**Docker tutorial:**

Docker is a set of Platforms as a service (PaaS) products that use Operating system-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries, and configuration files; they can communicate with each other through well-defined channels. All containers are run by a single operating system kernel and therefore use fewer resources than a virtual machine.

Docker gained its popularity due to its impact on the software development and deployment. The following are the some of the main reasons for docker becoming popular:

1. **Portability:**Docker facilitates the developers in packaging their applications with all dependencies into a single lightweight containers. It facilities in ensuring the consistent performance across the different computing environments.
2. **Reproducibility:**Through encapsulating the applications with their dependencies within a container it ensures in software setups remaining consistent across the development, testing and production environments.
3. **Efficiency:**Docker through its container based architecture it optimizes the resource utilization. It allows the developers to run the multiple isolated applications on a single host system.
4. **Scalability:**Docker’s scalability features facilitated the developers in making easier of their applications handling at time of workloads increment.

**Key Components of Docker**

The following are the some of the key components of Docker:

* **Docker Engine:**It is a core part of docker, that handles the creation and management of containers.
* **Docker Image:**It is a read-only template that is used for creating containers, containing the application code and dependencies.
* **Docker Hub:**It is a cloud based repository that is used for finding and sharing the container images.
* **Dockerfile:**It is a script that containing instructions to build a docker image.
* **Docker Registry**: It is a storage distribution system for docker images, where you can store the images in both public and private modes.

**Docker architecture:**

The Docker client talks with the docker daemon which helps in building, running, and distributing the docker containers. The Docker client runs with the daemon on the same system or we can connect the Docker client with the Docker daemon remotely. With the help of REST API over a  UNIX socket or a network, the docker client and daemon interact with each other.

**What is Docker Daemon?**

Docker daemon manages all the services by communicating with other daemons. It manages docker objects such as images, containers, networks, and volumes with the help of the API requests of Docker.

**Docker Client**

With the help of the docker client, the docker users can interact with the docker. The docker command uses the Docker API. The Docker client can communicate with multiple daemons. When a docker client runs any docker command on the docker terminal then the terminal sends instructions to the daemon. The Docker daemon gets those instructions from the docker client withinside the shape of the command and REST API’s request.

The main objective of the docker client is to provide a way to direct the pull of images from the docker registry and run them on the docker host. The common commands which are used by clients are **docker build, docker pull,**and **docker run.**

**Docker Host**

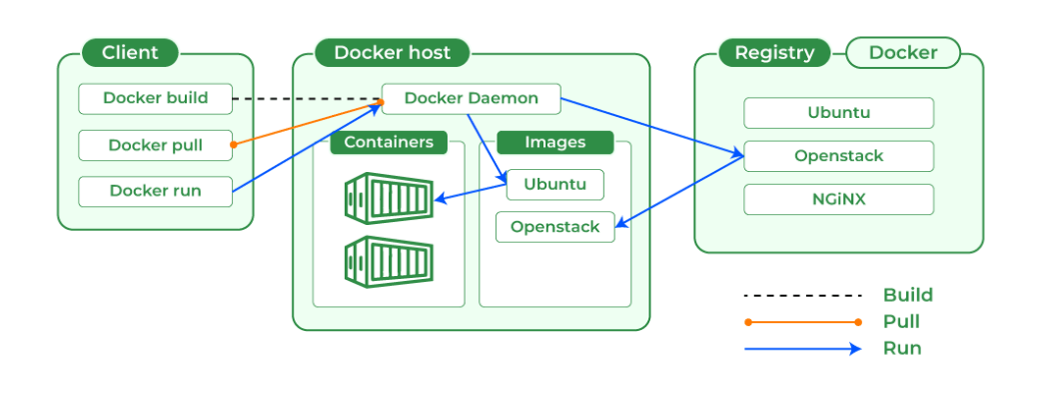
A Docker host is a type of machine that is responsible for running more than one container. It comprises the Docker daemon, Images, Containers, Networks, and Storage.

**Docker Registry**

All the docker images are stored in the docker registry. There is a public registry which is known as a  docker hubthat can be used by anyone. We can run our private registry also. With the help of **docker run**or**docker pull** commands, we can pull the required images from our configured registry. Images are pushed into configured registry with the help of the **docker push** command.

**Docker Objects**

Whenever we are using a docker, we are creating and use images, containers, volumes, networks, and other objects.



