- **FROM** (used for base image)

- **RUN** (to run command)

- We can RUN in two format

1.SHELL // RUN apt update// -->> RUN CMD <options> <options>

2.EXEC // EXEC ["apt","update"] -->

**EXEC ["CMD","<options","<options>"]**

examples :

**RUN apt update**

**RUN apt install -y git curl sudo**

**EXEC ["ls", "-l", "-r", "-t"]**

­

'''

***Docker prefers EXEC format, but industry we write in SHELL format***

'''

- **COPY** (used to copy files from local to image)

Note: we need to calculate from build path, not from system path

\* Only folders and file but not compressed or links

COPY <source\_path\_from\_build\_context> <destination\_inside\_image>

examples :

1. COPY test.txt /home/test.txt

2. COPY ./demo/file1.txt /home/demo/file1.txt or /home/demo/file\_to\_container.txt

to check this alway build image first and run container

- docker build -t myub:latest .

- docker run -it -d --name mycon myub:latest

- docker exec -it mycon /bin/bash

boot etc lib media opt root sbin sys usr

root@0a5f106e3033:/# cd home

root@0a5f106e3033:/home# ls

hack.txt hack\_test\_renamed.txt ubuntu

root@0a5f106e3033:/home# read escape sequence

- **ADD** (used to copy files from local to image but)

ADD supports extra source formats

\*If the source is a compressed file then ADD will automatically uncompressed it in the destination.

\*If the source is a downloadable link then ADD will automatically download the file in the destination.

example: -

**ADD https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.93/bin/apache-tomcat-9.0.93-deployer.tar.gz /home/**

- **CMD vs ENTRYPOINT**

\* Both CMD and ENTRYPOINT are used to define the default execution command of the container (the command which will be executed in the container

as main process).

\* If we use multiple CMD or ENTRYPOINT in the same Dockerfile only the last one will be considered.

\* If we use both CMD and ENTRYPOINT in the same Dockerfile, then ENTRYPOINT gets the highest priority and the command defined using CMD will be

as parameters to ENTRYPOINT.

**Difference**

- CMD can be completely overridden at the runtime (with docker run at the end

we can provide the command to override the CMD).

- ENTRYPOINT can't be overridden at the runtime but the command passed at the runtime will become parameters to ENTRYPOINT command defined in Dockerfile.

Syntax: we can define command in 2 ways

1. shell format

**CMD "ls -lrt"**

2. EXEC format

Always first element is command.

Except first element all the other elements are parameters

to command.

**CMD ["ls","-lrt"]**

**Overriding ENTRYPOINT :**

**docker run -it --name my-con --entrypoint echo my-image:latest daya**

**ENV**

* This instruction is used to set the environment variable inside the container.
* Using this instruction we can create ENV variables at build time which means in the docker images
* ENV variables should be in capital letters
  + - * EVN TEST test\_value or
      * ENV TEST=test\_value1

**ENV F\_NAME="daya"**

**ENV L\_NAME="shankar"**

**1. For individual variable**

**ENV <variable\_name> <value>**

**(OR)**

**ENV <variable\_name>=<value>**

**2. For multiple variable**

**ENV <variable\_name1>=<value1> <variable\_name2>=<value2> .....**

**ENV F\_NAME="daya" L\_NAME="shankar" FF\_NAME="mm"**

To create environment variables at run time (means in containers)

1. With the docker run command

**docker run -e <variable\_name>=<value> -e <variable\_name>=<value>**

**docker run -it --name my-con -e demo=value1 -e demo2=value2 my-image:latest**

2. With a list of variables in a file (.env file)

**docker run --env-file <file\_path> ...**

**What to do if we have more than 10 ENV variables?**

We can use .env filve / sometime .config

With a list of variables in a file (.env file)

**docker run --env-file <file\_path> …**

**docker run -it --rm --name my-con --env-file .env my-image:latest**

­

**WORKDIR**

This is used to set the working directory for all the instructions that follows it. Such as RUN, CMD, ENTRYPOINT, COPY, ADD .…

**ex: WORKDIR <path\_in\_container>**

**WORKDIR /home/daya/**

**ARG**

Using this instruction we can pass parameters to Dockerfile as user inputs.

**ex: ARG <arg\_variable\_name>=<value>**

Note: <value> acts as default value to the arg\_variable means if user does not set the arg value at build time this value will be used.

