**Docker**

**Containers vs VM –**

Understand the components in Hardware and Software –

A whiteboard with writing on it

AI-generated content may be incorrect.

Workflow – When we double click on let’s say .txt file then it will pass an information to OS and OS pass the request to HDD (where the file is stored). But how software components convert the request to hardware instructions? Here comes the **Kernal. Kernal converts request into hardware instructions. It acts as bridge between Hardware and Software.**

**VM vs Containers:**

A screenshot of a computer screen

AI-generated content may be incorrect.

Key pointers:

* Hypervisor – Understands resource requirements from VMs and provide the same from infrastructure. Acts as a core component between Infrastructure and VM.
* Each VM has its own OS and kernal hence it consumes a lot of resources from infrastructure.
* Container engines – eg. docker, containerd, crio
* Containers doesn’t have OS, they use majority of the resources from infrastructure.

**First things first –**

Before installing docker (only ethernet0 is present) –

A screenshot of a computer program

AI-generated content may be incorrect.

After installing docker (docker0 got added after docker installation) –

A screenshot of a computer screen

AI-generated content may be incorrect.

When containers want to access the internet, they will first send request to docker0 then docker0 will communicate with et0 (ethernet) and then et0 will pass the request to internet. Hence docker0 acts as a bridge network between containers and et0.

A black screen with white text

AI-generated content may be incorrect.

Docker Architecture –

Docker architecture is based on a client-server model.

A screenshot of a computer

AI-generated content may be incorrect.

Workflow – let’s say if we hit “docker run”

It will do a http api call to a docker daemon. Docker daemon will download the required image from registry and then container will be created out of the image.

**Docker daemon is a process.**

**Important docker commands -**

**Run container in detached mode and remove once it is stopped –**

**# docker run --rm -d --name frontend nginx echo "Hello, World!"**

This runs a nginx container, executes echo "Hello, World!", and removes the container immediately after it stops.

**Port forwarding –**

**# docker run --rm -d --name frontend -p 8000:80 nginx**

Here 80 is container port and 8000 is docker host’s port.

**# docker inspect, docker logs, docker ps, docker rmi.**

**Directory change –**

By default, docker stores all container logs and other data to /var/lib/docker. Which consumes root space. We need to avoid that by changing the directory. For this, we can create an ebs volume -> attach it to docker host (ec2) isnatnce -> create a new directory with the ebs storage. Change docker service file to use the new directory.

**Custom network in docker –**

Let’s say we have created 2 containers in default bridge network varies app1, app2. If we exec in app1 and try to ping IP of app2 then we will get response. But if we try to ping with container name (app2) then it will not work ->

A computer screen shot of a program

AI-generated content may be incorrect.

To create connectivity using container names, we need to create a custom network ->

A computer screen shot of a program

AI-generated content may be incorrect.

Now if we create 2 more containers varies app3 and app4 then they can communicate using containers name as well ->

