**설치**

WSL2에서 Docker CLI를 설치하는 방법을 설명한다.

Windows에 설치하려면 아래 기능이 활성화 되어야 한다.

* 하이퍼V, 가상화, 컨테이너, WSL2

**하이퍼V 활성화(파워쉘에서 관리자 모드로 진행):**

| Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All |
| --- |

**가상화 활성화 (필요한 경우):**

바이오스에 들어가서 설정해야 함.

**컨테이너 활성화(파워쉘에서 관리자 모드로 진행):**

| Enable-WindowsOptionalFeature -Online -FeatureName containers -All |
| --- |

**WSL2 설치(파워쉘에서 관리자 모드로 진행):**

| wsl.exe --install wsl --set-default-version 2 |
| --- |

**docker 설치:**

| # Add Docker's official GPG key: sudo apt-get update sudo apt-get install ca-certificates curl sudo install -m 0755 -d /etc/apt/keyrings sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc sudo chmod a+r /etc/apt/keyrings/docker.asc  # Add the repository to Apt sources: echo \  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \  $(. /etc/os-release && echo "${UBUNTU\_CODENAME:-$VERSION\_CODENAME}") stable" | \  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null sudo apt-get update  sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin |
| --- |

**docker 데몬 실행: (윈도우의 서비스 같은 걸 리눅스에서는 데몬이라 부른다.)**

| sudo service docker start |
| --- |

**이미지란?**

모든 설정 명령과 코드가 포함된 공유 가능한 패키지(템플릿)

일단 구워지고 나면, 기본적으로 읽기 모드가 된다.

**이미지 빌드:**

| // Dockerfile 이 있는 경로에서 실행, 꼭 이름이 Dockerfile 이어야 함 docker build -t <name>:<tag> . |
| --- |

**이미지 목록 확인:**

| docker images |
| --- |

**이미지 삭제:**

| // 해당 이미지를 사용하는 컨테이너가 있으면 삭제 불  docker rmi <Name or ImageId> |
| --- |

**컨테이너에서 사용하고 있지 않은 이미지 모두 삭제:**

**\*\*\* 컨테이너가 실행 중이던 정지 중이던 상관 없이 사용중으로**

| // 태그가 없는 dangling 이미지만 삭제  docker image prune  // 태그가 있는 이미지도 삭제  docker image prune -a |
| --- |

**특정 이미지 살펴보기**

| docker image inspect <name> |
| --- |

**기존 이미지를 복사하여, 새로운 tag 이름으로 추가:**

**(중요) 기존 이미지가 삭제되지 않음.**

**기존 이미지를 삭제해도 새로 tag 한 이미지 삭제되지 않음.**

| docker tag <imageName>:<tag> <newImageName>:<newTag> |
| --- |

**컨테이너란?**

컨테이너는 이미지를 기반으로 생성된 실행 가능한 인스턴스.

**컨테이너 실행:**

| // Tag 생략하면, latest로 자동으로 들감  // 로컬에서 이미지 못찾으면 DockerHub에서 찾음  docker run --name <containerName> <imageName>:<imageTag> |
| --- |

**컨테이너 대화형 모드로 실행:**

| // i는 인터렉티브 모드를 뜻하며, 컨테이너 안의 프로그램이 STDIN을 하게 함  // t는 슈도 터미널을 통해 입력을 받겠다는 의미  docker run -it <imageName> |
| --- |

**컨테이너 detached 모드로 실행 (백그라운드 모드):**

| docker run -d <imageName> |
| --- |

**컨테이너 포트포워딩 설정 및 실행**

| // 호스트 8080 포트로 접속하면 컨테이너의 80 포트로 전달됨.  docker run -p 8080:80 <imageName> |
| --- |

