CLOUD SQUAD

클라우드 3주차

2025.04.30

BY 김대현

Contents

외부 공유 금지!!!
카카오 내부 자료도 있기 때문에....

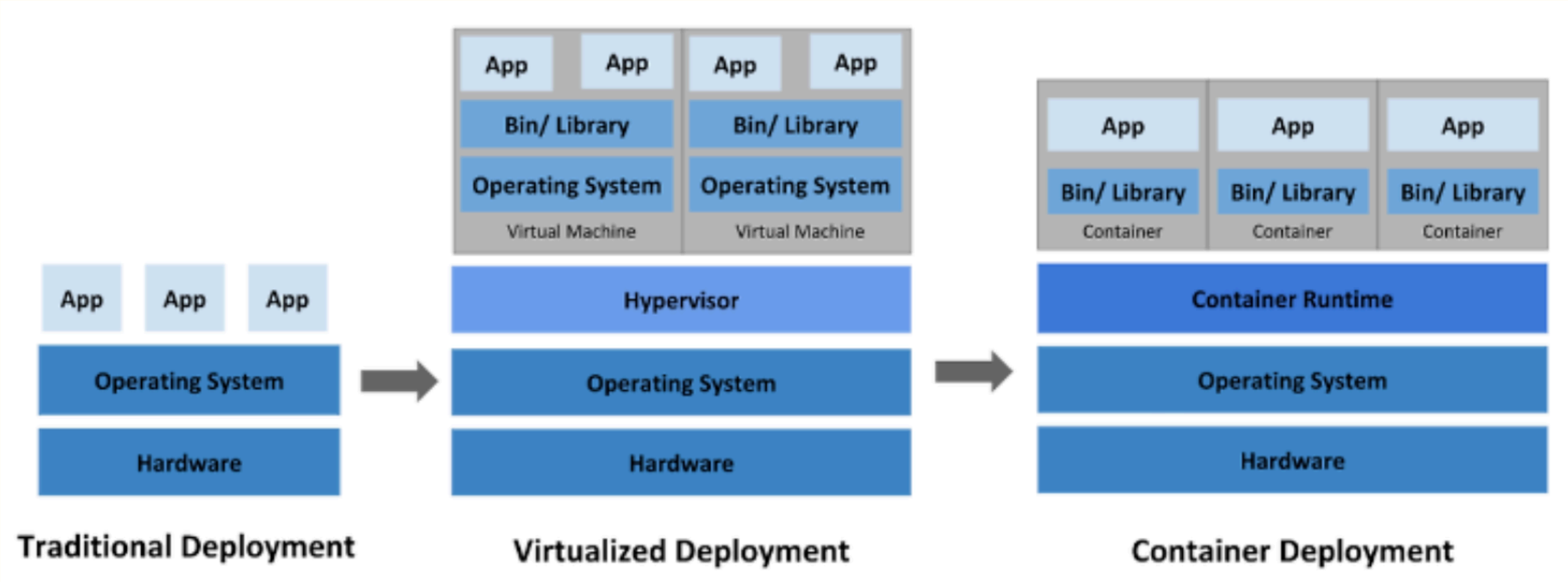
Container (컨테이너)

01 Container

Container

- **Container란?** : 소프트웨어를 실행하기 위한 경량화된 가상화 환경이자 패키징 방식

- 운영체제 수준에서 CPU, 메모리, 스토리지, 네트워크 등 리소스를 쉽게 공유할 수 있음
  - 운영체제에서 프로세스를 격리하여 별도의 실행 환경을 제공



배포 방식

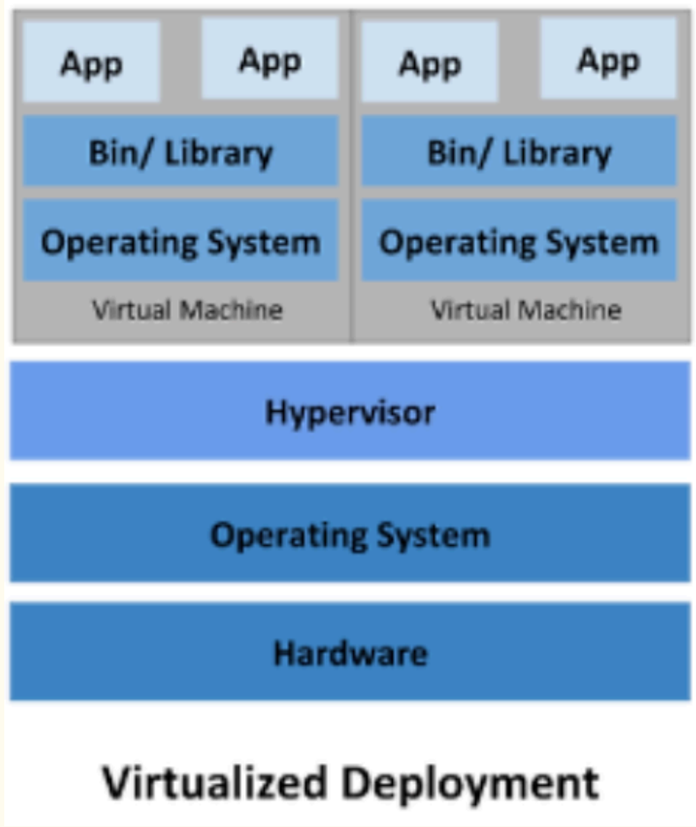
1. 전통적 배포 (Traditional Deployment)

