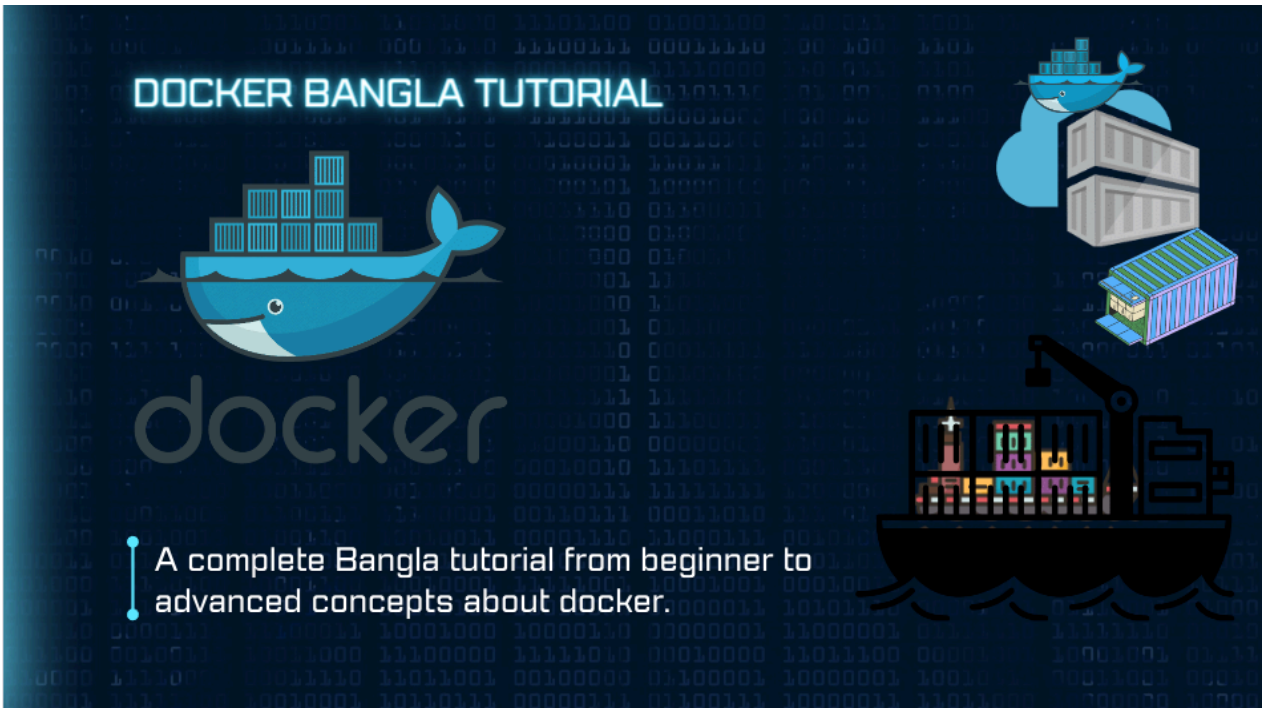
DOCKER BANGLA TUTORIAL



1) Introduction

Welcome to the Docker Bangla course. In this course, we'll start with the basics and take you all the way to advanced Docker concepts. By the end of this course, you'll get a better idea about Docker and can implement it in your working area.

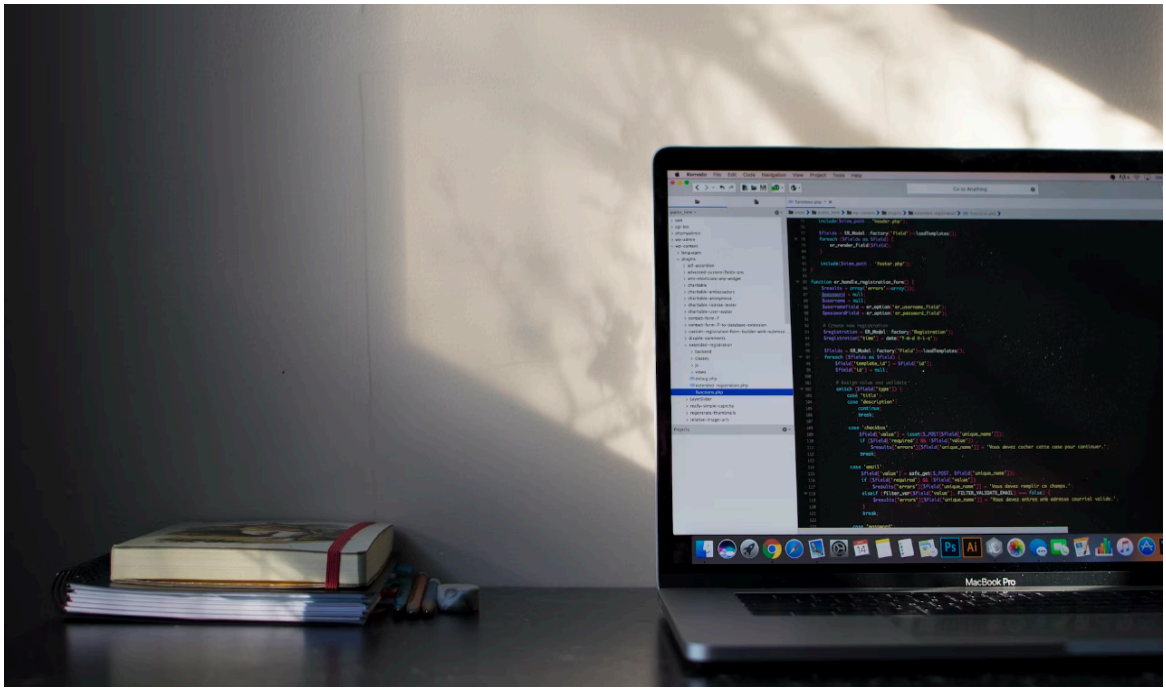
We'll begin with a basic and essential docker command and then move on to deploying a complete application, including the front-end, back-end, and database components, tailored to our project requirements, as if you can relate to real-life project.

