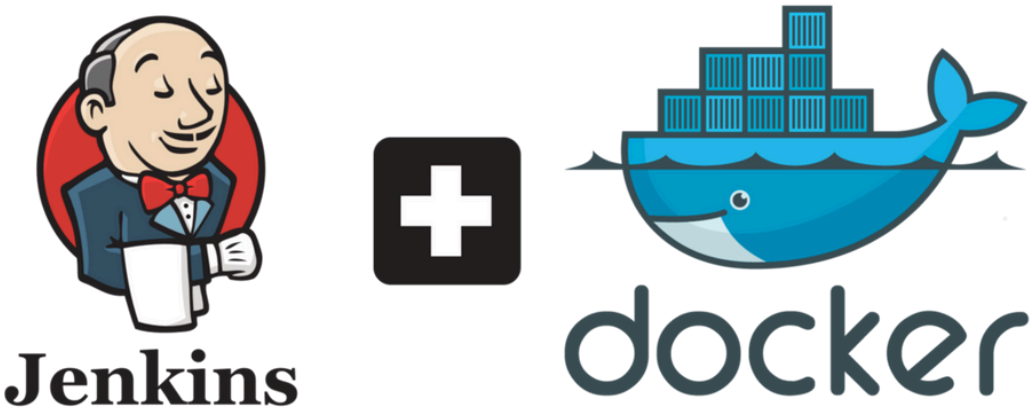


Docker Topics

- Docker Terminologies
- Writing Docker files, Building Images, Running containers
- Docker Volumes
- Docker Network
- Multi-stage Builds
- Docker compose

Jenkins Topics

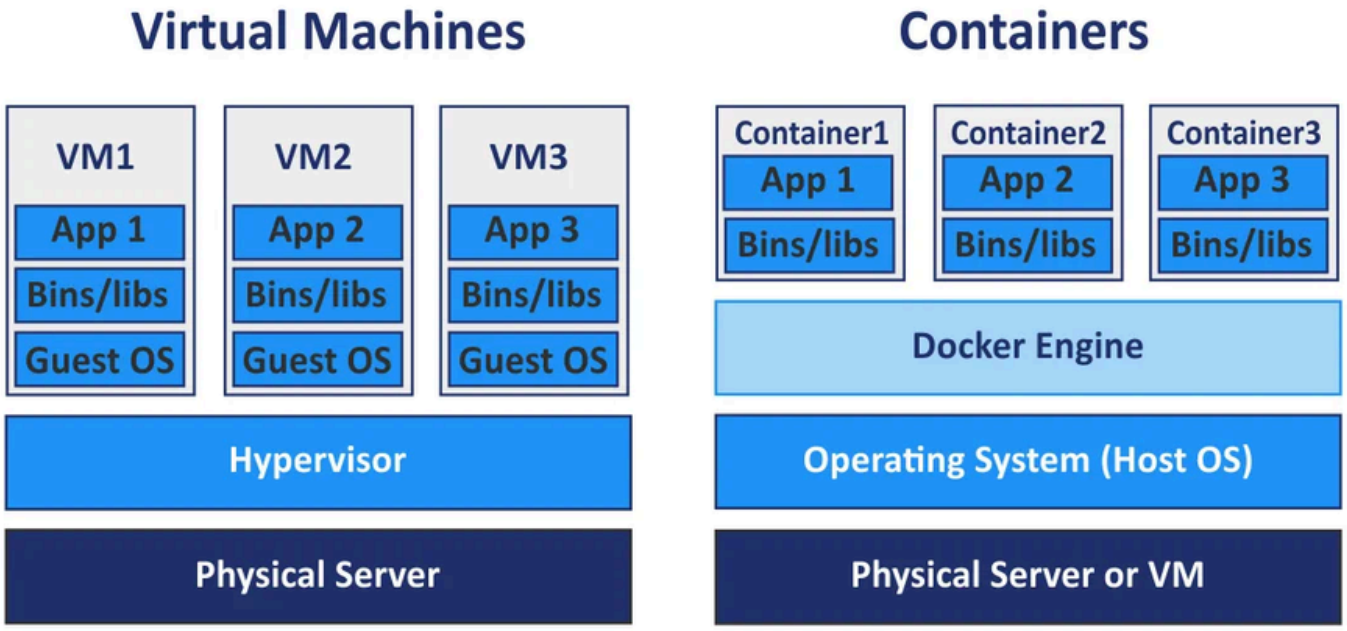
- Jenkins and CI/CD introduction
- Jenkins Installation
- Jenkins job creation
- Jenkins Agents

- by Nikitha Jain

Docker:

Docker is an '***open-source tool***' that facilitates deployment of *applications* inside the software *containers*.

- Docker is a tool is a **Platform as a service** (PAAS) that performs **O.S level Virtualization** whereas VMware uses Hardware level Virtualization.
- Docker uses the Host O.S to run applications. It allows applications to use the same Linux kernel as a system on the host computer, rather than creating a whole virtual O.S.
- We can Install docker on any O.S but Docker engine runs natively on Linux distribution.
- Docker is written in "**go**" Language.



Let's consider a scenario,

Developers use to work on application or code and send it to tester for testing the code. But due to incompatibility in the versions of its dependency in code, tester use to face issues while testing.. Now what do you think the tester should do? He need to again create the same environment just as the developer to make sure that the code is working and reliable.

So this problem is resolved by Virtualization.

Virtualization:

- Virtualization, here the name indicates that what we are doing is Virtualizing the physical resources we have.
- It's basically transforming the Hardware into Software.
- In other words, its just turning the physical resources into logical.
- Since its resources are virtualized, we can add and split them according the requirement we have.
- We have a Layer that makes this to happen, that makes resources to virtualize, guess what it could be? yes, its hypervisor.

- Hypervisor is a layer that turns physical resources to virtual ones, its allocates the hardware.
- We create virtual machines that carries all dependencies so that as we discussed the tester can easily test the code of developer.
- Hardware includes CPU, Memory, Storage and Networking.
- Isn't it fantastic then why are we using Virtualization.

Virtualization Disadvantages:

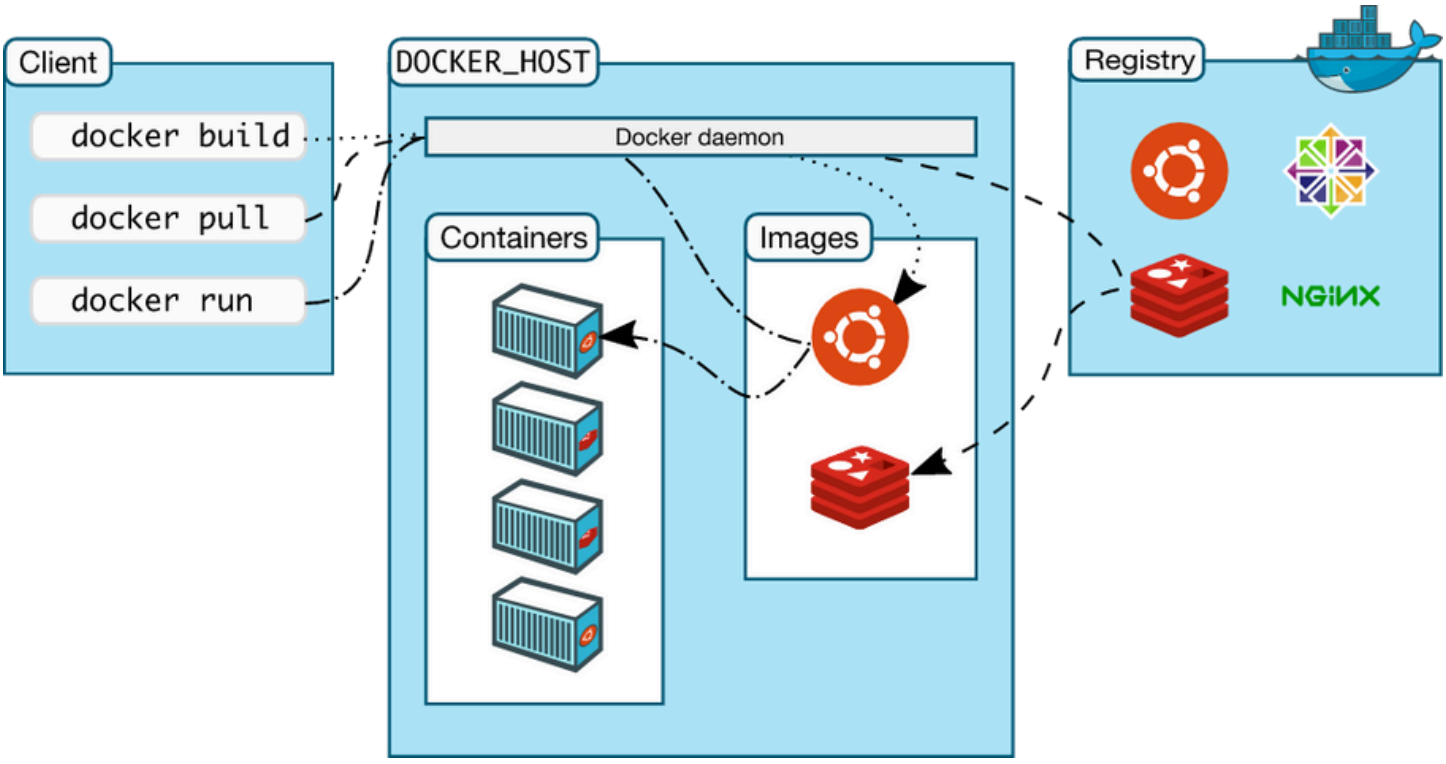
- Virtual Machine has to be allocated with a specific space before its creation.
- When the virtual machine is in use, we cannot allocate more space if it is needed or decrease the space allocated to it if it is not needed.
- So here wastage of resources started happening.
- Limitation of space sets a main draw back as we can scale it.
- Also the other issue is due to space limitations, we cannot more VMs.

