

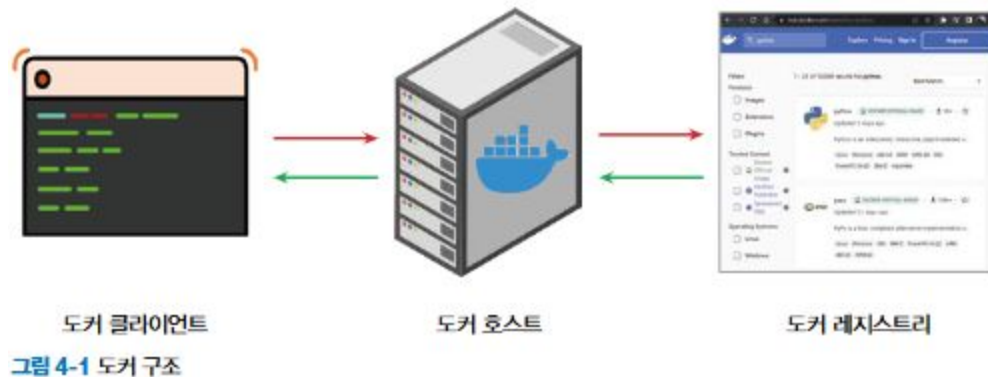
Chapter 4. 도커 기초

2024.10.05

4.1 도커 기초 개념

1. 도커 명령어를 입력했을 때 프로그램이 작동하는 방식
2. 도커 이미지
3. 도커 컨테이너

4.1.1 도커 작동 방식



도커 클라이언트: 도커에 명령을 내릴 수 있는 CLI(Command Line Interface)

컨테이너, 이미지, 볼륨 등을 관리

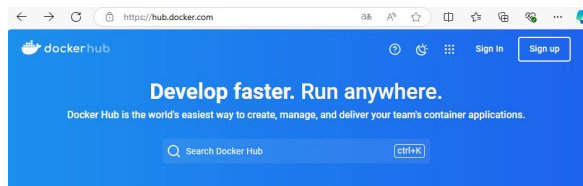
도커 호스트: 도커를 설치한 서버 or 가상머신 서버

도커 레지스트리: 도커 이미지를 저장하거나 배포하는 시스템

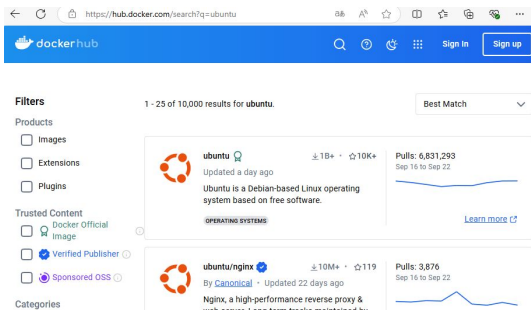
- Public 레지스트리: 예. 도커 허브(누구나 이미지 다운로드, 업로드 가능)
- Private

4.1.1 도커 작동 방식

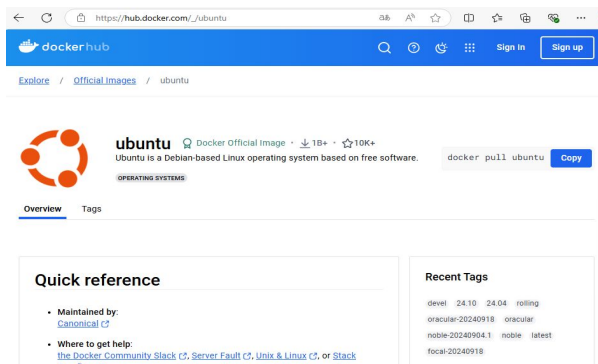
1. 도커 허브 웹사이트: <http://hub.docker.com>



2. ubuntu 검색



3. 도커 이미지 화면



도커 상세 구조

도커 클라이언트 명령어 입력

→ 도커 호스트의 데몬이 명령어 받음

→ 도커 호스트에 이미지가 없으면 도커 레지스트리에서 다운로드

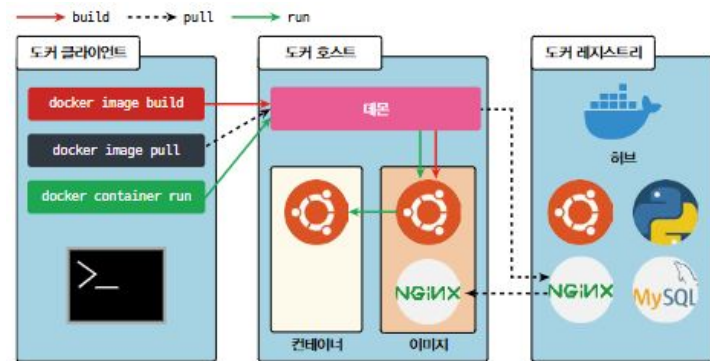


그림 4-5 도커 상세 구조

4.1.2 도커 이미지

도커 이미지:

- 컨테이너 형태
- 소프트웨어를 배포하기 위해 필요한 모든 요소(코드, 라이브러리, 설정 등)를 실행할 수 있는 포맷
- 컴파일 및 빌드한 패키지
- 독립적, 경량화된 패키지
- 특정 시점의 도커 컨테이너 상태를 담은 스냅샷(동일한 환경을 가진 여러개의 컨테이너를 손쉽게 생성)
- 여러 개의 레이어로 구성
- 도커 허브와 같은 중앙 저장소에 저장되어 관리(업로드, 다운로드 가능)

4.1.3 도커 컨테이너

도커 컨테이너:

- 도커 이미지를 실행할 수 있는 인스턴스
- 도커 이미지로부터 생성
- 도커 컨테이너에 대해 실행, 중지, 재실행, 삭제 등의 명령을 내릴 수 있음
- 자체 파일 시스템을 가지고, 각 컨테이너는 독립적 실행
- 가벼움
 - 자체적으로 운영체제 전부 포함하지 않음
 - 도커 엔진과 운영체제 공유(도커 엔진이 설치되어 있는 호스트 운영체제 이용)
 - 프로그램을 실행시키기 위해 최소한으로 필요한 바이너리, 라이브러리와 같은 구성 요소로 이뤄짐

4.1.4 hello world 실행 과정

```
eevee@myserver01:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
719385e32844: Pull complete
Digest: sha256:88ec0acaa3ec199d3b7eaf73588f4518c25f9d34f58ce9a0df68429c5af48e8d
Status: Downloaded newer image for hello-world:latest

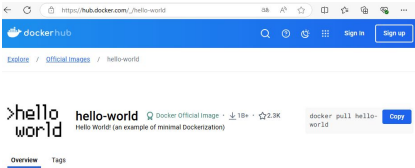
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

- 1 **docker run hello-world**에서 docker는 도커 관련 명령어를 입력하겠다는 의미입니다. 그리고 run은 컨테이너를 실행하겠다는 뜻이고 hello-world는 컨테이너 이름입니다.
 - 2 로컬에서 'hello-world:latest'라는 이미지를 찾을 수 없다는 뜻입니다. 우리는 'hello-world:latest'라는 이미지를 다운로드한 적이 없으므로 당연한 결과입니다.
 - 3 로컬에 'hello-world:latest'라는 이미지가 없으므로 library/hello-world에서 pull을 받겠다는 의미입니다. pull이란 도커 이미지를 원격 저장소에서 로컬로 다운로드하는 것으로 [그림 4-6]과 같은 도커 허브에서 hello-world 이미지를 다운로드한다는 것을 의미합니다.
- 
- 4 pull이 완료되었다는 의미입니다.
 - 5 도커 이미지들은 식별값으로 해시값을 갖게 됩니다. 참고로 Digest란 해시 함수를 거쳐 나온 후의 데이터를 의미합니다.
 - 6 도커 이미지 'hello-world:latest'의 다운로드가 완료되었음을 말합니다.
 - 7 해당 사이트는 도커 허브라고 부르는 사이트인데 해당 사이트를 이용하면 다양한 도커 이미지를 다운로드하거나 업로드할 수 있습니다.

4.2 도커 기초 명령어

도커 이미지, 컨테이너 다룰 수 있는 명령어 학습

4.2.1 도커 이미지 다운로드

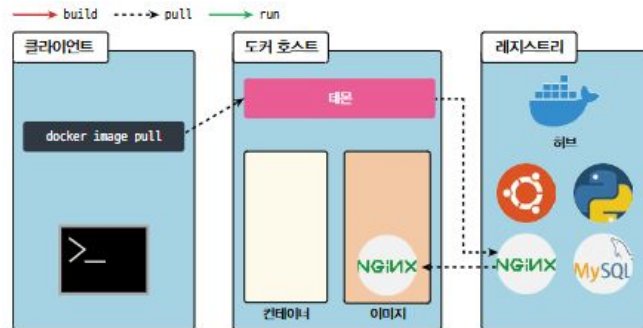


그림 4-7 도커 이미지의 다운로드 과정

`docker image pull {이미지 이름:태그 이름}`

도커 데몬이 도커 호스트에 해당 이미지 확인 후, 없으면 도커 레지스트리에서 다운로드

4.2.1 도커 이미지 다운로드

```
eevee@myserver01:~$ docker image pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
æce8493d397: Pull complete
Digest: sha256:2b7412e6465c3c7fc5bb21d3e6f1917c167358449fecac8176c6e496e5c1f85f
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

- 1 도커 이미지를 다운로드하기 위한 명령어를 입력합니다.
- 2 **docker image pull [이미지 이름:태그 이름]**을 입력하게 되는데 태그명을 입력하지 않으면 자동으로 latest 태그가 적용됩니다.
- 3 우분투 이미지의 latest 태그의 우분투 이미지를 다운로드한다는 메시지가 표시됩니다. 도커 허브에서 다양한 이미지 태그를 확인할 수 있습니다.
- 4 이미지 레이어 다운로드가 완료되었다는 Pull complete 메시지가 나타납니다. 이 메시지는 이미지 레이어의 개수만큼 나타납니다. Pull complete 메시지가 한 개 나타난 것을 보아 우분투 이미지는 한 개의 레이어로 구성되어 있음을 알 수 있습니다. 그리고 이때 해시값은 도커 이미지가 빌드될 때 생성된 ID입니다.
- 5 다운로드한 모든 레이어와 메타정보를 포함하는 이미지의 해시값을 나타냅니다.
- 6 해시값은 하나의 레이어에 대한 해시값을 나타내는 것과는 다르다는 것을 알 수 있습니다.
- 6 latest 태그를 통해 우분투 이미지를 다운로드했다는 상태 메시지를 확인할 수 있습니다.
- 7 끝으로 다운로드한 이미지의 URL이 나타납니다.

