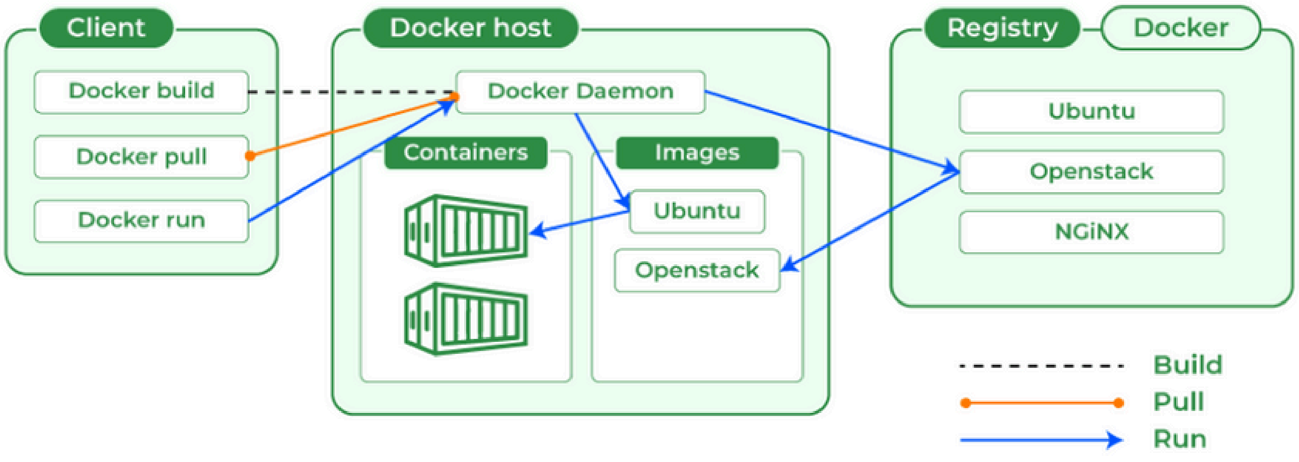
DOCKER :

Docker is an open-source platform that automates the deployment, scaling, and management of applications within lightweight, portable containers. It enables developers to package applications with all their dependencies, ensuring that they can run consistently across different computing environments. Here’s a deeper look at Docker:

Docker Architecture :



Components of Docker:

1)Container

2)Images

3) Volumes

4)Networks

Docker Container:

A Docker container is a lightweight, portable, and executable unit that packages software and its dependencies in a standardized format. Containers are created from Docker images and provide a runtime environment that isolates applications from the host system, allowing them to run consistently across various environments.

Key Features of Docker Containers

1. **Isolation**: Containers run in isolated environments, meaning they do not interfere with one another or the host system. Each container has its own filesystem, processes, and

network stack.
2. **Lightweight**: Containers share the host's operating system kernel, making them more lightweight compared to traditional virtual machines. This allows for faster startup times and lower resource usage.
3. **Portability**: A container can run on any machine with Docker installed, regardless of the underlying infrastructure, ensuring consistency between development, testing, and production environments.
4. **Scalability**: Containers can be easily replicated, scaled up or down to meet demand, and orchestrated with tools like Docker Swarm or Kubernetes.
5. **Immutability**:Containers are generally considered immutable. If changes are needed, a new container image is created, rather than modifying the existing one. This helps maintain consistency and reduces deployment issues.

Structure of a Docker Container

- **Filesystem**: Each container has its own filesystem that is created from a Docker image. This filesystem is built in layers, inheriting from the underlying image.
- **Processes**: Containers run processes in their own isolated space, separate from the host and other containers.
- **Networking**: Containers can communicate with each other and the host through defined network configurations.

Creating and Running Containers

create and run containers using the docker run command.

```
docker run -d --name my_container my_image:latest
```

Managing Docker Containers

- **Listing Containers**: You can see all running containers with → `docker ps -a`
- **Stopping a Container**: To stop a running container → `docker stop my_container`
- **Removing a Container**: To delete a stopped container → `docker rm my_container`
- **Viewing Logs**: To check the logs of a container → `docker logs my_container`

Use Cases

- **Microservices**: Each microservice can be deployed in its own container, allowing for independent scaling and management.
- **Development Environments**: Developers can run containers that mimic production environments, reducing discrepancies and bugs.
- **Testing**: Containers provide a clean and consistent environment for testing applications.

Docker Images :

Docker images are lightweight, standalone, and executable packages that contain everything needed to run a piece of software, including the code, runtime, libraries, environment variables, and configuration files. Here’s a closer look at Docker images:

Key Features of Docker Images

1. **Layered Architecture**: Docker images are built in layers, with each layer representing a set of file changes. This allows for efficient storage and sharing, as common layers can be reused among different images.
2. **Portability**: Images can be easily shared and run on any machine that has Docker installed, ensuring consistent behavior across different environments.
3. **Versioning**: Each image can have multiple versions (tags), allowing you to manage and deploy specific versions of your applications easily.
4. **Immutable**: Once an image is built, it does not change. If you need to make changes, you create a new image. This immutability helps in maintaining consistency.
5. **Base Images**: You can create images from existing base images (like ubuntu, alpine, or node), which provide a foundation for building your own applications.

A Docker image typically consists of:

- **Filesystem Layers**: Each layer can add or modify files in the image. The final layer is the one that runs when a container is started.
- **Metadata**: Information about the image, such as environment variables, entry points, and command defaults.

Use Cases

- **Application Deployment**: Package applications with all dependencies to ensure they run reliably in different environments.
- **Microservices**: Each microservice can be encapsulated in its own Docker image, allowing for independent development, testing, and deployment.
- **Development Environments**: Set up consistent development environments that match production setups.

- **Build the image :** docker build -t my_image:latest .
- **Listing Images**: You can see all your images with → docker images
- **Removing Images**: To delete an image → docker rmi image_name
- **Tagging Images**: You can tag images for easier management → docker tag image_name new_name:tag

Docker Volumes:

Volumes : A Docker volume is a persistent storage mechanism that allows you to store data outside of your Docker containers. Volumes are managed by Docker and can be shared among multiple containers, which makes them ideal for scenarios where you need to preserve data even if a container is stopped or deleted.

Key Features of Docker Volumes:

1. **Persistence**: Data in a volume remains even if the container using it is removed. This is crucial for database storage, user uploads, and other data that needs to persist.
2. **Isolation**: Volumes provide a way to isolate the storage from the container filesystem, which can simplify backups and data management.
3. **Sharing**: Multiple containers can access the same volume, allowing for easy data sharing and collaboration between containers.
4. **Performance**: Volumes are designed for high performance and can be optimized for different storage backends.
5. **Management**: Docker provides commands to easily create, inspect, and manage volumes.