To pass the value at build time

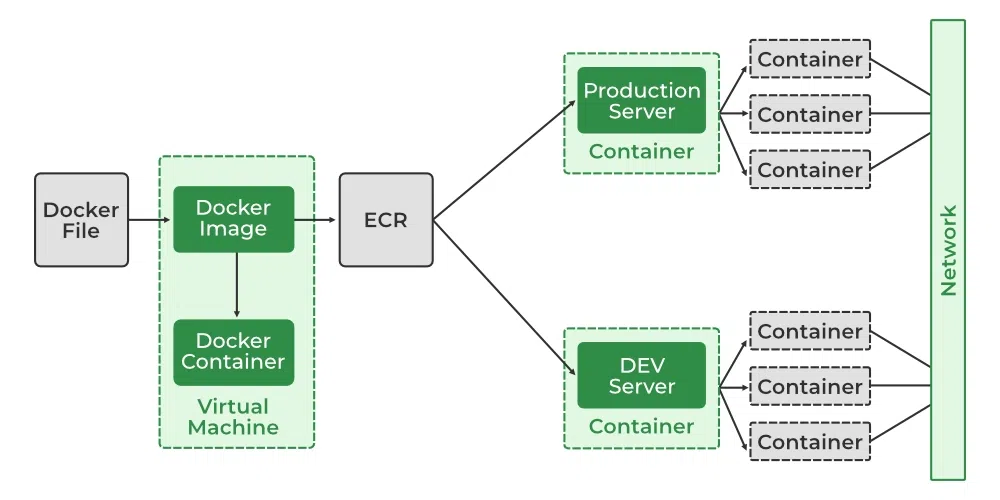
**docker build --build-arg <arg\_variable\_name>=<user\_value>**

**docker build -t myub:latest --build-arg IN=/home .**

**docker build -t myub:latest --build-arg WORK\_DIR=/home –build-arg DIR\_NAME=test\_arg .**

Docker Networks

The Docker network is a virtual network created by Docker to enable communication between Docker containers. If two containers are running on the same host they can communicate with each other without the need for ports to be exposed to the host machine.



### Network Drivers

There are several default network drivers available in Docker and some can be installed with the help of plugins, Command to see the list of containers in Docker mentioned below.

**docker network ls**

**ghost@ghost-pc:~$ docker network ls**

**NETWORK ID NAME DRIVER SCOPE**

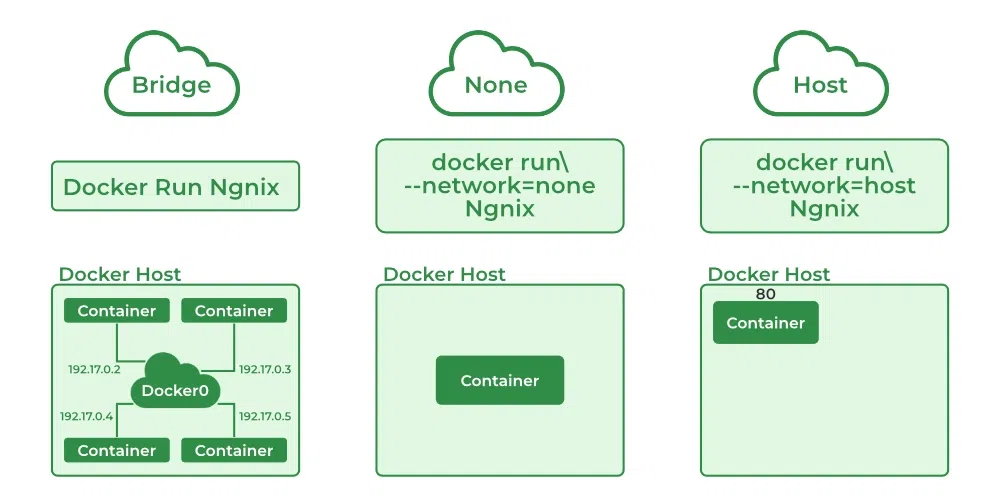
**5f3cdf4f141f bridge bridge local**

**dda80d7a9ee5 host host local**

**56c8c7f89b35 none null local**

### Types of Network Drivers

1. **bridge:** If you build a container without specifying the kind of driver, the container will only be created in the bridge network, which is the default network.
2. **host:** Containers will not have any IP address they will be directly created in the system network which will remove isolation between the docker host and containers.
3. **none:** IP addresses won’t be assigned to containers. These containers are not accessible to us from the outside or from any other container.
4. **overlay:** overlay network will enable the connection between multiple Docker demons and make different Docker swarm services communicate with each other.
5. **ipvlan:** Users have complete control over both IPv4 and IPv6 addressing by using the IPvlan driver.
6. **macvlan:** macvlan driver makes it possible to assign MAC addresses to a container.



### Launch a Container on the Default Network

1. **Understanding the Docker Network Command**

The Docker Network command is the main command that would allow you to create, manage, and configure your Docker Network. Let’s see what the sub-commands can be used with the Docker Network command.

