

## Ch 6. 도커 네트워크/볼륨

### 6.1 도커 네트워크

#### 출처

- 미즈구치 카츠야 저/이승룡 역, 『모두의 네트워크』, 길벗(2020)
- 고재성, 이상훈, 『IT 엔지니어를 위한 네트워크 입문』, 길벗(2020)
- Charles Severance, 『Introduction to Networking: How the Internet Works』, CreateSpace Independent Publishing Platform; 1 edition (May 29, 2015)

#### 6.1.0 네트워크



네트워크란 무엇일까요? 우리가 이용하고 있는 인터넷 환경도 네트워크가 구축되어 있기 때문에 사용이 가능하죠. 네트워크는 복수의 디바이스가 연결되어 있는 것입니다. 연결된 디바이스 간에 데이터를 주고받는 규칙을 **프로토콜(Protocol)** 이라고 하는데, 우리가 이미 익숙하게 알고 있는 [http\(80\)](http://80), [https\(443\)](https://443) 등이 바로 그것입니다. 본격적으로 도커 네트워크를 다루기에 앞서 네트워크와 관련된 배경지식을 짚어보도록 하겠습니다.

```
ifconfig # Linux/macOS ipconfig # Windows
```

위의 명령어를 OS에 맞게 입력해보도록 하겠습니다. 하단의 이미지는 [Ubuntu:18.04](#)에서 실행한 결과입니다.

뭔가 복잡한 문구들이 창을 가득 채우고 있습니다. 중요한 부분 위주로 살펴보겠습니다.

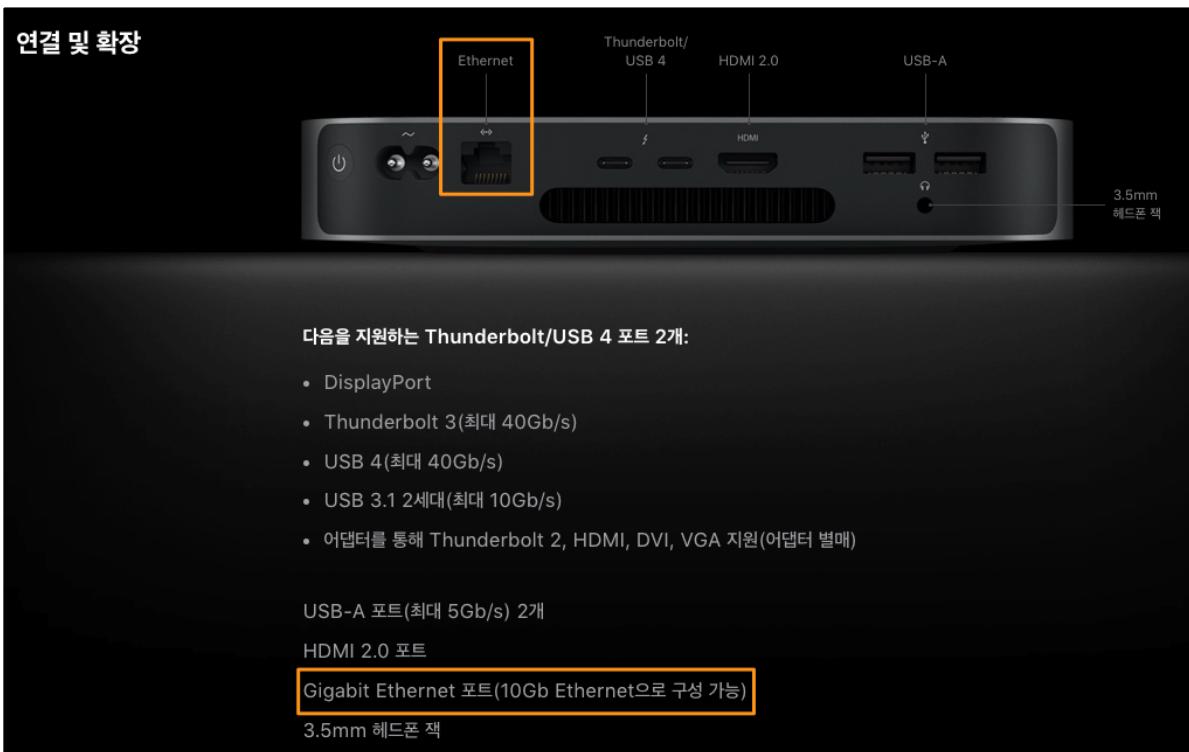
```
ggingmin@ubuntu_server:~$ ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
              inet6 fe80::42:19ff:fe10:474 prefixlen 64 scopeid 0x20<link>
                ether 02:42:19:10:04:74 txqueuelen 0 (Ethernet)
                  RX packets 140 bytes 8053 (8.0 KB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 155 bytes 11361 (11.3 KB)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 172.16.173.6 netmask 255.255.255.0 broadcast 172.16.173.255
              inet6 fe80::20c:29ff:fed0:2310 prefixlen 64 scopeid 0x20<link>
                ether 00:0c:29:d0:23:10 txqueuelen 1000 (Ethernet)
                  RX packets 21062 bytes 28075557 (28.0 MB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 2240 bytes 245742 (245.7 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
              inet6 ::1 prefixlen 128 scopeid 0x10<host>
                loop txqueuelen 1000 (Local Loopback)
                  RX packets 144 bytes 12688 (12.6 KB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 144 bytes 12688 (12.6 KB)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

### 1) ens33

- 디바이스의 NIC(Network Interface Controller)를 나타내고 있는 부분입니다. NIC는 일반적으로 LAN 카드라는 표현이 더 익숙한데, 호스트와 네트워크 통신망 사이에서 데이터 송수신 역할을 수행합니다.
- 일반적인 PC에는 물리적인 NIC가 내장되어 있습니다. 사양 페이지에 등장하는 Ethernet 관련 포트가 바로 유선으로 외부 통신망과 호스트 디바이스를 연결해주는 매개체입니다.



- 구버전 리눅스에서는 `eth0`, `eth1` 이었으나 지금의 형태로 변경되었습니다.

## 2) `lo`

- `lo` 는 **Loopback network interface**에서 비롯된 용어로 `localhost` 를 의미합니다. 디바이스의 관점에서 자기 자신을 가리키는 것이죠. 웹서버를 구동한 PC에서 실제 브라우저를 통해 접속하여 테스트해볼 수 있는 것이 바로 Loopback 네트워크 덕분입니다. 하나의 디바이스가 클라이언트와 서버의 역할을 겸하는 것이죠.
- OS 내부적으로 호스트의 이름을 IP와 매핑하여 가지고 있는 파일이 있는데, Ubuntu의 경우에는 `/etc/hosts` 경로에 `hosts`라는 파일로 세팅되어 있습니다. `127.0.0.1`이나 `localhost`로 접속해도 모두 동일한 결과를 볼 수 있는 것은 바로 이 설정 때문입니다.

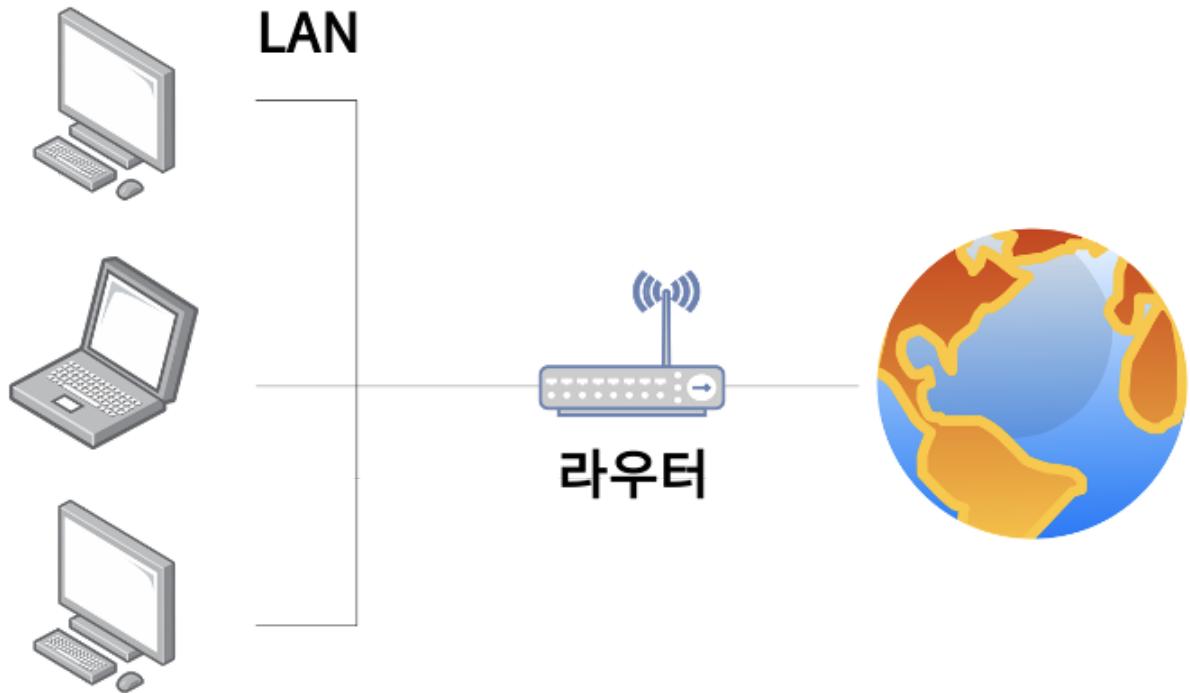
```
ggingmin@ubuntu_server:~$ sudo cat /etc/hosts
[sudo] password for ggingmin:
127.0.0.1 localhost
127.0.1.1 ubuntu_server

