

Docker Story



빠른 사용을 위한 도커 이야기

빠른 사용을 위한 도커 이야기는 도커를 잘 활용하기 위해 필요하다고 생각하는 개념을 이해하기 쉽게 정리한 강의입니다.

해당 강의에서는 가상화의 개념과 도커의 개념 및 도커를 구성하는 구성요소들의 개념들을 다룹니다.



DOCKER
DOCKER 2024

강의 개요

1

가상화

도커가 어떤 기술인지에 대해 알기 위해서는 가상화에 대해 알아봅니다.

2

도커의 구성

도커를 활용하면서 사용할 구성요소는 무엇이 있는지 알아봅시다.

4

도커 관련 툴

도커를 잘 활용하기 위해 만들어진 기술들의 개념을 알아봅시다.

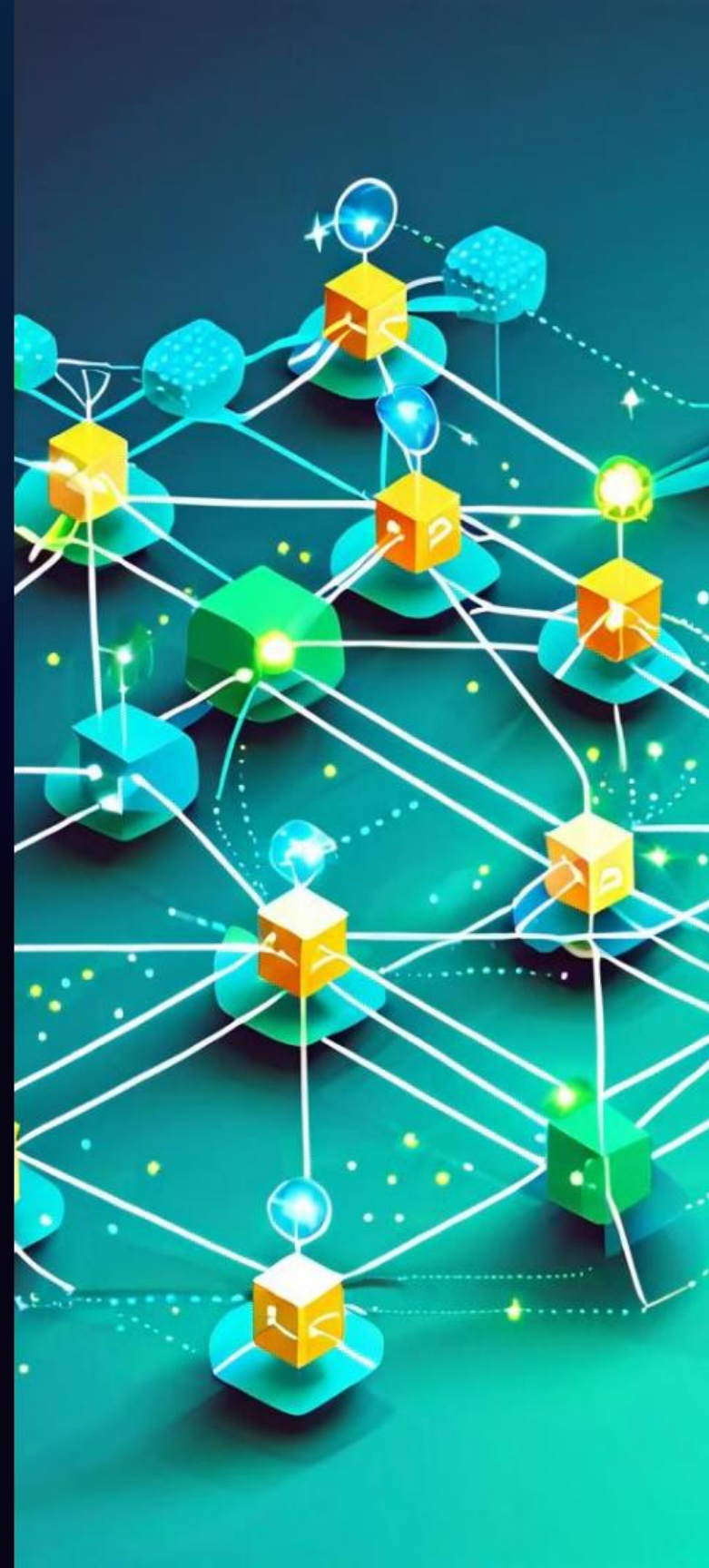
5

도커 사용 예시

나만의 이미지를 만들어서 활용해봅시다.

가상화

- 1 — 전통적 배포
- 2 — 가상머신 기반
- 3 — 컨테이너 기반



가상화

가상화의 정의

가상화는 물리적 하드웨어 리소스를 논리적으로 나누어, 여러 개의 독립된 운영체제와 애플리케이션을 동시에 실행할 수 있도록 하는 기술
즉, 하나의 컴퓨터를 여러 대의 컴퓨터처럼 활용이 가능



가상환경 툴



가상화 관련 툴

가상화

전통적 배포

컴퓨터에 특정 서비스를 설치하여 제공하는
것이 가장 일반적
다른 소프트웨어와 충돌 시 복구가 어려움



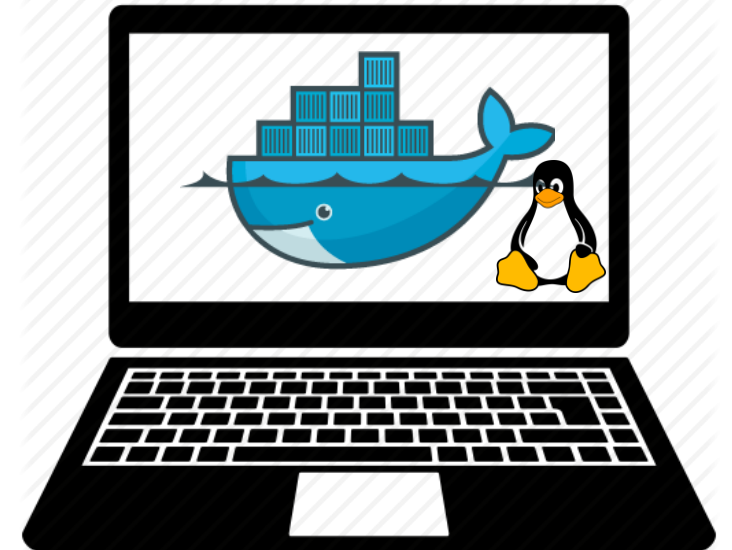
가상머신 기반

운영체제 내에서 운영체제를 구동 시키는
기술



컨테이너 기반

호스트 OS의 기능을 그대로 사용하면서
프로세스를 격리해 독립된 환경을 만드는
기술



가상화

전통적 배포

설치하기 위한 장비와 동일한
운영체제 및 하드웨어에
테스트 진행 후 배포

가상머신 기반

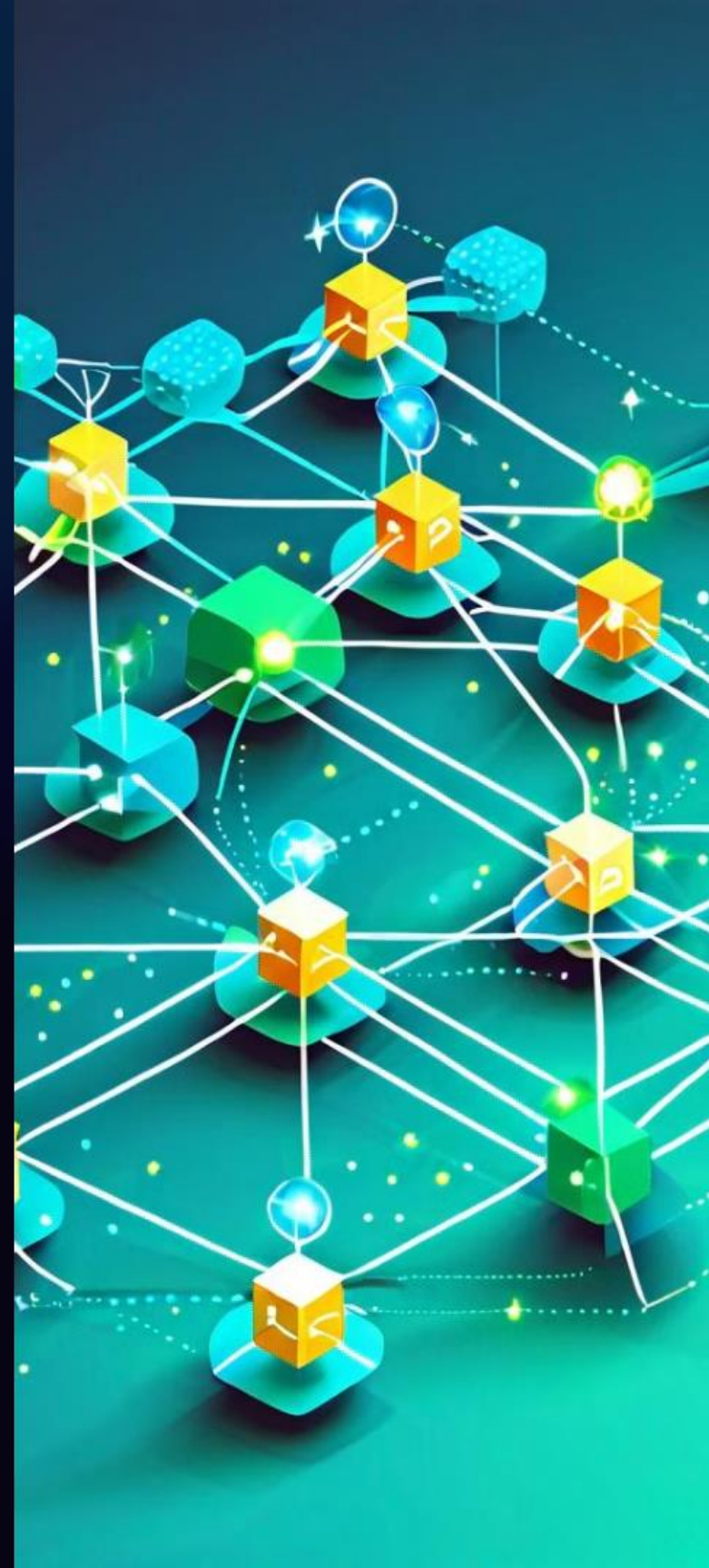
한 컴퓨터 내에서 테스트 할
운영체제들을 구동 시키는 기술
운영체제들 간에 서로 격리
용량을 많이 차지