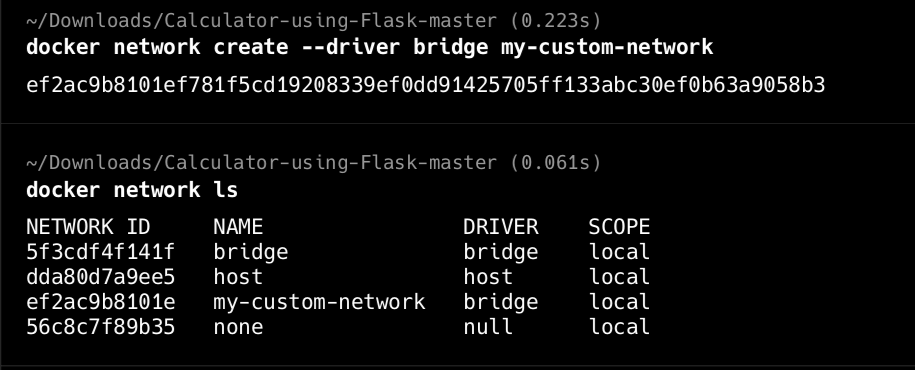
**docker network**

### 2. **Using Docker Network Create command / custom bridge**

With the help of the “Create” command, we can create our own docker network and can deploy our containers in it.

**sudo docker network create --driver <driver-name> <bridge-name>**

**docker network create --driver bridge my-custom-network**



3. **Using the Docker Network Connect command**

Using the “Connect” command, you can connect a running Docker Container to an existing Network.

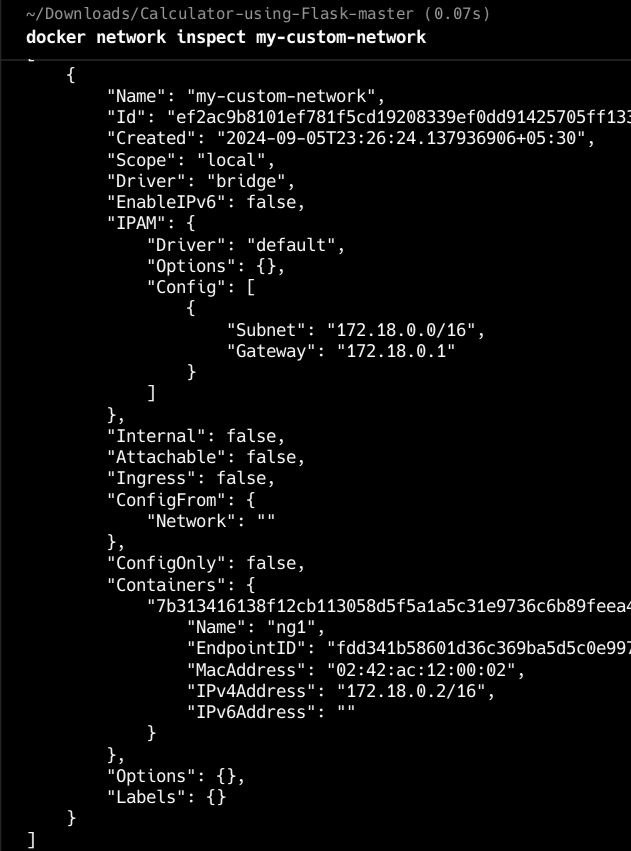
**sudo docker network connect <network-name> <container-name or id>**

**docker network connect my-custom-network ng1**

**4. Using the Docker Network Inspect command**

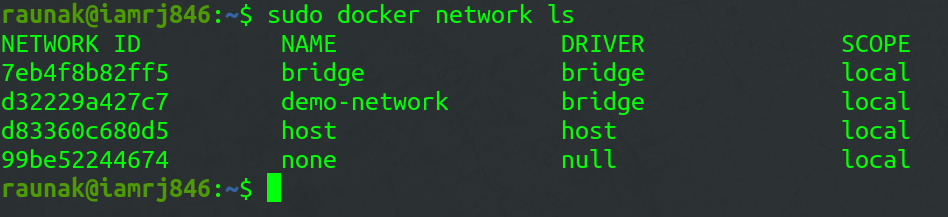
Using the Network Inspect command, you can find out the details of a Docker Network.

**sudo docker network inspect <network-name>**

  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
5. **Using the Docker Network ls command**

To list all the Docker Networks, you can use the list command.

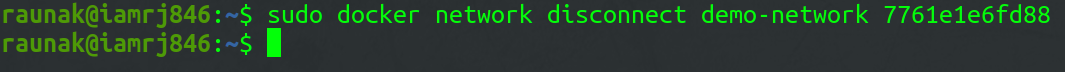
**sudo docker network ls**



6. **Using the Docker Network Disconnect command**

The disconnect command can be used to remove a Container from the Network.

**sudo docker network disconnect <network-name> <container-name>**

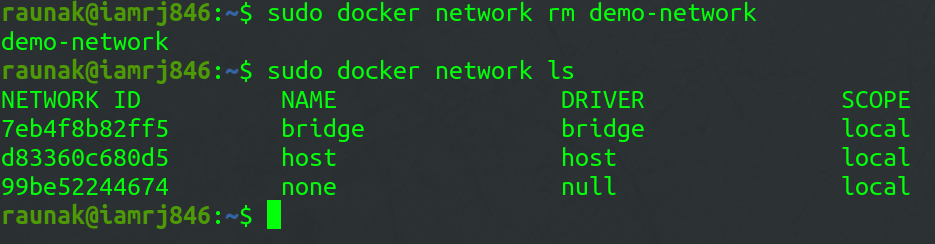


7. **Using the Docker Network rm command**

You can remove a Docker Network using the rm command.

