왑 파일에 대한 읽기 및 쓰기 권한을 업데이트
sudo chmod 600 /swapfile

Linux 스왑 영역을 설정
sudo mkswap /swapfile

스왑 공간에 스왑 파일을 추가하여 스왑 파일을 즉시 사용할 수 있도록 
sudo swapon /swapfile

절차가 성공했는지 확인합니다.
sudo swapon -s

/etc/fstab 파일을 편집하여 부팅시 스왑 파일을 활성화
sudo vi /etc/fstab

파일 끝에 다음 줄을 새로 추가하고 파일을 저장한 다음 종료합니다.
/swapfile swap swap defaults 0 0
```

▼ 캐시된게 너무 많을 때

```
// 이 목록을 삭제하면 다음번 apt-get update 시 새로운 패키지 목록
sudo rm /var/lib/apt/lists/* -vf

// 다운로드된 패키지 아카이브 파일들을 삭제하여 디스크 공간을 확보합
// 파일 설치 후 남아 있는 .deb 파일을 삭제하는 것과 동일
sudo apt-get clean

//sudo apt-get update : 패키지 목록을 최신으로 업데이트합니다.
```

```
sudo apt-get updat
캐시된 데이터가 너무 많을 때 해결방법
```

▼ 설치 순서

1. EC2와 Termius의 연결
    a. AWS 혹은 제공측에서 준 주소,key로 Termius 활성화

2. 설치 전 기본 세팅
    - 패키지 업데이트

```
sudo apt-get update
```

    - 패키지 안전관리

```
sudo apt-get install apt-transport-https ca-certificate
```

3. 도커 설치

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg

echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/us
  $(lsb_release -cs) stable" | sudo tee /etc/apt/source

sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd

sudo service docker start

##install docker compose

sudo curl -L https://github.com/docker/compose/releases
sudo chmod +x /usr/local/bin/docker-compose
sudo usermod -aG docker $USER
```

```
docker-compose --version
```

4. Jenknis 설치

```
cd /home/ubuntu && mkdir jenkins-data
```

5. 방화벽 구성
   - 쿠버네티스 쓸 때도 방화벽 안쓰는게 좋다.
     - Pod 간 통신, 서비스 디스커버리, 로드 밸런싱 등의 작업을 자동으로 처리
     - 방화벽 규칙이 존재하면 이러한 기능들이 원할하게 동작 안할 수 도 있음
     - 때문에 쿠버네티스 환경에서는 방화벽 규칙을 최소화 하거나 네트워크 보안 정책을 다른 방법으로 관리하는 것이 더 나은 접근 방식 일 수 있다.

```
sudo ufw allow 8080/tcp
sudo ufw reload
sudo ufw status
```

6. 젠킨스 컨테이너 생성 및 구동

```
sudo docker run -d -p 8080:8080 -v /var/run/docker.sock

##check your init password
sudo docker logs jenkins

## sign-in
http://주소:8080

## more install plugins
- Docker
- Docker pipeline
- Docker API
```

```
- Gitlab
- Generic Webhook Trigger Plugin
- SSH Agent
```

7. java 설치
   - 현재 가상 환경에서는 java가 설치되어있지 않음 따라서 java 설치해줘야함.

```
sudo apt-get install openjdk-17-jdk

export JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64
export PATH=$JAVA_HOME/bin:$PATH

echo 'export JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd
echo 'export PATH=$JAVA_HOME/bin:$PATH' >> ~/.bashrc

source ~/.bashrc
```

8. .jar 파일 생성
   - 권한 부여 후 .jar 파일 실행

```
chmod +x ./gradlew
./gradlew build
```

9. Dockerfile 작성
   - root 디레토리의 하위 디렉토리에 작성함. src 있는 부분
   - .jar 파일을 이미지로 바꿔줌.
   - **기본 이미지 설정**: `openjdk:17-jdk-slim` 이미지를 기반으로 사용합니다. 이는 Java 17 JDK가 포함된 슬림 버전의 이미지를 가져옵니다.
   - **시간대 설정**: `tzdata` 패키지를 설치하고, 서버의 기본 시간대를 "Asia/Seoul"로 설정합니다.
   - **작업 디렉토리 설정**: `/app` 디렉토리를 작업 디렉토리로 설정합니다.

