Ch 4. 도커 이미지

4. 도커 이미지

도커 이미지는 애플리케이션 자신 뿐만아니라 실행에 필요한 모든 것들을 담고 있습니다. OS, DBMS, 웹서버 등 시스템 상에서 이용되는 대부분의 요소를 이미지로 생성할 수 있으며, 그것은 전문가의 영역이 아닌 이 글을 보는 여러분 모두 하실 수 있는 작업입니다.

컨테이너를 생성할 때 어떻게 이미지가 이용되는지, 이미지는 어떻게 생성되는지 실습을 통해 이해해보도록 하겠습니다. 더불어, 이미지를 생성하기 위한 Dockerfile 은 어떻게 작성해야 하는지도 살펴보는 시간을 가져보겠습니다.

4.1 도커 이미지 개요

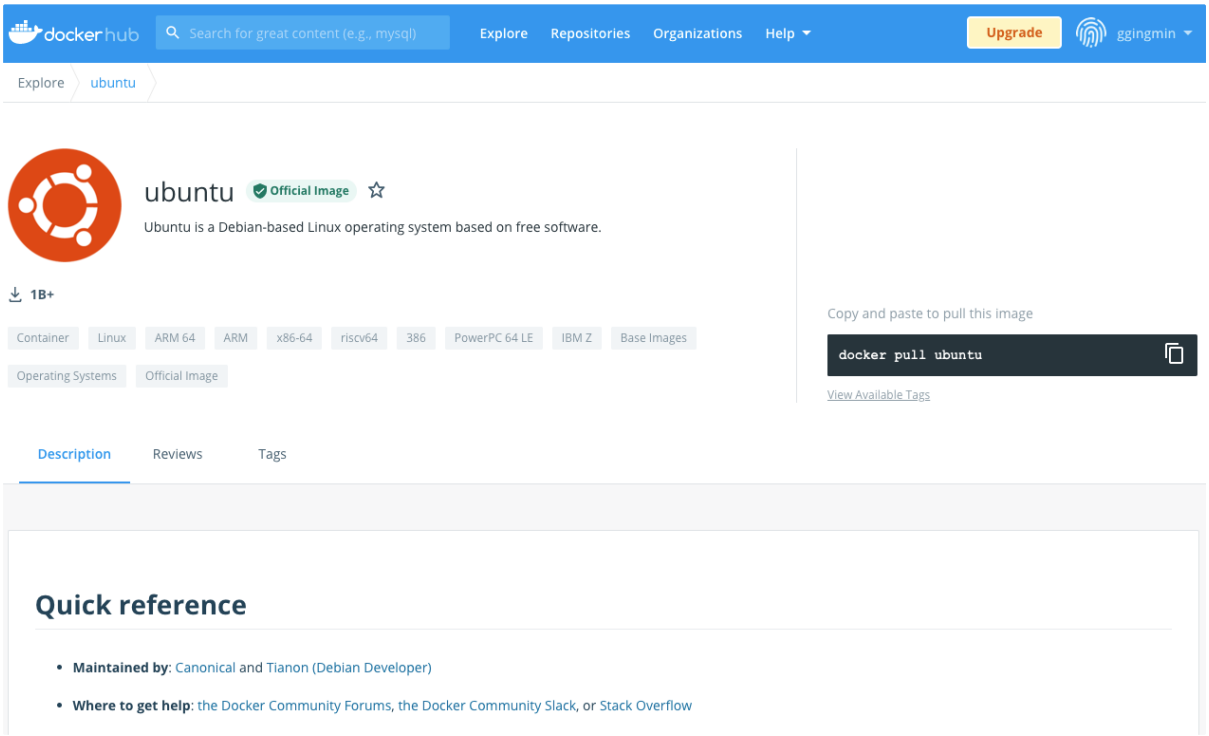
도커 컨테이너를 실행할 때 `run` 명령어의 인자로 어떤 것을 작성했는지 기억 나시나요? `-it`, `--name` 등의 옵션 등도 중요하지만 가장 중요한 건 바로 어떤 이미지를 실행할 것인지 입력하는 것입니다.

```
sudo docker container run -it --name ubuntu ubuntu:latest
```

명령어를 하나 작성해보았습니다. 마지막에 `ubuntu:latest` 라고 되어있는 부분을 주목해주세요. 기본적으로 우리 로컬 환경에는 필요한 도커 이미지가 없습니다. 그렇기 때문에 컨테이너를 실행할때도 아래와 같은 메세지가 뜹니다. 그리고 어디선가 이미지를 받아서 컨테이너를 실행시킵니다.

```
➜  ~ sudo docker container run -it --name ubuntu ubuntu:latest
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
be0de17fe24f: Pull complete
Digest: sha256:82becede498899ec668628e7cb0ad87b6e1c371cb8a1e597d83a47fac21d6af3
Status: Downloaded newer image for ubuntu:latest
root@3cd22a69d7ce:/#
```

그렇다면 이미지는 도대체 어디에서 오는 것일까요? 도커는 로컬에 필요한 이미지가 없을 경우 Docker Hub 라는 레지스트리에서 이미지를 받아옵니다. 이 레지스트리에는 도커가 인증한 공식 이미지와 더불어 개인이 직접 빌드한 이미지도 확인할 수 있습니다. 아래는 'Docker Hub' 에서 확인할 수 있는 'ubuntu'의 공식 이미지 페이지입니다.



컨테이너를 실행할 때 단일 이미지만을 이용하는 경우는 거의 없습니다. 겉으로 보기에는 단일 이미지이지만 내부를 살펴보면 다중 이미지로 구성이 되어있음을 알 수 있죠. 결국 실제 운영환경에서 실행되는 이미지는 베이스가 되는 이미지와 설정 파일, 그리고 명령들을 한 데 모아 빌드한 것이고, 그렇게 빌드된 이미지로 컨테이너를 실행합니다. 여기서 등장하는 것이 바로 **Dockerfile** 입니다.

