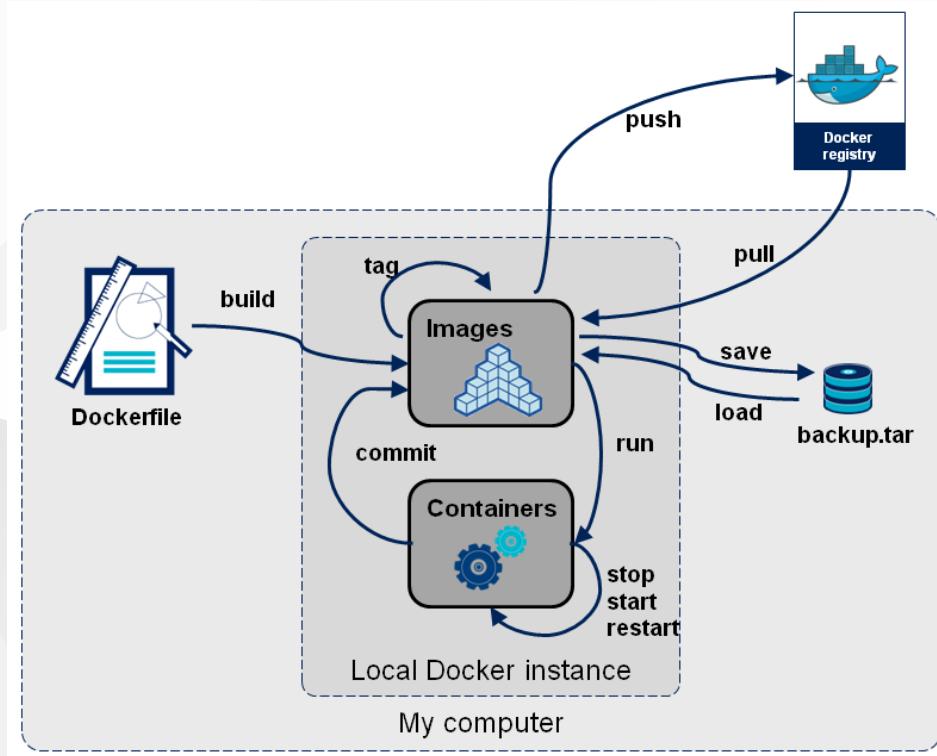


Contents

- **Dockerfile**
- **Instructions**
 - **FROM**
 - **LABEL**
 - **RUN**
 - **CMD , ENTRYPOINT**
 - **EXPOSE**
 - **ENV**
 - **COPY , ADD**
 - **USER**
 - **WORKDIR**
 - **VOLUME**



Dockerfile



Docker에서 이미지를 생성하는 방법은 다음과 같은 방법들이 있습니다.

- Dockerfile로 Build하기
- Container로부터 Commit하기
- Image로부터 Tag하기

Dockerfile

이중 가장 일반적이고 CI/CD로 활용되어지는 방법은 **Dockerfile**로 빌드해 이미지를 생성하는 방법입니다.

여기서 **Dockerfile**은 이미지를 생성하기 위해 필요한 **명령어(Instruction)**들을 순차적으로 나열한 Text문서입니다.
그리고, 이 **Dockerfile**을 이용해서 이미지를 빌드할 때는 **build context** 를 참조하게 됩니다.

Build context는 `docker build` 명령을 실행할 때 사용되는 파일들이 위치하는 디렉토리입니다.

사용법 (docker build)

일반적인 사용법은 다음과 같습니다.

```
$ Usage: docker build [OPTIONS] PATH | URL | -
```

e.g.) `$docker build -t my-image:v1.0.0 .`

그리고, 자주 사용되는 Option들은 아래와 같은 것들이 있습니다.

Options	Description
<code>-f, --file string</code>	Name of the Dockerfile (Default is 'PATH/Dockerfile')
<code>-t, --tag list</code>	Name and optionally a tag in the 'name:tag' format

Dockerfile

`docker build` 명령은 **Docker daemon**에 의해 실행됩니다. (CLI가 아님.)