**컨테이너 정지:**

| docker stop <name> |
| --- |

**정지된 컨테이너 실행:**

| // run에 입력한 인자를 다시 입력할 필요없다.  docker start <name> |
| --- |

**정지된 컨테이너 attach 모드로 실행:**

| docker start -a <name> |
| --- |

**정지된 컨테이너 대화형 모드로 실행:**

| docker start -i -a <name> |
| --- |

**컨테이너가 정지되면 자동으로 삭제되게 실행:**

| docker start --rm <name> |
| --- |

**컨테이너 삭제:**

| docker rm <name> |
| --- |

**모든 중지된 컨테이너를 한 번에 삭제:**

| docker container prune |
| --- |

**실행 중인 컨테이너에 attach 하기:**

| docker attach <name> |
| --- |

**실행 중인 컨테이너의 과거 로그 보기:**

| docker logs <name> |
| --- |

**실행 중인 컨테이너의 과거 로그를 보고, attach 모드로 전환하기;**

| docker logs -f <name> |
| --- |

**실행 중인 컨테이너 안으로, 또는 실행 중인 컨테이너 밖으로 파일 또는 폴더 복사:**

| docker cp <source> <destination>  docker  ex\_)  docker cp dummy/. server1:/test  docker cp server1:/test dummy  docker cp server1:/test/test.txt dummy  // 여기서 server1은 컨테이너 이름이다. |
| --- |

**실행중인 컨테이너 조회:**

| docker ps |
| --- |

**모든 컨테이너 조회:**

| docker ps -a |
| --- |

**Dockerfile 이란?**

컨테이너 이미지를 생성하기 위한 설정 파일로,

애플리케이션 실행 환경을 코드로 정의하는 역할을 한다.

**Dockerfile 예시:**

| FROM node:14 # Node.js 14 버전 기반의 컨테이너 이미지 사용  WORKDIR /app # 작업 디렉토리를 /app으로 설정  COPY . . # 현재 디렉토리의 모든 파일을 컨테이너로 복사  RUN npm install # 필요한 Node.js 패키지 설치  EXPOSE 80 # 컨테이너가 80번 포트에서 실행됨을 명시  CMD ["node", "server.js"] # server.js 파일을 실행하여 컨테이너 시작 |
| --- |

**도커 이미지 레이어란?**

도커는 이미지를 레이어로 구성하고, 각 레이어는 변경사항만 추가합니다.

동일한 레이어는 재사용되어 빌드 속도를 향상시킵니다.

빌드 시 변경되지 않은 레이어는 캐시를 사용하지만,

Dockerfile의 명령어 순서나 내용에 따라 캐시가 무효화될 수 있습니다.

예를 들어, COPY . .가 변경되면 이후의 RUN npm install 명령도 다시 실행됩니다.

아래 처럼 코드를 바꾸면 괜찮습니다.

FROM node:14

WORKDIR /app

COPY paskage.json .

RUN npm install

COPY . .

EXPOSE 80

CMD ["node", "server.js"]

컨테이너가 실행될 때, 이미지의 환경과 코드를 복사하지 않습니다.

이미지의 레이어 위에 실행 레이어를 더 해서 실행됩니다.

**도커 이미지 공유 방법:**

Docker Hub 또는 개인 Registry에 이미지를 Push 또는 Pull 할 수 있다.

**Docker Hub 에 공유하는 방법:**

push를 하려면 로그인을 먼저 해야 한다.

**docker hub 로그인:**

| docker login |
| --- |

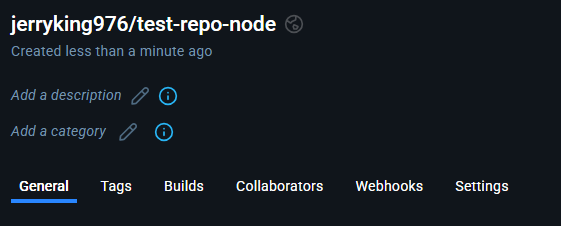
**docker hub 로그아웃:**

| docker logout |
| --- |

docker hub 홈페이지에서 repo를 만들어야 한다.

test-repo-node 라는 이름으로 만들었을 시,

아래처럼 앞에 계정 이름이 붙는다.



