7/9

Docker 볼륨 & 네트워크

프론트랑 백엔드 나눴을 때 서버가 굉장히 여러 개가 있고 db도 여러 개임

그럴 때 프론트 서버 컨테이너 1 백 서버 컨테이너 2 db 서버 컨테이너 3

이렇게 각 컨테이너 사이의 통신이 필요

백과 db만 통신 가능하게끔 보안그룹 백과 프론트랑만 통신하면 됨

프론트는 db랑 통신 필요 x 그렇게 서브네팅하면 된다.

volumn은 파일을 공유할 때

ec2 백엔드 서버 안에 여러 개의 컨테이너 작동 log 수집기, api서버, 매트릭 수집기, 이렇게 여러 컨테이너 사용 이런 수집기도 이미지로 배포가 가능하다.

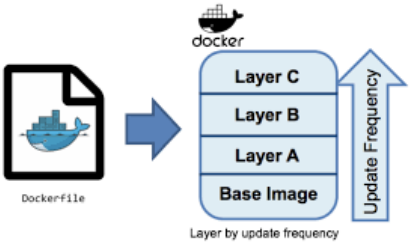
host를 매개로 api 서버 로그 디렉토리랑 log 주입기랑 각각 마운트해서 파일 공유 가능

도커에 volumn이라는 컴포넌트가 있고 각각의 컨테이너가 거길 마운트해서 파일 공유

db는 stateful하기에 컨테이너 잘 사용하지 않음 컨테이너가 삭제되면 다 날라가니까

그런데 vol이랑 마운트 되어있으면 dr마냥 다 저장이 되어 있기에 컨테이너 재생성해도 사용이 가능하다.

볼륨



볼륨이 서버1 콘테이너에 마운트를 한다. 서버2가 볼륨에 마운트할 경우 서버2도 볼륨을 통해 서버1 컨테이너랑 마운트 된 곳에 접근 가능.

서버 1이 볼륨에 로그 쓰고 서버2가 볼륨에 저장된 로그 가져오기

1.5. Docker 볼륨

```
Usage:  docker volume COMMAND

Manage volumes

Commands:
  create      Create a volume
  inspect     Display detailed information on one or more volumes
  ls          List volumes
  prune       Remove all unused local volumes
  rm          Remove one or more volumes
```

- 컨테이너에서 사용 및 관리하는 저장 공간입니다.
- 컨테이너에 종속되어 있지 않으므로, 컨테이너의 라이프 사이클을 따라가지 않습니다.

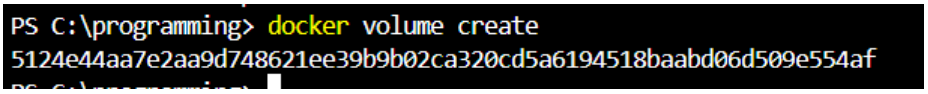
볼륨 생성

> https://docs.docker.com/engine/reference/commandline/volume_create/

```
docker volume create [OPTIONS] [VOLUME]
```

주요 옵션

Option	Short	Default	Description
`--name`			이름을 지정합니다.



이렇게 해시값만 나오기에 구분 힘들다 - - name옵션을 필수적으로 해야..

볼륨 목록 보기

> https://docs.docker.com/engine/reference/commandline/volume_ls/

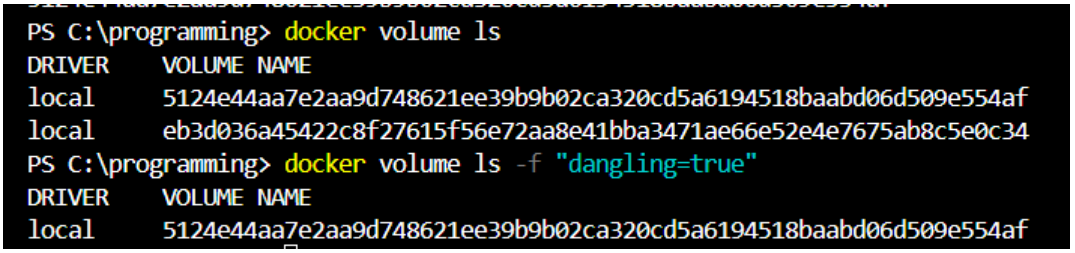
```
docker volume ls [OPTIONS]
```

주요 옵션

Option	Short	Default	Description
`--filter`	`-f`		지정된 조건에 맞는 볼륨만 표시합니다.
`--quiet`	`-q`		볼륨 이름만 표시합니다.

[예시] 사용하지 않는 볼륨 목록 보기

```
$ docker volume ls -f "dangling=true"  # or "dangling=1"
DRIVER    VOLUME NAME
local     fecea4ffd139a0d85b735f9ea8fe247bfabc9227df31ac9ff1d8e66f6b77d229
```



볼륨 정보 보기

> https://docs.docker.com/engine/reference/commandline/volume_inspect/

```
docker volume inspect [OPTIONS] VOLUME [VOLUME...]
```

[예시] 볼륨 생성 일자 확인하기

```
$ docker volume inspect --format "{{ .CreatedAt }}" <VOLUME_NAME>
2023-12-22T01:23:07Z
```

볼륨 삭제

> https://docs.docker.com/engine/reference/commandline/volume_rm/

```
docker volume rm [OPTIONS] VOLUME [VOLUME...]
```

`volume rm`은 한 개 이상의 명시된 볼륨들을 삭제하는 명령입니다.

모든 볼륨 삭제

> https://docs.docker.com/engine/reference/commandline/volume_prune/

```
docker volume prune [OPTIONS]
```

`volume prune`은 사용하지 않는 모든 `anonymous` 볼륨을 삭제하는 명령어 입니다.

주요 옵션

Option	Short	Default	Description
`--all`	`-a`		사용하지 않는 모든 볼륨을 삭제합니다.
`--filter`			필터링 조건을 설정합니다.

