

불편한 점

매번 새 환경으로 많고 복잡한 장비들을 옮겨 새로 조립, 설치해야하는 것

도커란?

장비들을 설치된 그 상태로 복제해 사본을 만들. 하나 하나 복제할 수도 있고 서로 연결되어 조립된 모습으로도 복제 가능함, 문서를 저장하듯 보관할 수 있다는 것

도커의 장점

- 도커허브라는 곳에서 클라우드를 사용하듯 어디서든 내려받아 설치할 수 있음
- 컨테이너라는 곳에 분리된 작업 공간을 여러개 만들어 독립된 업무환경을 구축할 수 있음 (방해x)
- 한 번 저장된 장비들은 얼마든지 새로 불러낼 수 있음
- 문제가 생겨도 고칠 필요없이 새로 복원하면 됨

도커 상세 설명

1. 각 요소들이 설치된 모습을 '이미지'란 형태로 박제해서 저장한다. (각 제품마다 공식적인 이미지가 있고 내가 원하는 대로 만들 수도 있음)
2. 저장된 도커 이미지들은 DockerHub라는 곳에 업로드, 공유/다운 가능
3. 이미지로 저장된 항목들이 함께 연결되어 동작하도록 설정된 상태를 명령어 텍스트나 문서 형태로 저장 가능
4. 도커는 이를 컴퓨터에 바로 설치하지 않고 컨테이너라는 독립된 가상 공간을 만들어 복원함 : ex) 서로 다른 버전의 자바를 돌리는 서비스들도 각각의 컨테이너 안에서 서로 방해받는 일 없이 실행될 수 있음
5. 버추어박스같은 가상 컴퓨팅과는 다른 구조, 가상 컴퓨팅은 한 물리적 컴퓨터 안에 각 OS를 돌리는 가상 컴퓨터들이 물리적 자원을 분할해서 쓰기 때문에 성능에 한계가 있음 but 도커는 실행 환경만 독립적으로 돌리기 때문에 컴퓨터에 직접 요소를 설치한 것과 별 차이 없는 성능을 낼 수 있음
6. 도커를 사용하면 서버 오류를 고쳐야 하거나 일부를 업그레이드해야 하거나 할 때 일일이 요소들을 정지하고 삭제하고 재설치할 필요 없이 컨테이너들을 통째로 교체해서 새로 실행해주면 됨

* 사용자가 웹페이지에 방문해 본인의 이름을 적고 엔터를 치면 방문시간과 함께 표에 기록이 되는 웹페이지가 있다고 가정

* 프론트엔드 : Node.js의 http-server <-> 백엔드 : Flask 프레임워크(Python)
<-> 데이터베이스 : MySQL

* REST API로 데이터를 프론트와 주고 받음

= 서비스A라고 칭함

도커 컨테이너 장점

- 서비스A 이외에 다른 서비스들도 시작되었을 경우 버전이 다르거나 할 때 충돌의 위험이 있을 수 있음. 이를 가상환경을 깔아 해결할 수도 있지만 효율적이지 않은 방법임 (물리적 자원을 나누어 사용하므로 서버 운용에 있어 성능 문제가 생김)

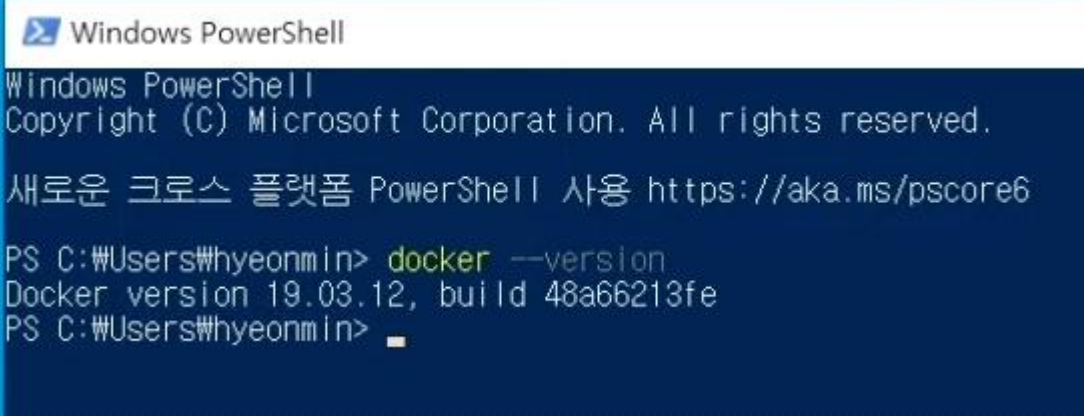
: 도커의 컨테이너를 활용한다면 업무공간이 확실히 분리되면서도 딱 필요한 공간만 활용하기 때문에 효율적