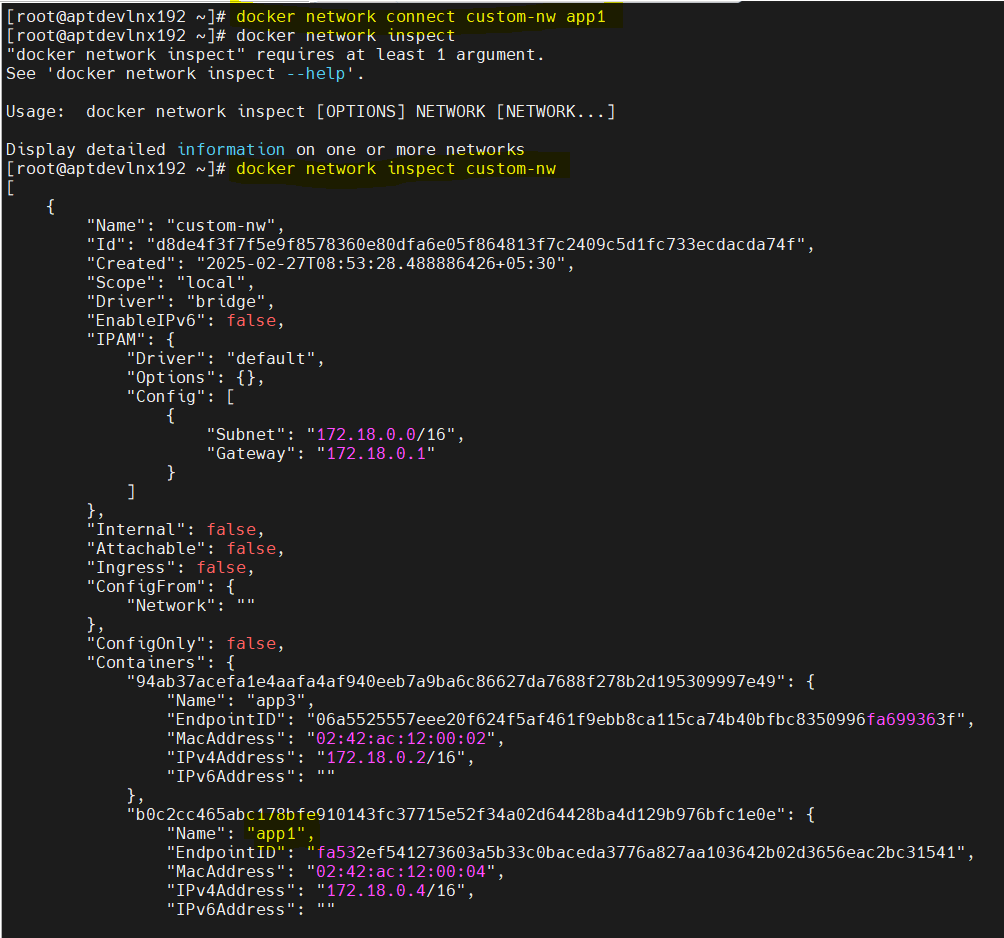
Create containers in custom network –

# **docker run --rm -d --name app4 --network custom-nw nginx**

A computer screen shot of a program

AI-generated content may be incorrect.

We can connect to containers in bridge network from custom network. To achieve this we need to connect custom-nw and bridge-nw.