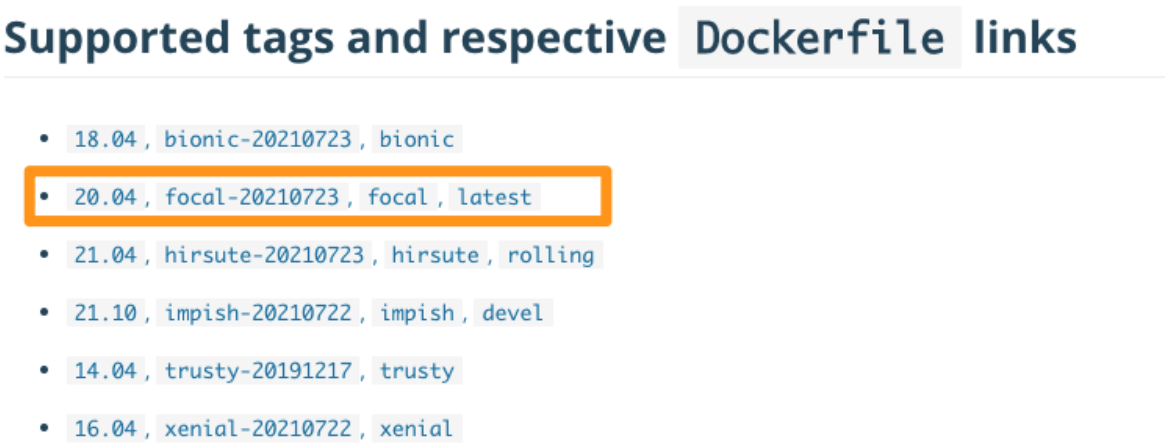
4.2 도커 이미지 관리

도커에서 이미지를 다루기 위해서는 기본적인 명령어를 숙지하고 있어야 합니다. 아래 명령어를 직접 실행해보면서 충분히 연습해봅시다.

1) `pull` - 도커 이미지 받기

```
sudo docker image pull ubuntu sudo docker image pull ubuntu:focal sudo
docker image pull ubuntu:latest sudo docker image pull 이미지명[:태그]
```

이미지를 가져올 때는 저장소명(repository)을 사용하며, 필요에 따라 **태그**를 추가로 작성합니다. 태그는 저장소에 저장된 이미지를 버전별로 관리하는 데에 사용됩니다.



위에 작성한 세 줄의 명령어는 사실 모두 같은 이미지를 가리키고 있습니다. 태그를 명시하지 않은 첫번째 명령은 `latest` 태그가 붙은 것과 같은 의미입니다. 태그의 경우 복수로 설정이 가능하기 때문에 보통 릴리즈 연도, 숫자형식의 버전, 영문 별칭 등이 다양하게 활용됩니다.

여러 태그 중 가장 중요한 것은 `latest` 입니다. 일반적으로 `latest` 태그는 장기 지원 버전(LTS)이나 안정화(stable) 버전에 붙입니다. Ubuntu의 경우 20.04 버전이 가장 최신의 장기 지원 버전이기 때문에 `latest` 태그가 붙어있는 것이죠. 이외 버전을 설치해야 하는 경우 반드시 태그를 명시해야 합니다.

```
→  ~ sudo docker container run -it --name ubuntu ubuntu:latest
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
be0de17fe24f: Pull complete
Digest: sha256:82becede498899ec668628e7cb0ad87b6e1c371cb8a1e597d83a47fac21d6af3
Status: Downloaded newer image for ubuntu:latest
root@3cd22a69d7ce:/#
```

이미지를 받아오는 과정을 들여다 보면 다이제스트라는 긴 문자열이 등장합니다. 이는 Docker Hub에서 이미지를 식별하기 위한 고유한 값입니다.

> 💡 **SHA256 - Secure Hash Algorithm**
> 해싱 알고리즘 중 하나인 SHA256은 데이터를 암호화 하는 데에 사용합니다. 256이라는 숫자는 총 256bit의 길이로 해시 값을 이룬다는 의미이며, 2^{256}의 경우의 수가 존재합니다.

2) `ls` - 이미지 목록 조회

▶ 옵션

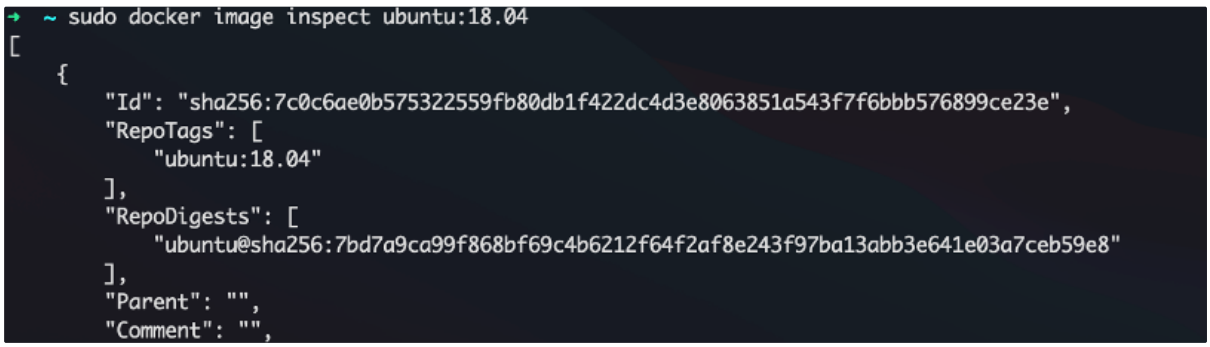
```
sudo docker image ls
```

```
ubuntu          20.04     58810d071ecb   2 weeks ago   65.6MB
ubuntu          latest    58810d071ecb   2 weeks ago   65.6MB
ubuntu          18.04     7c0c6ae0b575   2 weeks ago   56.6MB
```

이미지를 조회해보면 실체가 같은 20.04와 latest는 이미지 ID가 같지만 18.04는 다른 것을 확인할 수 있습니다.

3) `inspect` - 이미지 정보 조회

```
sudo docker image inspect ubuntu:18.04
```



```
~ sudo docker image inspect ubuntu:18.04
[
    {
        "Id": "sha256:7c0c6ae0b575322559fb80db1f422dc4d3e8063851a543f7f6bbb576899ce23e",
        "RepoTags": [
            "ubuntu:18.04"
        ],
        "RepoDigests": [
            "ubuntu@sha256:7bd7a9ca99f868bf69c4b6212f64f2af8e243f97ba13abb3e641e03a7ceb59e8"
        ],
        "Parent": "",
        "Comment": "",
```

이미지 정보를 조회해보면 굉장히 많은 내용이 JSON 형식으로 출력됩니다. 이 모든 내용에서 사용자가 원하는 내용만 조회하기 위해서는 다음과 같이 출력 포맷을 설정해주면 됩니다.

```
sudo docker image inspect --format="{{ .RepoTags }}" ubuntu:18.04
```

```
~ sudo docker image inspect --format="{{ .RepoTags }}" ubuntu:18.04
[ubuntu:18.04]
```

저는 이미지명과 태그를 조회하는 포맷을 작성했습니다. 이외에도 필요에 따라 다양한 값을 포맷을 활용해 조회할 수 있습니다.