빌드 프로세스에서 가장 먼저 하는 일은 전체 **Context**를 (재귀적으로) **Docker daemon**으로 보내는 것입니다. 그렇기 때문에, Build context에는 이미지 빌드에 필요한 파일들만 유지하는 것이 좋습니다.

Context는 PATH 또는 URL(git repository location)을 이용하여 지정함.

Instructions

Dockerfile의 형식은 아래와 같습니다.

```
# Comment  
INSTRUCTION arguments
```

e.g.) `RUN echo 'Hello docker'`

INSTRUCTION은 대소문자를 가리지는 않지만,
일반적으로는 arguments와 구분하기 위해서 모두 **대문자**로 표시합니다.

이제 이 **INSTRUCTION**으로 사용되는 것들을 하나씩 자세히 알아보겠습니다.

Instructions

FROM

Base image를 지정하는 Instruction으로, 지정된 Image를 Docker Hub 와 같은 Registry에서 Pull합니다. Base image를 지정할때는 `ubuntu:18.04` 처럼 Image명과 Tag까지 지정해주는것이 좋습니다.

| Tag가 생략되면 **latest tag** 를 사용하게 됩니다.

Syntax

```
FROM <image> [AS <name>]
```

```
FROM <image>[:<tag>] [AS <name>]
```

```
FROM <image>[@<digest>] [AS <name>]
```

Example

```
FROM ubuntu:18.04  
  
# Container에서 실행할 명령  
CMD ["/bin/echo", "hello docker"]
```

Dockerfile을 위와같이 작성한 다음 아래 명령어를 실행합니다.

Docker & Kubernetes - 05. Dockerfile

```
ubuntu@ip-10-0-1-14:~/app/temp$ docker build -t my-ubuntu:v1 .
Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM ubuntu:18.04
--> ad080923604a
Step 2/2 : CMD [ "/bin/echo", "hello docker" ]
--> Running in 228730d463bd
Removing intermediate container 228730d463bd
--> f34ba0754f11
Successfully built f34ba0754f11
Successfully tagged my-ubuntu:v1

ubuntu@ip-10-0-1-14:~/app/temp$ docker run my-ubuntu:v1
hello docker
```

Dockerfile에 작성한 대로 `ubuntu:18.04`를 Base image로 사용하여 my-ubuntu:v1 이미지를 만들니다.

`FROM`은 일반적으로 Dockerfile에서 가장 먼저(앞에) 사용되는 Instruction입니다.

그 보다 앞에 올 수 있는 Instruction이 하나 있는데, `ARG`가 그것입니다.

`ARG`는 아래 예제와 같이 FROM에서 사용될 값(변수)을 지정하는 데 사용됩니다.

```
ARG CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD /code/run-app
```

LABEL

`LABEL` instruction은 Label로 지정한 문자열을 이미지에 메타데이터로 추가합니다.

`LABEL` 은 key-value 쌍으로 작성하며 space를 포함시키기 위해서는 따옴표(" ")를, 이어쓰기를 위해서는 백슬래쉬(\)를 사용하면 됩니다.

Syntax

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

Example

```
FROM ubuntu:18.04

LABEL "com.example.vendor"="Samsung SDS"
LABEL multi.label1="value1" multi.label2="value2" other="value3"
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"

# Container에서 실행할 명령
CMD [ "/bin/echo", "hello docker" ]
```

Docker & Kubernetes - 05. Dockerfile

이렇게 추가된 Label은 `docker inspect` 명령어로 이미지나 컨테이너의 내용을 확인 할 수 있습니다.

다만 한 가지 주의할 것은 Base 이미지에 포함된 Label 값이 **상속된다**는 점이고, 만약 같은 Label값이 존재한다면 가장 최근에 적용된 Label값이 우선합니다.