Topic to be covered:

1) Introduction
2) Docker Installation
3) Getting Started with Dockers
4) Docker in Depth
5) Docker Image and Containers
6) Storage and Volumes
7) Networking and Security
8) Simple Example Project
9) Docker Compose
10) Container Orchestration

Prerequisites:

- Minimal IT background.
- Familiarity with software/app development and deployment, understanding concepts like frontend, backend, and databases is useful.
- Basic virtualization, Linux, or command-line understanding can be beneficial, but it's not mandatory.

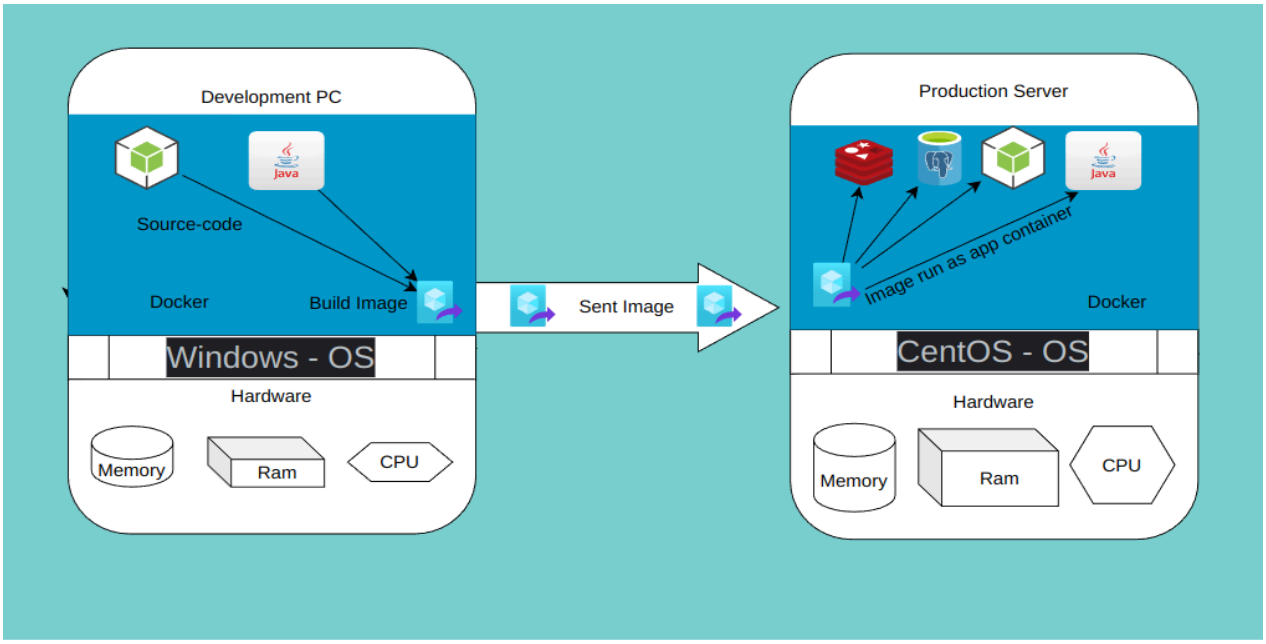


Which for this course:

- This course is suitable for DevOps professionals.
- Also required for Application developers, Software engineers, IT managers, system architects.
- Who have an interest in learning about Docker.

What is docker ?

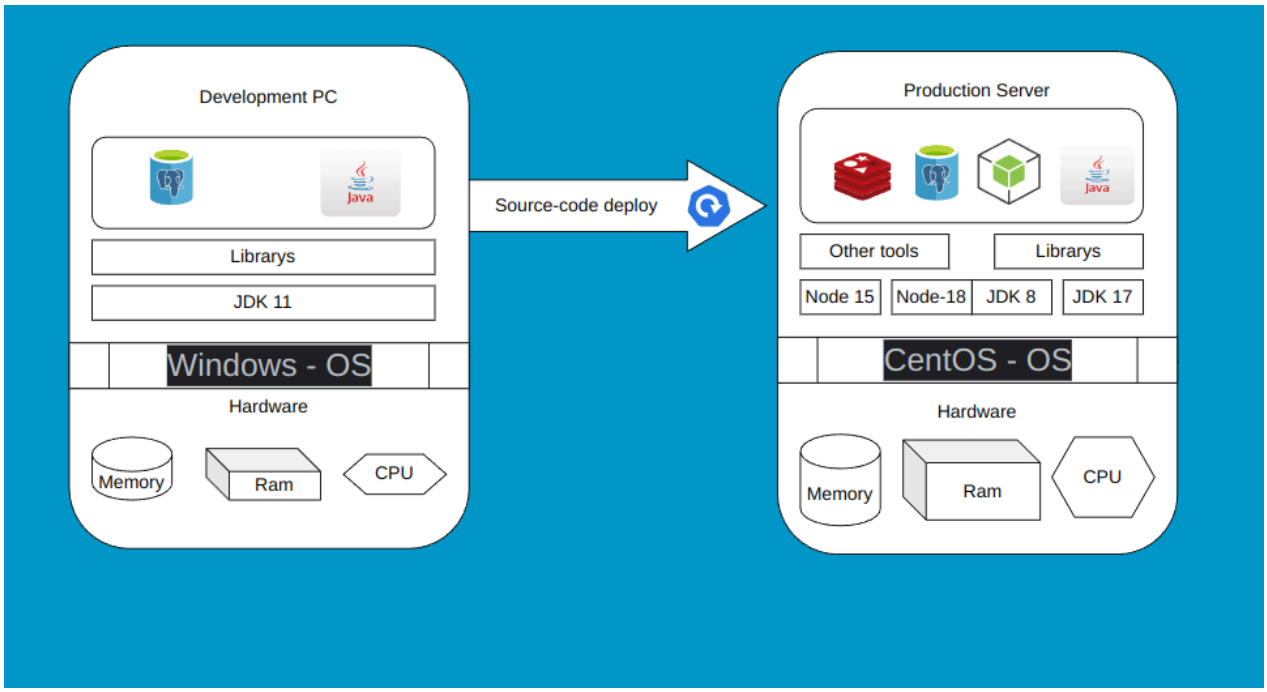
Docker is a platform for developing, shipping, and running applications in containers. Containers are lightweight, standalone, and executable packages that include everything needed to run a piece of software, including the code, runtime, libraries, and system tools.