Creating container 1

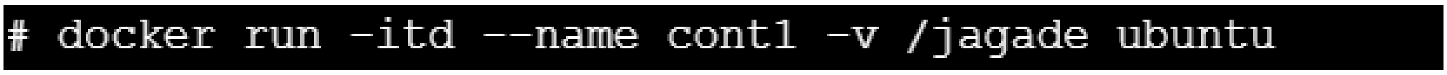
1)docker run -itd --name cont1 -v /jagade ubuntu → creating container

2)docker exec -it cont1 bash

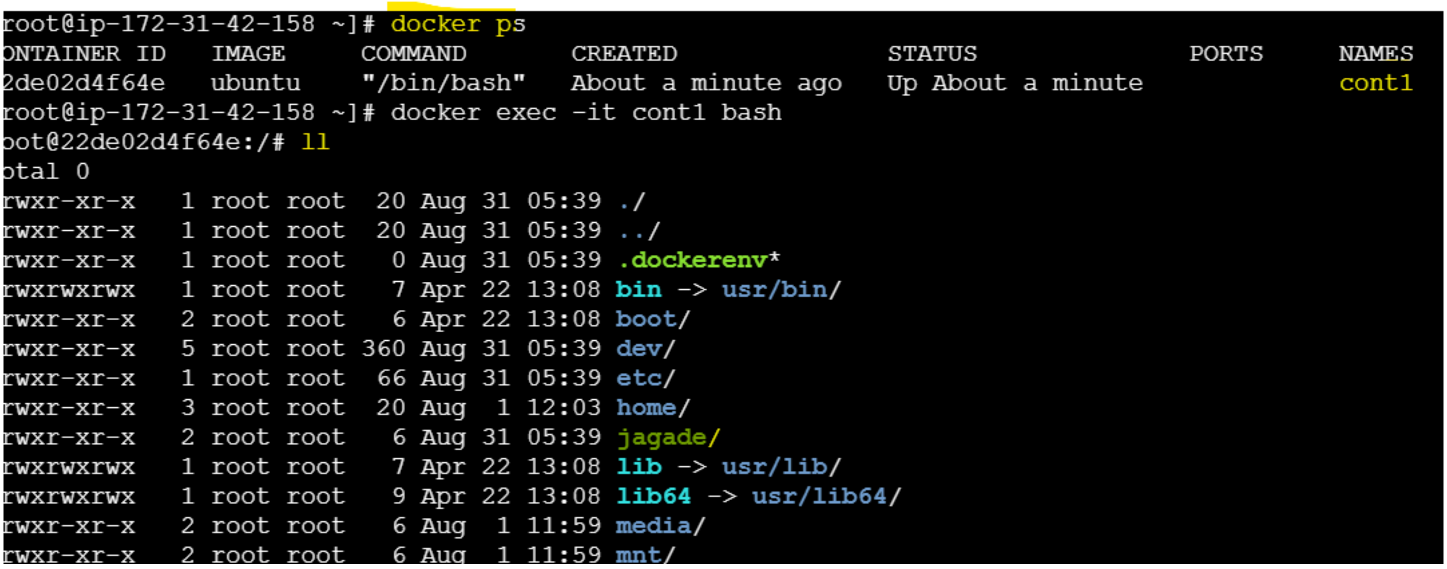
3) cd /(volume name)

4) create some dummy files (touch aws{1..5}

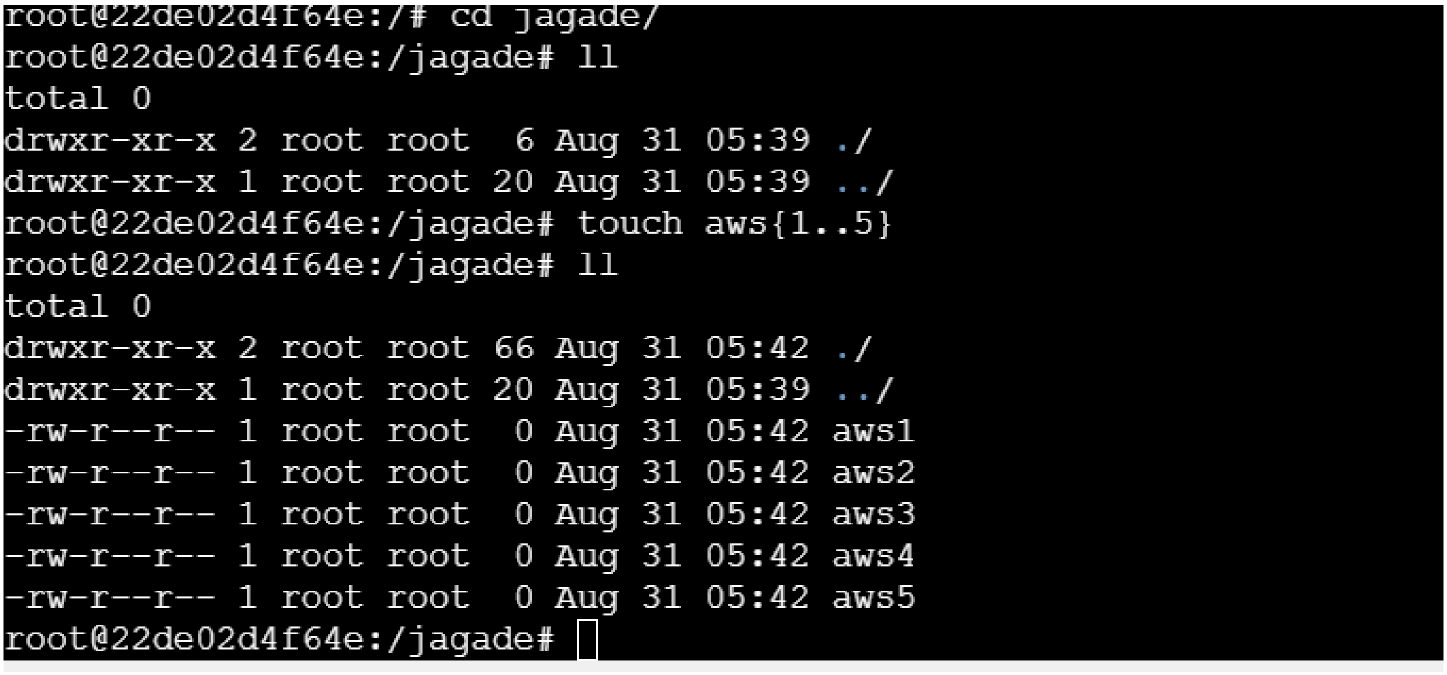
5)exit (from container 1)



```
# docker run -itd --name cont1 -v /jagade ubuntu
```



```
root@ip-172-31-42-158 ~]# docker ps
ONTAINER ID   IMAGE     COMMAND       CREATED              STATUS              PORTS     NAMES
2de02d4f64e   ubuntu    "/bin/bash"   About a minute ago   Up About a minute             cont1
root@ip-172-31-42-158 ~]# docker exec -it cont1 bash
oot@22de02d4f64e:/# ll
otal 0
rwxr-xr-x   1 root root  20 Aug 31 05:39 ./
rwxr-xr-x   1 root root  20 Aug 31 05:39 ../
rwxr-xr-x   1 root root   0 Aug 31 05:39 .dockerenv*
rwxrwxrwx   1 root root   7 Apr 22 13:08 bin -> usr/bin/
rwxr-xr-x   2 root root   6 Apr 22 13:08 boot/
rwxr-xr-x   5 root root 360 Aug 31 05:39 dev/
rwxr-xr-x   1 root root  66 Aug 31 05:39 etc/
rwxr-xr-x   3 root root  20 Aug  1 12:03 home/
rwxr-xr-x   2 root root   6 Aug 31 05:39 jagade/
rwxrwxrwx   1 root root   7 Apr 22 13:08 lib -> usr/lib/
rwxrwxrwx   1 root root   9 Apr 22 13:08 lib64 -> usr/lib64/
rwxr-xr-x   2 root root   6 Aug  1 11:59 media/
rwxr-xr-x   2 root root   6 Aug  1 11:59 mnt/
```


```
root@22de02d4f64e:/# cd jagade/
root@22de02d4f64e:/jagade# ll
total 0
drwxr-xr-x 2 root root  6 Aug 31 05:39 ./
drwxr-xr-x 1 root root 20 Aug 31 05:39 ../
root@22de02d4f64e:/jagade# touch aws{1..5}
root@22de02d4f64e:/jagade# ll
total 0
drwxr-xr-x 2 root root 66 Aug 31 05:42 ./
drwxr-xr-x 1 root root 20 Aug 31 05:39 ../
-rw-r--r-- 1 root root  0 Aug 31 05:42 aws1
-rw-r--r-- 1 root root  0 Aug 31 05:42 aws2
-rw-r--r-- 1 root root  0 Aug 31 05:42 aws3
-rw-r--r-- 1 root root  0 Aug 31 05:42 aws4
-rw-r--r-- 1 root root  0 Aug 31 05:42 aws5
root@22de02d4f64e:/jagade# 
```

