

## Container Registry

Login into Docker

```
$ docker login -u <username>
```

Attached tag to existing Docker image

```
$ docker tag <image_name> <repository_path>:<tag_name>
```

Published an image to the Docker Hub

```
$ docker push <repository_path>:<tag_name>
```

Pull an image form Docker Hub

```
$ docker pull <repository_path>/<image_name>:<tag_name>
```

# **<repository\_path>** = **account\_id / repository\_name**

# Instead of before pulling image from docker hub we can directly create container and add image path from the docker hub then it automatically pulling the image from the docker hub.

```
$ docker run -d --name <container_name> -p <host_port>:<container_port> <repository_path>:<tag_name>
```

Search Image in Docker Hub

```
$ docker search <image_name>
```

## Docker Container

Create and run a container an image, with custom name

```
$ docker run --name <container_name> <image_name>
```

Run a container with and publish a container's port to the host

```
$ docker run -p <host_port>:<container_port> <image_name>
```

Run the container in background

```
$ docker run -d <image_name>
```

Instead of using above three commands separately we can perform in single command

```
$ docker run -d --name <container_name> -p <host_port>:<container_port> <image_name>
```

Ex. 

```
$ docker run -d --name spider -p 80:80 nginx
```

Start an existing container

```
$ docker start <container_name> (or <container_id>)
```

Stop an existing container

```
$ docker stop <container_name> (or <container_id>)
```

Remove stopped container:

```
$ docker rm <container_name>
```

Forcefully remove container if is in running state

```
$ docker rm -f <container_name>
```

Remove unused(stopped) containers

```
$ docker container prune
```

Enter inside container and open shell inside container

```
$ docker exec -it <container_name> bash
```

To check details of container using inspect

```
$ docker inspect <container_name> (or <container_id>)
```

List running containers

```
$ docker ps
```

List all containers running as well as stopped

```
$ docker ps -a
```

View resource usage stats

```
$ docker container stats
```

Fetch and follow the logs of a container

```
$ docker logs -f <container_name>
```

---

## Docker Images

List local images

```
$ docker images
```

Delete image

```
$ docker rmi <image_name>
```

Forcefully delete image

```
$ docker rmi -f <image_name>
```

Remove unused images

```
$ docker image prune
```

---

## Volumes

1. **Bind Mount:** A **bind mount** in Docker is a way to **map a specific file or directory from your host machine to a container**, so that both the host and the container can access and share the same data.
  - Need to specify the exact host file/directory (absolute path required).
  - Host ↔ Container Sharing - Changes in one are reflected in the other **in real-time**.
  - Use Case Examples - Edit code on host and test inside container.
    - Share configs/logs.

Create an directory on locally

```
$ mkdir /opt/data(dir_name)
```

Create an file inside data directory

```
$ index.html
```

Create container and mount dir(volume) to the container

```
$ docker run -d --name <container_name> -p <host_port>:<container_port> -v <dir_to_mount>:<nginx/apache index.html path> <image_name>
```

Example:

```
$ docker run -d --name gaurav -p 80:80 -v /opt/data:/usr/share/nginx/html nginx
```

**-v /opt/data:/usr/share/nginx/html** – This syntax tells docker to use the host dir /opt/data and bind it to the container path /usr/share/nginx/index.html

2. **Docker Volume:** A **Docker volume mount** is a way to **persist and share data** between Docker containers and the host system using **Docker-managed storage** (called **volumes**).

Create docker volume

```
$ docker volume create <volume_name>
```

- This creates a named volume managed by docker
- Docker stores it under /var/lib/docker/volumes/...
- Data stays even if the container is deleted
- Multiple containers can use the same volume
- Easy with docker volume commands

Run container using the above volume

```
$ docker run -d --name <container_name> -p <host_port>:<container_port> -v <volume_name>:>:<nginx/apache index.html path> <image_name>
```

Example:

```
$ docker run -d --name bala -p 81:80 -v myvolume:/usr/local/apache2/htdocs httpd
```

List Volume

```
$ docker volume ls
```

Inspect a Volume

```
$ docker volume inspect <volume-name>
```

Remove a Volume

```
$ docker volume rm <volume-name>
```

Remove Unused Volumes


```
$ docker volume prune
```

Now, next steps:

- Copy files into the volume or creates files inside volume

## Docker File

- A Dockerfile is a **script with a set of instructions** that Docker uses to build a custom image automatically. Instead of creating containers manually and installing everything step by step, you define how to build your image in a Dockerfile.