- 또한 서버환경과 개발환경을 동일하게 맞춰야 하는데 크고 복잡한 서비스일수록 번거로운 일임

: 개발컴퓨터에서 작성한 코드와 각각의 컨테이너들이 하는 일을 설계도로 만들어 서버로 보내면 컨테이너를 그대로 설치하여 동일하게 서비스를 실행할 수 있도록 해줌

도커 실습하기

1. 도커 설치 후 버전 확인(윈도우 : 파워셸, 맥 : 터미널)

A screenshot of a Windows PowerShell terminal window. The title bar says "Windows PowerShell". The text inside the terminal reads: "Windows PowerShell", "Copyright (C) Microsoft Corporation. All rights reserved.", "새로운 크로스 플랫폼 PowerShell 사용 https://aka.ms/pscore6", "PS C:\Users\whyonmin> docker --version", "Docker version 19.03.12, build 48a66213fe", "PS C:\Users\whyonmin>".

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

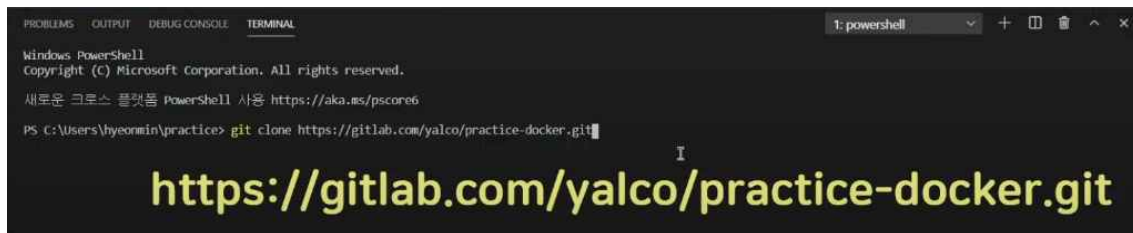
새로운 크로스 플랫폼 PowerShell 사용 https://aka.ms/pscore6

PS C:\Users\whyonmin> docker --version
Docker version 19.03.12, build 48a66213fe
PS C:\Users\whyonmin>
```

2. vs code 등 사용하기 편한 IDE 및 Git 설치하기

3. 원하는 곳에 실습을 진행할 폴더를 만든다 (단, 윈도우라면 로컬 디스크 C-users(사용자)-사용자명으로 된 폴더 또는 그보다 내부에 생성해주도록 함)

4. 이 폴더를 vs code로 열고 터미널을 연 다음 사진과 같이 git clone 뒤에 해당 주소를 붙여서 실행하면 예제로 만들어둔 프로젝트 폴더가 다운받아짐



```
1: powershell
windows PowerShell
Copyright (c) Microsoft Corporation. All rights reserved.

새로운 크로스 플랫폼 PowerShell 사용 https://aka.ms/pscore6

PS C:\Users\hyeonmin\practice> git clone https://gitlab.com/yalco/practice-docker.git
https://gitlab.com/yalco/practice-docker.git
```

** 여기서 만약 node.js와 http-server가 설치되어 있다면, 터미널에서

- cd practice-docker/frontend 명령어를 사용하여 디렉토리 이동
- http-server -p 8080 ./public 명령어를 사용해 웹서버 실행 가능

5. but, 현재는 node.js가 설치되어 있지 않다고 가정, http-server는 물론 기본적인 자바스크립트코드도 실행 할 수 x

- docker run -it node 명령어 사용 (오류가 나면 맨 앞에 sudo 붙여주기), node.js를 설치하지 않고 사용할 수 있는데 이는 도커허브에 node도 공식 이미지가 이미 업로드되어 있기 때문이다. 이 이미지는 컨테이너가 만들어져 설치되면 바로 node 명령어가 실행되도록 설계되어 있어 컨테이너가 만들어지자마자 입력창이 터미널에 뜬

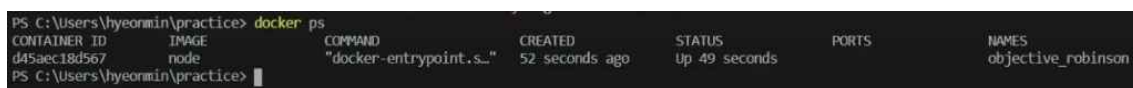
(run 명령으로 컨테이너를 만들고 -it 옵션으로 컨테이너 안에서 CLI를 사용한다는 의미)

6. docker images : 현재 다운받은 이미지들을 볼 수 있는 명령어

앞서 다운받았던 node를 run 명령어로 실행해보면 설치 과정 없이 바로 콘솔이 실행됨

7. docker ps : 현재 작업중인 컨테이너를 확인할 수 있는 명령어

(-a를 붙여 실행하면 작업중이 아닌 컨테이너도 확인할 수 있음 = 모든 컨테이너 확인 가능)