Creating container 2:

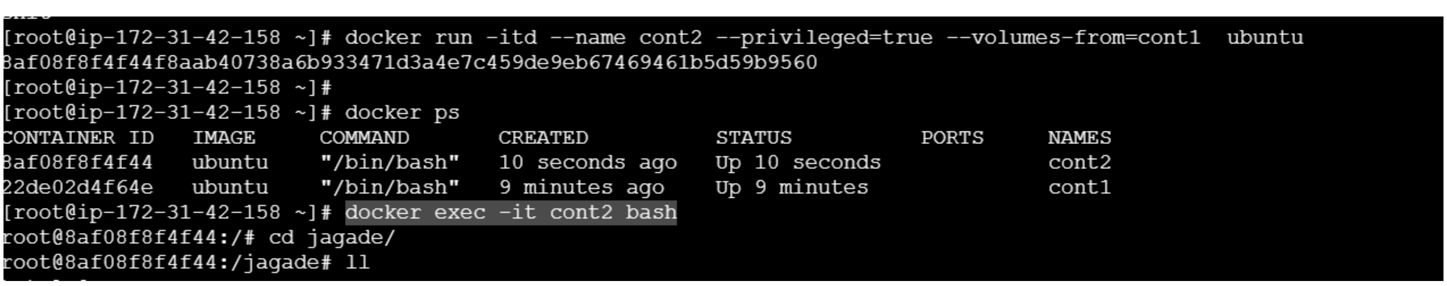
1) docker run -itd --name cont2 --privileged=true --volumes-from=cont1 ubuntu

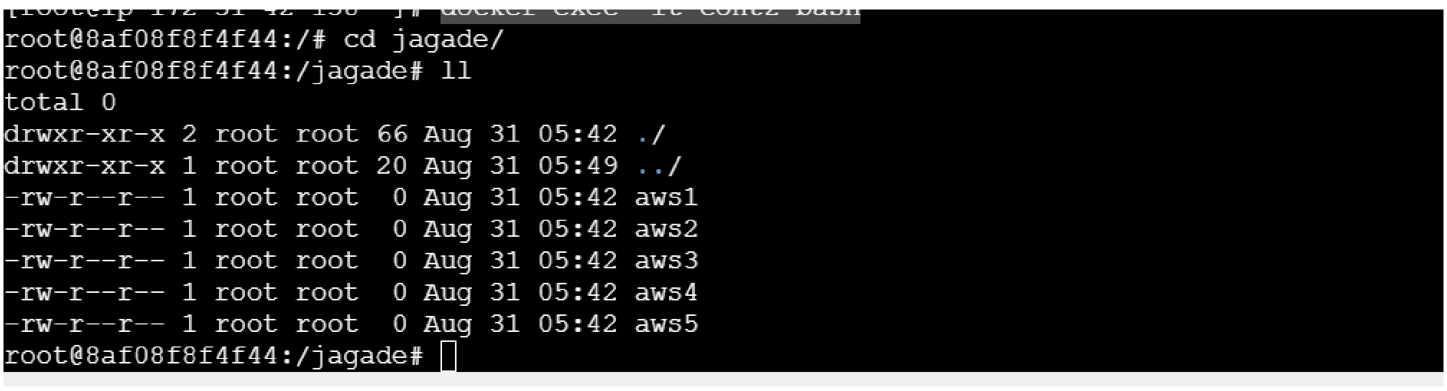
2)docker exec -it cont2 bash

3) cd /volume name

Note :1) Same file in container 1 and same file will be in container 2

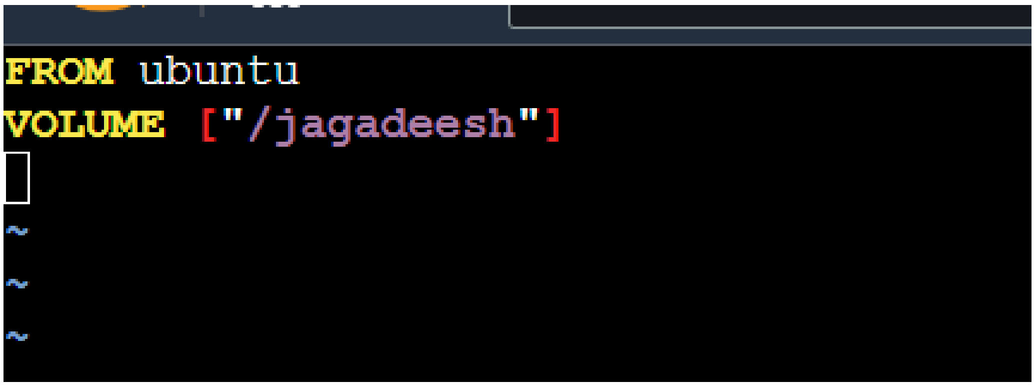
2) privileged=true → giving permission to cntainer2 && attaching volume to cont2


```
[root@ip-172-31-42-158 ~]# docker run -itd --name cont2 --privileged=true --volumes-from=cont1  ubuntu
8af08f8f4f44f8aab40738a6b933471d3a4e7c459de9eb67469461b5d59b9560
[root@ip-172-31-42-158 ~]#
[root@ip-172-31-42-158 ~]# docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS          PORTS     NAMES
8af08f8f4f44   ubuntu    "/bin/bash"   10 seconds ago   Up 10 seconds             cont2
22de02d4f64e   ubuntu    "/bin/bash"   9 minutes ago    Up 9 minutes              cont1
[root@ip-172-31-42-158 ~]# docker exec -it cont2 bash
root@8af08f8f4f44:/# cd jagade/
root@8af08f8f4f44:/jagade# ll
```


```
[root@ip-172-31-42-158 ~]# docker exec -it cont2 bash
root@8af08f8f4f44:/# cd jagade/
root@8af08f8f4f44:/jagade# ll
total 0
drwxr-xr-x 2 root root 66 Aug 31 05:42 ./
drwxr-xr-x 1 root root 20 Aug 31 05:49 ../
-rw-r--r-- 1 root root  0 Aug 31 05:42 aws1
-rw-r--r-- 1 root root  0 Aug 31 05:42 aws2
-rw-r--r-- 1 root root  0 Aug 31 05:42 aws3
-rw-r--r-- 1 root root  0 Aug 31 05:42 aws4
-rw-r--r-- 1 root root  0 Aug 31 05:42 aws5
root@8af08f8f4f44:/jagade# 
```