What kind of problem solved by docker:

Problem Scenario: Imagine you are a developer working on a web application that requires a specific version of a web server, a database, and some libraries. You develop the application on your laptop, which runs a particular operating system and has its own set of software dependencies.

When you're ready to deploy your application to a production server, it might run a different operating system and have a distinct configuration. The application could encounter issues due to these differences.



Dependency Hell:

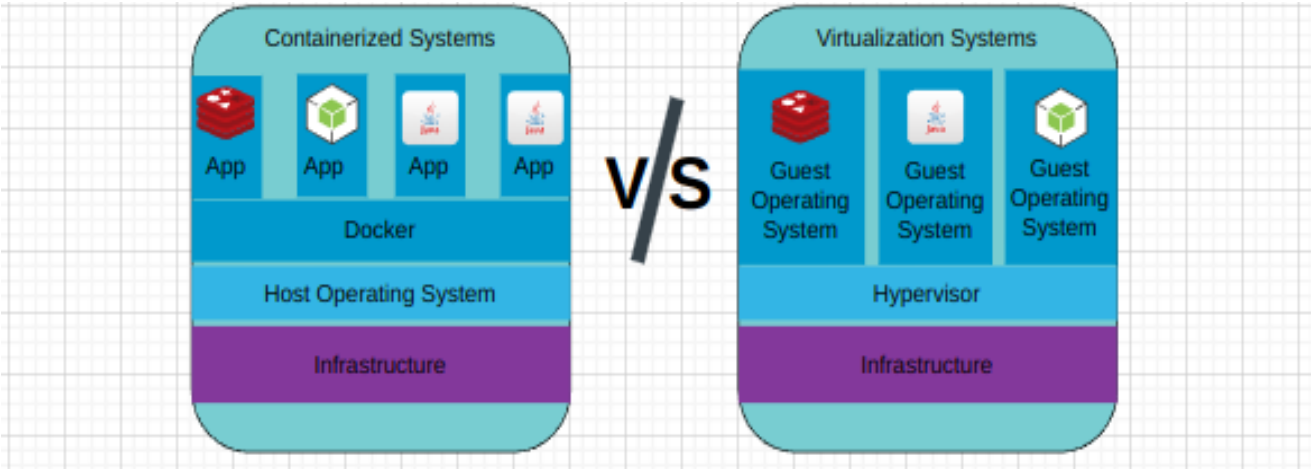
Applications might rely on specific versions of libraries, runtime environments, or configurations. When deploying on different systems, these dependencies could lead to compatibility problems, causing software to break.

Environment Consistency:

Development and production environments often differ, leading to "it works on my machine" issues. Developers might struggle to recreate the same environment as production during development, causing bugs to slip through.

Scalability and Resource Utilization:

Scaling applications up or down can be challenging. Virtual machines (VMs) were often used for isolation, but they're heavy and resource-intensive. Also application management.



2) Docker Installation

Install Docker on Windows

Install Docker on Linux

Docker Playground (If you can't install Docker on your system, you can also look into this online playground: https://labs.play-with-docker.com/)

Docker offers both free and paid versions of its containerization platform. You can download and use Docker Community Edition (CE) without any cost, is available for free and open source.

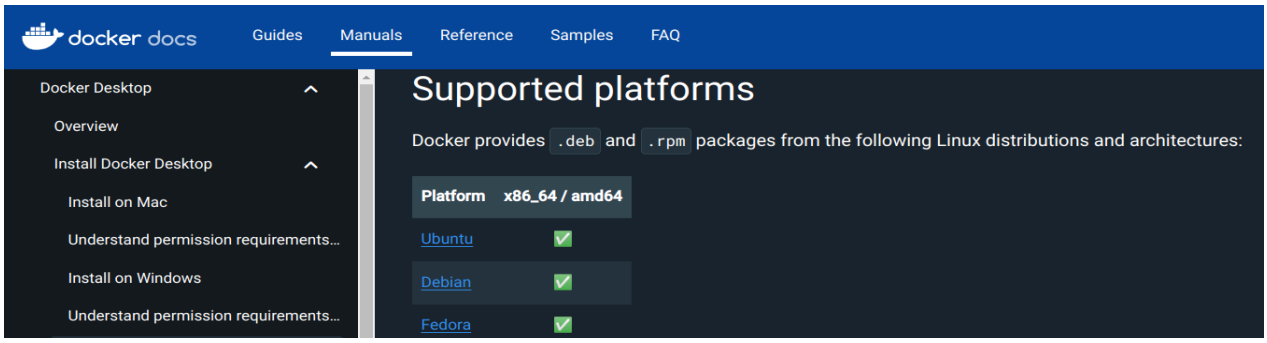
Docker paid version called Docker Enterprise, which offers additional features like container orchestration, security, and management tools, as well as official support and maintenance services.