# The following lines are desirable for IPv6 capable hosts
::1      ip6-localhost ip6-loopback
fe00::0  ip6-localnet
ff00::0  ip6-mcastprefix
ff02::1  ip6-allnodes
ff02::2  ip6-allrouters
```

## 3) `inet` `inet6`

- `inet` 은 IPv4를 의미하며, 우리에게 익숙한 12 자리 숫자체계입니다. 32비트로 구성되어 있으며, 10진수를 4부분으로 분할하여 표기합니다.
- `inet6` 은 IPv6를 의미하며, 32비트의 IPv4가 고갈될 것에 대비한 128비트 체계의 IP주소입니다. 16진수를 8부분으로 분할하여 표기합니다. `0` 값을 가지는 경우 생략이 가능하기 때문에 실제로 조회되는 주소값은 8부분보다 적은 경우가 있습니다.

- 실제로 모든 디바이스가 공인 IP(Public IP)를 사용하는 것은 아닙니다. 대신 라우터에 의해 설정된 대역에서 사설 IP(Private IP)를 사용하죠. 이에 따라 하나의 공인 IP를 가지고 여러 디바이스 들이 통신망에 접근 할 수 있게 되었습니다. 라우터로 묶인 네트워크 통신망을 LAN(Local Area Network) 이라고 하며 여러 LAN 들이 모여 커다란 WAN(Wide Area Network)을 구성합니다.



#### 4) port

이미지에서는 다루지 않지만 앞으로 자주 등장하는 Port를 잠시 살펴보겠습니다. IP와 Port는 들을 때마다 헷갈리는 부분이 많습니다. 하지만 쉽게 생각하면 제일 쉬운게 바로 IP와 Port입니다.

IP는 디바이스가 가지는 주소이죠. 우리가 사는 곳도 주소가 있습니다. 가령 현대오피스텔 202동 을 IP 주소라고 하면 각 1002호, 304호 등과 같은 호수를 Port라고 할 수 있습니다. 물리적으로는 같은 위치에 있지만 각 호수별로 기능이 나뉘는 것이죠. 우리가 사용하는 컴퓨터도 이렇게 서비스를 Port 단위로 나누고 특정 Port로만 서비스가 통신할 수 있도록 할당합니다.



### Well-known Port

미국의 IANA(Internet Assigned Numbers Authority)는 인터넷 할당 번호를 관리하는 기관으로 특정 서비스에 대한 Port 를 Well-known Port 로 지정하였습니다.

FTP : 20, 21

SSH : 22

TELNET: 23

SMTP : 25

DNS : 53

HTTP : 80

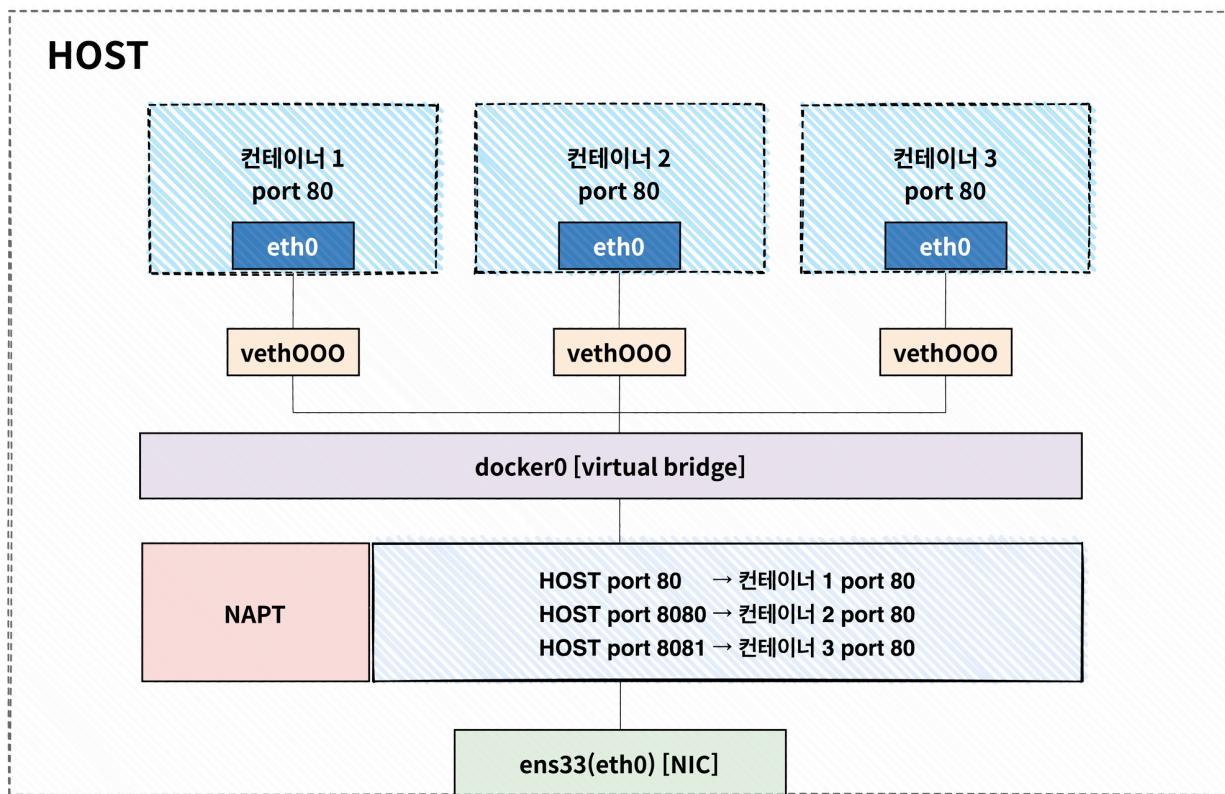
POP3 : 110

HTTPS : 443

출처 : Internet Assigned Numbers Authority (IANA) <https://www.iana.org/>

## 6.1.1 호스트-컨테이너 네트워크

먼저 호스트와 도커 엔진 간 네트워크 구성도를 보도록 하겠습니다.



뭔가 복잡하고 생소한 용어가 등장해서 당황하셨을 수도 있습니다. 용어를 확인하고 실제 서버에 구축되어 있는 네트워크 설정을 조회해보면서 이해하면 어렵지 않은 내용이니 함께 확인해봅시다.

일단 용어 및 각 단계의 역할을 정리해보겠습니다.

## 1. NIC(Network Interface Controller)

- 흔히 우리가 LAN 카드라고 부르는 것이 바로 이 NIC입니다. 호스트와 네트워크간 데이터를 송수신하는 인터페이스 역할을 수행합니다.