컨테이너 실행하기

> https://docs.docker.com/engine/reference/commandline/run/

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

주요 옵션

Option	Short	Default	Description
`--name`			컨테이너의 이름을 지정합니다.
`--detach`	`-d`		컨테이너를 백그라운드에서 실행합니다.
`--env`	`-e`		환경 변수를 설정합니다.
`--env-file`			환경 변수를 저장한 파일을 설정합니다.
`--expose`			포트 또는 포트 범위를 노출합니다.
`--publish`	`-p`		컨테이너의 포트를 공개합니다.
`--rm`			컨테이너가 종료되면 자동으로 삭제합니다.
`--interactive`	`-i`		`STDIN` 을 활성화합니다.
`--tty`	`-t`		`pseudo-TTY` 를 할당합니다.
`--volume`	`-v`		볼륨을 설정합니다.

-v 볼륨 디렉토리 : 마운트할 컨테이너 디렉토리

연습1 DB를 사용하여 VOLUME 작동 확인

Practice

[연습] Volume에 DB 데이터 저장하기

> - `postgres:16.1-bullseye` 이미지를 사용합니다.
> - 컨테이너의 이름은 `psql_db`로 설정합니다.
> - 컨테이너 생성시 다음 환경변수가 설정되어야 합니다.
> - POSTGRES_PASSWORD
> - 생성한 Volume은 컨테이너의 `/var/lib/postgresql/data`에 마운트합니다.

`DB`에서 사용할 볼륨을 다음과 같이 생성합니다.

```
$ docker volume create db_data
db_data
```

생성된 볼륨은 `docker volume list` 명령어를 통해서 확인할 수 있습니다.

```
$ docker volume list
DRIVER    VOLUME NAME
local     db_data
```

`PostgreSQL`의 데이터는 `/var/lib/postgresql/data`에 저장됩니다.
그러므로, `db_data` 볼륨을 해당 디렉토리에 마운트합니다.

```
$ docker run --rm -d --name psql_db -v db_data:/var/lib/postgresql/data -e
POSTGRES_PASSWORD=mysecretpassword postgres:16.1-bullseye
```

컨테이너 터미널에 접근하여 `psql`을 이용해 DB에 접속합니다.

```
$ docker exec -it psql_db /bin/bash
root@7523c983f729:/# psql -U postgres
psql (16.1 (Debian 16.1-1.pgdg110+1))
Type "help" for help.
```

SQL을 이용하여 테이블을 생성합니다.

```
# SQL
CREATE TABLE IF NOT EXISTS cloud_wave (
    id SERIAL PRIMARY KEY,
    timestamp timestamp
);
```

생성한 테이블을 `\dt`를 이용하여 확인합니다.

```
postgres=# \dt
              List of relations
 Schema |    Name    | Type  |  Owner
--------+------------+-------+----------
 public | cloud_wave | table | postgres
(1 row)
```

DB를 종료합니다.

```
$ docker stop psql_db
psql_db

# 컨테이너 이름에 `psql_db`가 포함된 컨테이너 목록을 보여줍니다.
$ docker ps -a -f name=psql_db
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

컨테이너를 재생성 합니다.

```
$ docker run --rm -d --name psql_db -v db_data:/var/lib/postgresql/data -e
POSTGRES_PASSWORD=mysecretpassword postgres:16.1-bullseye
```

컨테이너 내부에서 DB 테이블이 남아있는 것을 확인합니다.

```
$ docker exec -it psql_db /bin/bash
root@3f19c9efb04b:/# psql -U postgres
psql (16.1 (Debian 16.1-1.pgdg110+1))
Type "help" for help.

# Table 목록
postgres=# \dt
              List of relations
 Schema |    Name    | Type  |  Owner
--------+------------+-------+----------
 public | cloud_wave | table | postgres
(1 row)
```

연습2

[연습] `bind mount`를 사용하여 소스코드 변경하기

> - Host의 `./app`는 컨테이너의 `/code/app`와 바인드 되어야 합니다.
> - 이미지 이름은 `was`로 설정합니다.

폴더 구조는 다음과 같습니다.

```
.
├── app
│   ├── __init__.py
│   └── main.py
├── Dockerfile
└── requirements.txt
```

main.py

```
from typing import Union

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}
```

Dockerfile

```
FROM python:3.9

WORKDIR /code

COPY ./requirements.txt /code/requirements.txt

RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80", "--reload"]
```

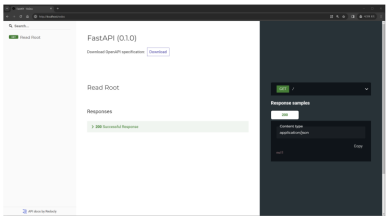
`Dockerfile`이 위치한 폴더에서 다음 명령어를 실행하여 이미지를 빌드합니다.

```
$ docker build -t was:fast.1 .
```

다음 명령어를 통해서 `FastAPI` 서버를 실행합니다.

```
$ docker run --name bind -p 80:80 -v <FOLDER_PATH>\app:/code/app was:fast.1
INFO:     Will watch for changes in these directories: ['/code']
INFO:     Uvicorn running on http://0.0.0.0:80 (Press CTRL+C to quit)
INFO:     Started reloader process [1] using statreload
INFO:     Started server process [8]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

`http://localhost/redoc`에 접속하여 API 목록을 확인합니다.

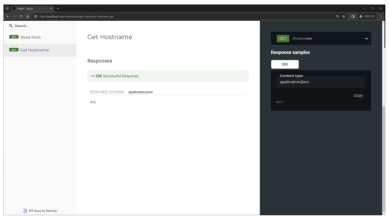


File 업데이트 하기

`main.py`를 다음과 같이 수정합니다.

```
import socket
from typing import Union

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/hostname")
def get_hostname():
    return {"name": socket.gethostname()}
```

`http://localhost/redoc`에 접속하면 `get_hostname` API가 추가된 것을 확인할 수 있습니다.



바인드를 통해서 호스트 디렉토리랑 컨테이너 디렉토리가 바인드 되어서 호스트에서 수정한 파일이 컨테이너에도 즉각즉각 반영이 되는 것

```
requirement.txt
안에는
fastapi
```

인터프리터 언어에서만 이런 작업이 가능하고 자바 스프링은 빌드를 해야하기 때문에 이런 작업이 힘들다

<folder_path> ⇒ .\

네트워크

1.6. Docker 네트워크

```
Usage:  docker network COMMAND

Manage networks

Commands:
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one or more networks
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks
```

네트워크 생성

> https://docs.docker.com/engine/reference/commandline/network_create/

```
docker network create [OPTIONS] NETWORK
```

주요 옵션

Option	Short	Default	Description
`--driver`	`-d`	`bridge`	네트워크에서 사용할 드라이버를 선언합니다.
`--label`			네트워크의 라벨을 설정합니다.