※ docker image pull {이미지 이름@DIGEST} → DIGEST 이미지 다운로드

4.2.1 도커 이미지 다운로드

파이썬 이미지 다운로드

```
eevee@myserver01:~$ docker image pull python:3.11.6
3.11.6: Pulling from library/python
8457fd5474e7: Pull complete
13baa2029dde: Pull complete
325c5bf4c2f2: Pull complete
7e18a660069f: Pull complete
98a59f0ffede: Pull complete
3a5444633a33: Pull complete
bbbc9b405dab: Pull complete
d9992232ef9b: Pull complete
Digest: sha256:6deadd529bed8232c98895a58fa8d689bdba285e9ceb92cda1f5fd8fa4a78fa1
Status: Downloaded newer image for python:3.11.6
docker.io/library/python:3.11.6
```

1. 앞선 실습과는 다르게 이미지 태그 이름을 명시해 3.11.6 태그의 파이썬 이미지를 다운로드합니다.
2. 이미지를 구성하는 각 레이어의 해시값을 확인할 수 있습니다.
3. 위 레이어들 모두 포함하는 이미지 DIGEST 값을 확인합니다. 이를 그림으로 나타내면 [그림 4-9]와 같습니다.

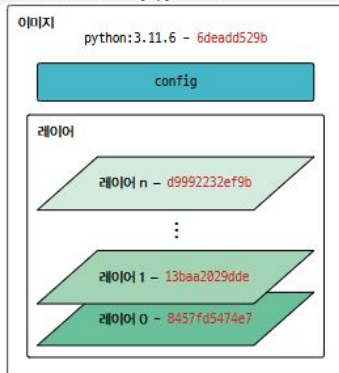


그림 4-9 이미지 구조

4.2.2 도커 이미지 상세 구조

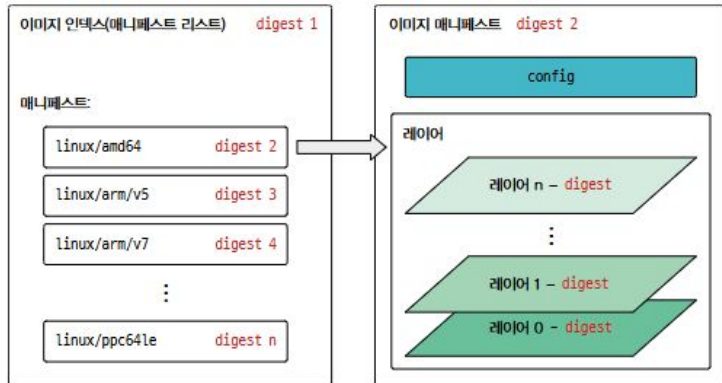


그림 4-10 이미지 상세 구조

이미지 구조:

- 이미지 인덱스: 이미지 다운로드할 때 결과창에 출력되는 digest, 다수 매니페스트로 구성
- 이미지 매니페스트: 다양한 운영체제 및 아키텍처에서 해당 이미지
활용할 수 있도록 설정값과 다양한 레이어 제공
- 레이어

※ 도커 허브의 이미지 태그 정보

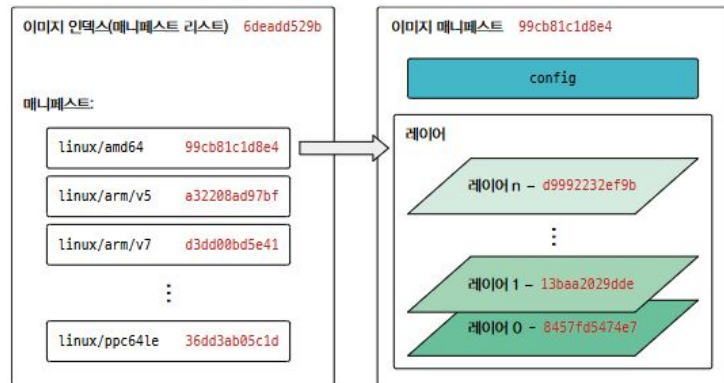
- 운영체제 및 아키텍처별 이미지 매니페스트 digest 확인 가능

TAG	OS/ARCH	DIGEST	VULNERABILITIES	COMPRESSED SIZE
3.11.6	linux/386	11c2b915850a	1 C 0 H 8 M 99 L	362.98 MB
	windows/amd64	8ba7d08c6a1	0 H 1 M 0 L	1.95 GB
	windows/amd64	06b5f6523f0	0 H 1 M 0 L	1.79 GB
	linux/amd64	99cb81c1d8e4	1 C 0 H 8 M 92 L	360.6 MB
	linux/arm/v5	a32208a97b7f	1 C 0 H 8 M 92 L	328.54 MB
	linux/arm/v7	d3d40b0d5e41	1 C 0 H 8 M 92 L	314.17 MB
	linux/arm64/v8	a7dc7501a494	1 C 0 H 8 M 92 L	351.59 MB
	linux/ppc64le	36d43ab05c1d	1 C 0 H 8 M 92 L	374.91 MB
	linux/s390x	668ec6c5388b	1 C 0 H 8 M 92 L	331.16 MB

그림 4-11 도커 허브 이미지 정보

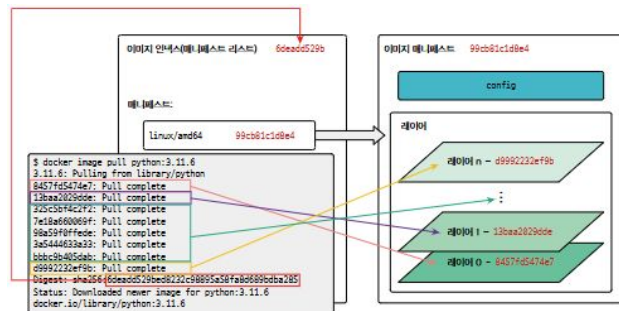
4.2.2 도커 이미지 상세 구조

python:3.11.6 이미지 상세 구조



다운로드한 이미지 출력 해시값: 이미지 인덱스

그림 4-12 python:3.11.6 이미지 구조




다운로드할 때 어떤 레이어를 다운로드하는지 확인

그림 4-13 docker image pull 명령어와 digest

4.2.3 도커 이미지 목록 확인

python:3.11.6 이미지 상세 구조



```
eevee@myserver01:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python	3.11.6	3f7984adbac4	2 weeks ago	1.01GB
ubuntu	latest	e4c58958181a	3 weeks ago	77.8MB
hello-world	latest	9c7a54a9a43c	6 months ago	13.3kB

- 1 **docker image ls** 명령어로 다운로드한 이미지 목록을 확인할 수 있습니다.
- 2 REPOSITORY는 이미지 이름을 의미하며 TAG는 이미지 태그를 의미합니다. IMAGE ID는 다운로드한 이미지의 ID를 나타내는데, 이때 IMAGE ID는 다운로드할 때의 DIGEST 값과 다르다는 것을 알 수 있습니다. 다운로드할 때의 DIGEST 값은 도커 레지스트리에 존재하는 이미지의 DIGEST 값이고, **docker image ls**의 결과값으로 나오는 IMAGE ID 값은 다운로드한 후에 로컬에서 할당받은 IMAGE ID 값에 해당합니다. CREATED는 이미지가 만들어진 시간을 의미하며, SIZE는 이미지 크기를 나타냅니다.
- 3 이번에 다운로드한 파이썬 이미지에 대한 정보를 확인할 수 있습니다. 이번 실습에서 파이썬 이미지는 3.11.6 태그 버전을 다운로드했습니다.
- 4 우분투 이미지에 대한 정보를 확인할 수 있습니다.
- 5 앞서 도커를 설치한 직후 다운로드한 hello-world 이미지를 확인할 수 있습니다.

4.2.4 도커 컨테이너 실행

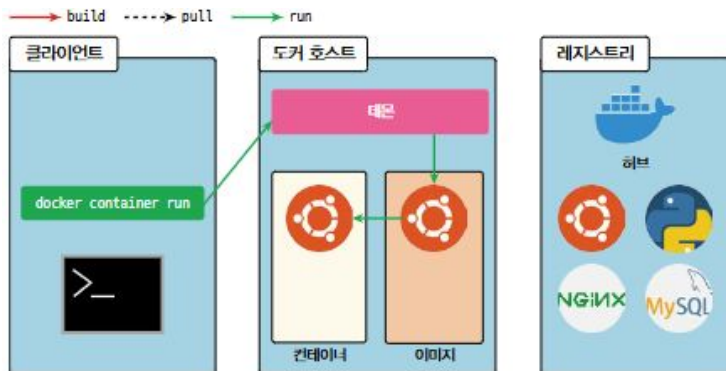


그림 4-14 docker container run

docker container run [이미지명]

- 호스트의 데몬이 실행 명령 받음
- 호스트 이미지를 컨테이너 형태로 실행

※ docker run [이미지명]: 초기 버전

```
eevee@myserver01:~$ docker container run python:3.11.6
eevee@myserver01:~$
```

4.2.5 도커 컨테이너 목록 확인

```
eevee@myserver01:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
209b1ac7a1be	python:3.11.6	"python3"	28 seconds ago	Exited (0)
26 seconds ago		musings_hamilton		
19e30e8d5a98	ubuntu	"/bin/bash"	39 seconds ago	Exited (0)
38 seconds ago		epic_goldwasser		
b68b3e20dc68	hello-world	"/hello"	About an hour ago	Exited (0)
About an hour ago		nervous_shannon		

`docker container ls -a`: a 옵션으로 실행 중 및 정지 상태 컨테이너 모두 확인

CONTAINER ID: 하나의 이미지로 다수 컨테이너 생성할 수 있으므로 각 컨테이너는 **CONTAINER ID**를 갖음

Exited (0): 컨테이너가 정상적(0)으로 종료

컨테이너 실행 후 컨테이너 내부 프로세스가 모두 종료되면 해당 컨테이너 역시 종료

※ `docker container ls`는 실행 중인 컨테이너만 보여줌

4.2.6 컨테이너 내부 접속

터미널 1

```
eevee@myserver01:~$ docker container run -it ubuntu
root@d76df685ca90:/# ls
bin  etc  lib32  media  proc  sbin  tmp  boot  home  lib64
mnt  root  srv   usr  dev  lib   libx32  opt  run  sys  var
```

터미널 2

```
eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS
d76df685ca90   ubuntu   "bash"    About a minute ago   Up         About a minute
hungry_gauss
```

터미널 2

```
eevee@myserver01:~$ docker container stop d76df685ca90
d78c2cbe3001
```

터미널 1

```
eevee@myserver01:~$ docker container start d76df685ca90
d76df685ca90

eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
d76df685ca90   ubuntu   "bash"    12 minutes ago   Up         5 seconds   hungry_gauss

eevee@myserver01:~$ docker container attach d76df685ca90
root@d76df685ca90:/# exit
exit
eevee@myserver01:~$
```

① `-it` 옵션에서 `i`는 interactive의 줄임말로 표준 입력(STDIN)을 열어놓는다는 의미이며, `t`는 tty의 줄임말로 가상 터미널을 의미합니다. 즉 `-it` 옵션을 활용하면 가상 터미널을 통해 키보드 입력을 표준 입력으로 컨테이너에 전달할 수 있는 것입니다.