컨테이너 기반

호스트의 OS 기능을 그대로
활용하면서 프로세스들을 서로
격리해 독립된 환경을 만드는 기술

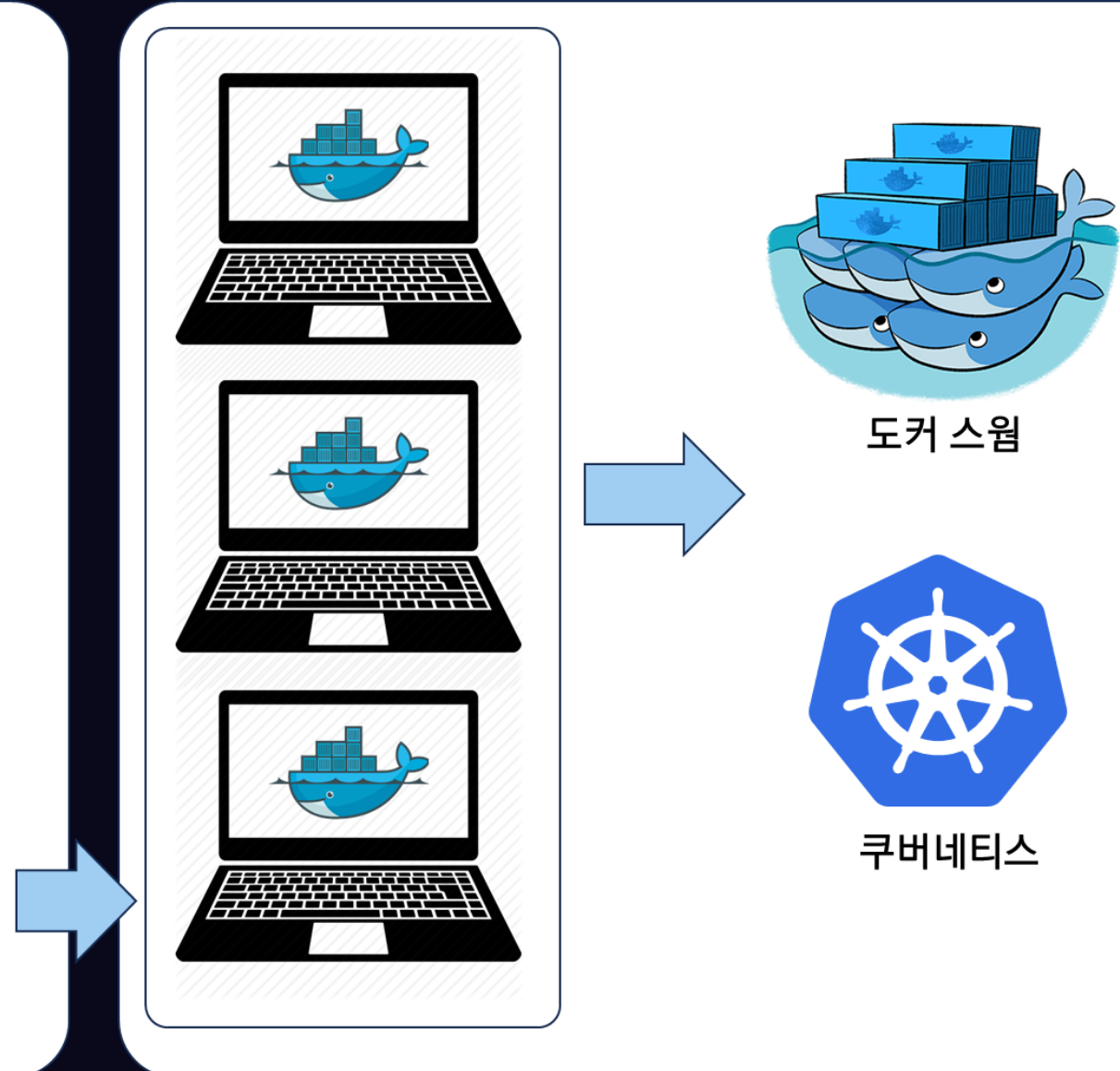
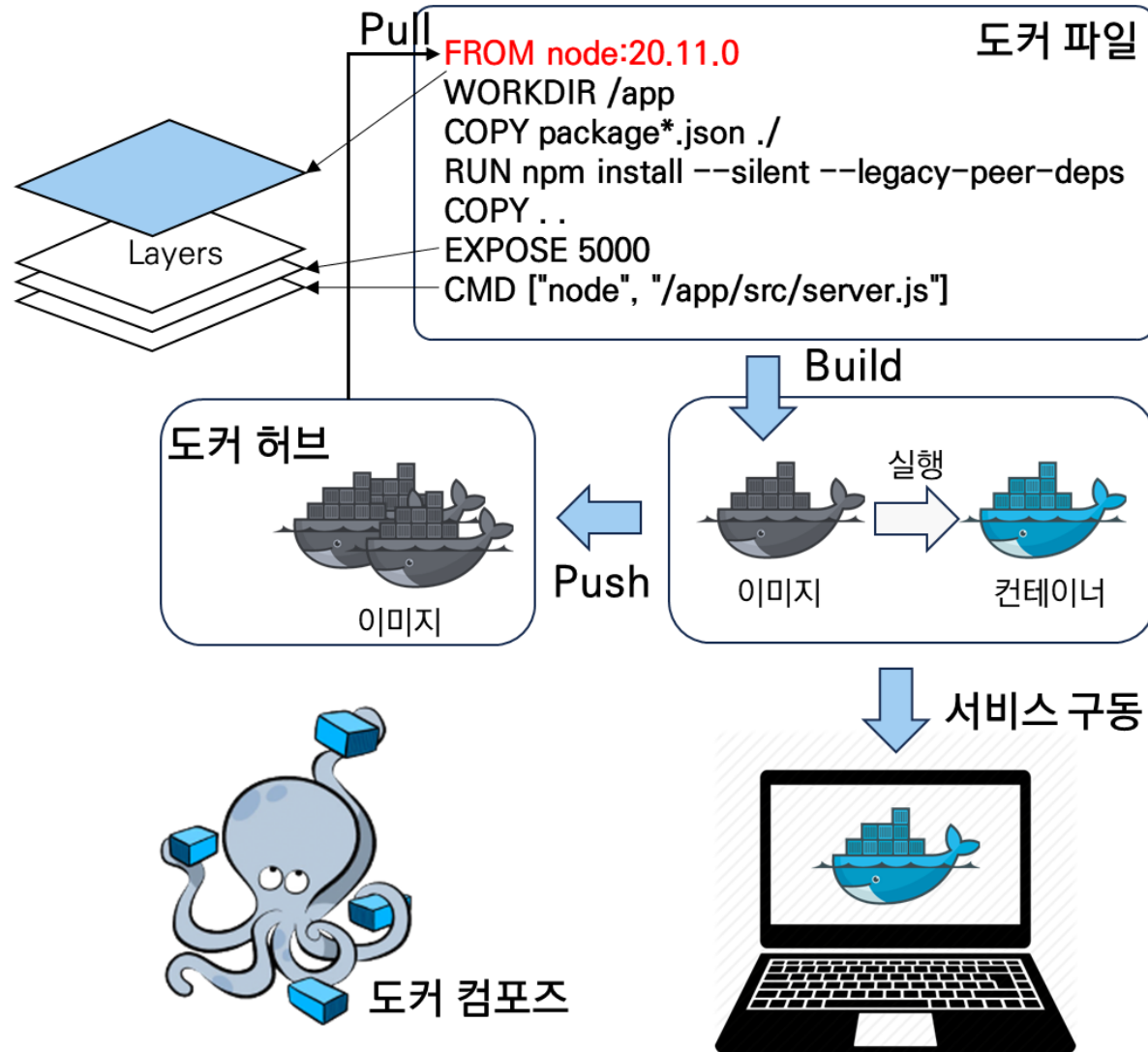
도커의 구성

- 1 — 도커 파일
- 2 — 도커 이미지
- 3 — 도커 컨테이너



도커의 구성

도커 생태계



도커의 구성

도커 파일, 이미지, 컨테이너


도커 관련 명령어는 `docker --help` 를 통해 확인 가능
도커 파일 관련 명령어는 링크 참고(<https://docs.docker.com/reference/dockerfile>)
주로 사용하는 도커 파일 생성을 위한 명령어는 아래와 같음

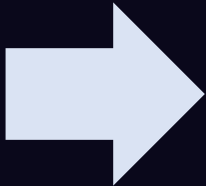
```
FROM ubuntu:24.04

WORKDIR /app
RUN apt-get update -y
RUN apt-get install vim -y

ENV TEST "test"

COPY ./test.txt ./
```

 `docker build -t my_app:latest -f ./test.Dockerfile .`



명령어	설명
FROM	특정 이미지에서 새 빌드 단계 생성
RUN	빌드 명령어 실행
ENV	환경 변수 설정
WORKDIR	명령어 기본 작업 경로 변경
EXPOSE	특정 포트 수신 대기
CMD	기본 명령어 지정
ENTRYPOINT	기본 실행파일 지정
USER	사용자 및 그룹 ID 설정
MAINTAINER	이미지 작성자 지정
LABEL	이미지 메타 데이터 작성

도커의 구성

도커 파일, 이미지, 컨테이너

도커는 컨테이너 기반의 가상화 기술(여러 개의 운영체제 및 애플리케이션을 동시에 실행하는 기술)
도커 파일은 도커 이미지를 만들기 위한 스크립트 파일이며, 도커 이미지가 실행된 상태를 컨테이너라 함

```
FROM ubuntu:24.04
```

```
WORKDIR /app
```