- 물리 서버에 직접 OS를 설치하고, 그 위에 애플리케이션을 실행
- 서버 하나당 하나의 운영체제, 여러 앱이 한 OS 위에서 동작
- 앱 간 프로세스·라이브러리 충돌 위험

배포 방식

2. 가상화 배포 (Virtualized Deployment)

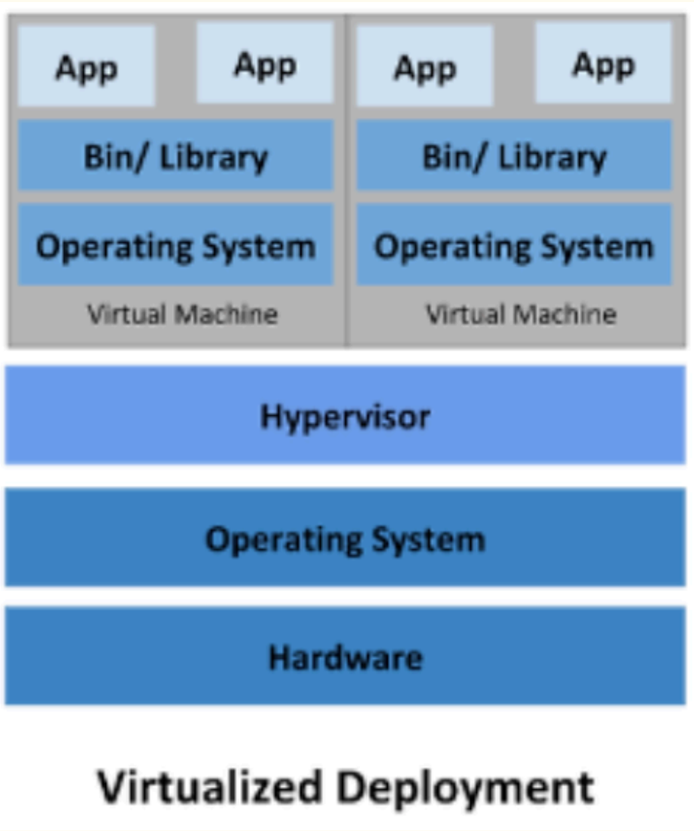
- 하이퍼바이저 위에 여러 개의 가상 머신(VM) 을 띄움
- 각 VM은 독립적인 게스트 OS를 필요로 하므로 무겁고, 부팅 시간이 김
- 물리 자원을 VM 단위로 강력하게 격리할 수 있음



배포 방식

3. 컨테이너 배포 (Container Deployment)

- 호스트 OS의 커널 공유
  - VM처럼 별도의 커널이 아니라, 호스트 OS 커널을 네임스페이스(namespace)와 cgroup 으로 격리해서 사용
- 경량화된 격리
  - 각 컨테이너에는 필요한 바이너리/라이브러리와 애플리케이션만 포함
  - 불필요한 OS 구성 요소가 없어 VM에 비해 훨씬 작고 빠름
- 이미지 기반 배포
  - 컨테이너 이미지는 계층화된 파일시스템으로 구성
  - → 변경된 레이어만 캐시하여 빌드 속도 ↑, 저장공간 ↓
- 이식성(portability)
  - “개발 환경 ↔ 테스트 환경 ↔ 프로덕션” 어디서나 동일하게 실행
- 신속한 스케일 아웃
  - 컨테이너 시작(수초 이내) → 수천 개 단위의 마이크로서비스도 빠르게 확장 가능



Container vs Virtual Machine (VM)

그러면 컨테이너는 VM과 어떠한 차이가 있을까?

특성	Virtual Machine	Container
가상화 방식	하드웨어 가상화 (Hypervisor 사용)	운영체제 가상화 (커널 공유)
성능	오버헤드가 높아 상대적으로 성능 저하	경량화된 구조로 높은 성능 제공
시작 시간	수 분에서 수 초	몇 초 이내
리소스 효율성	비효율적 (전체 OS 인스턴스 필요)	효율적 (애플리케이션만 포함)
이미지 크기	크고 복잡함	작고 간단함
격리 수준	높은 수준의 격리 (완전한 OS 격리)	낮은 수준의 격리 (프로세스 수준 격리)
사용 사례	전통적인 데이터센터 및 서버 가상화	마이크로서비스, CI/CD 파이프라인
호환성	다양한 OS 실행 가능	동일한 커널을 공유하는 OS만 실행 가능
관리 도구	vSphere, vCenter	Docker, Kubernetes
배포 복잡성	복잡함 (하드웨어 및 소프트웨어 설정 필요)	간단함 (이미지로부터 바로 실행 가능)
업데이트 및 유지보수	복잡함 (전체 VM을 업데이트해야 함)	간단함 (컨테이너 이미지 업데이트)