앞의 Dockerfile로 만들어진 이미지나 컨테이너의 Label을 확인해보면 아래와 같습니다.

```
"Labels": {  
    "com.example.vendor": "Samsung SDS",  
    "multi.label1": "value1",  
    "multi.label2": "value2",  
    "other": "value3_modified"  
}
```

RUN

RUN Instruction은 Base image위의 새로운 layer에서 **Command**를 실행하는데 사용됩니다.
일반적으로 패키지를 설치할 때 자주 사용됩니다.

Syntax

shell form : command 로 입력받은 명령어는 셸에서 수행되며 디폴트로 리눅스에서는 `/bin/sh -c` 이 윈도우에서는 `cmd /S /C` 가 사용됩니다.

```
RUN <command>
```

exec form :

```
RUN ["executable", "param1", "param2"]
```

exec form은 JSON array로 파싱되므로, double-quotes(“) 를 이용해야 함.

Example

```
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'  
RUN ["/bin/bash", "-c", "echo hello"]  
RUN npm install --silent  
RUN apt-get update && apt-get install -y \  
    package-a \  
    package-b \  
    package-c
```

CMD

CMD Instruction은 Docker Container가 시작될때 실행 할 커맨드를 지정하는 지시자이며 아래와 같은 특징과 기능을 제공합니다.

- CMD의 주용도는 컨테이너를 실행할 때 사용할 **default 명령어를 설정**하는 것입니다. `docker run` 실행 시 실행할 커맨드를 주지 않으면 CMD로 지정한 default 명령이 실행됩니다.
- **ENTRYPOINT**의 파라미터를 설정할 수도 있습니다.
- **RUN** Instruction과 기능은 비슷하지만 차이점은 **CMD**는 image를 빌드할때 실행되는 것이 아니라 container가 시작될때 실행됩니다. 주로 docker image로 빌드된 application을 실행할때 사용됩니다.

Syntax

CMD instruction은 아래와 같이 3가지 포맷을 지원합니다.

- `CMD ["executable", "param1", "param2"]` (*exec form, this is the preferred form*)
- `CMD ["param1", "param2"]` (*as default parameters to ENTRYPPOINT*)
- `CMD command param1 param2` (*shell form*)

`exec form`은 JSON array로 파싱되므로, double-quotes(“)를 이용해야 함.

Example

```
FROM ubuntu
...
CMD [ "apache2" , "-DFOREGROUND" ]
```

```
FROM ubuntu
...
CMD [ "python" ]
```

```
FROM ubuntu
CMD echo "This is a test."
```

```
FROM ubuntu
CMD [ "/usr/bin/wc" , "--help" ]
```

```
...
CMD [ "catalina.sh" , "run" ]
```

위와같이 컨테이너가 시작될 때 실행할 명령어를 지정합니다.
맨 아래는 우리가 익숙한 **Tomcat**의 **Dockerfile**의 마지막 라인입니다.

ENTRYPOINT

`CMD`와 마찬가지로 컨테이너가 실행될때 기본 command를 지정합니다.

`CMD`와 비슷하지만 `CMD`는 `docker run` 명령어의 인자들로 override 되고, `ENTRYPOINT`는 항상 실행된다는 것입니다.

`docker run` 커맨드에 인자들을 추가하여 실행하면, 그 인자들은 `CMD`의 요소들을 대체하여 `ENTRYPOINT` Instruction에 지정된 커맨드의 인자로 추가됩니다.

Syntax

`ENTRYPOINT` instruction 는 아래와 같이 2가지 포맷을 지원합니다.