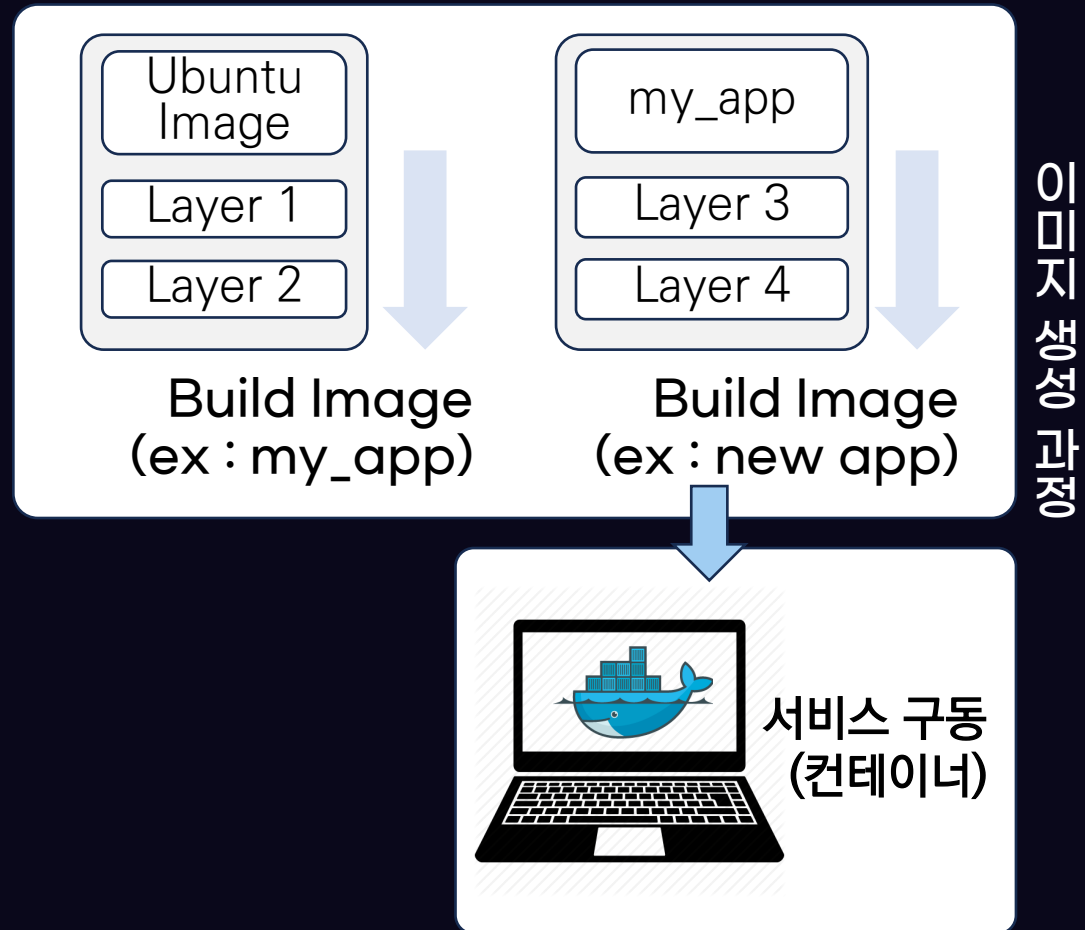
```
RUN apt-get update -y
```

```
RUN apt-get install vim -y
```

```
ENV TEST "test"
```

```
COPY ./test.txt ./
```

➡ `docker build -t my_app:latest -f ./test.Dockerfile .`

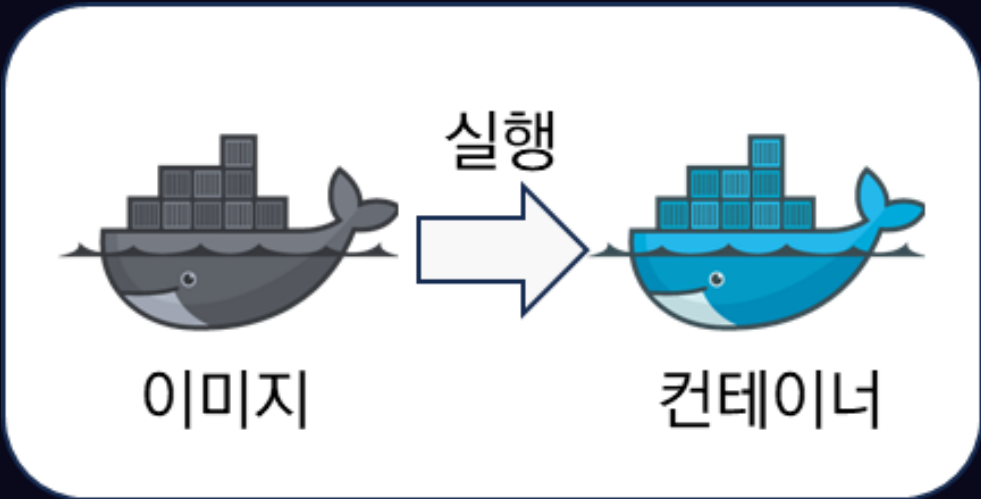


도커의 구성

도커 파일, 이미지, 컨테이너

도커 관련 명령어는 `docker --help` 를 통해 확인 가능
도커 실행 관련 명령어는 `docker run --help` 를 통해 확인 가능
주로 사용하는 도커 이미지 실행 명령어는 아래와 같음

```
docker run -d --name=〈컨테이너명〉  
--restart=always -p 3306:3306  
-v /data/mysql:/var/lib/mysql  
-e MYSQL_ROOT_PASSWORD=〈비밀번호〉  
mysql:latest
```



명령어	설명
--name	컨테이너 이름 설정
--publish, -p	포트 포워딩 설정
--volume, -v	볼륨(경로 내/외부) 설정
--env, -e	환경변수 설정
--rm	컨테이너 종료 시 삭제
--network	네트워크 설정(host 자주 사용)
--cpus	cpu 제한
--gpus	gpu 제한
--memory, -m	메모리 제한
--detach, -d	컨테이너 백그라운드 실행
--restart	재시작 여부
--interactive, -i	명령어 상호 작용
--tty, -t	shell 통신