② 사용자 이름과 호스트 이름이 변경된 것을 알 수 있습니다. 이때 사용자 이름은 root이고 호스트 이름은 컨테이너 ID 인 것을 알 수 있습니다. `ls` 명령어를 입력하면 컨테이너 내부의 파일 시스템을 확인할 수 있습니다.

Up: 컨테이너 실행 중

컨테이너 종료

- 터미널 1에서 `exit`
- 터미널 2에서 `docker container stop` [컨테이너ID]: 10초 후 종료
- 터미널 2에서 `docker container kill` [컨테이너ID]: 즉시 종료

① `start` 명령어를 이용해 컨테이너를 실행합니다.

② 실행 중인 컨테이너를 확인할 수 있습니다.

③ `attach` 명령어를 이용해 내부에 접속할 수 있습니다.

④ 접속을 종료하여 실행을 마칩니다.

4.2.7 컨테이너 삭제

```
eevee@myserver01:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
d76df685ca90	ubuntu	"bash"	15 minutes ago	Exited (0)
209b1ac7a1be	python:3.11.6	"python3"	18 minutes ago	Exited (0)
19e30e8d5a98	ubuntu	"/bin/bash"	19 minutes ago	Exited (0)

컨테이너 목록 확인

```
eevee@myserver01:~$ docker container rm d76df685ca90
```

컨테이너 ID d76df685ca90 삭제

```
eevee@myserver01:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
209b1ac7a1be	python:3.11.6	"python3"	20 minutes ago	Exited (0)
19e30e8d5a98	ubuntu	"/bin/bash"	20 minutes ago	Exited (0)

컨테이너 목록 재확인

```
eevee@myserver01:~$ docker container rm 19e30e8d5a98 b68b3e20dc68
```

```
eevee@myserver01:~$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
209b1ac7a1be	python:3.11.6	"python3"	21 minutes ago	Exited (0)

컨테이너 다수 삭제

4.2.8 이미지 삭제

```
eevee@myserver01:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python	3.11.6	3f7984adbac4	2 weeks ago	1.01GB
ubuntu	latest	e4c58958181a	3 weeks ago	77.8MB
hello-world	latest	9c7a54a9a43c	6 months ago	13.3kB

❶ 이미지 목록을 확인합니다. 위와 같은 이미지 중 hello-world 이미지를 삭제하겠습니다.

```
eevee@myserver01:~$ docker image rm 9c7a54a9a43c
```

Untagged: hello-world:latest
Deleted: sha256:88ec0acaa3ec199d3b7eaf73588f4518c25f9d34f58ce9a0df68429c5af48e8d
Deleted: sha256:9c7a54a9a43cca047013b82af109fe963fde787f63f9e016fdc3384500c2823d
Deleted: sha256:01bb4fce3eb1b56b05adf99504dafd31907a5aadac736e36b27595c8b92f07f1

❶ docker image rm [이미지 ID]를 입력하면 해당 이미지를 삭제할 수 있습니다. 이번 실습에서는 hello-world 이미지에 해당하는 이미지 ID를 입력했습니다.

```
eevee@myserver01:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python	3.11.6	3f7984adbac4	2 weeks ago	1.01GB
ubuntu	latest	e4c58958181a	3 weeks ago	77.8MB

❶ 이미지 목록을 다시 확인하면 hello-world 이미지가 삭제된 것을 확인할 수 있습니다.

4.2.9 도커 이미지 변경

기존 도커 이미지 수정 후, 새로운 이미지 생성

```
터미널 1
eevee@myserver01:~$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
python 3.11.6 3f7984dbac4 2 weeks ago 1.01GB
ubuntu latest e4c58958181a 3 weeks ago 77.8MB
eevee@myserver01:~$ docker container run -it ubuntu
root@67f64db65b10:/#
```

- 1 첫 번째 터미널을 열고 이미지를 확인합니다.
- 2 -it 옵션을 사용해 우분투 컨테이너를 실행합니다.
- 3 해당 컨테이너 내부에 접속한 것을 알 수 있습니다.

```
터미널 1
root@67f64db65b10:/# ifconfig
bash: ifconfig: command not found

root@67f64db65b10:/# apt update && apt install net-tools
Get:1 http://security.ubuntu.com/ubuntu jammy-security InRelease [119 kB]
Get:2 http://archive.ubuntu.com/ubuntu jammy InRelease [270 kB]
Get:3 http://archive.ubuntu.com/ubuntu jammy-updates InRelease [119 kB]
...
Unpacking net-tools (1.60+git20181103.0eebece-1ubuntu5) ...
Setting up net-tools (1.60+git20181103.0eebece-1ubuntu5) ...
```

- 1 컨테이너 내부 IP를 확인하려고 ifconfig 명령어를 입력한다고 해서 ifconfig 명령어를 사용할 수는 없습니다. ifconfig 명령어를 사용하려면 net-tools를 설치해야 합니다.
- 2 따라서 해당 명령어를 입력하면 net-tools가 설치됩니다.

```
터미널 1
root@67f64db65b10:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.2 netmask 255.255.0.0 broadcast 172.17.255.255
    ether 02:42:ac:11:00:02 txqueuelen 0 (Ethernet)
    RX packets 6174 bytes 28783616 (28.7 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 3977 bytes 219865 (219.8 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

위 실습은 컨테이너 IP를 확인하는 내용입니다.

- 1 ifconfig 명령어를 입력합니다.
- 2 컨테이너의 IP가 172.17.0.2인 것을 확인할 수 있습니다.

```
터미널 2
eevee@myserver01:~$ docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
67f64db65b10 ubuntu "bash" 5 minutes ago Up 5 minutes upbeat_noyce
```

이를 위해 기존 터미널 1을 종료하지 않고 터미널 2를 실행합니다.

- 1 컨테이너 목록을 확인하면 컨테이너가 실행 중임을 확인할 수 있습니다.

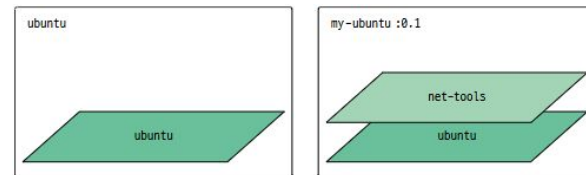


그림 4-17 이미지 변경

```
터미널 2
eevee@myserver01:~$ docker container commit 67f64db65b10 my-ubuntu:0.1
sha256:8252e4215d83f9d3ee56d21232d441306557cd6d1dfedd275f08ec824b8ebf96
```

4.2.9 도커 이미지 변경

```
터미널 1
root@67f64db65b10:/# exit
exit

eevee@myserver01:~$ docker container ls -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS         NAMES
67f64db65b10   ubuntu        "bash"                  7 minutes ago   Exited (0)
31 seconds ago   upbeat_noyce
209b1ac7a1be   python:3.11.6 "python3"               41 minutes ago   Exited (0)
41 minutes ago   musing_hamilton

eevee@myserver01:~$ docker container rm 67f64db65b10
67f64db65b10

eevee@myserver01:~$ docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
my-ubuntu     0.1       8252e4215d03   About a minute ago   125MB
python        3.11.6    3f7984adbac4   2 weeks ago       1.01GB
ubuntu        latest    e4c58958181a   3 weeks ago       77.8MB

eevee@myserver01:~$ docker container run -it my-ubuntu:0.1

root@eafefe31241c:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 172.17.0.2  netmask 255.255.0.0  broadcast 172.17.255.255
    ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
    RX packets 9  bytes 806 (806.0 B)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 0  bytes 0 (0.0 B)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
    inet 127.0.0.1  netmask 255.0.0.0
    loop  txqueuelen 1000  (Local Loopback)
    RX packets 0  bytes 0 (0.0 B)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 0  bytes 0 (0.0 B)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@eafefe31241c:/# exit
exit
```

- 1 기존 컨테이너가 실행 중인 터미널 1에서 컨테이너를 빠져나옵니다.
- 2 컨테이너 전체 목록을 확인합니다.
- 3 기존 우분투 컨테이너를 삭제합니다.
- 4 이미지 목록을 확인하면 앞서 생성한 my-ubuntu:0.1 이미지가 생성되어 있는 것을 볼 수 있습니다.
- 5 새롭게 만든 my-ubuntu:0.1 이미지를 이용해 -it 옵션을 활용해 컨테이너를 실행합니다.
- 6 이 과정을 통해 컨테이너 내부에 접속하게 되는데 ifconfig 명령어를 통해 IP 주소를 확인해 봅니다. 앞서 기본적인 ubuntu 컨테이너에서는 ifconfig 명령어를 사용할 수 없었던 반면 my-ubuntu 이미지를 이용해 생성한 컨테이너에서는 ifconfig 명령어를 사용할 수 있다는 사실을 확인할 수 있습니다.
- 7 컨테이너를 빠져나옵니다.

4.2.10 도커 이미지 명령어 모음

명령어	설명
<code>docker image build</code>	Dockerfile로부터 이미지 빌드합니다.
<code>docker image history</code>	이미지 히스토리를 확인합니다.
<code>docker image import</code>	파일 시스템 이미지 생성을 위한 타볼 ^{tarball} 콘텐츠를 임포트합니다.
<code>docker image inspect</code>	이미지 정보를 표시합니다.
<code>docker image load</code>	타볼로 묶인 이미지를 로드합니다.
<code>docker image ls</code>	이미지 목록을 확인합니다.
<code>docker image prune</code>	사용하지 않는 이미지를 삭제합니다.
<code>docker image pull</code>	레지스트리로부터 이미지를 다운로드합니다.
<code>docker image push</code>	레지스트리로 이미지를 업로드합니다.
<code>docker image rm</code>	하나 이상의 이미지를 삭제합니다.
<code>docker image save</code>	이미지를 타볼로 저장합니다.
<code>docker image tag</code>	이미지 태그를 생성합니다.