- 물리적으로 구성된 NIC는 ens33으로 인식 됩니다. 구버전 리눅스 배포판에서는 eth0로 인식되었고 시중의 많은 도서도 eth0로 기술되어 있는데 모두 물리적인 NIC로 보시면 됩니다.

## 2. NAPT(Network Address Port Translation)

- NAPT는 IP와 Port를 변환하는 기술입니다. 왜 이 두 가지 요소의 변환이 필요한 걸까요? 도커 엔진이 컨테이너에 부여한 IP는 사설 IP로서, 호스트에서 직접 접근이 불가능 합니다. 접근을 가능케 하기 위해서 Host의 각 포트에 컨테이너의 IP와 포트를 맵핑 시켜주는 것이 NAPT의 역할입니다.

## 3. docker0

- bridge라는 형태의 네트워크이며 도커 엔진을 실행하면 자동으로 생성됩니다. 컨테이너를 실행하면 내부의 모든 포트를 docker0라는 가상 bridge에 개방합니다. 호스트는 NIC에서 직접 컨테이너와 연결하는 것이 아니라 이 bridge를 경유하게 됩니다. 포트를 통해 각 컨테이너에 접근할 수 있도록 다리를 놔주는 역할을 수행합니다.

## 4. vethOOO

- 컨테이너를 띄우고 Host에서 ifconfig 명령어를 수행하면 docker0, lo, ens33 이외에 veth로 시작하는 것이 새롭게 생긴 것을 확인 할 수 있습니다. 이는 컨테이너가 Host와 통신하기 위한 가상의 NIC입니다.
- veth + 난수의 형태로 생성됩니다.
- 구조도 내 컨테이너에 eth0이라고 표시된 이유는 Host 입장이 아닌 컨테이너 입장에서 가상 NIC를 eth0라고 인식하기 때문입니다.

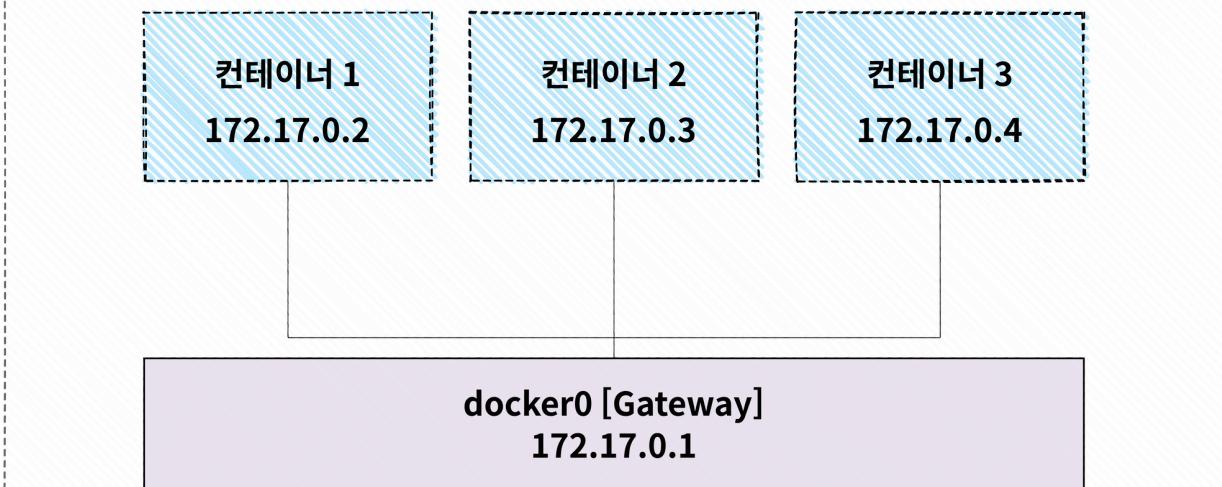
### 6.1.2 도커 네트워크

호스트와 컨테이너 간 네트워크의 작동 방식에 대해 살펴보았습니다. 지금부터 알아볼 컨테이너 간 네트워크도 매우 중요한데요, 이는 우리가 구동할 서비스가 단일 컨테이너로만 이루어지는 경우는 없기 때문입니다.

만약 API를 통해 데이터를 DB에 등록하려고 하는 서비스를 구축하는 경우, API 서버와 DBMS 서버를 각각 컨테이너에 구축해야 합니다. 컨테이너 생성 뿐만 아니라 이 두 컨테이너를 통신이 가능하도록 설정해주어야 하죠. 컨테이너 간 네트워크의 구성 방식을 살펴보도록 하겠습니다.

#### 1) bridge

## DOCKER DAEMON HOST



도커 엔진 내부에서 동작하는 `bridge` 네트워크는 위와 같이 구성되어 있습니다. `docker0` 를 Gateway로 하여 컨테이너가 생성된 순서대로 IP가 부여됩니다. 기본적으로 컨테이너를 구동할 때 별도의 네트워크 설정을 하지 않으면 모두 이 `docker0` 라는 `bridge` 네트워크에 연결이 됩니다. 같은 대역의 네트워크에 컨테이너가 구성되었기 때문에 통신도 서로 가능하죠. 만약 커스터마이징한 별도의 `bridge` 네트워크를 구축해서 컨테이너를 구성한다면 기존의 `docker0` 에 구성된 컨테이너와는 통신이 불가능 합니다.

물리적인 스위치를 생각하면 좀더 이해가 쉽습니다. A 스위치와 B 스위치에 각각 구성된 디바이스는 당연히 통신을 할 수 없습니다. 도커에서는 `bridge` 네트워크가 물리적인 스위치 역할을 한다고 보시면 됩니다.



### 게이트웨이(Gateway)

네트워크에 접근하기 위한 출입구를 가리킵니다. 우리가 스마트폰과 노트북으로 와이파이에 접속하는 것도 모두 게이트웨이를 통해 인터넷 망에 접근하는 것입니다. 이와 같이 서로 다른 네트워크 간 접근을 위해서는 게이트웨이 주소가 필요하며, 기본적으로 IP 마지막 자리를 `1` 로 설정합니다.