**sudo docker network rm <network-name>**

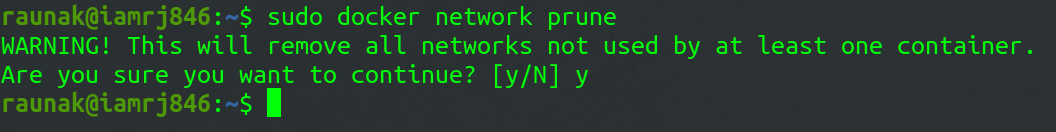
Note that if you want to remove a network, you need to make sure that no container is currently referencing the network.



8. **Using the Docker Network prune command**

To remove all the unused Docker Networks, you can use the prune command.

**sudo docker network prune**

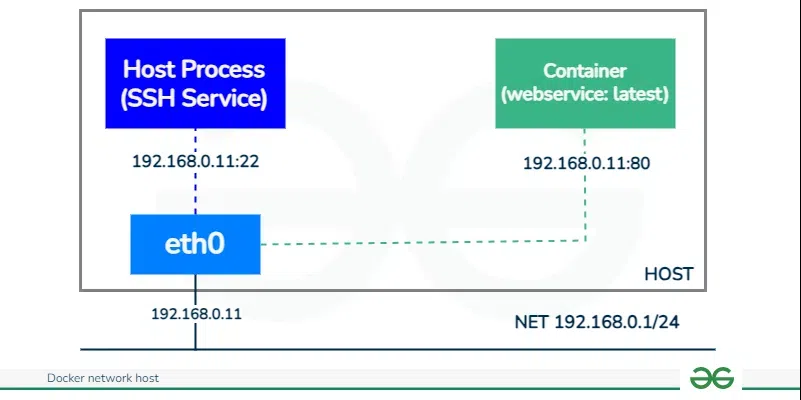


### Common Operations

1. **docker network inspects:**  We may examine the configuration information of a specific network, such as the name of the network, the containers that have linked to this network, the type of driver used to construct this network, and other characteristics, by using the “docker network inspect” command.
2. **docker network ls:** We can see all of the networks that are available on the current host by using “docker network ls”.
3. **docker network creates:** Using the command “docker network create” and the name of the driver, such as bridge, overlay, or macvlan, we can establish a new network.
4. **docker network connects:** In order to use this command, we must first confirm that the appropriate network has already been formed on the host. Then, using docker “network connect”, we may attach the container to the necessary network.

### What is Docker Network Host?

A container which shares its network namespace with the Docker host machine runs in the Docker network host, also known as Docker host networking. In this option, the container utilizes the network interfaces, IP addresses, and ports of the Docker host directly rather than having its own isolated network stack. Due to this, the container acts as if it were running directly on the host computer, facilitating easy access to host resources and services without the need for network address translation (NAT) or port mapping. Applications that need to interact closely with services running on the Docker host or require high-performance networking usually utilize this networking alternative.

  
  
  
  
  
  
  
  
  
  
  
  
  
  
**Host network driver**

Below is an example Docker command to run a container in host networking mode:

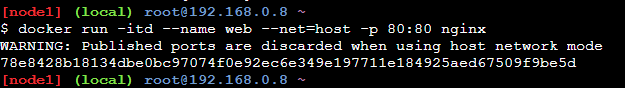
**docker run -itd --name <ContainerName> --network=host -p <HostPort>:<ContainerPort> <image-name>**

**docker run:** A Docker container is run in this command.

**--network host**: By selecting this option, the container will not create its own different network namespace; instead, it’s going to use the host’s network stack.

**-p: HostPort** is the port on the host machine to which you want to map the container’s port. **ContainerPort** the port inside the container that you want to expose.

**nginx:** Its Docker image containing your web server application is known as this.

  
  
**What are the Use Cases of Docker Network Host?**

Why would you use Docker host networking mode? A Docker network host can provide performance improvements and better performance over other Docker network options, e.g., “none” and “bridge” options. Additionally, Docker host networking does not require network address translation (NAT), making it easy to use multiple ports at the same time. However, users working in Docker host networking mode should be careful to avoid port conflicts.

**- Simplified Networking:** It allows Docker containers to directly use the network settings of the Docker host, without any network isolation. This makes networking configuration simpler and more transparent.

**- Performance Optimization:** Containers running in host network mode can achieve better network performance because they bypass the additional overhead of Docker’s network virtualization.

**- Access to Host Services:** Containers can easily access services running on the Docker host without needing to publish or expose ports explicitly. This simplifies communication between containers and host services.