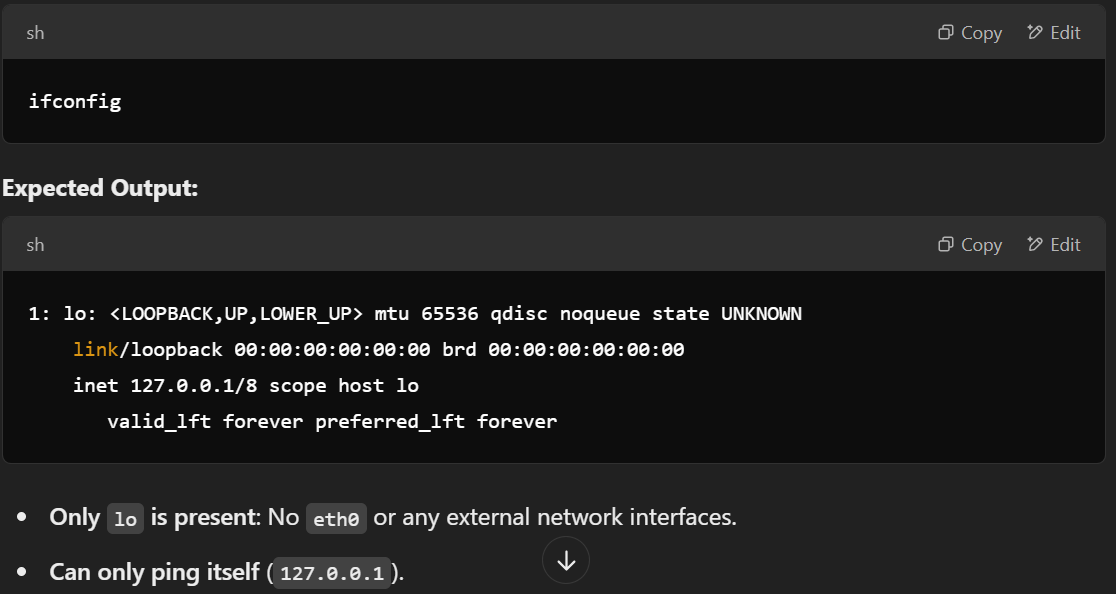


**None network in docker –**

The none network in Docker is a special type of network mode that completely isolates a container from any network communication. When a container is attached to the none network, it has no access to other containers, the host machine, or the internet.

**Key Characteristics:**

* No network interface except the **loopback (lo) i.e. 127.0.0.1**
* No access to external networks (including the host and other containers).
* Used for containers that **don’t need networking**, such as security-sensitive applications or data-processing tasks.

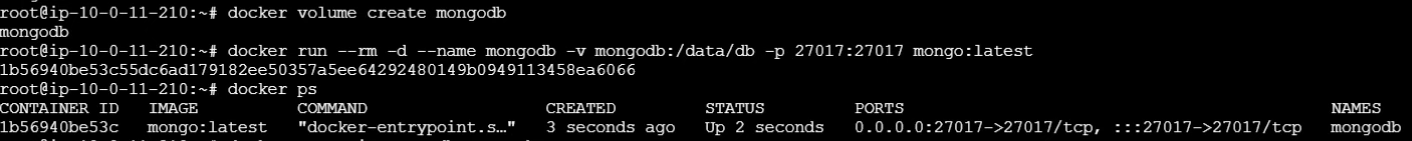


A screenshot of a computer

AI-generated content may be incorrect.

**Bind mounts and Volumes –**

Docker volume is used to persist a container’s data to a host machine.



Now, even if container is restarted, data will not be lost. We can find the data on host machine at location –

**/var/lib/docker/<volume\_name>/**

In case we want to directly bind the host directory to container directory we can use bind mounts. The command will be same as above only change will be instead of <volume\_name> (mongodb) we need to provide the directory path on host machine.

Volumes –

A black background with white text

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

Bind mount –

A black background with white text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Dockerfile, docker build –**

* It is not mandatory to keep file name as **Dockerfile**, we can keep any name and pass it while building the image with ‘**-f <file\_name>’**
* When we create an image then data will be stored in cache memory and if we create the same image again then cache memory will be used and image will be creted in less time. If we don’t want to use cache memory, we can pass the flag **‘--no-cache’**
* Env vs. ARG –

arg will be available only while building container and not inside the container.

env will be available inside the container as well.

* Use build-arg to pass ARG values to Dockerfile -

**# docker build -t <image\_name:tag> -f <Dockerfile\_location> --build-arg JAVA\_VERSION=18-jdk**

* Use flag **‘—progress=plain’** to see detailed output of docker build.
* Pass env variables –

**#** **docker run -e USER\_NAME=admin -e PASSWORD=pwd --name test -d nginx**

* **Dangling images** are Docker images that **exist but are not tagged or referenced by any container**.

**# docker images -f "dangling=true"**

* **docker run vs docker start** –

docker run – Creates and starts a **new** container from an image. If the container does not exist, it will create one from the specified image.

docker run = docker create + docker start

start – Starts an **existing** stopped container. Cannot create a new container; it only works on containers that were previously created.

* CMD vs ENTRYPOINT –

CMD can be **overridden** at runtime while ENTRYPOINT cannot be overridden.

When we override CMD vs When we override ENTRYPOINT -

A screenshot of a phone

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Generally, we use combination of both.

**EXPOSE vs -p (publish) in docker:**

The EXPOSE instruction in a Dockerfile is used to **document** which ports the container listens on at runtime. However, it **does not actually publish the port** to the host machine.

A black rectangular object with white text

AI-generated content may be incorrect.

How to Actually Expose the Port –

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a phone

AI-generated content may be incorrect.