- **JAR 파일 복사**: `build/libs/` 디렉토리의 JAR 파일을 컨테이너의 `app.jar`로 복사합니다.
- **JAR 파일 실행**: 컨테이너가 시작될 때 `java -jar app.jar` 명령을 실행하여 애플리케이션을 실행합니다.

```
# Use a base image with JDK
FROM openjdk:17-jdk-slim

# Install tzdata package and configure timezone
RUN apt-get update && apt-get install -y tzdata && \
    ln -fs /usr/share/zoneinfo/Asia/Seoul /etc/localtim
    dpkg-reconfigure --frontend noninteractive tzdata

# Add a directory for the application
WORKDIR /app

# Copy the JAR file into the container
COPY build/libs/*.jar app.jar

# Run the JAR file
ENTRYPOINT ["java", "-jar", "app.jar"]
```

10. 도커 허브에 도커 이미지 올리기
    - 도커 허브 username 확인 및 로그인

```
docker info | grep Username
docker login
```

    - 이미지 태그 지정 및 이미지 푸시
    - **tagename?**

```
//이미지 생성 (e6932는 올라간 이미지?)
docker tag e69320dc3458 jaechanjj/test:tagname

docker push jaechanjj/test:tagname
```

-

10. 쿠버네티스에 image 올리기
    - k3s 설정
    - deployment.yaml 파일 작성

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-app
  template:
    metadata:
      labels:
        app: spring-app
    spec:
      containers:
      - name: spring-app
        image: jaechanjj/test:tagname
        ports:
        - containerPort: 8081
```

    - service.yaml 파일 작성

```yaml
apiVersion: v1
kind: Service
metadata:
  name: spring-app-service
spec:
  selector:
    app: spring-app
  ports:
  - protocol: TCP
    port: 80
```

```
    targetPort: 8081
  type: LoadBalancer
```

▼ 상태확인

kubectl

- 클러스터 상태 확인
  - 클러스터의 상태 확인, 마스터 및 주요 서비스 URL

```
kubectl cluster-info
```

- 노드 상태 확인
  - 현재 클러스터에서 사용중인 모든 노드의 상태 확인

```
kubectl get nodes
```

- Pod 상태 확인

```
kubectl get pods

kubectl get pods -n <namespace>
```

- 서버 용량 확인

```
df -h
htop
```

▼ Nginx

HISTORY OF WEB

웹의 역사

1. **HTML**이 처음 개발되어 정적(static)인 페이지를 제공하는 방식으로 시작.

2. **JavaScript**와 **Single Page Application(SPA)**의 등장으로 동적인 데이터 처리 가능.
3. 웹 애플리케이션의 복잡도가 증가하면서, 정적 페이지와 동적 페이지를 각각 처리하는 **Web Server(WS)**와 **Web Application Server(WAS)**로 역할이 분리됨.
    a. **WS (Web Server)**: 클라이언트의 정적 요청(HTML, CSS, 이미지 파일 등)을 처리.
    b. **WAS (Web Application Server)**: DB 조회, 비즈니스 로직 등을 처리.
4. **NCSA HTTPd**의 등장: 초기 웹 서버의 시초.
5. **Apache**의 등장: NCSA HTTPd를 기반으로 하여 발전된 오픈소스 웹 서버.
6. 대규모 클라이언트와의 동시 연결 처리 문제, 즉 **C10K 문제**(동시 1만 클라이언트 처리 문제)가 대두.
7. 대용량 트래픽을 처리하는 데 있어 **Apache**의 구조적 한계 발생.
    - Apache는 **멀티 프로세스/멀티 스레드** 모델을 사용하여 요청당 별도의 프로세스를 생성하기 때문에, 많은 수의 연결을 처리하는 데에 비효율적일 수 있음.
8. **Nginx**의 탄생: 이벤트 기반 처리 방식으로 C10K 문제를 해결하고 고성능을 지향하는 웹 서버로 등장.

Why Nginx?!

- **Apache**는 요청을 처리할 때 각 요청마다 프로세스를 할당하며, 요청이 없는 상태에서도 자원을 낭비하는 경우가 있음. 이에 비해 **Nginx**는 자원을 더 효율적으로 사용.
- Nginx는 **이벤트 기반(event-driven)** 구조를 사용하여, 요청이 들어오지 않으면 새로운 프로세스를 생성하지 않으며, 기존 커넥션에 요청이 들어오면 이를 효율적으로 처리.
- **오래 걸리는 요청**이 있을 경우, Nginx는 **쓰레드 풀(thread pool)**을 이용하여 해당 요청을 별도로 처리하고, **worker process**는 다른 요청을 처리하여 자원을 효율적으로 관리.

    - **Worker process**는 CPU 코어 수에 맞게 생성되며, 이를 통해 CPU의 **context-switching** 비용을 줄임.
    - **Context-switching**은 프로세스나 스레드 간 전환 과정에서 CPU가 소모하는 부가 작업인데, Nginx는 이를 줄여 성능을 극대화함.
- **요약**: Nginx는 Apache에 비해 **CPU의 context-switching**을 줄이고, **프로세스와 컴퓨터 자원**을 훨씬 효율적으로 사용하는 서버.

▼ Jenkins

▼ BE-pipeline