4) `tag` - 이미지에 태그 설정

```
sudo docker image tag ubuntu:18.04 ggingmin/ubuntuos:1.0 sudo docker
image tag 대상이미지명[:태그] [사용자명/]이미지명[:태그]
```

```
→  ~ sudo docker image tag ubuntu:18.04 ggingmin/ubuntuos:1.0
→  ~ sudo docker image ls
REPOSITORY            TAG       IMAGE ID       CREATED        SIZE
myubuntu              1.0       0d036823a042   11 days ago    87.1MB
ggingmin/apacheweb    1.1       1a068eebb2e6   11 days ago    128MB
ggingmin/apacheweb    1.0       206b693d8c9d   11 days ago    131MB
ubuntu                20.04     58810d071ecb   2 weeks ago    65.6MB
ubuntu                latest    58810d071ecb   2 weeks ago    65.6MB
ggingmin/ubuntuos     1.0       7c0c6ae0b575   2 weeks ago    56.6MB
ubuntu                18.04     7c0c6ae0b575   2 weeks ago    56.6MB
httpd                 latest    617ccd1493f0   3 weeks ago    131MB
```

태그를 설정한다는 것은 현재 도커 엔진에 저장된 이미지의 복사본을 만들어 새로운 이미지명과 태그를 붙인다는 뜻입니다. 실행 결과를 조회해보면 두 이미지의 ID가 같죠. 왜 굳이 실체가 같은 이미지를 복잡하게 다른 이름을 붙여 관리할까요?

우리가 생성한 도커 이미지는 Docker Hub를 비롯한 다양한 클라우드 사업자의 레지스트리에 저장할 수 있습니다. 하지만 각 클라우드 사업자 별로 이미지를 업로드할 때 사용되는 이미지명의 규칙이 달라 이를 맞춰주어야 합니다. 우리가 레지스트리를 학습 후에 실습해 볼 Docker Hub와 Google 클라우드도 이미지명 규칙이 다릅니다.

5) `rm` - 이미지 삭제, 이미지 태그 해제

▶ 옵션

```
sudo docker image rm ggingmin/ubuntuos:1.0 sudo docker image rm ubuntu:18.04 sudo docker image rm 이미지명[:태그] sudo docker image rm 이미지ID
```

```
→  ~ sudo docker image rm ggingmin/ubuntuos:1.0
Untagged: ggingmin/ubuntuos:1.0
→  ~ sudo docker image rm ubuntu:18.04
Untagged: ubuntu:18.04
Untagged: ubuntu@sha256:7bd7a9ca99f868bf69c4b6212f64f2af8e243f97ba13abb3e641e03a7ceb59e8
Deleted: sha256:7c0c6ae0b575322559fb80db1f422dc4d3e8063851a543f7f6bbb576899ce23e
```

원 이미지에서 태그된 이미지에 `rm` 명령어를 실행하면 태그 해제만 되고 실체는 삭제되지 않습니다. 나머지 이미지까지 명령을 실행하면 그 때 이미지가 삭제 되죠. 만약 이 두 가지 이미지를 한꺼번에 삭제하려면 이미지 ID와 강제 삭제를 위한 `-f` 옵션을 사용해야 합니다. 여러 곳에서 동일한 이미지를 참조하고 있는 경우 강제 옵션 없이 이미지 삭제가 불가능 합니다.

```
→  ~ sudo docker image rm 7c0c6ae0b575
Error response from daemon: conflict: unable to delete 7c0c6ae0b575 (must be forced) - image is referenced in multiple repositories
→  ~ sudo docker image rm -f 7c0c6ae0b575
Untagged: ggingmin/ubuntuos:1.0
Untagged: ubuntu:18.04
Untagged: ubuntu@sha256:7bd7a9ca99f868bf69c4b6212f64f2af8e243f97ba13abb3e641e03a7ceb59e8
Deleted: sha256:7c0c6ae0b575322559fb80db1f422dc4d3e8063851a543f7f6bbb576899ce23e
```

6) `container commit` - 실행 중인 컨테이너로부터 이미지 생성

▶ 옵션

```
sudo docker container run -d -p 80:80 --name apache httpd # 사전에 실행
sudo docker container commit -a "ggingmin" apache ggingmin/apacheweb:1.0
```

```
→  ~ sudo docker container commit -a "ggingmin" apache ggingmin/apacheweb:1.0
sha256:206b693d8c9d52951bcee62f156ccd2bb7bc96c23372cfebbae880c69986bc9b
→  ~ sudo docker image ls
REPOSITORY            TAG       IMAGE ID       CREATED          SIZE
ggingmin/apacheweb    1.0       206b693d8c9d   21 seconds ago   131MB
ubuntu                latest    58810d071ecb   8 days ago       65.6MB
httpd                 latest    617ccd1493f0   13 days ago      131MB
```

이미지를 생성하는 명령어이지만 특이하게 `container` 명령을 사용합니다. `commit` 은 실행 중인 컨테이너에 수행되는 명령으로서, 컨테이너를 실행하는데에 사용했던 이미지와는 다른 이미지를 만들어 냅니다. 컨테이너의 당시 상태를 그대로 본뜬 스냅샷을 이미지로 만들었기 때문이죠.

7) `container export` - 실행 중인 컨테이너로부터 파일 생성

```
sudo docker container export apache > apache.tar sudo docker container
export 컨테이너명 > [경로/]파일명
```

`export` 역시 `container` 와 함께 사용되는 명령어 입니다. 현재 컨테이너를 본떠 파일로 내보내는 역할을 하고 리눅스에서 많이 사용되는 파일 압축 형식인 `.tar` 를 사용하였습니다. 이렇게 생성된 파일은 import 명령어를 사용하여 이미지를 생성할 수 있습니다.

8) `import` - 파일을 이미지로 생성

```
sudo docker image import apache.tar ggingmin/apacheweb:1.1
```

```
→  ~ sudo docker container export apache > apache.tar
→  ~ sudo docker image import apache.tar ggingmin/apacheweb:1.1
sha256:ed577a6e26e0f7bbe1c93edf536c53c69ac627de0c6b93d7e2f624a047634a3d
```

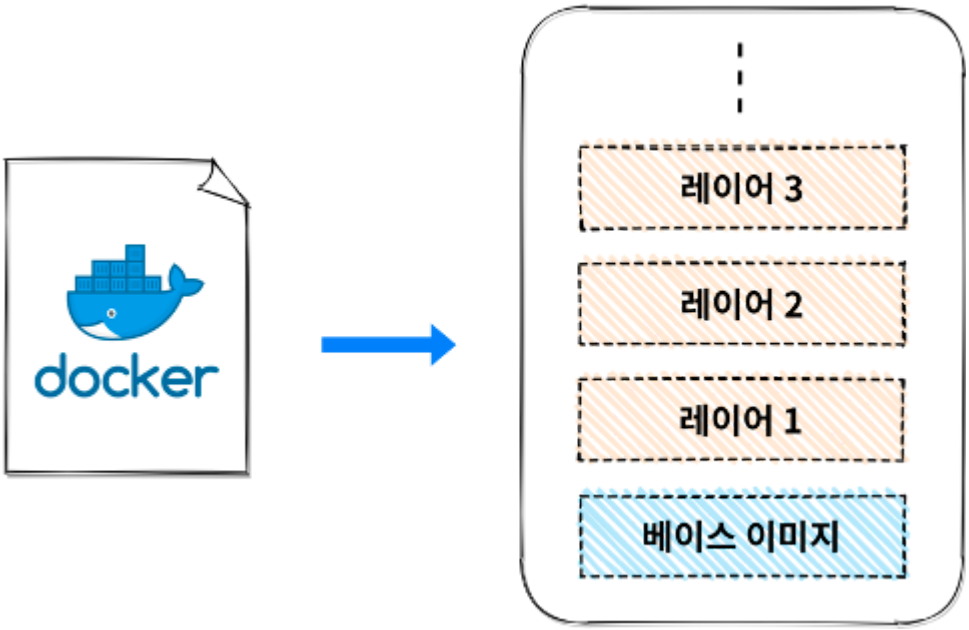
위에서 생성했던 파일을 기반으로 이미지를 생성하는 명령입니다. `container` 가 아닌 `image` 와 함께 명령이 작성된다는 점을 유의합니다.