01 Container

Container (추가 개념)

- **Container란?** : 앱을 실행하기 위해 격리된 경량 프로세스

- 프로세스 - 실행중인 프로그램 (메모장, 크롬브라우저 등..)
  - 메모리, 파일 시스템, 네트워크 등 컴퓨터의 다양한 자원 사용
  - 서로 간섭하지 않도록 메모리 수준에서 격리
- 컨터이너는 기본 프로세스보다 더 많은 자원을 격리
  - (파일시스템, 네트워크도 같이)
- **컨테이너가 필요한 이유?** → 하나의 컴퓨터(서버)에서도 격리된 환경 필요, 앱간 간섭 줄이기 & 특정 앱간 장애가 서로 영향을 주지 않기 위함.

01 Container

Container (추가 개념)

- **요구사항에 따라 격리 수준을 다르게 할수도 있음** : host(컴퓨터)가 어떤 자원 층에서 격리 하냐에 따라 달라짐
  - → 메모리, 파일시스템, 네트워크, 라이브러리, OS, H/W 등등…

- **ex) 격리 기술**
  - **host os 수준**: 하이퍼바이저 → 단위는 가상머신 (vm, 높은 수준의 격리 제공, 보안상 이점. 단, os를 구성하기 위한 모든 실행요소가 필요. 무거워짐, 초기 구동속도 느림)
  - **app 수준**: Container → 단위: 컨테이너 (운영체제 관련 내용 없음, vm보다 가벼움 → 경량 process, 초기 구동속도 빠름 & 표준 기술로 자리잡음)
- **container는 process이기 때문에 정보를 전달할수 있는 수단이 있어야 함 →** **이게 바로 image**

01 Container

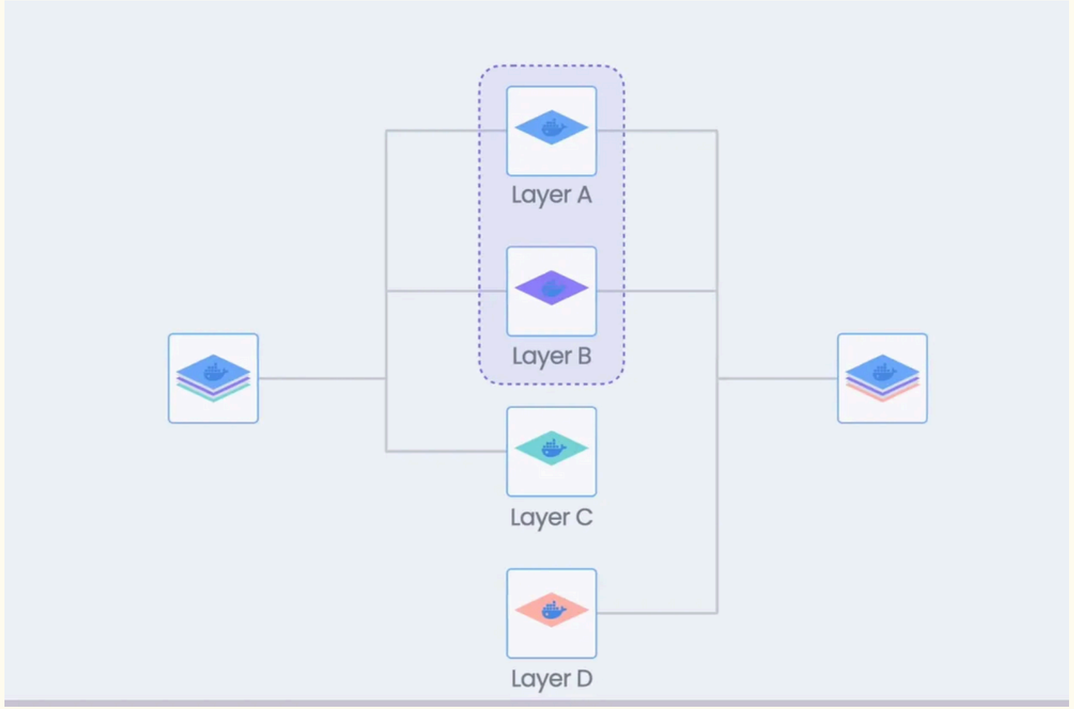
Image?