네트워크 드라이버

- `bridge`

동일한 `Host` 컴퓨터 내에서 컨테이너끼리 통신하기 위해 사용합니다.

- `host`

컨테이너가 `Host`와 동일한 네트워크를 사용합니다.

- `none`

`Host`와 완벽히 격리된 네트워크입니다.

주의사항

- `host` 네트워크는 인스턴스 별로 한 개만 생성할 수 있습니다.

[예시] `bridge` 네트워크 생성하기

```
$ docker network create -d bridge private
```

브릿지 = 컨테이너끼리 사설 ip로 묶는다 따라서 호스트에서 컨테이너로 전송은 안된다.

호스트 = 로컬에서 컨테이너로 마음대로 전송 접속 가능

호스트랑 컨테이너가 네트워크가 묶어진 것이기에 둘이 같은 포트로 작업 불가능하다.

none = ip가 할당이 안된 수준 접근 불가 통신 안하겠다

```
PS C:\programming\test4> docker network create -d bridge private
cd3a6fb9794c6ba972a66871c4f26bc158591eccc6fe34bc1ae59301be58c54d
PS C:\programming\test4> docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
a32dab3c0b5b   bridge    bridge    local
c007d12986a4   host      host      local
74a194193d67   none      null      local
cd3a6fb9794c   private   bridge    local
PS C:\programming\test4>
```

네트워크 목록 보기

> https://docs.docker.com/engine/reference/commandline/network_ls/

```
docker network ls [OPTIONS]

Aliases:
  docker network ls, docker network list
```

주요 옵션

Option	Short	Default	Description
`--filter`	`-f`		지정된 조건에 맞는 네트워크만 표시합니다.
`--quiet`	`-q`		네트워크 ID만 표기합니다.

[예시] `host` 네트워크만 조회하기

```
$ docker network ls -f driver=host
NETWORK ID     NAME      DRIVER    SCOPE
b107e82764b5   host      host      local
```

네트워크 연결

> https://docs.docker.com/engine/reference/commandline/network_connect/

```
docker network connect [OPTIONS] NETWORK CONTAINER
```

주요 옵션

Option	Short	Default	Description
`--alias`			네트워크 alias를 추가합니다.
`--ip`			IPv4 주소를 지정합니다.

네트워크 연결 제거

> https://docs.docker.com/engine/reference/commandline/network_disconnect/

```
docker network disconnect [OPTIONS] NETWORK CONTAINER
```

네트워크 삭제

> https://docs.docker.com/engine/reference/commandline/network_rm/

```
docker network rm NETWORK [NETWORK...]
```

`network rm`은 한 개 이상의 명시된 네트워크들을 삭제하는 명령입니다.

모든 네트워크 삭제

> https://docs.docker.com/engine/reference/commandline/network_prune/

```
docker network prune [OPTIONS]
```

`network prune`은 **사용하지 않는** 모든 네트워크들을 삭제하는 명령어 입니다.

주요 옵션

Option	Short	Default	Description
`--filter`			필터링 조건을 설정합니다.

연습1

연습 문제

[연습] bridge 네트워크를 이용하여 컨테이너 연결하기

사용할 네트워크를 다음과 같이 생성합니다.

```
$ docker network -d bridge create private
private
```

생성된 네트워크는 `docker network list` 명령어를 통해서 확인할 수 있습니다.

```
$ docker network list
NETWORK ID     NAME      DRIVER    SCOPE
72bf83fc2d7c   bridge    bridge    local
01c55e1a1560   host      host      local
ceed0373ae73   none      null      local
5bce335632dc   private   bridge    local
```

`Postgresql` 과 `pgadmin` 컨테이너를 생성합니다. 이 때, `pgadmin` 컨테이너는 위에서 생성한 네트워크를 사용하여 생성합니다.

```
# Postgresql DB 생성
$ docker run --rm -d --name db -e POSTGRES_PASSWORD=mysecretpassword
postgres:16.1-bullseye

# PgAdmin Application 생성
$ docker run --rm -d -p 80:80 --name pgadmin -e
PGADMIN_DEFAULT_EMAIL=user@sample.com -e PGADMIN_DEFAULT_PASSWORD=SuperSecret --
network private dpage/pgadmin4:7.4
```

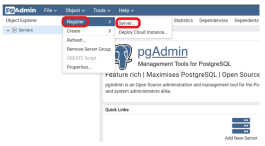
다음 명령어를 통해서 `db` 컨테이너의 `network` 별 IP 주소를 확인합니다.

```
$ docker inspect db -f '{{range $k, $v := .NetworkSettings.Networks}}{{print
$k}}={{println $v.IPAddress}}{{end}}'
bridge=172.17.0.3
```

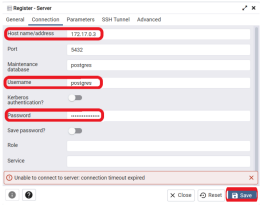
`bridge` 네트워크에 할당된 `IP`를 이용하여 `PgAdmin`에서 연결을 시도합니다.

- `PgAdmin` 페이지에 접속합니다.
  - http://localhost:80

- `Servers`를 클릭한 다음, 좌측 상단 `Object` > `Register` > `Server`를 클릭합니다.



- `General` 탭에서 `Name`을 기입합니다.
- `Connection` 탭에서 접속할 `DB`서버 정보를 입력합니다.
  - Host: `bridge` 네트워크에서 할당된 IP
  - Username: `postgres`
  - Password: 컨테이너 실행시 입력한 `POSTGRES_PASSWORD`

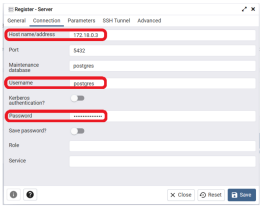


- `DB` 연결에 실패한 것을 확인할 수 있습니다.