## Docker Network Host vs Bridge

| **Feature** | **Host Network** | **Bridge Network** |
| --- | --- | --- |
| Network Namespace | Shares network namespace with Docker host | Each container has its own network namespace |
| IP Addressing | Uses host’s IP address(es) | Containers have unique IP addresses within bridge network |
| Network Performance | Higher, as there’s no overhead from NAT or routing through a bridge | Slightly lower due to NAT and routing through bridge |
| Port Conflicts | Possible if multiple containers bind to the same port | Avoided, each container has its own isolated network stack |
| Use Cases | Applications requiring high network performance, or need direct access to host’s network interfaces | Most common choice for running multiple containers on a single host, offers network isolation and avoids port conflicts |
| Default | Not the default, must be explicitly specified | Default network mode in Docker |

### Docker Network Host – FAQs

**How does host networking impact container isolation?**

Host networking mode reduces the network separation between the container and the host. Unlike a bridge network, where containers have their own network, containers using a host network share the same network namespace as the host. This may mean safety and may not be suitable for sites that require strict separation between containers.

**Can multiple containers use host networking simultaneously?**

No, any container host using host networking mode has direct access to the network stack, and port conflicts can occur if multiple containers try to bind to the same port on the host Host networking ensures no port conflicts does not exist when running containers Let’s do it.

**Are there any specific security considerations when using host networking mode?**

Using host networking mode can expose packing containers to security risks, on the grounds that they proportion the internet namespace with the host.

It’s vital to ensure that a containerized utility is secure, observe high-quality practices for box protection, and pay near attention to whether or not host networking is necessary on your use case.

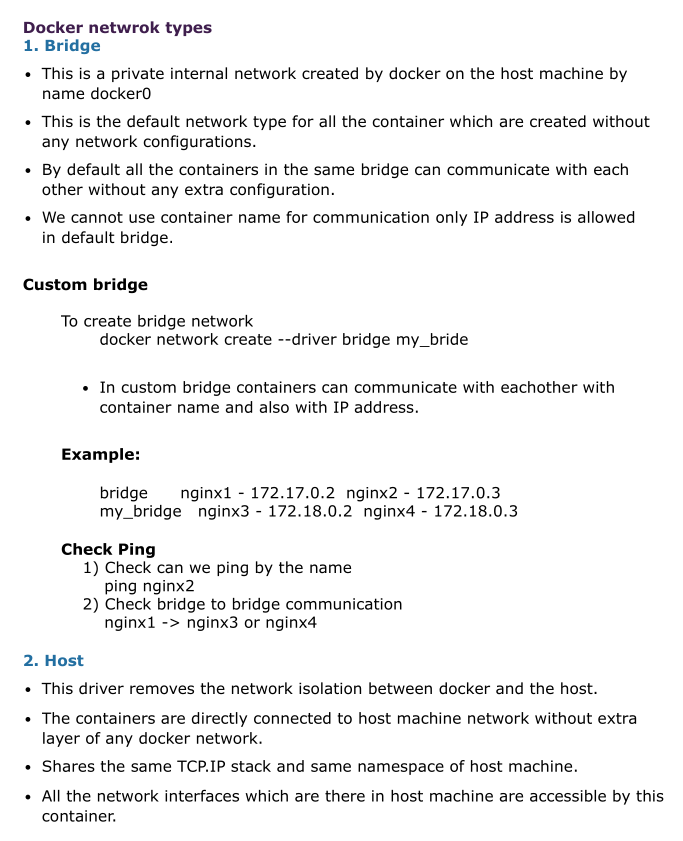
**Can I use Docker Compose with host networking?**

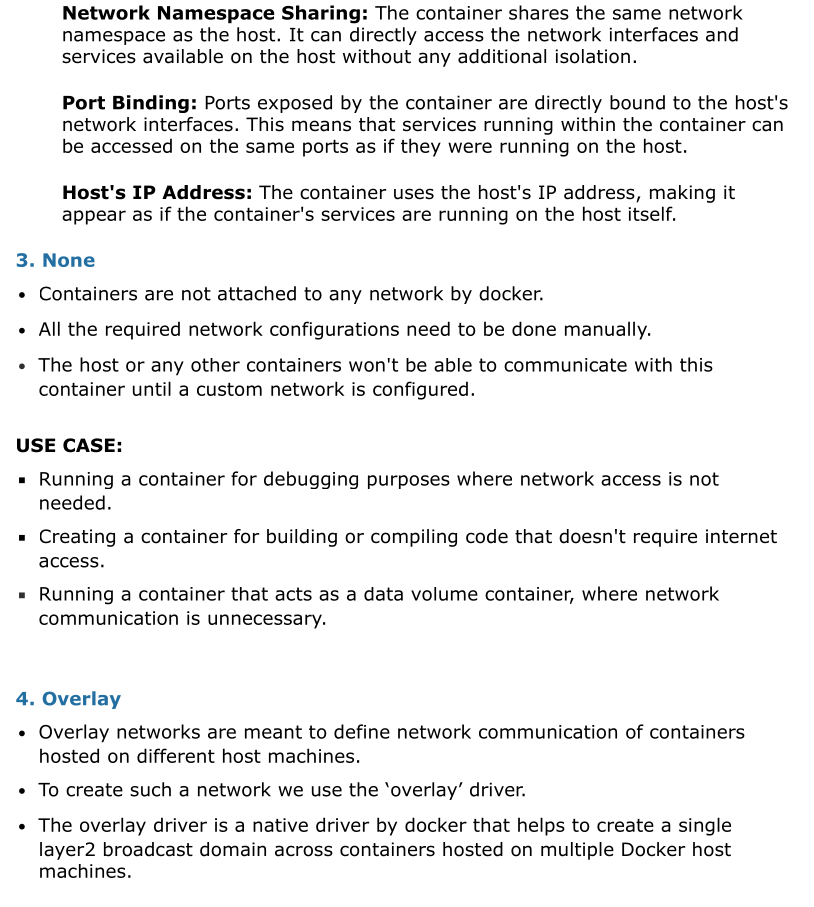
Docker Compose does not directly support the –network host option. If you want to use the host network with Docker Compose, you can configure the services to use the host network in the docker-compose.yml file.