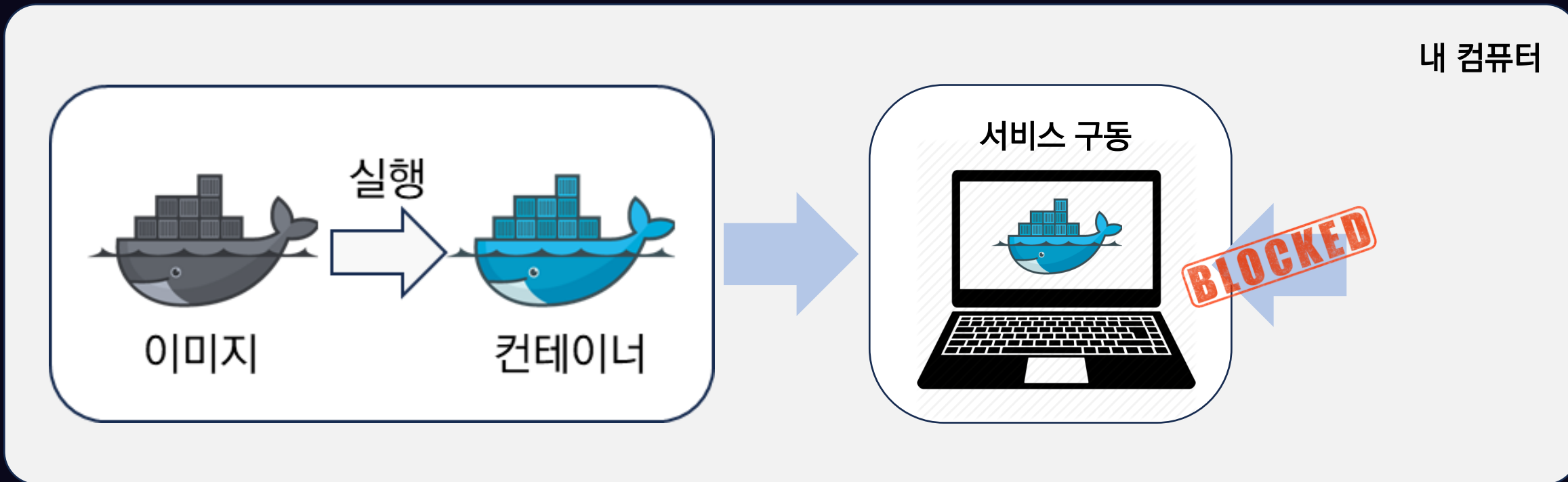
도커의 구성

도커 파일, 이미지, 컨테이너

도커 관련 명령어는 `docker --help` 를 통해 확인 가능

도커 목록은 `docker ps -a`

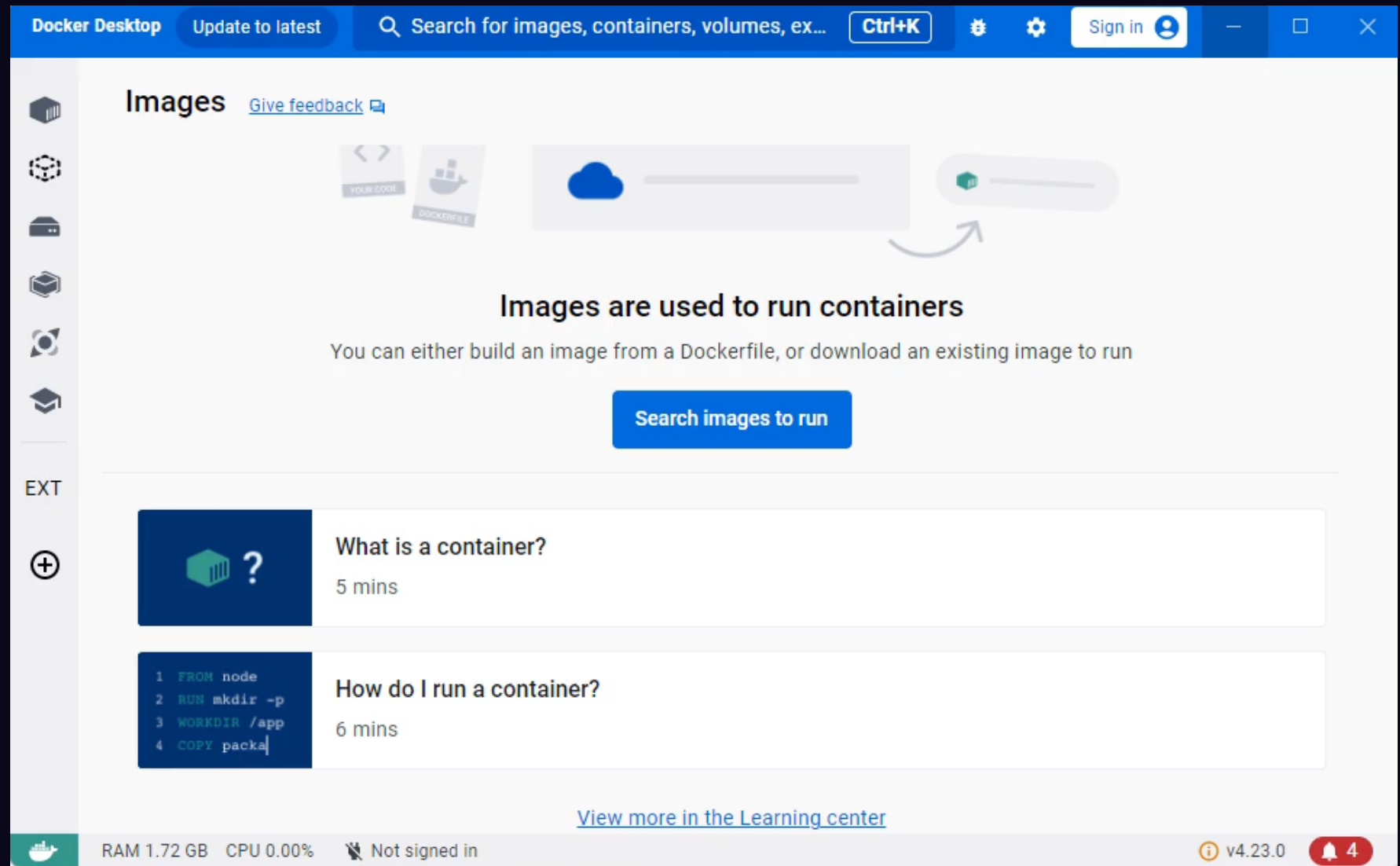
도커 컨테이너 접속(연결)은 `docker exec -it <컨테이너이름> bash`



도커의 구성

도커 파일, 이미지, 컨테이너

1. 도커 파일을 활용한 이미지 생성(5개의 레이어 생성 과정 확인)
2. 도커 이미지를 컨테이너로 실행
3. 도커 컨테이너 내부 접속
4. 환경 변수 적용 확인
5. 도커 이미지 조회
6. 도커 컨테이너 조회
7. 도커 컨테이너 중지
8. 도커 컨테이너 삭제
9. 도커 이미지 삭제



도커의 구성

도커 파일

도커 이미지를 만드는
스크립트 파일로 build
명령어를 활용해 도커
이미지를 생성할 수 있음

도커 이미지

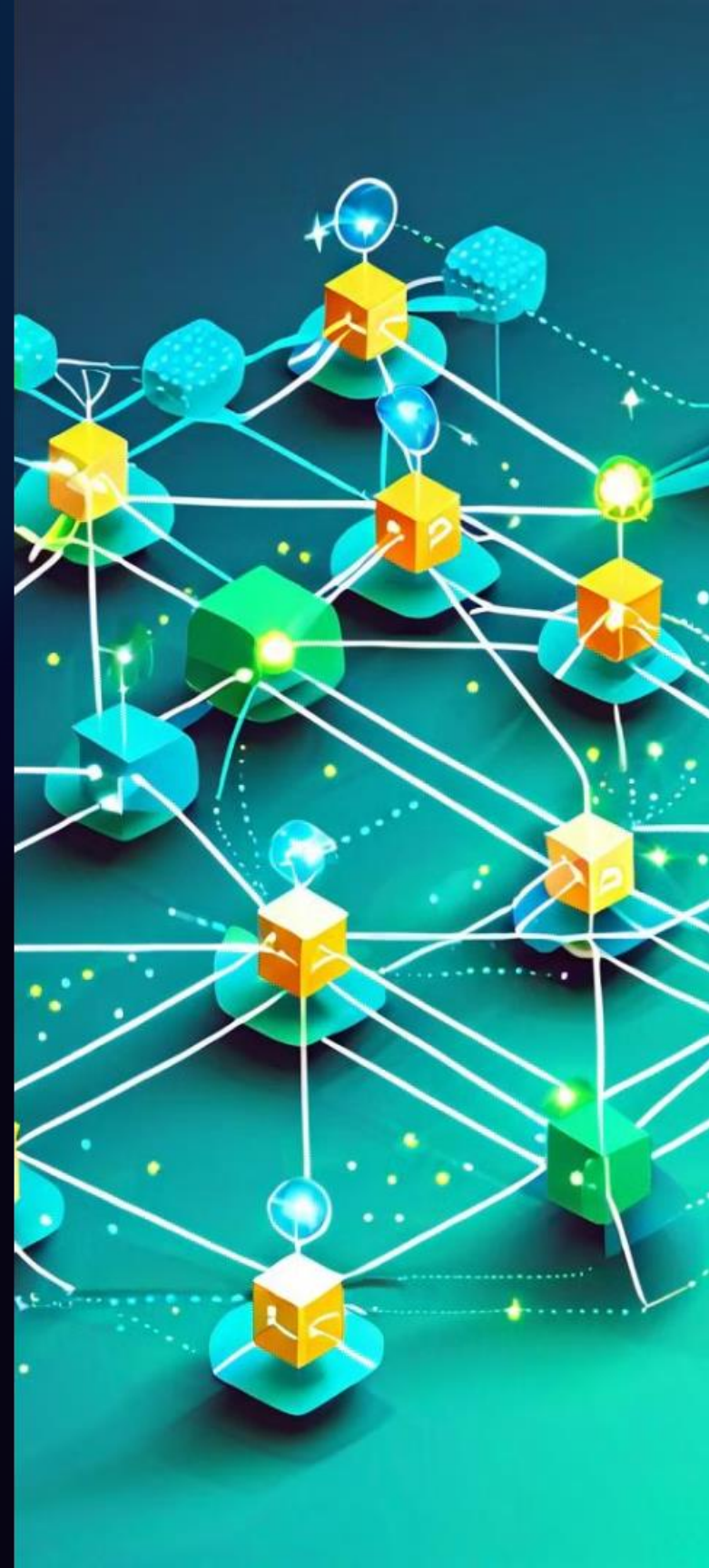
도커 이미지는 도커가
컨테이너 상태로 실행되기
전의 상태이며, 도커
파일로 구축된 상태를 의미

도커 컨테이너

도커 이미지가 실행된
상태를 의미

도커 관련 툴

- 1 — 도커 컴포즈
- 2 — 도커 스웸
- 3 — 쿠버네티스



도커 관련 툴

도커 컴포즈

Docker run 명령어로 실행 및 관리가 가능하지만, 컨테이너가 많아지면, 관리가 힘들어짐

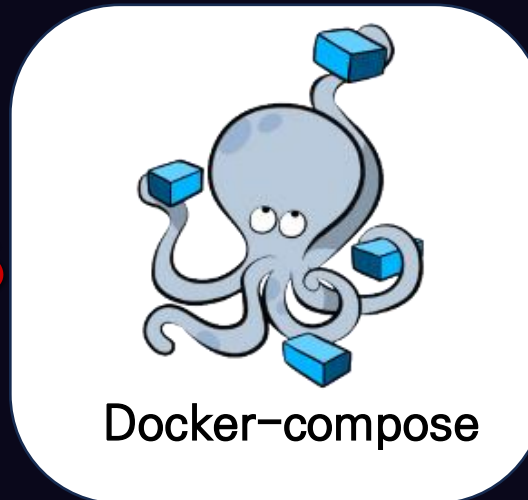
구동한 컨테이너에 어떤 옵션을 주었는지 기억하는 것은 비효율적

도커 컴포즈는 docker run 명령어 대신 yaml 확장자로 컨테이너가 실행 될 수 있게 관리 해주는 툴

docker-compose.yml



VS



```
docker run -d --name=〈컨테이너명〉
--restart=always -p 3306:3306
-v /data/mysql:/var/lib/mysql
-e MYSQL_ROOT_PASSWORD=〈비밀번호〉
mysql:latest
```

