

가상화 플랫폼의 이해



목차

- Why Docker?
- Docker 실습환경
- Docker Commands
- 기타 명령어 옵션
- Docker Images & Dockerfile
- CMD vs ENTRYPOINT
- Docker Storage
- Docker Compose

Why Docker?

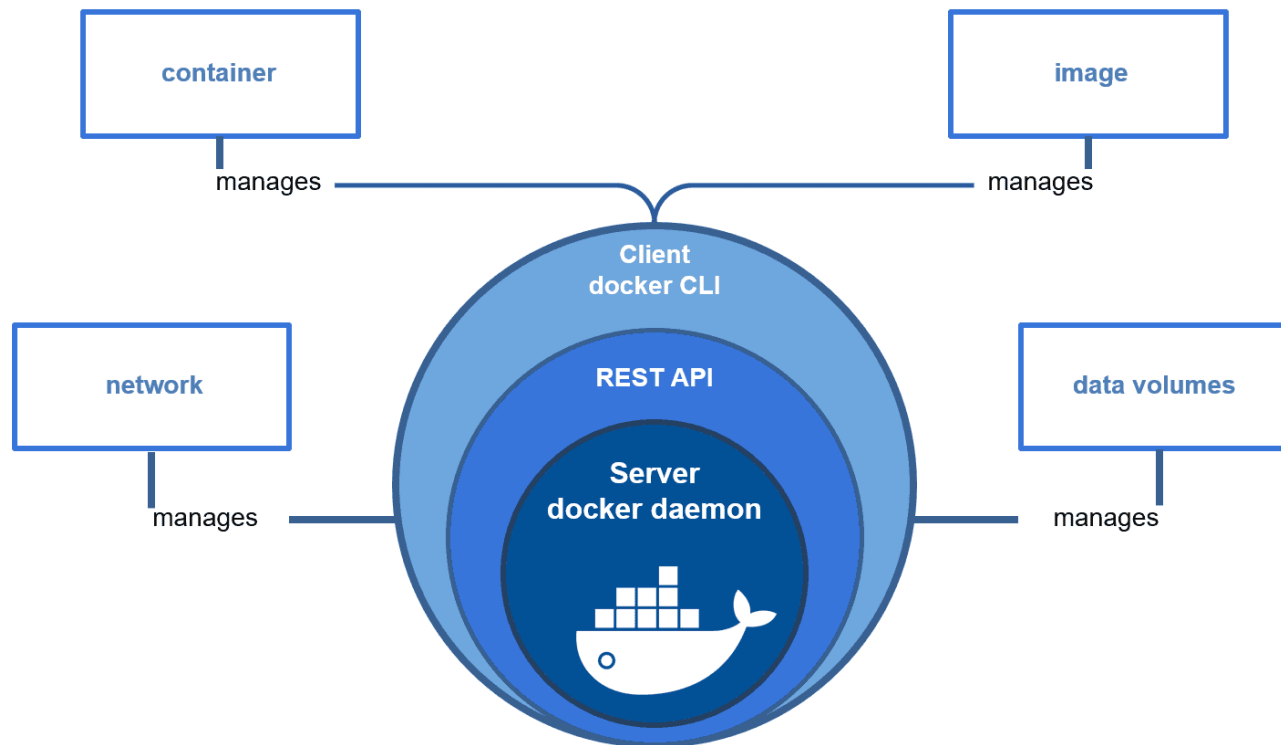
필요한 사전지식

- Ubuntu 사용법
- Web 기초
- 파이썬 패키지 설치 방법

Why docker?

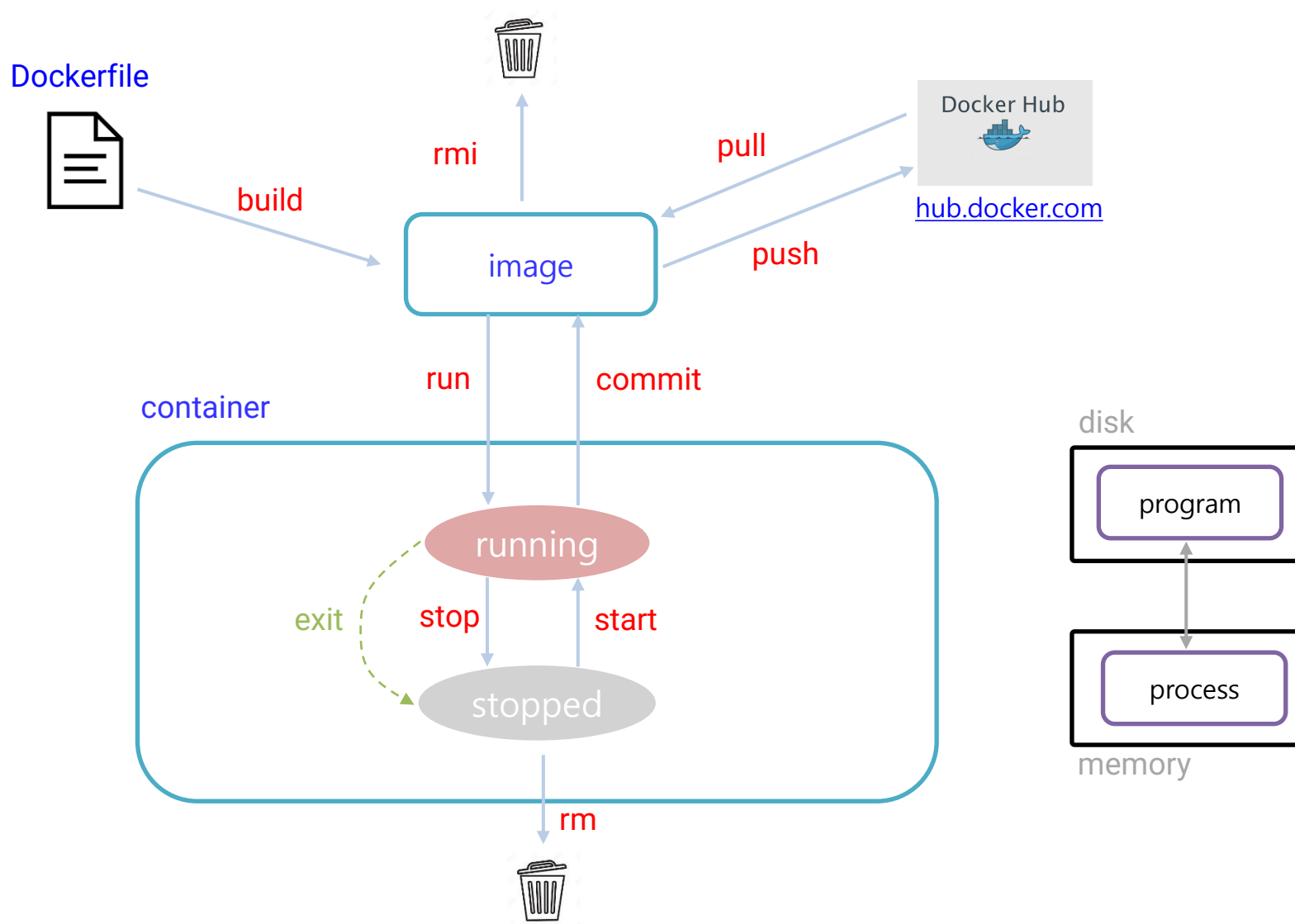
- Compatibility / Dependency
 - the right version of each components
- Long setup time
- Different Develop/Test/Production environment

Docker Engine Components



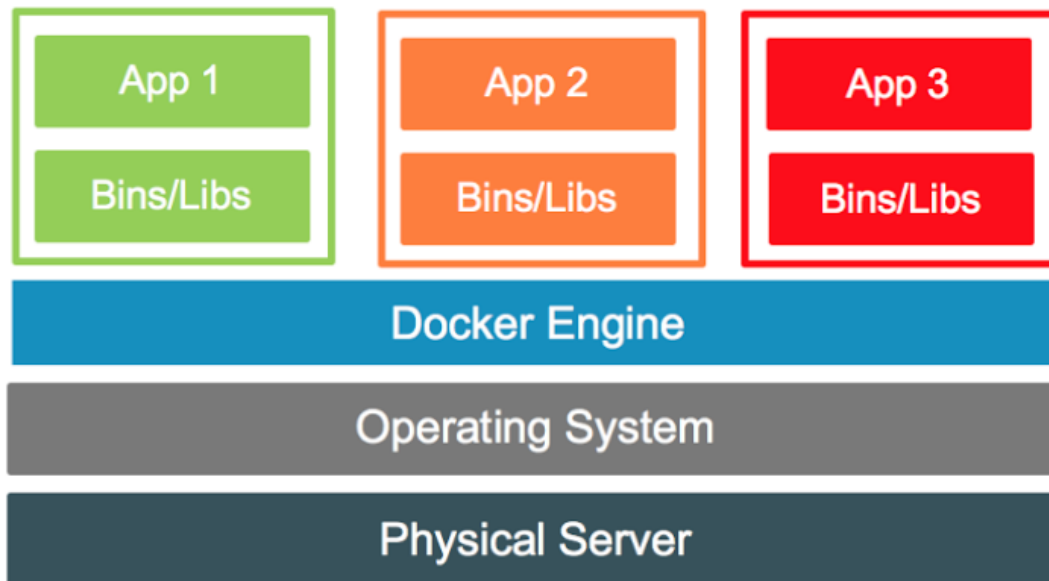
<https://k21academy.com/docker-kubernetes/docker-architecture-docker-engine-components-container-lifecycle/>

Docker 구성 및 동작

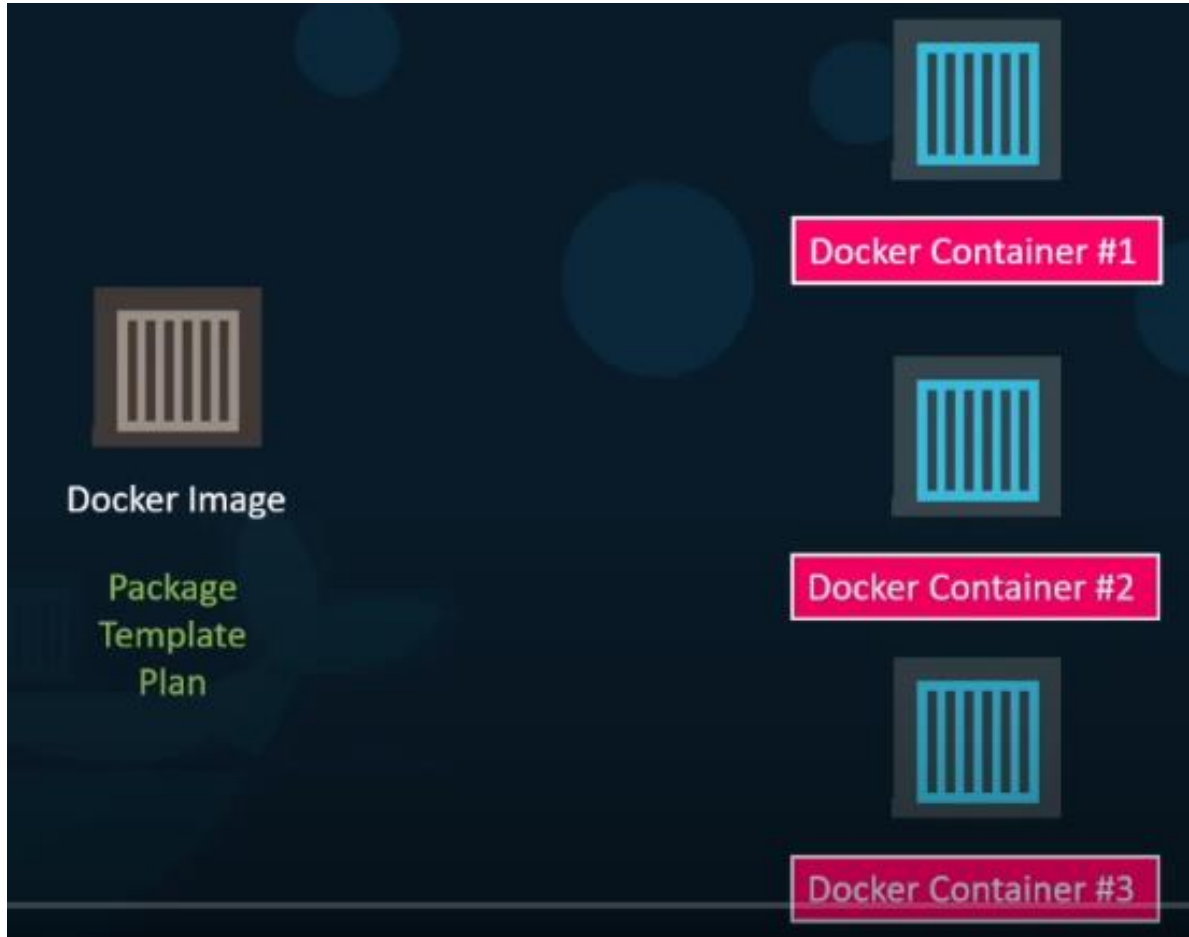


Container

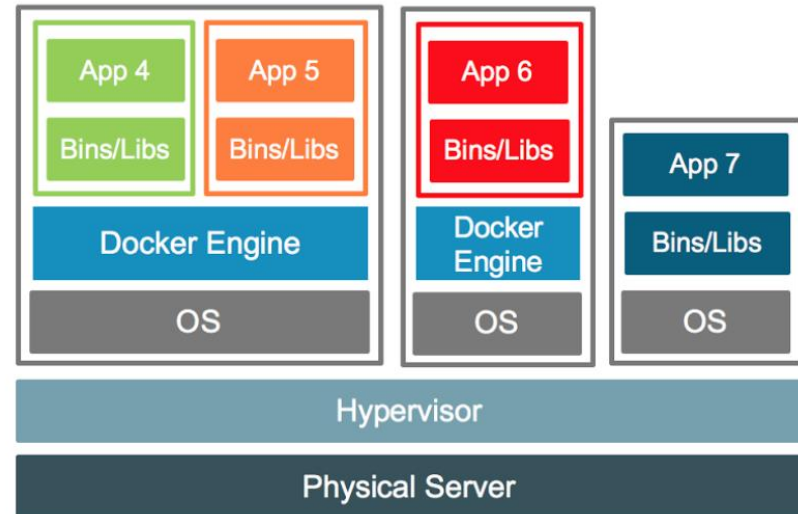
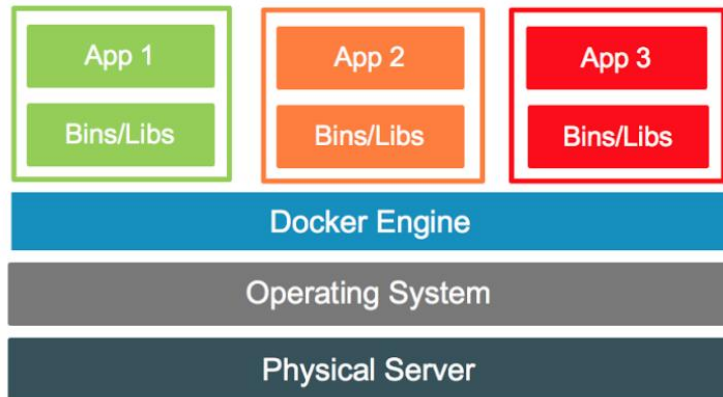
- completely isolated environments
 - 프로세스와 네트워크 등을 독자적으로 가지고 있는 격리된 환경
- OS 커널을 공유
- 특징
 - physically : process
 - logically : system



docker image vs container



Containers vs Virtual Machines



Docker 실습환경 (UBUNTU)