```
PS C:\Users\hyeonmin\practice> docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|-------|--------------------------|----------------|---------------|-------|--------------------|
| d45aec18d567 | node | "docker-entrypoint.s..." | 52 seconds ago | Up 49 seconds | | objective_robinson |

```
PS C:\Users\hyeonmin\practice>
```

8. docker exec -it{컨테이너 이름} bash : 컨테이너 안으로 들어갈 수 있음, 가상의 리눅스 환경으로 들어간다.

- 해당 컨테이너 내에서 bash shell을 실행한다는 의미
- ^C 입력 후 다시 docker ps 입력해보면 컨테이너가 나타나지 않음

9. 한꺼번에 컨테이너 중지하기

```
docker stop $(docker ps -aq)
```

```
docker system prune -a
```

질문에 y

** 어떤 OS에서 도커를 돌리든, 도커의 컨테이너들은 리눅스 가상환경의 형태로 돌아감

** 어느 곳에서든 docker ps 명령어를 통해 컨테이너들을 확인 할 수 있다 = 도커에 의해 다른 곳에서 관리되고 있음을 의미

** 이외 도커 명령어들 : https://www.yalco.kr/36_docker/

10-1. 나만의 이미지 만들기 - node.js

: 다운받은 node 공식 이미지에는 http-server가 포함되어 있지 않기 때문에 http-server를 포함한 나만의 이미지를 만들어 사용한다.

- Dockerfile 작성

```
FROM node:12.18.4

# 이미지 생성 과정에서 실행할 명령어
RUN npm install -g http-server

# 이미지 내에서 명령어를 실행할(현 위치로 잡을) 디렉토리 설정
WORKDIR /home/node/app

# 컨테이너 실행시 실행할 명령어
CMD ["http-server", "-p", "8080", "./public"]
```

- 이미지 생성 명령어 (도커파일과 같은 디렉토리에서)

```
docker build -t {이미지명} .
```

(Dockerfile로의 상대 경로를 적어주는데 파일명이 Dockerfile이면 따로 명시할 필요없음)