```
sudo docker container run -d -p 80:80 --name webserver1 httpd
sudo docker container run -d -p 8080:80 --name webserver2 httpd
sudo docker container run -d -p 8081:80 --name webserver3 httpd
```

3개의 컨테이너에 각각 웹서버를 구동하고 IP가 어떻게 부여됐는지 확인해보겠습니다.

```
sudo docker container inspect --format="{{ .NetworkSettings.IPAddress }}"
webserver1 webserver2 webserver3
```

```
gingmin@ubuntu_server:~$ sudo docker container inspect --format=
172.17.0.2
172.17.0.3
172.17.0.4
```

2) `host`

## DOCKER DAEMON HOST

컨테이너 1

HOST IP

**host** 네트워크는 단어 그대로 별도의 경유 없이 도커 엔진이 작동하고 있는 Host의 IP 주소를 그대로 사용하는 것을 뜻합니다. **bridge** 를 통해 NAPT의 과정을 거치지 않아도 되며, 명시적으로 컨테이너의 포트를 명령어에 작성하지 않아도 됩니다. 다시 말해 컨테이너를 격리된 네트워크로 구성하지 않고 도커 엔진과 한 공간에 두는 것이라고 할 수 있습니다.

**host** 형식의 네트워크는 Host OS가 리눅스인 경우에만 지원하며, Docker Desktop의 형태로 사용하는 경우 지원하지 않습니다.