표 4-1 도커 이미지 명령어

4.2.10 도커 컨테이너 명령어 모음

명령어	설명
docker container attach	실행 중인 컨테이너의 표준 입출력 스트림에 붙습니다(attach).
docker container commit	변경된 컨테이너에 대한 새로운 이미지를 생성합니다.
docker container cp	컨테이너와 로컬 파일 시스템 간 파일/폴더를 복사합니다.
docker container create	새로운 컨테이너를 생성합니다.
docker container diff	컨테이너 파일 시스템의 변경 내용을 검사합니다.
docker container exec	실행 중인 컨테이너에 명령어를 실행합니다.
docker container export	컨테이너 파일 시스템을 타블로 추출합니다.
docker container inspect	하나 이상의 컨테이너의 자세한 정보를 표시합니다.
docker container kill	하나 이상의 실행 중인 컨테이너를 kill합니다.
docker container logs	컨테이너 로그를 불러옵니다.
docker container ls	컨테이너 목록을 확인합니다.
docker container pause	하나 이상의 컨테이너 내부의 모든 프로세스를 정지합니다.
docker container port	특정 컨테이너의 매핑된 포트 리스트를 확인합니다.
docker container prune	멈춰 있는(stopped) 모든 컨테이너를 삭제합니다.
docker container rename	컨테이너 이름을 다시 찾습니다.
docker container restart	하나 이상의 컨테이너를 재실행합니다.
docker container rm	하나 이상의 컨테이너를 삭제합니다.
docker container run	이미지로부터 컨테이너를 생성하고 실행합니다.
docker container start	멈춰 있는 하나 이상의 컨테이너를 실행합니다.
docker container stats	컨테이너 리소스 사용 통계를 표시합니다.
docker container stop	하나 이상의 실행 중인 컨테이너를 정지합니다.
docker container top	컨테이너의 실행 중인 프로세스를 표시합니다.
docker container unpause	컨테이너 내부의 멈춰 있는 프로세스를 재실행합니다.
docker container update	하나 이상의 컨테이너 설정을 업데이트합니다.
docker container wait	컨테이너가 종료될 때까지 기다린 후 exit code를 표시합니다.

표 4-2 도커 컨테이너 명령어

- docker container create [이미지ID]: 새로운 컨테이너 생성
- docker container start [컨테이너ID]: 정지 상태 컨테이너 실행
- docker container run: 컨테이너 생성 후 실행, **create**와 **start** 합친 명령어
- docker container exec: 실행 중인 컨테이너 내부에서 명령어 실행, 새로운 프로세스 시작해서 컨테이너 내 작업 수행
- docker container attach: 실행 중인 컨테이너의 표준 입력(stdin), 표준 출력(stdout), 표준 오류(stderr) 스트림에 연결할 때 사용, 기존 실행 중 프로세스에 연결

4.3 도커 컨테이너 네트워크

- 도커 컨테이너가 통신하는 방법
- 호스트와 컨테이너 간 파일 전송 구조 이해

4.3.1 도커 컨테이너 네트워크 구조

도커 호스트와 도커 컨테이너의 네트워크 구성 확인

터미널 1

```
eevee@myserver01:~$ docker container ls -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS         NAMES
eafefe31241c   my-ubuntu:0.1 "bash"                  8 minutes ago Exited (0)
7 minutes ago optimistic_germain
209b1ac7a1be   python:3.11.6 "python3"               50 minutes ago Exited (0)
50 minutes ago musing_hamilton
```

❶ 터미널 1에서 컨테이너 전체 목록을 확인합니다. 이번 실습에서는 앞서 my-ubuntu:0.1 이미지를 활용해 만든 eafefe31241c 컨테이너를 다시 실행할 것입니다. 만약 앞선 실습 이후에 컨테이너를 삭제했다면 이미지를 활용해 **docker container run -it my-ubuntu:0.1** 명령어를 활용해 다시 실행하면 됩니다.

터미널 1

```
eevee@myserver01:~$ docker container start eafefe31241c
eafefe31241c

eevee@myserver01:~$ docker container attach eafefe31241c
```

```
root@eafefe31241c:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 172.17.0.2  netmask 255.255.0.0  broadcast 172.17.255.255
    ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
    RX packets 10  bytes 876 (876.0 B)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 0  bytes 0 (0.0 B)
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
    inet 127.0.0.1  netmask 255.0.0.0
    loop txqueuelen 1000  (Local Loopback)
    RX packets 0  bytes 0 (0.0 B)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 0  bytes 0 (0.0 B)
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0
```

- ❶ container start 명령어를 이용해 my-ubuntu 이미지로 생성한 컨테이너를 가동합니다.
- ❷ attach 명령어를 이용해 컨테이너 내부에 접속합니다.
- ❸ ifconfig 명령어를 이용해 IP 주소를 확인합니다.
- ❹ 컨테이너 내부 IP는 172.17.0.2라는 것을 알 수 있습니다.

4.3.1 도커 컨테이너 네트워크 구조

```
터미널 2
eevee@myserver01:~$ ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    inet6 fe80::42:92ff:fe10:c81e prefixlen 64 scopeid 0x20<link>
    ether 02:42:92:10:c8:1e txqueuelen 0 (Ethernet)
    RX packets 3977 bytes 164187 (164.1 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 6181 bytes 28784730 (28.7 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.4 netmask 255.255.255.0 broadcast 10.0.2.255
    inet6 fe80::a00:27ff:fec3:3b5d prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:c3:3b:5d txqueuelen 1000 (Ethernet)
    RX packets 389523 bytes 583658414 (583.6 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 64305 bytes 4379973 (4.3 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 332 bytes 36527 (36.5 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 332 bytes 36527 (36.5 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth9816136: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::38a1:eaff:fe2e:a926 prefixlen 64 scopeid 0x20<link>
    ether 3a:a1:ea:2e:a9:26 txqueuelen 0 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 13 bytes 1086 (1.0 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

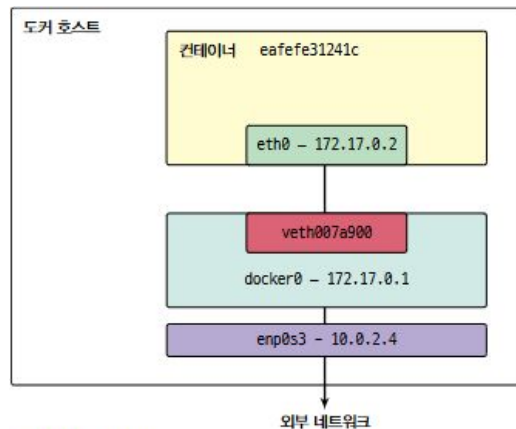


그림 4-18 도커 네트워크

도커 호스트와 컨테이너 간의 네트워크 구조

도커 호스트 안에 docker0, enp0s3, veth007a900 인터페이스

- docker0: 도커 호스트와 컨테이너 연결하는 bridge veth 가상 인터페이스가 eth0와 docker0 연결
- enp0s3: 도커 호스트 자체 보유한 네트워크 인터페이스

4.3.2 도커 네트워크 확인

```
터미널 2
eevee@myserver01:~$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
be8f8b63505a	bridge	bridge	local
2d1a75c7bcc7	host	host	local
0feadfe2e928	none	null	local

터미널 2에서 `docker network ls`를 입력하면 도커 네트워크를 확인할 수 있습니다. 위 결과와 같이 도커에서는 기본적으로 bridge, host, none이라는 세 가지 네트워크 드라이버를 제공합니다.

- **bridge 드라이버:** 컨테이너를 생성할 때 제공하는 기본 드라이버이며 각 컨테이너는 각각의 네트워크 인터페이스를 가집니다. 이는 도커 호스트의 docker0과 비인딩됩니다. bridge 드라이버는 컨테이너 생성 시 사용되는 기본 네트워크 드라이버이므로 우리가 지금까지 컨테이너를 생성할 때는 모두 bridge 드라이버를 사용한 것입니다. bridge 드라이버는 따로 옵션을 사용하지 않아도 적용되지만 `--network=bridge` 옵션을 사용할 수도 있습니다.
- **host 드라이버:** 컨테이너를 생성할 때 컨테이너 자체적으로 네트워크 인터페이스를 가지지 않고 호스트 네트워크 인터페이스를 공유합니다. host 드라이버를 사용하려면 컨테이너 실행 시 `--network=host`를 사용합니다. 다음은 host 드라이버를 활용해 컨테이너를 실행한 것입니다.
- **none 드라이버:** 실행한 컨테이너가 네트워크 인터페이스를 가지지 않아 컨테이너 외부와의 통신이 불가능합니다. none 드라이버를 사용하려면 컨테이너 실행 시 `--network=none` 옵션을 사용합니다. 다음은 none 드라이버를 활용해 컨테이너를 생성한 것입니다.

host 드라이버로 컨테이너 실행 후 네트워크 정보 확인

```
터미널 1
eevee@myserver01:~$ docker container run -it --network=host my-ubuntu:0.1
root@myserver01:/# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    inet6 fe80::42:92ff:fe18:c81e prefixlen 64 scopeid 0x20<link>
    ether 02:42:92:18:c8:1e txqueuelen 0 (Ethernet)
    RX packets 3977 bytes 164187 (164.1 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 6181 bytes 28784730 (28.7 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.4 netmask 255.255.255.0 broadcast 10.0.2.255
    inet6 fe80::a00:27ff:fec3:3b5d prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:c3:3b:5d txqueuelen 1000 (Ethernet)
    RX packets 389578 bytes 583664246 (583.6 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 64345 bytes 4387413 (4.3 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 332 bytes 36527 (36.5 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 332 bytes 36527 (36.5 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@myserver01:/# exit
exit
```

none 드라이버로 컨테이너 실행 후 네트워크 정보 확인

네트워크 인터페이스 없음

```
터미널 1
eevee@myserver01:~$ docker container run -it --network=none my-ubuntu:0.1
root@07247fe4c3c1:/# ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@07247fe4c3c1:/# exit
exit
```