Install docker (UBUNTU)

- 설치

```
$ curl -fsSL https://get.docker.com/ | sh
```

- sudo 없이 docker 사용하기 설정

```
$ usermod -aG docker $USER
```

➔ 윈도우10 에서 docker 설치하려면 복잡함

➔ KODECLOUD 서비스를 이용하면 윈도우10에서 우분투를 무료로 사용 가능함

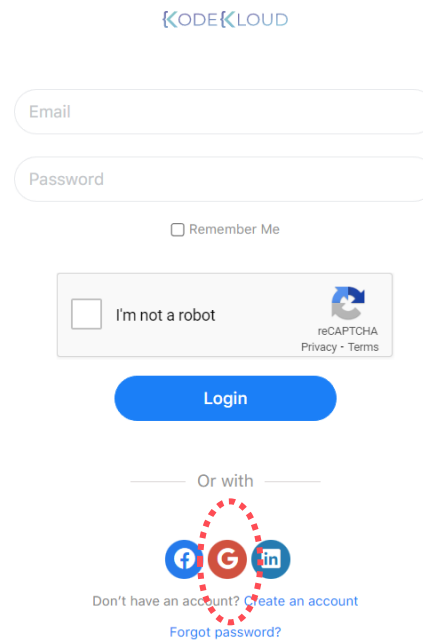
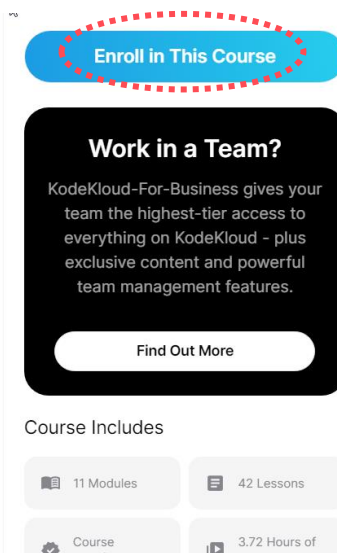
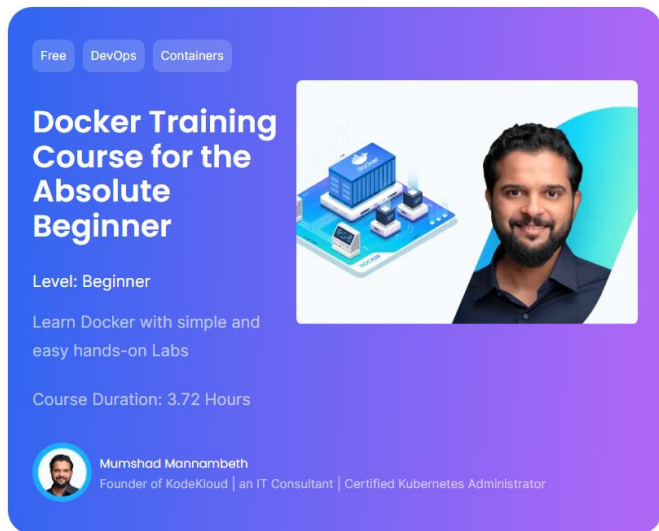
docker 실습 환경 setup

1. 크롬 브라우저로 kodecloud 사이트 접속

<https://learn.kodecloud.com/user/courses/docker-training-course-for-the-absolute-beginner>

2. 'Enroll in This Course' 버튼 선택하여 등록

➔ 로그인 화면 나오면 구글 메일로 계정생성하여 로그인



docker 실습 환경 setup

3. Course Content 에서 'Docker Command' 선택한 후 'Labs: Basic Docker Commands' 클릭

Course Content

[Expand All](#)

▶ Introduction

6 Lessons

0%

▼ Docker Commands

4 Lessons

0%

Module Content

0% Complete

0/4 Lessons

▶ Basic Docker Commands 08:00

○

▶ Demo - Docker Commands 17:59

○

▶ Labs: Basic Docker Commands

○

▶ Download Course Deck

○

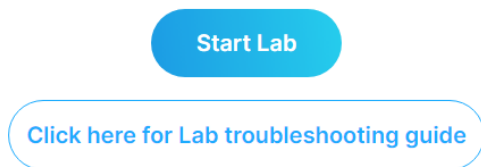
▶ Docker Run

4 Lessons

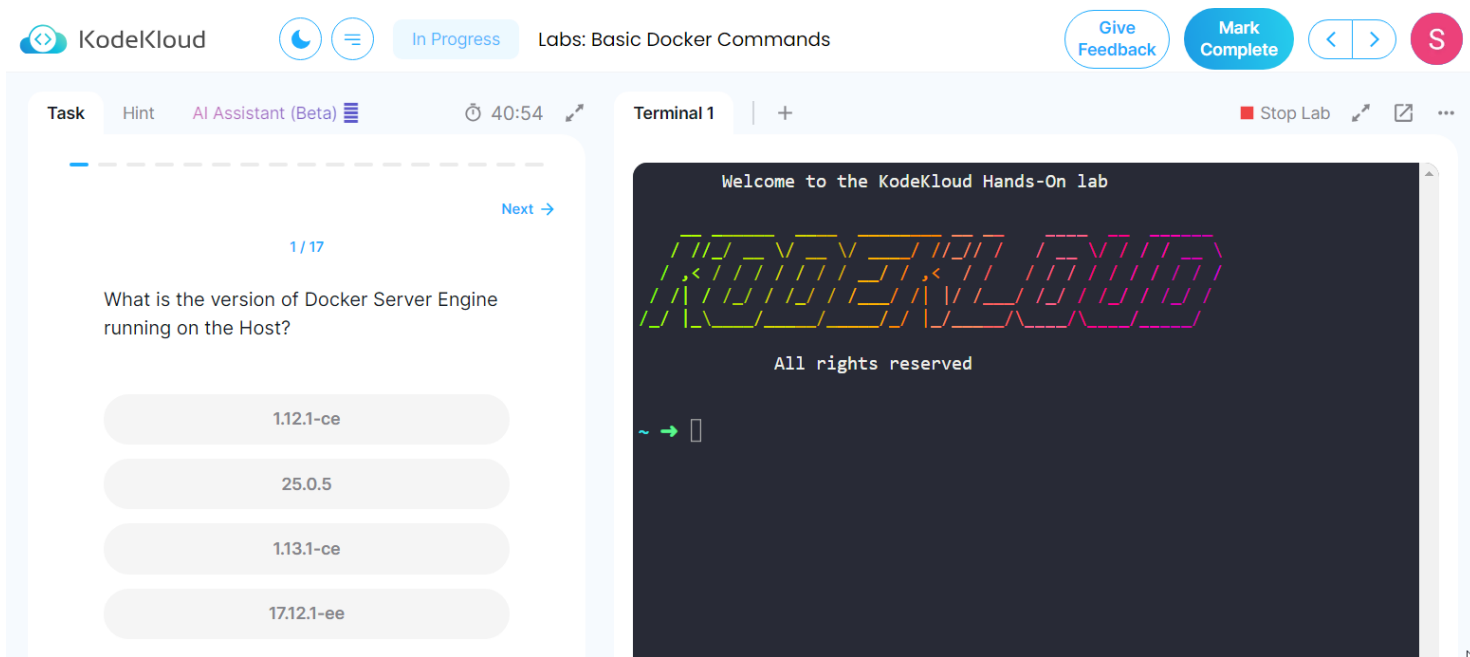
0%

docker 실습 환경 setup

4. 'START LAB' 버튼이 나오면 누른다.



5. 아래와 같은 터미널 화면이 나오면 실습 가능한 상태임
➔ 이미 도커가 설치된 상태이므로 도커를 바로 사용 가능함



docker 설치 확인

```
$ docker version
```

```
$ docker run docker/whalesay cowsay Hello-World!
```

```
$ docker run docker/whalesay cowsay Hello-World!
Unable to find image 'docker/whalesay:latest' locally
latest: Pulling from docker/whalesay
Image docker.io/docker/whalesay:latest uses outdated schema1 manifest format. Please upgrade to a schema2 image for
better future compatibility. More information at https://docs.docker.com/registry/spec/deprecated-schema-v1/
e190868d63f8: Pull complete
909cd34c6fd7: Pull complete
0b9bfabab7c1: Pull complete
a3ed95caeb02: Pull complete
00bf65475aba: Pull complete
c57b6bcc83e3: Pull complete
8978f6879e2f: Pull complete
8eed3712d2cf: Pull complete
Digest: sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3d31e18aaf55f3258ba93b
Status: Downloaded newer image for docker/whalesay:latest
```

```
< Hello-World! >
```

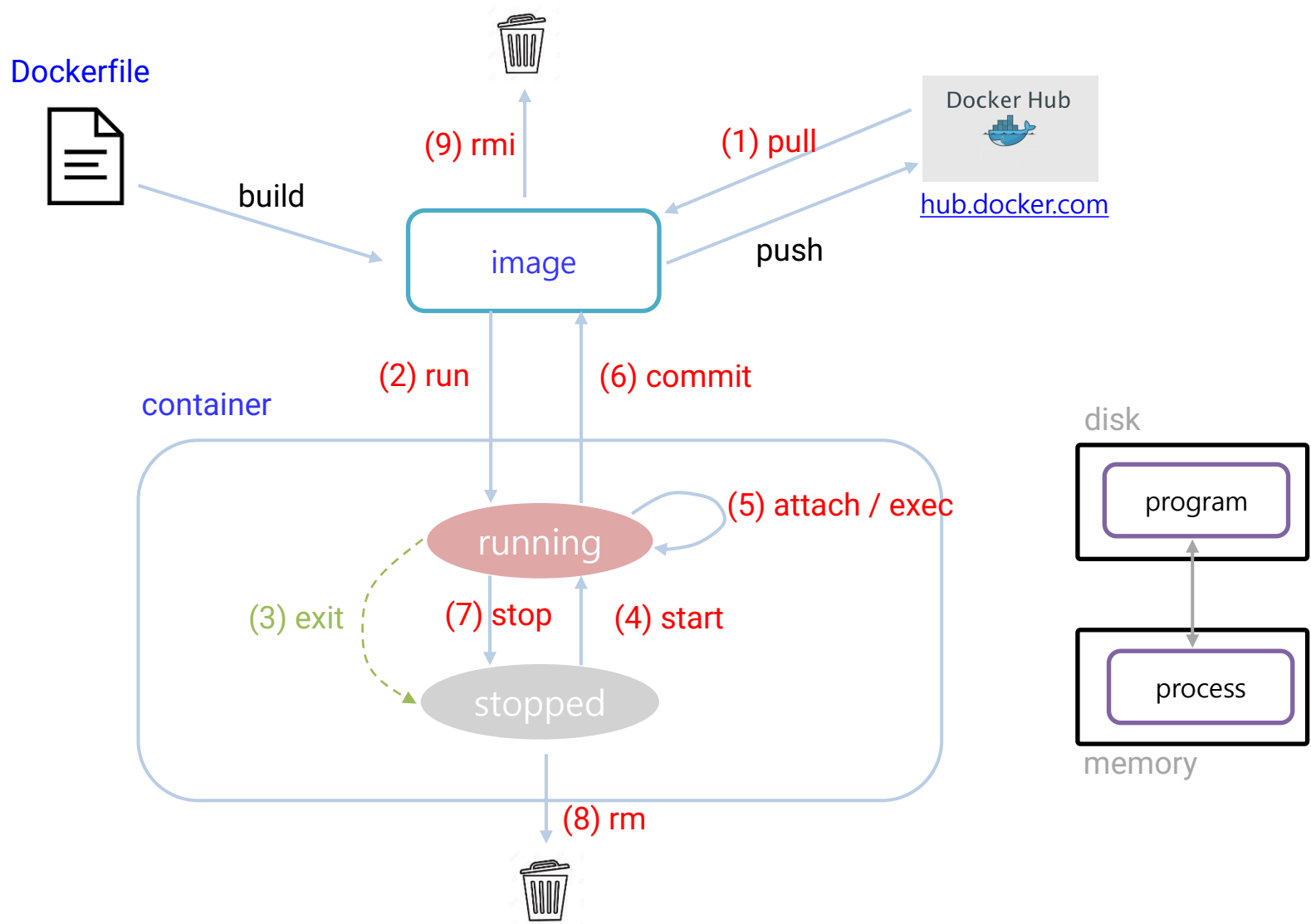

kodekloud 제약사항

무료 서비스이므로 몇가지 제약사항이 있음.