```
pipeline {
    agent any

    environment {
        DOCKERHUB_CREDENTIALS_ID = 'dockerhub-credentia
        GIT_CREDENTIALS_ID = 'gitlab-credentials-id' //
        IMAGE_NAME = "jaechanjj/flickeruser"
        VERSION = "${env.BUILD_NUMBER}" // Jenkins 빌드
    }

    stages {
        stage('gitlab_clone') {
            steps {
                script {
                    withCredentials([usernamePassword(c
                        // 리포지토리가 이미 존재하는 경우 변경
                        sh '''
                        if [ -d ".git" ]; then
                            git fetch origin BE/release
                            git checkout BE/release/use
                            git pull origin BE/release/
                        else
                            git clone -b BE/release/use
                            git checkout BE/release/use
                        fi
                        '''
                    }
                }
```

```
        }
    }

    stage('Print Branch') {
        steps {
            script {
                def currentBranch = sh(script: 'git
                echo "Current branch: ${currentBran
            }
        }
    }

    stage('delete application.properties') {
        steps {
            echo 'Building BE/release branch..'
            // Gradle 빌드 명령 실행
            sh '''
                cd backend
                cd user
                rm -f src/main/resources/applicatio
            '''
        }
    }

     stage('Copy application-be-user.properties and
        steps {
            // Jenkins에 저장된 application-be-user.p
            withCredentials([file(credentialsId: 'a
                script {
                    // application-be-user.properti
                    sh '''
                    cd backend
                    cd user

                    cp $APP_PROPERTIES src/main/res
                    chmod +x ./gradlew
                    ./gradlew build -x test
                    '''
```

```
                }
            }
        }
    }


    stage('Install Docker') {
        steps {
            script {
                // Docker가 설치되어 있는지 확인
                sh '''
                if ! command -v docker > /dev/null;
                    curl -fsSL https://get.dock
                    sh get-docker.sh
                else
                    echo "Docker is already install
                fi
                '''
            }
        }
    }


    stage('Build Docker Image') {
        steps {
            echo 'Building Docker image..'
            sh "cd backend && cd user && docker bui
        }
    }



    stage('Push Docker Image') {
        steps {
            echo 'Pushing Docker image to DockerHub
            // DockerHub에 로그인 후 이미지를 푸시
            withCredentials([usernamePassword(crede
                sh '''
```

```
                    echo ${DOCKERHUB_PASSWORD} | docker
                    docker push ${IMAGE_NAME}:${VERSION
                    '''
                }
            }
        }

        stage('Update and Apply Deployment') {
            steps {
                 script {
            withCredentials([usernamePassword(credentia
                sh '''
                # GitLab 리포지토리 클론
                git clone -b BE/release/user https://${
                cd temp-repo