4.3.3 호스트에서 컨테이너로 파일 전송

도커 호스트에서 컨테이너로 파일 전송

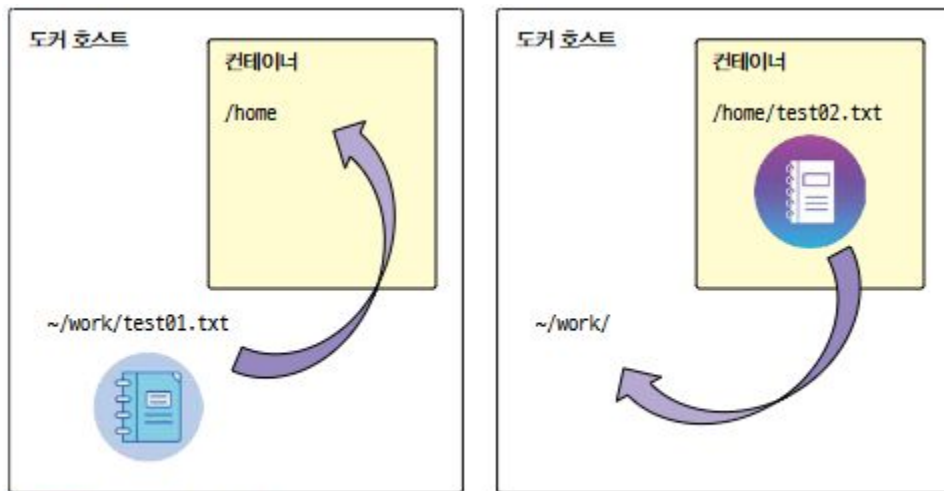


그림 4-19 호스트, 컨테이너 간 파일 전송

4.3.3 호스트에서 컨테이너로 파일 전송

터미널1: 호스트(호스트에 존재하는 파일을 컨테이너로 전송)

터미널2: 컨테이너 내부(컨테이너에 전송된 파일 확인)

 **터미널 1**

```
eevee@myserver01:~$ cd work/  
eevee@myserver01:~/work$ mkdir ch04  
eevee@myserver01:~/work$ ls  
ch04  
eevee@myserver01:~/work$ cd ch04/  
eevee@myserver01:~/work/ch04$ mkdir ex01  
eevee@myserver01:~/work/ch04$ ls  
ex01  
eevee@myserver01:~/work/ch04$ cd ex01/  
eevee@myserver01:~/work/ch04/ex01$ vim test01.txt  
Hello, I am Cheolwon.  
eevee@myserver01:~/work/ch04/ex01$ cat test01.txt  
Hello, I am Cheolwon.  
eevee@myserver01:~/work/ch04/ex01$ pwd  
/home/eevee/work/ch04/ex01
```

- 1 work 디렉터리로 이동합니다.
- 2 4장 실습 파일을 저장할 디렉터를 만들어줍니다.
- 3 파일 목록을 확인합니다.
- 4 ch04 디렉터리로 이동합니다.
- 5 이번 실습에 사용할 파일을 저장할 ex01이라는 디렉터를 만듭니다.
- 6 제대로 만들어졌는지 확인합니다.
- 7 ex01 디렉터리로 이동합니다.
- 8 vim 편집기를 통해 test01.txt라는 파일을 만들고 적절한 문구를 입력합니다.
- 9 cat 명령어를 통해 해당 파일 내용을 출력하면 데이터가 올바르게 저장되어 있는 것을 알 수 있습니다.
- 10 pwd 명령어를 통해 현재 위치를 파악합니다.

4.3.3 호스트에서 컨테이너로 파일 전송

터미널 2

```
eevee@myserver01:~$ docker container run -it ubuntu
root@fdf411cb471e:/# ls
bin  dev  home  lib32  libx32  mnt  proc  run  srv  tmp  var
boot  etc  lib  lib64  media  opt  root  sbin  sys  usr
root@fdf411cb471e:/# cd home/
root@fdf411cb471e:/home# ls
root@fdf411cb471e:/home#
```

다음 실습은 터미널 2에서 수행합니다.

- 1 ubuntu 컨테이너를 실행합니다.
- 2 컨테이너 내부에서 ls 명령어를 통해 파일 목록을 출력합니다.
- 3 home 디렉터리로 이동합니다.
- 4 ls 명령어를 입력하면 home 디렉터리 내부에는 파일이 존재하지 않는 것을 알 수 있습니다.

4.3.3 호스트에서 컨테이너로 파일 전송

```
터미널 1
eevee@myserver01:~/work/ch04/ex01$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS        NAMES
fdf411cb471e   ubuntu   "/bin/bash"             About a minute Up           About a minute
musing_brattain

eevee@myserver01:~/work/ch04/ex01$ docker container cp ./test01.txt fdf411cb471e:/home
Successfully copied 2.05kB to fdf411cb471e:/home

eevee@myserver01:~/work/ch04/ex01$
```

```
터미널 2
root@fdf411cb471e:/home# ls
test01.txt
root@fdf411cb471e:/home# cat test01.txt
Hello, I am Cheolwon.
```

터미널 2로 이동합니다.

- ① 파일 목록을 확인하면 test01.txt 파일이 존재하는 것을 볼 수 있습니다.
- ② 내용물을 출력하면 앞서 생성한 파일임을 확인할 수 있습니다.

다시 터미널 1로 돌아옵니다.

- ① 터미널 1에서 실행 중인 컨테이너 목록을 확인하면 터미널 2에서 실행했던 컨테이너가 실행 중임을 알 수 있습니다.
- ② `docker container cp` 명령어를 통해 호스트에 존재하는 파일을 컨테이너 내부로 복사합니다. 파일 복사 명령어는 다음과 같습니다.

`docker container cp` [출발 경로/보내고 싶은 파일명] [도착 컨테이너/파일 저장 경로]

4.3.4 컨테이너에서 호스트로 파일 전송

```
터미널 2
root@fdf411cb471e:/home# ls
test01.txt
root@fdf411cb471e:/home# cp test01.txt test02.txt
root@fdf411cb471e:/home# ls
test01.txt  test02.txt
```

- 1 터미널 2에서 파일 목록을 확인하면 이전에 호스트로 받은 test01.txt 파일을 확인할 수 있습니다.
- 2 cp 명령어를 통해 test01.txt 파일을 복사해 test02.txt 파일을 생성합니다.
- 3 다시 파일 목록을 확인하면 test02.txt 파일이 생성된 것을 볼 수 있습니다. 우리는 이렇게 생성한 test02.txt 파일을 호스트로 복사할 것입니다. 터미널 1로 이동합니다.

```
터미널 1
eevee@myserver01:~/work/ch04/ex01$ pwd
/home/eevee/work/ch04/ex01

eevee@myserver01:~/work/ch04/ex01$ ls
test01.txt

eevee@myserver01:~/work/ch04/ex01$ docker cp fdf411cb471e:/home/test02.txt /home/
eevee/work/ch04/ex01

Successfully copied 2.05kB to /home/eevee/work/ch04/ex01

eevee@myserver01:~/work/ch04/ex01$ ls
test01.txt  test02.txt

eevee@myserver01:~/work/ch04/ex01$ cat test02.txt
Hello, I am Cheolwon.
```

- 1 컨테이너로부터 전송받은 파일을 저장할 호스트 경로를 터미널 1에서 확인합니다.
- 2 파일 목록을 확인하면 이전에 생성한 test01.txt 파일만 존재하는 것을 알 수 있습니다.
- 3 docker cp 명령어를 통해 파일을 복사합니다. 출력 메시지를 보면 성공적으로 복사된 것을 볼 수 있습니다.
- 4 파일 목록을 확인하면 test02.txt 파일이 복사되어 있는 것을 알 수 있습니다.
- 5 파일 내용이 정확히 출력되는 것을 알 수 있습니다.

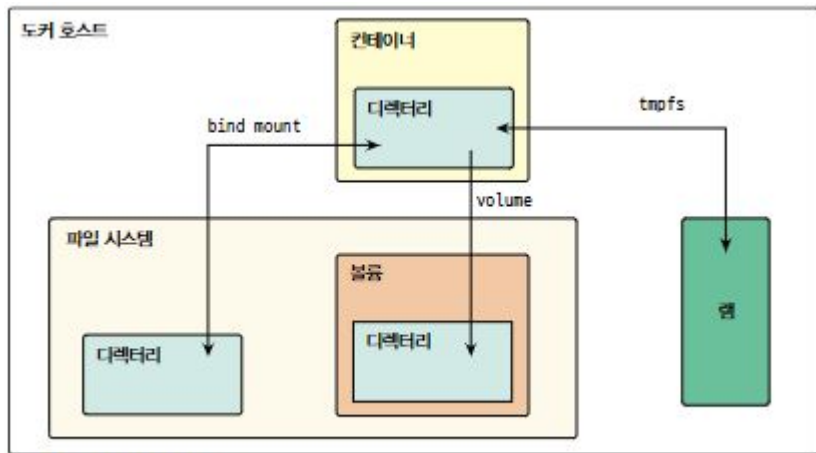
4.4 도커 스토리지

- 도커 스토리지 개념과 필요성
- 도커 스토리지 종류

4.4.1 도커 스토리지의 개념

컨테이너 삭제되면 컨테이너 내부에 파일도 함께 삭제

도커 스토리지: 도커 컨테이너에서 생성되는 데이터를 보존하기 위해...