Windows Install:
Docker Desktop Installer Download url: https://docs.docker.com/desktop/install/windows-install/

On Clicking this button automatically start downloading **docker-desktop.exe** file

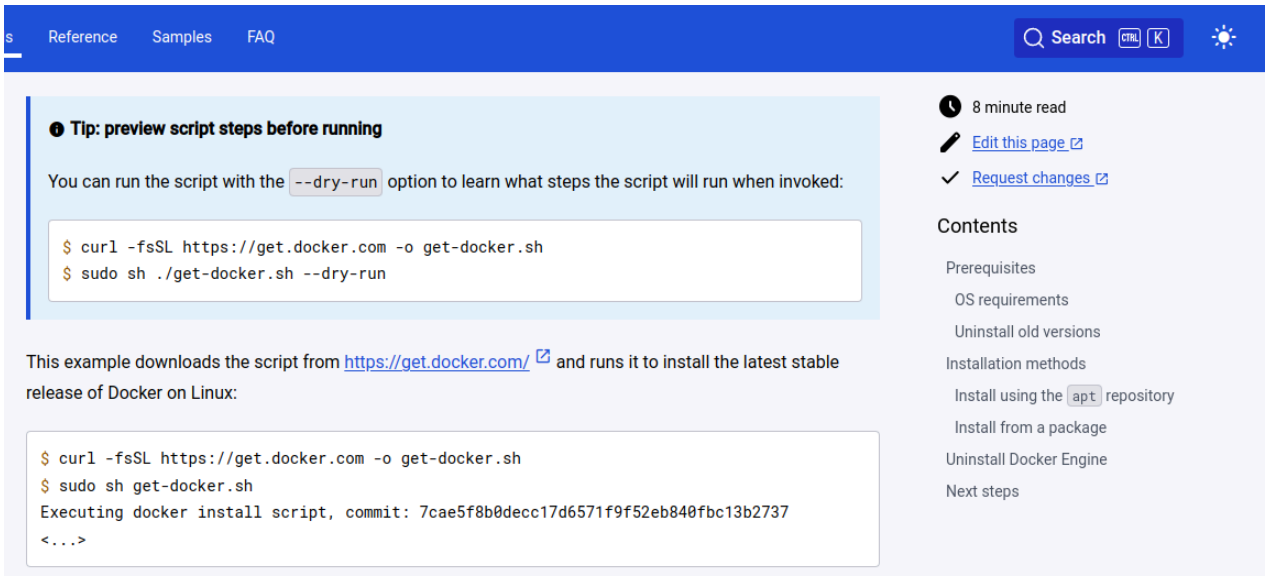
Linux Install:

Docker Desktop Installer Download url: https://docs.docker.com/desktop/install/linux-install/



For linux, flow the Instruction for your Linux distribution package.

Install using the convenience script: Page Link: https://docs.docker.com/engine/install/ubuntu/



3) Getting Started with Dockers: (Basic Docker command)

=>docker info

=>sudo systemctl status docker
=>sudo systemctl stop docker
=>sudo systemctl restart docker
For linux system

=>docker images ls
This command is used to list all the Docker images that in the docker

=>docker ps
Used to list currently running containers.

=>docker ps -a
As docker ps, but also including those that are not currently running.

=>docker container ls
Same as docker ps.

=>docker pull imageName
Download a Docker image from a registry.

=>docker rmi imageId
Remove a Docker image.

=>docker rm containerId
Remove a Docker container.

=>docker run imageName
Start a new container from an image.

=>docker start containerName
Start a stopped container.

=>docker stop containerId
Stop a running container.

=>docker restart containerId
Restart a container.

=>docker exec -it containerId someCommand
Run a command in a running container.

=>docker logs conteinerId
View container logs.

=>docker network ls
List Docker networks.

=>docker volume create MyDataVol
=>docker volume ls
Create and list Docker volumes.

=>docker login/logout
Log/Logout in to a Docker registry.

=>docker system prune
Remove unused data like stopped containers, dangling images, and more.

=>docker build -t imageName DockerfilePath
Build a Docker image from a Dockerfile.

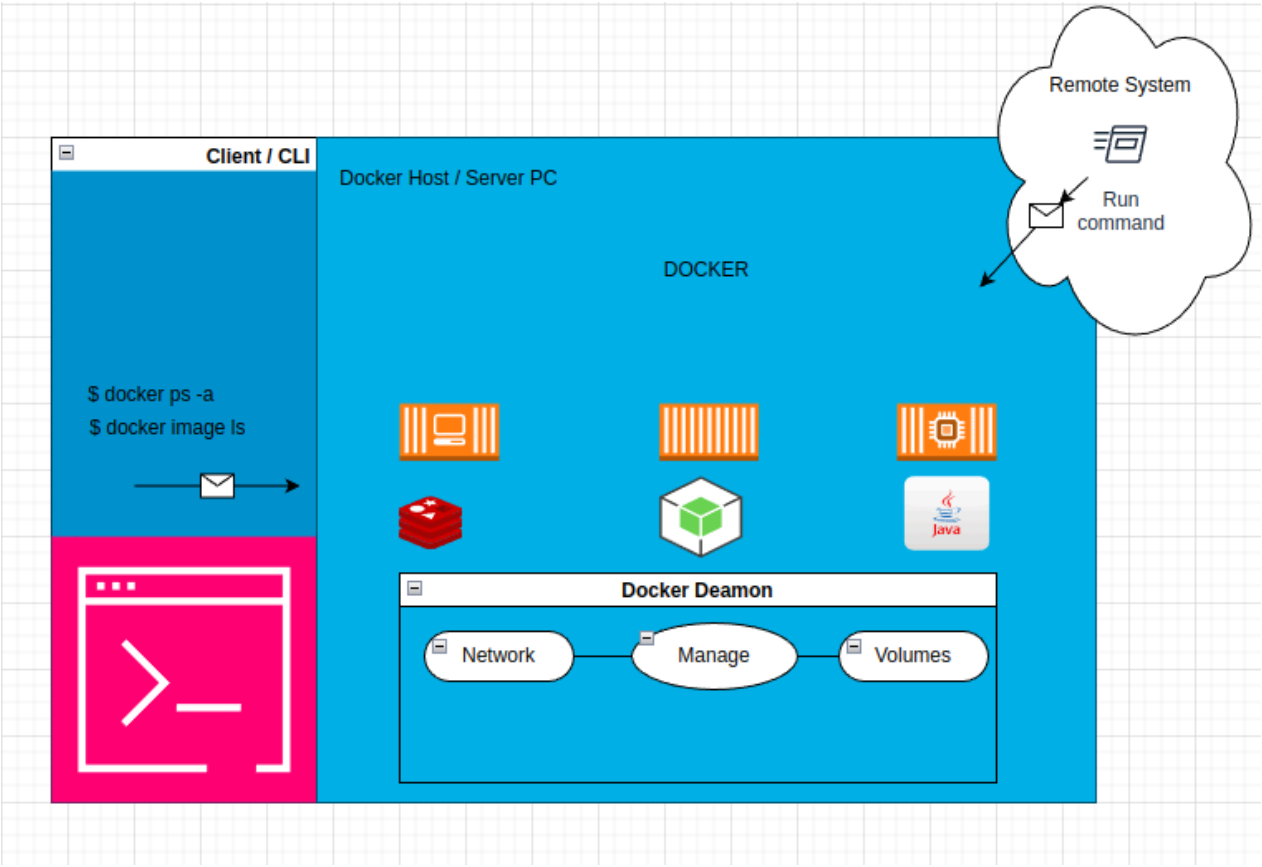
=>docker push imageName
Push a Docker image to a registry.