Docker Advantages:

- No pre-Allocation of RAM
- Light in weight
- Scalable
- Low cost
- Consumes less time to create containers.
- You can reuse the image.

You can see the Docker Architecture and its Components below:

Docker Architecture and its Components:



Docker Client:

- Docker client interacts with the Docker Daemon. Docker client is a CLI terminal that we write all our commands that interacts with Docker daemon and sends all the commands to it.
- It's also possible for the Docker Daemon to connect with more than one Daemon.

Docker Daemon:

- Docker Daemon runs on the Docker host as you can see from the image of Docker Architecture.
- Docker Daemon communicates with other daemon.
- It is responsible for running the containers.

Docker Host:

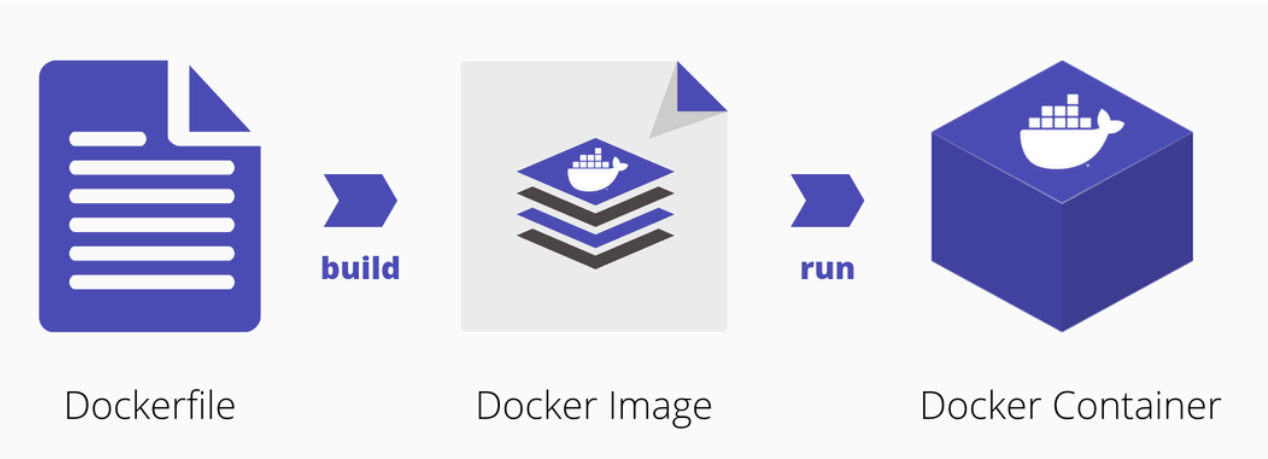
- Docker Host includes Docker Daemon, Docker images, Docker Containers.
- So its a server that holds all things and we have client that's a terminal where we write all the commands.
- It provides a environment that runs and executes the container.

Docker Registry:

Manages & stores the Docker images.

Docker registry has 2 types.

1. Public registry : Docker hub is a place where we have all the Docker images over the internet and is easily accessible.
2. Private registry: Private registry is not accessible by everyone over the Internet. It is specially for the Enterprise only few people will have access to it.



Docker file: Its a set of Instructions written to create the Docker Image.

Docker Image:

- Its a template used to create the Docker container.

- Docker Image is reusable and we push this image to the Docker hub and also pull the Image from the Docker hub.
- Docker image is read-only binary template.

Docker container:

- If the Image is in running state, then it becomes the Docker container. It is light weight and easy to create, manage and destroy.
- Docker container should be only used with non-root user, if it is used by the root user, then it has high level permissions and if accidently deletes then it will be deleted as it has full permissions.
- Non-root user will have only permissions that will ensure the more security in the container.
- Hence when non-user is logged in then the security is improved.
- Root user will be dangerous while log in into the container.

Docker installation:

Let's launch an AWS EC2 instance,

Name the server, consider ubuntu as the AMI image and 8gb storage and create new key pair.

connect it with ssh,

→ sudo apt-get update

→ sudo apt-get install docker.io

→ sudo apt-get install docker-compose-v2

→ docker --version

→ sudo usermod -aG docker $USER

→ newgrp docker

→ systemctl status docker

if it is not active running then do,

→ systemctl start docker

→ systemctl enable docker

→ systemctl status docker

→ docker ps

```
ubuntu@ip-172-31-19-80:~$ sudo usermod -aG docker $USER
ubuntu@ip-172-31-19-80:~$ newgrp docker
ubuntu@ip-172-31-19-80:~$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
ubuntu@ip-172-31-19-80:~$ systemctl status docker
● docker.service - Docker Application Container Engine
     Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; preset: enabled)
     Active: active (running) since Fri 2025-02-28 07:24:14 UTC; 2min 27s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 1922 (dockerd)
      Tasks: 8
     Memory: 27.3M (peak: 29.8M)
        CPU: 258ms
     CGroup: /system.slice/docker.service
             └─1922 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

```
ubuntu@ip-172-31-19-80:~$ docker --version
Docker version 26.1.3, build 26.1.3-0ubuntu1~24.04.1
```

→git clone https://github.com/LondheShubham153/online_shopping_app.git

→ ls

→ rm -v dockerfile

```
ubuntu@ip-172-31-19-80:~$ cd online_shopping_app/
ubuntu@ip-172-31-19-80:~/online_shopping_app$ ls
COMMANDS.md      README.md      docker-compose.yml      index.css          public
CONTRIBUTING.md  ROADMAP.md     docker_installation.sh  index.html         scripts
Dockerfile       appspec.yml    eslint.config.js        package-lock.json  src
LICENSE          buildspec.yml  image_report.md         package.json       vite.config.js
ubuntu@ip-172-31-19-80:~/online_shopping_app$ rm -v Dockerfile
removed 'Dockerfile'
ubuntu@ip-172-31-19-80:~/online_shopping_app$ ls
COMMANDS.md      ROADMAP.md          docker_installation.sh  index.html         scripts
CONTRIBUTING.md  appspec.yml         eslint.config.js        package-lock.json  src
LICENSE          buildspec.yml       image_report.md         package.json       vite.config.js
README.md        docker-compose.yml  index.css               public
```

Here we can see that we have package.json file. So this application needs a build tool Nodejs.

Nodejs → npm

Python → pip

Java → Maven

- We can get info of which build tool to use from the developer.
- Just explore the code for Port we can use them to create Docker file.
- We need a Nodejs in the operating system, we need a base image.
- When we create a container we need to have a base image.
- So to get the base image, then we need to use FROM command.

Now let's write a Docker file,

```
#Base image for NodeJs
FROM node:18

#making a Working directory for all the required files & code
WORKDIR /app

#Copy everything from the source (Host) to the Destination (container)
COPY . .

#Install packages: this makes the container ready for the application
RUN npm install

# Expose the port
EXPOSE 5173

#Serve the application
CMD ["npm","run","dev"]

~
~
~
```

→ docker build -t online_shop:latest .

→ docker images

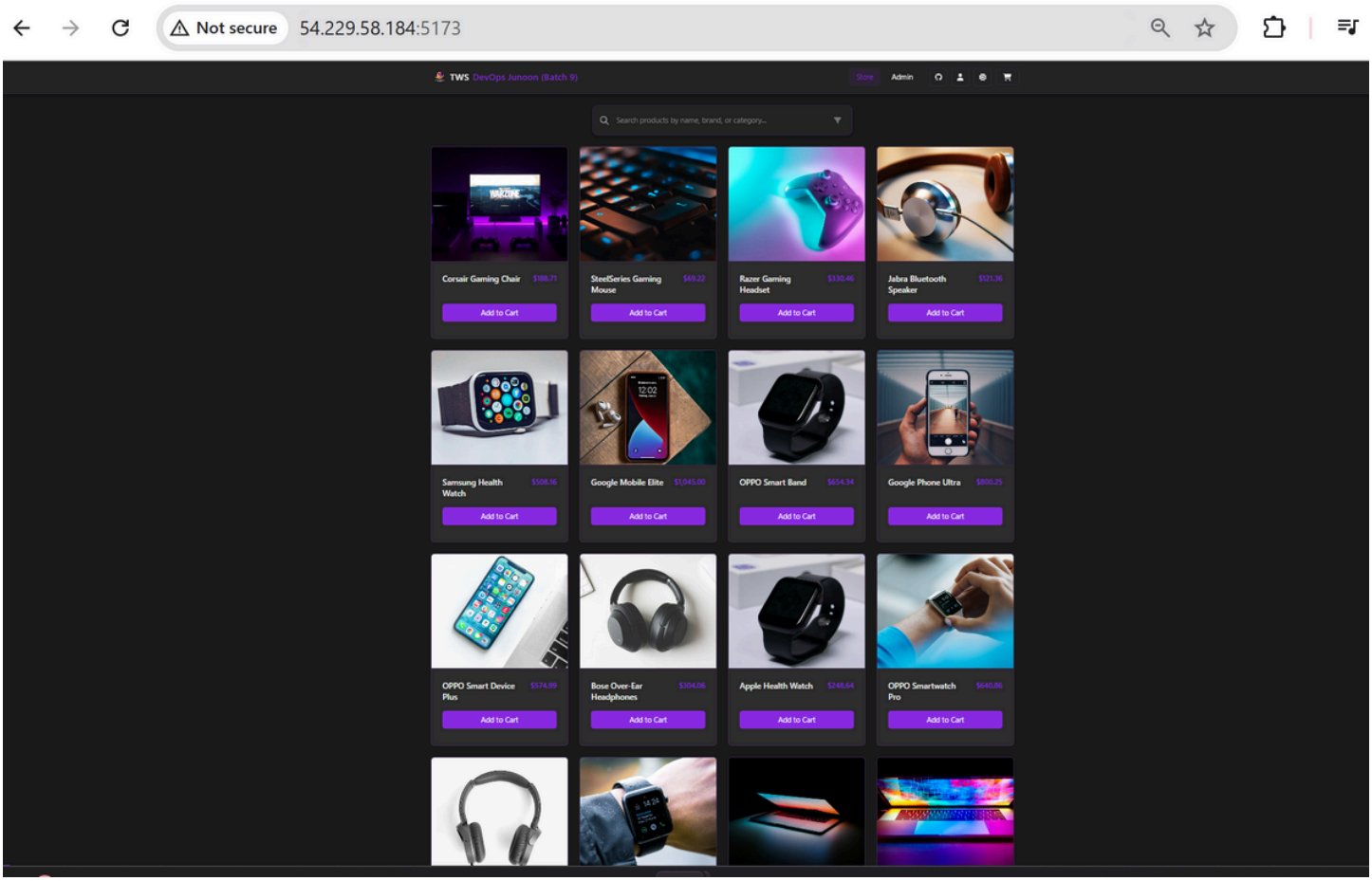
→ docker run -d -p 5173:5173 online_shop:latest

→ docker ps (to check the running containers)