위에서 만든 repo로 내 image를 push 하기 위해서는,

imageName이 repoName과 정확히 일치해야 한다.

repoName은 계정명을 포함한 형태이므로 아래와 같다.

jerryking976/test-repo-node

**push:**

| docker push <ImageName> |
| --- |

**pull:**

| docker pull <**ImageName**> |
| --- |

**(중요) 볼륨(Volume)이란?**

docker의 이미지는 읽기 전용이고, 컨테이너는 읽기와 쓰기 모두 가능하다.

기본적으로 컨테이너에 무언가를 저장하면 임시적으로 저장된 것이다.

컨테이너가 삭제되면 저장한 내용이 사라진다.

볼륨은 컨테이너의 데이터를 지속적으로 저장하고 공유할 수 있도록 도와주는 저장소 방식이다. 컨테이너가 삭제되더라도 데이터가 유지된다.

볼륨은 호스트에 있는 폴더이고 이게 컨테이너 내부 폴더에 연결 된다.

**볼륨은 크게 2가지가 있다.**

1. **익명 볼륨:**--rm 옵션을 사용하는 컨테이너가 삭제되면 볼륨도 같이 삭제된다.   
   익명 볼륨은 하나의 컨테이너에만 밀접하게 연관되어 사용된다.  
   익명 볼륨이 마운트되는 경로가 이미지 내 해당 경로와 동일하면,  
   도커는 이미지 내의 데이터를 볼륨으로 복사한다.  
   “컨테이너 내의 데이터가 다른 것에 의해 덮어 써지는 것을 방지하는데 사용한다.”
2. **명명된 볼륨:**명명된 볼륨은 컨테이너가 삭제 되더라도, 볼륨이 삭제되지 않는다.  
   여러 컨테이너가 접근하여 사용할 수 있다.

두 볼륨 모두 일부 폴더와 경로를 호스트 머신에 세팅한다.

매핑되는 호스트 머신의 경로는 도커가 알아서 관리한다.

**익명 볼륨 만드는 방법:**

VOLUME [ "<path>" ] # Dockerfile 에서 사용

# 또는

docker -run -v <path> imageName

**명명된 볼륨 만드는 방법:**

익명 볼륨 처럼 Dockerfile에 VOLUME 키워드를 사용하지 않고,

컨테이너를 실행 할 때, 옵션을 통해 만든다.

| docker run -v <**volumeName**>:<**path**> <**imageName**> |
| --- |

**볼륨 리스트 확인:**

| docker volume ls |
| --- |

**볼륨 상세 정보 확인:**

| docker volume inspect <volumeName> |
| --- |

**익명 볼륨 삭제:**

| docker volume prune # 사용하지 않는 볼륨 모두 삭제  # 또는 docker volume rm <volumeName> |
| --- |

**바인드 마운트란?**

호스트의 특정 디렉터리나 파일을 컨테이너 내부에 직접 연결하는 방식의 마운트이다.

이를 통해 컨테이너 내부에서 호스트의 파일을 실시간으로 읽고 쓸 수 있다.

여러 컨테이너에서 동일한 데이터에 접근할 수 있다.

파일 또는 폴더 단위로도 바인딩 할 수 있다.

**개발 할 때 쓰고, 배포할 때는 쓰지 않는 것이 좋다.**

**호스트 파일 시스템을 직접 참조하기 때문에,**

**배포 환경이 바뀌면 경로가 달라져 동작이 보장되지 않는다.**

**바인드 마운트 설정하기:**

경로에 특수 문자 또는 공백이 포함되는 경우 “”로 묶으면 된다.

볼륨과 유사하게 컨테이너를 run 할 때 v 옵션을 넣지만, 값을 약간 다르게 넣는다.