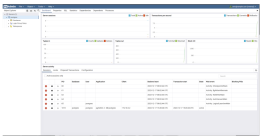
이번엔 위에서 생성한 `private` 네트워크를 `DB`에 연결한 이후, `private` 네트워크에서 할당된 `IP` 주소를 확인합니다.

```
$ docker network connect private db

$ docker inspect db -f '{{range $k, $v := .NetworkSettings.Networks}}{{print
$k}}={{println $v.IPAddress}}{{end}}'
bridge=172.17.0.3
private=172.18.0.3
```

해당 `IP`를 이용하여 `PgAdmin`에서 연결의 시도합니다.



정상적으로 연결이 되는 것을 확인할 수 있습니다.



연습2

[연습] `alias`를 이용하여 `ip`없이 컨테이너 통신하기

다음과 같이 `ubuntu` 컨테이너를 생성하고 필요한 `Package`를 생성합니다.

```
$ docker run --name main -itd ubuntu:22.04
$ docker exec main /bin/bash -c "apt-get update && apt-get upgrade && apt-get
install -y wget dnsutils"
```

`nginx` 컨테이너를 다음과 같이 3개 생성합니다.

```
$ docker run --rm -d --net private --net-alias web_app --name nginx1 nginx:latest
$ docker run --rm -d --net private --net-alias web_app --name nginx2 nginx:latest
$ docker run --rm -d --net private --net-alias web_app --net-alias ready --name
nginx3 nginx:latest
```

`main` 컨테이너에서 `web_app`에 대해 `dig`을 사용하면, 다음과 같이 `DNS` 질의에 대한 응답이 없는 것을 확인할 수 있습니다.

> `nslookup web_app`을 이용하여 확인해도 됩니다.

```
$ docker exec main dig web_app

; <<>> DiG 9.18.18-0ubuntu0.22.04.1-Ubuntu <<>> web_app
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 53782
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: bcdc7ef17f1d57fd (echoed)
;; QUESTION SECTION:
```

```
;web_app.           IN  A

;; Query time: 5 msec
;; SERVER: 127.0.0.11#53(127.0.0.11) (UDP)
;; WHEN: Sat Dec 23 07:59:12 UTC 2023
;; MSG SIZE  rcvd: 48
```

`main` 컨테이너에 `private` 네트워크를 다음과 같이 연결합니다.

```
$ docker network connect --alias main private main
$ docker inspect main -f "Alias:{{ println
.NetworkSettings.Networks.private.Aliases }}IP:{{ println
.NetworkSettings.Networks.private.IPAddress }}"
Alias:[main 51c471535b96]
IP:172.19.0.5
```

다시 한번 `main` 서버에서 `dig`를 사용하면 3개 컨테이너의 `IP`가 반환된 것을 확인할 수 있습니다.

```
$ docker exec main dig web_app

; <<>> DiG 9.18.18-0ubuntu0.22.04.1-Ubuntu <<>> web_app
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 43503
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;web_app.           IN  A

;; ANSWER SECTION:
web_app.        600 IN  A   172.19.0.4
web_app.        600 IN  A   172.19.0.2
web_app.        600 IN  A   172.19.0.3

;; Query time: 0 msec
;; SERVER: 127.0.0.11#53(127.0.0.11) (UDP)
;; WHEN: Sat Dec 23 07:51:11 UTC 2023
;; MSG SIZE  rcvd: 94
```

다음과 같이 `ping`을 사용할 때마다 응답하는 `IP`가 바뀌는 것을 볼 수 있습니다.

```
$ docker exec main apt-get install -y iputils-ping
$ docker exec main ping web_app
PING web_app (172.19.0.2) 56(84) bytes of data.
64 bytes from nginx1.private (172.19.0.2): icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from nginx1.private (172.19.0.2): icmp_seq=2 ttl=64 time=0.075 ms
64 bytes from nginx1.private (172.19.0.2): icmp_seq=3 ttl=64 time=0.076 ms
...
$ docker exec main ping web_app
PING web_app (172.19.0.4) 56(84) bytes of data.
64 bytes from nginx3.private (172.19.0.4): icmp_seq=1 ttl=64 time=0.136 ms
64 bytes from nginx3.private (172.19.0.4): icmp_seq=2 ttl=64 time=0.073 ms
...
```

,

```
root@a85f93f19ce4:/# ping web_app
PING web_app (172.18.0.4) 56(84) bytes of data.
64 bytes from nginx1.private (172.18.0.4): icmp_seq=1 ttl=64 time=0.029 m
s
64 bytes from nginx1.private (172.18.0.4): icmp_seq=2 ttl=64 time=0.105 m
s
^C
--- web_app ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1050ms
rtt min/avg/max/mdev = 0.029/0.067/0.105/0.038 ms
root@a85f93f19ce4:/# ping web_app
PING web_app (172.18.0.6) 56(84) bytes of data.
64 bytes from nginx3.private (172.18.0.6): icmp_seq=1 ttl=64 time=0.095 m
s
64 bytes from nginx3.private (172.18.0.6): icmp_seq=2 ttl=64 time=0.099 m
s
64 bytes from nginx3.private (172.18.0.6): icmp_seq=3 ttl=64 time=0.130 m
s
^C
--- web_app ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2040ms
rtt min/avg/max/mdev = 0.095/0.108/0.130/0.015 ms
root@a85f93f19ce4:/# ping web_app
PING web_app (172.18.0.5) 56(84) bytes of data.
64 bytes from nginx2.private (172.18.0.5): icmp_seq=1 ttl=64 time=0.081 m
s
64 bytes from nginx2.private (172.18.0.5): icmp_seq=2 ttl=64 time=0.077 m
```

바뀌는 거 확인 가능하다

```
root@a85f93f19ce4:/# nslookup web_app
Server:         127.0.0.11
Address:        127.0.0.11#53

Non-authoritative answer:
Name:   web_app
Address: 172.18.0.5
Name:   web_app
Address: 172.18.0.4
Name:   web_app
Address: 172.18.0.6
```