```
Successfully built 946082e40acb
Successfully tagged online_shop:latest
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED              SIZE
online_shop   latest    946082e40acb   About a minute ago   1.22GB
node          18        512bc7f93b1c   7 days ago           1.09GB
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker run -d -p 5173:5173 online_shop:latest
bb7aadbf98862445da2f096a214154f3f4280e0ded7bbc85eadb8f6b4bbcdba3
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED         STATUS         PORTS
                    NAMES
bb7aadbf9886   online_shop:latest   "docker-entrypoint.s…"   3 seconds ago   Up 2 seconds   0.0.0.0:5173->5173
/tcp, :::5173->5173/tcp   elastic_germain
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker logs bb7aadbf9886

> online-shop@0.0.0 dev
> vite --host


  VITE v5.4.14  ready in 274 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: http://172.17.0.2:5173/
```

Open the port 5173, then public ip address:5173

Here's the application,



Docker volumes:

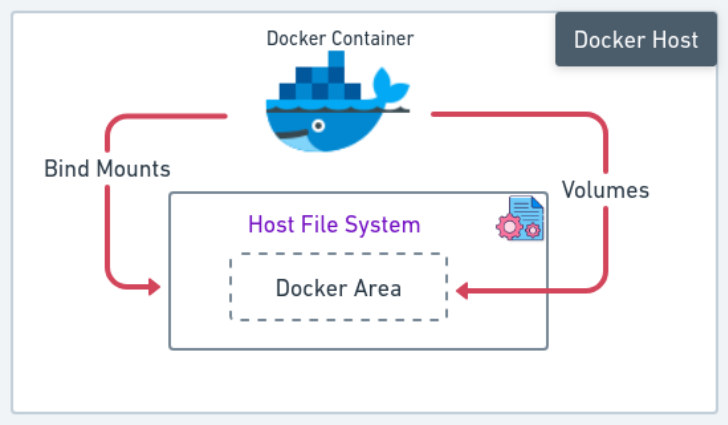
- Container will have a life cycle i.e., create, run, start, stop and pause and then it dies.
- So container will lose its data.
- So map the container data with the Hard disk.
- Hence we can persist the volume of the container to hard disk.
- Volume can be only attached while creating new container.
- Volume is simply a directory inside our container.
- You can't create volume from existing container.
- you can share one volume across any number of container.
- you can map the directory created for volume with the container data.

you can map volume in 2 ways,

- Host to container
- Container to Container

Advantages of Volumes:

- we can decouple storage from containers.
- Share volume with different containers.
- Attach volume to containers.
- Even if container gets crashed,



Quick Command Recap 📜

- docker volume ls → to check the existing volumes
- docker ps → to check containers
- docker images → to check images
- docker volume create my_volume → Create a new volume.
- docker run -d -v my_volume:/path/in/container my_image → Attach a volume to a container.
- docker volume inspect my_volume → Inspect details of a volume.
- docker volume rm my_volume → Remove a volume.
- docker volume prune → Remove unused volumes.

So lets stop the existing container & remove it,

→ docker stop container_id

→ docker rm container_id

```
ubuntu@ip-172-31-19-80:~/online_shopping_app$ cd ..
ubuntu@ip-172-31-19-80:~$ mkdir volume
ubuntu@ip-172-31-19-80:~$ ls
online_shopping_app  volume
ubuntu@ip-172-31-19-80:~$ cd volume/
ubuntu@ip-172-31-19-80:~/volume$ mkdir online_shop
ubuntu@ip-172-31-19-80:~/volume$ cd online_shop/
ubuntu@ip-172-31-19-80:~/volume/online_shop$ pwd
/home/ubuntu/volume/online_shop
ubuntu@ip-172-31-19-80:~/volume/online_shop$ cd ..
ubuntu@ip-172-31-19-80:~/volume$ cd ..
ubuntu@ip-172-31-19-80:~$ cd online_shopping_app/
```

Creating a new container with volume,

→ docker run -v /home/ubuntu/volume/onlline_shop:/logs -p 5173:5173 online_shop:latest

→ docker ps

→ docker exec -it container_id bash

→ create some files as shown below

→ stop & delete the container

→ Now go to this loc /home/ubuntu/volume/onlline_shop, to check if the volume persists.

→ You can see below, volume persists and see the files created.

```
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker run -d -v /home/ubuntu/volume/online_shop:/logs -p 5173:5
173 online_shop:latest
5944528c0d600d6dca1b35ef96c9de9a9d15e53837484a7a920096438109d298
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED         STATUS         PORTS
                          NAMES
5944528c0d60   online_shop:latest   "docker-entrypoint.s…"   3 seconds ago   Up 3 seconds   0.0.0.0:5173->5173
/tcp, :::5173->5173/tcp   brave_davinci
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker exec -it 5944528c0d60 bash
root@5944528c0d60:/app# ls
appspec.yml    eslint.config.js  index.css    node_modules       package.json  scripts  vite.config.js
buildspec.yml  image_report.md   index.html   package-lock.json  public        src
root@5944528c0d60:/app# cd /logs
root@5944528c0d60:/logs# ls
root@5944528c0d60:/logs# echo "this is a log line" > app.log
root@5944528c0d60:/logs# echo "this ia also a log line" >> app.log
root@5944528c0d60:/logs# echo "more logs" >> app.log
root@5944528c0d60:/logs# exit
exit
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker rm 59
Error response from daemon: cannot remove container "/brave_davinci": container is running: stop the container
 before removing or force remove
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker stop 59
59
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker rm 59
59
ubuntu@ip-172-31-19-80:~/online_shopping_app$ cd
ubuntu@ip-172-31-19-80:~$ cd volume/
ubuntu@ip-172-31-19-80:~/volume$ cd online_shop/
ubuntu@ip-172-31-19-80:~/volume/online_shop$ cat app.log
this is a log line
this ia also a log line
more logs
```

Docker Networking

- In Docker, containers are like isolated mini-machines, and they usually need to talk to each other or access the outside world (internet).
- Docker provides different ways to handle these connections, which are called network drivers.

Types of Docker Networking

Docker provides several **network drivers**, each with a different purpose. Let's look at the main ones:

1. **Bridge Network (Default)**

- By default, when you run a container, it connects to a "bridge" network (unless you specify otherwise). This bridge network connects your container to the host system and other containers

on the same host.
- Think of it like a private network where containers on the same Docker host can talk to each other.
- Great for when you have multiple containers that need to communicate with each other, but they don’t need access to the outside world directly.

2. **Host Network**

- The container doesn’t have its own IP address; it just shares the host’s.
- The container uses the **host machine’s network** directly, meaning it shares the same IP address and network interface as the host.
- This is useful for performance-sensitive apps that need low network overhead or when you need the container to directly access the host machine’s services.

3. **Overlay Network**

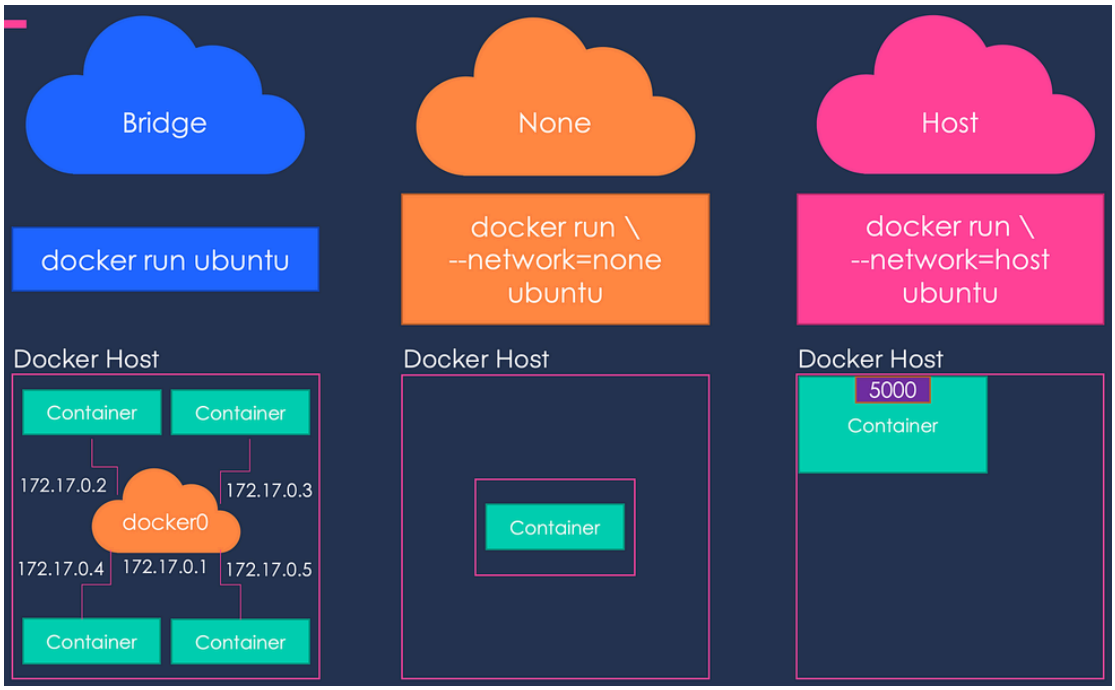
- Docker uses **Swarm mode** or **Kubernetes** to manage containers on different hosts. With an overlay network, containers on different Docker hosts can still communicate with each other as if they were on the same network.
- Perfect for distributed applications where containers need to talk to each other across different physical machines or even across data centers.

4. **None Network**

- The container will not be able to access anything outside its own environment.
- This can be useful for running containers that don't need any networking, like when you just want to run something that doesn’t need to talk to the outside world.
- The container is **completely isolated** with no network access at all.

5.User-defined bridge:

- In Docker, a **user-defined bridge network** is a type of custom network that allows you to create isolated, flexible environments for your containers on a single Docker host.
- It is based on the default bridge driver but gives you more control over settings such as the IP range, subnet, and gateway.
- When containers are connected to a user-defined bridge network, they can communicate with each other using container names as hostnames (via Docker's internal DNS), while being isolated from containers on other networks.



6.IPVLAN: Containers share the same network stack.

7.Macvlan: Containers get their own IP addresses on the physical network.

Docker Networking Commands

- *docker network ls* : to check existing docker networks.
- docker network create <network_name> : to create new network.
- *docker network connect <network_name> <container ID>* : to connect the network with container.
- *docker network inspect <network_name> :* to inspect the network.
- *docker network disconnect < network_name> <container_ID> :* to disconnect the network with container.
- *docker network rm <network_name> :* to delete the existing network.

Let's do Hands-on,

→ docker network ls (to check the networks)

→ docker network create my-net