- docker images로 이미지 목록들을 보면

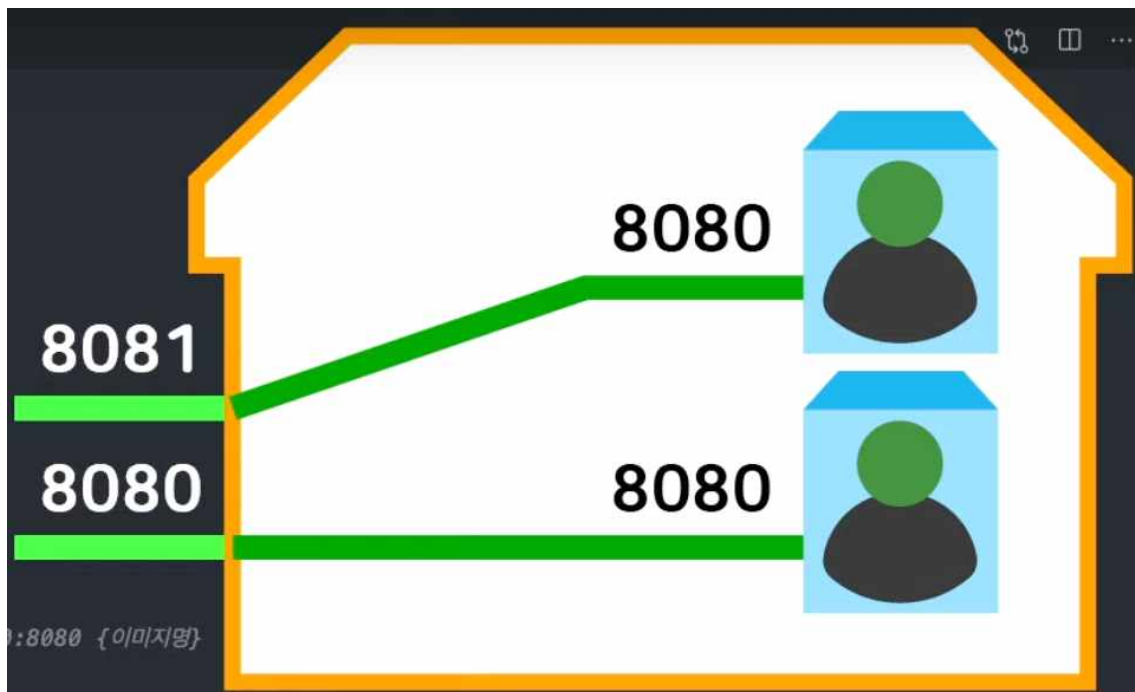
| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|--------------|---------|--------------|--------------------|-------|
| frontent-img | latest | b0b2a7ba88b8 | About a minute ago | 922MB |
| node | 12.18.4 | 28faf336034d | 9 days ago | 918MB |

node 공식이미지와 Dockerfile&build 명령어로 만든 frontent-img (http-server가 깔려있는) 이미지를 볼 수 있음

10-2. 직접 만든 이미지를 컨테이너로 run, 실행해보기

`docker run --name {컨테이너명} -v $(pwd):/home/node/app -p 8080:8080 {이미지명}`

- 아까는 도커가 임의의 컨테이너 명을 지어줬지만 이번에는 생성될 컨테이너의 이름을 지정해준다.
- v 옵션은 volume의 약자로 컨테이너와 특정 폴더를 공유하는 것을 의미, pwd는 현재 위치를 출력하는 명령어이므로 지금 위치한 이 폴더 안의 내용들을 컨테이너의 `home/node/app` 폴더에 넣어준다는 뜻
- p 옵션은 포트, 같은 이미지의 다른 컨테이너 2개일 경우



- 컨테이너 모두 종료 및 삭제

11-1. 나만의 이미지 만들기2 - mysql 5.7버전

- 도커 파일 작성, ENV 명령어로 생성될 컨테이너 안의 환경변수 미리 지정 (실전에서는 보안요소를 이렇게 작성하면 위험하므로 따로 관리되는 파일에 넣거나 해서 안전하게 관리)

```
FROM mysql:5.7
```

```
# 이미지 환경변수들 세팅
```

```
# 실전에서는 비밀번호 등을 이곳에 입력하지 말 것!
```

```
# 서버의 환경변수 등을 활용하세요.
```

```
ENV MYSQL_USER mysql_user
```

```
ENV MYSQL_PASSWORD mysql_password
```

```
ENV MYSQL_ROOT_PASSWORD mysql_root_password
```

```
ENV MYSQL_DATABASE visitlog
```

- mysql 이미지가 컨테이너로 실행될때 scripts 폴더 내 sql 파일들에 적힌 쿼리 명령어들을 실행하도록 설계한다. (sql 파일 내부, 아래 사진 참조)

① 서비스에서 사용할 테이블을 만드는 쿼리

```
practice-docker > database > scripts > 🍌 create_table.sql
1 CREATE TABLE visits (
2   visitor_name varchar(25) not null,
3   visit_datetime datetime default NOW() not null
4 );
```

② 기본적으로 들어갈 데이터들을 넣어주는 쿼리

```
practice-docker > database > scripts > 🍌 insert_data.sql
1 INSERT INTO visits (visitor_name) VALUES ('John');
2 INSERT INTO visits (visitor_name) VALUES ('Mark');
3 INSERT INTO visits (visitor_name) VALUES ('Robin');
```

```
# 도커환경에서 컨테이너 생성시 스크립트를 실행하는 폴더로
# 미리 작성된 스크립트들을 이동
COPY ./scripts/ /docker-entrypoint-initdb.d/
```