-v 옵션을 여러 개 추가하여, 볼륨과 같이 사용할 수 있다.

hostPath는 $(pwd) 이런 식으로도 할 수 있다.

hostPath를 직접 입력할 경우, 반드시 절대 경로를 입력해야 한다.

| docker run -v <**hostPath**>:<**path**> <**imageName**> |
| --- |

**읽기 전용으로 마운트 하는 방법:**

볼륨 마운트에도 동일하게 사용 가능하다.

path 뒤에 :ro 를 붙이면 된다.

| docker run -v <**hostPath**>:<**path**>:ro <**imageName**> |
| --- |

**바인드 마운트 성능 최적화 (안전성 down):**

Docker는 delegated 옵션을 사용하여 성능을 우선시하고,

컨테이너에서 변경된 데이터가 즉시 호스트에 반영되지 않아도 괜찮다는 가정 하에 적용 된다.

| docker run -v <**hostPath**>:<**path**>:delegated <**imageName**> |
| --- |

**볼륨 마운트와 바인드 마운트 언제 사용해야 할까?**



**.dockerignore 파일이란?**

COPY 명령어로 이미지에 복사되지 않을 것들을 필터하는 역할이다.

gitignore랑 비슷하다고 생각하면 된다.

ex\_)

| # .dockerignore 파일 내부 # .git 폴더와 Dockerfile 파일을 무시합니다 Dockerfile .git |
| --- |

**빌드 타임 인수와 환경 변수:**

빌드 타임 인수는 이미지를 빌드할 때 사용하는 인수를 의미한다.

빌트 타임 인수는Dockerfile 내부에서만 사용 가능한다.

런타임 환경 변수는 코드 내에서 프로세스 환경 변수에 접근해 사용 가능하다.

빌드 타임 인수, 환경 변수의 key는 대문자로 지정한다.

**Dockerfile 에서 환경 변수 지정하기:**

| ENV <**key**> <**value**> |
| --- |

**Dockerfile 에서 환경 변수 사용하기:**

$를 붙이고, key를 입력하면 된다.

| EXPOSE $<**key**> |
| --- |

**docker run 명령에서 환경 변수 지정하기:**

Dockerfile 내에 ENV로 key, value가 선언되어 있어야 한다.

| docker run --env <key>=<value> <imageName>  # 또는  docker run -e <key>=<value> <imageName> |
| --- |

**docker run 명령에서 환경 변수를 저장한 파일을 사용하기:**

파일 내에서는 <key>=<value> 값을 입력해 놓으면 된다.

| docker run --env-file <filePath> <imageName> |
| --- |

**Dockerfile에서 빌드 타임 인수 지정하기:**

| ARG key=value |
| --- |

**docker build 명령에서 빌드 타임 인수 값 변경하기:**

| docker build --build-arg <key>=<value> . |
| --- |

**Dockerfile 에서 빌드 타임 인수 사용하기:**

| ENV PORT $<**key**> # 이렇게 환경 변수에 사용할 값으로도 사용 가능 |
| --- |

**컨테이너에서 외부로 WWW 통신하기:**

포트만 열려있다면 기본적으로 통신 가능하다. ex) REST API

**컨테이너에서 로컬 호스트 pc와 통신하기:**

localhost 대신 **host.docker.internal** 을 사용해야 한다.

**컨테이너간 통신하기 (기본):**

아래 명령어로 알아낸 컨테이너의 상세 정보로,

통신하고자 하는 컨테이너의 privateIp를 알아내어 연결하면 된다.

| docker container inspect <name> |
| --- |

**컨테이너간 통신하기 (추천):**

**컨테이너 네트워크**를 구성하여 통신한다.

docker run 명령을 할 때 네트워크를 지정하면 같은 networkName을

가지는 애들끼리 동일한 네트워크에 밀어 넣을 수 있다.

같은 네트워크에 있는 애들은 port를 열지 않아도 된다.