1. 한번 접속하면 1시간 동안 사용할 수 있고, 1시간이 지나면 초기화되며 다시 접속하여 사용해야 함(이전 사용내역 모두 삭제됨)
2. 15분 이상 사용을 하지 않으면 접속이 해제 됨

Docker Commands

실행 시나리오



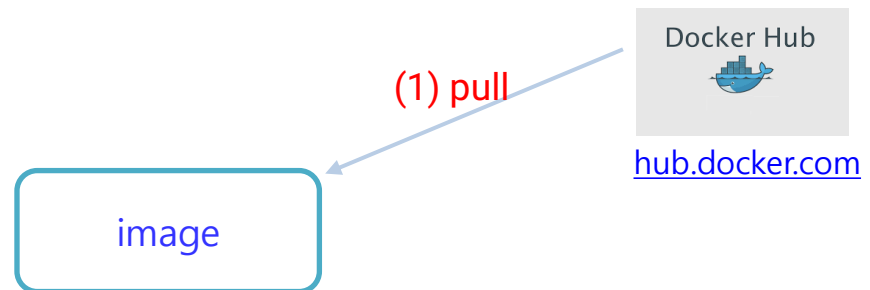
pull

```
docker pull [image name]
```

```
$ docker pull python:latest
```

tag

- Docker Hub에서 python 이미지를 다운로드
- python:latest에서 latest 생략가능



images

docker images

\$ docker images

- 이미지 리스트를 표시

```
[home-ubuntu: snap]$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
python               latest              3189819ced3e       6 days ago         934MB
```

도커 이미지
이름

도커 이미지
ID

이미지 크기

run

```
docker run [options] [image name] [command]
```

```
$ docker run -it --name python3.8 python /bin/bash
```

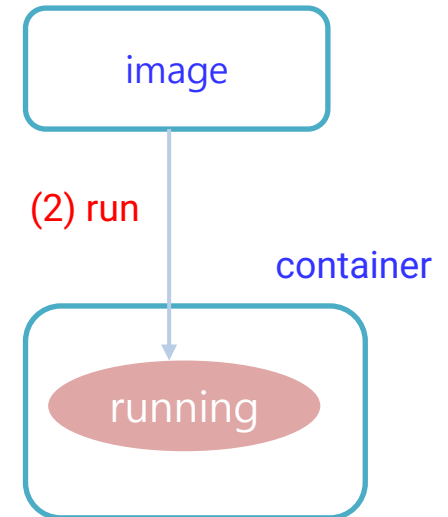
'python' 이미지를 이용하여 컨테이너 python3.8을 생성한 후 bash 실행

options	의미
-i	interactive mode
-t	tty
--name	컨테이너 이름 설정

도커 컨테이너로
들어가면 prompt가
'#' 으로 표시됨에
주의!!

```
[home-ubuntu: snap]$ docker run -i -t --name python3.8 python /bin/bash  
root@3b3584a926cb: /#
```

-i -t 옵션을 사용하면 컨테이너가 terminal에 attach 되어 터미널 모드를 사용할 수 있음

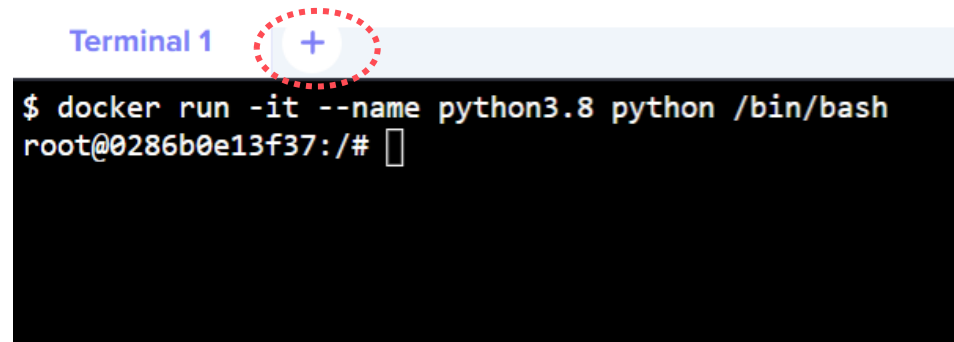


KODEKLOUD 터미널 창 추가

- Terminal 1

- 현재 도커가 실행 중임

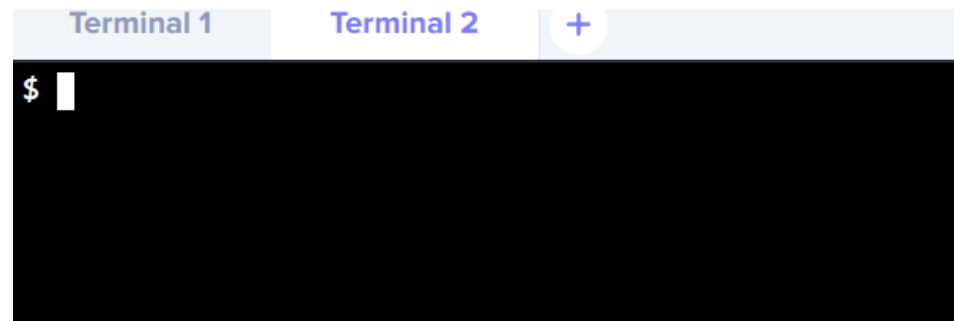
➔ '+' 를 눌러 새로운 터미널 창 Terminal2 를 생성한다.



A screenshot of a terminal window titled 'Terminal 1'. At the top, there is a blue bar with the text 'Terminal 1' and a red dashed circle around a '+' icon. The terminal content shows the command '\$ docker run -it --name python3.8 python /bin/bash' being executed, followed by the prompt 'root@0286b0e13f37:/#' and a cursor.

- Terminal 2

- 호스트(우분투) 터미널로 사용



A screenshot of a terminal window titled 'Terminal 2'. At the top, there is a blue bar with the text 'Terminal 2' and a '+' icon. The terminal content shows a host terminal prompt '\$' followed by a cursor.

➔ 도커 컨테이너의 prompt는 '#' 이고 호스트 터미널의 prompt는 '\$' 으로 표시되므로 prompt로 식별하면 됨!

ps

- ps : 현재 실행 중인 컨테이너 리스트를 표시
- 호스트(우분투) 터미널에서 아래 명령 실행 (주의 : 도커에서 실행하면 안됨)

```
docker ps
```

```
$ docker ps
```

```
[home-ubuntu: snap]$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
3b3584a926cb	python	"/bin/bash"	5 minutes ago	Up 5 minutes		python3.8

컨테이너
ID

도커 이미지
이름

컨테이너
이름

컨테이너에서 exit

```
root@[container id]# exit
```

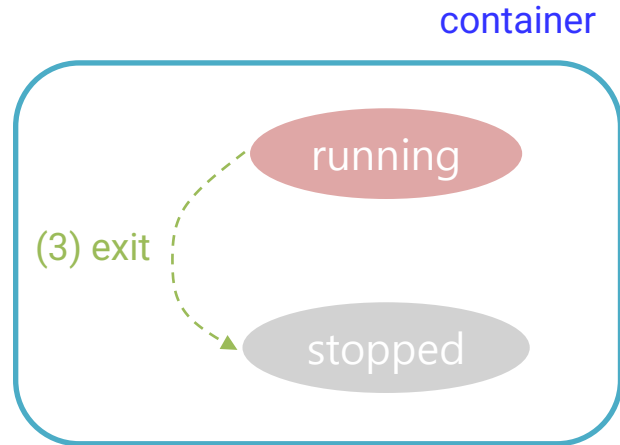
- 도커 명령아님
- 컨테이너에서 작업을 끝낸 후 exit 하면 컨테이너가 중지(stopped) 상태가 됨

user

컨테이너 ID

```
root@3b3584a926cb:/# ls
bin boot dev etc home lib lib64 media mnt opt
root@3b3584a926cb:/# exit
exit
[home-ubuntu: snap]$
```

exit하면 컨테이너가
중지됨



ps -a

```
docker ps -a
```

```
$ docker ps -a
```

- 현재 실행 중인 컨테이너와 중지된 컨테이너를 모두 표시

```
[home-ubuntu: snap]$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
```

'a' 옵션이 없으면 실행 중인 컨테이너만 표시되므로 표시되는 내용이 없음

```
[home-ubuntu: snap]$ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
3b3584a926cb   python    "/bin/bash"             44 minutes ago  Exited (0) 21 minutes ago          python3.8
```

'a' 옵션을 사용하면 중지된 컨테이너도 표시됨

'Exited' 상태임

start (1)

```
docker start [container name]  
docker start [container ID]
```

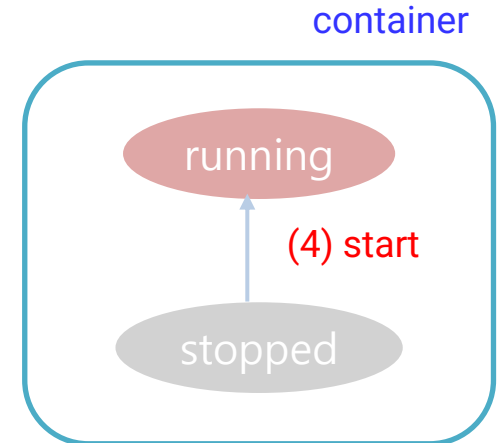
```
$ docker start python3.8  
$ docker start 3b3584a926cb
```

- 중지된 컨테이너를 다시 시작
- 컨테이너를 지정할 때 이름 혹은 ID 사용 (이후 설명에서는 컨테이너 이름만 사용)

```
[home-ubuntu:snap]$ docker start python3.8  
python3.8  
[home-ubuntu:snap]$
```

컨테이너가 시작되었는데 이전과 다름!

👉 컨테이너 프롬프트가 안보임



start (2)

- 'docker ps' 를 실행하면 분명히 python3.8 컨테이너가 실행중으로 나오지만 컨테이너 프롬프트는 안보이는 상태임

```
[home-ubuntu: snap]$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
3b3584a926cb	python	"/bin/bash"	2 hours ago	Up 4 minutes		python3.8

☞ 2가지 선택 방법이 있으며 상황에 따라 선택

- ① 컨테이너 밖에서 컨테이너 내부 명령을 실행하는 방법 : **docker exec**
- ② 컨테이너 내부로 다시 들어가는 방법(프롬프트 사용) : **docker attach**

exec

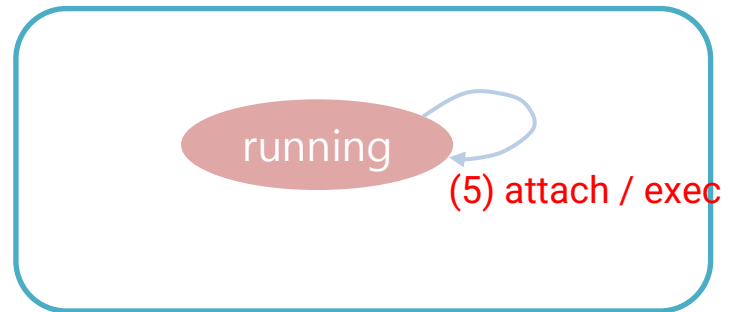
```
docker exec [container name] [command] [parameter]
```

```
$ docker exec python3.8 ls -la
```

- 컨테이너 밖에서 컨테이너 내부 명령을 실행

```
[home-ubuntu: snap]$ docker exec python3.8 ls -la
total 72
drwxr-xr-x 64 root root 4096 Jul 28 15:25 .
drwxr-xr-x 64 root root 4096 Jul 28 15:25 ..
-rwxr-xr-x 1 root root 0 Jul 28 15:02 .dockerenv
drwxr-xr-x 2 root root 4096 Jul 22 03:01 bin
drwxr-xr-x 2 root root 4096 May 2 16:39 boot
drwxr-xr-x 5 root root 360 Jul 28 16:32 dev
drwxr-xr-x 58 root root 4096 Jul 28 15:02 etc
drwxr-xr-x 2 root root 4096 May 2 16:39 home
drwxr-xr-x 12 root root 4096 Jul 22 03:02 lib
drwxr-xr-x 2 root root 4096 Jul 20 00:00 lib64
drwxr-xr-x 2 root root 4096 Jul 20 00:00 media
drwxr-xr-x 2 root root 4096 Jul 20 00:00 mnt
drwxr-xr-x 2 root root 4096 Jul 20 00:00 opt
dr-xr-xr-x 304 root root 0 Jul 28 16:32 proc
TeamViewer - 2 root root 4096 Jul 28 15:25 root
drwxr-xr-x 3 root root 4096 Jul 20 00:00 run
drwxr-xr-x 2 root root 4096 Jul 22 03:01/sbin
drwxr-xr-x 2 root root 4096 Jul 20 00:00 srv
dr-xr-xr-x 13 root root 0 Jul 28 15:25 sys
drwxrwxrwt 2 root root 4096 Jul 22 12:29 tmp
drwxr-xr-x 38 root root 4096 Jul 20 00:00 usr
drwxr-xr-x 27 root root 4096 Jul 20 00:00 var
[home-ubuntu: snap]$
```

container



exec 실행 후 컨테이너 밖(host 머신)에 남아 있음

attach(1)

```
docker attach [container name]
```

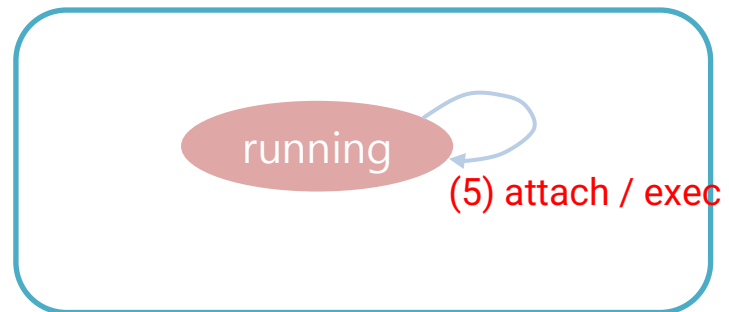
```
$ docker attach python3.8
```

- 컨테이너 내부로 다시 들어감

```
[home-ubuntu: snap]$ docker attach python3.8  
root@3b3584a926cb: /#  
root@3b3584a926cb: /#
```