- `ENTRYPOINT ["executable", "param1", "param2"]` (*exec form, preferred*)
- `ENTRYPOINT command param1 param2` (*shell form*)

Example

```
FROM ubuntu
ENTRYPOINT ["/bin/echo", "Hello world"]
```

ENTRYPOINT & CMD Instruction 예제

ENTRYPOINT와 CMD의 차이를 알 수 있는 예제입니다.

```
FROM centos
ENTRYPOINT ["/bin/echo", "Hello docker"]
CMD ["world"]
```

위와같은 Dockerfile을 작성한 후 아래와 같이 이미지를 빌드하고 실행합니다.

```
$ docker build -t my-ubuntu:v3 .

$ docker run my-ubuntu:v3
Hello docker world

$ docker run my-ubuntu:v3 place
Hello docker place
```

인자를 주지않고 실행한 첫 번째 컨테이너는 **ENTRYPOINT**의 명령어와 인자, 그리고 **CMD**의 인자를 모두 그대로 사용하여 실행합니다.
인자를 주고 실행한 두 번째 컨테이너는 **CMD**의 내용을 명령줄의 내용으로 치환하여 실행합니다. (world -> place로 변경)

EXPOSE

EXPOSE Instruction은 Container가 Runtime시 수신대기할 포트를 지정하는 명령어입니다. TCP 또는 UDP를 지정할 수 있으며, 지정하지 않을 경우 기본값인 TCP로 지정됩니다.

주의해야 할 점은 EXPOSE 명령어 자체가 실제로 포트를 열지는 않는다는 점입니다. EXPOSE 명령은 이미지를 만드는 사람과 이미지를 사용하는 사람이 포트 및 프로토콜 규약을 명시해놓은 문서와 같은 역할을 하는 것입니다.
실제로 Container의 포트는 다음과 같은 방법으로 publish 됩니다.

- `--publish`(또는 `-p`) flag로 컨테이너의 port와 Host 머신의 port를 지정하여 오픈
- `--publish-all`(또는 `-P`) flag로 `EXPOSE`로 지정된 모든 포트를 오픈

Syntax

```
EXPOSE <port> [<port>/<protocol>...]
```

Example

```
EXPOSE 80/udp
```

```
EXPOSE 80/tcp  
EXPOSE 80/udp
```

```
$ docker run -d --name my-nginx -p 8080:80 nginx
```

ENV

ENV Instruction은 환경변수를 설정하는데 사용됩니다.

ENV를 사용하여 설정된 환경변수는 Container 실행 시에도 유지되며, docker run 명령어에 --env flag를 이용하여 변경할 수도 있습니다.

Syntax

```
ENV <key> <value>
```

```
ENV <key>=<value> ...
```

두 가지 다 사용가능하나, 아래 방법을 사용하는 것을 권장함.

Example

```
ENV MY_NAME="Tom Cruise" MY_FIGHTER_JET=F14\ Tomcat \
MY_BIKE=Kawasaki
```

값에 공백을 넣으려면 double-quotes(")로 감싸거나 backslashes(\)를 사용함.

```
FROM busybox
ENV FOO=/bar
WORKDIR ${FOO}    # WORKDIR /bar
ADD . $FOO        # ADD . /bar
COPY \$FOO /quux # COPY $FOO /quux
```

COPY

호스트의 Context 내의 파일 또는 디렉토리들을 Container의 파일시스템으로 복사하는 명령어입니다. 또한 `--chown` 옵션으로 파일 및 디렉토리에 대한 유저와 그룹을 지정할 수 있습니다.

Syntax

```
COPY [--chown=<user>:<group>] <src>... <dest>
```

```
COPY [--chown=<user>:<group>] [<src>, ... <dest>]
```

| 경로에 공백이 포함된 경우 아래 방법을 사용.

Example

```
COPY . /bar/          # adds all files and directories
COPY test.jar /bin/test/ # adds a test.jar file
COPY hom* /mydir/      # adds all files starting with "hom"
COPY hom?.txt /mydir/   # ? is replaced with any single character, e.g., "home.txt"
COPY --chown=55:mygroup files* /somedir/
COPY --chown=bin files* /somedir/
COPY --chown=1 files* /somedir/
COPY --chown=10:11 files* /somedir/
```

`COPY --chown`에 의해 주어진 유저명과 그룹명으로 지정되지 않은 모든 파일 및 디렉토리들은 Container안에서 UID, GID 0번으로 생성됩니다.