4.3 Dockerfile

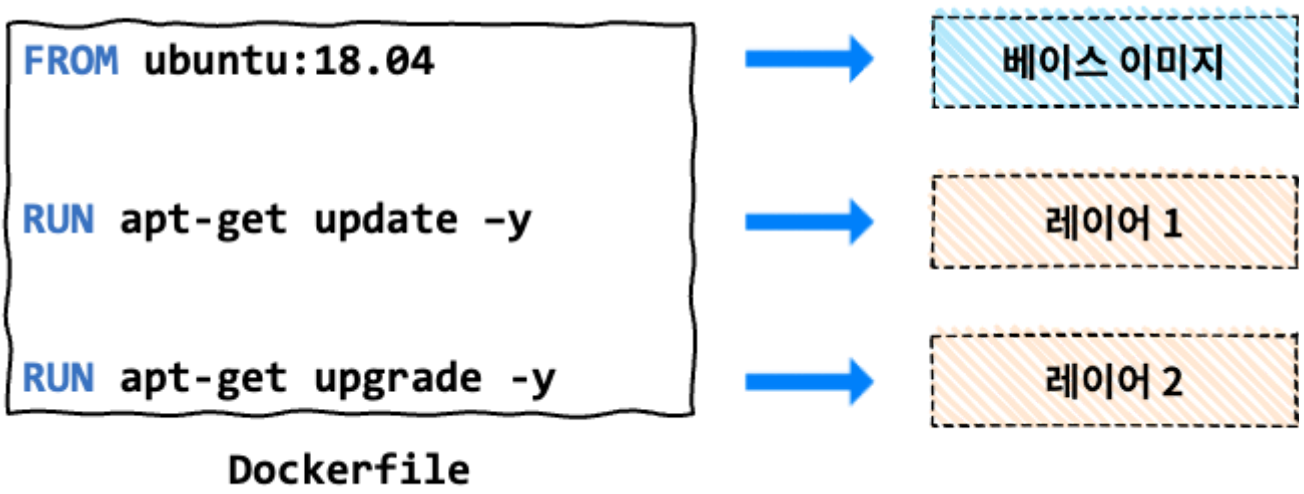
Dockerfile의 역할은 간단합니다. 새로운 이미지를 빌드하는 데에 필요한 이미지와 설정들을 작성한 파일...! 어떤 이미지를 사용할지, 어떤 포트를 열어놓을지 등에 대해서 정의가 완료되면 이 내용을 Dockerfile의 양식에 맞게 작성하기만 하면 됩니다. 먼저 간단한 Dockerfile 예시를 살펴보겠습니다.

```
FROM ubuntu:18.04 RUN apt-get -y update && apt-get -y upgrade RUN apt-get
-y install nginx EXPOSE 80 CMD ["nginx", "-g", "daemon off;"]
```

위 명령은 ubuntu에 nginx 웹서버를 설치하고 80 포트를 열어놓는 내용을 담고 있습니다. 이 파일의 내용을 기반으로 빌드가 진행되면 그 결과물로 새로운 이미지가 나오게 됩니다. Dockerfile로 인프라를 사전에 빌드해 놓는다면 어플리케이션 개발자는 이를 받아서 컨테이너를 실행하기만 하면 되니 매우 간편합니다.



좀 더 자세히 Dockerfile의 구조를 살펴 보겠습니다. `FROM` 으로 시작되는 명령어에서는 베이스이 미지를 설정하게 됩니다. 이후 실행되는 명령은 모두 이 베이스 이미지를 기반으로 실행됩니다.



각각의 명령줄은 하나의 이미지 레이어를 구성합니다. 즉, 각 명령 스텝별로 이미지가 생성되는 것이죠. 이 과정에서 생성되는 중간 이미지는 캐싱되어 다른 이미지를 생성할 때도 사용됩니다.

Dockerfile에서 사용할 수 있는 명령어는 용도에 따라 아주 다양합니다. 작성 시 대소문자의 구분은 없으나 통상 대문자로 작성하게 됩니다. 그러면 위의 Dockerfile이 각각 어떤 작업을 수행하는지 나눠서 보겠습니다.

```
FROM ubuntu:18.04
```

FROM 절은 베이스 이미지를 세팅합니다. 빌드할 이미지의 베이스를 'ubuntu'로 설정한다는 의미죠. 여기까지만 본다면 ubuntu의 이미지를 그대로 복제한 것과 같습니다.

```
RUN apt-get -y update && apt-get -y upgrade RUN apt-get -y install nginx
```

RUN 절은 이미지를 생성에 필요한 미들웨어나 어플리케이션을 세팅하기 위한 명령을 실행합니다. 여기서는 `apt-get` 명령어를 통해 'nginx'를 설치하고 있습니다.

```
EXPOSE 80
```

'nginx' 웹서버를 통해 외부와 통신하기 위해서는 기본적으로 포트가 열려있어야 합니다. HTTP 방식의 통신을 위해 '80'번 포트를 열어주었습니다. 포트는 역할별로 지정되어 있는 것들이 있는데, 이는 하나의 약속으로 이메일을 송수신 하거나 서버 접속방식 등에 포트가 할당 되어있습니다.

```
CMD ["nginx", "-g", "daemon off;"]
```

nginx 설치 및 포트설정이 완료된 후에는 웹서버를 구동시킵니다. CMD 명령은 기 생성된 이미지를 기반으로 구동된 컨테이너에서 명령을 실행하는 것이며, 하나의 Dockerfile에서 한 번의 명령만 유효합니다. 복수의 CMD 명령이 있을 경우 마지막 CMD 명령만 실행됩니다.

여기까지 웹서버 구축을 위한 Dockerfile 예시를 살펴보았습니다. 각 명령어 별 상세 역할은 이후에 추가로 확인하기로 하죠. 실제로 컨테이너를 띄우려면 빌드의 과정이 필요합니다. 다음으로 넘어가 봅시다.