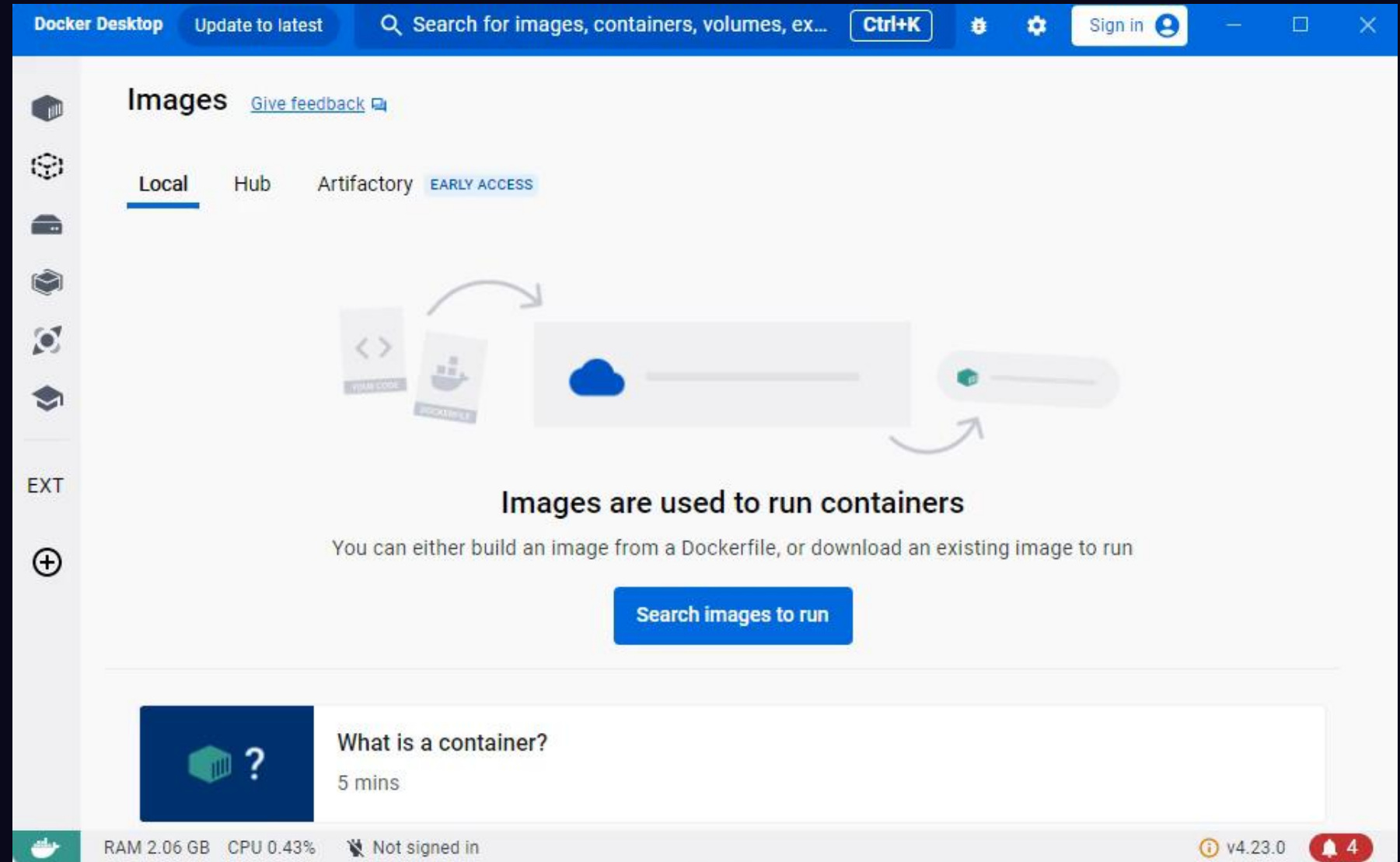
```
docker-compose up -d
```

```
version: "3.8"
services:
  image: mysql:latest
  container_name: 컨테이너명
  volumes:
    - ./mysql:/var/lib/mysql
  ports:
    - 3306:3306
  environment:
    - MYSQL_ROOT_PASSWORD: 〈비밀번호〉
```


도커 관련 툴

도커 컴포즈

1. 도커 이미지를 컨테이너로 실행
 - 1-1. 도커 허브의 mysql:latest 이미지 가져오기(=docker pull mysql)
2. 도커 컨테이너 삭제
3. 도커 컴포즈를 활용한 컨테이너 실행
4. 도커 컴포즈를 활용한 컨테이너 삭제

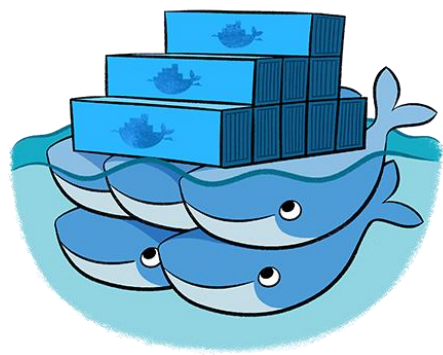


도커 관련 툴

도커 스웜 및 쿠버네티스

도커가 하나의 컴퓨터를 여러 컴퓨터처럼 사용하는 것 이었다면,
도커 스웜이나 쿠버네티스는 여러 컴퓨터를 하나처럼 사용하는 기술

도커 스웜은 주로 하둡과 같은 클러스터 서버를 구축할 때 주로 사용
쿠버네티스는 무중단 배포나 로드밸런싱 등 서비스 관리에 주로 사용



Docker Swarm

VS

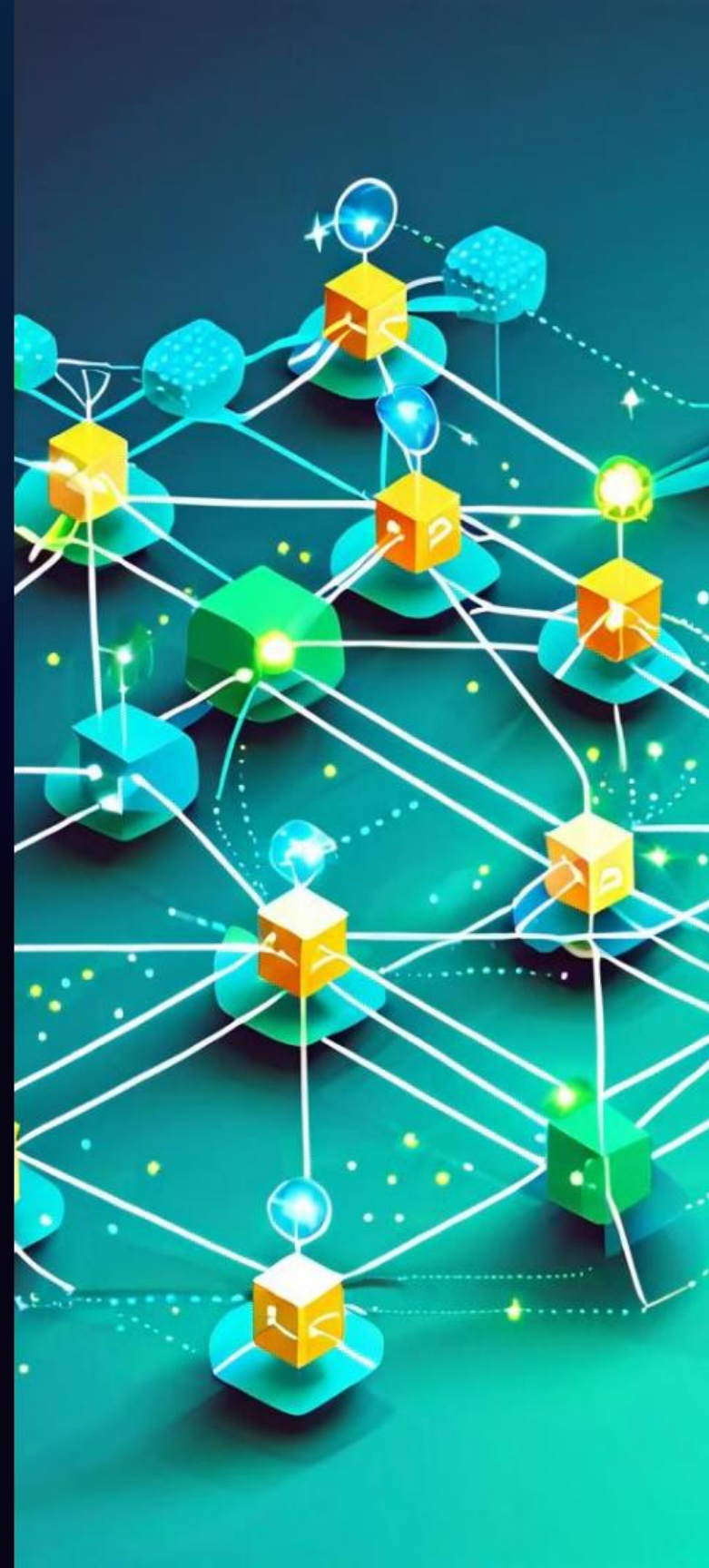


Kubernetes

구분	Docker Swarm	Kubernetes
개발 회사	Docker Inc	Google
컨테이너 관리 규모	10~20	100~1000
설치 및 접근성	간단	복잡
확장성	낮은 자유도	높은 자유도

도커 사용 예시

- 1 — GPT API
- 2 — SQLD 실습환경
- 3 — 웹 서버 세팅



도커 사용 예시

GPT API

docker를 활용한 python Backend(Chat GPT API 구현)
API TEST를 위해 POSTMAN 과 같은 소프트웨어를 많이 사용함

```
FROM python:3.8.19-slim  
WORKDIR /app
```

```
COPY requirements.txt ./  
RUN pip install --no-cache-dir -r requirements.txt
```

```
docker build -t my_gpt_api:0.0.0 -f ./gpt_api.Dockerfile . --no-cache
```

```
docker run --rm -p 3002:3002 --name my-gpt -v  
${PWD}/scripts:/app my_gpt_api:0.0.0 uvicorn app:app --host  
0.0.0.0 --port 3002
```

```
import os  
from fastapi import FastAPI  
from fastapi.responses import JSONResponse  
from fastapi.middleware.cors import CORSMiddleware  
import langchain  
from langchain.chat_models import ChatOpenAI  
from pydantic import BaseModel
```

```
class Item(BaseModel):  
    key: str  
    comment: str
```

```
app = FastAPI()  
app.add_middleware(  
    CORSMiddleware,  
    allow_origins=["*"],  
    allow_credentials=True,  
    allow_methods=["GET", "POST", "PUT", "DELETE"],  
    allow_headers=["*"],  
)
```

```
@app.post("/pre-post")
```

```
async def process_item(item: Item):
```