도커 스토리지 종류

- **bind mount:** 도커 호스트 디렉터리 직접 공유
- **volume:** 도커를 활용해 볼륨 생성 후 컨테이너의 디렉터리와 공유
- **tmpfs:** 도커 호스트 메모리에 파일이 저장되는 방식, 컨테이너를 삭제하면 해당 파일도 삭제

그림 4-20 도커 볼륨의 종류

4.4.2 도커 스토리지의 필요성

PostgreSQL 데이터베이스 사용 실습

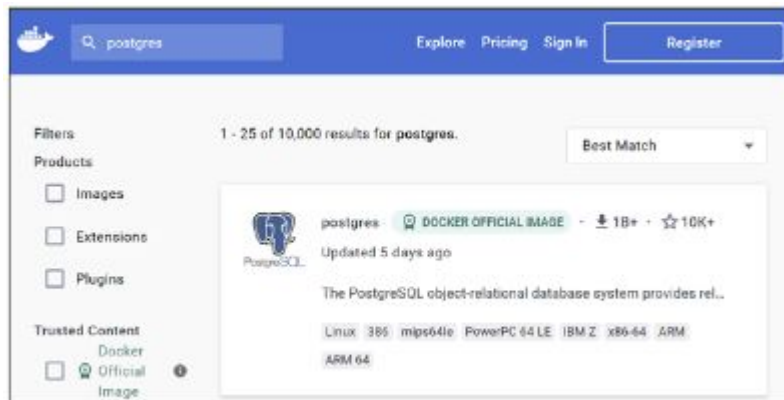


그림 4-21 도커 허브에서 PostgreSQL 확인

```
eevee@myserver01:~$ docker image pull postgres
Using default tag: latest
latest: Pulling from library/postgres
578acb154839: Pull complete
8a9a8dd839ec: Pull complete
...(중략)
f560313d0d85: Pull complete
fd463ca121fb: Pull complete
Digest: sha256:d8048a55b0952189c0f6fcc89ec46d78e1b56ac25eae0b2b85e449a57aa0ce44
Status: Downloaded newer image for postgres:latest
docker.io/library/postgres:latest
```

```
eevee@myserver01:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
my-ubuntu	0.1	8252e4215d03	46 minutes ago	125MB
python	3.11.6	3f7984adbac4	2 weeks ago	1.01GB
ubuntu	latest	e4c58958181a	3 weeks ago	77.8MB
postgres	latest	fbd1be2cbb1f	6 weeks ago	417MB

① 이 명령어로 이미지 목록을 확인합니다.

② 그러면 postgres 이미지를 확인할 수 있습니다.

4.4.2 도커 스토리지의 필요성

```
eevee@myserver01:~$ docker container run --name some-postgres -e POSTGRES_
PASSWORD=mysecretpassword -d postgres
a9b35064d02f41a63b6ef1c54fd00b3b2495d8c984064050dbd633a0937bd495

eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED      STATUS
PORTS          NAMES
a9b35064d02f   postgres  "docker-entrypoint.s..." 8 seconds ago Up 8 seconds
5432/tcp      some-postgres
```

- 1 컨테이너를 실행하는 명령어입니다. --name 옵션은 컨테이너 이름을 의미하며 -e는 환경 변수를 설정하는 옵션입니다. 이번 실습에서는 PostgreSQL의 비밀번호를 입력합니다. -d 옵션은 백그라운드 실행을 의미합니다.
- 2 컨테이너 목록을 확인하면 PostgreSQL 컨테이너가 실행 중임을 확인할 수 있습니다.

```
eevee@myserver01:~$ docker container exec -it a9b35064d02f /bin/bash
root@a9b35064d02f:/# psql -U postgres
psql (16.0 (Debian 16.0-1.pgdgl20+1))
Type "help" for help.

postgres=# CREATE USER user01 PASSWORD '1234' SUPERUSER;
CREATE ROLE

postgres=# CREATE DATABASE test01 OWNER user01;
CREATE DATABASE

postgres=# \c test01 user01
You are now connected to database "test01" as user "user01".

test01=# CREATE TABLE table01(
id INTEGER PRIMARY KEY,
name VARCHAR(20)
);
CREATE TABLE

test01=# \dt
List of relations
Schema | Name   | Type | Owner
```

```
public | table01 | table | user01
(1 row)

test01=# SELECT * FROM table01;
id | name
-----
(0 rows)

test01=# INSERT INTO table01 (id, name)
VALUES(
1,
'Cheolwon'
);
INSERT 0 1

test01=# SELECT * FROM table01;
id | name
-----
1 | Cheolwon
(1 row)

test01=# \q
root@a9b35064d02f:/# exit
exit
eevee@myserver01:~$
```

위 실습은 PostgreSQL에 접속해서 사용자, 데이터베이스, 테이블을 생성하는 코드입니다.

- 1 먼저 `exec -it` 옵션을 이용해 실행 중인 컨테이너 내부에 접속합니다.
- 2 접속한 컨테이너 내부에서 `psql` 명령어를 이용해 postgres 계정으로 PostgreSQL에 접속합니다. 이때 -U는 username을 의미합니다.
- 3 SUPERUSER 권한을 부여한 user01이라는 새로운 사용자를 생성합니다.
- 4 test01이라는 새로운 데이터베이스를 생성합니다. 소유자는 user01이라고 설정합니다.
- 5 실습이 끝났으니 user01으로 test01 데이터베이스에 접속합니다.
- 6 test01 데이터베이스에 table01이라는 간단한 테이블을 생성합니다.
- 7 테이블 목록을 확인하면 방금 생성한 table01 테이블을 확인할 수 있습니다.
- 8 table01 테이블의 데이터를 조회하면 아무것도 없는 것을 알 수 있습니다.
- 9 INSERT INTO를 이용해 table01에 데이터를 삽입합니다.

- 10 다시 table01 테이블의 데이터를 조회하면 데이터를 확인할 수 있습니다.
- 11 실습이 끝났으니 PostgreSQL에서 빠져나갑니다.
- 12 컨테이너에서도 빠져나갑니다.

```
eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED      STATUS
PORTS          NAMES
a9b35064d02f   postgres  "docker-entrypoint.s..." 14 minutes ago Up
14 minutes     5432/tcp   some-postgres

eevee@myserver01:~$ docker container stop a9b35064d02f
a9b35064d02f

eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED      STATUS
PORTS          NAMES
a9b35064d02f   postgres  "docker-entrypoint.s..." 15 minutes ago Exited (0)
12 seconds ago   some-postgres

eevee@myserver01:~$ docker container start a9b35064d02f
a9b35064d02f
```

위 실습은 실행 중인 PostgreSQL 컨테이너를 정지시키는 코드입니다.

- 1 실행 중인 컨테이너를 확인하면 PostgreSQL 컨테이너가 실행 중인 것을 알 수 있습니다.
- 2 `docker container stop` [컨테이너 ID] 명령어를 이용해 실행 중인 컨테이너를 정지시킵니다.
- 3 실행 중인 컨테이너 목록을 확인하면 없는 것을 볼 수 있습니다.

그런 이렇게 정지시킨 PostgreSQL 컨테이너를 재실행하고 이전 실습에서 생성했던 데이터가 보존되는지 확인하겠습니다.

- 4 컨테이너 전체 목록을 확인해 PostgreSQL 컨테이너의 컨테이너 ID를 확인합니다.
- 5 `docker container start` [컨테이너 ID] 명령어를 입력하면 정지 상태의 컨테이너를 실행할 수 있습니다.

4.4.2 도커 스토리지의 필요성

```
eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED     STATUS
PORTS          NAMES
a9835064d02f   postgres "docker-entrypoint.s..." 19 minutes ago Up
3 minutes     5432/tcp   some-postgres

eevee@myserver01:~$ docker container exec -it a9835064d02f /bin/bash

root@a9835064d02f:/# psql -U postgres
psql (16.0 (Debian 16.0-1.pgdg120+1))
Type "help" for help.

postgres=# \c test01 user01
You are now connected to database "test01" as user "user01".

test01=# \dt
          List of relations
Schema | Name | Type | Owner
-----+-----+-----+-----
public | table01 | table | user01
(1 row)

test01=# SELECT * FROM table01;
 id | name
----+-----
  1 | Cheolwon
(1 row)

test01=# \q
root@a9835064d02f:/# exit
exit
eevee@myserver01:~$
```

① user01로 test01 데이터베이스에 접속합니다. 이 명령어가 정상적으로 실행되는 것을 보면 앞서 생성한 user01이라는 사용자 데이터가 그대로 보존되고 있는 것을 알 수 있습니다.

② 테이블 목록을 확인하면 앞서 생성한 table01을 확인할 수 있습니다.

③ table01의 데이터를 조회하면 앞서 생성한 데이터가 그대로 유지되는 것을 볼 수 있습니다.

위 실험과 같이 컨테이너를 정지시킨 후 다시 실행해도 앞서 생성했던 테이블이 여전히 살아있는 것을 볼 수 있습니다.

④ 실험이 끝났으므로 PostgreSQL을 종료합니다.

⑤ 컨테이너에서도 빠져나옵니다.

위 실험은 다시 실행한 PostgreSQL 컨테이너 내부에 접속해 이전 실험에서 생성한 데이터가 유지되는지 확인하는 것입니다.

① 실행 중인 컨테이너 목록을 확인하면 이전 실험에서 다시 실행한 PostgreSQL이 실행 중임을 알 수 있습니다.

② docker container exec -it [컨테이너 ID] 명령어를 이용해 실행 중인 컨테이너 내부에 접속합니다.

③ 앞선 실험과 마찬가지로 psql 명령어를 이용해 postgres 계정으로 PostgreSQL에 접속합니다.

4.4.2 도커 스토리지의 필요성

```
eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED    STATUS    PORTS
a9035064d02f   postgres "docker-entrypoint.s..." 23 minutes ago Up 7 minutes 5432

eevee@myserver01:~$ docker container stop a9035064d02f
a9035064d02f

eevee@myserver01:~$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND                  CREATED    STATUS    PORTS    NAMES
a9035064d02f   postgres "docker-entrypoint.s..." 24 minutes ago Exited (0) 7 seconds ago some-postgres

eevee@myserver01:~$ docker container rm a9035064d02f
a9035064d02f

eevee@myserver01:~$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND                  CREATED    STATUS    PORTS    NAMES
```

위 실습은 컨테이너를 삭제하는 것입니다.

- 1 실행 중인 컨테이너를 확인합니다.
- 2 실행 중인 PostgreSQL 컨테이너를 정지시킵니다.
- 3 컨테이너 전체 목록을 확인합니다.

1 앞서 정지시킨 PostgreSQL 컨테이너를 삭제합니다. 컨테이너 삭제를 할 때는 **docker container rm [컨테이너 ID]** 명령어를 사용합니다.

2 컨테이너 전체 목록을 확인하면 해당 컨테이너가 삭제된 것을 확인할 수 있습니다.

새로운 컨테이너를 생성한 후 데이터가 보존되는지도 살펴보겠습니다.