2.3.3 Dockerfile 빌드

'빌드' 라는 용어는 프로그래밍 전방위에서 접하게 됩니다. '모바일 앱을 빌드한다.', '스프링 프로젝트를 빌드한다.' 등등 소스 코드를 컴파일 한다는 의미로 주로 사용되죠. 도커에서 빌드는 Dockerfile을 기반으로 이미지를 생성한다는 의미를 가지고 있습니다. 위에서 작성한 Dockerfile을 기반으로 이미지를 만들어 봅시다.

```
mkdir docker && cd docker touch Dockerfile nano Dockerfile
```

Ubuntu Server가 설치된 VM에서 위의 명령어를 입력하여 Dockerfile을 생성한 후, nano 혹은 vi를 통해 위에서 확인했던 명령어를 작성합니다.

```
sudo docker build -t sample:1.0 /home/(USER)/docker
```

```
ggingmin@ubuntu_server:~/docker/web-server$ sudo docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
sample       1.0       04eb65beeb7e   8 minutes ago   164MB
nginx        latest    d1a364dc548d   3 weeks ago     133MB
```

위 명령어를 실행하면 순차적으로 이미지가 빌드됩니다. 각종 패키지가 설치되기 때문에 약간의 시간이 소요되지만 그렇게 오래걸리지는 않습니다. 옵션별로 어떤 의미를 가지는지 살펴 보겠습니다.

`sudo docker build -t` 는 도커 이미지를 생성하기 위한 기본 명령어 입니다. `sample:1.0` 은 태그라고 부르는데 이름표를 붙인다고 생각하시면 되고 `[이미지명]:[버전]` 의 형태로 작성합니다. 마지막 부분에는 경로를 입력하며, Dockerfile이 있는 경로로 이동해서 현재 경로를 나타내는 `.` 를 사용하거나 절대 경로, 상대 경로를 입력하는 것 모두 가능합니다.

4.4 Dockerfile 명령어

▶ Dockerfile 명령어 일람

- Best practices for writing Dockerfiles

1) `FROM` - 베이스 이미지 설정

```
FROM ubuntu:18.04
```

베이스 이미지를 세팅하는 명령으로서 기본적으로 Docker Hub에서 이미지를 탐색해 빌드에 사용합니다. 하지만 상황에 따라 사용자가 직접 만든 베이스 이미지를 통해 빌드하는 경우도 있습니다.

2) `RUN` - 이미지를 빌드할 때 실행할 명령 설정

```
RUN apt-get -y update # Shell 형식 RUN ["/bin/bash", "-c", "apt-get -y
update"] # Exec 형식
```

`RUN` 을 통해 작동하는 명령은 아직 컨테이너가 작동하는 상태가 아닙니다. 베이스 이미지에 추가적으로 명령을 실행해 필요한 패키지나 미들웨어를 설치하기 위해 많이 사용합니다.

명령은 위 예시처럼 Shell 형식, 혹은 Exec 형식으로 작성할 수 있습니다. Shell 형식에서 `RUN` 뒤에 바로 명령어를 작성하는 것은 `/bin/sh -c` 이 붙어있는 것과 마찬가지라고 이해하시면 헷갈리지 않습니다.

두 가지가 혼용되는 경우가 많지만 실행할 셸 혹은 프로그램을 지정한다는 측면에서 Exec 형식이 권장됩니다.

3) `CMD` - 이미지를 통해 생성된 컨테이너 내부에서 실행되는 명령

```
FROM ubuntu:18.04 CMD echo "Hello, Docker!" # Shell 형식 CMD ["/bin/echo", "Hello, Docker!"] # Exec 형식
```

컨테이너는 `docker container run` 명령어를 통해 실행됩니다. 이렇게 컨테이너가 실행될 때 수행할 명령은 `CMD` 를 통해 세팅이 가능합니다. 유의할 점은 Dockerfile 전체에서 `CMD` 명령은 단 하나만 유효하고, 여러 개의 명령이 있다면 마지막 것만 실행이 된다는 점입니다.

`CMD` 명령 역시 `RUN` 과 같이 Shell, Exec 형식 모두 사용 가능합니다.

4) `ENTRYPOINT` - 이미지를 통해 생성된 컨테이너 내부에서 실행되는 명령

```
FROM ubuntu:18.04 ENTRYPOINT echo "Hello, Docker!" # Shell 형식 ENTRYPOINT ["/bin/echo", "Hello, Docker!"] # Exec 형식
```

뭔가 이상한데... 라는 생각과 함께 위에 `CMD` 명령을 다시 올려다보셨을 겁니다. `ENTRYPOINT` 의 역할이 `CMD` 와 같게 쓰여져 있기 때문이죠. 실제로 이 둘은 컨테이너가 실행된 후 그 내부에서 명령을 실행한다는 동일한 기능을 가지고 있습니다. 하지만 이 둘 사이에는 중요한 차이점이 있습니다. 다음의 명령을 함께 보겠습니다.

```
FROM ubuntu:18.04 ENTRYPOINT ["/bin/echo"] CMD ["Hello, Docker!"]
```

```
sudo docker image build -t hellodocker . sudo docker container run -it hellodocker sudo docker container run -it hellodocker 'Hi, Docker!'
```

```
→  docker sudo docker container run -it hellodocker

Hello, Docker!
→  docker sudo docker container run -it hellodocker 'Hi, Docker!'


Hi, Docker!
```

`ENTRYPOINT` 의 명령은 사용자가 어떤 인수를 명령으로 넘기더라도 Dockerfile에 명시된 명령을 그대로 실행합니다. 반면 `CMD` 는 컨테이너를 실행할 때 사용자가 인수를 넘기면 기존에 작성된 내용을 덮어 씁니다. 이러한 특성을 이해한다면 좀 더 효율적인 이미지 생성이 가능합니다.