```
ubuntu@ip-172-31-19-80:~$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
7500cd34569d   bridge    bridge    local
058eee96703c   host      host      local
3413l2792f03   my-net    bridge    local
06a405c30b83   none      null      local
```

→ to inspect use, docker inspect my-net

```
ubuntu@ip-172-31-19-80:~$ docker inspect my-net
[
    {
        "Name": "my-net",
        "Id": "341312792f03c9be6d277b27cfc3ae6a6cd1dd7d1faabf3ab8469e993d0bd80c",
        "Created": "2025-02-28T08:35:49.851436808Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Here you can use in the containers below that we do not have any network mapped to container.

→ docker run -d --network my-net -p 80:80 nginx:latest

```
ubuntu@ip-172-31-19-80:~$ docker run -d --network my-net -p 80:80 nginx:latest
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
7cf63256a31a: Pull complete
bf9acace214a: Pull complete
513c3649bb14: Pull complete
d014f92d532d: Pull complete
9dd21ad5a4a6: Pull complete
943ea0f0c2e4: Pull complete
103f50cb3e9f: Pull complete
Digest: sha256:9d6b58feebd2dbd3c56ab5853333d627cc6e281011cfd6050fa4bcf2072c9496
Status: Downloaded newer image for nginx:latest
bb3593826f85d46cc9c4fac4e03af07617fca06d15917babc8f25c1b5a834d0b
ubuntu@ip-172-31-19-80:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS
           NAMES
bb3593826f85   nginx:latest   "/docker-entrypoint.…"   17 seconds ago   Up 16 seconds   0.0.0.0:80->80/tcp, ::
:80->80/tcp   boring_sutherland
```

Now when you inspect the my-net by using cmd,

→ docker inspect my-net

```
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "bb3593826f85d46cc9c4fac4e03af07617fca06d15917babc8f25c1b5a834d0b": {
                "Name": "boring_sutherland",
                "EndpointID": "c3c0028147f0376a630aa5ae105b783bcfef973d32d9b87d9c7a1bac215ec8da",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Here, you can see that the container is mapped with my-net

```
ubuntu@ip-172-31-19-80:~$ docker stop bb35
bb35
ubuntu@ip-172-31-19-80:~$ docker rm bb35
bb35
ubuntu@ip-172-31-19-80:~$ docker run -d --name nginx --network my-net -p 80:80 nginx:latest
b48f009312cc2d0fff66175523f4feb976b968bef4b495e0a17da8374d0a0472
ubuntu@ip-172-31-19-80:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS         PORTS
          NAMES
b48f009312cc   nginx:latest   "/docker-entrypoint.…"   6 seconds ago   Up 4 seconds   0.0.0.0:80->80/tcp, :::8
0->80/tcp   nginx
```

→ Stopped and removed the container to give it a new name nginx.

→ use docker inspect my-net to check the name of the container as you can see below,

```
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "b48f009312cc2d0fff66175523f4feb976b968bef4b495e0a17da8374d0a0472": {
                "Name": "nginx",
                "EndpointID": "ae8129f96db75af1e8908a5b3d1583fcc04018811fc6cb0657cfb54d9a548de7",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Again added the online_shop app container to the my-net to the docker network we created.

```
ubuntu@ip-172-31-19-80:~$ docker run -d --name online_shop --network my-net -p 5173:5173 online_shop:latest
876fcfd5cd426232c5436ef6bc1e6ebfe41a6f4a3546890cc1cd829dc129047c
ubuntu@ip-172-31-19-80:~$ docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED              STATUS              PORTS
                                    NAMES
876fcfd5cd42   online_shop:latest   "docker-entrypoint.s…"   5 seconds ago        Up 3 seconds        0.0.0.0:
5173->5173/tcp, :::5173->5173/tcp   online_shop
b48f009312cc   nginx:latest         "/docker-entrypoint.…"   About a minute ago   Up About a minute   0.0.0.0:
80->80/tcp, :::80->80/tcp           nginx
```

Now try to inspect the docker network again,


```
        },
        "ConfigOnly": false,
        "Containers": {
            "876fcfd5cd426232c5436ef6bc1e6ebfe41a6f4a3546890cc1cd829dc129047c": {
                "Name": "online_shop",
                "EndpointID": "bdb49b9c47ec302892e39d7ad7e4b3453afbd4750a677eff7c4dc905f5912e90",
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            },
            "b48f009312cc2d0fff66175523f4feb976b968bef4b495e0a17da8374d0a0472": {
                "Name": "nginx",
                "EndpointID": "ae8129f96db75af1e8908a5b3d1583fcc04018811fc6cb0657cfb54d9a548de7",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```


Here you can see that they are two containers mapped to same network which we created my-net

Now, go inside in online_shop container and use curl http://nginx:80

Here container name acts like a IP address and able to connect to it.

Welcome to nginx!!!!!

```
ubuntu@ip-172-31-19-80:~$ docker exec -it 876fcfd5cd42 bash
root@876fcfd5cd42:/app# curl http://nginx:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

→ Two containers online_shop container and nginx container both are in same network my-net

→ Here with in the online_shop container we can connect with nginx container. So we can make both the containers communicate with each other using the Custom bridge network.

Multi Stage builds:

- Multi stage builds is creating the builds part into multi stages.
- It optimizes the docker images, hence it reduces the size of the docker image.
- Improves build performance.
- Handles dependencies more efficiently.
- With multi-stage builds, you can use different images for different stages and only copy the necessary artifacts from each stage into the final image.
- Each stage in a multi-stage build is defined using the FROM instruction, and you can label each stage with a name using the AS keyword.