```
eevee@myserver01:~$ docker container run --name some-postgres -e POSTGRES_
PASSWORD=mysecretpassword -d postgres
7cce263324f44cced419b57d6783a0f1c230e34ec6c232cc9807890254fc094e

eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED    STATUS
PORTS    NAMES
7cce263324f4   postgres "docker-entrypoint.s..." 5 seconds ago Up
5 seconds 5432/tcp some-postgres

eevee@myserver01:~$ docker container exec -it 7cce263324f4 /bin/bash
root@7cce263324f4:/# psql -U postgres
psql (16.0 (Debian 16.0-1.pgdg120+1))
Type "help" for help.

postgres=# \c test01 user01
connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: FATAL:
role "user01" does not exist
Previous connection kept

postgres=# \q
root@7cce263324f4:/# exit
exit
eevee@myserver01:~$
```

위 실습은 새로운 컨테이너를 생성하고 앞선 실습에서 생성한 데이터가 보존되었는지 확인하는 것입니다.

- 1 postgres 이미지를 이용해 새로운 컨테이너를 실행합니다.
- 2 컨테이너 목록을 확인하면 PostgreSQL 컨테이너가 실행 중인 것을 알 수 있습니다.
- 3 **docker container exec -it [컨테이너 ID]** 명령어로 실행 중인 컨테이너에 접속합니다.
- 4 컨테이너 내부에서 **psql** 명령어를 이용해 postgres 사용자로 PostgreSQL에 접속합니다.
- 5 user01로 test01 데이터베이스에 접속하는 명령어를 실행하면 여러 메시지가 뜨는데 이를 살펴보면 user01이라

는 사용자가 존재하지 않는 것을 알 수 있습니다. 이와 같이 새롭게 생성한 컨테이너에는 앞서 만들었던 user01이라는 사용자가 존재하지 않는 것을 볼 수 있습니다.

- 3 실습이 끝났으므로 PostgreSQL을 종료합니다.
- 7 컨테이너에서 빠져나옵니다.

```
eevee@myserver01:~$ docker container stop 7cce263324f4
7cce263324f4
eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED    STATUS    PORTS    NAMES
```

실습이 끝났으니 다음 과정으로 마무리합니다.

- 1 해당 컨테이너를 정지합니다.
- 2 컨테이너가 성공적으로 정지되었는지 확인합니다.

4.4.3 volume

volume 방식을 활용한 PostgreSQL 데이터베이스 관리

volume은 컨테이너가 삭제되어도 컨테이너에서 생성되는 데이터 유지



```
eevee@myserver01:~$ docker volume ls
DRIVER      VOLUME NAME
local       77c983c5230ba4709c1e777cb44cf39ad8ec9e35fe7a7a8848712b7edd5ed7df
local       97e2725973a722816e5f10e47ff57c776a5672ea4a20b9d34f51b2ec18b13c0a
```

```
eevee@myserver01:~$ docker volume create myvolume01
myvolume01
```

```
eevee@myserver01:~$ docker volume ls
DRIVER      VOLUME NAME
local       77c983c5230ba4709c1e777cb44cf39ad8ec9e35fe7a7a8848712b7edd5ed7df
local       97e2725973a722816e5f10e47ff57c776a5672ea4a20b9d34f51b2ec18b13c0a
local       myvolume01
```

❶ docker volume ls 명령어를 입력하면 도커 볼륨 리스트를 확인할 수 있습니다.

❷ docker volume create [도커 볼륨명] 명령어를 입력하면 자신이 원하는 도커 볼륨을 생성할 수 있습니다. 이 실습에서는 myvolume01이라는 이름으로 생성했습니다.

❸ 도커 볼륨을 다시 확인합니다.

❹ myvolume01이라는 도커 볼륨이 생성된 것을 확인할 수 있습니다.

4.4.3 volume

```
eevee@myserver01:~$ docker container run -e POSTGRES_PASSWORD=mysecretpassword --mount
type=volume,source=myvolume01,target=/var/lib/postgresql/data -d postgres ❶

7adb986dc5cbe7c3e16f5a8d6ee90f24a39e0af671a16052727a5dc2d900085

eevee@myserver01:~$ docker container ls ❷
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS
PORTS         NAMES
7adb986dc5c   postgres "docker-entrypoint.s..." 11 seconds ago Up
10 seconds    5432/tcp  wizardly_heisenberg
```

- ❶ 도커 볼륨 myvolume01과 연동시켜 PostgreSQL 컨테이너를 실행하겠습니다. `--mount` 옵션을 활용해 **source=[도커 볼륨명]**, **target=[컨테이너 내부 경로]** 형태로 사용합니다. 명령어 중 쉼표(.)를 사용할 때 띄어쓰기를 하지 않는다는 점에 주의해야 합니다. 위 과정은 myvolume01 볼륨과 컨테이너 내부의 `/var/lib/postgresql/data` 경로를 연결시키는 것을 의미합니다.
- ❷ 실행 중인 컨테이너를 확인하면 앞서 생성한 PostgreSQL 컨테이너가 실행 중임을 확인할 수 있습니다.

4.4.3 volume

```
eevee@myserver01:~$ docker container exec -it 7adbb986dc5c /bin/bash ❶

root@7adbb986dc5c:/# psql -U postgres ❷
psql (16.0 (Debian 16.0-1.pgdg120+1))
Type "help" for help.

postgres=# CREATE USER user01 PASSWORD '1234' SUPERUSER; ❸
CREATE ROLE

postgres=# \du ❹

          List of roles
Role name | Attributes
-----+-----
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS
user01    | Superuser

postgres=# \q ❺

root@7adbb986dc5c:/# cd /var/lib/postgresql/data/ ❻

root@7adbb986dc5c:/var/lib/postgresql/data# ls ❼
base          pg_ident.conf  pg_serial      pg_tblspc      postgresql.auto.conf
global        pg_logical     pg_snapshots   pg_twophase     postgresql.conf
pg_commit_ts  pg_multixact   pg_stat        PG_VERSION     postmaster.opts
pg_dynshmem   pg_notify      pg_stat_tmp     pg_wal          postmaster.pid
pg_hba.conf   pg_replslot    pg_subtrans    pg_xact

root@7adbb986dc5c:/var/lib/postgresql/data# exit ❸
exit
```

❶ 컨테이너 내부에 접속합니다.

❷ **psql** 명령어를 입력해서 postgres 사용자로 PostgreSQL에 접속합니다.

❸ user01이라는 사용자를 SUPERUSER 권한으로 생성합니다.

❹ 사용자 목록을 확인하면 방금 생성한 user01이라는 사용자를 확인할 수 있습니다. 바로 이 user01이라는 사용자 데이터가 도커 볼륨에 의해 유지될 예정입니다.

❺ PostgreSQL을 종료합니다.

❻ /var/lib/postgresql/data/로 이동합니다.

❼ 파일 목록을 확인합니다. **ls** 출력 결과로 나오는 파일들이 도커 볼륨에 저장될 예정입니다.

❸ 실험이 끝났으면 컨테이너에서 빠져나옵니다.

4.4.3 volume

컨테이너 삭제 후, 데이터 유지 확인

```
eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED     STATUS
PORTS         NAMES
7adbb986dc5c   postgres "docker-entrypoint.s..." 10 minutes ago Up
10 minutes    5432/tcp   wizardly_heisenberg

eevee@myserver01:~$ docker container stop 7adbb986dc5c
7adbb986dc5c

eevee@myserver01:~$ docker container rm 7adbb986dc5c
7adbb986dc5c
```

이번에는 앞선 실습과 같이 컨테이너를 완전히 삭제한 후 새로운 컨테이너를 생성했을 때 데이터가 유지되는지 확인하겠습니다.

- 1 먼저 실행 중인 컨테이너를 확인합니다.
- 2 컨테이너를 정지시킵니다.
- 3 컨테이너를 삭제합니다.

```
eevee@myserver01:~$ docker container run -e POSTGRES_PASSWORD=mysecretpassword -v myvolume01:/var/lib/postgresql/data -d postgres
41cd059e3d97e231aa0186e910e250a70ac12a1c4dc43b32db60d56546a39233

eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED     STATUS
PORTS         NAMES
41cd059e3d97   postgres "docker-entrypoint.s..." 7 seconds ago Up
6 seconds    5432/tcp   relaxed_albattani

eevee@myserver01:~$ docker exec -it 41cd059e3d97 /bin/bash
root@41cd059e3d97:/# psql -U postgres
psql (16.0 (Debian 16.0-1.pgdg120+1))
Type "help" for help.

postgres=# \du
                                List of roles
Role name |
-----+-----
```

```
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS
user01   | Superuser

postgres=# \q
root@41cd059e3d97:/# exit
exit
```

앞선 실습과 같이 컨테이너를 삭제한 이후 새로운 컨테이너를 생성해서 기존 데이터가 유지되는지 확인하겠습니다.

- 1 postgres 이미지를 이용해 PostgreSQL 컨테이너를 실행합니다. 앞선 실습과 마찬가지로 myvolume01 볼륨과 컨테이너 내부 경로 /var/lib/postgresql/data를 연결해서 실행합니다. 도커 볼륨을 컨테이너 내부 경로에 사용할 때는 앞선 실습과 같이 --mount 옵션을 사용할 수도 있고 이번 실습처럼 -v 옵션을 사용할 수도 있습니다. -v에서 v는 volume의 줄임말로 --volume 형태로 사용할 수도 있습니다.
- 2 실행 중인 컨테이너 목록을 확인하면 PostgreSQL 컨테이너가 실행 중임을 볼 수 있습니다.
- 3 컨테이너 내부에 접속합니다.
- 4 psql 명령어를 이용해 postgres 계정으로 PostgreSQL에 접속합니다.
- 5 사용자 목록을 확인하면 이전 실습에서 생성했던 user01이 존재하는 것을 볼 수 있습니다. 이처럼 도커 볼륨을 활용하면 컨테이너가 삭제되어도 컨테이너 내부 데이터를 관리하기 편리하다는 것을 알 수 있습니다.
- 6 실습이 끝났으니 종료합니다.
- 7 컨테이너에서 빠져나갑니다.

4.4.3 volume

컨테이너 삭제 후, 데이터 유지 확인

```
eevee@myserver01:~$ docker volume inspect myvolume01
[
  {
    "CreatedAt": "2023-11-02T05:15:10Z",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/myvolume01/_data",
    "Name": "myvolume01",
    "Options": null,
    "Scope": "local"
  }
]
```

- 1 도커 호스트에서 `inspect` 명령어를 사용하면 볼륨의 정보를 확인할 수 있습니다.
- 2 Mountpoint가 컨테이너의 데이터를 보관하는 로컬 호스트 경로입니다. 즉, myvolume01이라는 볼륨에서 관리하는 데이터가 존재하는 경로는 `/var/lib/docker/volumes/myvolume01/_data`라는 뜻입니다.