5) `ONBUILD` - 이미지 빌드가 완료된 후에 실행되는 명령

```
# ./docker/Dockerfile.base FROM ubuntu:18.04 ONBUILD RUN ["apt-get", "update"] ONBUILD RUN ["apt-get", "-y", "install", "vim"]
```

```
sudo docker image build -f ./Dockerfile.base ./ -t baseimage # Parent Image
```

```
→  docker sudo docker image build -f ./Dockerfile.base ./ -t baseimage
[+] Building 2.0s (5/5) FINISHED
 => [internal] load build definition from Dockerfile.base                                                0.0s
 => => transferring dockerfile: 148B                                                                     0.0s
 => [internal] load .dockerignore                                                                        0.0s
 => => transferring context: 2B                                                                          0.0s
 => [internal] load metadata for docker.io/library/ubuntu:18.04                                          1.9s
 => CACHED [1/1] FROM docker.io/library/ubuntu:18.04@sha256:7bd7a9ca99f868bf69c4b6212f64f2af8e243f97ba13abb  0.0s
 => exporting to image                                                                                   0.0s
 => => exporting layers                                                                                  0.0s
 => => writing image sha256:e51247658616ea5edc41f9ab27df79f7b2b6e3a66c6f632bc570cff28b85e060             0.0s
 => => naming to docker.io/library/baseimage                                                             0.0s
```

```
# ./docker/Dockerfile FROM baseimage
```

```
sudo docker image build -t hellodocker-child . # Child Image
```

```
→  docker sudo docker image build -t hellodocker-child .
[+] Building 18.6s (7/7) FINISHED
 => [internal] load build definition from Dockerfile                                0.0s
 => => transferring dockerfile: 110B                                                0.0s
 => [internal] load .dockerignore                                                   0.0s
 => => transferring context: 2B                                                     0.0s
 => [internal] load metadata for docker.io/library/baseimage:latest                 0.0s
 => CACHED [1/1] FROM docker.io/library/baseimage                                   0.0s
 => [2/1] RUN ["apt-get", "update"]                                                 8.5s
 => [3/1] RUN ["apt-get", "-y", "install", "vim"]                                   9.6s
 => exporting to image                                                              0.4s
 => => exporting layers                                                             0.4s
 => => writing image sha256:965653108635fa7b1568ed9df03ead960de84ed58348b7dca22af8443e9f0687  0.0s
 => => naming to docker.io/library/hellodocker-child                                0.0s
```

이 명령어는 언뜻 헷갈리기 쉽습니다. 이미지가 빌드될 당시에 뭔가를 실행시키는 것 같지만 실제로 빌드를 해보면 아무 일도 일어나지 않습니다. 대신 `ONBUILD` 가 작성되어 있는 Dockerfile을 빌드해서 생성된 이미지를 베이스로 하여 자식 이미지를 생성할 때 그 명령이 실행되죠. 마치 `RUN` 명령을 물려주는 것과 같습니다.

위와 같이 Dockerfile을 작성해서 베이스 이미지를 빌드해 놓으면 모든 이미지에 공통적으로 설치되어야 하는 패키지나 실행 명령을 빌드 당시에 적용할 수 있습니다.

6) `HEALTHCHECK` - 컨테이너의 작동상태 체크

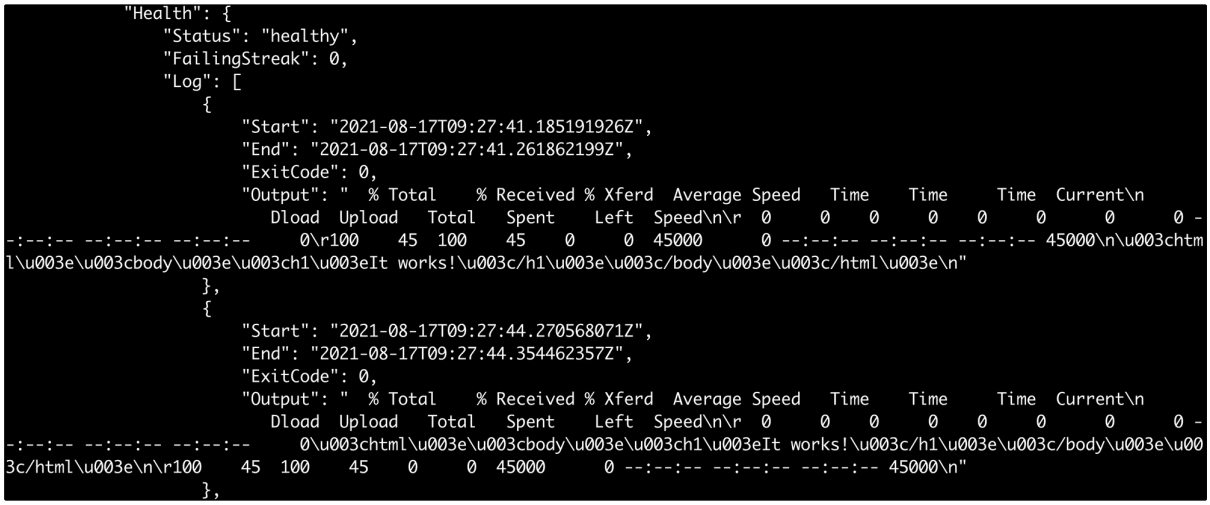
▶ 옵션

```
# ./docker/Dockerfile FROM httpd RUN ["apt-get", "update"] RUN ["apt-get", "-y", "install", "curl"] HEALTHCHECK --interval=3s --timeout=5s --retries=3 CMD curl --fail http://localhost:80/ || exit 1
```

```
sudo docker image build -t web-healthcheck .
```

```
sudo docker container run -d -p 80:80 --name=apache-hc web-healthcheck
```

```
→  docker sudo docker ps
CONTAINER ID   IMAGE             COMMAND              CREATED          STATUS                    PORTS
          NAMES
f3a4ece0ac65   web-healthcheck   "httpd-foreground"   52 seconds ago   Up 3 seconds (healthy)    0.0.0.0:80->80/tcp, :::80-
>80/tcp   apache-hc
→  docker sudo docker ps
CONTAINER ID   IMAGE             COMMAND              CREATED          STATUS                     PORTS
           NAMES
f3a4ece0ac65   web-healthcheck   "httpd-foreground"   59 seconds ago   Up 10 seconds (healthy)   0.0.0.0:80->80/tcp, :::80
->80/tcp   apache-hc
→  docker sudo docker ps
CONTAINER ID   IMAGE             COMMAND              CREATED              STATUS                     PORTS
           NAMES
f3a4ece0ac65   web-healthcheck   "httpd-foreground"   About a minute ago   Up 11 seconds (healthy)   0.0.0.0:80->80/tcp, :
::80->80/tcp   apache-hc
```



```
            "Health": {
                "Status": "healthy",
                "FailingStreak": 0,
                "Log": [
                    {
                        "Start": "2021-08-17T09:27:41.185191926Z",
                        "End": "2021-08-17T09:27:41.261862199Z",
                        "ExitCode": 0,
                        "Output": "  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current\n
                                       Dload  Upload   Total   Spent    Left  Speed\n\r  0     0    0     0    0     0      0      0 -
-:--:-- --:--:-- --:--:--     0\r100    45  100    45    0     0  45000      0 --:--:-- --:--:-- --:--:-- 45000\n\u003chtm
l\u003e\u003cbody\u003e\u003ch1\u003eIt works!\u003c/h1\u003e\u003c/body\u003e\u003c/html\u003e\n"
                    },
                    {
                        "Start": "2021-08-17T09:27:44.270568071Z",
                        "End": "2021-08-17T09:27:44.354462357Z",
                        "ExitCode": 0,
                        "Output": "  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current\n
                                       Dload  Upload   Total   Spent    Left  Speed\n\r  0     0    0     0    0     0      0      0 -
-:--:-- --:--:-- --:--:--     0\u003chtml\u003e\u003cbody\u003e\u003ch1\u003eIt works!\u003c/h1\u003e\u003c/body\u003e\u00
3c/html\u003e\n\r100    45  100    45    0     0  45000      0 --:--:-- --:--:-- --:--:-- 45000\n"
                    },
```