실습 1

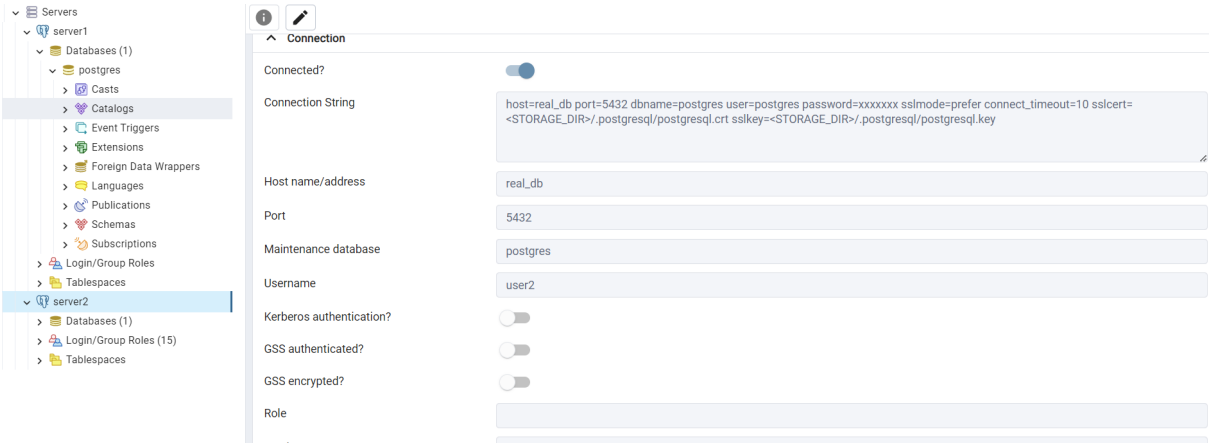
[실습] DB를 alias를 이용하여 연결하기

> 연습 문제(bridge 네트워크를 이용하여 컨테이너 연결하기)에서 IP 대신 DNS를 이용하세요

- PostgreSQL과 PgAdmin 컨테이너를 생성합니다
- private 네트워크를 DB에 연결하면서 Alias를 설정합니다.
- PgAdmin에서 IP 대신 Alias를 이용하여 DB에 연결합니다.

서버 레지스터 할 때 ip가 아닌 alias를 넣어보자

```
PS C:\programming\test4> docker run --rm -d --net private --net-alias rea
l_db --name real_db -e POSTGRES_PASSWORD=mysecretpassword postgres:16.1-b
ullseye
da463e67606174272a2d9077a9912a86638c92631b6789116ae7985478261bc3
```



Docker commit

1.7. Docker Advance - Image

Docker commit

> https://docs.docker.com/engine/reference/commandline/commit/

```
docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

주요 옵션

Option	Short	Default	Description
`--author`	`-a`		`Author`를 설정합니다.
`--change`	`-c`		추가로 적용할 `Dockerfile` 명령어를 설정합니다.
`--message`	`-m`		Commit message
`--pause`	`-p`	`true`	`commit` 동안 컨테이너를 중지합니다.

주의사항

- `Production`에서 사용할 이미지라면 `commit` 대신 `Dockerfile`을 기반으로 제작하는 것이 좋습니다.
- 볼륨(`volume`)에 저장된 데이터는 포함되지 않습니다.
- `--pause` 옵션을 설정하지 않은 경우, `commit`하는 동안 컨테이너를 중지(`pause`)됩니다.
- `--change`에서 지원하는 명령어는 다음과 같습니다
  - CMD
  - ENTRYPOINT
  - ENV
  - EXPOSE
  - LABEL
  - ONBUILD
  - USER
  - VOLUME
  - WORKDIR

일종의 스냅샷이다. 어떠한 이미지에 변경을 가했을 때 그 변경까지 저장하며 다른 이미지로 기록하고 싶을 때 commit을 한다.

연습1

x

연습 문제

[연습] Commit을 이용하여 패키지가 추가로 설치된 이미지 생성하기

다음과 같이 `Ubuntu` 컨테이너를 실행합니다.

```
$ docker run -it -d --name base ubuntu:22.04
ebb9fe13e5cfb0747e5bea1db7c41ef30bf416bdcd643d7311e161ee4300b628
```

`curl` 설치 여부를 확인합니다.

```
$ docker exec base apt list --installed "curl*"
WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

Listing...
```

다음과 같이 `Curl` 을 설치한 후, 설치 여부를 재확인합니다.

```
$ docker exec base /bin/bash -c "apt-get update && apt-get upgrade && apt-get
install -y curl"

$ docker exec base apt list --installed "curl*"

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

Listing...
curl/jammy-updates,jammy-security,now 7.81.0-1ubuntu1.15 amd64 [installed]
```

`commit` 을 이용하여 `base` 컨테이너를 다음과 같이 저장합니다.

```
$ docker commit base commit:v1
sha256:56c923d569eccda8bd094286ca7356ea2fa1a3a7794df6f22369980dd78bc943

$ docker images commit
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
commit       v1        56c923d569ec   16 seconds ago   132MB
```

저장된 이미지를 실행하여 `curl` 이 설치되어 있는지 확인합니다.

```
$ docker run --name restore commit:v1 apt list --installed "curl*"

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

Listing...
curl/jammy-updates,jammy-security,now 7.81.0-1ubuntu1.15 amd64 [installed]
```

[연습] 실행중인 컨테이너의 Port를 추가로 `Expose` 하기

컨테이너를 다음과 같이 생성하고, `Port`가 노출되지 않은 것을 체크합니다다.

```
$ docker run -it -d --name base ubuntu:22.04
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
  PORTS     NAMES
51c471535b96   ubuntu:22.04   "/bin/bash"              2 hours ago    Up 2 hours
          base
```

`--change` 옵션을 이용하여 80번 포트를 `Expose`합니다.

이 때, 기존에 실행중인 컨테이너가 멈추지 않도록 `--pause`를 `false`로 설정합니다.

```
$ docker commit --change="EXPOSE 80" --pause=false base commit:v1
sha256:f908d82f79101ec792d5383a627c8ffa9dd92d65bf5674cf0ad8e9c0c7c943b2
```

저장한 이미지를 사용하여 새로운 컨테이너를 생성하고, 80번 포트가 열려있는 것을 확인합니다.

```
$ docker run -itd commit:v1
c551f14323f9bc6dccf23b0a68fb9c4610e73079e7d76fe09306ed35aa8c0f57

$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED
 STATUS              PORTS     NAMES
c551f14323f9   commit:v1      "/bin/bash"              About a minute ago   Up
About a minute   80/tcp    commit
51c471535b96   ubuntu:22.04   "/bin/bash"              2 hours ago          Up 2
hours                      base
```

`inspect`를 사용하면 다음과 같이 `commit:v1` 이미지에 새로운 레이어가 추가되어 있는 것을 확인할 수 있습니다.

> `jq`가 설치되어 있지 않은 경우, `Appendix`를 참고하여 설치해주세요.