컨테이너 프로프트가 나옴

container



attach(2)

- 컨테이너에서 아래와 같이 mydata 폴더와 docker.txt 파일 생성

```
root@3b3584a926cb:/# mkdir mydata
root@3b3584a926cb:/# cd mydata

root@3b3584a926cb:/# echo "hello world" > docker.txt
root@3b3584a926cb:/# more docker.txt
root@3b3584a926cb:/# ls -la
root@3b3584a926cb:/# pwd
```

```
root@3b3584a926cb:/mydata# more docker.txt
hello world
root@3b3584a926cb:/mydata# ls -la
total 12
drwxr-xr-x  2 root root 4096 Jul 28 17:50 .
drwxr-xr-x 65 root root 4096 Jul 28 17:20 ..
-rw-r--r--  1 root root  12 Jul 28 17:50 docker.txt
root@3b3584a926cb:/mydata# pwd
/mydata
root@3b3584a926cb:/mydata#
```

- 👉 컨테이너 내용이 변경되었음
- 👉 변경된 컨테이너를 새로운 이미지로 저장하여 백업시켜 놓을 수 있음
- 👉 "commit"

commit

```
docker commit [options] [container name] [image name]
```

```
$ docker commit python3.8 python3.8
```

컨테이너 이름

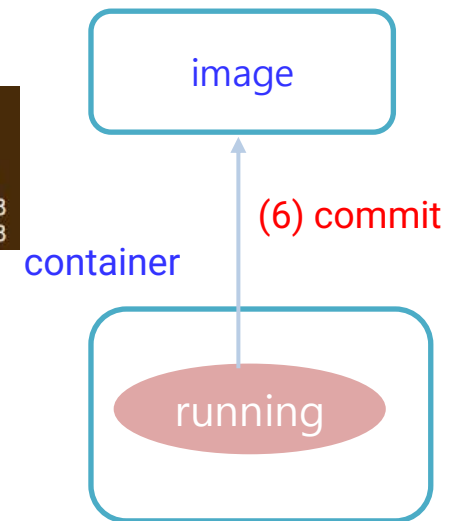
이미지 이름

- 변경된 컨테이너를 새로운 이미지로 저장
- 예시는 새로운 이미지 이름을 컨테이너 이름과 동일하게 함
 - 'docker images' 명령으로 도커 이미지 리스트를 조회할 수 있음

```
[home-ubuntu: snap]$ docker commit python3.8 python3.8
sha256:af20fda15fdd21fb7519818cd2d5e0d0b3704fc4405df5369a3610d4ebf0e6c7
[home-ubuntu: snap]$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python3.8	latest	af20fda15fdd	6 seconds ago	934MB
python	latest	3189819ced3e	6 days ago	934MB

기존 이미지와 새로 생성된
이미지 2개가 표시됨



실습

- docker ps 및 docker ps -a 명령을 실행해서 차이점을 확인하세요

stop / rm

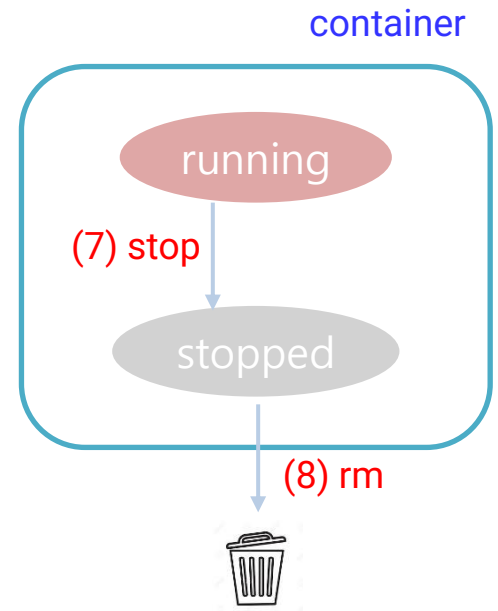
```
docker stop [container name]
docker rm  [container name]
```

```
$ docker stop python3.8
$ docker rm  python3.8
```

- 컨테이너가 실행중이라면 컨테이너를 먼저 중지시킨 후 삭제해야 함
- 컨테이너를 삭제

```
[home-ubuntu:snap]$docker rm python3.8
Error response from daemon: You cannot remove a running container
tempting removal or force remove
[home-ubuntu:snap]$docker stop python3.8
python3.8
```

```
[home-ubuntu:snap]$docker rm python3.8
python3.8
[home-ubuntu:snap]$
```



실습

- docker ps 및 docker ps -a 명령을 실행하여
 - stop 전후 컨테이너 상태를 확인하세요
 - rm 전후 컨테이너 상태를 확인하세요

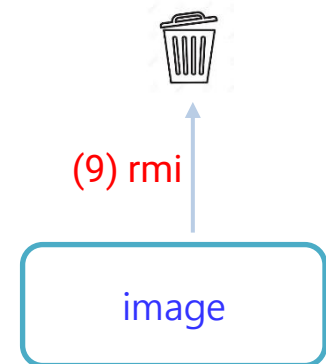
rmi (1)

```
docker rmi [image name]  
docker rmi [image ID]
```

```
$ docker rmi python  
$ docker rmi af20fda15fdd
```

- 도커 이미지를 삭제
- 해당 이미지의 컨테이너가 남아 있다면 컨테이너를 먼저 삭제시킨 후 이미지를 삭제해야 함

※ 'docker rmi ubuntu' 처럼 이미지 이름만 지정하면 태그는 다르지만 ubuntu 이름을 가진 모든 이미지가 삭제됨에 주의



rmi (2)

- 앞에서 저장한 python3.8 이미지가 필요 없다면 삭제
 - 👉 삭제하려는 이미지의 이름 혹은 ID를 확인

```
[home-ubuntu: snap]$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python3.8	latest	af20fda15fdd	50 minutes ago	934MB
python	latest	3189819ced3e	6 days ago	934MB

👉 이미지 삭제 후 삭제 확인

```
[home-ubuntu: snap]$ docker rmi af20fda15fdd
Untagged: python3.8:latest
Deleted: sha256:af20fda15fdd21fb7519818cd2d5e0d0b3704fc4405df5369a3610d4ebf0e6c7
Deleted: sha256:6b5f93790cf8a8d5472278fe3cbc3b456bdb024c964c5b0b5de6d5731e145395
[home-ubuntu: snap]$ docker images
```

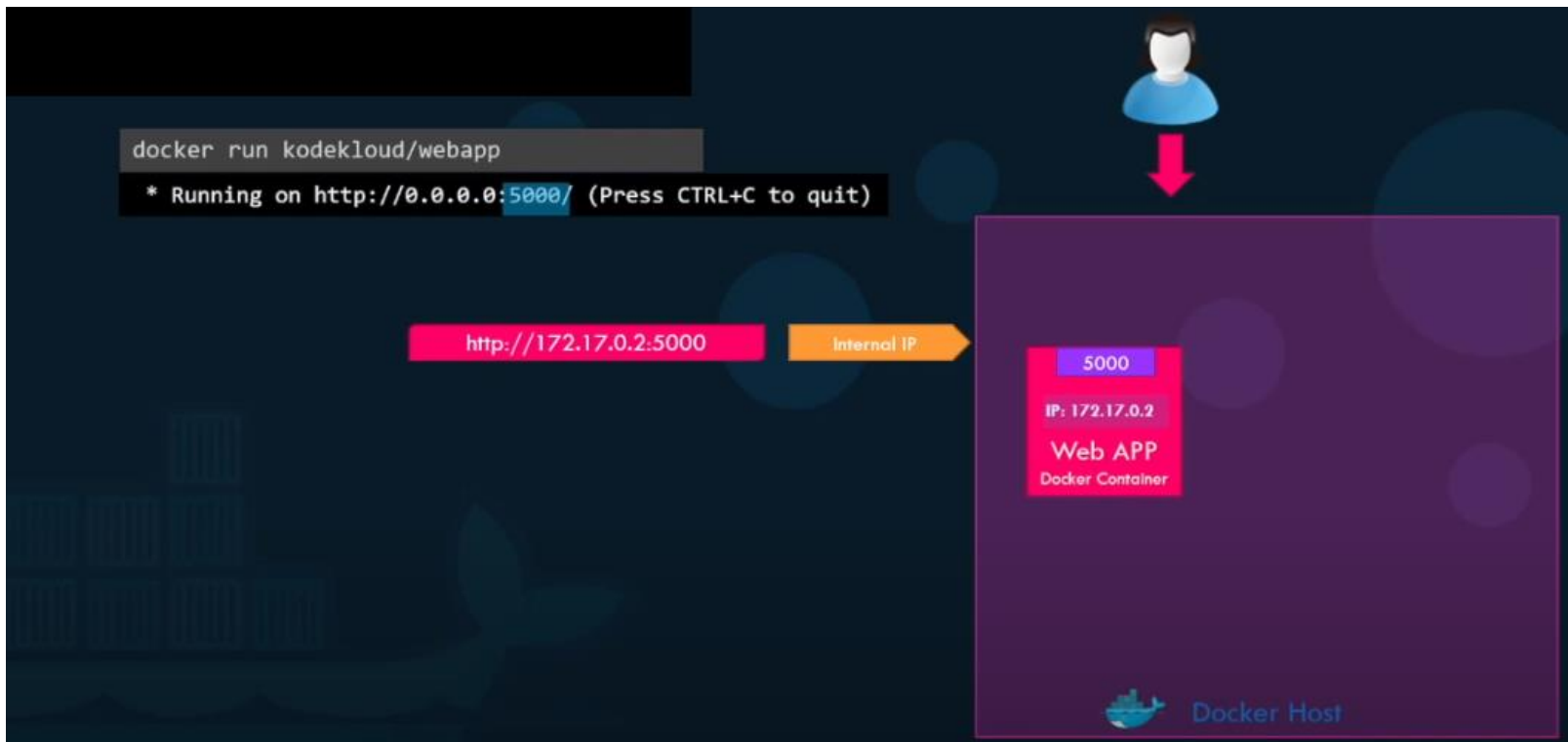
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python	latest	3189819ced3e	6 days ago	934MB

기타 명령어 옵션

run - Port mapping

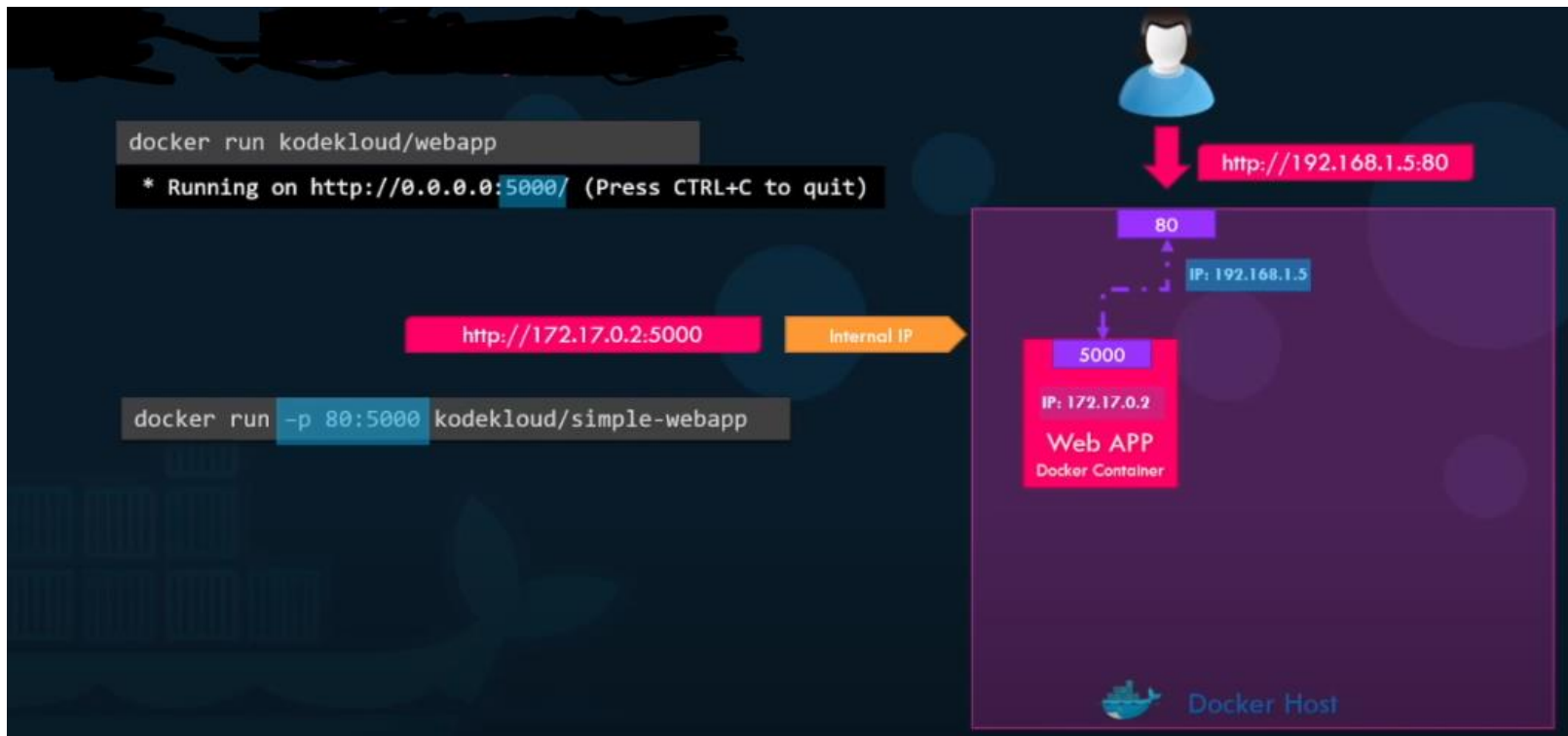
- 도커에서 webapp을 실행
 - 도커의 internal ip : 172.17.0.2 (포트 : 5000)
- 172.17.0.2 는 도커 내부 ip 이기 때문에 도커 밖에서(host) 도커 webapp에 접근할 수 없음