**Docker Images**

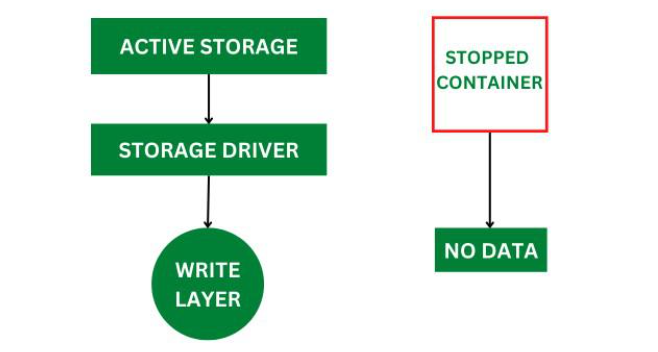
An image contains instructions for creating a docker container. It is just a **read-only template**. It is used to store and ship applications. Images are an important part of the docker experience as they enable collaboration between developers in any way which is not possible earlier.

**Docker Containers**

Containers are created from docker images as they are ready applications. With the help of Docker API or CLI, we can start, stop, delete, or move a container. A container can access only those resources which are defined in the image unless additional access is defined during the building of an image in the container.

**Docker Storage:**

We can store data within the writable layer of the container but it requires a storage driver. Storage driver controls and manages the images and containers on our docker host.



**Types of Docker Storage**

1. **Data Volumes:**Data Volumes can be mounted directly into the filesystem of the container and are essentially directories or files on the Docker Host filesystem.
2. **Volume Container**: In order to maintain the state of the containers (data) produced by the running container, Docker volumes file systems are mounted on Docker containers. independent container life cycle, the volumes are stored on the host. This makes it simple for users to exchange file systems among containers and backup data.
3. **Directory Mounts:** A host directory that is mounted as a volume in your container might be specified.
4. **Storage Plugins:**Docker volume plugins enable us to integrate the Docker containers with external volumes like Amazon EBS by this we can maintain the state of the container.

**Docker Networking :**

Docker Networking complete isolation for docker containers. It means a user can link a docker container to many networks. It requires very less OS instances to run the workload.

**Types of Docker Network**

1. Bridge: It is the default network driver. We can use this when different containers communicate with the same docker host.
2. Host: When you don’t need any isolation between the container and host then it is used.
3. Overlay: For communication with each other, it will enable the swarm services.
4. None: It disables all networking.
5. macvlan: This network assigns MAC(Media Access control) address to the containers which look like a physical address.

**Docker Supporting OS:**

1.Linux

2.Windows(Docker Desktop)

3.MacOS

Installation Method:

**Docker installation debian related os**

#sudo apt update

#sudo apt upgrade -y

#sudo apt install apt-transport-https ca-certificates curl software-properties-common -y

#curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

#echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

#sudo apt update

#sudo apt install docker-ce docker-ce-cli containerd.io -y

#docker --version

#sudo usermod -aG docker $USER

**Docker installation Redhat related OS**

#sudo yum update -y

#sudo yum install -y yum-utils

#sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo

#sudo yum install docker-ce docker-ce-cli containerd.io -y

#sudo systemctl start docker

#sudo systemctl enable docker

#docker --version

**2. Installation on Windows**

Windows 10/11 (Professional, Enterprise, Education)

Download Docker Desktop:https://docs.docker.com/desktop/setup/install/windows-install/

Go to Docker Desktop Downloads.

Install Docker Desktop:

Double-click the downloaded .exe file.

Follow the installation wizard and ensure WSL 2 is enabled.

Start Docker Desktop:

After installation, start Docker Desktop from the Start Menu.

Verify installation:

Open a terminal (e.g., PowerShell) and run:

#docker –version

**Docker Vs Virtualization:**

| **Docker** | **Virtualization** |
| --- | --- |
| It boots in a few seconds. | It takes a few minutes for VMs to boot. |
| Pre-built docker containers are readily available. | Ready-made VMs are challenging to find. |
| Docker has a complex usage mechanism consisting of both third-party and docker-managed tools. | Tools are easy to use and more straightforward to work with. third-party. |
| Limited to Linux. | Can run a variety of guest OS. |
| Dockers make use of the execution engine. | VMs make use of the hypervisor. |
| It is lightweight. | It is heavyweight. |
| Host OS can be different from container OS. | Host OS can be different from guest OS. |
| Can run many docker containers on a laptop. | Cannot run more than a couple of VMS  on an average laptop. |
| Docker can get a virtual network adapter. It can have separate IPs ad Ports. | Each VMS gets its virtual network adapter. |
| Sharing of files is possible. | Sharing library and files are not possible. |
| Lacks security measures. | Security depends on the hypervisor. |
| A container is portable. | VMS is dependent on a hypervisor. |
| It allows running an application in an isolated environment known as a container | It provides easiness in managing applications, recovery mechanisms, and isolation from the host operating system |