```
$ docker inspect ubuntu:22.04 | jq ".[0].RootFS.Layers"
[
  "sha256:a1360aae5271bbbf575b4057cb4158dbdfbcae76698189b55fb1039bc0207400"
]
$ docker inspect commit:v1 | jq ".[0].RootFS.Layers"
[
  "sha256:a1360aae5271bbbf575b4057cb4158dbdfbcae76698189b55fb1039bc0207400",
  "sha256:36005e181ab5ff954b4d4191c6a3f6a69a62cf2d3367c53e6889ee2d14757c44"
]
```

Docker build x

Docker buildx

> https://docs.docker.com/engine/reference/commandline/buildx/

```
Usage:  docker buildx [OPTIONS] COMMAND

Extended build capabilities with BuildKit

Options:
      --builder string   Override the configured builder instance

Management Commands:
  imagetools  Commands to work on images in registry

Commands:
  bake        Build from a file
  build       Start a build
  create      Create a new builder instance
  du          Disk usage
  inspect     Inspect current builder instance
  ls          List builder instances
  prune       Remove build cache
  rm          Remove a builder instance
  stop        Stop builder instance
  use         Set the current builder instance
  version     Show buildx version information

Run 'docker buildx COMMAND --help' for more information on a command.
```

doucker buildx build <> ⇒ builder 를 사용

builder 목록 확인하기

> https://docs.docker.com/engine/reference/commandline/buildx_ls/

```
docker buildx ls
```

[예시] 현재 사용중인 `builder` 확인하기

현재 선택된 `builder`는 `*`이 붙어있습니다.

```
$ docker buildx ls
NAME/NODE       DRIVER/ENDPOINT STATUS  BUILDKIT             PLATFORMS
default *       docker
  default       default         running v0.11.6+616c3f613b54 linux/amd64,
linux/amd64/v2, linux/amd64/v3, linux/arm64, linux/riscv64, linux/ppc64le,
linux/s390x, linux/386, linux/mips64le, linux/mips64, linux/arm/v7, linux/arm/v6
```

builder 생성하기

> https://docs.docker.com/engine/reference/commandline/buildx_create/

```
docker buildx create [OPTIONS] [CONTEXT|ENDPOINT]
```

Option	Short	Default	Description
`--driver`			사용할 드라이버를 설정합니다. (`docker-container`, `kubernetes`, `remote`)
`--driver-opt`			드라이버에 따른 옵션을 설정합니다.
`--name`			`builder` 이름을 지정합니다.
`--platform`			`platform`을 지정합니다.
`--use`			생성 후 해당 `builder`를 사용합니다.

Dirver별 특징

> https://docs.docker.com/build/drivers/

Feature	`docker`	`docker-container`	`kubernetes`	`remote`
Automatically load image	✅			

Feature	`docker`	`docker-container`	`kubernetes`	`remote`
Cache export	√*	✅	✅	✅
Tarball output		✅	✅	✅
Multi-arch images		✅	✅	✅
BuildKit configuration		✅	✅	Managed externally

[예시] `Multi-platform`용 `builder` 생성하기

```
$ docker buildx create --driver docker-container --name multi-builder --platform
linux/amd64,linux/arm64

$ docker buildx inspect multi-builder

Name:          multi-builder
Driver:        docker-container
Last Activity: 2024-07-07 06:39:02 +0000 UTC

Nodes:
Name:      multi-builder0
Endpoint:  npipe:////./pipe/docker_engine
Status:    inactive
Platforms: linux/amd64*, linux/arm64*
```

builder 정보 조회

> https://docs.docker.com/engine/reference/commandline/buildx_inspect/

```
docker buildx inspect [NAME]
```

[예시] 현재 사용중인 `builder` 조회하기

```
$ docker buildx inspect --bootstrap
Name:          default
Driver:        docker
Last Activity: 2023-12-22 02:37:31 +0000 UTC

Nodes:
Name:      default
Endpoint:  default
Status:    running
Buildkit:  v0.11.6+616c3f613b54
```

```
Platforms: linux/amd64, linux/amd64/v2, linux/amd64/v3, linux/arm64,
linux/riscv64, linux/ppc64le, linux/s390x, linux/386, linux/mips64le,
linux/mips64, linux/arm/v7, linux/arm/v6
Labels:
 org.mobyproject.buildkit.worker.moby.host-gateway-ip: 192.168.65.254
GC Policy rule#0:
 All:           false
 Filters:
type==source.local,type==exec.cachemount,type==source.git.checkout
 Keep Duration: 172.8µs
 Keep Bytes:    2.764GiB
GC Policy rule#1:
 All:           false
 Keep Duration: 5.184ms
 Keep Bytes:    20GiB
GC Policy rule#2:
 All:        false
 Keep Bytes: 20GiB
GC Policy rule#3:
 All:        true
 Keep Bytes: 20GiB
```

builder 선택하기

> https://docs.docker.com/engine/reference/commandline/buildx_use/

```
docker buildx use [OPTIONS] NAME
```

Option	Short	Default	Description
`--default`			`default`로 설정합니다.

이미지 빌드하기

> https://docs.docker.com/engine/reference/commandline/buildx_build/

```
docker buildx build [OPTIONS] PATH | URL | -
```

주요 옵션

Option	Short	Default	Description
`--build-arg`			`ARG`를 설정합니다.
`--file`	`-f`		`Dockerfile`의 경로를 지정합니다.

Option	Short	Default	Description
`--label`			라벨을 추가합니다.
`--no-cache`			이미지 빌드시 캐시를 사용하지 않습니다.
`--platform`			`platform`을 지정합니다.
`--pull`			관련된 이미지를 저장 유무에 관계없이 `pull` 합니다.
`--load`			이미지를 `host`에 저장합니다. (`--output=type=docker`와 동일합니다.)
`--push`			이미지를 `registry`에 저장합니다 (`--output=type=registry`와 동일합니다.)
`--tag`	`-t`		이름과 `Tag`를 설정합니다.

주의사항

- `docker` 드라이버를 사용하는 `builder`는 단일 `platform` 이미지만 빌드할 수 있습니다.
- `docker` 드라이버가 아닌 다른 드라이버를 사용하는 경우 `--load`를 사용하여야 이미지를 가져올 수 있습니다.
  - `--load`는 `export`를 이용하여 이미지를 추출하고 `import` 합니다.
- 2개 이상의 `platform`을 지원하는 이미지를 제작하는 경우 `--load`가 불가능하므로 `--push`를 통해 `registry`로 바로 업로드해야 합니다.

[예시] `linux/arm/v7`용 이미지 제작하기

현재 `builder`에서 지원하는 `platform`을 다음과 같이 확인합니다.

