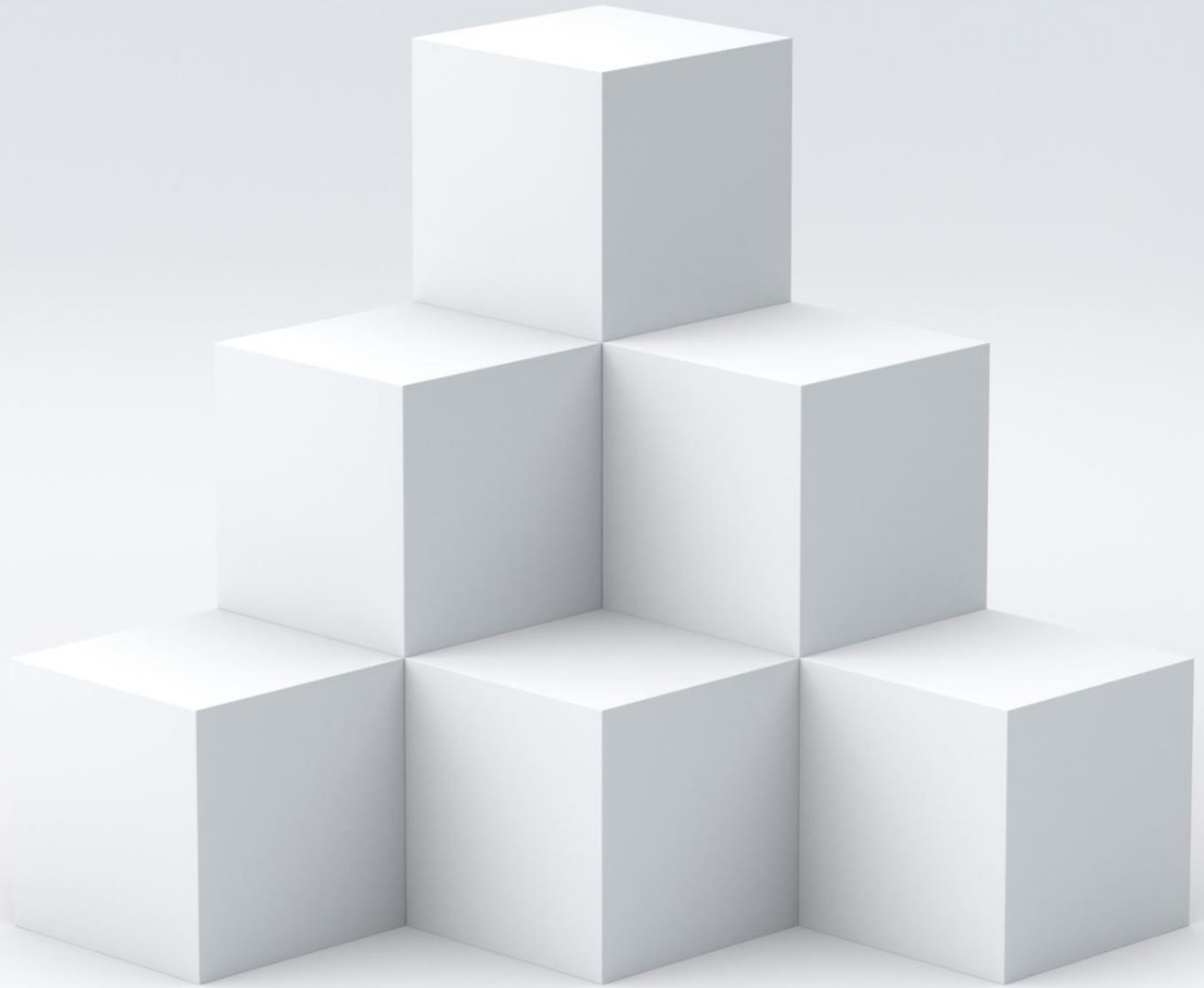
Software Engineering

- dockerfile -



Professor Han-gyoo Kim

2022

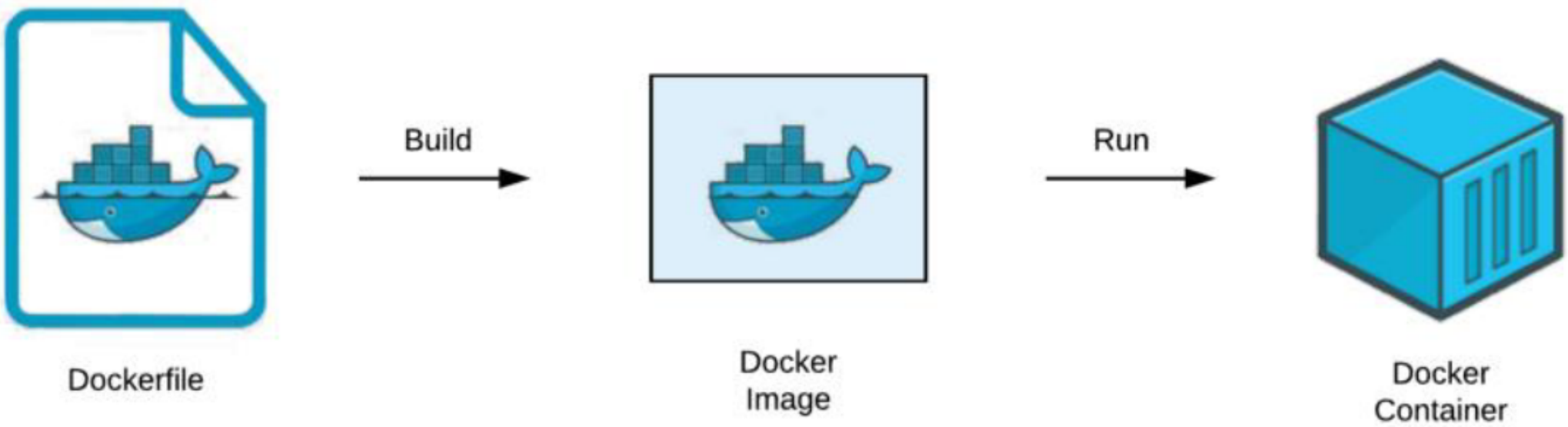
Dockerfile

= docker image를 만들기 위해 내리는 command들을 모아 둔 text 파일

docker **build** command 를 사용하여 docker 파일로부터 container 생성

docker command들을 입력하여 container 를 수동으로 생성하는 대신에 dockerfile을 작성하고 이후 build command 를 통하여 모든 과정을 수행하여 해당 container 생성

Dockerfile



Dockerfile

- 지금까지의 docker container 작업 과정은

  image –(run)-> container –(exec)-> inside the container –(do jobs)->(exit)->(commit)->save a new image

⇒Dockerfile에 이러한 수작업 명령을 기재하여 자동화함

⇒Dockerfile은 docker image를 만들기 위한 모든 순차적인 명령들을 포함한 text 파일

⇒Dockerfile 기본 구문

FROM, ADD, RUN, CMD, ENTRYPOINT, ENV

- FROM – base image 지정
- ADD <source> <destination in container> - 파일들 복사 (COPY)
  - ADD vs COPY 서로 비슷, ADD는 <source>로 url을 사용할 수 있음
- RUN – base image 위에 추가 SW 계층 설치
- CMD – container 안에서 실행하려는 command
- ENETRYPOINT – CMD 와 유사,

  Docker는 /bin/sh –c 를 default entrypoint로 함 그러나 –c (command 입력 option) 뒤에 입력할 command는 default로 지정된 것이 없음

  우리가 사용하던 docker run –it ubuntu bash 의 예에서는 entrypoint가 /bin/sh –c, image는 ubuntu, command는 bash

  docker run -it ubuntu <cmd>. <cmd> 는 the entrypoint의 parameter

  CMD에 비해 ENTRYPOINT는 내가 만든 image를 다른 사람이 run할 때 run이 시작될 때 다른 command를 입력하여 dockerfile의 CMD에 지정된 명령을 생략할 수 없게 할 때 사용
- ENV – container에서 사용할 environment 변수 지정

Dockerfile 예

```
FROM ubuntu
## for apt to be noninteractive
ENV DEBIAN_FRONTEND noninteractive
ENV DEBCONF_NONINTERACTIVE_SEEN true
RUN apt-get update
RUN apt-get install –y apache2