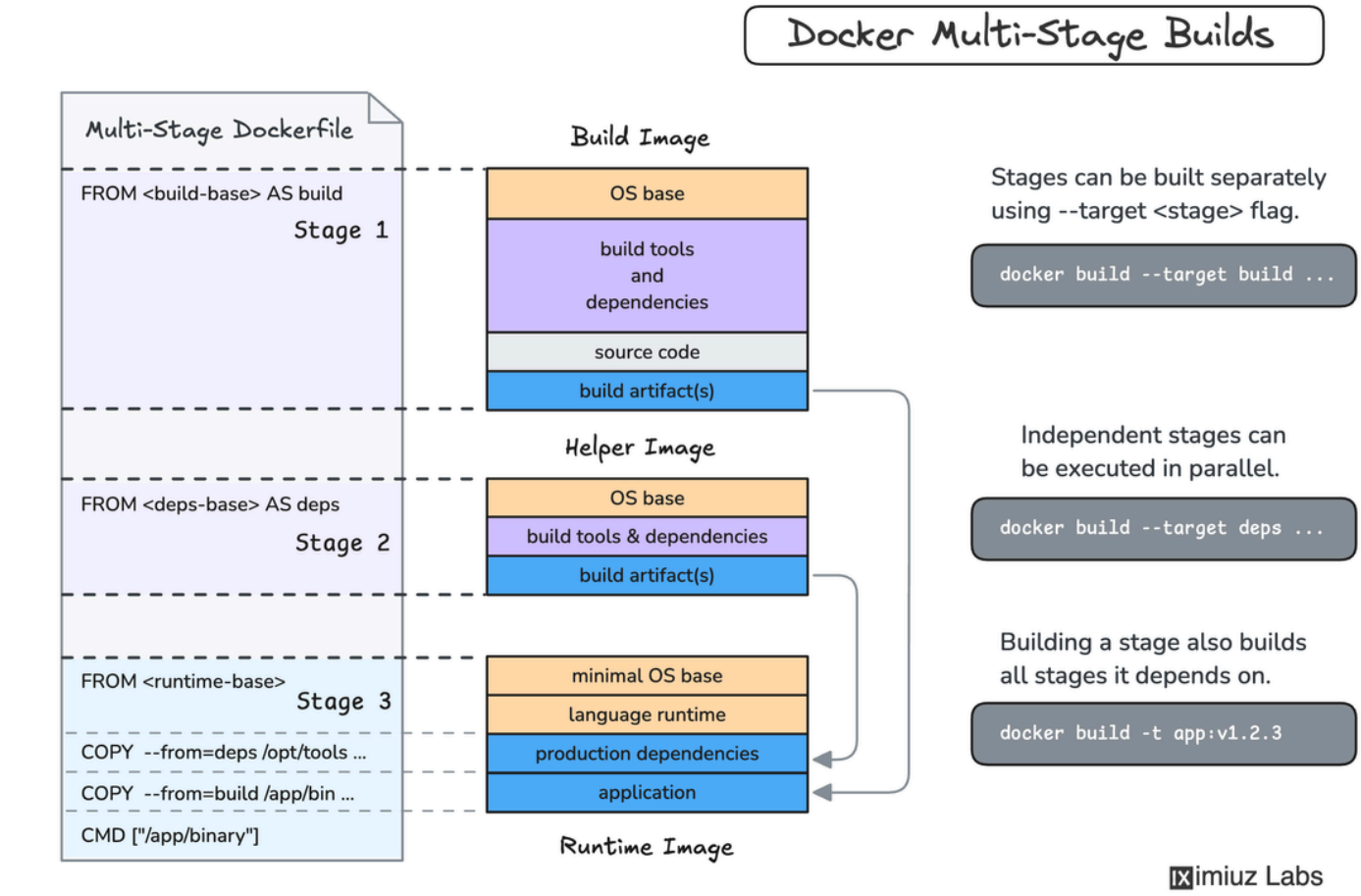
How Multi-Stage Builds Work

- **Stage 1**: This stage is used for compiling, building, or preparing dependencies

(e.g., source code compilation or downloading dependencies).

- **Stage 2**: This stage is used for running the application, and only the necessary artifacts

(e.g., binaries, static files) from the first stage are copied into the final image.




```
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
online_shop   latest    946082e40acb   2 hours ago     1.22GB
node          18        512bc7f93b1c   7 days ago      1.09GB
nginx         latest    b52e0b094bc0   3 weeks ago     192MB
```

Let’s create multi-stage docker file for online-shopping-app,

```
#base image to build NPM related packages install (stage 1) (big image) 1GB

FROM node:18 AS builder

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .



#Base image to run the application only (Stagee 2) (Small image) [500GB]

FROM node:18-alpine

WORKDIR /app /app

COPY --from=builder /app .

EXPOSE 5173

CMD ["npm","run","dev"]
~
~
~
~
```

Docker file - Docker image command

→ docker build -t online_shop_small -f ./Dockerfile-multi

Docker image to Docker container command

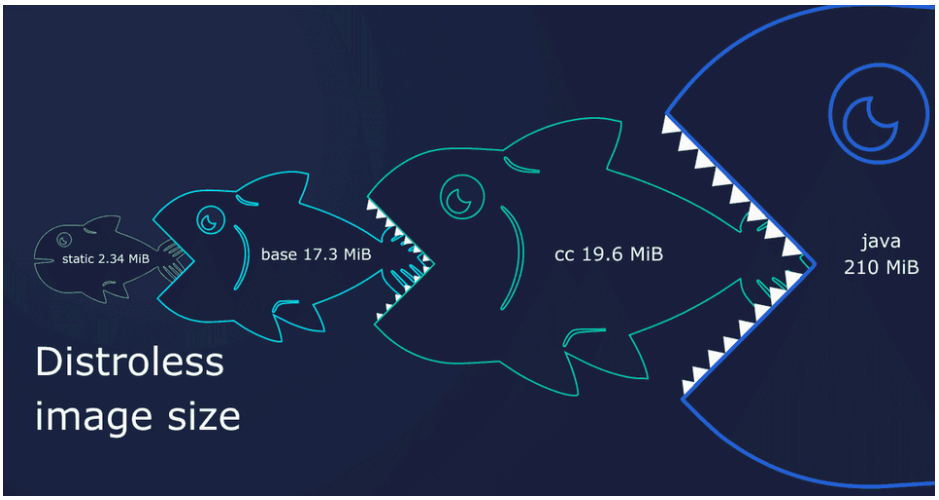
→ docker run -d -p 5173:5173 online_shop_small:latest

```
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker images
REPOSITORY          TAG          IMAGE ID       CREATED          SIZE
online_shop_small   latest       6aa346aef949   10 seconds ago   230MB
<none>              <none>       56e21bc72fba   56 seconds ago   1.22GB
online_shop         latest       946082e40acb   2 hours ago      1.22GB
node                18           512bc7f93b1c   7 days ago       1.09GB
node                18-alpine    70649fe1a0d7   7 days ago       127MB
nginx               latest       b52e0b094bc0   3 weeks ago      192MB
```

Here you can see the difference between the image size of online_shop (1.22GB) & online_Shop_small (230MB)

Distroless image:

- **Distroless images** are Docker images that contain only the application and its runtime dependencies, without any unnecessary OS libraries or tools.
- The idea behind distroless images is to keep the Docker image as small as possible.
- Distroless images are generally used in production to ensure both a smaller attack surface and faster security patching.
- Often used with multi-stage builds to separate the build environment from the final runtime image.
- Smaller images lead to faster container start times and less disk usage.
- Significantly smaller than traditional base images (e.g., Alpine, Ubuntu).




```
#base image to build NPM related packages install (stage 1) (big image) 1GB

FROM node:18 AS builder

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .



#Base image to run the application only (Stagee 2) (Small image) [500GB]

FROM gcr.io/distroless/nodejs18-debian12

WORKDIR /app /app

COPY --from=builder /app .

EXPOSE 5173

CMD ["npm","run","dev"]
~
~
```

→ in the same file of the multi-stage builds, we can take the distroless image this time and check if it reduces the size of the image.

→ docker build -t online_shop_distroless -f ./Dockerfile-multi .

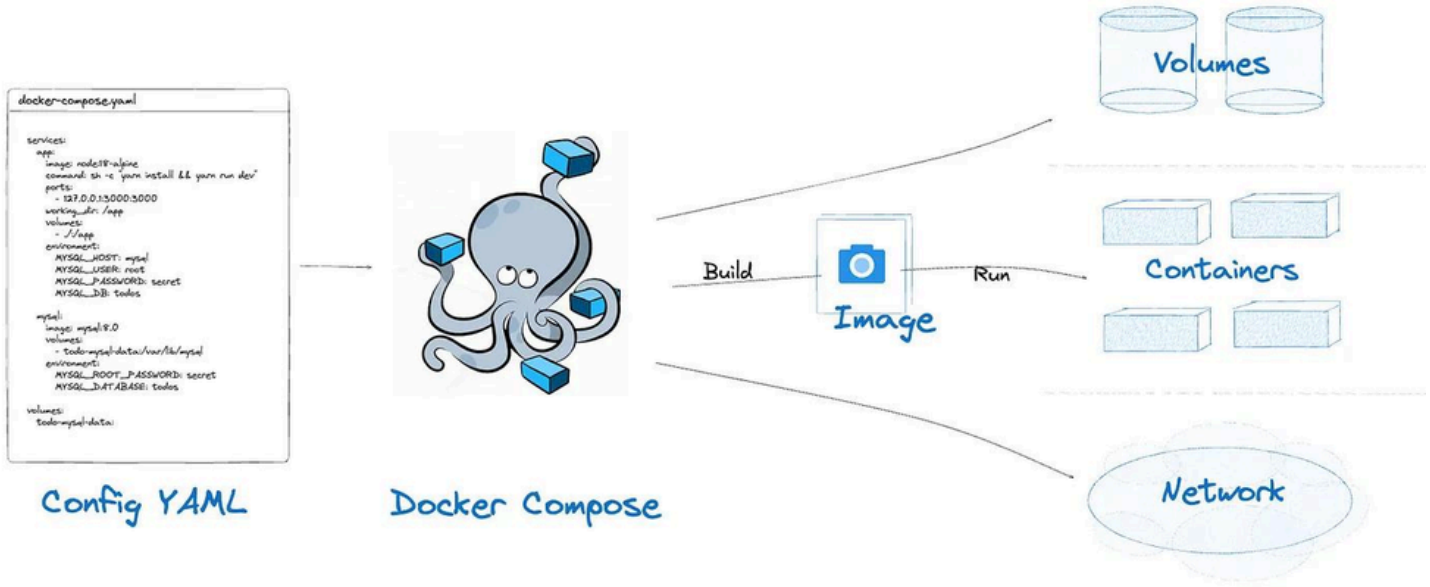
→ docker images

```
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker images
REPOSITORY                                TAG         IMAGE ID       CREATED              SIZE
online_shop_distroless                    latest      1fee2e123db5   9 seconds ago        218MB
<none>                                    <none>      eeb7dd4f05f9   About a minute ago   1.22GB
online_shop_small                         latest      6aa346aef949   16 minutes ago       230MB
<none>                                    <none>      56e21bc72fba   17 minutes ago       1.22GB
online_shop                               latest      946082e40acb   3 hours ago          1.22GB
node                                      18-alpine   70649fe1a0d7   7 days ago           127MB
node                                      18          512bc7f93b1c   7 days ago           1.09GB
nginx                                     latest      b52e0b094bc0   3 weeks ago          192MB
gcr.io/distroless/nodejs18-debian12       latest      bb3148467fc6   N/A                  115MB
```

As you can see clearly, their is a difference between online_shop_small and distroless image

Docker compose

- Docker Compose is a tool for defining and running multi-container Docker applications.
- It uses a YAML file (docker-compose.yml) to define services, networks, and volumes in a single configuration.
- Makes it easy to define and manage multiple containers that work together (e.g., a web app, database, and cache)
- Each container in the application is defined as a *service* in the YAML file.
- With a single docker-compose command, you can start, stop, and manage multiple containers.