```
eevee@myserver01:~$ sudo -i
[sudo] password for eevee:
root@myserver01:~# cd /var/lib/docker/volumes/myvolume01/_data/
root@myserver01:/var/lib/docker/volumes/myvolume01/_data# ls
base  pg_commit_ts  pg_hba.conf  pg_logical  pg_notify  pg_serial  pg_stat
pg_subtrans  pg_twophase  pg_wal  postgresql.auto.conf  postmaster.opts
global  pg_dynshmem  pg_ident.conf  pg_multixact  pg_replslot  pg_snapshots
pg_stat_tmp  pg_tblspc  PG_VERSION  pg_xact  postgresql.conf  postmaster.pid
root@myserver01:/var/lib/docker/volumes/myvolume01/_data# exit
logout
eevee@myserver01:~$
```

다음의 과정을 통해 데이터가 모두 저장되었는지 확인합니다.

- 1 루트 권한으로 접속합니다.
- 2 해당 경로로 이동합니다.
- 3 컨테이너에서 확인했던 `/var/lib/postgresql/data/` 경로의 데이터가 모두 저장되어 있는 것을 알 수 있습니다.

4.4.3 volume

컨테이너 삭제 후, 데이터 유지 확인

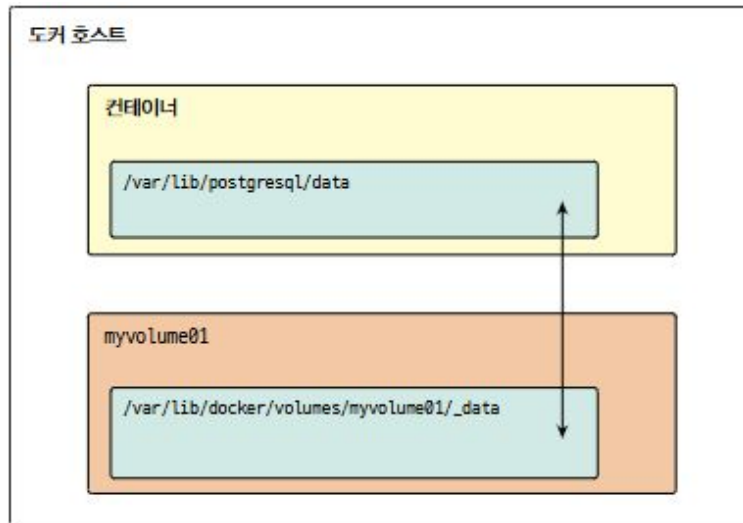


그림 4-23 volume

4.4.4 bind mount

도커 호스트 디렉터리와 컨테이너 디렉터리를 연결시켜 데이터를 보관하는 방식

```
터미널 1
eevee@myserver01:~$ ls
work
eevee@myserver01:~$ cd work/ch04/ex01/
eevee@myserver01:~/work/ch04/ex01$ ls
test01.txt test02.txt
eevee@myserver01:~/work/ch04/ex01$ pwd
/home/eevee/work/ch04/ex01
```

먼저 컨테이너 내부와 연결시킬 도커 호스트 경로를 설정하겠습니다.

- 1 터미널 1에서 도커 호스트 목록을 확인합니다.
- 2 /home/eevee/work/ch04/ex01 경로로 이동합니다.
- 3 해당 디렉터리에 존재하는 파일을 컨테이너에 유시시켜 보겠습니다. 즉, test01.txt 파일과 test02.txt 파일이 도커 호스트뿐만 아니라 컨테이너에도 저장될 예정입니다.
- 4 해당 디렉터리의 절대 경로를 확인합니다.

```
터미널 1
eevee@myserver01:~/work/ch04/ex01$ docker container run -e POSTGRES_
PASSWORD=mysecretpassword --mount type=bind,source=/home/eevee/work/ch04/ex01,target=/
work -d postgres
a09d02ab05e605daf50189523c0b8885b76d614b92f37d20f5889f672b718f45

eevee@myserver01:~/work/ch04/ex01$ docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
a09d02ab05e6 postgres "docker-entrypoint.s..." 6 seconds ago Up
5 seconds 5432/tcp sad_shtern

eevee@myserver01:~/work/ch04/ex01$ docker container exec -it a09d02ab05e6 /bin/
bash
root@a09d02ab05e6:/# ls
bin docker-entrypoint-initdb.d lib libx32 opt run sys var
boot etc lib32 media proc sbin tmp work
dev home lib64 mnt root srv usr

root@a09d02ab05e6:/# cd work/
root@a09d02ab05e6:/work# ls
test01.txt test02.txt
```

PostgreSQL 컨테이너를 실행하겠습니다.

- 1 위 코드와 같이 컨테이너를 실행할 때 도커 볼륨을 사용하기 위해 --mount 옵션을 사용하고 도커 호스트의 /home/eevee/work/ch04/ex01 경로와 도커 컨테이너 내부의 /work 경로를 연결시켜줍니다. 명령어 중 점표 (.)를 작성할 때 띄어쓰기를 하지 않도록 주의해야 합니다.
- 2 실행 중인 컨테이너 목록을 확인하면 원활하게 작동 중인 것을 알 수 있습니다.
- 3 실행 중인 컨테이너 내부에 접속합니다.
- 4 파일 목록을 확인합니다.
- 5 work 디렉터리로 이동합니다.
- 6 파일 목록을 확인하면 도커 호스트에 존재했던 test01.txt 파일과 test02.txt 파일이 존재하는 것을 알 수 있습니다.

4.4.4 bind mount

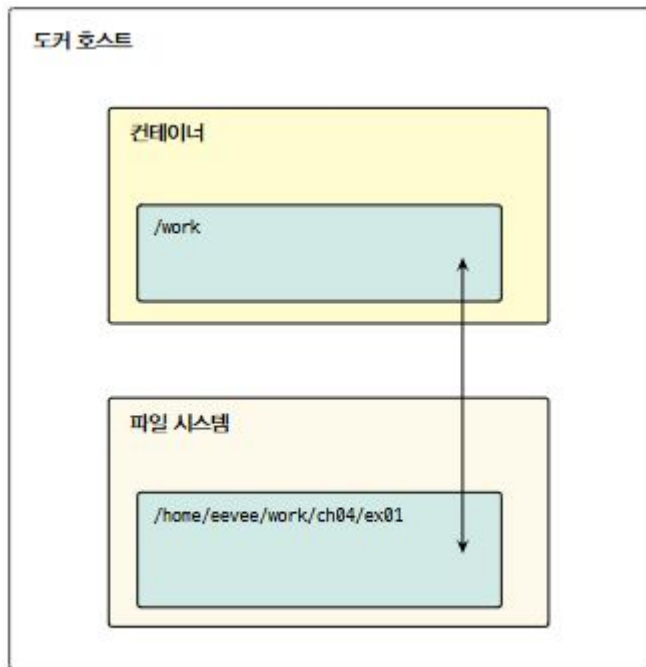


그림 4-24 bind mount

4.4.4 bind mount

컨테이너 내부에서 새로운 디렉터리 생성 후 변화 확인

터미널 1

```
root@a09d02ab05e6:/work# mkdir test_dir
root@a09d02ab05e6:/work# ls
test01.txt test02.txt test_dir
```

❶

❷

- ❶ 컨테이너 내부의 /work 경로에서 **mkdir** 명령어를 이용해 **test_dir**이라는 디렉터리를 생성합니다.
- ❷ 파일 목록을 확인합니다.

터미널 2

```
eevee@myserver01:~$ cd work/ch04/ex01/
eevee@myserver01:~/work/ch04/ex01$ ls
test01.txt test02.txt test_dir
```

❶

❷

- ❶ 도커 호스트에서 /work/ch04/ex01/ 경로로 이동합니다.
- ❷ 해당 경로의 파일 목록을 확인합니다. 그러면 이전 실습에서 컨테이너 내부에서 생성했던 **test_dir**이라는 디렉터리가 도커 호스트에도 생성되어 있는 것을 알 수 있습니다. 즉, 컨테이너 내부에서 파일이 변화면 연결되어 있는 도커 호스트 경로도 함께 변화하는 것을 알 수 있습니다.

터미널 2

```
eevee@myserver01:~/work/ch04/ex01$ rm -rf test_dir
eevee@myserver01:~/work/ch04/ex01$ ls
test01.txt test02.txt
```

❶

❷

그렇다면 도커 호스트 경로에서 변화가 발생하면 컨테이너에는 어떤 영향을 미칠까요?

- ❶ 터미널 2를 통해 도커 호스트에서 앞서 생성했던 **test_dir** 디렉터를 삭제합니다.
- ❷ 제대로 삭제되었는지 확인합니다.

터미널 1

```
root@a09d02ab05e6:/work# ls
test01.txt test02.txt
root@a09d02ab05e6:/work# exit
exit
```

❶

❷

그리고 다시 터미널 1로 이동합니다.

- ❶ 터미널 1에서 컨테이너 내부의 파일 목록을 다시 한번 확인하면 이전과는 달리 **test_dir**가 삭제되어 있는 것을 알 수 있습니다. 즉, 도커 호스트 경로에서 변경 사항이 생기면 컨테이너에도 동일한 영향을 미친다는 것을 알 수 있습니다.
- ❷ 실습이 끝났으니 컨테이너에서 빠져나갑니다.

4.4.5 tmpfs mount

중요한 데이터를 일시적으로 도커 호스트 메모리에 저

컨테이너 간 데이터 공유를 지원하지 않음

컨테이너 정지시 tmpfs mount도 삭제

```
eevee@myserver01:~$ docker container run \
-e POSTGRES_PASSWORD=mysecretpassword \
--mount type=tmpfs,destination=/var/lib/postgresql/data \
-d postgres
```

```
eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS
PORTS         NAMES
88d4c391e316   postgres "docker-entrypoint.s..." 8 seconds ago Up
8 seconds     5432/tcp   reverent_meitner
```

```
eevee@myserver01:~$ docker inspect reverent_meitner --format '{{
```

..(중략)

"Mounts": [

```
{
  "Type": "tmpfs",
  "Source": "",
  "Destination": "/var/lib/postgresql/data",
  "Mode": "",
  "RW": true,
  "Propagation": ""
}
```

],

..(중략)

❶ --mount 옵션을 활용해 type=tmpfs라고 설정해주고 저장하고자 하는 경로를 destination으로 입력합니다.

❷ 실행 중인 컨테이너 목록을 확인하면 원활하게 작동하고 있는 것을 볼 수 있습니다.

❸ inspect 명령어를 이용해 확인해보면 tmpfs 타입으로 마운트된 것을 볼 수 있습니다.

```
eevee@myserver01:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS
PORTS         NAMES
88d4c391e316   postgres "docker-entrypoint.s..." 3 minutes ago Up
3 minutes     5432/tcp   reverent_meitner
eevee@myserver01:~$ docker container stop 88d4c391e316
88d4c391e316
eevee@myserver01:~$ docker container rm 88d4c391e316
88d4c391e316
```