- **Image는 컨테이너를 생성하기 위해 필요한 모든 것을 포함한 파일 묶음**
  - 서버 실행 소스코드, 명령어, 옵션 정보등을 포함
- image는 파일이기 때문에 네트워크를 통해서 쉽게 주고 받을수 있음 → 단, 정확히 어떤 이미지 사용했는지 알아야함
- 또한 이미지는 버전관리 시스템을 가짐: Digest (다이제스트)

1. **Digest (다이제스트) → 자동으로 생성되는 해시값. 그래서 별칭을 정함 (태그)**
   - 태그는 별칭이기 때문에 시점에 따라 다른 Digest 참고 → 보통 최신 이미지 태그 (latest, 최신 다이제스트)
2. **저장 효율 → 이미지 크기가 커지면 네트워크를 통해 주고 받을때 시간이 오래 걸릴수도.**
   a. 그래서 Layer 방식으로 저장 → 이미지를 Layer 단위의 파일로 나눠서 저장
   b. 그렇게 하면 같은 내용을 중복 저장 방지, 저장 효율을 높이기 위해 사용

01 Container

이미지를 Layer 단위의 파일로 나눠서 저장
같은 내용을 중복 저장 방지, 저장 효율을 높이기 위해 사용



Dockerfile

```
Dockerfile > ...
Daehyun Kim, 4 months ago | 2 authors (You and one other)
# Stage 1: Build dependencies      Daehyun Kim, 4 months ago • Update dockerfile …
FROM python:3.12-slim AS builder

WORKDIR /app

# 시스템 종속성 설치
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    libffi-dev \
    libssl-dev \
    ffmpeg \
    libsndfile1 \
    libgl1-mesa-glx \
    libglib2.0-0 \
    libjpeg-dev \
    zlib1g-dev \
    libcairo2 \
    libpango-1.0-0 \
    libpangocairo-1.0-0 \
    && rm -rf /var/lib/apt/lists/*

# Python 종속성 설치
COPY requirements.txt .
RUN pip install --upgrade pip
RUN pip install --no-cache-dir -r requirements.txt

# Stage 2: Runtime stage
FROM python:3.12-slim

WORKDIR /app

# OpenCV 런타임 의존성 및 ffmpeg 설치
RUN apt-get update && apt-get install -y --no-install-recommends \
    libgl1-mesa-glx \
    libglib2.0-0 \
    ffmpeg \
    && rm -rf /var/lib/apt/lists/*

# 빌드된 Python 패키지 복사
COPY --from=builder /usr/local /usr/local

# /app/logs 디렉토리 생성
RUN mkdir -p /app/logs

COPY . .

ENV PYTHONPATH=/app

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000", "--reload", "--log-config", "logging_config.json"]
```

이 Dockerfile에 적혀있는것이 서버 실행 소스코드 명령어, 옵션. 패키지 명령어 들을 포함

(ex: Python AI BE Dockerfile)

01 Container

Container 기능?

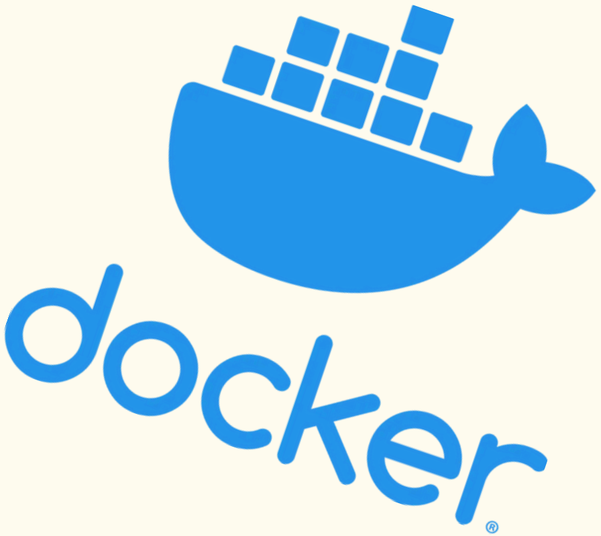
Container는 어떻게 하나의 운영체제위에 여러가지 애프리케이션을 배포할 수 있을까?

- ControlGroup
- Namespace
- Network
- 빌드, 실행, 네트워크 관리 등 다양한 기능 존재

Container 장점?

- 가벼움
- 유지관리
- 다양한 환경지원
- 이식 가능성

Docker
Part.1
(Dockerfile & Image)

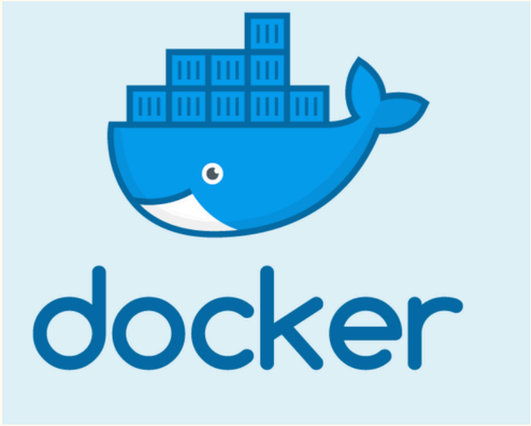


02 Docker Part.1

Docker

- **Docker란?** : **“Container”**라는 가벼운 **가상화 기술**을 **쉽게 다룰 수 있게 해 주는 플랫폼이자 툴링 세트**입니다.