However, keep in mind that not all features of Docker Compose can work well with host networking, and experimentation is recommended.

**What is the host network IP address of Docker container?**

The host network IP address of a Docker container is the same as the IP address of the Docker host itself, as the container shares the network namespace with the host. It uses the host’s network stack directly.





**How to have communication b/w containers which are in 2 different bridge network. How to**

**create image from a container ? What is docker save, load, export and import ?**

# ----------- Docker Volumes -----------

### What Is Docker Volume?

Docker containers enable apps to execute in an isolated environment. All modifications made inside the container are lost by default when it ends. Docker volumes and bind mounts can be useful for storing data in between runs. One way to store data outside of containers is with volumes. All volumes are kept in a specific directory on your host, typically /var/lib/docker/volumes for Linux systems, and are controlled by Docker.

### What is the Docker File System?

A Docker container executes the software stack specified in a Docker image. Images are built up of read-only layers that operate on the Union File System. When we start a new container, Docker adds a read-write layer on top of the image layers, allowing the container to function like a conventional Linux file system. So, each file modification within the container generates a functioning copy in the read-write layer. However, when the container is stopped or removed, the read-write layer disappears.

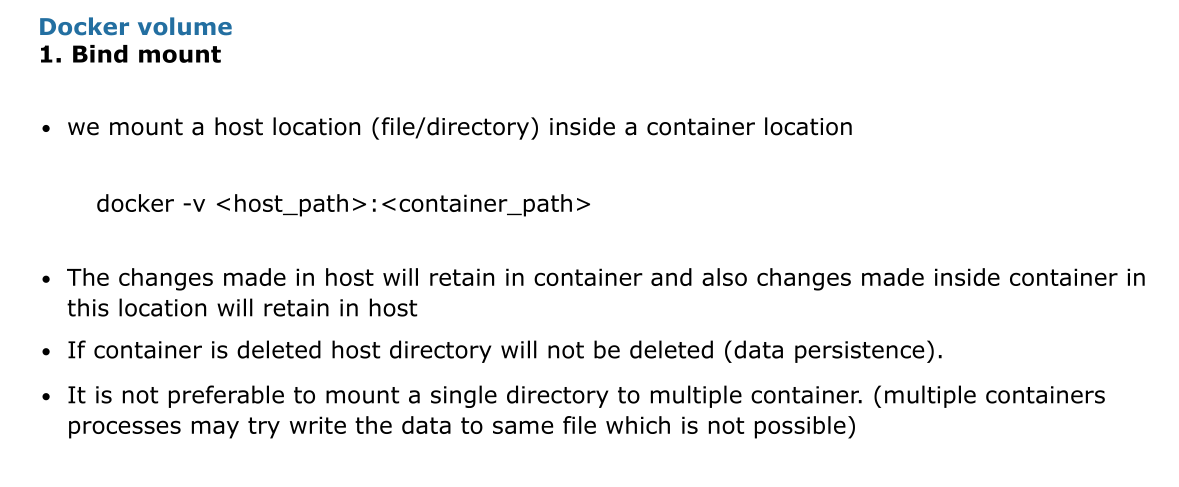
### Types Of Mounts in Docker

The data appears the same from within the container in all mount modes. In the filesystem of the container, it is shown as a directory or a single file.

* **Volumes:** Docker manages volumes kept in a section of the host filesystem (/var/lib/docker/volumes on Linux). This portion of the filesystem shouldn’t be altered by non-Docker processes. In Docker, volumes are the most effective way to store data. Using the docker volume create command, we may directly create a volume, or Docker can do it for us when it creates a container or service.
* **Named Pipes:** To facilitate communication between a container and the Docker host, a named pipe mount can be employed. Using a named pipe to connect to the Docker Engine API while running a third-party program inside a container is the typical use case.
* **Bind Mounts:** On the host system, bind mounts can be kept anywhere. These might be crucial system folders or files. They are always modifiable by non-Docker processes running on a Docker host or in a Docker container. Comparatively speaking, bind mounts are less useful than volumes.
* Tmpfs Mounts: These mounts are never written to the host system’s filesystem; instead, they are kept solely in the memory of the host system. Neither on the Docker host nor in a container is it stored on a disc. Sensitive or non-persistent state data can be stored on the tmpfs mount for the duration of the container.