```
$ docker buildx inspect --bootstrap
Name:          multi-arch-builder
Driver:        docker-container
Last Activity: 2023-12-23 13:46:16 +0000 UTC

...
Platforms: linux/amd64, linux/amd64/v2, linux/amd64/v3, linux/arm64,
linux/riscv64, linux/ppc64le, linux/s390x, linux/386, linux/mips64le,
linux/mips64, linux/arm/v7, linux/arm/v6
...
```

`--platform` 옵션을 이용하여 `linux/arm/v7`용 이미지를 제작합니다.

```
$ docker buildx build --platform linux/arm/v7 -t cloudwave:arm .
...

$ docker image inspect cloudwave:arm -f "arch: {{ .Architecture }}/{{ .Variant
}}"
arch: arm/v7
```

build할 때 load나 push 둘 중 하나는 무조건 해야 한다. 안그러면 로컬에 남아있기 때문에 사용 불가

연습1

[연습] `buildx`를 이용하여 `ARM`용 `cloudwave:base.v1` 이미지 제작하기

> [연습] 실습용 `ubuntu` 이미지 제작하기

`buildx ls`를 이용하여 현재 사용중인 `builder`가 지원하는 `platform` 목록을 확인합니다.

```
$ NAME/NODE             DRIVER/ENDPOINT               STATUS  BUILDKIT
  PLATFORMS
default *             docker
  default             default                       running v0.11.6+616c3f613b54
linux/amd64, linux/amd64/v2, linux/amd64/v3, linux/arm64, linux/riscv64,
linux/ppc64le, linux/s390x, linux/386, linux/mips64le, linux/mips64,
linux/arm/v7, linux/arm/v6
```

`buildx build` 명령어를 이용하여 `linux/arm/v7`용 이미지를 제작합니다.

```
# builder의 driver가 docker인 경우
$ docker buildx build --platform linux/arm/v7 -t cloudwave:arm.v1 .

# builder의 driver가 docker-container인 경우 `--load` 옵션을 추가합니다.
$ docker buildx build --platform linux/arm/v7 -t cloudwave:arm.v1 --load .
```

다음 명령어를 실행하여 생성한 이미지의 `Architecture`를 확인 할 수 있습니다.

```
$ docker image inspect cloudwave:arm.v1 -f "arch: {{ .Architecture }}/{{ .Variant
}}"
arch: arm/v7
```

Dockerfile

```
FROM linuxserver/code-server:4.90.3

ENV TZ="Asia/Seoul"
ENV PUID=1000
ENV PGID=1000

RUN apt-get update && apt-get -y upgrade

RUN apt install -y ca-certificates curl gnupg software-properties-common wget unzi
```

```
# Docker CLI
RUN install -m 0755 -d /etc/apt/keyrings  \
    && curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyring
    && chmod a+r /etc/apt/keyrings/docker.asc

RUN echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docke
      $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
      tee /etc/apt/sources.list.d/docker.list > /dev/null

RUN apt-get update && apt-get install -y docker-ce docker-ce-cli
```

해당 dockerfile로 할 경우 - -platform을 linux/arm64로 해야 한다.

docker buildx build --platform linux/arm64 -t cloudwave:arm.v1

실습

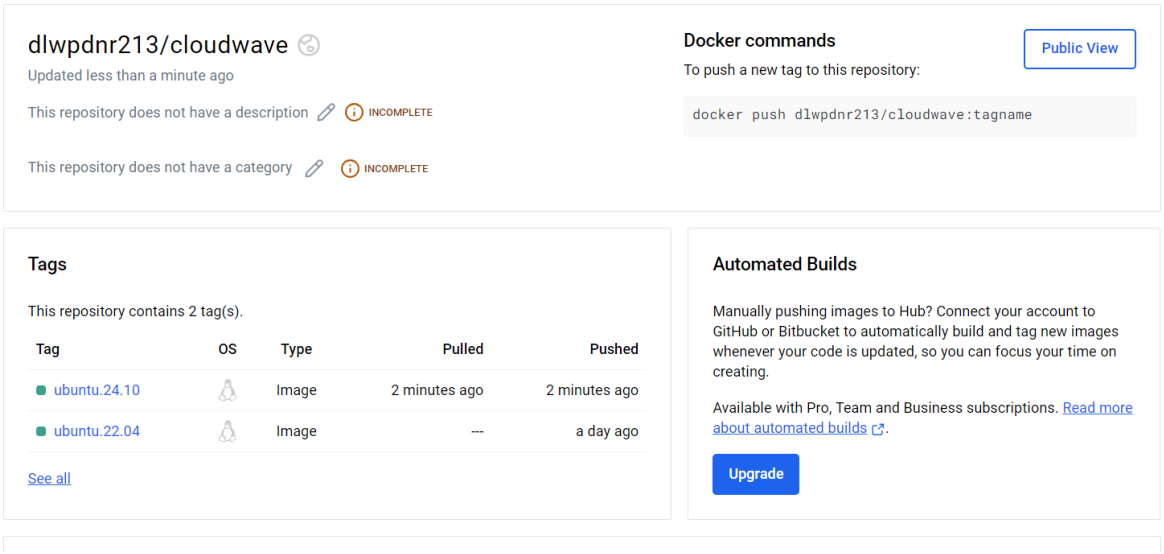
[실습] buildx를 이용하여 Multi-platform 이미지 제작하기

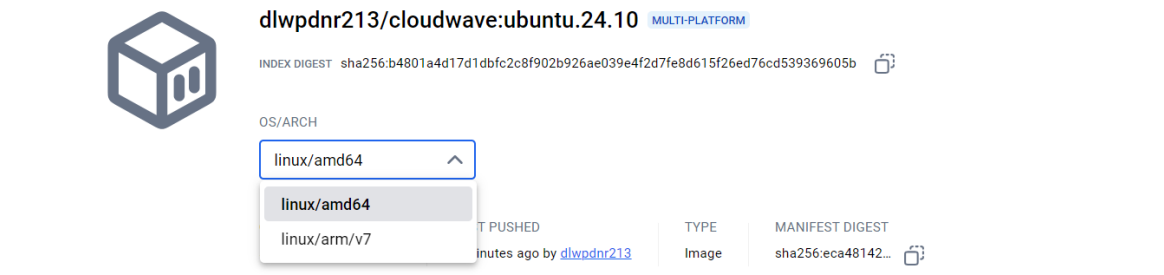
1. Docker Hub에 로그인 합니다.
2. docker-container를 사용하는 builder를 생성합니다.
    - --use 옵션을 사용하여 바로 사용할 수 있도록 설정합니다.
3. 다음 조건을 만족하는 Multi-platform 이미지를 build 합니다

    - --push를 이용하여 Docker Hub로 업로드해야 합니다.
    - 이미지는 linux/amd64, arm/v6를 지원해야 합니다.
4. Docker Hub에서 업로드한 이미지를 확인합니다.