컨테이너를 작동시킬 때, 이 컨테이너가 정상적으로 작동하고 있는지 체크를 해야하는 경우가 있습니다. Dockerfile을 작성할 때 `HEALTHCHECK` 명령을 추가하면 이를 컨테이너 내부에 로그로 남길 수 있습니다. 컨테이너에서 이 명령어가 잘 작동하는지는 `docker ps` 명령어를 통해 `STATUS` 항목을 살펴보면 됩니다. 구동 시간 옆에 (healthy) 라는 문구가 보입니다. 더 정확하게 로그를 보려면 `docker container inspect <컨테이너명>` 을 통해 "Health" 에 해당하는 값을 확인하면 됩니다.

7) `ENV` - 환경변수 설정

8) `WORKDIR` - 작업 디렉토리 할당

```
# ./docker/Dockerfile FROM ubuntu:18.04 RUN mkdir /keypair ENV KEY
/keypair WORKDIR $KEY RUN echo "dummy key" > key.txt
```

```
sudo docker image build -t key-setting . sudo docker container run -it --
name=key-setting key-setting
```

```
➜  docker sudo docker image build -t key-setting .
[+] Building 2.2s (8/8) FINISHED
 => [internal] load build definition from Dockerfile                                                                0.0s
 => => transferring dockerfile: 145B                                                                                0.0s
 => [internal] load .dockerignore                                                                                   0.0s
 => => transferring context: 2B                                                                                     0.0s
 => [internal] load metadata for docker.io/library/ubuntu:18.04                                                     2.1s
 => [1/4] FROM docker.io/library/ubuntu:18.04@sha256:7bd7a9ca99f868bf69c4b6212f64f2af8e243f97ba13abb3e641e03a7ceb59  0.0s
 => CACHED [2/4] RUN mkdir /keypair                                                                                 0.0s
 => CACHED [3/4] WORKDIR /keypair                                                                                   0.0s
 => CACHED [4/4] RUN echo "dummy key" > key.txt                                                                     0.0s
 => exporting to image                                                                                              0.0s
 => => exporting layers                                                                                             0.0s
 => => writing image sha256:855485035741efdbaaa110780b37e7eae7aa850975c461b0b4b580edcbf97d4b                        0.0s
 => => naming to docker.io/library/key-setting                                                                      0.0s
➜  docker sudo docker container run -it --name=key-setting key-setting
root@d6171a5524a5:/keypair# ls -al
total 12
drwxr-xr-x 1 root root 4096 Aug 17 10:06 .
drwxr-xr-x 1 root root 4096 Aug 17 10:13 ..
-rw-r--r-- 1 root root   10 Aug 17 10:06 key.txt
root@d6171a5524a5:/keypair# cat key.txt
dummy key
```

이번에는 두 가지의 명령어를 함께 실습해보았습니다. 환경변수를 설정하고 이를 이용해 작업 경로를 세팅하였는데, `ENV` 에서 설정된 환경변수는 `RUN`, `CMD`, `ENTRYPOINT` 명령에서 모두 사용 가능합니다. 선언된 환경변수를 사용하기 위해서는 `$` 를 앞에 붙이면 됩니다.

`WORKDIR` 는 리눅스의 `cd` 와 유사합니다. Dockerfile 내부에서 경로를 이동할 때도 쓰일 뿐만 아니라, 이렇게 빌드된 이미지를 대화형 컨테이너로 실행할 때 프롬프트의 최초 위치를 결정하기도 합니다.

9) `USER` - 유저 할당

10) `LABEL` - 이미지 버전 정보, 작성자 등 레이블 정보 등록

11) `ARG` - Dockerfile 내부의 변수 할당

```
# ./docker/Dockerfile FROM ubuntu:18.04 ARG newuser=ggingmin LABEL
version="1.0" LABEL description="custom user test" RUN useradd $newuser
USER $newuser
```

```
sudo docker image build -t custom-user . sudo docker container run -it --
name=custom-user custom-user
```

```
→  docker sudo docker image build -t custom-user .
Password:
[+] Building 2.7s (6/6) FINISHED
 => [internal] load build definition from Dockerfile                                                0.0s
 => => transferring dockerfile: 179B                                                                0.0s
 => [internal] load .dockerignore                                                                   0.0s
 => => transferring context: 2B                                                                     0.0s
 => [internal] load metadata for docker.io/library/ubuntu:18.04                                     2.3s
 => CACHED [1/2] FROM docker.io/library/ubuntu:18.04@sha256:7bd7a9ca99f868bf69c4b6212f64f2af8e243f97ba13abb3e641e03  0.0s
 => [2/2] RUN useradd ggingmin                                                                      0.2s
 => exporting to image                                                                              0.0s
 => => exporting layers                                                                             0.0s
 => => writing image sha256:a88b06650f1ec50eef096233c06df086692e05890547b2c0c2e683902634a8d7        0.0s
 => => naming to docker.io/library/custom-user                                                      0.0s
→  docker sudo docker container run -it --name=custom-user custom-user
ggingmin@e107e8847a7b:/$
```

```
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "bash"
            ],
            "Image": "",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "description": "custom user test",
                "version": "1.0"
            }
```

기본적으로 Ubuntu 를 베이스로 이미지를 빌드하고 컨테이너를 실행하면 `root` 사용자로 로그온이 됩니다. 하지만 보안 혹은 기타 접근 제한을 위해 로그온 계정을 이미지 생성 단계에서 설정해야 하는 경우 `USER` 명령어를 활용합니다.

`LABEL` 은 각종 메타데이터를 이미지에 기록하는 역할을 합니다. `docker image inspect` 명령어를 실행한 뒤 Labels 라는 키 항목에서 확인이 가능합니다.

`ARG` 는 빌드가 되는 시점에만 유효한 변수를 할당해주는 역할을 합니다. `ENV` 명령어로 할당된 변수는 컨테이너가 구동된 후에도 유효하지만 `ARG` 는 `build` 명령어를 실행할 때만 작동하게 됩니다. 보통 키 정보나 기타 인증정보를 변수로 선언하는 경우가 많은데 이러한 정보는 `docker history` 로 조회가 가능해 보안에 매우 취약하니 주의해야 합니다.