 How It Works:

You write a Dockerfile → Docker reads it → Docker builds an image → You run containers from that image.

Run dockerfile

```
$ docker build -f Dockerfile .
```

OR

```
$ docker build .
```



## Docker Compose

- **Docker Compose** is a tool for **defining and running multi-container Docker applications**. It uses a YAML file (typically called docker-compose.yml) to **configure services, networks, and volumes** in one place.
- YAML: Yet Another Markup Language.

### Why Use Docker Compose?

Without Compose:

- You run individual docker run commands per container.
- Hard to manage complex apps (frontend, backend, database, etc.)

With Compose:

- One file (docker-compose.yml) manages everything.
- One command (docker-compose up) runs the whole stack.

Run Build and start all containers

```
$ docker compose up
```

Run in detached (background) mode

```
$ docker compose up -d
```

Stop and remove all containers/networks

```
$ docker compose down
```

Build images defined in the file

```
$ docker compose build
```

View logs from all containers

```
$ docker compose logs
```

List all running containers

```
$ docker compose ps
```

## Docker Networking

**Docker Networking** allows containers to **communicate with each other**, with the **host machine**, and with the **outside world**. It handles IP addressing, DNS, routing, and port mapping so containers can talk to each other securely and efficiently.

## Types of Docker Networks

Docker provides several **built-in network drivers**, each serving different use cases:

Network Driver	Purpose	Container-to-Container Communication	Host Access	Custom DNS
bridge	Default for standalone containers	✓ Yes (manually or via custom bridge)	✓	✓
host	Shares host network stack	⚠ Not isolated (shares host IP/ports)	✓	✗
none	Disables networking	✗ No networking	✗	✗
overlay	Cross-host networking (Swarm)	✓ Yes (multi-host)	✓	✓
macvlan	Assigns containers a MAC & IP on host network	✓	✓	✗



### 📌 1. bridge Network (Default)

- Each container gets a private IP.
- Can communicate **via container name** if on the **same custom bridge**.
- Expose ports to communicate with host.

```
$ docker network create my-bridge  
$ docker run -d --name app1 --network my-bridge nginx  
$ docker run -d --name app2 --network my-bridge busybox
```

### 📌 2. host Network

- Container shares host network (no isolation).
- Used for high-performance or system-level apps.

```
$ docker run --network host nginx
```

📌 Note: Only works on **Linux**.

### 📌 3. none Network

- Completely **isolated** network (no internet, no internal access).
- Use for security or testing.

```
$ docker run --network none
```

#### 📌 4. overlay Network (Swarm Mode)

- Used for **multi-host container communication**.
- Requires Docker Swarm.
- Useful in distributed systems (e.g., frontend on one host, backend on another).

```
$ docker swarm init  
$ docker network create --driver overlay my-overlay
```

#### 📌 5. macvlan Network

- Containers get IP from **host's physical network**.
- Appears as separate machine on the network.

```
docker network create -d macvlan --subnet=192.168.1.0/24 --gateway=192.168.1.1 -o parent=eth0 macvlan-net
```

#### 🔧 Useful Commands

List all networks

```
$ docker network ls
```

View details of a network

```
docker network inspect <network>
```

Create a custom network

```
$ docker network create <name>
```

Remove a network

```
$ docker network rm <name>
```

Connect a container to a network

```
$ docker run --network <network>
```

Attach/detach container to/from network

```
$ docker network connect/disconnect
```

### Example: Custom Bridge Network

```
$ docker network create mynet  
$ docker run -d --name db --network mynet mysql  
$ docker run -d --name app --network mynet myapp
```

✅ **app** can now resolve **db** using container name.

When two containers are connected to the **same custom Docker bridge network**, Docker sets up an **internal DNS** system. This means:

**Containers can talk to each other using their container names as hostnames.**

✅ Communicating from BusyBox to MySQL (db)

🧱 1. Create a Custom Docker Network (important!)

```
$ docker network create mynet
```

This allows containers to resolve each other by **name** (via Docker's internal DNS).

🔗 2. Run MySQL (db) Container on that Network

```
$ docker run -d --name db --network mynet -e MYSQL_ROOT_PASSWORD=root mysql
```

- Name: db (this becomes the hostname for other containers)
- Password is mandatory for MySQL to start
- Attached to mynet

🔧 3. Run BusyBox (app) Container on Same Network

```
$ docker run -it --name app --network mynet busybox
```

Once inside the BusyBox shell:

```
$ ping db          # Should work
```