Docker volume creating via Docker file :

vim Dockerfile



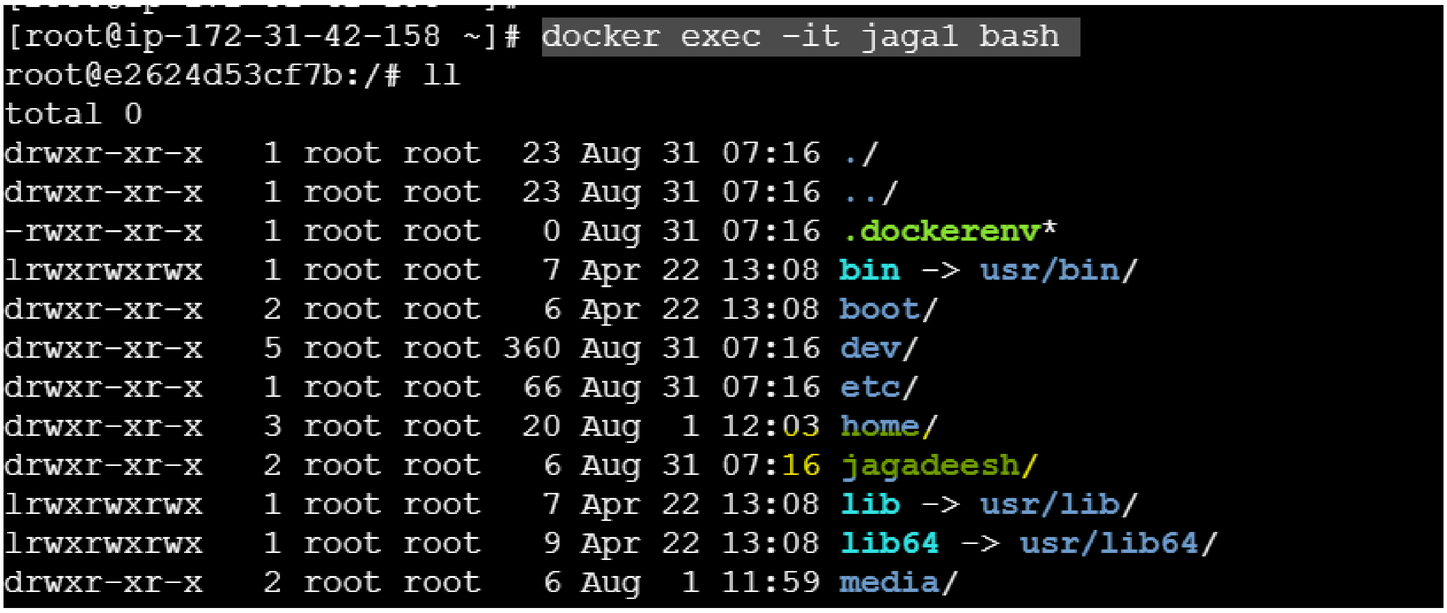
```
FROM ubuntu
VOLUME ["/jagadeesh"]

~
~
~
```

docker build -t image2 .

docker run -itd --name jaga1 image2

docker exec -it jaga1 bash



```
[root@ip-172-31-42-158 ~]# docker exec -it jaga1 bash
root@e2624d53cf7b:/# ll
total 0
drwxr-xr-x   1 root root  23 Aug 31 07:16 ./
drwxr-xr-x   1 root root  23 Aug 31 07:16 ../
-rwxr-xr-x   1 root root   0 Aug 31 07:16 .dockerenv*
lrwxrwxrwx   1 root root   7 Apr 22 13:08 bin -> usr/bin/
drwxr-xr-x   2 root root   6 Apr 22 13:08 boot/
drwxr-xr-x   5 root root 360 Aug 31 07:16 dev/
drwxr-xr-x   1 root root  66 Aug 31 07:16 etc/
drwxr-xr-x   3 root root  20 Aug  1 12:03 home/
drwxr-xr-x   2 root root   6 Aug 31 07:16 jagadeesh/
lrwxrwxrwx   1 root root   7 Apr 22 13:08 lib -> usr/lib/
lrwxrwxrwx   1 root root   9 Apr 22 13:08 lib64 -> usr/lib64/
drwxr-xr-x   2 root root   6 Aug  1 11:59 media/
```

Creating an image from a container Real time Scenario :

Creating an image from a container is a common practice in Docker for several reasons:

- **Capture State:** When you've configured a container (installed software, modified settings, etc.), you can create an image to capture that exact state. This allows you to reproduce the environment easily in the future.
- **Version Control:** By creating images from containers at different stages, you can maintain versions of your application or environment. This makes it easier to roll back to previous configurations if needed.
- **Sharing and Distribution:** Once an image is created, it can be shared and distributed via a Docker registry (like Docker Hub). This enables others to run your application in the same environment without needing to replicate the setup process.
- **Consistency:** Images ensure that environments are consistent across different stages (development, testing, production). This reduces issues that arise from environmental differences.

- **Backing Up:** Creating an image from a container can serve as a backup. If the container fails or is deleted, you can easily restore it from the image.
- **Simplifying Deployment:** By creating an image that includes everything needed to run an application, deployment becomes simpler. You can spin up new containers from the image without having to repeat setup steps.

Command to create :

docker run -itd --name cont3 -v /Jagadeesh ubuntu

docker exec -it cont3 bash (create some files, mkdir some directory in container root to verify)

docker commit cont3 image3

docker run -itd --name cont3 image3

docker exec -t cont4 bash (ll → you can see the root files)

cd / Jagadeesh (Not able to access the files in the v→ no files are available .

Host to Container :

The "Host to Container" concept in Docker refers to the interactions and data flow between the host machine (where Docker is installed) and the containers running on that host. Understanding this relationship is crucial for effectively managing containerized applications. Here are the key aspects:

1. Networking:

- Ports: Containers typically run on isolated networks, but they can communicate with the host and other containers. You can expose container ports to the host by using the -p flag in the docker run command. For example, -p 8080:80 maps port 80 in the container to port 8080 on the host.
- Bridge Network: By default, Docker uses a bridge network, allowing containers to communicate with each other and the host.

2. File System Access:

- Volumes and Bind Mounts: You can share files between the host and containers using Docker volumes or bind mounts. A volume is managed by Docker, while a bind mount directly links a host directory to a container directory. This is useful for persisting data or sharing configuration files.
- Example: To mount a directory from the host into a container, you can use the -voption:
- bash
- Copy code
- docker run -v /path/on/host:/path/in/container my_image

3. Resource Allocation:

- Docker allows you to specify resource limits for containers, such as CPU and memory usage. This ensures that containers do not overwhelm the host's resources.
- You can set these limits using options like --memory and --cpus in the docker run command.

4. Process Isolation:

- Containers run in isolated environments but share the host's kernel. This means that while processes in containers are separate, they are still subject to the host's resource constraints and can interact with the host's processes under certain conditions.

5. Inter-Container Communication:

- Containers can communicate with the host and with each other over the network. You can set up custom networks to control this communication or use Docker's default bridge network.