```
~$ sudo docker container run -it --network host --name ubuntu-host  
ubuntu:18.04 /# apt-get update /# apt-get install net-tools
```

```
ggingmin@ubuntu_server:~$ sudo docker container run -it --network host --name ubuntu-host ubuntu:18.04  
root@ubuntu_server:/# apt-get update  
Get:1 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]  
Get:2 http://archive.ubuntu.com/ubuntu bionic InRelease [242 kB]  
Get:3 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages [2295 kB]  
Get:4 http://archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]  
Get:5 http://archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]  
Get:6 http://archive.ubuntu.com/ubuntu bionic/main amd64 Packages [1344 kB]  
Get:7 http://security.ubuntu.com/ubuntu bionic-security/multiverse amd64 Packages [26.7 kB]  
Get:8 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages [1426 kB]  
Get:9 http://security.ubuntu.com/ubuntu bionic-security/restricted amd64 Packages [543 kB]  
Get:10 http://archive.ubuntu.com/ubuntu bionic/multiverse amd64 Packages [186 kB]  
Get:11 http://archive.ubuntu.com/ubuntu bionic/universe amd64 Packages [11.3 MB]  
Get:12 http://archive.ubuntu.com/ubuntu bionic/restricted amd64 Packages [13.5 kB]  
Get:13 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 Packages [2731 kB]  
Get:14 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 Packages [2200 kB]  
Get:15 http://archive.ubuntu.com/ubuntu bionic-updates/restricted amd64 Packages [575 kB]  
Get:16 http://archive.ubuntu.com/ubuntu bionic-updates/multiverse amd64 Packages [34.4 kB]  
Get:17 http://archive.ubuntu.com/ubuntu bionic-backports/universe amd64 Packages [11.4 kB]  
Get:18 http://archive.ubuntu.com/ubuntu bionic-backports/main amd64 Packages [11.3 kB]  
Fetched 23.2 MB in 8s (2919 kB/s)  
Reading package lists... Done
```

```

root@ubuntu_server:/# apt-get install net-tools
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  net-tools
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 194 kB of archives.
After this operation, 803 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu bionic/main amd64 net-tools amd64 1.60+git20161116.90da8a0-1ubuntu1 [194 kB]
Fetched 194 kB in 2s (93.8 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package net-tools.
(Reading database ... 4051 files and directories currently installed.)
Preparing to unpack .../net-tools_1.60+git20161116.90da8a0-1ubuntu1_amd64.deb ...
Unpacking net-tools (1.60+git20161116.90da8a0-1ubuntu1) ...
Setting up net-tools (1.60+git20161116.90da8a0-1ubuntu1) ...
root@ubuntu_server:/#

```

기본적으로 ubuntu 이미지를 컨테이너로 올리면 `ifconfig` 명령이 포함된 `net-tools` 패키지를 설치해주어야 합니다. 위와 같이 설치를 먼저 진행하겠습니다.

```

root@ubuntu_server:/# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
      inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:53:11:ab:74  txqueuelen 0  (Ethernet)
          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

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 172.16.173.6  netmask 255.255.255.0  broadcast 172.16.173.255
        ether 00:0c:29:d0:23:10  txqueuelen 1000  (Ethernet)
          RX packets 18218  bytes 25798149 (25.7 MB)
          RX errors 0  dropped 0  overruns 0  frame 0
          TX packets 2181  bytes 202417 (202.4 KB)
          TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

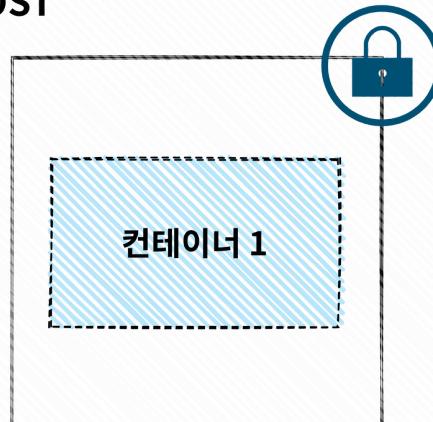
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
      inet 127.0.0.1  netmask 255.0.0.0
        ether 00:00:00:00:00:00  txqueuelen 1000  (Local Loopback)
          RX packets 142  bytes 12334 (12.3 KB)
          RX errors 0  dropped 0  overruns 0  frame 0
          TX packets 142  bytes 12334 (12.3 KB)
          TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

```

명령을 실행해보면 가상머신의 IP와 컨테이너와 IP가 동일한 것을 확인할 수 있습니다.

### 3) `none`

#### DOCKER DAEMON HOST



`none` 은 말 그대로 컨테이너를 격리된 공간에 차단시켜놓고 어떠한 네트워크와도 연결하지 못하게 하는 것을 가리킵니다.

`host` 와 같은 방법으로 `none` 네트워크가 세팅된 컨테이너를 만들고 패키지 설치를 시도해보겠습니다.