4) Docker-in-Depth:

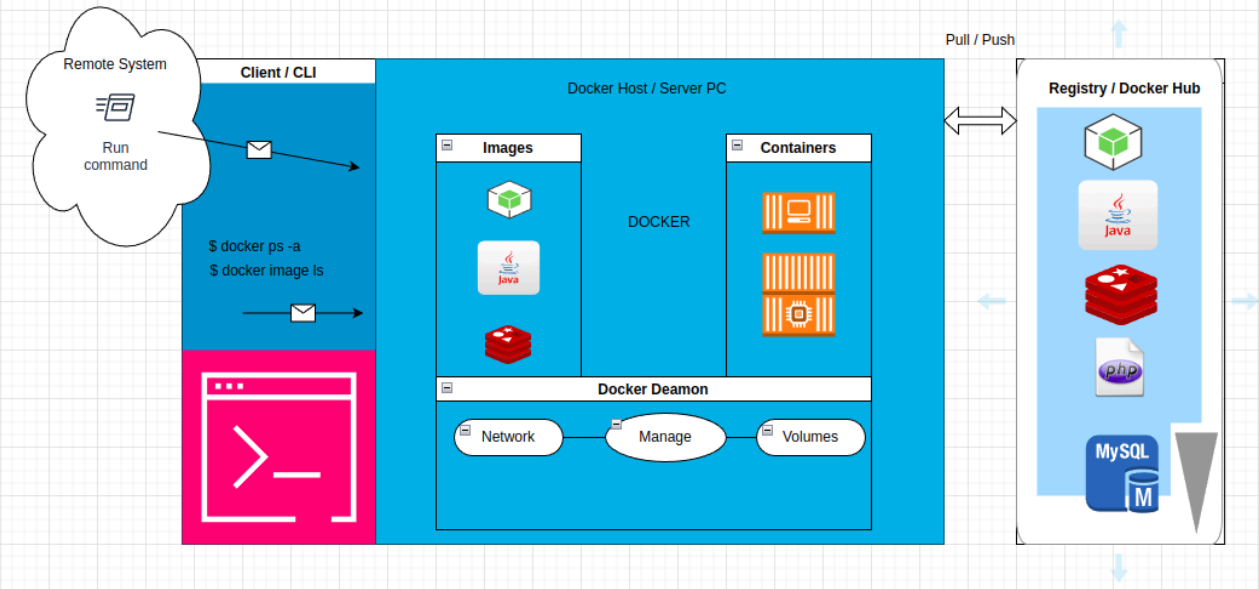
- Docker Arcture
- Docker Components
- Namespaces
- Control Groups (cGroups)

Docker Arcture:

Docker uses a client-server architecture. The Docker client communicates with the Docker daemon, which manages Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.



Docker Components: (Key components of Docker architecture)



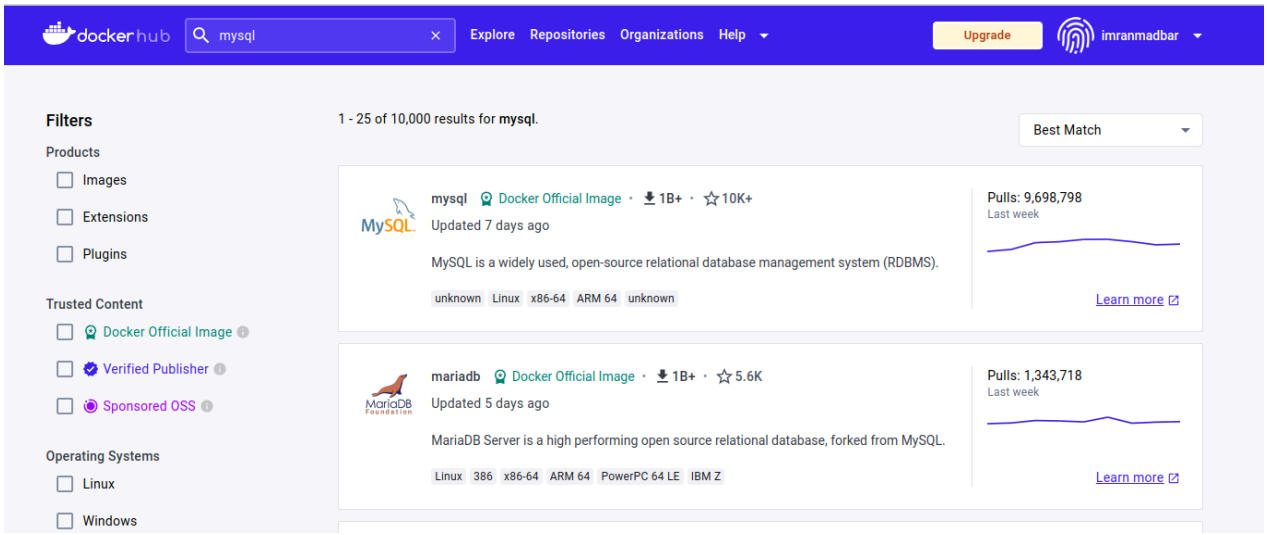
- **Docker Daemon:** The Docker daemon (dockerd) is a background process that manages Docker containers on a system. It handles container creation, running, and termination. The daemon listens for Docker API requests and can communicate with other Docker daemons to manage containers in a cluster.