6. Host Configuration:

- The host's configuration, including firewall settings and network settings, can impact how containers operate and communicate. It's important to configure the host correctly to allow desired interactions.

Use Cases:

- Web Applications: A web application running in a container can be accessed through a port mapped to the host, enabling users to access it via a browser.
- Data Processing: A container processing data can write output files to a directory on the host, making it easy to retrieve results.

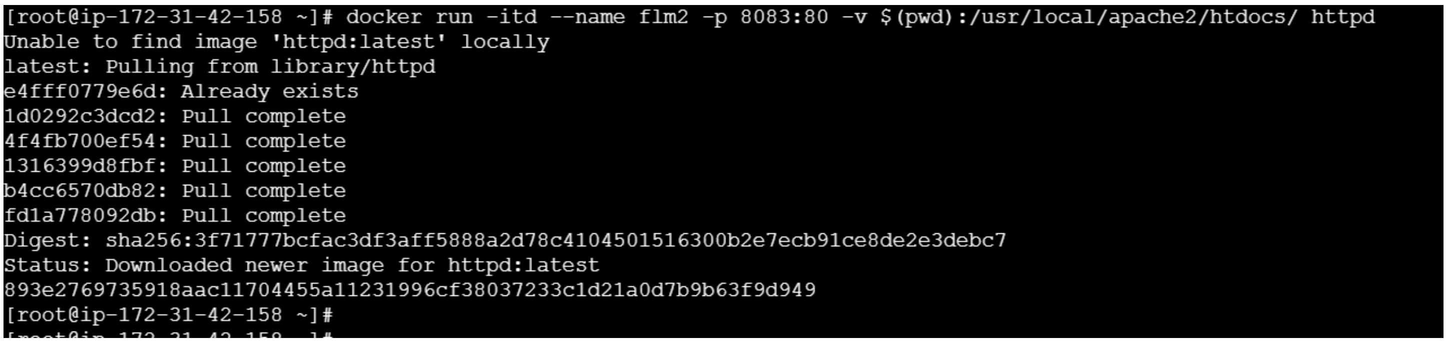
create index.html

docker run -itd --name flm -p 8082:80 -v $(pwd):/usr/share/nginx/html/ nginx

(docker run -itd --name flm2 -p 8083:80 -v $(pwd):/usr/local/apache2/htdocs/ httpd)

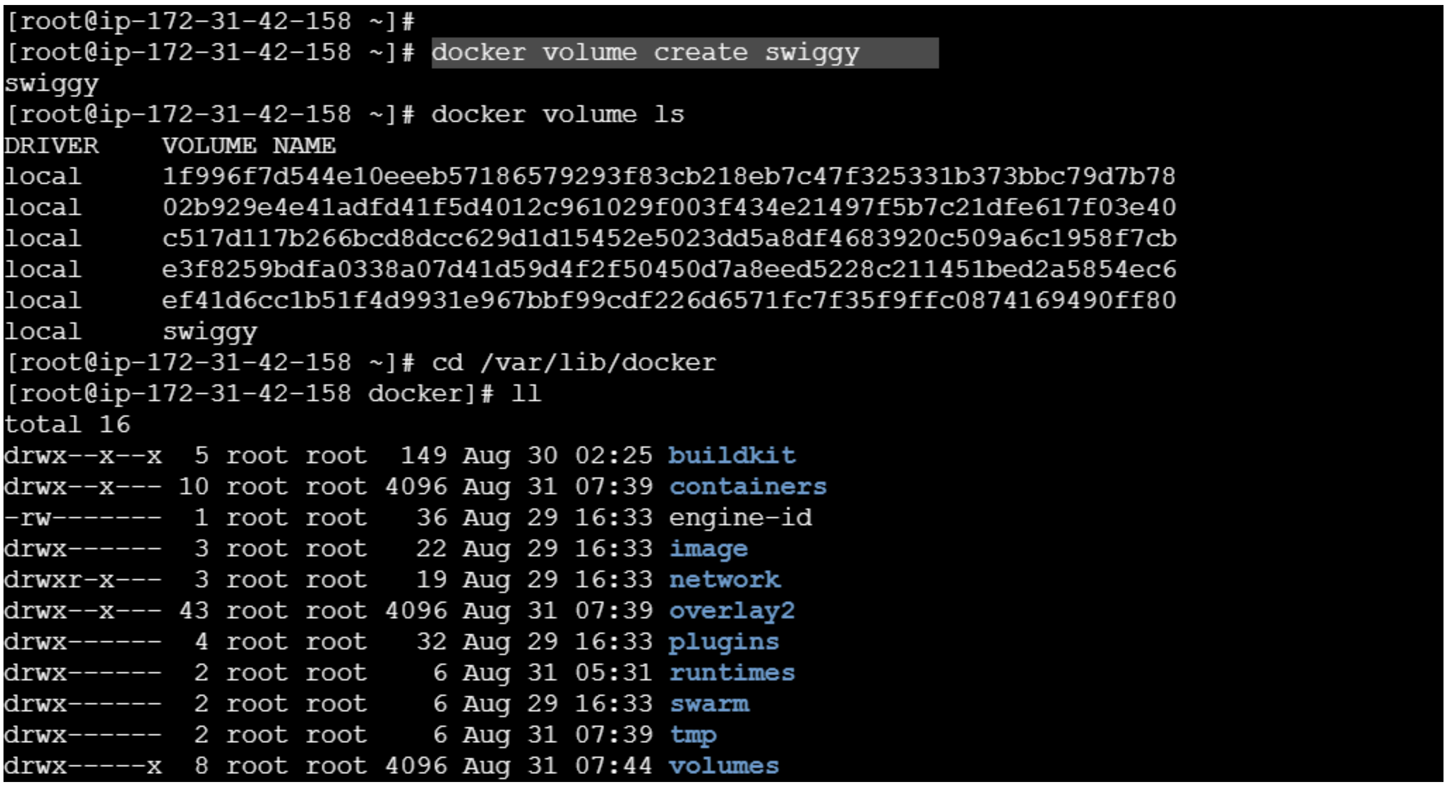
default path : /usr/share/nginx/html for nginx