```
docker buildx create --use --driver docker-container --name multi-builder2 --platf
```

```
docker buildx build --push --platform linux/amd64,linux/arm/v7 -t dlwpdnr213/cloud
```





Multi Stage

dockerfile (go - compile)

FROM 베이스 이미지로부터 시작

CMD 로 끝나는 것이 일반적

FROM이 두번 쓰였다

```
FROM golang:1.19-alpine as build
WORKDIR /app
COPY src ./
RUN CGO_ENABLED=0 go build -o main
FROM scratch as release
COPY --from=build /app/main /app/
```

```
WORKDIR /app
CMD ["/app/main"]
```

Build time시의 종속성과 Run Time 시의 종속성은 전혀 같지 않다.

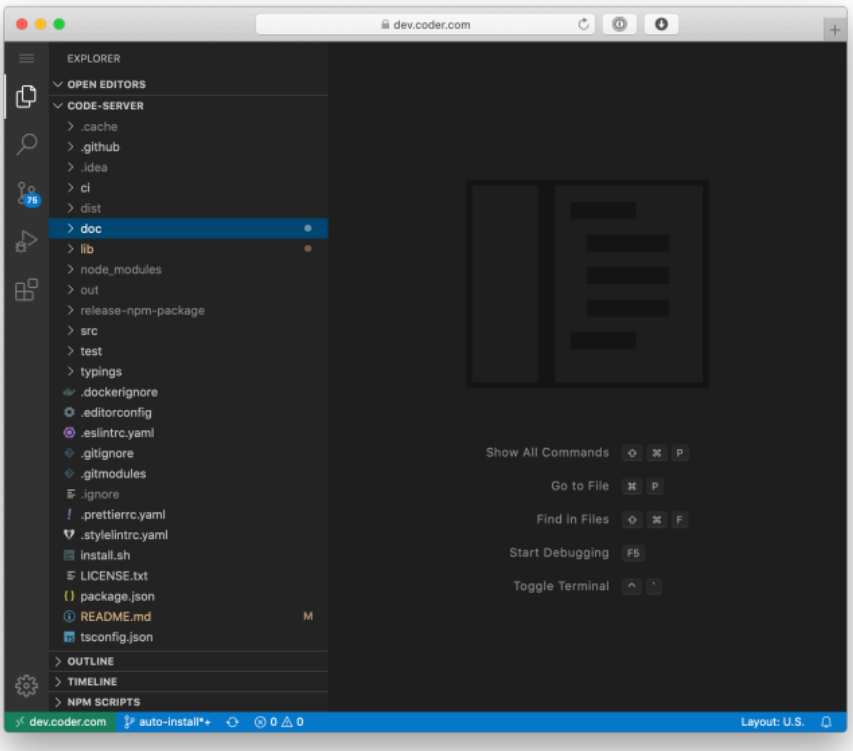
scratch는 compile할 수는 없지만 compile된 bin 파일을 실행할 능력은 가진 이미지임
COPY --from=build /app/main /app/ 이 코드를 통해서 bin파일을 가져오는 것임

즉 멀티 스테이지는 오류 가능성을 줄이고 런타임 시에 필요한 것만 남겨놓기 위한 기법
이 것은 옵셔널한 방법 굳이 안해도 되지만 하면 매우 좋다

종합 실습 1

종합 문제

[실습-1] `code-server` 실행하기

> `code-server`는 Web에서 VS Code를 사용할 수 있게 만들어주는 서비스입니다.
>
> 
>

- `linuxserver/code-server:4.90.3` 이미지를 사용하여 컨테이너를 생성합니다.
  - 컨테이너의 이름은 `ide`로 설정합니다.
  - 8443 포트를 노출합니다.
- 다음과 같이 환경변수를 설정합니다.

```
PASSWORD=password
DEFAULT_WORKSPACE=/code
PUID=1000
PGID=1000
TZ="Asia/Seoul"
```

- 다음과 같이 Volume을 설정합니다.
  - 로컬 머신의 `src` 디렉토리를 컨테이너의 `/code` 디렉토리에 마운트 합니다.

  - 로컬 머신의 `/var/run/docker.sock`를 컨테이너의 `/var/run/docker.sock`에 마운트 합니다.
- 브라우저에서 `localhost:8443` 로 접속하여 서비스가 사용 가능한지 확인합니다.

local은 상관없지만 컨테이너 포트가 8443

환경변수 5개

```
docker volume create config

docker run -d --name ide -p 8443:8443 -e PASSWORD=password -e SUDO_PASSWORD=wave -
```

실습2

[실습-2] `Code-server`에 `Docker` 설치하기

- `exec` 를 사용하여 컨테이너의 Shell에 접근합니다.
- 다음 스크립트를 사용하여 Docker를 설치합니다.
  - https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository

```
apt-get update
apt-get install ca-certificates curl
install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
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 \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  tee /etc/apt/sources.list.d/docker.list > /dev/null
apt-get update
apt-get install -y docker-ce docker-ce-cli
```

- `vsc` 터미널에서 다음 명령어를 사용하여 Docker 버전을 확인합니다.

```
docker --version
```