* COPY vs ADD –

copy will work only for source and destination files present on host.

add has all functionalities as copy and additional functionality to download.

A black screen with red text

AI-generated content may be incorrect.

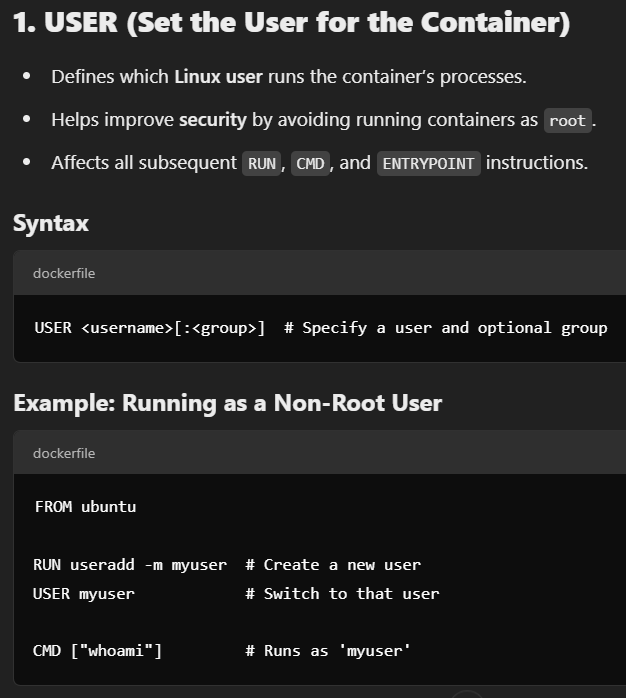
A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

* USER and WORKDIR -



A screenshot of a computer

AI-generated content may be incorrect.

A black and white text

AI-generated content may be incorrect.

**Chained WORKDIR commands –**

A screenshot of a computer

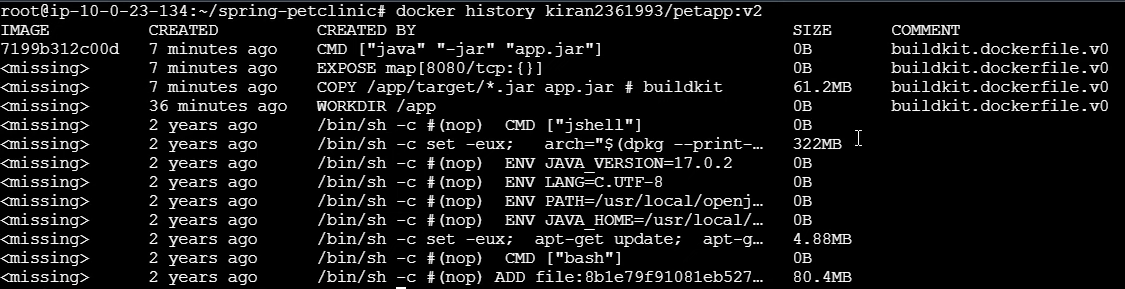
AI-generated content may be incorrect.

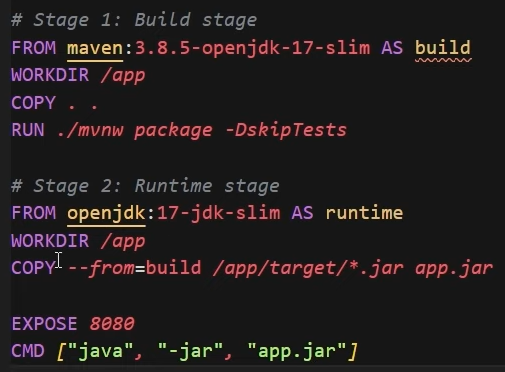
* Lesser the commands in Dockerfile lesser will be the size of the images.

**Multi-stage build –**

It is used to reduce the image size.

To check layerwise size –





Using **distroless** images as base image we can further reduce the image size.

Redhat provides **ubi** image which is an alternative to distroless images.