Key Components of docker-compose.yml

1. **version**: Specifies the Compose file version (e.g., version: '3.8').
2. **services**: Defines the containers (services) that will run. Each service can have:
   - image: The Docker image to use.
   - build: Path to the Dockerfile.
   - ports: Mapping of container ports to host ports.
   - volumes: Mounting directories or files.
   - environment: Environment variables for the container.
   - networks: Specifies which network(s) the service should connect to.
3. **volumes**: Defines named volumes for data persistence.
4. **networks**: Defines custom networks to isolate or connect services.

→ sudo apt-get install docker-compose-v2

→ vim docker-compose.yml

```yaml
services:
  online_shop:
    build:
      context: .

    container_name: online_shop
    ports:
      - "5173:5173"
    networks:
      - my-net

  nginx:
    image: nginx:latest
    container_name: nginx
    ports:
      - "80:80"
    networks:
      - my-net

networks:
  my-net:
~
~
~
```

→ docker compose up -d (to create)

→ docker compose down (to remove)

```
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker compose up -d
[+] Running 3/3
 ✔Network online_shopping_app_my-net  Created                                    0.1s
 ✔Container nginx                     Started                                    1.1s
 ✔Container online_shop               Started                                    1.1s
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED         STATUS         PORTS
                                  NAMES
54b100abc08d   online_shopping_app-online_shop   "docker-entrypoint.s…"   7 seconds ago   Up 5 seconds   0.0.0
.0:5173->5173/tcp, :::5173->5173/tcp   online_shop
e47c898559d3   nginx:latest                   "/docker-entrypoint.…"   7 seconds ago   Up 5 seconds   0.0.0
.0:80->80/tcp, :::80->80/tcp           nginx
```

```
ubuntu@ip-172-31-19-80:~/online_shopping_app$ docker compose down
[+] Running 3/2
 ✔Container nginx                        Removed
 ✔Container online_shop                  Removed
 ✔Network online_shopping_app_my-net     Removed
```

Jenkins

Let's learn CI/CD

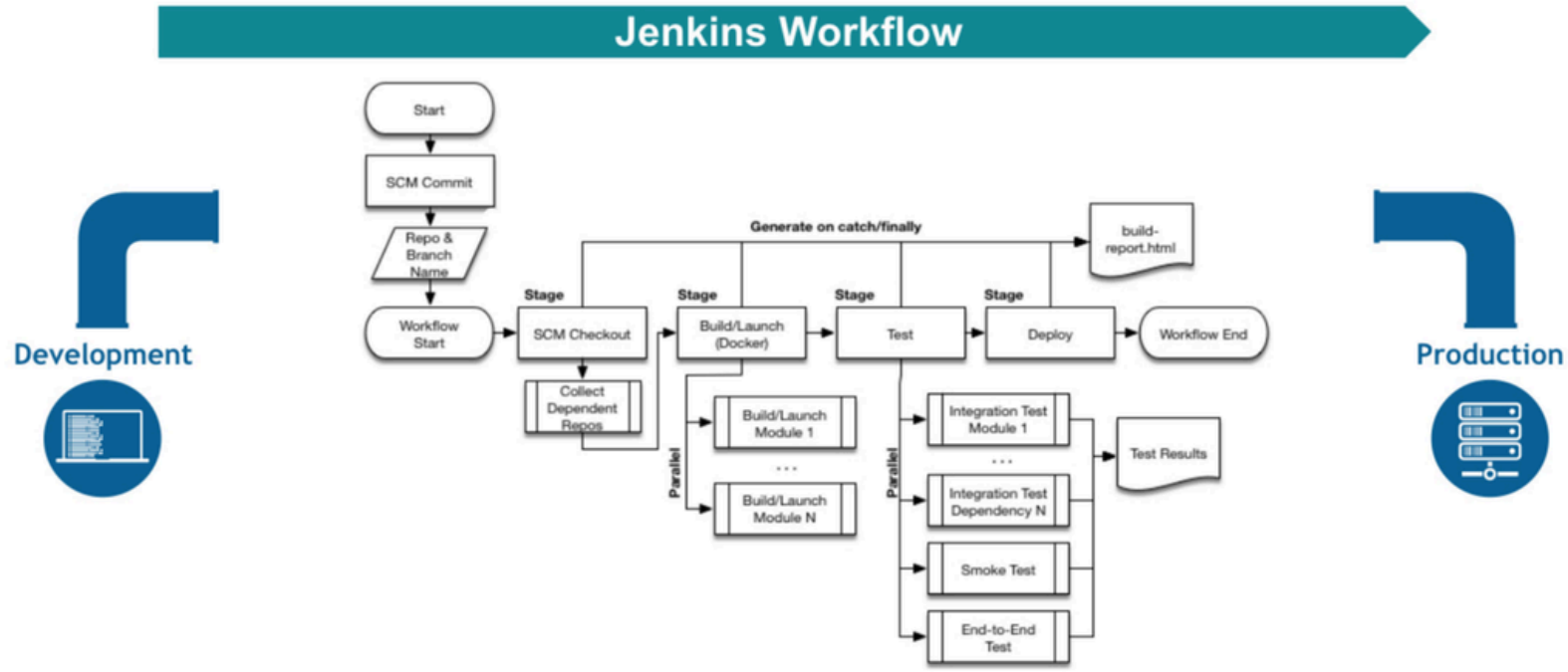
- **CI** means Continuous Integration, and **CD** can mean Continuous Deployment or Continuous Delivery.
- CI/CD is a methodology that helps to integrate with many other tools & ensures shorter and smooth Software Development Life Cycle.
- **Continuous Integration (CI)** is about automatically adding code changes from different people into a shared codebase several times a day. Each change is checked by an automated build and tests, which helps find bugs early.
- **Continuous Delivery (CD)** ensures that the code is always ready to be deployed. **Continuous Deployment** takes it further by automatically deploying every change that passes the CI tests to production.
- Automation and providing Integration
- Shorter & Smooth SDLC process
- Automated Test
- Reliable deployments

Jenkins:

- Jenkins is a Open source tool that is written in Java, hence it is free and community supported & might be your first choice tool for CI.
- Jenkins can run on any major platform without any Compatibility issues.
- We can install Jenkins on Windows, Linux and Mac.
- Jenkins has a groovy syntax.

Jenkins Workflow:

- → Code commit
- → build
- → Test
- → staging
- → Deploy



Installation of Jenkins:

Install Java

→ sudo apt update

→ sudo apt install fontconfig openjdk-17-jre

→ java -version

→ sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \

https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key

→ echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]" \

https://pkg.jenkins.io/debian-stable binary/ | sudo tee \

/etc/apt/sources.list.d/jenkins.list > /dev/null

→ sudo apt-get update

→ sudo apt-get install jenkins -y

→ sudo systemctl status jenkins

Open port 8080

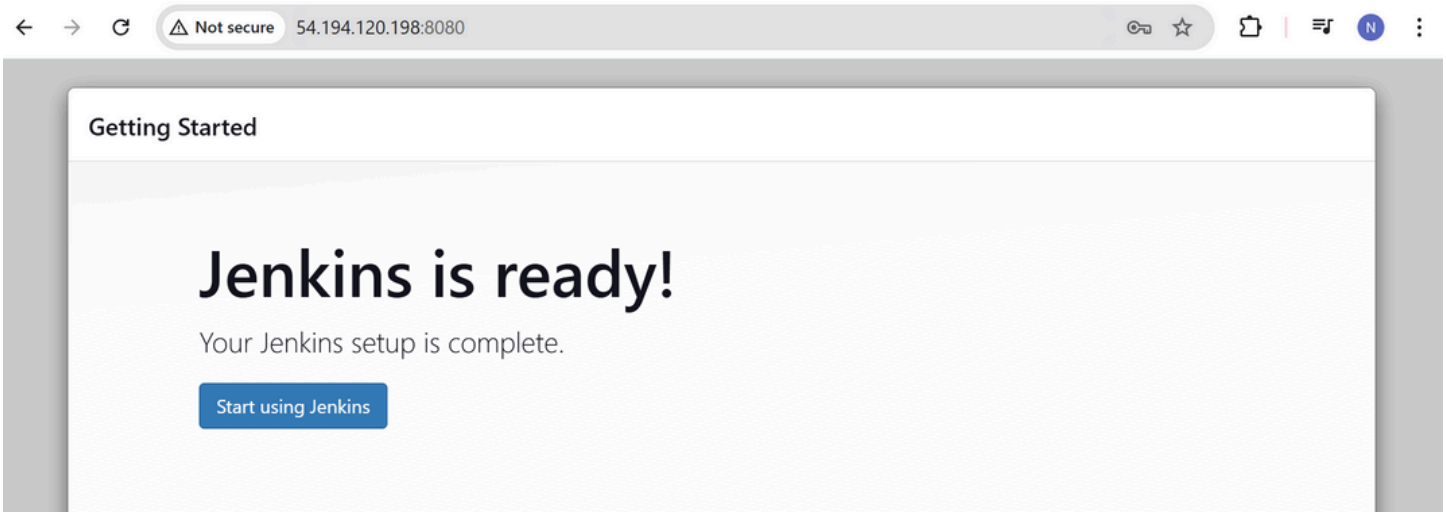
Unlock jenkins using,

→ sudo cat /var/lib/jenkins/secrets/initialAdminPassword

→ Install suggested pulgins

→ create first admin user

→ Jenkins is ready!!!!