👉 Port mapping 이 필요함



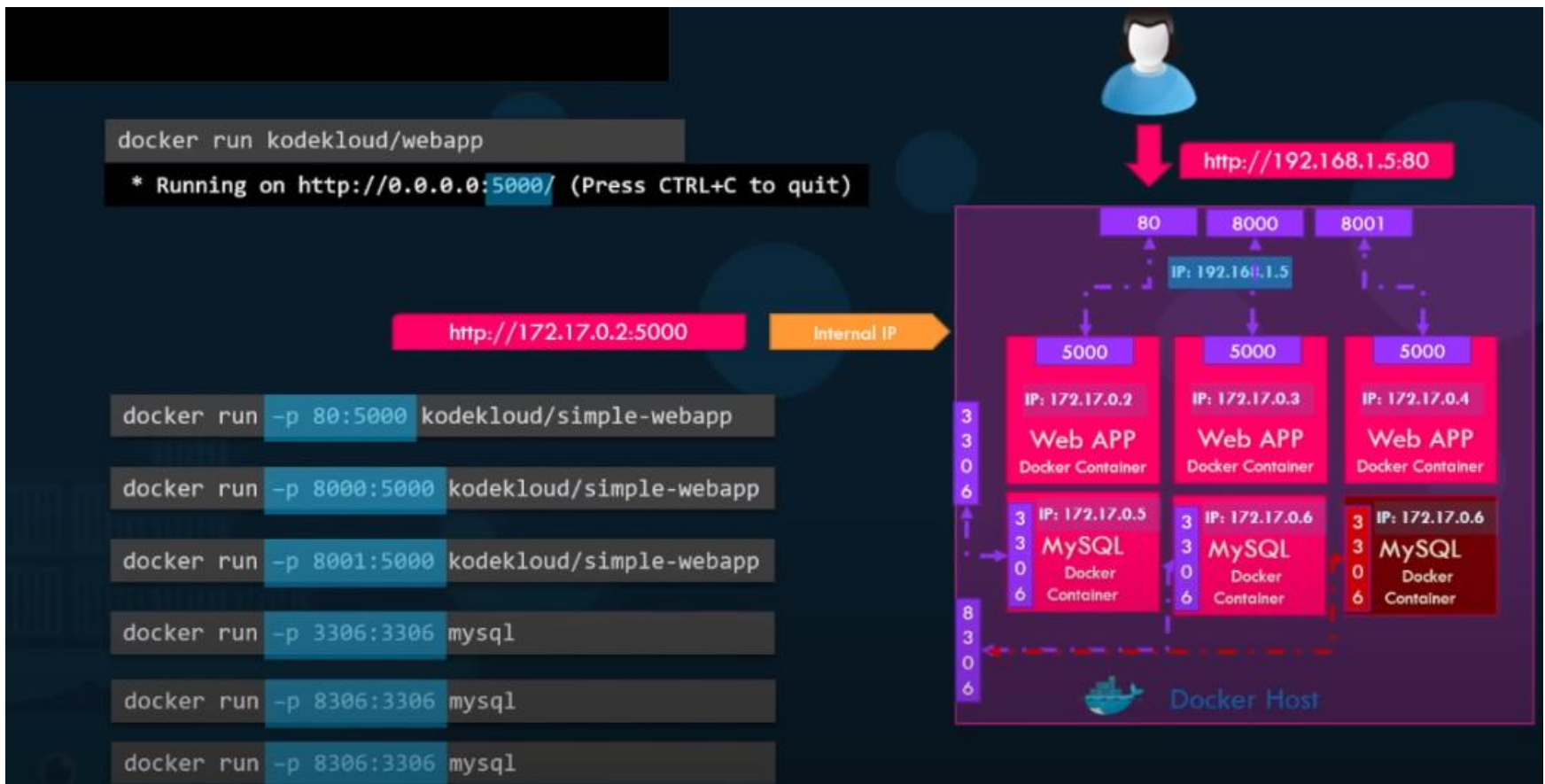
run - port mapping

- host ip + port (192.168.1.5:80) 를 도커내부 ip + port (172.17.0.2:5000) 으로 mapping 하는 방법
 - 👉 `docker run -p 80:5000 kodecloud/simple-webapp`
- host 브라우저에서 `http://192.168.1.5:80` 실행하면 도커 webapp에 접근가능해짐
 - 👉 host 80 포트의 모든 트래픽이 도커 5000 포트로 라우팅됨



run - port mapping

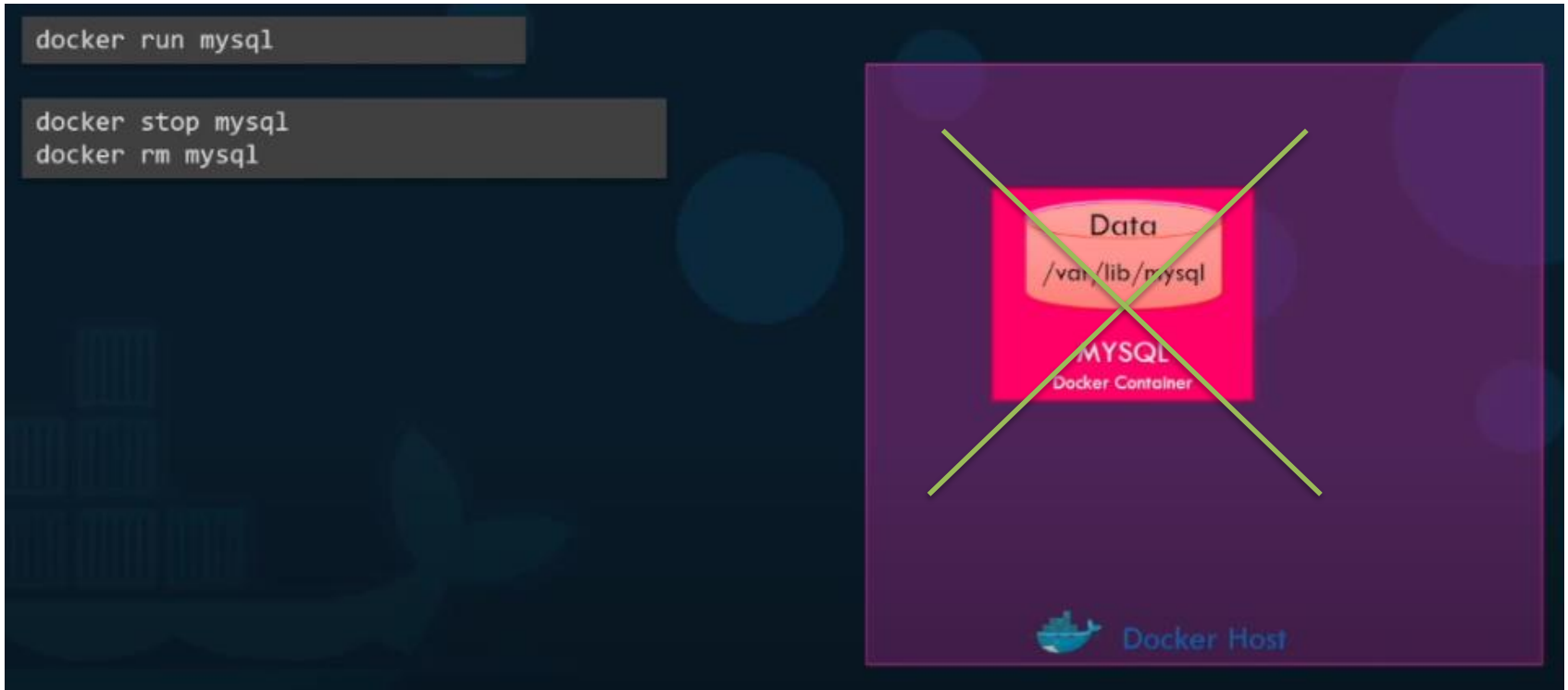
- 다른 host 포트를 사용하여 도커 내부에 webapp을 여러 instance로 실행가능
- MySQL 과 같은 다른 application도 port mapping을 통해 사용가능함



run – Volume mapping

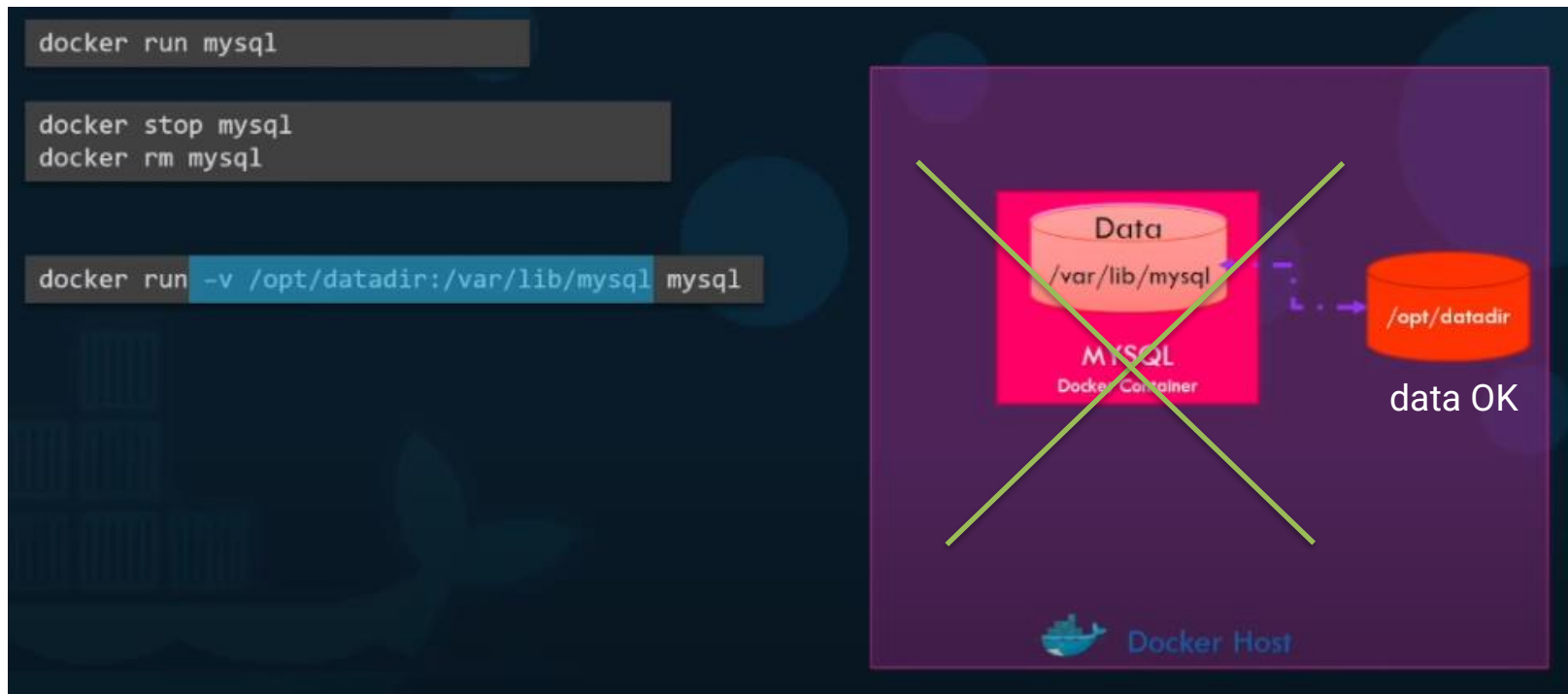
- 도커 컨테이너에서 MYSQL을 실행하여 데이터를 수정한 후,
컨테이너를 삭제하면 데이터 수정한 작업내용이 같이 삭제되는 문제가 발생함

👉 volume mapping 으로 해결 가능!



run – Volume mapping

- 도커 컨테이너 실행할 때 " -v " 옵션으로 volume을 지정할 수 있음
 - 👉 도커 컨테이너의 특정 폴더를 도커 밖에 있는 host의 특정 폴더에 항상 복제 유지됨
 - 👉 도커 컨테이너가 삭제되어도 host 폴더의 작업 내용은 삭제되지 않음



Inspect Container

- inspect 명령
 - 특정 컨테이너의 세부 정보 확인
 - 👉 "docker inspect" 는 모든 컨테이너의 간략한 정보를 제공

```
docker inspect blissful_hopper

[
  {
    "Id": "35505f7810d17291261a43391d4b6c0846594d415ce4f4d0a6ffbf9cc5109048",
    "Name": "/blissful_hopper",
    "Path": "python",
    "Args": [
      "app.py"
    ],
    "State": {
      "Status": "running",
      "Running": true,
    },
    "Mounts": [],
    "Config": {
      "Entrypoint": [
        "python",
        "app.py"
      ],
    },
    "NetworkSettings": {...}
  }
]
```