- **환경에서의 이점**
  - 다양한 Image 지원
  - contaienr의 복잡성을 간단하게 사용할 수 있게 지원
- **자원 관리**
  - VM 대비 확장 및 축소의 편리함
  - 프로세스 및 container 별 리소스 제한
- **독립된 환경에서 작업**
  - 격리된 공간에서 애플리케이션 운영
  - 논리적으로 설정된 프로세스 격리



02 Docker Part.1

Docker Desktop 설치

Docker for window

- https:/ /docs.docker.com/desktop/install/windows-install/

Docker for MacOS

- https:/ /docs.docker.com/desktop/install/mac-install/

Ubuntu, Linux는 알아서... 여기 홈페이지에 들어가서 설치해주십쇼

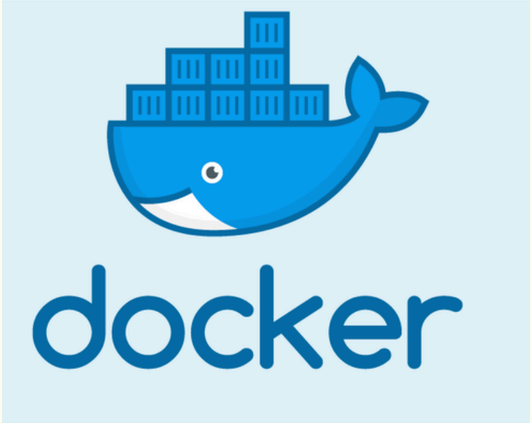


Image (다시 한번)

- **Image는 컨테이너를 생성하기 위해 필요한 모든 것을 포함한 파일 묶음**
  - 서버 실행 소스코드, 명령어, 옵션 정보등을 포함
- image는 파일이기 때문에 네트워크를 통해서 쉽게 주고 받을수 있음 → 단, 정확히 어떤 이미지 사용했는지 알아야함
- 또한 이미지는 버전관리 시스템을 가짐: Digest (다이제스트)

1. **Digest (다이제스트) → 자동으로 생성되는 해시값. 그래서 별칭을 정함 (태그)**
   - 태그는 별칭이기 때문에 시점에 따라 다른 Digest 참고 → 보통 최신 이미지 태그 (latest, 최신 다이제스트)
2. **저장 효율 → 이미지 크기가 커지면 네트워크를 통해 주고 받을때 시간이 오래 걸릴수도.**
   a. 그래서 Layer 방식으로 저장 → 이미지를 Layer 단위의 파일로 나눠서 저장
   b. 그렇게 하면 같은 내용을 중복 저장 방지, 저장 효율을 높이기 위해 사용