ADD . /var/www/html
ENTRYPOINT apache2ctl –D FOREGROUND
ENV env_var_name <변수 이름>
```

docker build . –t <image name> // dockerfile로부터 image 만들기

docker run -it -p <host port #:container port#> -d <image>
docker run -it -p <port #:port #> -v <local dir : mounting point
dir in container> -d <image>
docker run -it -p 81:80 –v ~/dockerfile:/var/www/html -d swedemo/test

Docker volume

- https://docs.docker.com/storage/volumes/
- Container는 cgroup 즉 container는 자신만의 OS 자원을 사용하며 다른 container의 OS와 별도의 자원, 즉 DRAM 영역, CPU 시간을 사용하도록 되어 있으며 디스크와 같은 persistent storage를 사용하지 않고 기본적인 in-memory 파일시스템을 사용 – **컨테이너는 read-only app이 기본**
- Container가 run 명령에 의해 시작될 때는 container의 filesystem은 read-only layer위에 read-write layer로 만들어진 virtual filesystem으로서 일단container를 시작한 후 만들어진 파일은 container가 동작하는 동안에는 우리에게 익숙한 persistent filesystem에 저장된 것처럼 사용되지만 실제로는persistent 장치(디스크)에 저장되는 것이 아니라 in-memory file system에 쓰이는 것이므로 container가 종료되면 사라짐
- 그래서 동일한 image로부터 container를 다시 만들어보면 예전에 filesystem에 저장했다고 생각한 데이터는 온데간데 없이 사라짐
- 이러한 단점(?)을 해소하기 위해 호스트 OS의 파일시스템을 mount 하여 container volume 기능이 제공되지만, container의 장점 중의 하나는 container들 사이에 filesystem을 통한 간섭이 없어서 보안이 뛰어나다는 것인데 volume 기능은 잘못된 코딩 또는 바이러스에 의해 이러한 장점을 훼손시킬 수 있음
- docker volume create <volume name>으로 호스트 기계의 /var/lib/docker/volumes에 <volume name>의 docker volume을 만들고 docker run --mount source=<volume name>, destination=<path in container> <image name>명령을 통해 원하는 docker volume을 마운트하여 container에서 호스트 기계의 persistent 스토리지에 데이터를 저장할 수 있고 이를 이용하여 여러 컨테이너들 사이에서 persistent한 데이터를 공유할 수 있음
- docker-compose 공부할 때 다시 공부함