Container Logs

- 특정 컨테이너의 log 내용 확인 가능

```
▶ docker logs blissful_hopper
```

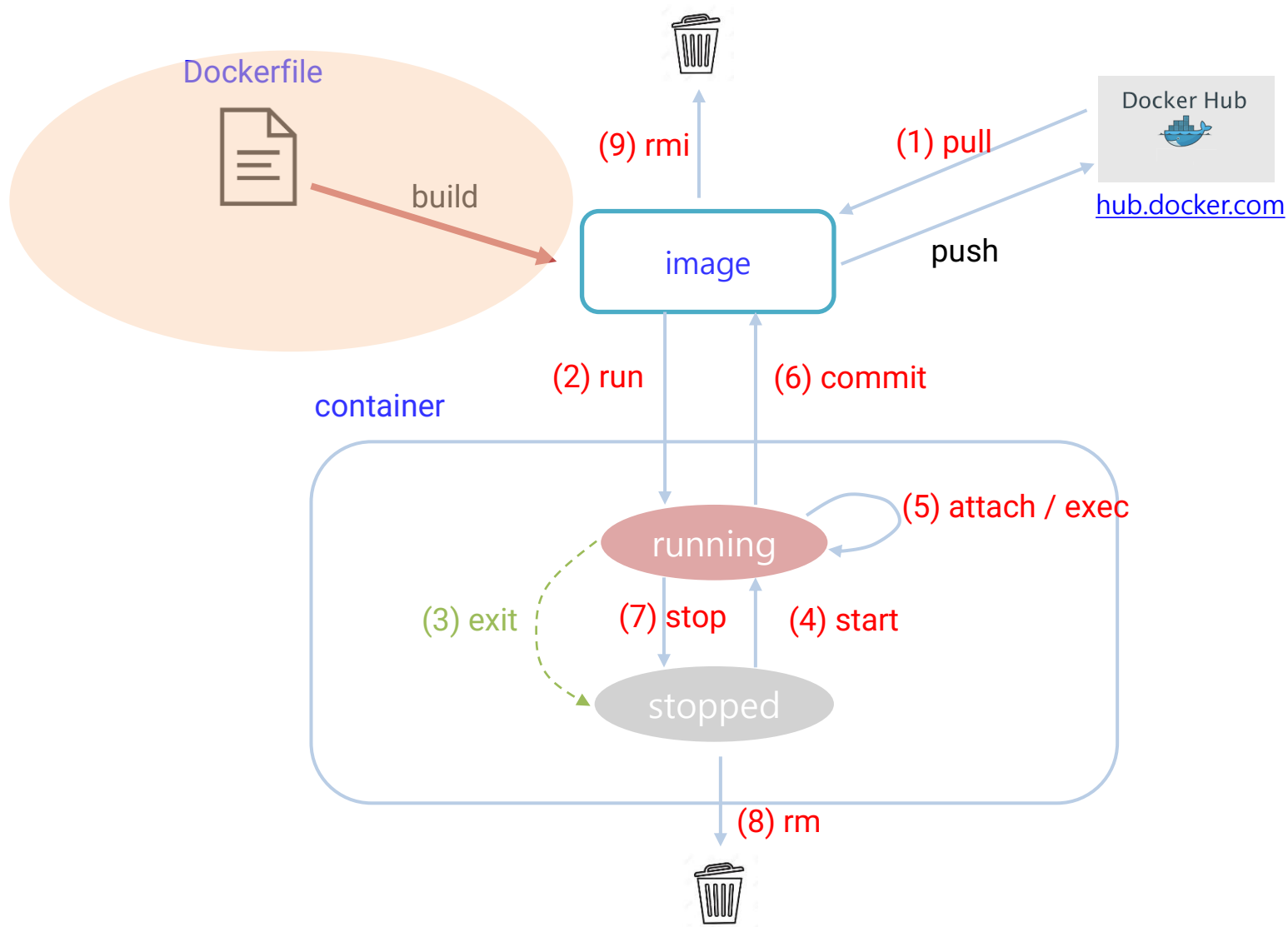
```
This is a sample web application that displays a colored background.  
A color can be specified in two ways.
```

1. As a command line argument with `--color` as the argument. Accepts one of red,green,blue,blue2,pink,darkblue
 2. As an Environment variable `APP_COLOR`. Accepts one of red,green,blue,blue2,pink,darkblue
 3. If none of the above then a random color is picked from the above list.
- Note: Command line argument precedes over environment variable.

```
No command line argument or environment variable. Picking a Random Color =blue  
* Serving Flask app "app" (lazy loading)  
* Environment: production  
  WARNING: Do not use the development server in a production environment.  
  Use a production WSGI server instead.  
* Debug mode: off  
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

Docker Images & Dockerfile

Dockerfile



Dockerfile

- 모든 Dockerfile은 첫번째 줄이 'FROM' 문장으로 시작함
 - 모든 docker image는 docker hub 에 있는 image를 이용하여 만들어지기 때문
 - 아래 예시는 우분투 image를 이용하여 새로운 이미지를 생성함을 나타냄
- RUN, COPY, ENTRYPOINT 등의 Dockerfile 명령 등이 있음
- ENTRYPOINT : 도커 image가 컨테이너로 실행될 때 자동으로 실행되어야 하는 명령을 지정

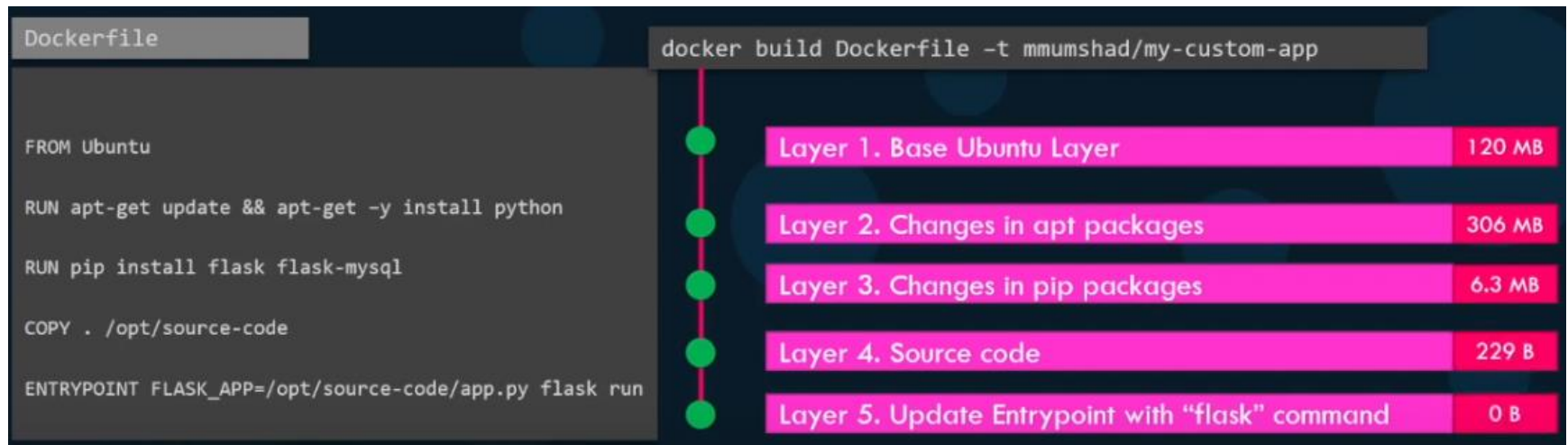
The diagram shows a Dockerfile with the following commands and their corresponding annotations:

```
Dockerfile
FROM Ubuntu
RUN apt-get update
RUN apt-get install python
RUN pip install flask
RUN pip install flask-mysql
COPY . /opt/source-code
ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

- FROM Ubuntu**: Start from a base OS or another image
- RUN apt-get update**
RUN apt-get install python: Install all dependencies
- RUN pip install flask**
RUN pip install flask-mysql
- COPY . /opt/source-code**: Copy source code
- ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run**: Specify Entrypoint

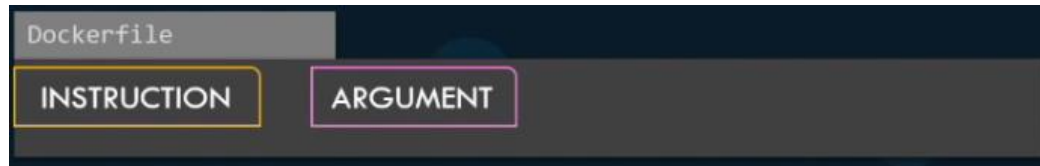
Layered architecture

- Dockfile의 각 줄은 layer 형태로 설치 실행됨
- 각 layer는 이전 layer에서 변경되는 부분(changes)만 실행하여 저장함
- 각 layer 설치결과는 도커가 캐쉬 형태로 보관하고 있음
 - 👉 도커 빌드 중에 에러가 발생하거나 새로운 패키지를 추가로 설치할 때 캐쉬를 이용하기 때문에 도커 빌드를 반복해도 설치 속도가 빠름



Dockerfile

- Dockerfile의 각 줄은 Instruction + Argument 로 구성됨



```
Dockerfile
FROM Ubuntu
RUN apt-get update
RUN apt-get install python
RUN pip install flask
RUN pip install flask-mysql
COPY . /opt/source-code
ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

A diagram illustrating a Dockerfile with instructions and arguments highlighted. The instructions are: FROM, RUN, COPY, and ENTRYPOINT. The arguments are: Ubuntu, apt-get update, apt-get install python, pip install flask, pip install flask-mysql, . /opt/source-code, and FLASK_APP=/opt/source-code/app.py flask run.

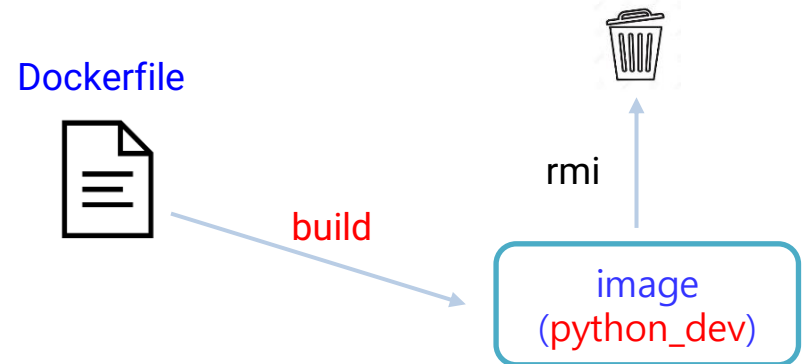
image 생성하기

```
docker build [options] [Dockerfile path]
```

```
$ docker build -t python_dev .      (-t : tag)
```

- Dockerfile을 Dockerfile path에서 찾을
- Dockerfile에 명시된 명령을 실행하여 image를 빌드
- 이미지 이름은 -t (--tag) 옵션으로 지정
 - 예시에서 build가 완료되면 'python_dev' 이미지가 생성됨

→ 'docker images' 명령으로 이미지 확인 가능



Dockerfile 명령

Command	의미
FROM	base image 설정
RUN	지정된 명령 실행
ENTRYPOINT	컨테이너를 실행하는 명령 설정
CMD	컨테이너가 실행될 때 실행되는 명령 설정(can be overwritten)
COPY	호스트 컴퓨터에서 컨테이너로 파일/폴더를 복사
ADD	COPY + unzip / download from URL
ENV	환경변수 설정
EXPOSE	호스트와 연결할 포트
WORKDIR	명령이 실행될 폴더 설정(RUN 명령 실행되는 폴더 위치 고정)

실습

- image 생성

- image 생성 절차를 'Dockerfile' 파일로 작성 (vi 혹은 nano 에디터 사용)

- nano 설치 : \$ apt update -y && apt install -y nano

(1) 폴더 생성	\$ mkdir python_app \$ cd python_app
(2) python_app 폴더에 Dockerfile 생성	FROM ubuntu:14.04 RUN apt-get update -y RUN apt-get install -y python-pip python-dev build-essential COPY . /app WORKDIR /app
(3) image 생성 (이미지 이름 : python_dev)	\$ docker build -t python_dev . (주의 : 끝에 '.' 입력해야 함)

```
[home-ubuntu:python_app]$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python_dev	latest	4a2c88b97756	23 minutes ago	392MB

이미지 이름 이미지 tag 이미지 ID

실습

- docker run

```
[home-ubuntu:python_app]$docker run -it --name python_develop python_dev /bin/bash  
root@0cd9f3b454fc:/app#
```

- docker 컨테이너에서 우분투 버전 확인
`lsb_release -a`

```
root@0cd9f3b454fc:/app# lsb_release -a  
No LSB modules are available.  
Distributor ID: Ubuntu  
Description:    Ubuntu 14.04.6 LTS  
Release:        14.04  
Codename:       trusty  
root@0cd9f3b454fc:/app#
```

우분투 버전

실습

- Dockerfile에서

- 우분투 버전을 16.04 로 변경하고
- 파이썬 패키지 'requests' 를 설치하도록 변경하여 image를 생성하세요

```
FROM ?????  
RUN apt-get update -y  
RUN apt-get install -y python-pip python-dev build-essential  
RUN ??????  
COPY . /app  
WORKDIR /app
```

👉 image 빌드 후 컨테이너 생성한 다음 아래 명령으로 정상적으로 빌드 되었는지 확인하세요

```
# more /etc/lsb-release
```

```
# pip freeze
```