default path : /usr/local/apache2/htdocs/ for httpd

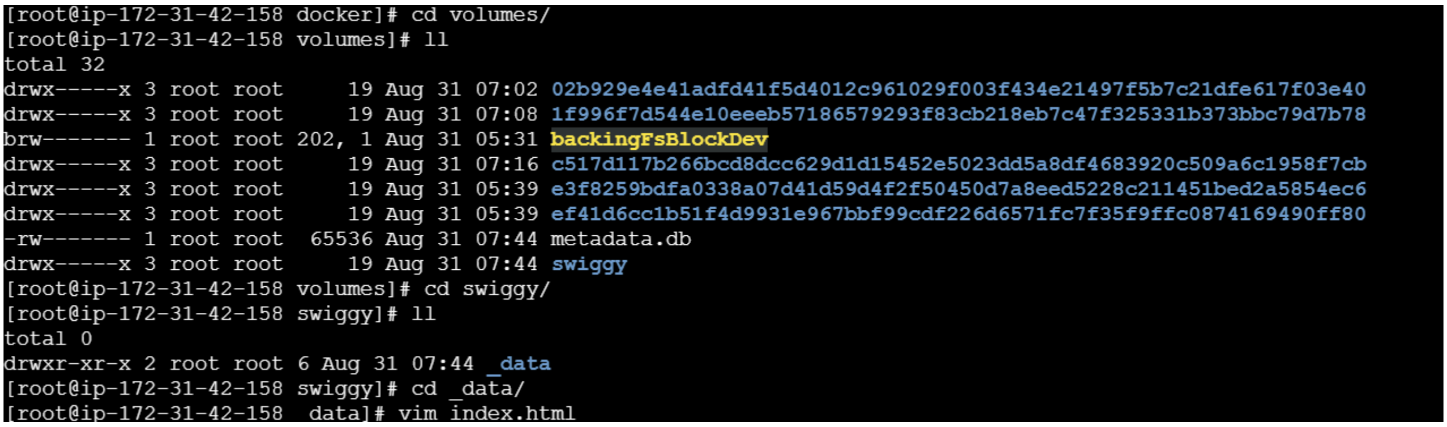


```
[root@ip-172-31-42-158 ~]# docker run -itd --name flm2 -p 8083:80 -v $(pwd):/usr/local/apache2/htdocs/ httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
e4fff0779e6d: Already exists
1d0292c3dcd2: Pull complete
4f4fb700ef54: Pull complete
1316399d8fbf: Pull complete
b4cc6570db82: Pull complete
fd1a778092db: Pull complete
Digest: sha256:3f71777bcfac3df3aff5888a2d78c4104501516300b2e7ecb91ce8de2e3debc7
Status: Downloaded newer image for httpd:latest
893e2769735918aac11704455a11231996cf38037233c1d21a0d7b9b63f9d949
[root@ip-172-31-42-158 ~]#
```

creating volume



```
[root@ip-172-31-42-158 ~]#
[root@ip-172-31-42-158 ~]# docker volume create swiggy
swiggy
[root@ip-172-31-42-158 ~]# docker volume ls
DRIVER    VOLUME NAME
local     1f996f7d544e10eeeb57186579293f83cb218eb7c47f325331b373bbc79d7b78
local     02b929e4e41adfd41f5d4012c961029f003f434e21497f5b7c21dfe617f03e40
local     c517d117b266bcd8dcc629d1d15452e5023dd5a8df4683920c509a6c1958f7cb
local     e3f8259bdfa0338a07d41d59d4f2f50450d7a8eed5228c211451bed2a5854ec6
local     ef41d6cc1b51f4d9931e967bbf99cdf226d6571fc7f35f9ffc0874169490ff80
local     swiggy
[root@ip-172-31-42-158 ~]# cd /var/lib/docker
[root@ip-172-31-42-158 docker]# ll
total 16
drwx--x--x  5 root root  149 Aug 30 02:25 buildkit
drwx--x--- 10 root root 4096 Aug 31 07:39 containers
-rw-------  1 root root   36 Aug 29 16:33 engine-id
drwx------  3 root root   22 Aug 29 16:33 image
drwxr-x---  3 root root   19 Aug 29 16:33 network
drwx--x--- 43 root root 4096 Aug 31 07:39 overlay2
drwx------  4 root root   32 Aug 29 16:33 plugins
drwx------  2 root root    6 Aug 31 05:31 runtimes
drwx------  2 root root    6 Aug 29 16:33 swarm
drwx------  2 root root    6 Aug 31 07:39 tmp
drwx-----x  8 root root 4096 Aug 31 07:44 volumes
```



```
[root@ip-172-31-42-158 docker]# cd volumes/
[root@ip-172-31-42-158 volumes]# ll
total 32
drwx-----x 3 root root     19 Aug 31 07:02 02b929e4e41adfd41f5d4012c961029f003f434e21497f5b7c21dfe617f03e40
drwx-----x 3 root root     19 Aug 31 07:08 1f996f7d544e10eeeb57186579293f83cb218eb7c47f325331b373bbc79d7b78
brw------- 1 root root 202, 1 Aug 31 05:31 backingFsBlockDev
drwx-----x 3 root root     19 Aug 31 07:16 c517d117b266bcd8dcc629d1d15452e5023dd5a8df4683920c509a6c1958f7cb
drwx-----x 3 root root     19 Aug 31 05:39 e3f8259bdfa0338a07d41d59d4f2f50450d7a8eed5228c211451bed2a5854ec6
drwx-----x 3 root root     19 Aug 31 05:39 ef41d6cc1b51f4d9931e967bbf99cdf226d6571fc7f35f9ffc0874169490ff80
-rw------- 1 root root  65536 Aug 31 07:44 metadata.db
drwx-----x 3 root root     19 Aug 31 07:44 swiggy
[root@ip-172-31-42-158 volumes]# cd swiggy/
[root@ip-172-31-42-158 swiggy]# ll
total 0
drwxr-xr-x 2 root root 6 Aug 31 07:44 _data
[root@ip-172-31-42-158 swiggy]# cd _data/
[root@ip-172-31-42-158 _data]# vim index.html
```

root pwd as volume , if we update volume in pwd , it will update in swiggy volume , if we update data in swiggy volume it will update in pwd .

Docker Networks

Docker networks allow containers to communicate with each other and with external systems. Here's a quick overview of the types of networks and how to manage them:

Types of Docker Networks

1. **Bridge Network**:
   - Default network type.
   - Containers can communicate with each other within the same bridge.
   - Use case: Running multiple containers on the same host.
2. **Host Network**:
   - Containers share the host's network stack.
   - No network isolation between the host and containers.

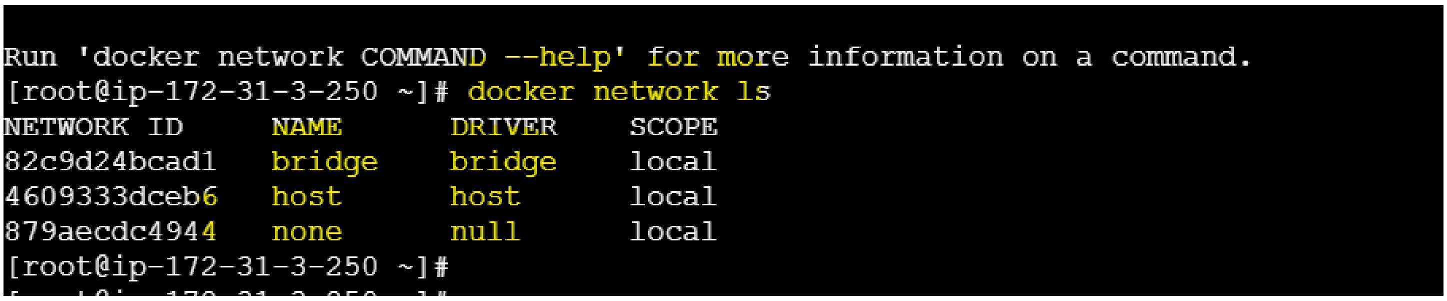
    - Use case: High-performance applications needing low latency.
3. **Overlay Network**:
    - Used in Docker Swarm mode.
    - Allows containers on different hosts to communicate.
    - Use case: Multi-host applications requiring service discovery.
4. **Macvlan Network**:
    - Assigns a MAC address to a container, making it appear as a physical device on the network.
    - Use case: Legacy applications needing direct access to the network.
5. **None Network**:
    - Disables all networking for the container.
    - Use case: For applications that do not require network access.