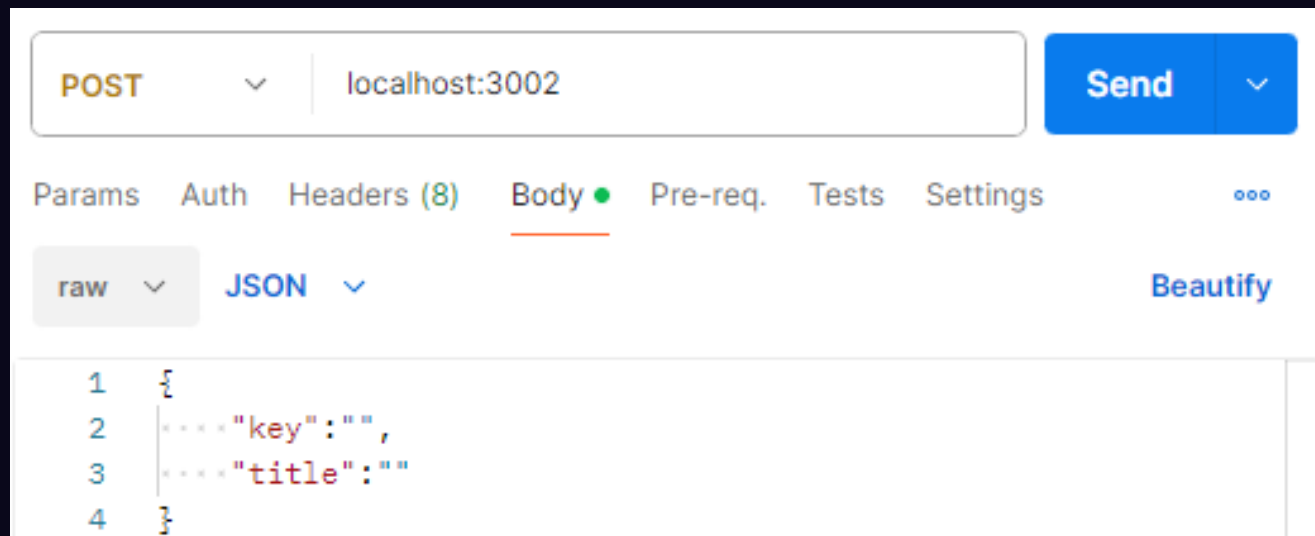
```
    try:  
        api_key = item.key  
        comment = item.comment  
        os.environ['OPENAI_API_KEY'] = api_key  
        chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)  
        answer=chat.predict(comment)  
        return JSONResponse(status_code=200, content={"result":answer})  
    except:  
        return JSONResponse(status_code=200, content={"result":"Open AI Error"})
```



도커 사용 예시

GPT API

docker를 활용한 python Backend(Chat GPT API 구현)
API TEST를 위해 POSTMAN 과 같은 소프트웨어를 많이 사용함



```
import os  
from fastapi import FastAPI  
from fastapi.responses import JSONResponse  
from fastapi.middleware.cors import CORSMiddleware  
import langchain  
from langchain.chat_models import ChatOpenAI  
from pydantic import BaseModel
```



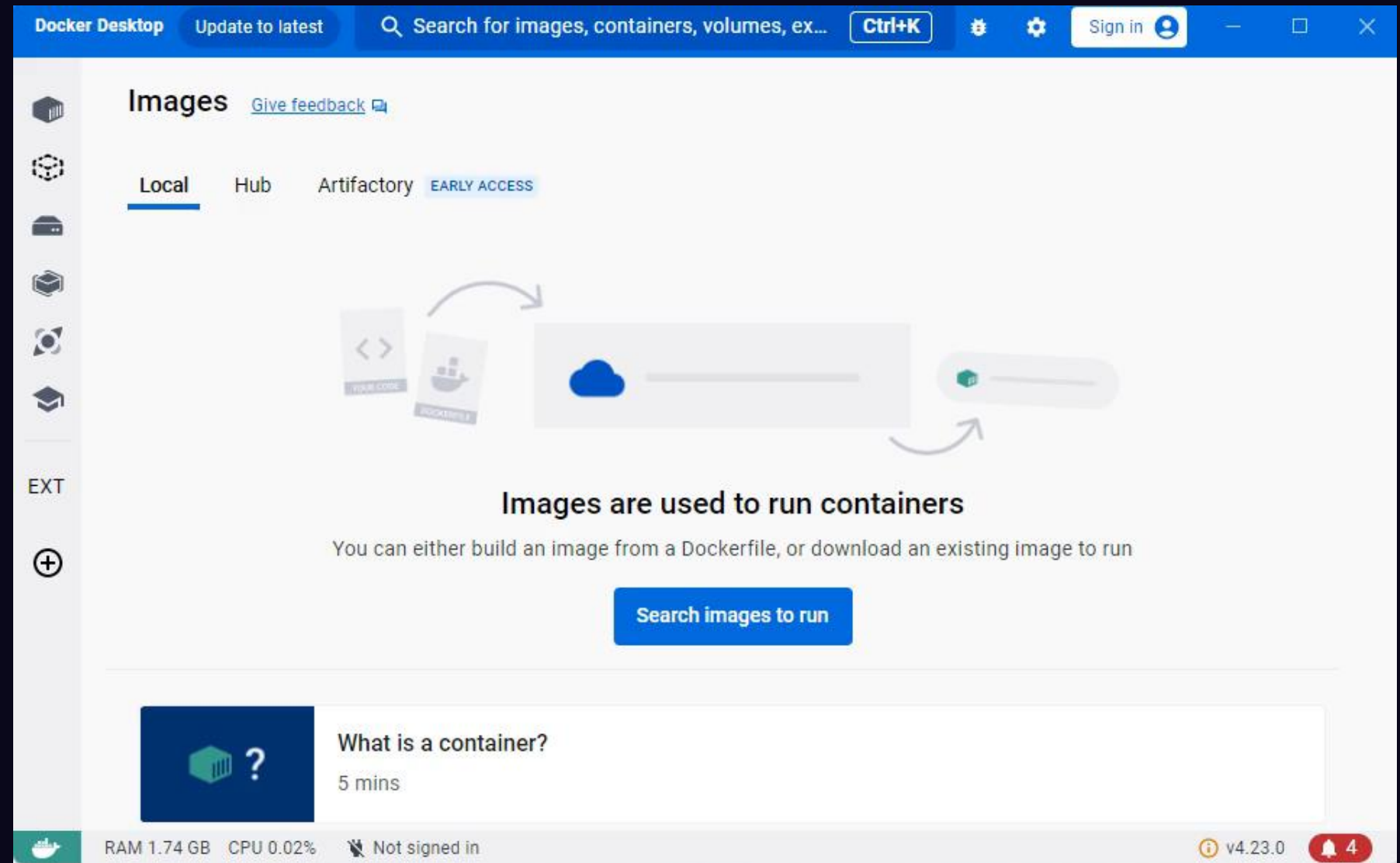
```
class Item(BaseModel):  
    key: str  
    comment: str
```

```
app = FastAPI()  
app.add_middleware(  
    CORSMiddleware,  
    allow_origins=["*"],  
    allow_credentials=True,  
    allow_methods=["GET", "POST", "PUT", "DELETE"],  
    allow_headers=["*"],  
)  
@app.post("/pre-post")  
async def process_item(item: Item):  
    try:  
        api_key = item.key  
        comment = item.comment  
        os.environ['OPENAI_API_KEY'] = api_key  
        chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)  
        answer=chat.predict(comment)  
        return JSONResponse(status_code=200, content={"result":answer})  
    except:  
        return JSONResponse(status_code=200, content={"result":"Open AI Error"})
```

도커 사용 예시

GPT API

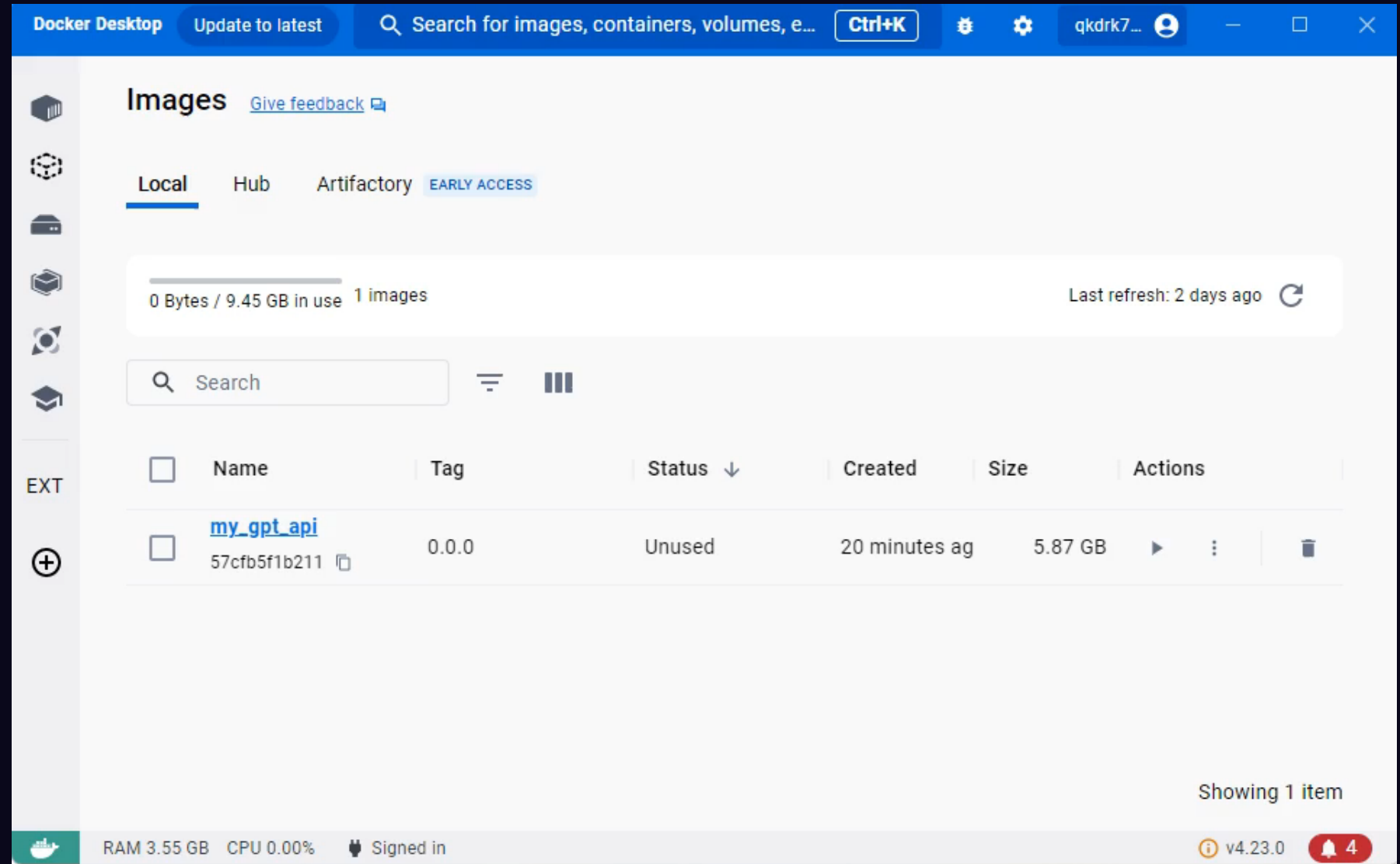
1. 도커 이미지 생성
2. 도커 컨테이너를 활용한 GPT API 구현



도커 사용 예시

GPT API

1. 도커 이미지 태그 변환
2. 도커 로그인
3. 도커 이미지 도커 허브로 전송
4. 도커 허브 이미지를 활용한 GPT API 구현



도커 사용 예시

SQLD 실습 환경

docker-compose 환경 구축은 오른쪽과 같음
DB의 버전에 따라서 명령어나 설정이 조금씩은
다를 수 있음

DB명	DB 관리 툴
Oracle	SQL Developer
MySQL, MariaDB	Work Bench
PostgreSQL	PgAdmin

```
version: '2'
services:
  oracle21g:
    image: gvenzl/oracle-xe
    container_name: Oracle
    volumes:
      - ${volume_path}/DB/Oracle/Data:/opt/oracle/odata
    restart: always
    ports:
      - 3305:1521
    environment:
      - ORACLE_PASSWORD=${PASSWORD}
```