02 Docker Part.1

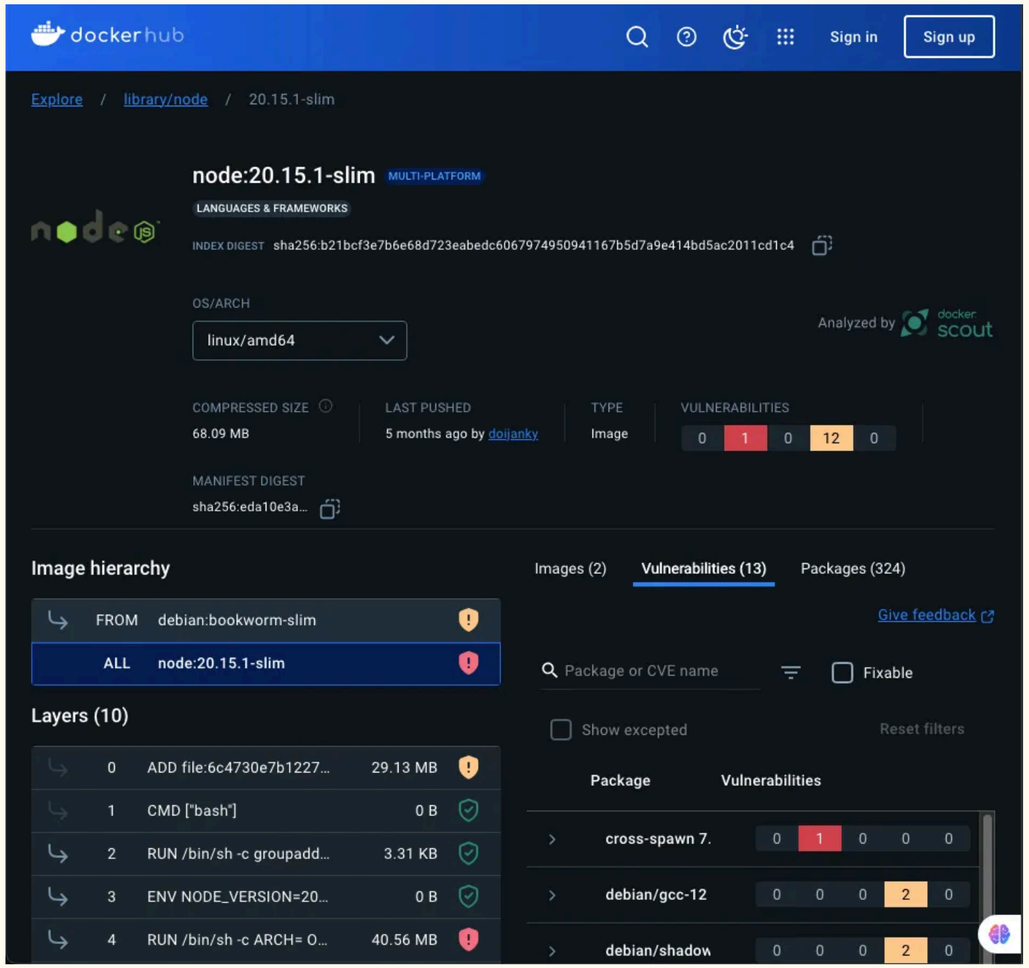
Docker Image 관리

- **image pull → 이미지 다운로드 (이미지 지칭시, 이미지의 repo(이름) & 버전(태그, 다이제스트) 지칭)**
  - 아무것도 지정 안하면 Latest Tag가 지정됨.

Docker Image download site

- https://hub.docker.com/

```
docker image pull node:20.15.1
```



Docker Image 정보 확인

- 태그가 없는건 → 다이제스트로 다운로드 하면 태그 없음
  - (태그 없는 이미지 → 댕글링 이미지)

```
docker image ls

REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
node         20.15.1   4efd4a4c24f5   6 months ago    1.09GB
```

- Docker Image 삭제 명령어

```
docker image rm node:20.15.1
```

- Docker hub에서 Docker image 가져오기 (다이제스트로 다운로드)

```
docker image pull
node@sha256:b21bcf3e7b6e68d723eabedc6067974950941167b5d7a9e414bd5ac2011cd1c4
```

Docker Image 정보 확인

- Docker Image 삭제중...

```
docker image rm node:20.15.1
Untagged: node:20.15.1
Untagged:
node@sha256:6326b52a508f0d99ffdbfaa29a69380321b215153db6f32974835bac71b38fa4
Deleted: sha256:4efd4a4c24f50c8425922423a2c347aea8e44a59000a201989e9c3c45b6e60eb
Deleted: sha256:c24122d83eb16333fa4ee75c16a49db43ff4622b0c9f4a57b7ad6e5adf1506bc
Deleted: sha256:6bb03172210fec21806f33712d2a9e4b05d2db1a699a7f82dfa94c0979b1fc3e
Deleted: sha256:f9f4404d6d88901c14b1721ec05f3b5573c193c849aa1e4ccb81e999d57c3175
Deleted: sha256:b28a326219ecc16a321ba0cbdb8d33d60a3d46d0a3e12fcba94e1e8de82f30f0
Deleted: sha256:ed97fe08483c357121401263360da61553520148f9e15750aa23ed222c77a4d3
Deleted: sha256:438e34db262f9fb5c961a57206d446db754479c448c8bfd9b3ff0ac6996c487e
Deleted: sha256:0803515be5a0100745335910154fa69328f0dc530960d90ed9f8412afb58338d
Deleted: sha256:ef4a09650a82f528ec59c5cc42c20195fb2e27f18929c169616a348be7068a6d
```

- 사용되지 않은 모든 이미지 삭제는 -a 태그 붙이기

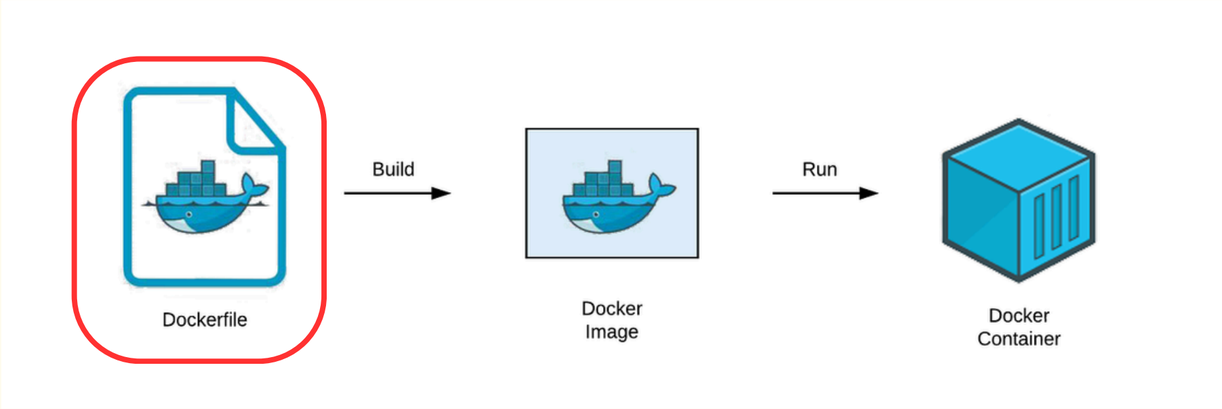
```
docker image prune -a

WARNING! This will remove all images without at least one container associated
to them.
Are you sure you want to continue? [y/N]
```