Jenkins Job creation & building script with groovy syntax

The Jenkins dashboard provides:

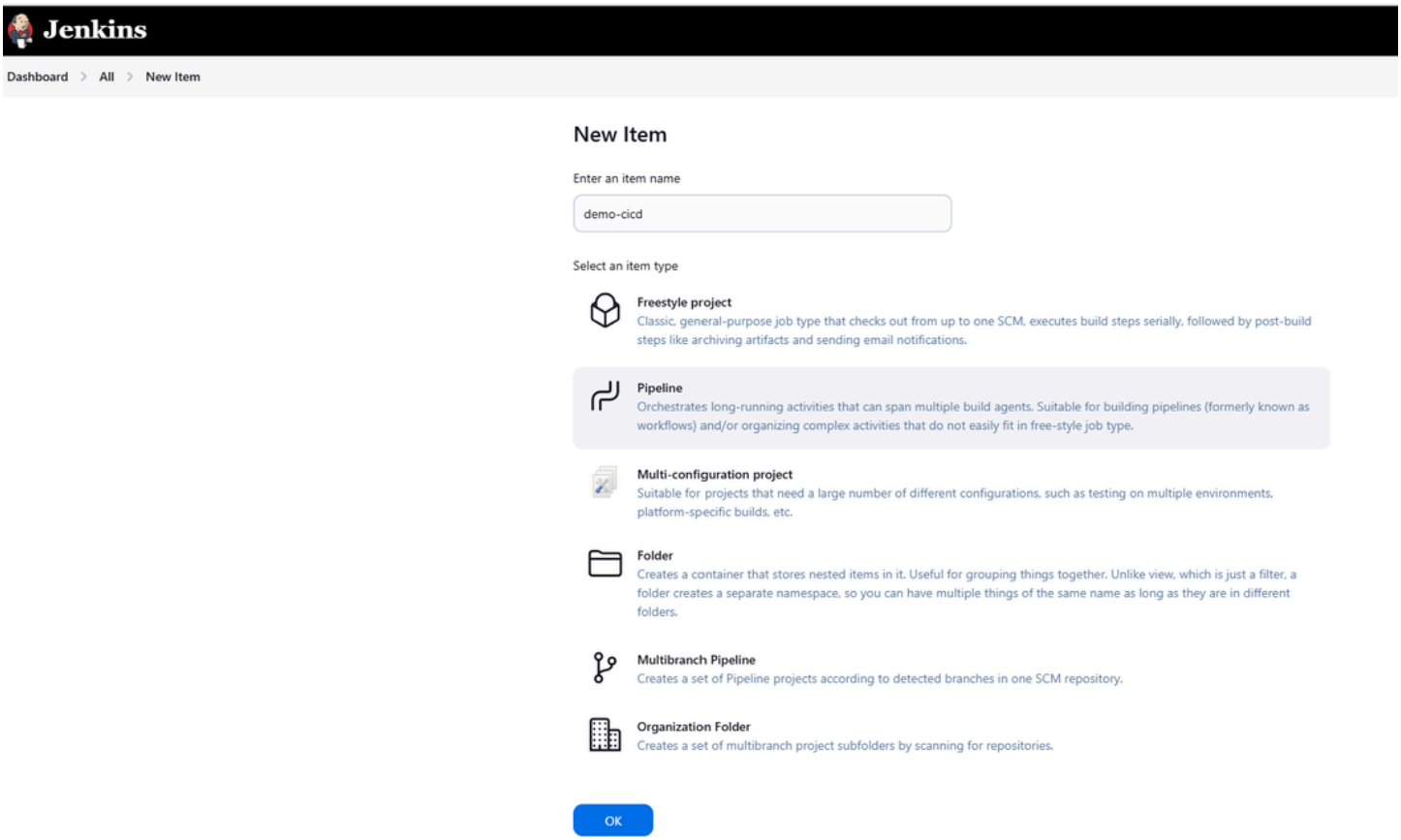
- **New Item**: Create a new job or pipeline.
- **Build History**: View the history of previous builds, including status (success/failure), and logs.
- **Manage Jenkins**: Configure global settings, install plugins, and manage users.
- **People**: View user activity, such as who triggered a build or who created a job.

Create Your First Jenkins Job

Example: Building a two-tier-flask app

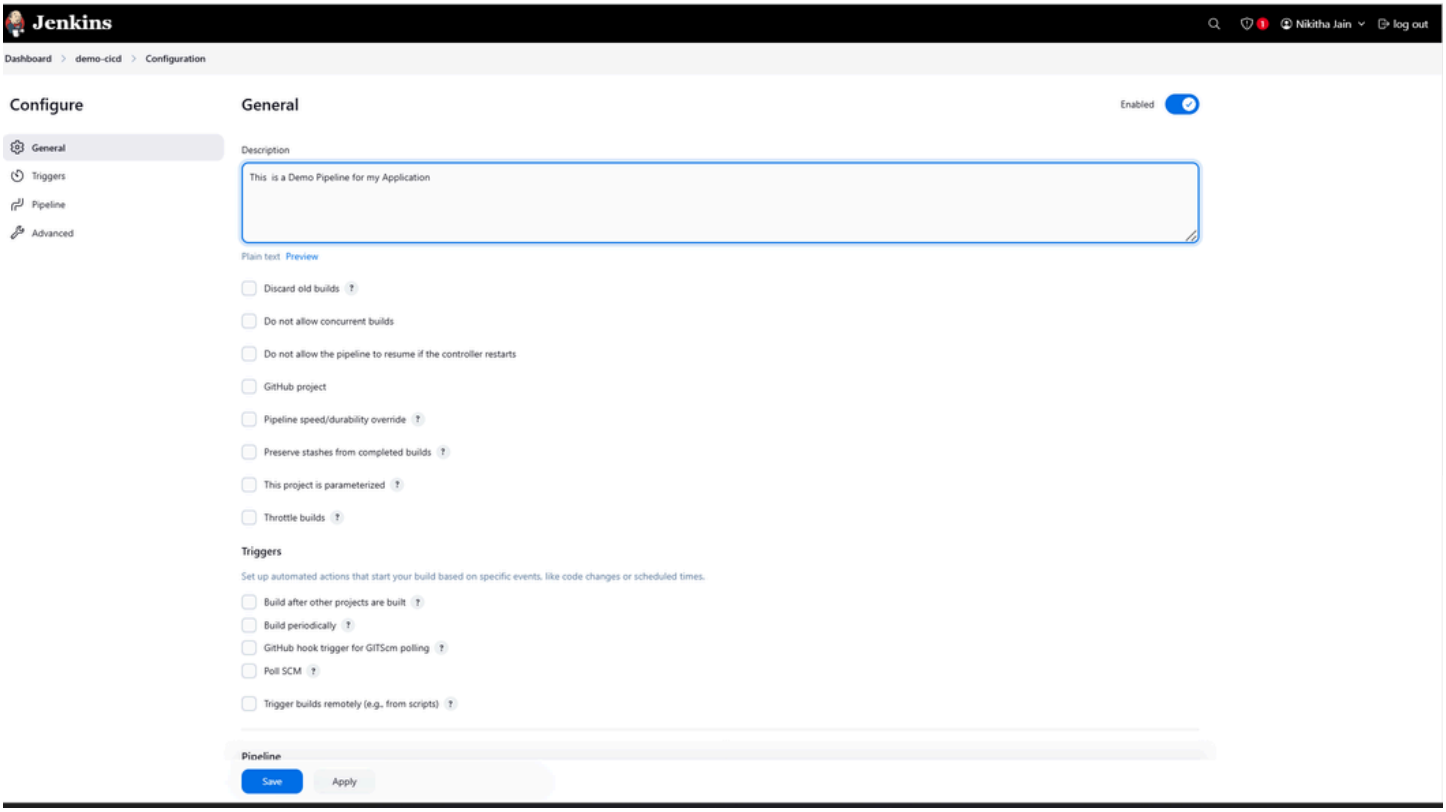
To set up CD:

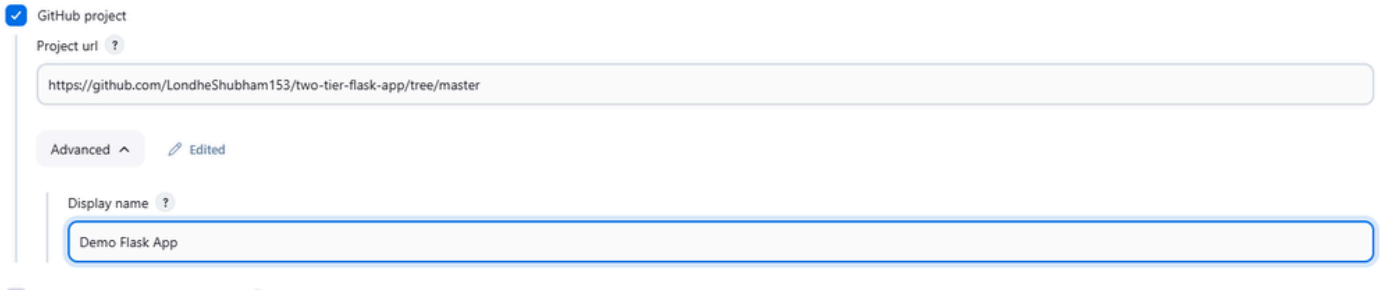
1. **Create a New Pipeline Job**:
    - In Jenkins, go to the **New Item** menu, choose **Pipeline**, and give it a name.