```
~$ sudo docker container run -it --network none --name ubuntu-none  
ubuntu:18.04 /# apt-get update
```

```
ggingmin@ubuntu_server:~$ sudo docker container run -it --network none --name ubuntu-none ubuntu:18.04  
root@1aa25676e754:/# apt-get update  
Err:1 http://security.ubuntu.com/ubuntu bionic-security InRelease  
Temporary failure resolving 'security.ubuntu.com'  
Err:2 http://archive.ubuntu.com/ubuntu bionic InRelease  
Temporary failure resolving 'archive.ubuntu.com'  
Err:3 http://archive.ubuntu.com/ubuntu bionic-updates InRelease  
Temporary failure resolving 'archive.ubuntu.com'  
Err:4 http://archive.ubuntu.com/ubuntu bionic-backports InRelease  
Temporary failure resolving 'archive.ubuntu.com'  
Reading package lists... Done  
W: Failed to fetch http://archive.ubuntu.com/ubuntu/dists/bionic/InRelease Temporary failure resolving 'archive.ubuntu.com'  
W: Failed to fetch http://archive.ubuntu.com/ubuntu/dists/bionic-updates/InRelease Temporary failure resolving 'archive.ubuntu.com'  
W: Failed to fetch http://archive.ubuntu.com/ubuntu/dists/bionic-backports/InRelease Temporary failure resolving 'archive.ubuntu.com'  
W: Failed to fetch http://security.ubuntu.com/ubuntu/dists/bionic-security/InRelease Temporary failure resolving 'security.ubuntu.co  
m'  
W: Some index files failed to download. They have been ignored, or old ones used instead.
```

외부와 통신이 불가능하기 때문에 `apt` 패키지 저장소로부터 업데이트가 정상적으로 진행되지 않습니다.

### 6.1.3 도커 네트워크 명령어

도커 엔진을 처음 실행하면 우리가 만들지 않은 네트워크가 설정되어 있음을 알 수 있습니다. 기본적으로 생성되는 네트워크를 확인해보고 이어서 여러 네트워크 관련 명령어에 대해 알아보겠습니다.

#### 1) `ls` - 네트워크 목록 조회

▼ 옵션

`ls`

Aa 명령어	☰ 이름	☰ 가능
	--filter	조회 필터를 설정합니다.

개수 1

```
sudo docker network ls sudo docker network ls --filter driver=bridge
```

```
ggingmin@ubuntu_server:~$ sudo docker network ls  
NETWORK ID      NAME      DRIVER      SCOPE  
06d7d0548b9f    bridge    bridge      local  
700d86a9f38c    host      host       local  
3b8cce6433ef    none      null       local
```

```
ggingmin@ubuntu_server:~$ sudo docker network ls --filter driver=bridge  
NETWORK ID      NAME      DRIVER      SCOPE  
06d7d0548b9f    bridge    bridge      local
```

도커에서는 `--filter` 를 통해 특정 조건에 부합하는 요소를 조회할 수 있습니다. 위와 같이 네트워크의 종류를 설정하여 조회할 수도 있으며, `name`, `scope` 등도 조건으로 설정할 수 있습니다.

## 2) `create` - 네트워크 생성

### ▼ 옵션

#### create

Aa 명령어	☰ 이름	☰ 기능
	<code>--driver</code>	네트워크 종류를 선택합니다. 기본값은 bridge 입니다.

개수 1

```
sudo docker network create --driver=bridge new-bridge
```

```
ggingmin@ubuntu_server:~$ sudo docker network create --driver=bridge new-bridge
29390bb3509c1aa98fdb2278b5f10e10bb95e6571c8fc64c6ab68fee616eeeb8
ggingmin@ubuntu_server:~$ sudo docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
06d7d0548b9f    bridge    bridge      local
700d86a9f38c    host      host      local
29390bb3509c    new-bridge  bridge      local
3b8cce6433ef    none      null      local
```

기본적으로 제공되는 네트워크 외에 새로운 네트워크를 구성하기 위해서는 위와 같이 명령어를 작성하면 됩니다. 보통은 명령어를 입력해서 생성하기보다는 뒤에 이어질 `docker-compose.yaml` 내에 세팅하는 경우가 더 많습니다.

## 3) `connect` - 컨테이너를 네트워크에 연결

```
sudo docker network connect new-bridge webserver1
sudo docker container inspect webserver1
```

```
ggingmin@ubuntu_server:~$ sudo docker network connect new-bridge webserver1
ggingmin@ubuntu_server:~$ sudo docker container inspect webserver1
```

```

"Networks": {
    "bridge": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": null,
        "NetworkID": "bfc1fe26be9d418d1c4b5f14497e1ba6ee112947e53f61653cd5c0d61ea747ab",
        "EndpointID": "1aa3fc6debd8855c13b07a0a6535a4e0c93aed17a3bdb3679c5c2301ea2a619",
        "Gateway": "172.17.0.1",
        "IPAddress": "172.17.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:00:02",
        "DriverOpts": null
    },
    "new-bridge": {
        "IPAMConfig": {},
        "Links": null,
        "Aliases": [
            "1815b36eb518"
        ],
        "NetworkID": "995af23b884af3c924874b3e8734d5bebbfca5b00e5fa651056b68a3a1d36d30",
        "EndpointID": "ccf4ec3c26192561cae7609855afb547560b132d5816037d8d89345ff19b663b",
        "Gateway": "172.18.0.1",
        "IPAddress": "172.18.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:12:00:02",
        "DriverOpts": {}
    }
}

```

기존에 컨테이너 생성시 디폴트로 연결되어 있는 `bridge` 네트워크는 유지되면서 새롭게 `new-bridge` 와 연결되었습니다. 즉 하나의 컨테이너가 두 개의 네트워크에서 각각 공유될 수 있는 것입니다.

자세히 들여다보면 이 두 네트워크는 게이트웨이가 각각 `172.17.0.1` 과 `172.18.0.1` 로 서로 다릅니다. 이렇게 대역이 다른 네트워크 끼리는 통신이 불가능 합니다. 컨테이너 입장에서는 두 가지 대역과 모두 통신이 가능한 상태가 되었네요.