                # deployment.yaml 파일의 이미지 버전 업데이트
                sed -i "s/jaechanjj\\/flickeruser:[0-9]

                cat deployment.yaml


                git config user.name "jenkins"
                git config user.email "jenkins@example.

                git add deployment.yaml
                git commit -m "Update image version to
                git push origin BE/release/user
                '''
            }
        }
    }
}

    }

    post {
```

```
        always {
            cleanWs() // 빌드 완료 후 작업 공간 정리
        }
    }
}
```

▼ FE-pipeline

```
pipeline {
    agent any

    environment {
        DOCKERHUB_CREDENTIALS_ID = 'dockerhub-credentia
        GIT_CREDENTIALS_ID = 'gitlab-credentials-id'
        IMAGE_NAME = "jaechanjj/flickerfront"
        VERSION = "${env.BUILD_NUMBER}"
    }

    stages {
        stage('Clone GitLab Repo') {
            steps {
                script {
                    withCredentials([usernamePassword(c
                        // 프론트엔드 리포지토리 클론
                        sh '''
                        if [ -d ".git" ]; then
                            git fetch origin FE/release
                            git checkout FE/release
                            git pull origin FE/release
                        else
                            git clone -b FE/release htt
                            cd frontend
                        fi
                        '''
                    }
                }
            }
        }
```

```
        stage('Install Docker') {
            steps {
                script {
                    // Docker가 설치되어 있는지 확인
                    sh '''
                    if ! command -v docker > /dev/null;
                            curl -fsSL https://get.dock
                            sh get-docker.sh
                    else
                        echo "Docker is already install
                    fi
                    '''
                }
            }
        }

        stage('Build Docker Image') {
            steps {
                echo 'Building Docker image for fronten
                sh '''
                cd frontend/frontend
                ls
                docker build -t ${IMAGE_NAME}:${VERSION
                '''
            }
        }

        stage('Push Docker Image') {
            steps {
                echo 'Pushing Docker image to DockerHub
                withCredentials([usernamePassword(crede
                    sh '''
                    echo ${DOCKERHUB_PASSWORD} | docker
                    docker push ${IMAGE_NAME}:${VERSION
                    docker logout
                    '''
                }
```

```
            }
        }

        stage('Update and Apply Deployment') {
            steps {
                script {
                    withCredentials([usernamePassword(c
                        sh '''
                        git clone -b FE/release https:/
                        cd temp-repo

                        # deployment.yaml 파일의 이미지 버
                        sed -i "s/jaechanjj\\/flickerfr

                        git config user.name "jenkins"
                        git config user.email "jenkins@

                        git add deployment.yaml
                        git commit -m "Update flickerfr
                        git push origin FE/release
                        '''
                    }
                }
            }
        }
    }

    post {
        always {
            cleanWs() // 빌드 완료 후 작업 공간 정리
        }
    }
}
```

▼ Argo

ArgoCD가 `kubectl apply` 명령어들을 대신 실행한다고 생각하면 됩니다.

이를 통해 MySQL 관련 리소스들이 자동으로 배포됩니다.

gitOps 기반의 CD 툴

▼ MySQL 연결

▼ 주의

```
만약 be-user와 be-movie라는 다른 네임스페이스에서 각각 10Gi의 스

두 가지 중요한 시나리오:
같은 호스트 경로 사용: 두 네임스페이스에서 같은 경로(/mnt/data/my

다른 경로 사용 권장: 각 네임스페이스에서 다른 물리적 경로를 사용하도

결론:
스토리지 크기는 네임스페이스 간의 설정에 상관없이 각각의 PVC마다 독
하지만 물리적 경로(/mnt/data/mysql)를 동일하게 사용하면 충돌 위
```

```
네, **hostPath**는 물리적인 스토리지 경로를 지정하는 항목이므로,

왜 hostPath와 storageClassName을 다르게 다루는지:
hostPath:
hostPath는 클러스터 노드의 로컬 디렉터리 경로를 의미합니다. 같은 경
storageClassName:
storageClassName은 PVC가 요청하는 스토리지를 어떻게 제공할지를 정
즉, 여러 PVC가 같은 스토리지 클래스를 사용하더라도, 다른 hostPath
결론:
**hostPath**는 네임스페이스 간 서로 다른 경로를 설정해야 데이터 충
**storageClassName**은 여러 네임스페이스에서 공유해도 문제가 없으
따라서 storageClassName을 동일하게 설정하고, hostPath만 각 네임
```

PV의 hostPath의 path에 대해서는 mkdir 후 chmod로 권한 줘야 함.