CMD vs ENTRYPOINT

docker container 특징

- docker run ubuntu
 - ☞ 'ubuntu' 컨테이너 실행 후 바로 exit 하게됨 (docker ps 실행하면 empty list)
- 도커는 Virtual machine과는 다르게 OS를 실행하는게 아니고 특정 task 혹은 process를 실행하기 때문임. 예를 들면 웹서버, DB서버, 파이썬 프로그램 등
- task 실행이 끝나면 도커 컨테이너는 exit 하게됨

```
▶ docker run ubuntu
▶ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
▶ docker ps -a					
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
45aacca36850	ubuntu	"/bin/bash"	43 seconds ago	Exited (0) 41 seconds ago	

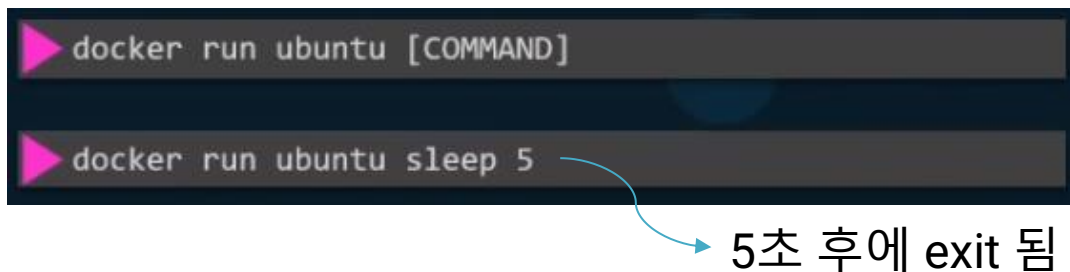
CMD

- 도커 컨테이너가 시작될 때 실행되어야 할 프로세스를 지정

```
# Install Nginx.  
RUN \  
    add-apt-repository -y ppa:nginx/stable && \  
    apt-get update && \  
    apt-get install -y nginx && \  
    rm -rf /var/lib/apt/lists/* && \  
    echo "\ndaemon off;" >> /etc/nginx/nginx.conf && \  
    chown -R www-data:www-data /var/lib/nginx  
  
# Define mountable directories.  
VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs", "/etc/nginx/conf.d"]  
  
# Define working directory.  
WORKDIR /etc/nginx  
  
# Define default command.  
CMD ["nginx"]
```

컨테이너 시작시 명령실행 방법

- 방법 1 : docker run 실행시
 - docker run ubuntu sleep 5



```
▶ docker run ubuntu [COMMAND]  
▶ docker run ubuntu sleep 5
```

5초 후에 exit 됨

- 방법2 : Dockerfile 에 명시

Dockerfile



```
FROM Ubuntu  
  
CMD sleep 5
```

CMD

- 첫번째 파라미터는 실행가능한 명령이어야 함

Dockerfile

```
FROM Ubuntu
```

```
CMD sleep 5
```

```
CMD command param1
```

```
CMD sleep 5
```

```
CMD ["command", "param1"]
```

```
CMD ["sleep", "5"]
```

```
CMD ["sleep 5"]
```



```
▶ docker build -t ubuntu-sleeper .
```

```
▶ docker run ubuntu-sleeper
```

ENTRYPOINT

- ENTRYPOINT : run instruction에 ENTRYPOINT instruction에 append 되어 실행됨
👉 (비교) CMD : run instruction에 CMD instruction 전체를 override하여 실행됨

The diagram illustrates the behavior of Docker containers using `CMD` and `ENTRYPOINT` instructions.

Scenario 1: CMD sleep 5

- FROM Ubuntu
- CMD sleep 5
- Command at Startup: sleep 10
- docker run ubuntu-sleeper sleep 10

Scenario 2: ENTRYPOINT ["sleep"]

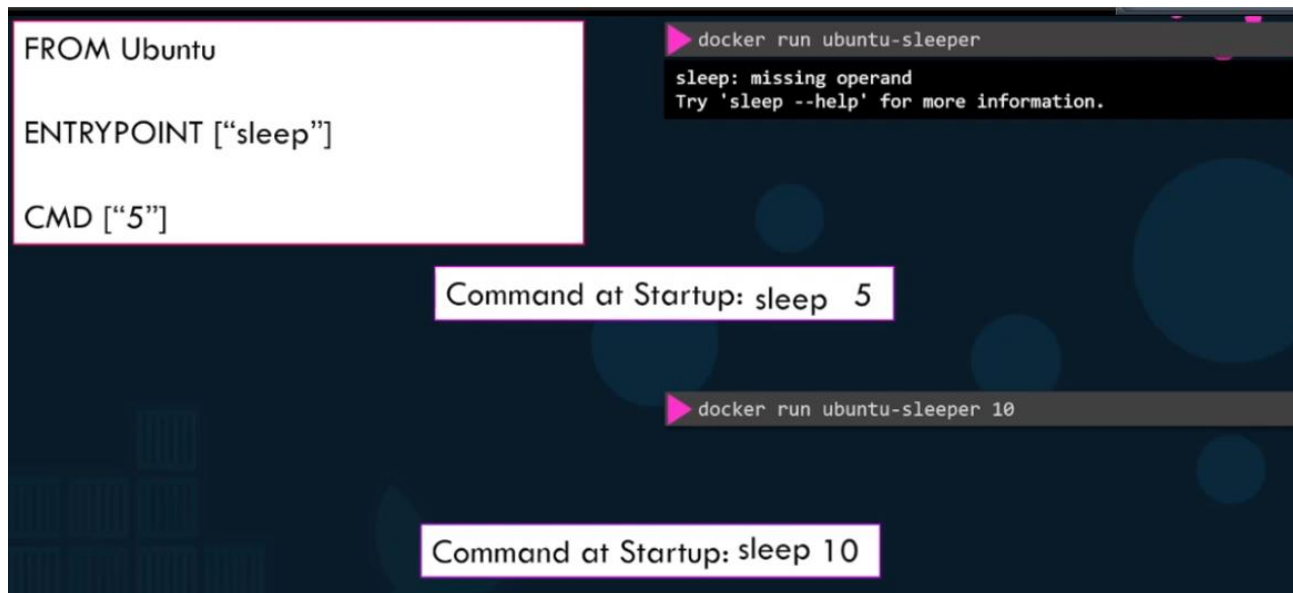
- FROM Ubuntu
- ENTRYPOINT ["sleep"]
- Command at Startup: sleep 10
- docker run ubuntu-sleeper 10

Scenario 3: ENTRYPOINT ["sleep"]

- FROM Ubuntu
- ENTRYPOINT ["sleep"]
- Command at Startup: sleep
- docker run ubuntu-sleeper
- sleep: missing operand
Try 'sleep --help' for more information.

ENTRYPOINT + CMD

- Dockerfile에 ENTRYPOINT와 CMD를 병행하여 사용하면 편리
 - 👉 ENTRYPOINT에 실행해야 하는 명령을 설정하고 CMD는 파라미터 설정
(예시) `docker run ubuntu-sleeper` : 5초 sleep
`docker run ubuntu-sleeper 10` : 10초 sleep



실습

- 편집기

Nano editor	\$ nano test.txt
-------------	------------------

- Project 폴더 만들기

(1) host 컴퓨터에 project 폴더 생성	\$ mkdir hello_docker_flask \$ cd hello_docker_flask
-----------------------------	---

- 필요한 package 설정하기

(2) requirements.txt 파일 생성	flask flask_restful
----------------------------	------------------------

실습

- application 구현

(3) app.py 파일 생성

```
# app.py - a minimal flask api using flask_restful
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(HelloWorld, '/')

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```


실습

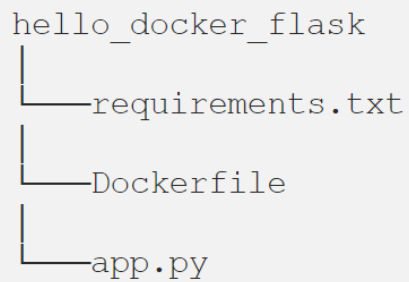
- Docker image 정의

(4) Dockerfile 생성

```
FROM python:3.8
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["app.py"]
```

실습

- hello_docker_flask 폴더에 3개 파일 생성되었는지 확인



```
hello_docker_flask
├── requirements.txt
├── Dockerfile
└── app.py
```

실습

- flask docker image 생성

(5) image 생성
(이미지 이름 : my_docker_flask)

```
$ docker build -t my_docker_flask .
```

```
Removing intermediate container 754b974840bf
--> 47dadccfcc3d
Step 5/6 : ENTRYPOINT ["python"]
--> Running in 333fe7b5e2ad
Removing intermediate container 333fe7b5e2ad
--> b364b78835e8
Step 6/6 : CMD ["app.py"]
--> Running in a02e63fc7910
Removing intermediate container a02e63fc7910
--> 6b5287995a9d
Successfully built 6b5287995a9d
Successfully tagged my_docker_flask:latest
$
```

실습

(6) image 생성 확인

\$ docker images

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my_docker_flask     latest             6b5287995a9d       About a minute ago 909MB
```

(7) docker run

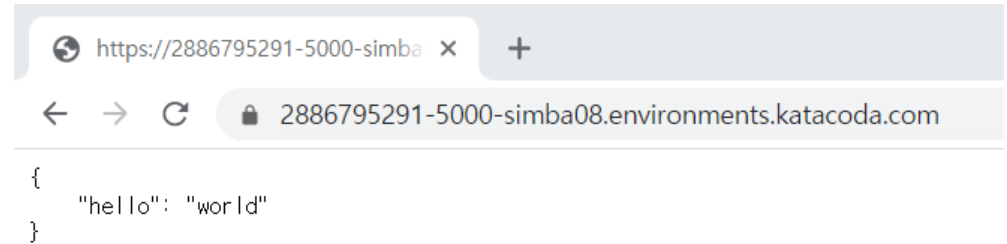
\$ docker run -p 5000:5000 my_docker_flask

```
$ docker run -p 5000:5000 my_docker_flask:latest
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 289-329-486
```

실습

(8) 브라우저에서 url 입력

http://localhost:5000



→ 본 실습환경에서는 아래 순서로 실행

The image illustrates a four-step process to view a service port:

- Step 1:** In the JupyterLab interface, click the menu icon (three dots) in the top right corner of the terminal area.
- Step 2:** From the dropdown menu, select the 'View Port' option.
- Step 3:** In the 'View Port' dialog box, enter the port number '5000' into the input field.
- Step 4:** Click the 'Open Port' button to open a new browser tab for the service.

Docker Storage

File system

- 도커 설치하면 /var/lib/docker 폴더가 생성됨
- 도커의 모든 파일은 /var/lib/docker 폴더와 서브폴더에 저장되어 관리됨

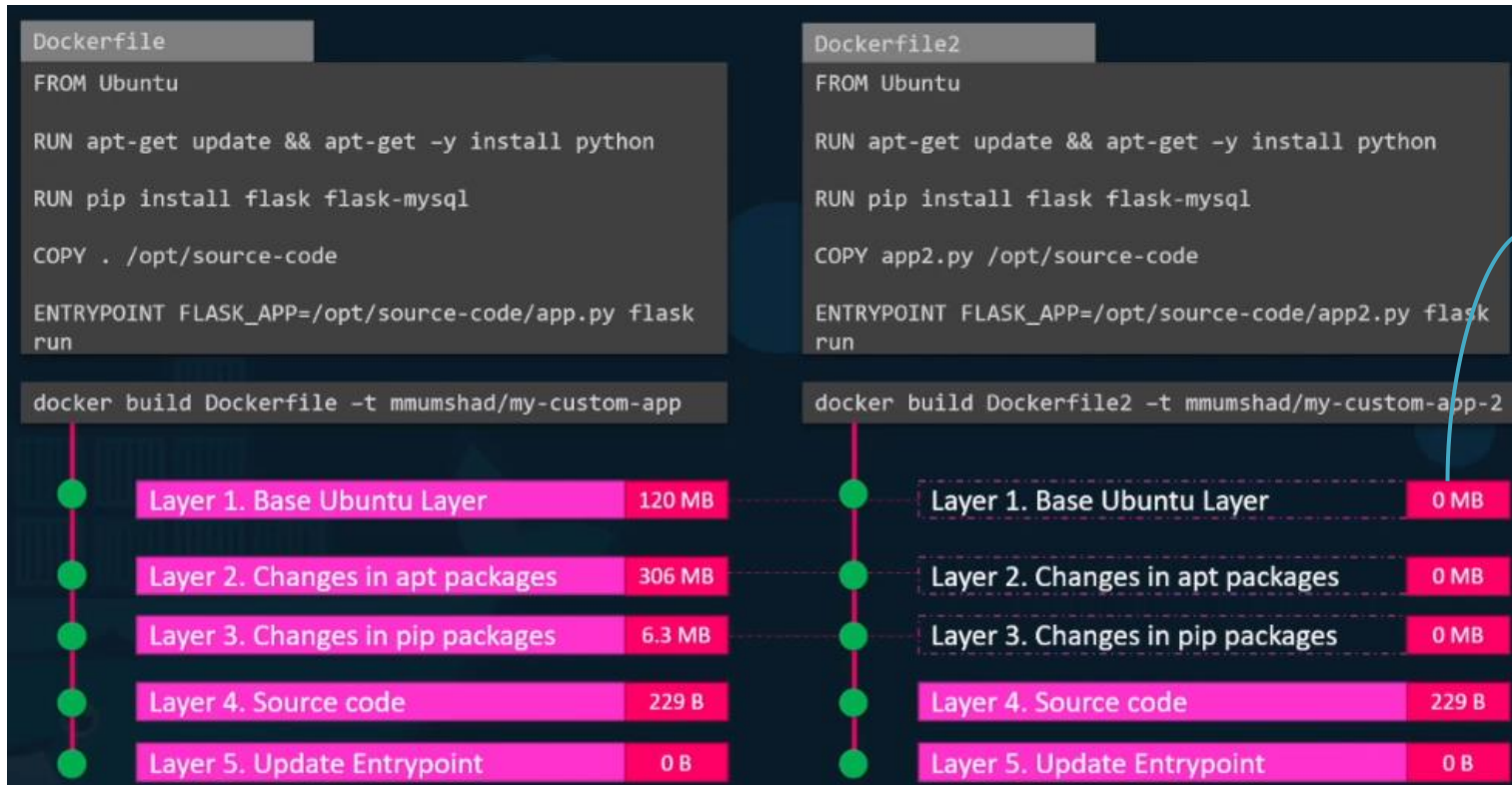


Layered architecture(1)

- layered architecture 장점

- 동일한 빌드 작업을 하지 않기 때문에 도커 image를 빠르게 생성할 수 있음
- 디스크 용량 절약

중복되는 layer는 새로 빌드
하지 않고 캐시에서 가져옴



Layered architecture(2)

- 'docker build' 에 의해 생성된 이미지는 Read Only 이며 변경할 수 없음
- 이미지를 변경하려면 다시 빌드해야 함



Layered architecture(3)

- 'docker build' 를 통해 도커 이미지가 생성되면 'docker run'으로 도커 컨테이너를 생성할 수 있음
- 도커 컨테이너는 writable layer 임
 - ☞ 컨테이너에서 생성되는 데이터를 저장할 수 있음
 - ☞ 단, 컨테이너가 삭제되면 해당 데이터도 모두 삭제됨에 주의!!



COPY-ON-WRITE

- 도커 이미지는 Read Only 이므로 도커 이미지에 있는 코드/데이터를 변경하려면 컨테이너에 복사해서 변경해야 함

👉 COPY-ON-WRITE

👉 컨테이너가 삭제되면 컨테이너에서 수정한 app.py와 temp.txt 파일이 같이 삭제됨

👉 'volumes' 을 통해 삭제 방지할 수 있음



Docker Compose

Docker-compose

- 호스트 컴퓨터에서 여러개의 도커 컨테이너를 사용하여 application을 실행하기 위한 도구
 - YAML 파일로 application 실행 방법을 정의
 - 컨테이너를 build 하고 deploy하는 방법 설정
- ➔ 여러 컨테이너를 연결하여 서비스를 실행할 때 사용

Multi Container Applications