```

sudo docker container run -d -p 8082:80 --name=webserver4 --net=new-bridge
httpd sudo docker container inspect webserver4

```

```

gingmin@ubuntu_server:~$ sudo docker container run -d -p 8082:80 --name=webserver4 --net=new-bridge httpd
"Networks": {
    "new-bridge": {
        "IPAMConfig": {},
        "Links": null,
        "Aliases": [
            "e40d1d98d368"
        ],
        "NetworkID": "995af23b884af3c924874b3e8734d5bebbfca5b00e5fa651056b68a3a1d36d30",
        "EndpointID": "93ae730667e9dfaf5ee09076015d8d9620212c5f90f17984c67d51d135abd448",
        "Gateway": "172.18.0.1",
        "IPAddress": "172.18.0.3",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:12:00:03",
        "DriverOpts": null
    }
}

```

`docker container run` 에 `--net` 옵션을 추가해서 컨테이너를 생성하면 디폴트 네트워크 설정값인 `bridge` 를 연결하지 않고 명령어에 설정된 `new-bridge` 만 연결됩니다.

4) `disconnect` - 컨테이너를 네트워크에서 해제

```
sudo docker network disconnect new-bridge webserver1
```

#### 5) `inspect` - 네트워크 상세정보 조회

```
sudo docker network inspect new-bridge
```

#### 6) `rm` - 네트워크 삭제

```
sudo docker network rm new-bridge
```

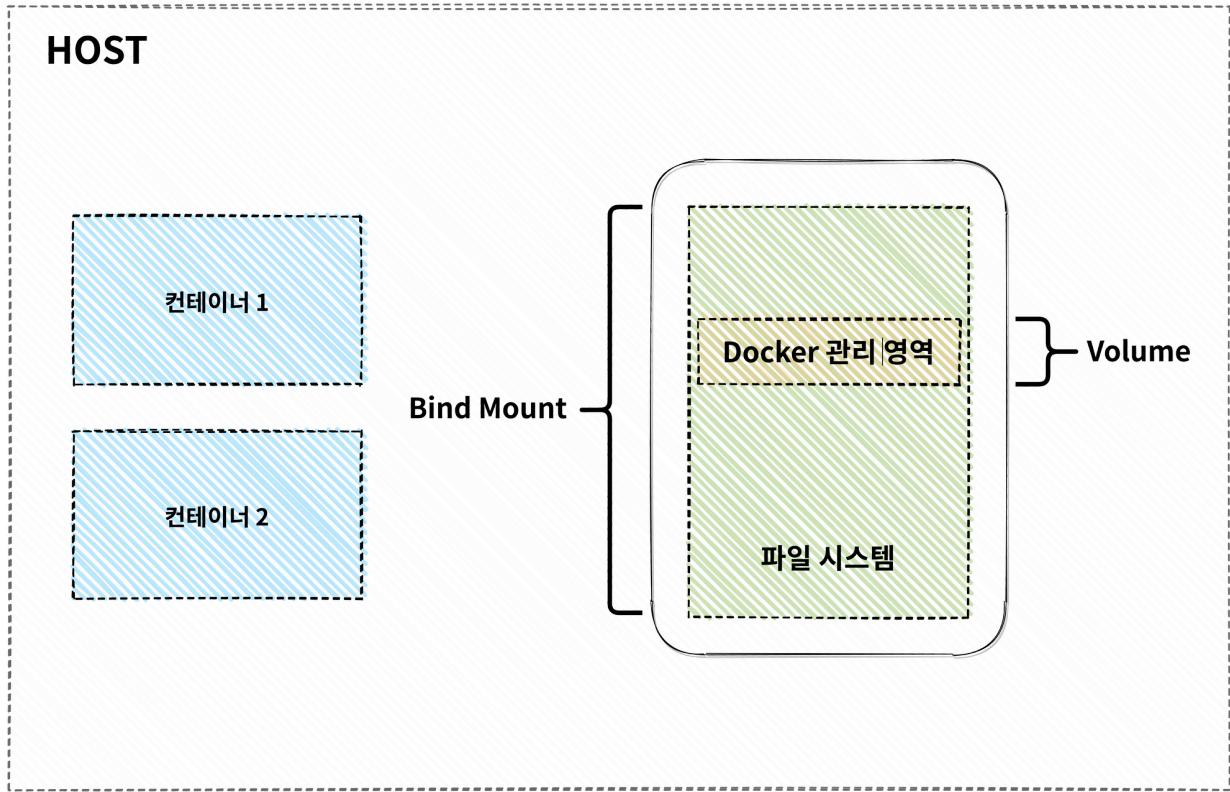
## 6.2 도커 볼륨

컨테이너 내부는 어플리케이션을 실행하기 위한 다양한 파일로 구성되어 있습니다. 이러한 파일들을 관리하기 위해 디스크 드라이브가 내부적으로 작동하고 있죠. Dockerfile에서 `COPY` 와 `ADD` 명령어를 사용하면서 로컬의 파일을 이미지 빌드 단계에서 복사했던 기억이 뇌리를 스칩니다.

그렇다면 컨테이너 내부의 파일은 영원할까요? 질문을 조금 수정해보겠습니다. 컨테이너는 영원한가요?

컨테이너는 언제든지 멈추고 삭제할 수 있습니다. 컨테이너 내부의 디스크에서 존재하던 파일 혹은 데이터 역시 컨테이너를 삭제하게 되면 함께 사라집니다. 영구적으로 혹은 일정 기간동안 데이터를 보관하는 용도로 사용해야 한다면 이를 관리하기 위해 다른 방법을 생각해야 합니다.

도커에서는 `Volume` 과 `Bind Mount`라는 개념으로 Host의 일부 영역을 할당해 스토리지로 사용할 수 있게 해준다. 두 개념을 비교한 후에 각각의 쓰임에 대해 알아보도록 하겠습니다.



### 6.2.1 Bind Mount

Bind Mount는 초기 도커에서 사용된 스토리지 기능으로, Host 경로 일부를 설정해 컨테이너에 마운트하는 기능을 가리킵니다. 쉽게 말해 '/Desktop'이라는 경로를 USB 삼아 컨테이너에 연결하는 것과 같죠. 사용하는 방법은 컨테이너를 실행할 당시에 `-v` 옵션을 작성해주는 것입니다.