Docker file deploy war file

FROM tomcat:9.0-jdk17

# Expose port 8080 to allow external access to the application

EXPOSE 8080

# Set the working directory for the application

WORKDIR /usr/local/tomcat/webapps

# Copy the WAR file into the webapps directory of Tomcat

COPY PatientAppoinment-0.0.1-SNAPSHOT.war ./ROOT.war

# Start the Tomcat server

CMD ["catalina.sh", "run"]

Create the image through below command

#docker build -t(tagname) patientapp(image\_name) .

Deploy the image through below command

#docker run -d –name patientapp(containername) -p 8080(hostport):8080(containerport) patientapp(imagenam)

Go inside of the container through root privilege

#docker exec -it -u 0 <container\_name> bash

Reference : <https://docs.docker.com/reference/cli/docker/container/run/>

**Docker commands:**

**Pull the image form dockerhub**

1.docker pull

Some images need docker hub authentication in that case use command

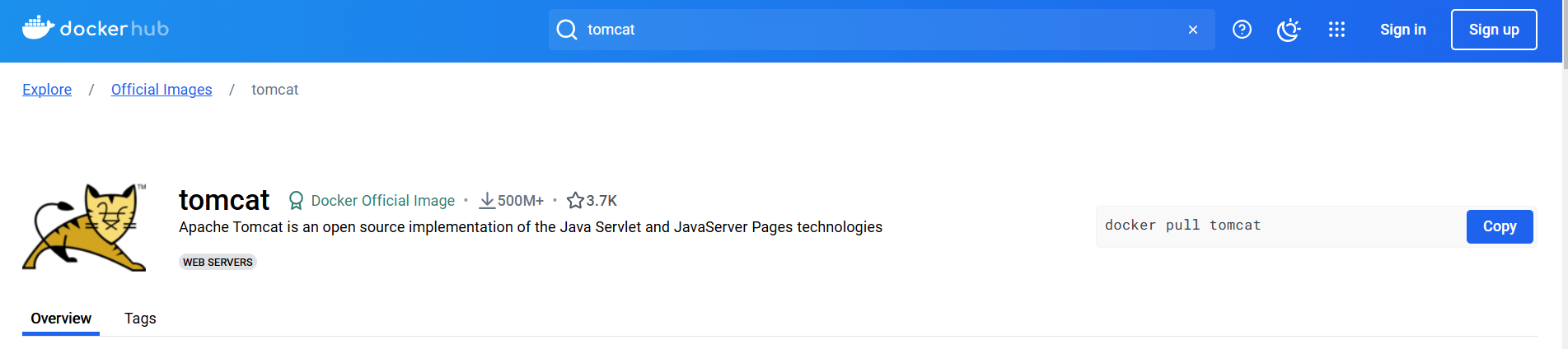
Go to 🡪 docker hub 🡪 Signup 🡪 Create your account

#docker login

Username:

Password:

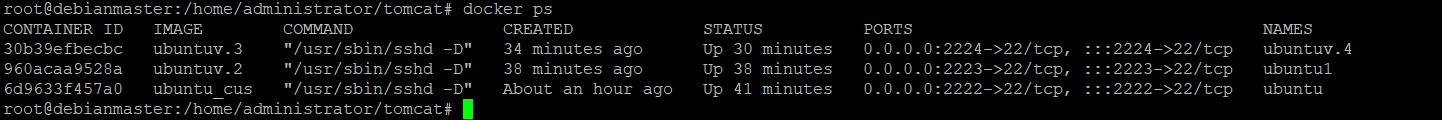
#docker pull tomcat:latest



2.docker ps

This command (by default) shows us a list of all the running containers. We can use various flags with it.

* **-a flag:** shows us all the containers, stopped or running.
* **-l flag:**shows us the latest container.
* **-q flag**: shows only the Id of the containers.



3.docker stop

This command allows you to stop a container if it has crashed or you want to switch to another one.

#docker stop <container\_ID>

4.Docker Start

Suppose you want to start the stopped container again, you can do it with the help of this command.