** COPY는 RUN처럼 이미지를 생성하는 과정에서 해당 이미지 안에 특정 파일을 미리 넣어두는 것

** Volume은 CMD처럼 컨테이너가 생성되어 실행될 때, 그 내부의 폴더를 외부의 것과 연결하는 것

- 현재 database 디렉토리에 도커 파일을 만들었으므로 터미널에서 `cd ../database`
- 10-1번과 같은 방식으로 이미지 생성 명령어 입력

11-2. run 명령어로 실행하기

- `docker run --name {컨테이너명} -it -p 3306:3306 {이미지명}`
- ** 실전이라면 데이터를 유지해야하므로 데이터 폴더를 `-v` 옵션으로 볼륨 공유
- ** mysql은 3306 포트가 기본 포트
- 이 컨테이너가 돌아가는 동안에는 터미널을 못 쓰기 때문에 새로 열어서 컨테이너 중지 및 삭제
- `d` 옵션을 사용해 안 보이는 곳에서도 알아서 컨테이너를 깔고 돌릴 수 있도록 해 줌
- = `docker run --name {컨테이너명} -p 3306:3306 -d {이미지명}` (`d` 옵션은 daemon의 줄임말)
- 생성된 컨테이너의 ID만 표시되고 터미널은 그대로 사용가능함

```
hyeonm1n ~ /GoogleBackup/ coding/practice/practice-docker/database master > docker run --name database-con -p 3306:3306 -d database-1ng
081e7461ff8de5bb9485adda4334ee4acbb856905b5d58023bcf506eb8d826f5
hyeonm1n ~ /GoogleBackup/ coding/practice/practice-docker/database master > docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
081e7461ff8d   database-1ng   "docker-entrypoint.s..." 17 seconds ago Up 16 seconds  0.0.0.0:3306->3306/tcp, 33060/tcp   database-con
```

- 실시간으로 로그 보기 : `docker logs -f {컨테이너명}`, `^C`로 빠져나올 수 있음

12-1. 나만의 이미지 만들기3 - 파이썬

- 도커 파일 만들기

```
FROM python:3.8.5

# 이미지 생성 과정에서 실행할 명령어
RUN pip3 install flask flask-cors flask-mysql

# 이미지 내에서 명령어를 실행할(현 위치로 잡을) 디렉토리 설정
WORKDIR /usr/src/app

# 컨테이너 실행시 실행할 명령어
CMD ["python3", "backend.py"]
```

- 위와 같은 방식으로 이미지 생성, 컨테이너 생성 및 실행해보기

13. 이 요소들을 연결해서 서비스를 간편하게 실행할 수는 없는지?

하나하나 run으로 실행하기 번거로움

- docker-compose 사용 : 거시적 설계도
- 실습을 위해 모든 컨테이너와 이미지들을 삭제
- docker-compose.yml 파일 살펴보기


```
version: '3'
services:
  database:
    # Dockerfile이 있는 위치
    build: ./database
    # 내부에서 개방할 포트 : 외부에서 접근할 포트
    ports:
      - "3306:3306"
  backend:
    build: ./backend
    # 연결할 외부 디렉토리 : 컨테이너 내 디렉토리
    volumes:
      - ./backend:/usr/src/app
    ports:
      - "5000:5000"
    # 환경변수 설정
    environment:
      - DBHOST=database
```

```
frontend:
  build: ./frontend
  # 연결할 외부 디렉토리 : 컨테이너 내 디렉토리
  volumes:
    - ./frontend:/home/node/app
  ports:
    - "8080:8080"
```

** 버전은 도커 버전마다 적합한 컴포즈 버전 확인해서 작성

** docker-compose 파일에서는 서비스를 실행하는 데이터베이스, 백엔드, 프론트엔드가 services란 항목의 내부 항목들로 들어간다

** run 명령어를 실행할 때마다 줘야 했던 옵션들을 문서에 미리 작성해둘 수 있는 것

** 환경변수도 컴포즈에서 설정해줄 수 있음

- docker-compose 파일이 있는 위치에서 docker-compose up 명령어 실행

- 컴포즈도 d 옵션을 사용하여 이미지 생성 후 뒤에서 알아서 일하도록 할 수 있다 (docker-compose up -d)