12) `EXPOSE` - 포트 번호 설정

```
# ./docker/Dockerfile FROM ubuntu:18.04 RUN apt-get -y update && apt-get -y upgrade RUN apt-get -y install nginx EXPOSE 80 CMD ["nginx", "-g", "daemon off;"]
```

```
sudo docker image build -t web-server . sudo docker container run -d -p 80:80 --name=web-server web-server
```

앞서 컨테이너로 웹서버를 띄우는 실습을 진행할 때 포트를 명령어에 같이 작성해주었던 기억이 납니다. 이미지에 포트를 작성하면 따로 컨테이너 실행 시에 `-p` 옵션으로 포트를 지정할 필요가 없는걸까요?

결론부터 말씀드리면 이미지에 작성된 `EXPOSE` 명령어는 단순히 문서의 성격만 가집니다. 실제로 호스트에서 컨테이너의 포트와 통신하도록 listening 상태를 만들어주지 않습니다. 외부로 포트가 노출되지 않는 것이죠. 그저 이미지를 빌드하고 컨테이너를 실행하는 개발자에게 어떤 포트로 어떤 방식을 이용해야 하는지 알려줄 뿐입니다. 컨테이너에서 호스트의 통신 요청에 응답하기 위해서는 반드시 `container run` 단계에서 `-p` 옵션을 통해 포트를 설정해주어야 합니다.

13) `ADD` - 파일 복사(URL 포함)

14) `COPY` - 파일 복사

```
# ./docker/Dockerfile FROM ubuntu:18.04 RUN ["apt-get", "update"] RUN
["apt-get", "-y", "install", "curl"] RUN mkdir /addmyfiles RUN mkdir
/copymyfiles # ./docker 디렉토리에 add.html, copy.html 파일을 미리 생성 ADD
add.html /addmyfiles COPY copy.html /copymyfiles
```

```
➜  docker sudo docker image build -t addcopy .
[+] Building 18.6s (12/12) FINISHED
 => [internal] load build definition from Dockerfile                                                0.0s
 => => transferring dockerfile: 230B                                                                0.0s
 => [internal] load .dockerignore                                                                   0.0s
 => => transferring context: 2B                                                                     0.0s
 => [internal] load metadata for docker.io/library/ubuntu:18.04                                     0.9s
 => [internal] load build context                                                                   0.0s
 => => transferring context: 121B                                                                   0.0s
 => CACHED [1/7] FROM docker.io/library/ubuntu:18.04@sha256:7bd7a9ca99f868bf69c4b6212f64f2af8e243f97ba13abb3e641e03  0.0s
 => [2/7] RUN ["apt-get", "update"]                                                                 7.6s
 => [3/7] RUN ["apt-get", "-y", "install", "curl"]                                                  9.0s
 => [4/7] RUN mkdir /addmyfiles                                                                     0.2s
 => [5/7] RUN mkdir /copymyfiles                                                                    0.4s
 => [6/7] ADD add.html /addmyfiles                                                                  0.0s
 => [7/7] COPY copy.html /copymyfiles                                                               0.0s
 => exporting to image                                                                              0.2s
 => => exporting layers                                                                             0.2s
 => => writing image sha256:5763e5c54093eef002320fcec6eb4a8f094fcf348f7e0f9a5c41b17d0af6d51b        0.0s
 => => naming to docker.io/library/addcopy                                                          0.0s
```

```
➜  docker sudo docker container run -it --name=addcopy addcopy
root@1f5aa2c59057:/# ls -al
total 92
drwxr-xr-x   1 root root 4096 Aug 17 11:52 .
drwxr-xr-x   1 root root 4096 Aug 17 11:52 ..
-rwxr-xr-x   1 root root    0 Aug 17 11:52 .dockerenv
drwxr-xr-x   1 root root 4096 Aug 17 11:52 addmyfiles
drwxr-xr-x   2 root root 4096 Jul 23 13:50 bin
drwxr-xr-x   2 root root 4096 Apr 24  2018 boot
drwxr-xr-x   1 root root 4096 Aug 17 11:52 copymyfiles
drwxr-xr-x   5 root root  360 Aug 17 11:52 dev
drwxr-xr-x   1 root root 4096 Aug 17 11:52 etc
drwxr-xr-x   2 root root 4096 Apr 24  2018 home
drwxr-xr-x   1 root root 4096 May 23  2017 lib
drwxr-xr-x   2 root root 4096 Jul 23 13:50 lib64
drwxr-xr-x   2 root root 4096 Jul 23 13:49 media
drwxr-xr-x   2 root root 4096 Jul 23 13:49 mnt
drwxr-xr-x   2 root root 4096 Jul 23 13:49 opt
dr-xr-xr-x 215 root root    0 Aug 17 11:52 proc
drwx------   2 root root 4096 Jul 23 13:50 root
drwxr-xr-x   5 root root 4096 Jul 23 13:50 run
drwxr-xr-x   2 root root 4096 Jul 23 13:50 sbin
drwxr-xr-x   2 root root 4096 Jul 23 13:49 srv
dr-xr-xr-x  13 root root    0 Aug 17 11:52 sys
drwxrwxrwt   1 root root 4096 Aug 17 11:52 tmp
drwxr-xr-x   1 root root 4096 Jul 23 13:49 usr
drwxr-xr-x   1 root root 4096 Jul 23 13:50 var
```

```
root@1f5aa2c59057:/# cd addmyfiles/
root@1f5aa2c59057:/addmyfiles# ls -al
total 12
drwxr-xr-x 1 root root 4096 Aug 17 11:52 .
drwxr-xr-x 1 root root 4096 Aug 17 11:52 ..
-rw-r--r-- 1 root root   24 Aug 17 11:49 add.html
root@1f5aa2c59057:/addmyfiles# cd ../copymyfiles/
root@1f5aa2c59057:/copymyfiles# ls -al
total 12
drwxr-xr-x 1 root root 4096 Aug 17 11:52 .
drwxr-xr-x 1 root root 4096 Aug 17 11:52 ..
-rw-r--r-- 1 root root   24 Aug 17 11:49 copy.html
```

`ADD`와 `COPY` 는 Dockerfile이 위치한 경로에 있는 파일을 이미지로 복사해오는 명령어 입니다. 겉보기에는 기능상에 큰 차이가 없어 보이고 위의 결과도 그렇게 보이는 듯 합니다. 하지만 `ADD` 추가적으로 두 가지의 기능을 가지고 있습니다.

1. URL을 통한 파일 복사
2. 로컬에서 압축파일 복사 시, 해제하여 복사
    - 웹에서 받은 압축파일은 압축해제만 되고 tar 형태는 유지됨

로컬에 있는 파일 뿐만 아니라 웹상에 있는 파일도 이미지를 빌드할 당시에 추가할 수 있는 것입니다. 또한 압축을 자동으로 해제하기 때문에 `tar`나 `tar.gz` 을 복사하는 경우 유의해야 합니다.