```
mkdir /home/ubuntu/bindmount-test cd /home/ubuntu/bindmount-test touch bindmount.txt echo "bind-mount" > bindmount.txt sudo docker container run -d -it --name bindmount -v /home/ubuntu/bindmount-test:/bindmount-test
ubuntu:18.04 sudo docker container run -v [호스트 경로]:[컨테이너 경로] [이미지명:태그]
```

```
ggingmin@ubuntu_server:/home/ubuntu/bindmount-test$ sudo docker container run -d -it --name bindmount -v /home/ubuntu/bindmount-test:/bindmount-test
2da67fc827c102ff12fb17e0a1555f5c7885ea4b9074bf5b3030454db1971c0
ggingmin@ubuntu_server:/home/ubuntu/bindmount-test$ sudo docker exec -it bindmount ls -al /bindmount-test
total 12
drwxr-xr-x 2 root root 4096 Aug 18 15:56 .
drwxr-xr-x 1 root root 4096 Sep  6 09:34 ..
-rw-r--r-- 1 root root   11 Aug 18 15:58 bindmount.txt
ggingmin@ubuntu_server:/home/ubuntu/bindmount-test$ ls -al
total 12
drwxr-xr-x 2 root root 4096 Aug 18 15:56 .
drwxr-xr-x 3 root root 4096 Aug 18 15:21 ..
-rw-r--r-- 1 root root   11 Aug 18 15:58 bindmount.txt
ggingmin@ubuntu_server:/home/ubuntu/bindmount-test$
```

위 명령어를 실행하면 호스트의 `/home/ubuntu/bindmount-test`, `/var/lib/docker/volumes/<해시값>`의 경로가 컨테이너의 `/bindmount-test` 와 연결됩니다. 이렇게 연결된 경로는 파일을 복사하는 개념이 아니라 완전히 같은 디렉토리로 취급됩니다.

### 6.2.2 Volume

Volume은 Bind Mount 이후에 나온 개념으로, 호스트의 경로를 임의로 지정할 수 없습니다. 대신 도커 엔진에서 관리하는 `/var/lib/docker/volumes/<볼륨명>` 으로만 세팅이 됩니다. 볼륨은 컨테이너를 실행하는 단계에서 생성할 수 있으며 볼륨만 개별적으로 생성하는 것도 가능합니다.

```
~$ sudo docker container run -d -it --name volume -v volume1:/volume-test  
ubuntu:18.04 ~$ sudo docker container attach volume # echo "volume" >  
./volume-test/volume.txt ~$ cd /var/lib/docker/volumes/volume1/_data ~$ cat  
volume.txt
```

```
ggingmin@ubuntu_server:/var/lib/docker/volumes$ sudo docker container run -d -it --name volume -v volume1:/volume-test ubuntu:18.04  
7ec714619165c54b0343ef7fe87f3d240a18a62a537fd1cc6d10067523abf722  
ggingmin@ubuntu_server:/var/lib/docker/volumes$ cd /var/lib/docker/volumes/  
ggingmin@ubuntu_server:/var/lib/docker/volumes$ ls -al  
ls: cannot open directory '.': Permission denied  
ggingmin@ubuntu_server:/var/lib/docker/volumes$ sudo ls -al  
total 88  
drwx----x 10 root root 4096 Sep 6 09:38 .  
drwx---x 13 root root 4096 Sep 6 04:08 ..  
drwx----x 3 root root 4096 Aug 20 06:49 076035c72b90457d6a8ad60d877bec453d43fde0b98f2c0effb4d5ecf3534d4b  
drwx----x 3 root root 4096 Aug 20 06:42 63220edb7506921e50a21483250f5071bc4fc5e92c280ce633e2160cf880228  
drwx----x 3 root root 4096 Aug 20 06:32 82d2daf601ae49c85485f01f72ce4934b339f678c701c0738b35931c4cf4306d  
drwx----x 3 root root 4096 Aug 20 06:51 8cccd0558818ee94eb2687026355bf95f66eee2c70c00674c080737a23b52dece  
drwx----x 3 root root 4096 Aug 20 06:23 951de72d89a56799045a0d740fa3f56d7ef05dc2e718f889b116cea52ca35f0f  
brw----- 1 root root 253, 0 Sep 6 04:08 backingFsBlockDev  
drwx----x 3 root root 4096 Aug 20 06:45 d375fb8940869122b78eaf91289a4bc402dd7868c9c14c8d668d016ec8bea5c  
drwx----x 3 root root 4096 Aug 20 06:37 ed4cc91c0b23dbe4c8c021702f332b16263212b86c3462e1373b8dd708b8e667  
-rw----- 1 root root 65536 Sep 6 09:38 metadata.db  
drwx----x 3 root root 4096 Sep 6 09:38 volume1  
  
ggingmin@ubuntu_server:/var/lib/docker/volumes/volume1$ sudo docker container attach volume  
root@259eb77c81a9:/# echo "volume" > ./volume-test/volume.txt  
root@259eb77c81a9:/# exit  
ggingmin@ubuntu_server:/var/lib/docker/volumes/volume1$ cd _data  
ggingmin@ubuntu_server:/var/lib/docker/volumes/volume1/_data$ ls -al  
total 12  
drwxr-xr-x 2 root root 4096 Sep 6 09:43 .  
drwx----x 3 root root 4096 Sep 6 09:38 ..  
-rw-r--r-- 1 root root 7 Sep 6 09:44 volume.txt  
ggingmin@ubuntu_server:/var/lib/docker/volumes/volume1/_data$ cat volume.txt  
volume  
ggingmin@ubuntu_server:/var/lib/docker/volumes/volume1/_data$
```

해당 경로를 확인해보면 컨테이너 생성 시 지정했던 볼륨명인 `volume1` 이름의 디렉토리가 생성된 것을 확인 할 수 있습니다. 볼륨을 통해 저장된 데이터는 `/var/lib/docker/volumes/[볼륨명]/_data` 에 저장되기 때문에 컨테이너에서 입력했던 문자열 텍스트 파일이 호스트에서도 동일한 내용으로 조회됩니다.

```
# 볼륨 개별 생성 sudo docker volume create volume2
```