- **Docker Client:** The Docker client (docker) is a command-line tool that allows users to interact with the Docker daemon. Users can use the Docker client to build, run, and manage Docker containers.

- **Docker Image:** A Docker image is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, libra;ries, and system tools. Images are often based on other images, forming a hierarchy.

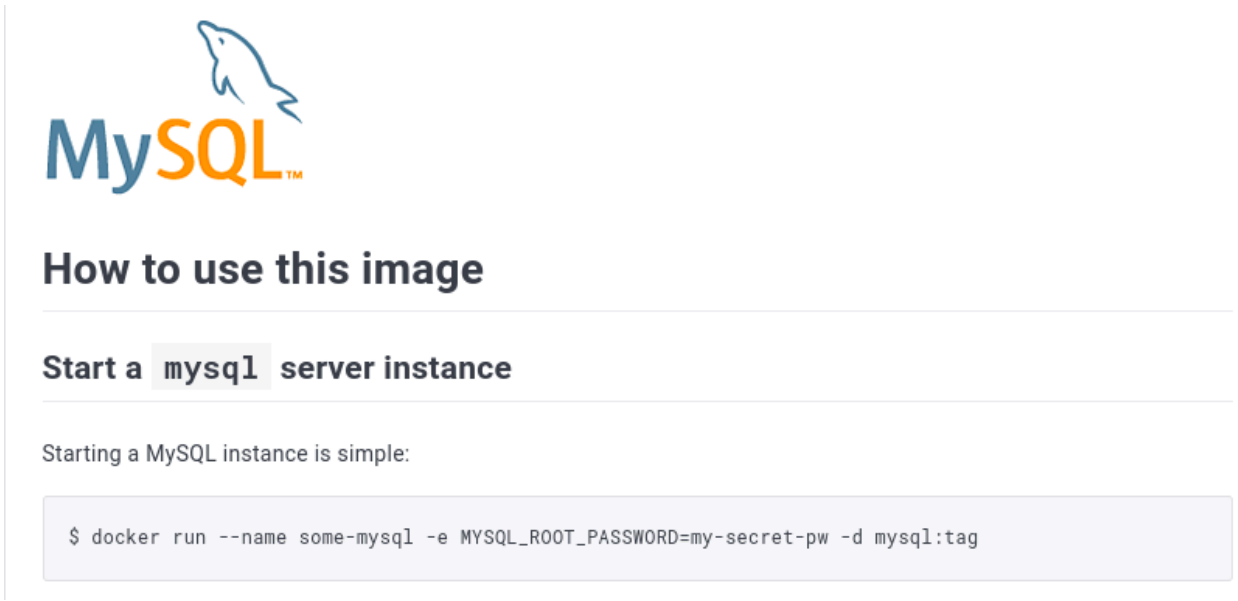
- **Docker Container:** A Docker container is a runnable instance of a Docker image. It encapsulates an application and its dependencies and runs in isolated environments. Containers share the host OS kernel but have their own file systems, processes, and network interfaces.

- **Docker Registry:** Docker images are stored in repositories, and these repositories are stored in Docker registries. A Docker registry stores and serves Docker images. Docker Hub is a public registry, but you can also set up private registries.

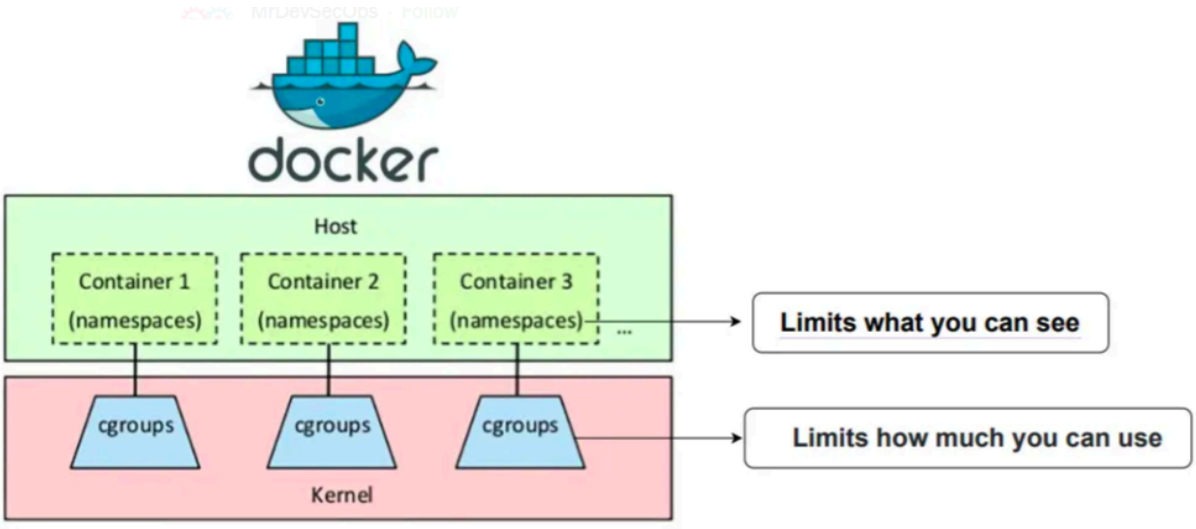


Link: https://hub.docker.com/

Instruction:



Need to konow two important thing: Namespaces and Control Groups (cGroups) are two crucial Linux kernel features that Docker leverages to provide containerization.



Namespaces provide isolation: Think of Namespaces as creating separate "worlds" for each container, ensuring they can't interfere with each other.

cGroups limit resources: Think of cGroups like giving containers their own "resource budget" to prevent them from consuming too much.

Namespaces:
Purpose: To provide isolation between containers, making each container think it has its own independent environment.

Example:
Consider two containers, X and Y, running on the same host. You want to make sure that processes in Container X can't interfere with processes in Container Y.