**Bind Mount**

**$ docker run -v $(pwd):/var/opt/project –name ng1 nginx:latest**



## 2. Docker Volumes

* The are docker managed filesystem and we use docker commands to manage these volumes.
* Volumes are easier to manage, backup and migrate than bind mounts.
* We can use different source filesystem called storage drivers (EBS, EFS, s3)
* Default location of docker volume is /var/lib/docker/volumes/<volume\_name>

To create a docker volume

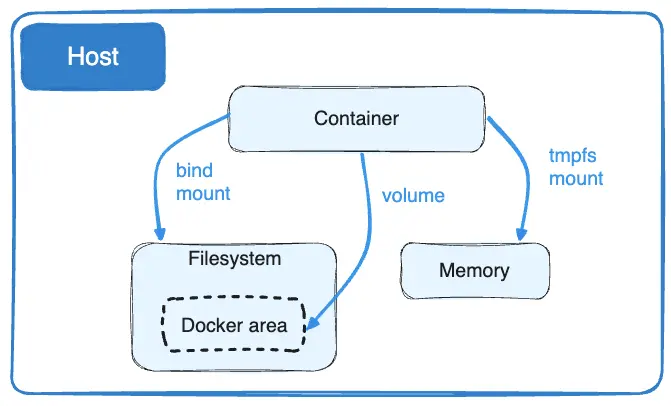
**docker volume create <volume\_name>**

To delete volume

**docker volume rm <volume\_name>**

To mount a volume

**docker -v <volume\_name>:<container\_path>**

Volumes are stored in a part of the host filesystem which is managed by

Docker (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.

* Bind mounts may be stored anywhere on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
* tmpfs mounts are stored in the host system's memory only, and are never written to the host system's filesystem.

## **[Good use cases for volumes](https://docs.docker.com/engine/storage/" \l "good-use-cases-for-volumes)**

Volumes are the preferred way to persist data in Docker containers and services. Some use cases for volumes include:

* Sharing data among multiple running containers. If you don't explicitly create it, a volume is created the first time it is mounted into a container. When that container stops or is removed, the volume still exists. Multiple containers can mount the same volume simultaneously, either read-write or read-only. Volumes are only removed when you explicitly remove them.
* When the Docker host is not guaranteed to have a given directory or file structure. Volumes help you decouple the configuration of the Docker host from the container runtime.
* When you want to store your container's data on a remote host or a cloud provider, rather than locally.
* When you need to back up, restore, or migrate data from one Docker host to another, volumes are a better choice. You can stop containers using the volume, then back up the volume's directory (such as /var/lib/docker/volumes/<volume-name>).
* When your application requires high-performance I/O on Docker Desktop. Volumes are stored in the Linux VM rather than the host, which means that the reads and writes have much lower latency and higher throughput.
* When your application requires fully native file system behavior on Docker Desktop. For example, a database engine requires precise control over disk flushing to guarantee transaction durability. Volumes are stored in the Linux VM and can make these guarantees, whereas bind mounts are remoted to macOS or Windows, where the file systems behave slightly differently.

[Good use cases for bind mounts](https://docs.docker.com/engine/storage/" \l "good-use-cases-for-bind-mounts)

In general, you should use volumes where possible. Bind mounts are appropriate for the following types of use case:

* Sharing configuration files from the host machine to containers. This is how Docker provides DNS resolution to containers by default, by mounting /etc/resolv.conf from the host machine into each container.
* Sharing source code or build artifacts between a development environment on the Docker host and a container. For instance, you may mount a Maven target/ directory into a container, and each time you build the Maven project on the Docker host, the container gets access to the rebuilt artifacts.

If you use Docker for development this way, your production Dockerfile would copy the production-ready artifacts directly into the image, rather than relying on a bind mount.

* When the file or directory structure of the Docker host is guaranteed to be consistent with the bind mounts the containers require.

[Good use cases for tmpfs mounts](https://docs.docker.com/engine/storage/" \l "good-use-cases-for-tmpfs-mounts)

tmpfs mounts are best used for cases when you do not want the data to persist either on the host machine or within the container. This may be for security reasons or to protect the performance of the container when your application needs to write a large volume of non-persistent state data.

## Benefits of Docker volume over Bind mounts

* Volumes are easier to back up or migrate than bind mounts.
* You can manage volumes using Docker CLI commands or the Docker API.
* Volumes can be more safely shared among multiple containers.
* Volume drivers let you store volumes on remote hosts or cloud provider storage solutions,
* encrypt the contents of volumes, or add other functionality.
* Volumes work on both Linux and Windows containers.
* New volumes can have their content pre-populated by a container.
* Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts

### How to optimise the docker build process?

### How to reduce the size of the docker image or container?

**1. Minimize layers:** Each instruction in a Dockerfile creates a layer. Fewer layers

mean smaller image sizes and faster builds. Combine multiple commands where

possible with RUN instruction, using &&, to reduce the number of layers.