우선 network를 먼저 만들어야 한다.

| docker network create <**networkName**> |
| --- |

그 다음 만들어진 network를 옵션에 사용한다.

| docker run --network <**networkName**> <**imageName**> |
| --- |

코드에서는 특정 ip를 하드코딩 하지 않고,

docker의 containerName을 그대로 쓰면 된다.

**만들어진 모든 네트워크 보기:**

| docker network ls # 기본 내장된 네트워크도 출력된다. |
| --- |

**네트워크 삭제:**

| docker network rm <networkName> |
| --- |

**네트워크 생성 시 설정할 수 있는 driver 종류:**

**bridge**(기본값): 컨테이너 간 통신을 위한 가상 네트워크 브리지를 생성한다.

같은 브리지 네트워크에 있는 컨테이너들끼리 직접 통신 가능하다. (동일 호스트 머신에서만 가능)

**host**: 컨테이너가 호스트의 네트워크 네임스페이스를 공유한다.

컨테이너 내부에서 실행하는 서비스가 호스트의 IP 주소와 포트를 그대로 사용한다.

**overlay**: 여러 호스트에서 실행되는 컨테이너를 하나의 네트워크에 있는 것처럼 연결한다.

여러 컨테이너를 연결하는 구식의 / 거의 사용되지 않는 방법인 'Swarm' 모드에서만 작동합니다.

**macvlan**: 컨테이너에 커스텀 MAC 주소를 설정할 수 있습니다.  
고성능이 필요하거나, 기존 네트워크 환경과 통합해야 할 때 사용.

**(중요)** **Docker Compose 란?**

단일 머신에서 다중 컨테이너 어플리케이션을 정의하고 실행할 수 있도록 해주는 도구입니다.

개발 환경에서 동일한 설정을 여러 명이 공유할 때,

CI/CD 파이프라인에서 테스트 환경을 빠르게 구성할 때 에도 사용합니다.

(배포 환경에서는 다중 머신을 사용하기 때문에 잘 사용하지 않는 듯 하다.)

docker build, run 같은 명령어를 일일이 입력하는 것을 대체한다.

**제공 기능:**

1. **YAML 파일(docker-compose.yaml)을 사용하여 서비스 정의**

여러 개의 컨테이너 설정을 한 곳에서 관리 가능

하나의 **서비스**는 하나의 **컨테이너**를 실행하는 역할을 함.

예: 웹 서버 + 데이터베이스 + 캐시 시스템 등

1. **단일 명령으로 컨테이너 관리 (up, down, restart 등)**

$ docker compose up # 모든 서비스 실행

$ docker compose down # 모든 서비스 중지 및 정리

1. **환경 변수 및 볼륨 관리 지원**

.env 파일로 환경 변수 설정 가능

데이터 볼륨을 사용해 컨테이너 종료 후에도 데이터 유지

1. **네트워크 자동 생성 및 서비스 간 통신 지원**

같은 docker-compose.yml에 정의된 컨테이너들은 자동으로 같은 네트워크에 포함됨

서비스 이름(service\_name)으로 직접 통신 가능 (db:3306 같은 방식)

**Docker Compose 설치 (리눅스):**

윈도우, 맥은 기본 내장 되어 있다.

| sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose   sudo chmod +x /usr/local/bin/docker-compose  sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose  # 아래 커맨드로 설치 확인  docker-compose --version |
| --- |

**Docker Compose로 실행 시 특징:**

1. 도커 컴포즈로 실행 시 넘기는 인자로, -d 옵션을 활성화 할 수 있다.  
   docker-compose.yaml 내에서도 지정할 수 있다.
2. 동일한 yaml 파일 내의 서비스들은 같은 도커 네트워크에 소속됨으로  
   특별한 상황이 아니면 별도로 네트워크를 지정하지 않아도 된다.