Define the Pipeline Script:

- In the job configuration, you'll see a **Pipeline** section where you can define your pipeline script in **Jenkinsfile** syntax.
- Discard old builds: to make sure our Jenkins is not taking lot of space, we can discard the old builds using log rotation strategy.
- Do not allow concurrent builds: It's just if you can on build it concurrently builds. This can take your RAM.
- Do not allow the pipeline to resume if the controller restarts: A controller check which jobs to start and which job to not to start.
- Github project: Can provide git url and Name
- Pipeline speed/durability override: if Jenkins fails, then it runs its pipeline speed.
- Preserve stashes from complete builds: After build completes, if their is any need for preserving logs or cache then it does that.
- This Project is parameterised: providing runtime arguments. Adding parameters like we do for shell scripting.
- Throttle builds: here you can mention number of builds and Time period required between each build.





we use a special syntax for pipeline which is on Groovy syntax,

- They are two types:
- **Declarative syntax and Static Syntax**
- Below you can see the Declarative syntax and static syntax is written with execute shell where you can write all the Commands like we do on CLI.

A basic pipeline script for deploying an application might look like this:

```
pipeline {


  agent any;

  stages {

    stage("Code") {

      steps {

        git url: "https://github.com/LondheShubham153/two-tier-flask-app.git",branch: "master"
```

```
        }
      }
      stage("Build") {
        steps {
          sh "docker build -t myapp ."
        }
      }
      stage("Test") {
        steps {
          echo "Developer/Tester tests likh ke dega.."
        }
      }
      stage("Deploy") {
        steps {
          sh "docker compose up -d"
        }
      }
  }
}
```

→ Install docker, sudo apt-get install docker.io

→ sudo usermod -aG docker jenkins

→ sudo usermod -aG docker $USER

→ newgrp docker

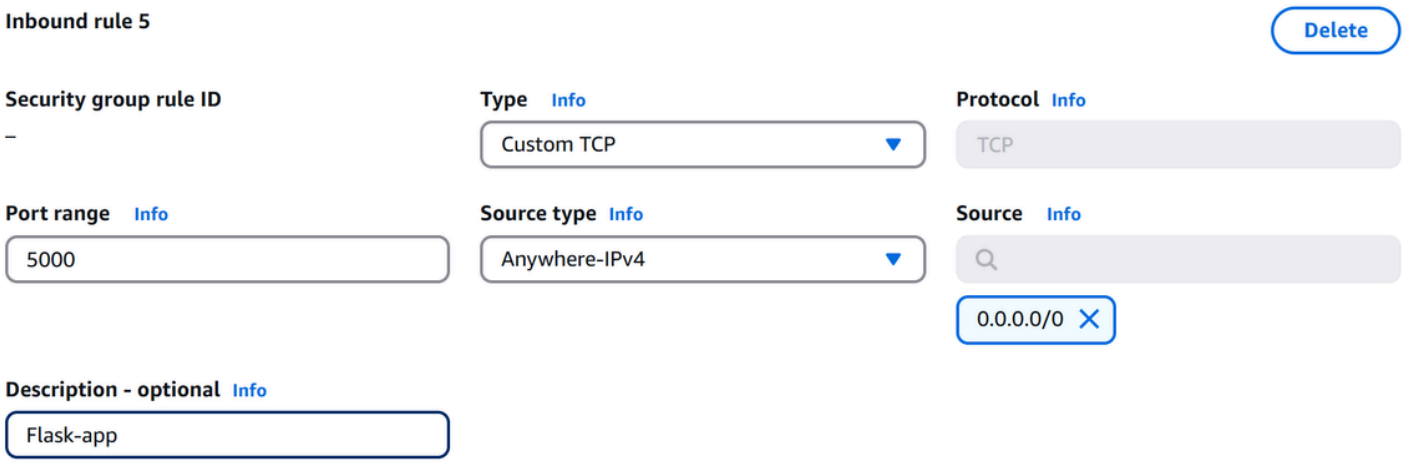
→ sudo systemctl restart jenkins

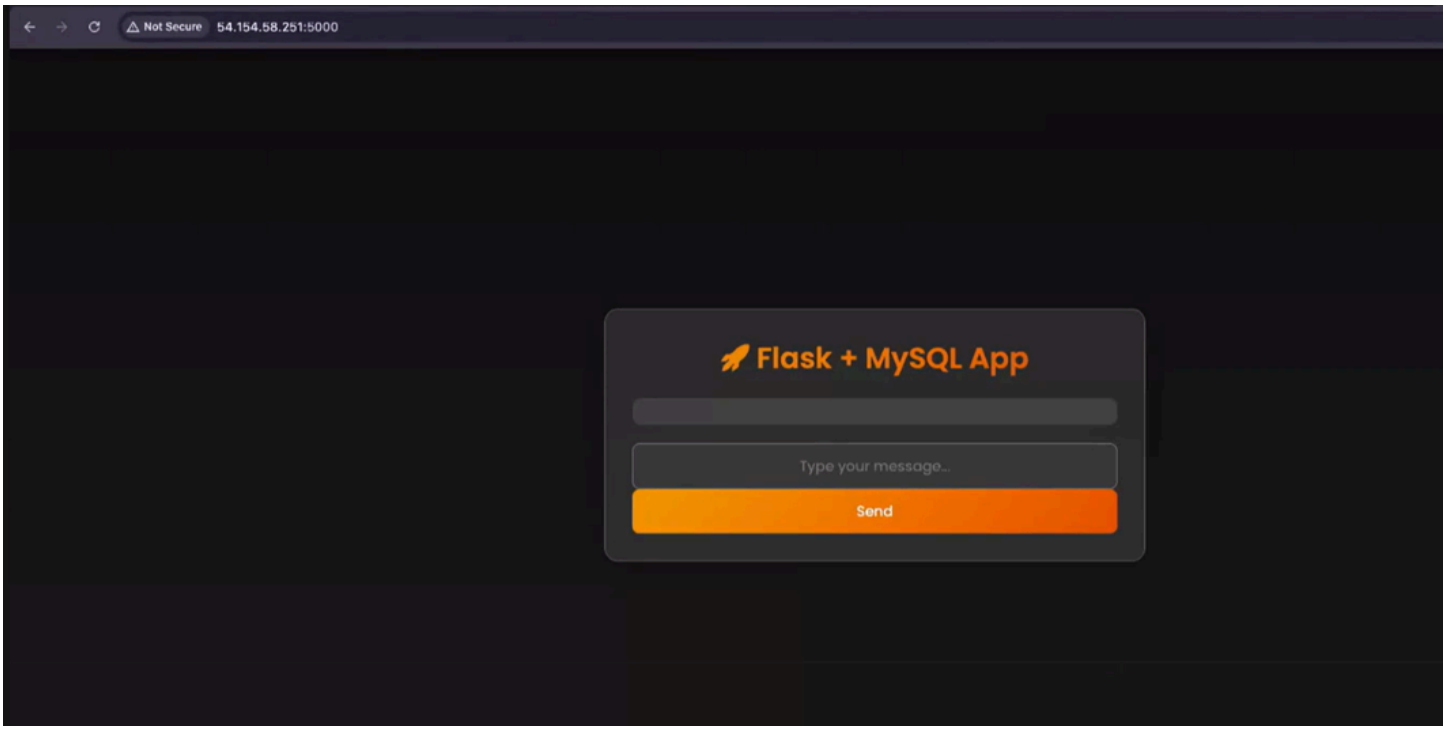
→ sudo apt-get install docker-compose-v2

```
ubuntu@ip-172-31-33-214:/var/lib/jenkins/workspace$ cd demo-cicd/
ubuntu@ip-172-31-33-214:/var/lib/jenkins/workspace/demo-cicd$ sudo usermod -aG docker jenkins
ubuntu@ip-172-31-33-214:/var/lib/jenkins/workspace/demo-cicd$ sudo usermod -aG docker $USER
ubuntu@ip-172-31-33-214:/var/lib/jenkins/workspace/demo-cicd$ newgrp docker
```


```
ubuntu@ip-172-31-33-214:/var/lib/jenkins/workspace/demo-cicd$ docker images
REPOSITORY   TAG         IMAGE ID       CREATED          SIZE
myapp        latest      82363b6b95be   24 minutes ago   391MB
python       3.9-slim    1a47c1aaa88f   2 months ago     126MB
```

Now go, to security group and open port 5000 as the app runs on port 5000





What is a Jenkins Agent?

A Jenkins agent, also referred to as a "slave" in older terminology, is a separate machine or process that connects to the Jenkins controller (formerly called the master). Its primary role is to execute specific tasks, such as building and testing software, as directed by the controller. By offloading tasks to agents, Jenkins achieves a distributed setup that can handle multiple parallel builds across diverse environments.

Key Features of Jenkins Agents

- **Scalability**: Agents allow Jenkins to distribute workloads, enabling teams to run multiple builds and tests simultaneously.
- **Platform Diversity**: They support varied operating systems and environments (Linux, Windows, macOS), making it possible to test software across different setups.
- **Flexibility**: Agents can be configured to run specific types of jobs, such as compiling code, running integration tests, or deploying applications

Setting Up a Jenkins Agent

There are several ways to set up and connect Jenkins agents:

1. **SSH Agents**: The agent is installed on a remote machine, and Jenkins communicates via SSH.
2. **Java Web Start (JWS)**: The agent is launched using a JAR file downloaded from the Jenkins UI.
3. **Containerized Agents**: Using Docker or Kubernetes to create and manage agent instances in isolated containers.
4. **Cloud-Based Agents**: Leveraging Jenkins plugins to connect to cloud environments.