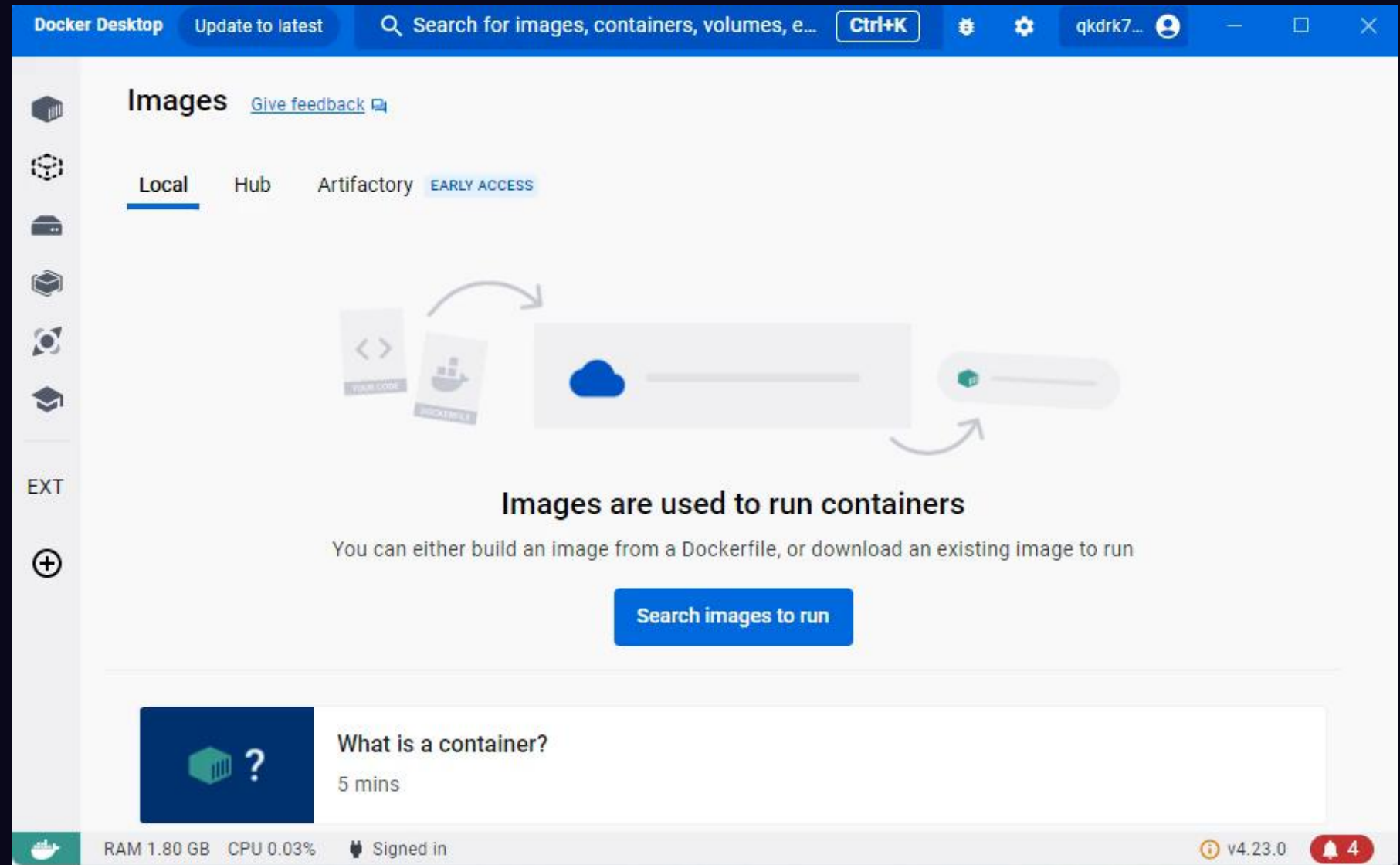
```
version: "3"
services:
  mysql:
    image: mariadb:latest
    restart: always
    container_name: MariaDB
    volumes:
      - ${volume_path}/DB/MariaDB:/var/lib/mysql
    ports:
      - 3307:3306
    environment:
      - MYSQL_ROOT_PASSWORD=${PASSWORD}
```



도커 사용 예시

SQLD 실습 환경

1. 도커 이미지를 컨테이너로 실행
 - 1-1. MariaDB, MySQL, ORACLE, PostgreSQL 컨테이너 실행
2. docker-compose 파일 하나로 관리



도커 사용 예시

웹 서버 세팅

nginx를 활용한 웹 서비스 구축
React 와 같은 JavaScript와
Node.js 통해 웹 서비스를 구축할 때
nginx를 통해 Backend를 로드밸런싱
하거나, SSL인증 과정을 수행할 수 있음



```
worker_processes 1; # default
events {
    worker_connections 1024;
}

http {
    upstream backend_server {
        server localhost:8080;
        # server backend_server2.example.com;
    }
    server {
        listen 80;
        listen [::]:80;

        root /app/build;
        index index.html;

        server_name localhost;

        location / {
            try_files $uri $uri/ /index.html;
        }

        location /api {
            rewrite ^/api/(.*) $1 break;
            proxy_pass http://backend_server;
            # proxy_pass http://localhost:8080;
            proxy_set_header Host $Host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
}
```

version: "3.7"

services:

nginx:

container_name: nginx_test

image: nginx:latest

ipc:host

ports:

- 180:80

- 1443:443

volumes:

- ./nginx.conf:/etc/nginx/nginx.conf:ro

- ./app/build:/app/build

restart: always

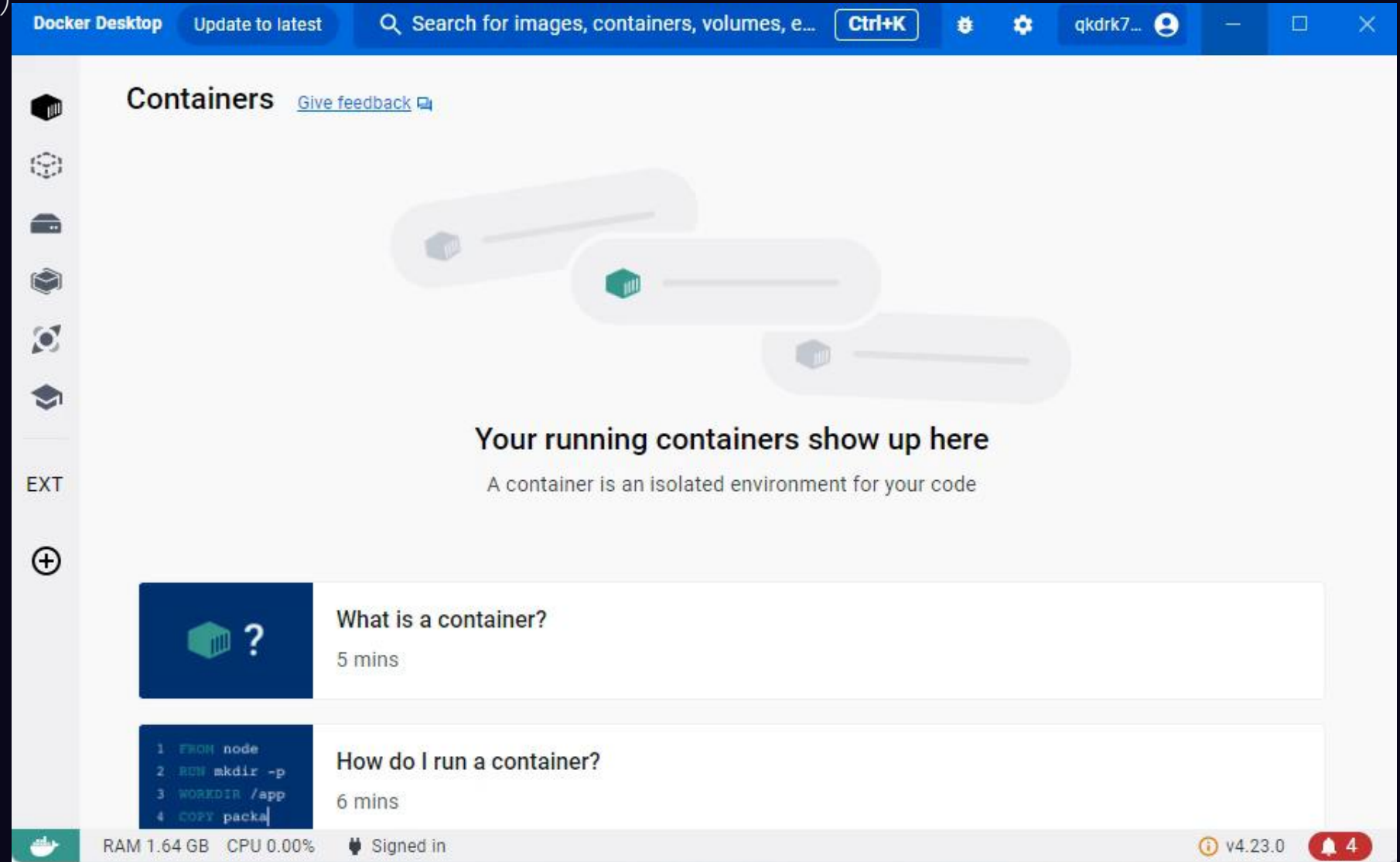
command: ["nginx", "-g", "daemon off;"]