Commands :

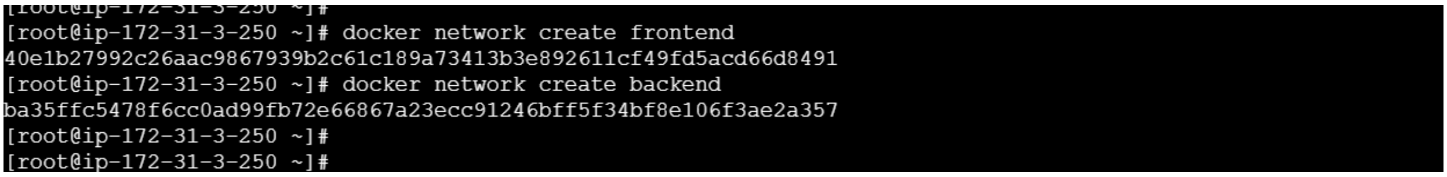
```
 yum install docker -y && systemctl start docker
 docker network ls
 docker network inspect bridge
 docker network create frontend  → creating networks
 docker network create backend →  creating networks
 docker run -itd --name fd_cont --network frontend ubuntu →  creating containers
 docker run -itd --name bd_cont --network backend  ubuntu →   creating containers
docker inspect fd_cont  → Inspecting container
docker inspect bd_cont →  Inspecting container
docker exec -it bd_cont bash  → inside to container
docker exec -it fd_cont bash → inside to container
docker network connect backend fd_cont → connect network
docker network connect frontend bd_cont  → connect network
docker inspect fd_cont
docker inspect bd_cont
docker exec -it fd_cont bash
```

Inside container --Commands

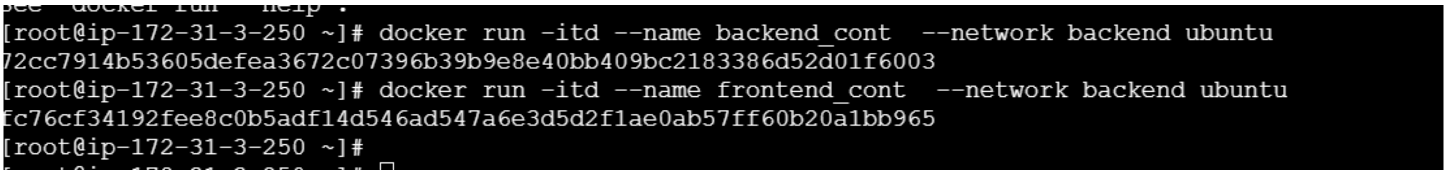
```
1  apt update -y
 2  apt install iputils-ping -y
 3  exit
 4  ping 172.19.0.3
 5  docker history
 6  history
```



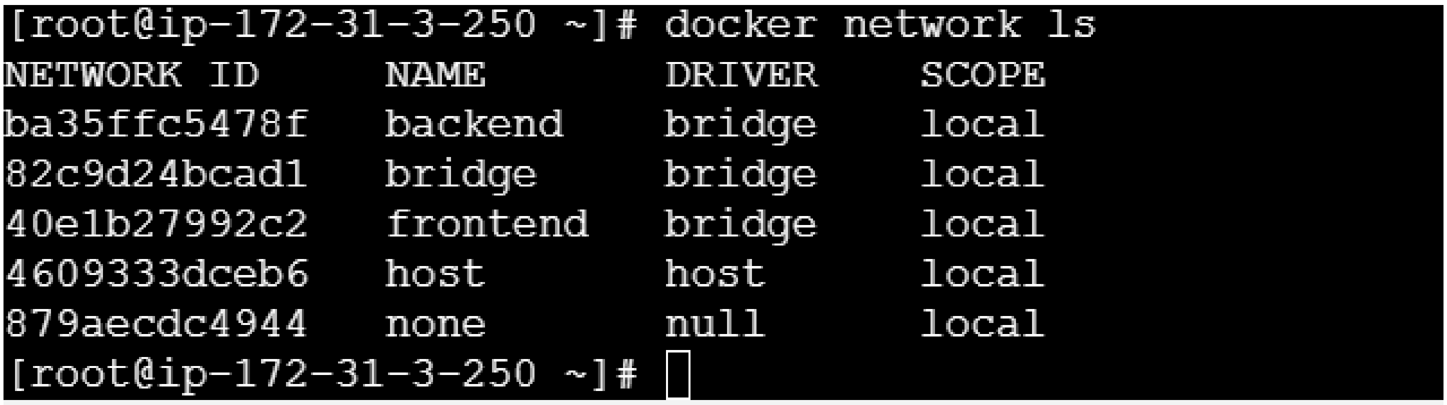
```
Run 'docker network COMMAND --help' for more information on a command.
[root@ip-172-31-3-250 ~]# docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
82c9d24bcad1   bridge    bridge    local
4609333dceb6   host      host      local
879aecdc4944   none      null      local
[root@ip-172-31-3-250 ~]#
```



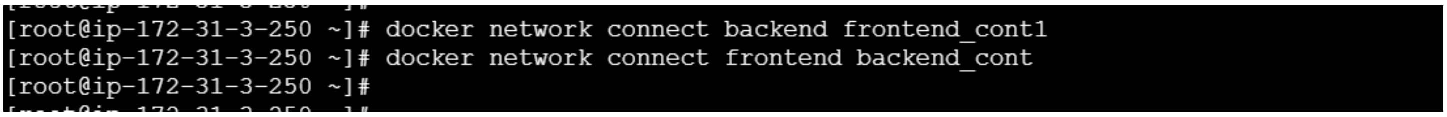
```
[root@ip-172-31-3-250 ~]# docker network create frontend
40e1b27992c26aac9867939b2c61c189a73413b3e892611cf49fd5acd66d8491
[root@ip-172-31-3-250 ~]# docker network create backend
ba35ffc5478f6cc0ad99fb72e66867a23ecc91246bff5f34bf8e106f3ae2a357
[root@ip-172-31-3-250 ~]#
[root@ip-172-31-3-250 ~]#
```