```
sudo mkdir -p /mnt/data/mysql-movie
```

```
sudo chmod 777 /mnt/data/mysql-movie
```

▼ 데이터 수집을 쉽게 하기 위해서 열어둬야함.

1. 쿠버네티스는 MySQL 설정 직접 적으로 변경하지 않음.

   Kubernetes는 "불변 인프라"를 채택합니다. 즉, 변경이 필요한 경우 기존 리소스를 수정하는 대신 새로운 리소스를 생성하여 교체하는 방식입니다. 이러한 방식은 안정성과 일관성을 높이는데 기여합니다. 따라서 직접적으로 Pod 내부의 설정 파일을 수정할 수 없습니다.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-config
  namespace: be-user
data:
  my.cnf: |
    [mysqld]
    bind-address = 0.0.0.0
    skip-host-cache
    skip-name-resolve
    datadir=/var/lib/mysql
    socket=/var/run/mysqld/mysqld.sock
    secure-file-priv=/var/lib/mysql-files
    user=mysql
```

이거를 같은 dir에 두고 실행 시킬 때 같이 ㄱㄱ → bind-address = 0.0.0.0 이부분을 넣어줘야함.

그래야 로컬 커뮤터에서 접근 가능!

kubectl exec -it mysql-7fc8c4ff5f-hfhhj -n be-user -- cat /etc/my.cnf

▼ Redis

```
#spring 설정 -> service 에서 설정해줘야 함.
```

```
spring.redis.host=redis-service  # Redis 서비스 이름
spring.redis.port=6379  # Redis 기본 포트
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-deployment
  namespace: test  # 사용할 네임스페이스 지정
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:latest  # 최신 Redis 이미지 사용
        ports:
        - containerPort: 6379  # Redis 기본 포트
        # Redis는 휘발성으로 사용하므로 Persistent Volume 생략
---

apiVersion: v1
kind: Service
metadata:
  name: redis-service
  namespace: test
spec:
  ports:
  - port: 6379  # Redis 기본 포트
    targetPort: 6379  # 컨테이너 내부 포트와 매칭
  selector:
```

```
    app: redis
  type: ClusterIP  # 클러스터 내부에서 접근 가능

~
```

▼ How2Test

```
별도의 Debug Pod 생성: Debugging을 위해 임시로 도구가 포함된 Pod를

bash
코드 복사
kubectl run -i --tty --rm debug --image=busybox -- sh
nslookup kafka-service.kafka.svc.cluster.local
```

▼ 메모리 확인 및 최적화

kubectl top pod movie-deployment-65cff7f977-hxtrh -n be-movie

▼ CPU vs 메모리

CPU는 사용자의 트래픽이나 요청에 따라 변동 가능하지만, 메모리는 일반적으로 앱이 처음 실행될 때 할당된 리소스가 그대로 유지됨.

트래픽이 증가하면서~
but 1. 세션 정보를 많이 담게 되면 메모리 증가할 수 있음

2. 캐시 활성화 되면 메모리 증가할 수 있음
3. 대량의 데이터를 처리하거나 임시 데이터를 저장하는 경우 사용량이 늘어 날 수 있음~!

fact → 일단 많이 몰릴 것 같은 부분은 CPU,메모리 값 크게 둬야함!