- 태그가 없는 댕글링 이미지만 삭제

```
docker image prune
```

02 Docker Part.1

Dockerfile



- **이미지를 생성하기 위한 용도로 작성되는 파일**

환경 일관성

• Dockerfile을 사용하면 개발 환경, 테스트 환경, 배포 환경 등 모든 환경에서 동일한 설정 유지

이식성

• Docker 이미지는 다양한 운영체제에서 동일하게 동작

• Dockerfile로 만든 이미지는 어디서든 실행될 수 있음

자동화

• 애플리케이션의 빌드 및 배포 과정을 자동화

• CI/CD 파이프라인 과정에서 빌드와 배포를 자동화하여 사용할 수 있음

반복 가능성

• 항상 동일한 방식으로 이미지를 빌드하기 때문에, 반복 가능한 환경을 제공 → 디버깅과 테스트를 용이하게 관리

확장성

• 애플리케이션을 마이크로 아키텍처로 쉽게 확장

Dockerfile 지시자

지시자	설명	예시
FROM	베이스 이미지를 지정합니다.	FROM ubuntu:20.04
RUN	컨테이너 안에서 명령어를 실행합니다.	RUN apt-get update && apt-get install -y vim
CMD	컨테이너가 시작될 때 실행할 명령어를 지정합니다.	CMD ["node", "app.js"]
LABEL	메타데이터를 추가합니다.	LABEL version="1.0" description="My App"
MAINTAINER	이미지 생성자 정보를 지정합니다.	MAINTAINER John Doe <john.doe@example.com>
EXPOSE	컨테이너가 수신 대기할 포트를 지정합니다.	EXPOSE 8080
ENV	환경 변수를 설정합니다.	ENV NODE_ENV=production
ADD	파일을 복사하거나 URL의 내용을 추가합니다.	ADD myfile.tar.gz /app
COPY	파일이나 디렉토리를 컨테이너로 복사합니다.	COPY . /app
ENTRYPOINT	컨테이너가 시작될 때 실행할 기본 명령어를 설정합니다.	ENTRYPOINT ["python", "app.py"]

Dockerfile 지시자

지시자	설명	예시
VOLUME	데이터 볼륨을 설정합니다.	VOLUME ["/data"]
USER	컨테이너 안에서 사용할 사용자 이름을 설정합니다.	USER appuser
WORKDIR	작업 디렉토리를 설정합니다.	WORKDIR /app
ARG	빌드 시 넘겨줄 변수 값을 정의합니다.	ARG version=1.0
ONBUILD	부모 이미지에서 상속될 명령어를 설정합니다.	ONBUILD RUN echo "Hello, World!"
STOPSIGNAL	컨테이너를 중지할 때 사용할 시스템 호출 신호를 설정합니다.	STOPSIGNAL SIGTERM
HEALTHCHECK	컨테이너의 상태를 확인하는 명령어를 설정합니다.	`HEALTHCHECK CMD curl --fail http://localhost/
SHELL	명령어를 실행할 셸을 설정합니다.	SHELL ["powershell", "-command"]

이렇게 많은데.... 제일 기초적이고 많이 쓰이는 지시자로 주로 설명드릴테니까 걱정하지 마십쇼..!
이거 다 알 필요는 없어여!

Dockerfile 만들기

1. 베이스 이미지 선택 (FROM / ARG)

- 가장 먼저 어떤 운영체제·런타임 환경 위에 앱을 올릴지 결정해야 합니다.
- 모든 후속 명령은 이 베이스 이미지 위에서 실행되기 때문입니다.

```
# (1) 버전을 유연하게 바꿀 수 있도록 ARG로 먼저 선언
ARG NODE_VERSION

# (2) 실제 베이스 이미지 지정
FROM node:${NODE_VERSION}
```

2. 작업 디렉토리 설정 (WORKDIR)

- 컨테이너 안에서 “내가 일할 위치”를 미리 정해 줍니다.
- 장점: 이후 COPY·RUN·ENTRYPOINT 명령어에 일일이 경로를 붙이지 않아도 됩니다..

```
# /app 디렉토리로 이동 (없으면 자동 생성)
WORKDIR /app
```

Dockerfile 만들기

3. 의존성 캐싱을 위한 소스 일부만 복사하기

- package.json, package-lock.json 등 라이브러리 목록만 먼저 복사해서 의존성 설치를 분리하면, 코드만 바뀔 때마다 npm ci 단계가 재실행되지 않아 빌드가 빨라집니다.

```
# 의존성 정의 파일만 복사
COPY package*.json ./

# 의존성 설치
RUN npm ci
```

4. 애플리케이션 소스 전체 복사 (COPY)