```
1  version: '3'
2  services:
3    web:
4      build: .
5      ports:
6        - "5000:5000"
7      volumes:
8        - ./code
9      depends_on:
10       - redis
11     redis:
12       image: redis
13
```

docker-compose.yml

```
$ docker run -it -v /compose_flask:/code -p 5000:5000 ...
```

docker-compose.yml

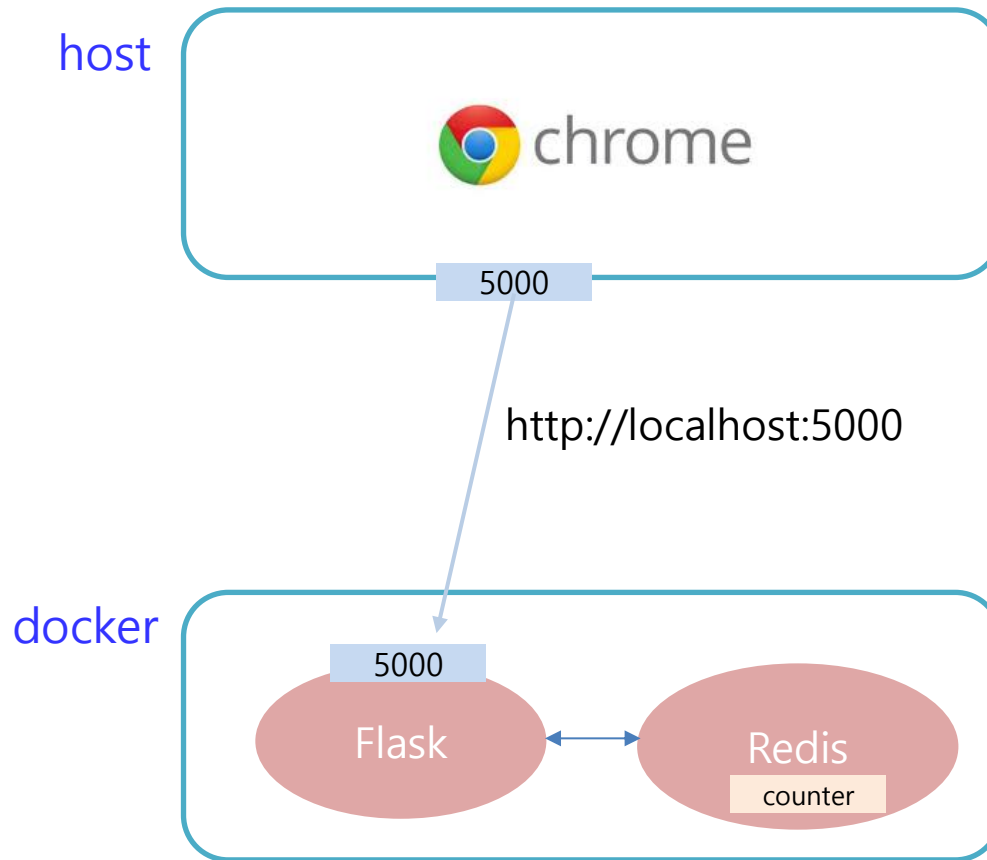
```
1  version: '3'
2  services:
3    web:
4      build: .
5      ports:
6        - "5000:5000"
7      volumes:
8        - ./code
9      depends_on:
10       - redis
11   redis:
12     image: redis
13
```

Docker-compose 실행 단계

- step1 : Dockerfile 작성
- step2 : docker-compose.yml 작성
- step3 : 'docker-compse up' 명령 실행

Docker-compose 예시

브라우저를 refresh 할 때마다 조회횟수 증가시켜 Redis DB에 저장하고
조회횟수를 브라우저에 표시



실습

- Project 폴더 만들기

(1) host 컴퓨터에 project 폴더 생성	<pre>\$ mkdir compose_flask \$ cd compose_flask</pre>
-----------------------------	--

- 필요한 package 설정하기

(2) requirements.txt 파일 생성	<pre>flask redis</pre>
----------------------------	----------------------------

실습

- application 구현

(3) app.py 파일 생성

```
# -*- coding: utf-8 -*-  
# compose_flask/app.py  
from flask import Flask  
from redis import Redis  
  
app = Flask(__name__)  
redis = Redis(host='redis', port=6379)  
  
@app.route('/')  
def hello():  
    redis.incr('hits')  
    return 'Flask demo 조회 횟수 = %s time(s).' % redis.get('hits')  
  
if __name__ == "__main__":  
    app.run(host="0.0.0.0", debug=True)
```


실습

- Docker image와 Service 정의

(4) Dockerfile 생성	FROM python:3.8 COPY . /code WORKDIR /code RUN pip install -r requirements.txt CMD python app.py
(5) docker-compose.yml 파일 생성	<pre>version: '3' services: web: build: . ports: - "5000:5000" volumes: - ./code depends_on: - redis redis: image: redis</pre> 

실습

- compose_flask 폴더에 4개 파일 생성되었는지 확인



app.py
docker-compose.yml
Dockerfile
requirements.txt

0

실습 : docker-compose 설치

- docker-compose 설치

1. docker-compose 다운로드

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

2. docker-compose 실행 권한 부여

```
sudo chmod +x /usr/local/bin/docker-compose
```

실습

- application(service) 실행

(6) compose_flask 폴더에서
docker-compose up 실행
(이때 도커 이미지가 자동으로 build됨)

\$ docker-compose up

```
[home-ubuntu:compose_flask]$docker-compose up
Creating network "compose_flask_default" with the default driver
Creating compose_flask_redis_1 ... done
Creating compose_flask_web_1 ... done

Attaching to compose_flask_redis_1, compose_flask_web_1
redis_1 | 1:C 29 Jul 2020 13:58:32.445 # o000o000o000o Redis is starting o000o000o000o
redis_1 | 1:C 29 Jul 2020 13:58:32.445 # Redis version=6.0.6, bits=64, commit=00000000, modified=
redis_1 | 0, pid=1, just started
redis_1 | 1:C 29 Jul 2020 13:58:32.445 # Warning: no config file specified, using the default con
redis_1 | fig. In order to specify a config file use redis-server /path/to/redis.conf
redis_1 | 1:M 29 Jul 2020 13:58:32.446 * Running mode=standalone, port=6379.
redis_1 | 1:M 29 Jul 2020 13:58:32.446 # WARNING: The TCP backlog setting of 511 cannot be enforc
redis_1 | ed because /proc/sys/net/core/somaxconn is set to the lower value of 128.
redis_1 | 1:M 29 Jul 2020 13:58:32.446 # Server initialized
redis_1 | 1:M 29 Jul 2020 13:58:32.446 # WARNING overcommit_memory is set to 0! Background save m
redis_1 | ay fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysct
redis_1 | l.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
redis_1 | 1:M 29 Jul 2020 13:58:32.446 # WARNING you have Transparent Huge Pages (THP) support en
redis_1 | abled in your kernel. This will create latency and memory usage issues with Redis. To fix this iss
redis_1 | ue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it
redis_1 | to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after
redis_1 | THP is disabled.
redis_1 | 1:M 29 Jul 2020 13:58:32.446 * Ready to accept connections
web_1 | * Serving Flask app "app" (lazy loading)
web_1 | * Environment: production
web_1 | WARNING: This is a development server. Do not use it in a production deployment.
web_1 | Use a production WSGI server instead.
web_1 | * Debug mode: on
web_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
web_1 | * Restarting with stat
web_1 | * Debugger is active!
web_1 | * Debugger PIN: 468-643-615
```

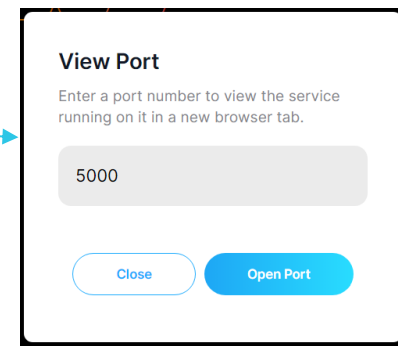
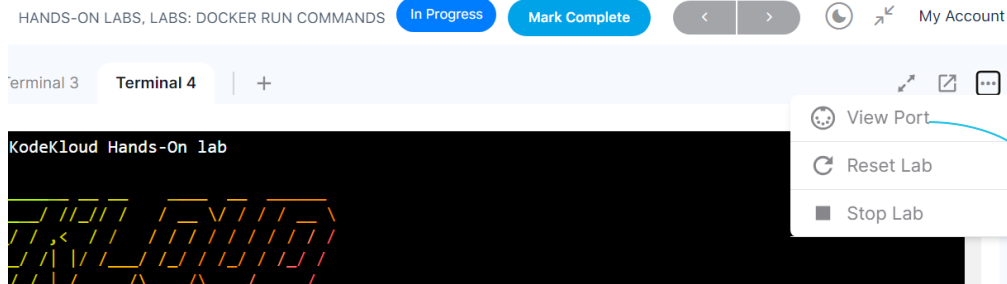
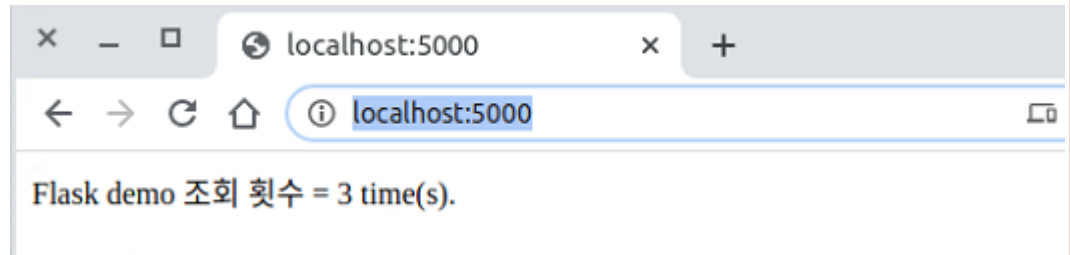
실습

- 외부에서 application 조회

(7) 브라우저에서 url 입력

http://localhost:5000

브라우저를 refresh 할 때마다 조회
횟수 증가됨



실습

- docker 컨테이너 확인

(8) 터미널을 추가로 열어서 docker 컨테이너 확인

\$ docker ps

\$ docker-compose ps

```
[home-ubuntu:compose_flask]$docker ps
CONTAINER ID   IMAGE               COMMAND                  CREATED        STATUS        PORTS                               NAMES
9d24f63c1623   compose_flask_web   "/bin/sh -c 'python ..." 4 minutes ago  Up 4 minutes  0.0.0.0:5000->5000/tcp              compose_flask_web_1
26cb449ffc31   redis              "docker-entrypoint.s..." 4 minutes ago  Up 4 minutes  6379/tcp                            compose_flask_redis_1
```

```
[home-ubuntu:compose_flask]$docker-compose ps
              Name                  Command                  State        Ports
-----
compose_flask_redis_1  docker-entrypoint.sh redis ...  Up          6379/tcp
compose_flask_web_1    /bin/sh -c python app.py        Up          0.0.0.0:5000->5000/tcp
[home-ubuntu:compose_flask]$
```

실습

- application(service) 중지

(9) docker 컨테이너 모두 중지

```
$ docker-compose down
```

```
$ docker ps
```

```
[home-ubuntu:compose_flask]$docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
[home-ubuntu:compose_flask]$
[home-ubuntu:compose_flask]$
[home-ubuntu:compose_flask]$
[home-ubuntu:compose_flask]$
[home-ubuntu:compose_flask]$docker-compose ps
Name      Command      State      Ports
-----
[home-ubuntu:compose_flask]$
```

실습

- 아래와 같이 내용을 수정한 후 docker-compose로 application 을 동작시키세요

변경전	변경후
파이썬 2.7	파이썬 3.6
Flask demo 조회 횟수 = 1 time(s)	Flask demo count = 1 time(s)

자주 사용하는 패턴

- 패턴 1
 - docker build
 - docker run
 - ctrl + p, q
 - docker attach
- 패턴 2
 - docker-compose up

참고자료

Docker Tutorial for Beginners

<https://www.youtube.com/watch?v=fqMOX6JJhGo>

도커 무작정 따라하기

<https://www.slideshare.net/pyrasis/docker-fordummies-44424016>

수고하셨습니다!