▼ 포트 충돌?

```
K3s와 Kubernetes에서 네임스페이스는 리소스를 분리하는 방법 중 하나

네임스페이스의 역할
리소스 격리:

네임스페이스는 Kubernetes 클러스터 내에서 리소스를 격리하여 관리할
```

```
포트 충돌 방지:

K3s 및 Kubernetes에서는 각 네임스페이스 내에서 서비스가 같은 포트
예시
namespace1에 my-service가 포트 80을 사용하고,
namespace2에 my-service가 포트 80을 사용하더라도,
내부적으로는 문제가 없지만, 외부적으로 노출하는 방식에 따라 충돌이 발
외부 접근을 고려할 경우
NodePort:

서로 다른 네임스페이스에 있는 서비스가 동일한 NodePort를 사용하면
LoadBalancer:

LoadBalancer 서비스 유형에서도 같은 IP와 포트를 사용하는 경우 문제
Ingress:

Ingress를 사용하면 여러 서비스의 요청을 같은 IP와 포트를 통해 분산
결론적으로, 네임스페이스가 다르더라도 동일한 포트를 사용하는 것은 가능
```

▼ SSL

SSL을 적용해야 하는 주요 이유:

1. **데이터 암호화**:
   SSL을 사용하면 클라이언트(브라우저)와 서버 간의 데이터가 암호화되어 전송됩니다. 이를 통해 전송 중에 발생할 수 있는 **도청, 중간자 공격(MITM)**과 같은 보안 위협으로부터 데이터를 보호할 수 있습니다. HTTP는 암호화되지 않은 평문으로 데이터를 전송하기 때문에, 중요한 정보(예: 사용자 로그인 정보, 금융 정보 등)가 쉽게 노출될 수 있습니다.
2. **데이터 무결성 보장**:
   SSL은 데이터의
   무결성을 보장합니다. 즉, 클라이언트와 서버 간에 전달되는 데이터가 중간에서 변

경되지 않았음을 확인할 수 있습니다. 이를 통해 데이터 위변조의 가능성을 줄이고, 전송 중에 데이터가 손상되거나 변조되는 위험을 줄입니다.

3. **인증서 기반 서버 신뢰성 보장**:
   SSL 인증서는 클라이언트가 연결하려는 서버가 신뢰할 수 있는 서버임을 보장합니다. **인증 기관(CA)**에서 발급한 SSL 인증서를 통해 사용자는 자신이 실제로 신뢰할 수 있는 사이트에 접속하고 있음을 알 수 있습니다. 이로 인해
   피싱 사이트나 **가짜 사이트**와의 연결을 방지할 수 있습니다.

4. **SEO 및 사용자 신뢰성 향상**:
   많은 웹 브라우저(특히 Chrome, Firefox 등)에서 SSL이 적용되지 않은 사이트(HTTP)를 "안전하지 않음"으로 표시합니다. 이는 사용자에게 부정적인 인식을 줄 수 있으며, 실제로 사이트 신뢰도에도 영향을 미칠 수 있습니다. 또한, Google과 같은 검색 엔진은
   HTTPS를 사용하는 사이트에 대해 **SEO 점수**를 더 높게 부여합니다.

5. **게이트웨이 보안 강화**:
   현재 Gateway에서 외부 요청을 받도록 설정되어 있습니다. 이 Gateway는 외부와 내부 네트워크를 연결하는 주요
   접점이기 때문에, 여기서 들어오는 트래픽을 보호하는 것이 매우 중요합니다.
   Gateway에 SSL을 적용함으로써 외부에서 내부 시스템으로 들어오는 모든 요청이 안전하게 암호화되어 전달되고, 보안 사고의 가능성을 줄일 수 있습니다.

6. **규정 및 컴플라이언스 준수**:
   금융, 의료 등 특정 산업에서는 데이터를 암호화해야 한다는 규정이 존재합니다. SSL을 통해 이러한
   법적 규제와 **컴플라이언스**를 준수할 수 있습니다. 특히, 개인 정보를 다루는 웹 애플리케이션은 SSL을 필수적으로 적용해야 합니다.

▼ HTTPS

- Cert-Manager , ClusterIssuer,Certificates
- Cert-Manager은 ClusterIsuuer 형식으로 선언함.
- 각 namespace 에 맞게 인증서를 발급함.
- FE와 gateway의 도메인을 일치시켜 인증서의 무결성을 높힘