Using Namespaces, Docker creates separate environments for each container.
For instance, the PID Namespace ensures that the process IDs in Container X are different and isolated from the process IDs in Container Y.
If a process in Container X thinks it has Process ID 1, it won't conflict with a process in Container Y also thinking it has Process ID 1.

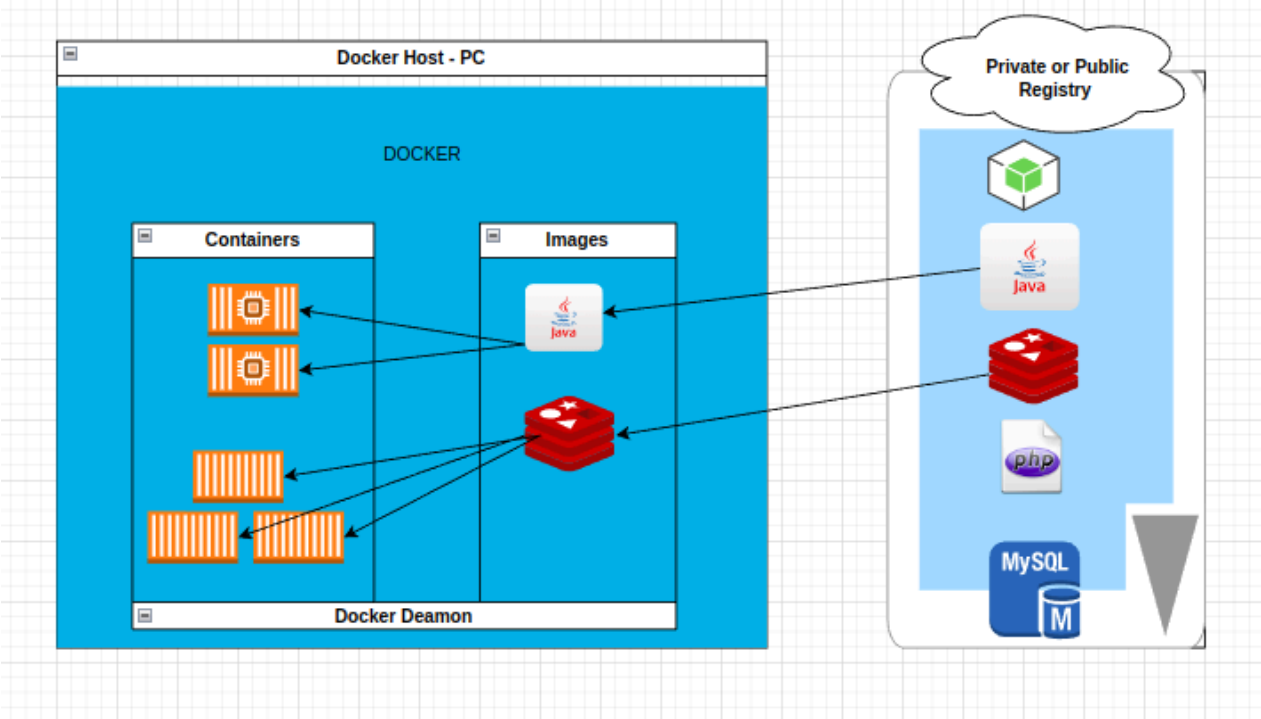
cGroups (Control Groups):
Purpose: To control and limit the resources that a container can use, such as CPU, memory, and more.

Example:
Suppose you have two containers, A and B, running on the same host. You want to ensure that Container A doesn't use more than 50% of the CPU, regardless of the overall system load.
Using cGroups, you can set up a rule that limits Container A to 50% of the CPU resources.
This ensures that even if the host machine has other processes running, Container A won't use more than its allocated share.

5) Docker Image and Containers:

Docker Images: Docker images are lightweight, standalone, executable packages that include everything needed to run a piece of software, including the code, runtime, libraries, and system tools.

We can create a image using a Dockerfile or Download Image from docker registry.



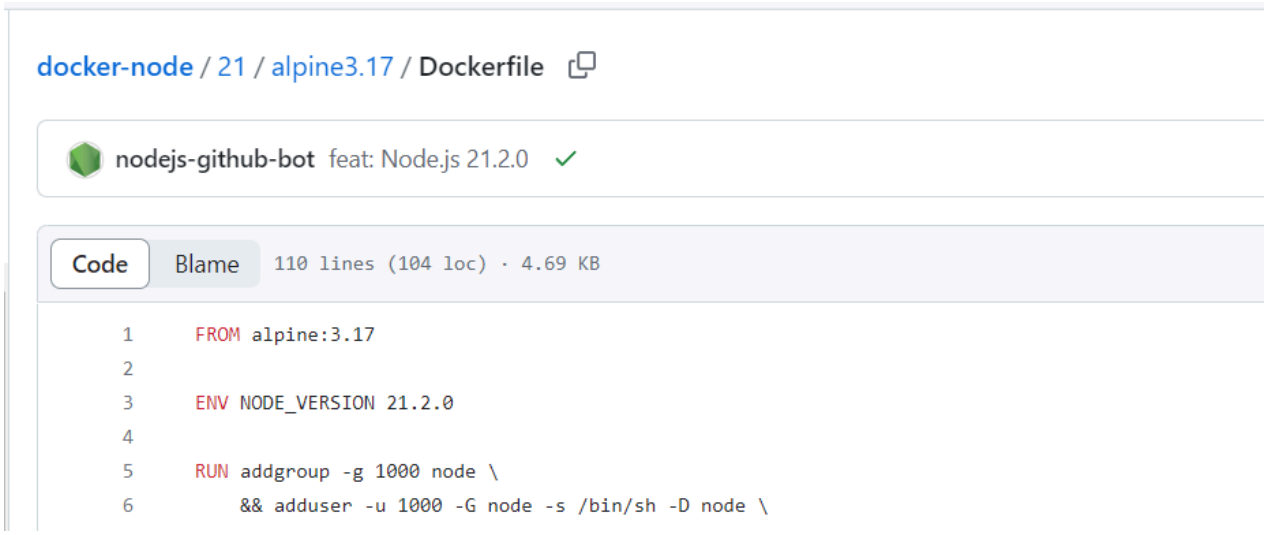
An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization.