**2. Use efficient base images:** Choose base images that are small and well-

maintained. slim/alpine/scratch is a popular choice due to its small size.

**3. Remove unnecessary dependencies and files:** Clean up after each step to

reduce the size of the final image. Use multi-stage builds to compile binaries or

install dependencies in one stage and copy only necessary files to the final stage.

**4. Optimize caching (with proper sequence of instrucitons):** Arrange your

Dockerfile instructions to maximize the benefit of Docker's caching mechanism.

Place frequently changing instructions towards the end of the Dockerfile.

Always COPY the source code to build after all dependencies are installed



The COPY . . instruction appears after the package management instructions, so

the builder can reuse the RUN go mod download layer.

5. Use .dockerignore: Create a .dockerignore file to exclude unnecessary files and directories from being copied into the Docker image. This helps reduce the build context size and speeds up the build process.

# 6. Multistage build

There are 2 problems with the normal build process

1. Size: the challenge is to keep the image and its container size as minimal as possible.

2. The larger the surface area more the application is vulnerable to attacks.

* Multistage build allows us to define multiple FROMs in the same Dockerfile.
* Dependency between multiple FROMs is maintained by naming **FROM using AS keyword** or we can use index starting from 0 and we can refer to this name in another FROM.

**FROM <base\_image> AS <STAGE\_NAME>**

* Only the final FROM image is created leaving back all the other FROM images/stages.
* Copy only the required files from the named FROM stage like below.

FROM final\_build

COPY --from=<STAGE\_NAME> <src\_named\_stage> <dest>

(OR)

FROM final\_build

COPY --from=0 <src\_named\_stage> <dest>

**FROM golang:1.23 AS build\_stage**

**WORKDIR /src/**

**COPY ./main.go ./main.go**

**RUN go build -o /bin/hello ./main.go**

**FROM scratch**

**COPY --from=build\_stage /bin/hello /bin/hello**

**CMD ["/bin/hello"]**

## Docker – EXPOSE Instruction

The EXPOSE instruction exposes a particular port with a specified protocol inside a Docker Container. In the simplest terms, the EXPOSE instruction tells Docker to get all the information required during the runtime from a specified port. These ports can be either TCP or UDP, but it’s TCP by default. It is also important to understand that the EXPOSE instruction only acts as an information platform (like Documentation) between the creator of the Docker image and the individual running the Container. Some points to be noted are:

* It can use TCP or UDP protocol to expose the port.
* The default protocol is TCP if no other protocol is specified.
* It does not map ports on the host machine.
* It can be overridden using the publish flag (-p) while starting a Container.

## What is the difference between “expose” and “publish” in Docker?

| **Feature** | **Expose** | **Publish** |
| --- | --- | --- |
| Purpose | Specifies ports exposed from the container to the container network | Maps container ports to specific host ports for external access |
| Usage | Exposes ports within the container network for inter-container communication | Maps container ports to host ports for external access |
| Scope | Limited to the container network | Allows access from outside the container network |
| Configuration | Defined in Dockerfile using EXPOSE directive | Defined at runtime with -p or –publish option |
| Visibility | Ports are visible to other containers in the same network | Ports can be accessed from the host machine and external systems |

## Publishing and EXPOSE Docker Ports

Follow the below steps to implement the EXPOSE instruction in a [**docker container**](https://www.geeksforgeeks.org/containerization-using-docker/):

### ****Step 1:****Exposing Ports

****Exposing Ports in Dockerfile:**** Ports can be exposed within a Dockerfile using the EXPOSE instruction. This instruction informs Docker that the container listens on the specified network ports at runtime. However, it does not actually publish the port

Let’s create a *Dockerfile*with two EXPOSE Instructions, one with TCP protocol and the other with UDP protocol.

Example:

FROM ubuntu:latest

EXPOSE 80/tcp

EXPOSE 80/udp

## EXPOSE

**Syntax: EXPOSE <port\_number>**

Used to expose a port to the docker network so that all the other containers

in the same docker network can access it.

Exposes the port within the host machine.

## Publish

Publish will bind the container application port to the host machine port so that

we can access from the outside world with the host machine port.

**Publish = port mapping of container to host machine + Expose**

To publish a port

**docker run -p <host\_port>:<container\_port>**

**To publish all the exposed ports**

**docker run -P**

-P publish\_all, It binds all the exposed ports of the container to the

host machine.

To map a direct IP address to the host

**port to port**

**docker run -p <ip>:<host\_port>:<container\_port>**

**Any to port**

**docker run -p <ip>::<container\_port>**

Range of ports

**many to many:** The total number of host ports in the range should be

the same as the container port range

**docker run -p 8080-8085:8086-8090**

**many to one:** This will map to any one of the host ports which is free

**docker run -p 8080-8090:8080**