도커 사용 예시

웹 서버 세팅

1. React를 통한 웹앱 빌드(Node 필요)
2. Nginx를 활용한 로드밸런싱 및 프록시 설정 수행

NGINX



도커 사용 예시

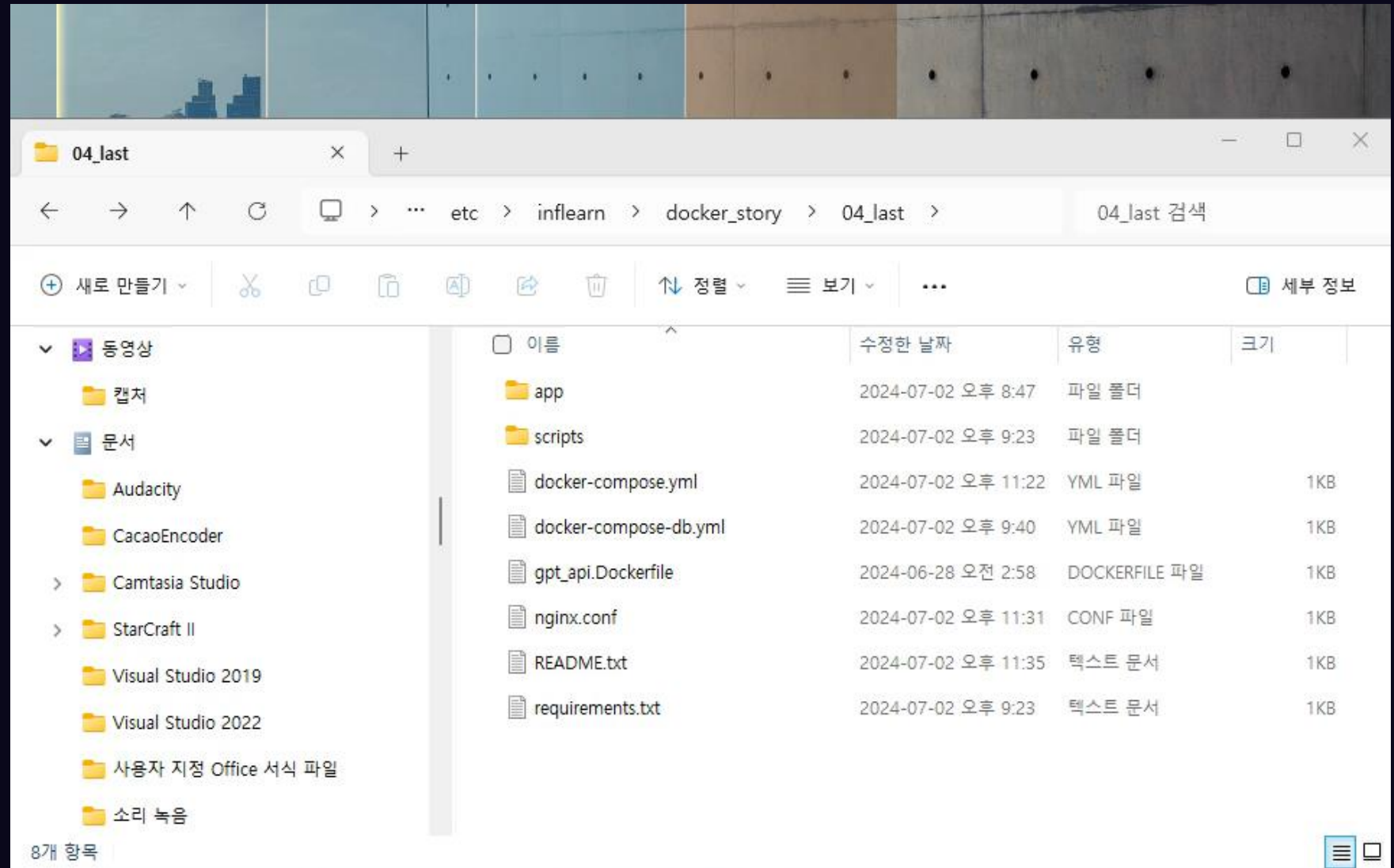
종합 예제

1. GPT API 와 DB, NGINX를 활용하는 예제
2. Work Bench를 통해 MySQL 연결 후 DB를 수동으로 생성
3. API 사용결과를 Comment와 Result로 DB의 Test 라는 Table에 저장
4. Nginx의 로드밸런싱(라운드로빈 방식) (API 요청시 docker-compose 순차적으로 API가 날아감)

[앞 예제에서 변경된 부분]

1. gpt_api.Dockerfile 부분 수정
2. app.py 및 nginx.conf 부분이 앞 예제와 조금 다름

NGINX



빠른 사용을 위한 도커 이야기

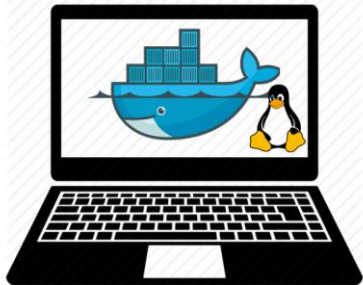
전통적 배포



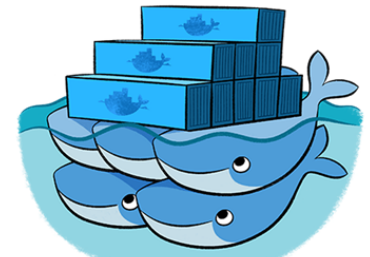
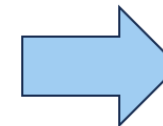
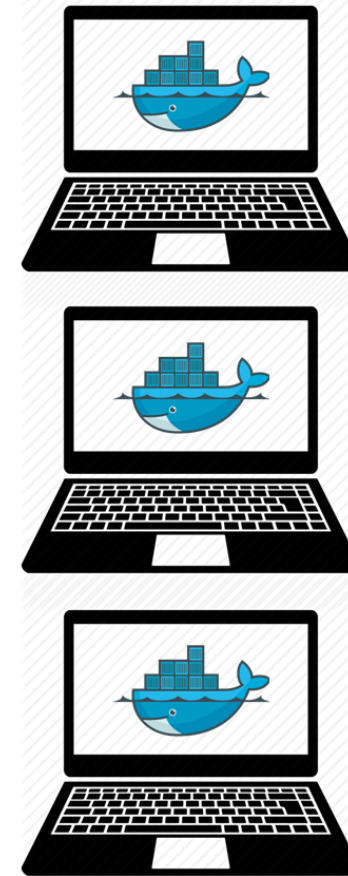
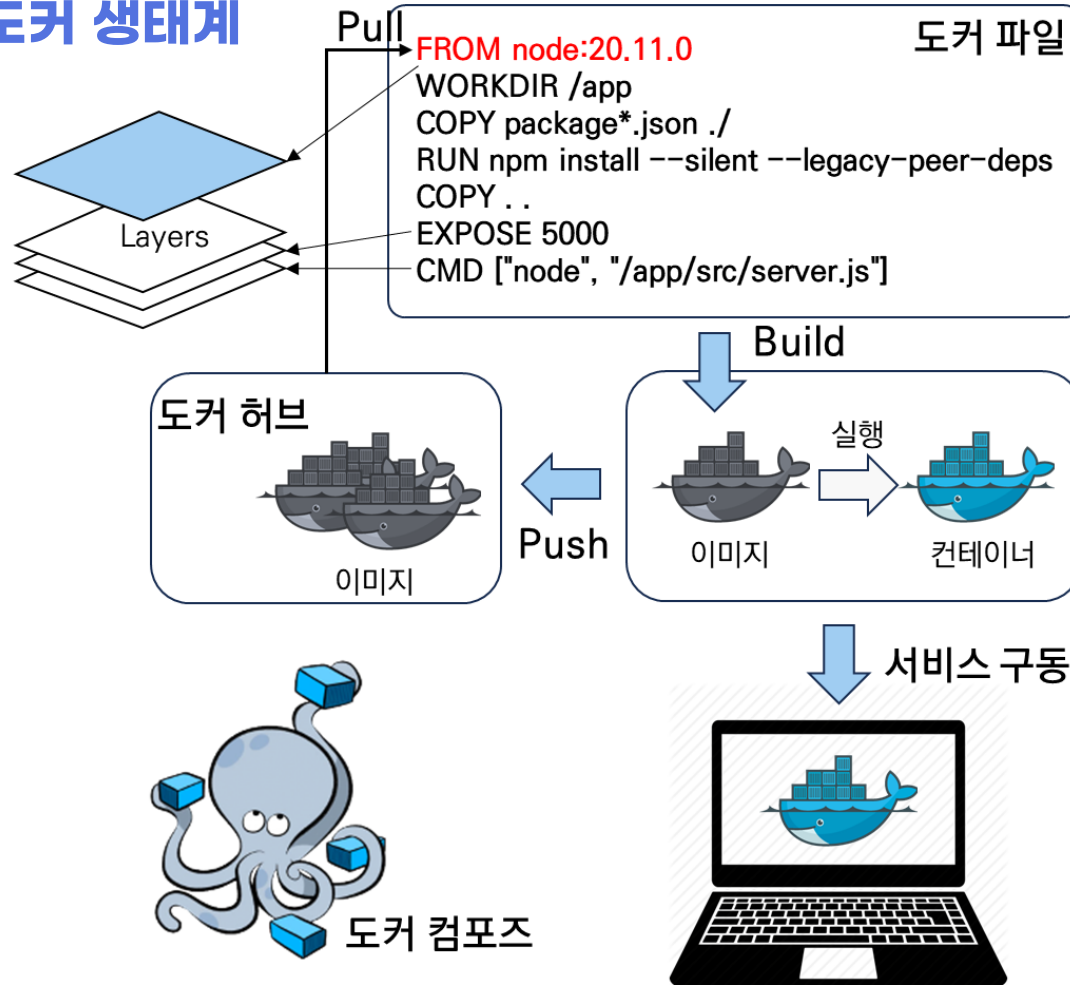
가상머신 기반



컨테이너 기반



도커 생태계



도커 스웜



쿠버네티스

클러스터 및
오케스트레이션