**docker-compose.yaml 예시 코드:**

yaml 가이드s 문서: <https://teddylee777.github.io/python/yaml/>

docker-compose.yaml 문서:<https://docs.docker.com/reference/compose-file/services/>

# docker-compose.yaml

name: myapp

services:

mongodb:

container\_name: mongodb # 컨테이너 이름 지정

image: 'mongo'

restart: always # 항상 재시작 (docker compose stop 으로만 멈추기 가능)

privileged: true # 호스트 시스템과 동일한 특권을 줌

volumes:

- data:/data/db

environment: # 런타임 환경 변수

MONGO\_INITDB\_ROOT\_USERNAME: max

MONGO\_INITDB\_ROOT\_PASSWORD: secret

# env\_file:

# - <filePathAndName>

backend:

container\_name: backend

# 이미지를 굽는 동작으로, Dockerfile 이 있는 경로를 입력한다.

# Dockerfile은 복사할 폴더와 같은 곳에 있어야 한다.

build: ./backend

# 아래와 같은 방식으로도 사용 가능

# build:

# context: ./backend

# dockerfile: Dockerfile

# args: # 빌드 인자를 의미한다.

# some-arg: 1

volumes:

- logs:/app/logs

- ./backend:/app # 바인드 마운트인데, 상대 경로로 설정할 수 있다.!

- /app/node\_modules

ports:

- '80:80'

environment:

MONGO\_INITDB\_ROOT\_USERNAME: max

MONGO\_INITDB\_ROOT\_PASSWORD: secret

# depends\_on은 서비스 간의 의존성을 설정한다.

# 예를 들어, mongodb가 실행된 후 backend가 실행되게 한다.

# 만약, 특정 서비스만 지정해서 실행 하더라도

# 해당 서비스의 depends\_on인 서비스는 같이 실행 해준다.

depends\_on:

- mongodb

frontend:

container\_name: frontend

build: ./frontend

volumes:

- ./frontend/src:/app/src

ports:

- '3000:3000'

depends\_on:

- backend

stdin\_open: true # -i 활성화

tty: true # -t 활성화

# entrypoint: [“command”, “arg”] 요런식으로 entrypoint도 지정 가능하다.

# command: ["command", “arg”] 요런식으로 cmd도 지정 가능하다.

# 명명된 볼륨만 여기에 지정한다.

volumes:

data:

logs:

**docker-compose로 실행:**

이미지도 빌드되고, 컨테이너도 실행된다.

| docker-compose up |
| --- |

**docker-compose로 실행 시 특정 서비스들만 실행:**

| docker-compose up <serviceName> <serviceName> |
| --- |

detach 모드로 실행하려면 -d 옵션을 추가한다.

| docker-compose up -d |
| --- |

**docker-compose 종료:**

컴포즈로 실행된 모든 컨테이너 종료 및 삭제한다.

이미지랑 볼륨은 유지 된다.

| docker-compose down |
| --- |

볼륨도 삭제하려면 -v 옵션을 추가해야 한다.

| docker-compose down -v |
| --- |

**docker-compose로 이미지 리빌드:**

docker-compose up 할 때 알아서 이미지 리빌드를 하지 않기 때문에, 때때로 필요하다.

| docker-compose build  # 요런 식으로 up 하면서 rebuild도 하게 할 수 있다.  docker-compose up --build |
| --- |

**docker-compose의 특정 서비스만 실행:**

| docker-compose start <**serviceName**> |
| --- |

**docker-compose로 실행된 특정 서비스만 종료:**

| docker-compose stop <serviceName> |
| --- |

**docker-compose로 실행된 서비스들 조회:**

| docker-compose ps |
| --- |

**docker-compose의 특정 서비스의 log 조회:**

| docker-compose logs <serviceName> # -f 옵션 사용가능 |
| --- |

**docker-compose의 실행중인 특정 서비스에 명령을 요청:**

(자세한 설명은 ‘docker exec 명령이란?’ 부분 참고)

| docker-compose exec <**serviceName**> <**command**> |
| --- |

**docker-compose의 특정 서비스를 일회성으로 명령을 처리하게 실행:**

(자세한 설명은 ‘docker run을 통하여 명령어 전달하는 방법’ 참고)

| docker-compose run <serviceName> <command>  # --rm 명령어를 같이 쓰는게 좋습니다.  # ex\_) docker-compose run --rm <serviceName> <command> |
| --- |

**docker exec 명령이란?**

실행 중인 컨테이너 내부에서 새로운 프로세스를 실행할 때 사용된다.