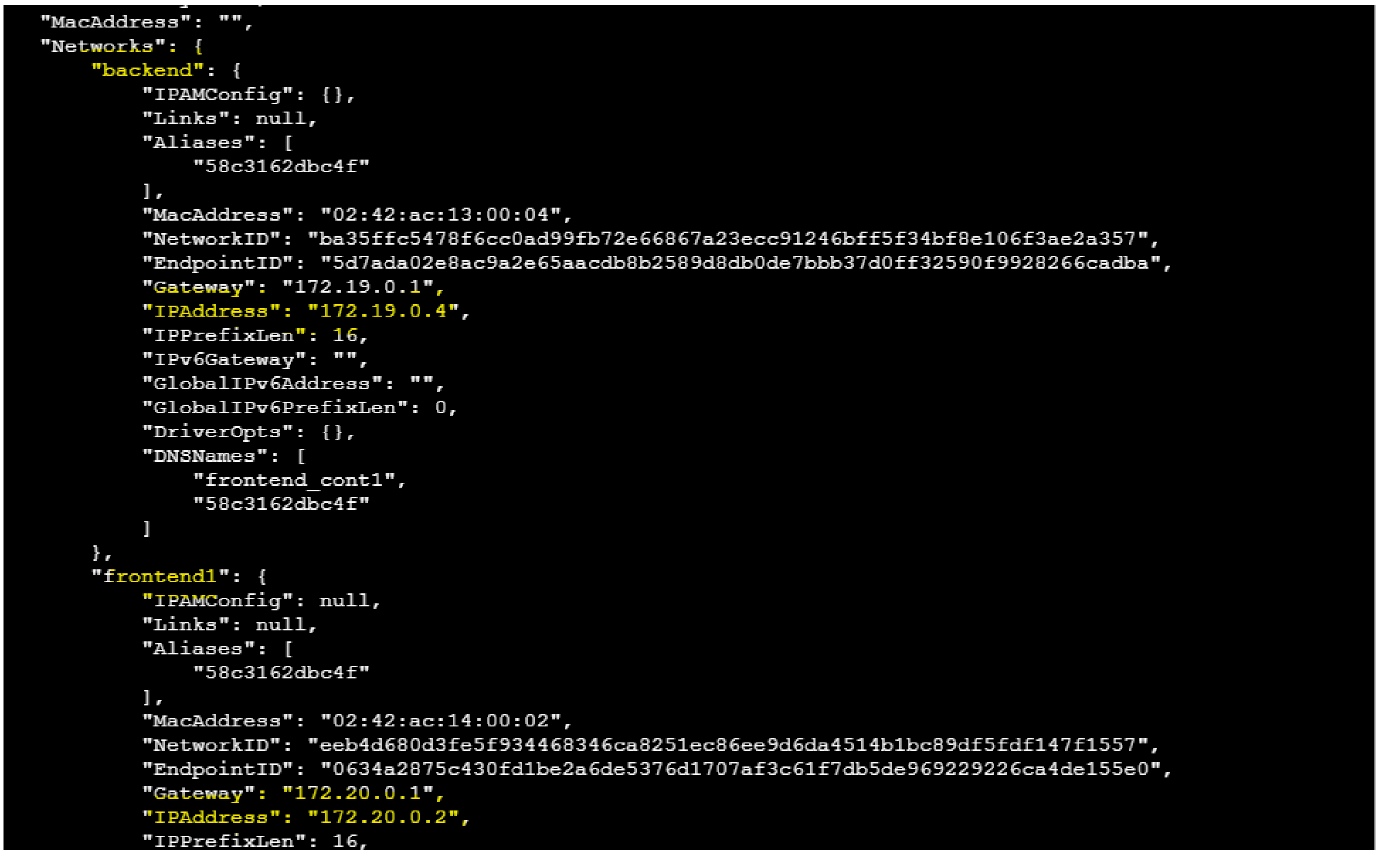
```
[root@ip-172-31-3-250 ~]# docker run -itd --name backend_cont  --network backend ubuntu
72cc7914b53605defea3672c07396b39b9e8e40bb409bc2183386d52d01f6003
[root@ip-172-31-3-250 ~]# docker run -itd --name frontend_cont  --network backend ubuntu
fc76cf34192fee8c0b5adf14d546ad547a6e3d5d2f1ae0ab57ff60b20a1bb965
[root@ip-172-31-3-250 ~]#
```



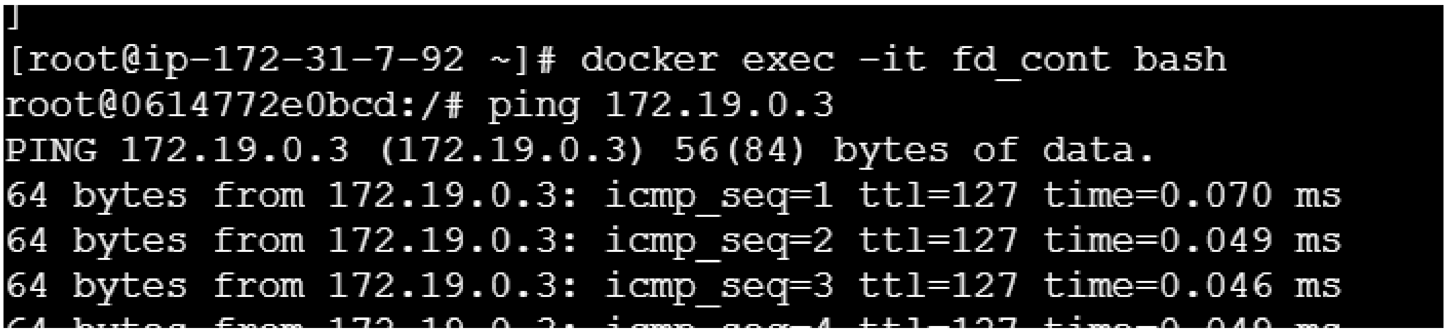
```
[root@ip-172-31-3-250 ~]# docker network ls
NETWORK ID     NAME       DRIVER    SCOPE
ba35ffc5478f   backend    bridge    local
82c9d24bcad1   bridge     bridge    local
40e1b27992c2   frontend   bridge    local
4609333dceb6   host       host      local
879aecdc4944   none       null      local
[root@ip-172-31-3-250 ~]#
```



```
[root@ip-172-31-3-250 ~]# docker network connect backend frontend_cont1
[root@ip-172-31-3-250 ~]# docker network connect frontend backend_cont
[root@ip-172-31-3-250 ~]#
```



```
   "MacAddress": "",
   "Networks": {
       "backend": {
           "IPAMConfig": {},
           "Links": null,
           "Aliases": [
               "58c3162dbc4f"
           ],
           "MacAddress": "02:42:ac:13:00:04",
           "NetworkID": "ba35ffc5478f6cc0ad99fb72e66867a23ecc91246bff5f34bf8e106f3ae2a357",
           "EndpointID": "5d7ada02e8ac9a2e65aacdb8b2589d8db0de7bbb37d0ff32590f9928266cadba",
           "Gateway": "172.19.0.1",
           "IPAddress": "172.19.0.4",
           "IPPrefixLen": 16,
           "IPv6Gateway": "",
           "GlobalIPv6Address": "",
           "GlobalIPv6PrefixLen": 0,
           "DriverOpts": {},
           "DNSNames": [
               "frontend_cont1",
               "58c3162dbc4f"
           ]
       },
       "frontend1": {
           "IPAMConfig": null,
           "Links": null,
           "Aliases": [
               "58c3162dbc4f"
           ],
           "MacAddress": "02:42:ac:14:00:02",
           "NetworkID": "eeb4d680d3fe5f934468346ca8251ec86ee9d6da4514b1bc89df5fdf147f1557",
           "EndpointID": "0634a2875c430fd1be2a6de5376d1707af3c61f7db5de969229226ca4de155e0",
           "Gateway": "172.20.0.1",
           "IPAddress": "172.20.0.2",
           "IPPrefixLen": 16,
```



```
]
[root@ip-172-31-7-92 ~]# docker exec -it fd_cont bash
root@0614772e0bcd:/# ping 172.19.0.3
PING 172.19.0.3 (172.19.0.3) 56(84) bytes of data.
64 bytes from 172.19.0.3: icmp_seq=1 ttl=127 time=0.070 ms
64 bytes from 172.19.0.3: icmp_seq=2 ttl=127 time=0.049 ms
64 bytes from 172.19.0.3: icmp_seq=3 ttl=127 time=0.046 ms
```