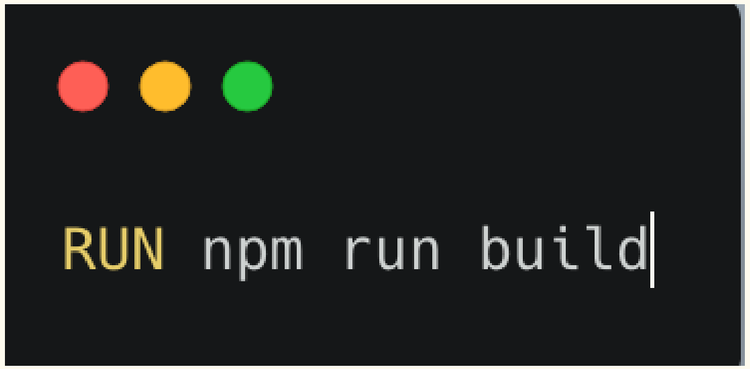
- 의존성 설치가 끝나면, 실제 앱 소스 코드를 모두 복사합니다.

```
# 이제 나머지 소스코드를 한꺼번에 복사
COPY . .
```

Dockerfile 만들기

5. 애플리케이션 빌드 (RUN)

- 필요하다면 빌드 스텝을 추가합니다.
  - (예: TypeScript → JS, 번들링 등)



6. 실행 환경 설정 (ENV / EXPOSE)

- 컨테이너 실행 중에 필요한 환경 변수나 포트를 선언해 줍니다

Dockerfile 만들기

7. 컨테이너 시작 명령어 (ENTRYPOINT 또는 CMD)

- 컨테이너가 시작될 때 어떤 프로세스를 실행할지 정의합니다.
- ENTRYPOINT는 고정된 실행 명령으로
- CMD는 사용자가 docker run <이미지> 다른명령 으로 바꿀 수 있는 기본값으로 사용합니다.

```
ENTRYPOINT ["npm", "run", "start"]
# 또는
# CMD ["npm", "run", "start"]
```

8. .dockerignore 파일 작성 (빌드 효율화)

- 도커 컨텍스트로 불필요한 파일(.git, node_modules 등) 이 복사되지 않도록 .dockerignore를 만들어 두면, 이미지 용량 절감과 빌드 속도 향상에 도움이 됩니다.

```
# .dockerignore 예시
node_modules
.git
Dockerfile
README.md
```

전체 예시 도커파일

```dockerfile
# Node 설치. (이미지 관점 - 맨 아래 Layer (Base image))
# FROM 베이스 이미지
ARG NODE_VERSION
FROM node:${NODE_VERSION}

WORKDIR /app

# 1) 의존성만 먼저 복사 → 설치
COPY package*.json ./
RUN npm ci

# 2) 나머지 소스 복사
COPY . .

# 3) 앱 빌드
RUN npm run build

# 4) 환경 변수·포트
ENV PORT=3000
EXPOSE 3000

# 5) 컨테이너 기동 명령
ENTRYPOINT ["npm", "run", "start"]
```

Summary

- 베이스 이미지(FROM)
- 작업 디렉토리(WORKDIR)
- 의존성 캐싱용 복사 + 설치(COPY + RUN npm ci)
- 전체 소스 복사(COPY . .)
- 빌드(RUN npm run build)
- 환경 변수·포트 설정(ENV, EXPOSE)
- 실행 명령(ENTRYPOINT/CMD)
- .dockerignore로 불필요 파일 제외

03 과제

과제 1. Docker Hub에서 MySQL 이미지 다운로드

- **Docker Hub에서 MySQL image 찾고, 다운로드 후 이미지 목록 확인, 이미지 삭제 완료**한 터미널 **로그 캡쳐해서 제출**

과제 2. Dockerfile 작성

- 다음주에 **Docker Container 빌드 실습을 위한 Dockerfile을 작성해오는것이 과제**. (아래 조건 만족해야함)
- 코드 Repo 주소: https://github.com/hufs-pnp/25-Cloud-mbti (Nest.js 기반 간단한 웹 서비스)
  - 코드 Repo는 참고하셔도 되지만 굳이...?
  - 아래 작성 조건 보고 해도 충분히 하실 난이도 입니다! 쉬워요

Docker File 작성 조건

1. **베이스 이미지: node 20.15.1**
2. **이미지 내 소스 코드 경로: /apps/mbti**
3. **환경 변수**
   - **PORT: 3001**
   - **DB_HOST: localhost**
   - **DB_PORT: 3306**
   - **DB_NAME: db_mbti**
   - **DB_USERNAME: user_mbti**
   - **DB_PASSWORD: pw_mbti**
4. **node_modules 디렉토리는 복사 대상에서 제외하기**

03 과제

제출 방식

- **과제1: Notion Page에 넣어서 링크 전송**
  - **Page Link 제출 하는곳: Cloud Squad Notion - Follow up System (4주차)**
- **과제2: Github Repo에 작성 완료한 Dockerfile push**
  - **[Week 3 / 본인 이름 디렉토리] 안에 넣고 Dockerfile commit 하시면 됩니다)**
  - **Repo 주소: https://github.com/hufs-pnp/25-Cloud-Squad**
- **Due Date: 5/7 (수 19:30 까지)**

다음 미팅

- **Squad Meeting Week 4**
  - 일시: 05.07 (수) 20:00 ~ 21:00
  - 예정시간: 50~60분
  - 내용 Docker Part.2 (이미지 빌드, 컨테이너 실행, Docker 네트워크, Docker Volume)