#docker start <container\_ID>

5.Docker rm

To delete a container. By default when a container is created, it gets an ID as well as an imaginary name such as tomcat,ubuntu etc. You can either mention the container name or its ID.

Some important flags:

* **-f flag:**remove the container forcefully.
* **-v flag:**remove the volumes.
* **-l flag:**remove the specific link mentioned

#docker rm {options} <container\_name or ID>

6. **Docker RMI**

To delete the image in docker. You can delete the images which are useless from the docker local storage so you can free up the space

#docker rmi <image ID/ image name>

7. **Docker Images**

Lists all the pulled images which are present in our system.

#docker images

8. **Docker exec**

This command allows us to run new commands in a running container. This command only works until the container is running, after the container restarts, this command does not restart.

Some important flags:

* **-d flag:**for running the commands in the background.
* **-i flag:**it will keep STDIN open even when not attached.
* **-e flag:**sets the environment variables

#docker exec -it <container\_name> bash

9. **Docker Ports (Port Mapping)**

In order to access the docker container from the outside world, we have to map the port on our host( Our laptop for example), to the port on the container. This is where port mapping comes into play.

#docker run -d -p <port\_on\_host> <image\_name>

10. **Docker Push**

Once you build your own customized image by using Dockerfile you need to store the image in the remote registry which is DockerHub for that you need to push your image by using the following command

#docker push <Image name/Image ID>

11. **Docker Build**

The docker build command is used to build the docker images with the help of Dockerfile.

#docker build -t image\_name:tag .

In the place of **image\_name** use the name of the image you build with and give the **tag number** and **. “dot”**represents the current directory.

12. **Stop Multiple Containers**

Instead of stopping a single container. You can stop multiple containers at a time by using the following commands.

#docker stop container1 container2 container3

13. **Docker Inspection**

Docker containers will run into some errors in real time to debug the container’s errors you can use the following commands.

#docker inspect container\_name\_or\_id

14. **Docker Commit command**

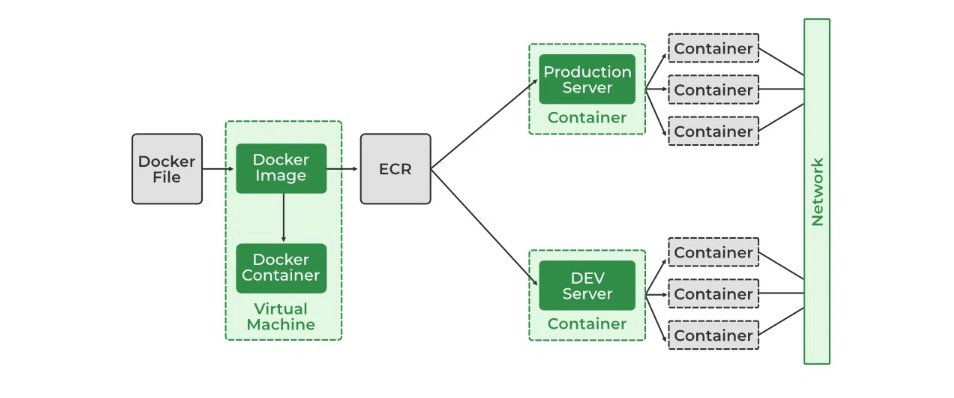
After running the containers by using the current image you can make the updates to the containers by interacting with the containers from that containers you can create an image by using the following commands.

#docker commit container\_name\_or\_id new\_image\_name:tag

Docker command reference: <https://www.cherryservers.com/blog/docker-commands-cheat-sheet>

**Docker Networking**

Docker Networking allows you to create a Network of Docker Containers managed by a master node called the manager. Containers inside the Docker Network can talk to each other by sharing packets of information. In this article, we will discuss some basic commands that would help you get started with Docker Networking.



**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

**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 containments 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.

### 1. **Bridge Network (Default)**

The bridge network is the default driver if you do not specify any other driver. It creates a private internal network on your host system and only allows containers connected to this bridge network to communicate with each other

#docker run -d --name mycontainer --network bridge nginx

**2.Host Network**

The host network mode allows the container to share the network namespace of the host. Containers in this mode have no network isolation and can use the host's IP address.

#docker run -d --name mycontainer --network host nginx

3.**None Network**

The none network mode removes the network interface for the container, meaning it cannot access any external network or communicate with other containers.