기존 컨테이너에서 추가 작업을 수행할 때 유용 하다.

컨테이너가 이미 실행 중이어야 한다. (docker run과 다름)

컨테이너의 기본 PID 1 프로세스와 별개로 명령 실행 한다.

**docker exec 사용법?**

| docker exec [option] <**containerName**> <**command**>  # ex\_) docker exec -it my\_container bash |
| --- |

**docker run 을 통하여 명령어를 전달하는 방법:**

아래 방법을 사용하면, Dockerfile의 CMD 명령어는 무시된다.

Dockerfile ENTRYPOINT는 무조건 명령을 실행 한다.

docker run의 command는 ENTRYPOINT의 인자로 들어가는 형태가 된다.

| docker run <imageName> <command>  # --rm 명령어를 같이 쓰는게 좋습니다. # ex\_) docker run --rm ubuntu ls # ls 명령어 수행하고 container 종료됨 |
| --- |

| #Dockerfile FROM node:14  WORKDIR /app  ENTRYPOINT [ "npm" ]  # 아래처럼 인자 까지 넘겨도 됨.  # ENTRYPOINT [ "npm", “init” ] |
| --- |

**Dockerfile의 CMD와 ENTRYPOINT:**

**CMD**:

컨테이너 실행 시 기본 실행 명령어.

**ENTRYPOINT**:

컨테이너 실행 시 반드시 실행될 명령어.

**둘의 차이**:

CMD는 ENTRYPOINT의 기본 인자로 사용되며,

ENTRYPOINT는 실행 명령을 강제로 지정.

**CMD, ENTRYPOINT 둘 다 작성 안했다면?:**

FROM으로 받은 Base 이미지의 CMD, ENTRYPOINT가 호출된다.

**애플리케이션 컨테이너, 유틸리티 컨테이너?**

**애플리케이션 컨테이너:**

환경과 앱 코드가 같이 있어, 앱을 구동 하는 컨테이너 이다.

**유틸리티 컨테이너:**

환경만 있고 이 환경을 사용하여 특정 명령어를 전달해서 사용하는 컨테이너 이다.

환경만 실행하고 docker exec를 쓰거나, docker run의 command 인수 그리고, dockerfile의 ENTRYPOINT를 활용하여 목적에 맞게 사용하는 형태이다.

**멀티 스테이지 빌드란?**

하나의 Dockerfile에서 여러 개의 빌드 단계를 사용하여

불필요한 파일을 포함하지 않고 최적화된 이미지를 생성하는 기법이다.

**새로운 스테이지를 만드는 방법:**

Dockerfile에서 FROM을 하면 새로운 스테이지가 만들어 진다.

첫 번째 스테이지에서 설치한 패키지나 파일은 두 번째 스테이지로 자동으로 넘어가지 않는다.

최종적으로 빌드된 이미지는 마지막 스테이지에서의 파일만 포함된다.

**스테이지에 별칭을 붙이는 방법:**

FROM 문 맨 뒤에 as 와 별칭을 쓴다.

| FROM <image> as <stageName> |
| --- |

**이전 스테이지의 파일을 복사하고자 할 때:**

이때 sourcePath는 컨테이너 상의 경로를 입력한다.

| COPY --from=<**stageName**> <**sourcePath**> <**targetPath**> |
| --- |

**특정 스테이지만 이미지로 빌드하고 싶을 때:**

| docker build --target <**stageName**> -f <**DockefileName**> <**dockerfilePath**> |
| --- |

**멀티 스테이지 빌드 Dockerfile 예시:**