ADD

`ADD`는 Syntax 및 기능면에서 `COPY`와 유사하나, URL을 지정해 파일을 복사 할 수 있고 압축파일(tar)을 자동으로 풀어서 복사한다는 점에서 차이가 있습니다.

| Tar archive format : identity, gzip, bzip2, xz

Syntax

```
ADD [--chown=<user>:<group>] <src>... <dest>
```

```
ADD [--chown=<user>:<group>] [<src>, ... <dest>]
```

| 경로에 공백이 포함된 경우 아래 방법을 사용.

Example

```
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
RUN make -C /usr/src/things all
```

USER

Docker로 Container를 수행할때 Container 실행 유저는 Docker의 기본설정으로 root로 설정입니다.

```
$ docker run ubuntu whoami  
root
```

USER 명령어는 이러한 Container안에서 명령을 실행 할 유저명(또는 UID)과 유저그룹(or GID)을 설정하는 명령어이며, Dockerfile 안에서 USER명령어를 셋팅한 이후의 RUN, CMD, ENTRYPONT 명령어에 적용됩니다.

Syntax

```
USER <user>[ <group>]
```

```
USER <UID>[ <GID>]
```

Example

```
# 시스템 그룹 및 유저로 postgres를 추가한다.  
RUN groupadd -r postgres && useradd --no-log-init -r -g postgres postgres  
  
# Run the rest of the commands as the `postgres` user  
USER postgres
```

WORKDIR

`WORKDIR` 명령어는 `WORKDIR` 명령어에 뒤따르는 `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, `ADD` 명령어의 작업디렉토리를 지정하는 명령어입니다.
`WORKDIR`로 지정한 디렉토리가 없는 경우에는 자동으로 생성되며, `WORKDIR`을 지정하지 않는 경우에는 `\`가 작업 디렉토리로 사용됩니다.

Syntax

```
WORKDIR /path/to/workdir
```

Example

```
WORKDIR /a      # 작업디렉토리를 /a로 이동합니다  
WORKDIR b      # /a에서 하위의 b디렉토리로 이동합니다.  
WORKDIR c      # /a/b에서 하위의 c디렉토리로 이동합니다.  
RUN pwd        # RUN명령어가 실행되는 디렉토리는 WORKDIR로 이동한 /a/b/c가 됩니다.
```

아래와 같이 `ENV`로 지정한 환경변수와 함께 쓰일 수도 있습니다.

```
ENV DIRPATH /path  
ENV DIRNAME dname  
WORKDIR $DIRPATH/$DIRNAME  
RUN pwd      # 출력결과 /path/dname
```

VOLUME

VOLUME Instruction은 Mount point를 지정하는 데 사용됩니다.

컨테이너에서 생성된 데이터를 영구보존하고자 할때 사용되는 명령어로, 컨테이너가 실행될때 Volume 명령어로 선언된 Container의 지점이 Host 머신의 특정지점과 마운트되어 실행됩니다.

이때 Host에 생성되는 경로는 /var/lib/docker/volumes/ 아래가 됩니다.

Syntax

```
VOLUME [ "/data" ]
```

Example

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
CMD ["/bin/bash"]
```

```
$ docker build -t volumetest:v1 .
$ docker volume create my-volume
$ docker run -it --name volumetest --mount source=my-volume,target=/myvol volumetest:v1 /bin/bash
```

`docker volume create` 명령으로 먼저 Volume을 생성한 후, `docker run` 명령어에서 `--mount` 나 `-v` flag를 이용하여 마운트 위치를 지정합니다.

Summary

- Dockerfile
- docker build
- Build context
- Instructions
 - FROM
 - LABEL
 - RUN
 - CMD , ENTRYPOINT
 - EXPOSE
 - ENV
 - COPY , ADD
 - USER
 - WORKDIR
 - VOLUME

문의처 : 정상업 / rogallo.jung@samsung.com