#docker run -d --name mycontainer --network none nginx

4. **Overlay Network**

An overlay network allows containers to communicate across multiple Docker hosts. This is especially useful in Docker Swarm mode where containers on different hosts need to communicate.

#docker network create -d overlay my\_overlay\_network

#docker service create --name myservice --network my\_overlay\_network nginx

5. **IPvlan Network**

The IPvlan driver allows users to assign IPv4 and IPv6 addresses to containers. This driver provides more control over container networking by using a virtual network interface (veth).

#docker network create -d ipvlan --subnet 192.168.1.0/24 --gateway 192.168.1.1 my\_ipvlan\_network

#docker run -d --name mycontainer --network my\_ipvlan\_network nginx

6. **Macvlan Network**

The macvlan driver allows you to assign a unique MAC address to a container, making it appear as a physical device on your network. This is useful for legacy applications or devices that require a specific MAC address.

#docker network create -d macvlan --subnet 192.168.2.0/24 --gateway 192.168.2.1 -o parent=eth0 my\_macvlan\_network

#docker run -d --name mycontainer --network my\_macvlan\_network nginx

**Docker volumes:**

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

**Key Points About Docker Volumes**

1. **Managed by Docker**: Docker handles the volume lifecycle, so they are easy to use and portable.
2. **Works with Multiple Containers**: Volumes can be shared and reused among multiple containers.
3. **Persistent Data**: Data in a volume persists even if the container is deleted.

By default docker volume stored in /var/lib/docker/volumes but we need to main the separate folder to maintain the docker volumes and if we can share the data through NFS.

Below are the steps to follow:

# Stop Docker service

sudo systemctl stop docker

# Create new directory for Docker volumes

sudo mkdir -p /opt/docker

# Modify Docker's data-root in the daemon.json configuration file

sudo bash -c 'cat <<EOF > /etc/docker/daemon.json

{

"data-root": "/opt/docker"

}

EOF'

# Start Docker service again

sudo systemctl start docker

Create the docker volume for tomcat application

#docker volume create tomcat

#docker run -d --name patientapp -p 8080:8080 -v tomcat:/usr/local/tomcat patientapp

But read only need

#docker run -d --name patientapp -p 8080:8080 -v tomcat:/usr/local/tomcat:ro patientapp

Bind Mount:

When you use a bind mount, a file or directory on the host machine is mounted from the host into a container.

#docker run -d --name patientapp -p 8081:8080 -v tomcat:/usr/local/tomcat(docker volume) -v /opt/tomcat:/opt/webapps(bind mount) patientapp5

**Docker Compose:**

Docker Compose is a tool for defining and running multi-container applications. It is the key to unlocking a streamlined and efficient development and deployment experience.

Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file. Then, with a single command, you create and start all the services from your configuration file.

Compose works in all environments; production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

* Start, stop, and rebuild services
* View the status of running services
* Stream the log output of running services
* Run a one-off command on a service

**Docker compose installation:**

#sudo apt-get update

#sudo apt-get install docker-compose-plugin

#docker volume create nginx

#docker volume create nginx\_conf

#docker volume create postgres

#mkdir nginx\_docker

#cd nginx\_docker

#nano docker-compose.yaml

version: '3.8'

services:

nginx:

image: nginx:latest

container\_name: nginx

ports:

- "8085:80" # Map host port 8085 to container port 80

volumes:

- nginx:/usr/share/nginx/html # Bind the volume to serve content

restart: always

volumes:

nginx:

external: true # Use the existing volume

**Deploy multiple container**

services:

nginx:

image: nginx:latest

container\_name: nginx\_server

ports:

- "8086:80" # Map host port 8080 to container port 80

volumes:

- nginx:/usr/share/nginx/html # Mount Docker-managed volume for HTML files

- nginx\_conf:/etc/nginx/conf.d # Mount Docker-managed volume for Nginx configuration

depends\_on:

- postgres # Ensure Nginx starts after PostgreSQL

restart: always

postgres:

image: postgres:latest

container\_name: postgres\_db

environment:

POSTGRES\_USER: myuser

POSTGRES\_PASSWORD: mypassword

POSTGRES\_DB: mydatabase

volumes:

- postgres:/var/lib/postgresql/data # Mount Docker-managed volume for PostgreSQL data

ports:

- "5432:5432" # Expose PostgreSQL database

restart: always

volumes:

nginx:

external: true (modify from outside)

nginx\_conf:

external: true

postgres:

external: true