Docker images are based on a layered architecture, Each layer in a Docker image represents a set of file changes or instructions in the Dockerfile used to build the image.

```
bs960@BS-960:~$ docker image ls
REPOSITORY                                          TAG       IMAGE ID       CREATED         SIZE
nginx                                               latest    a6bd71f48f68   7 days ago      187MB
mysql                                               latest    a3b6608898d6   5 weeks ago     596MB
ubuntu                                              latest    e4c58958181a   7 weeks ago     77.8MB
hello-world                                         latest    9c7a54a9a43c   6 months ago    13.3kB
imranmadbar/hello-world-java-maven-web-spring-boot  latest    b5fbb11e245f   21 months ago   124MB
imranmadbar/hello-world-java                        latest    a44f8f07a6f3   2 years ago     105MB
bs960@BS-960:~$
```

Base Image: Its the parent of all image.

Base images serve as the foundation for creating Docker containers. Using **“FROM”** directive specifies the base image.



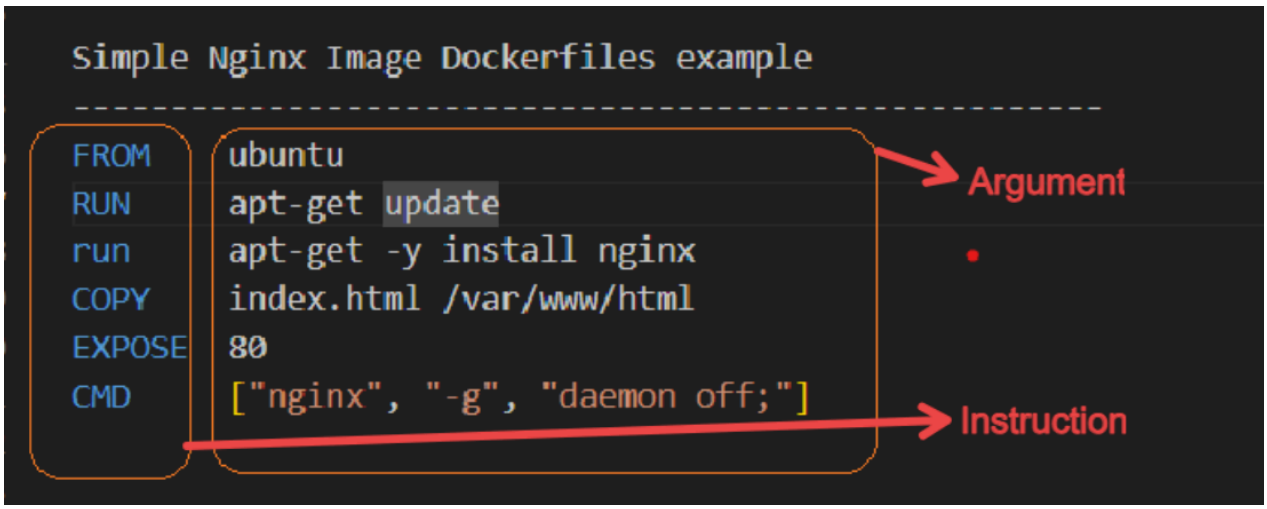
The subsequent commands in the Dockerfile then build on top of this base image by installing additional software, setting the working directory, and copying application code into the container.

“FROM scratch”

The scratch image is the most minimal image in Docker. This is the base line for all other images.

```
# syntax=docker/dockerfile:1
FROM scratch
ADD hello /
CMD ["/hello"]
```

Let's build a Docker image: using Dockerfile.
Dockerfiles are text documents that allow you to build images for Docker.



This Dockerfile performs the following actions:

- It starts from the base image **ubuntu**
- It updates the package
- It copies **index.html** file from current local directory into **/var/www/html** directory in the image.
- Expose container **port 80**
- It sets command to run the **nginx**

Docker Image Build command: **$docker build -t image-name .**

Here, **-t** specifies the image name and tag, and **.** indicates that the Dockerfile is in the current directory.

Another example using python:

```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

In the example above, each instruction creates one layer:

- `FROM` creates a layer from the `ubuntu:22.04` Docker image.
- `COPY` adds files from your Docker client's current directory.
- `RUN` builds your application with `make`.
- `CMD` specifies what command to run within the container.

When you build this Dockerfile using the docker build command, Each instruction in the Dockerfile adds a new layer to the image.
In this case, you would have layers corresponding to the **FROM, COPY**, **RUN,** and **CMD** instructions.

When you inspect the image, you'll see a section called "Layers" that lists the IDs of each layer.
For example: **$docker image inspect imageId**

Save Docker Image as file:
$docker save -o image.tar imageId
$docker load -i image.tar

Run a Docker Container from the Image:

Once the Docker image is built, you can run a container from it using the docker **run** command.

$docker run -d busybox sleep 100
$docker run -ti --rm ubuntu /bin/bash
$docker run --name simpleNginx -d -p8080:80 nginx

```
nas@NAS:/Docker$ sudo docker pull jellyfin/jellyfin
Using default tag: latest
latest: Pulling from jellyfin/jellyfin
a603fa5e3b41: Downloading [==================================================> ]  31.19MB/31.41MB
085d9da18665: Downloading [=========================================>          ]   224MB/278.9MB
421a38ac6f1f: Downloading [==================================================> ]  55.31MB/55.83MB
9787ba67dc71: Waiting
```

For List of Container:

$docker ps
$docker ps -a

```
Windows PowerShell
PS C:\Users\BS960> docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED         STATUS              PORTS                  NAMES
a1d64148dc88   imranmadbar/ubuntu   "/bin/bash system-mo…"   3 minutes ago   Up 3 minutes                               distracted_booth
71b297529e38   nginx                "/docker-entrypoint.…"   41 hours ago    Up About a minute   0.0.0.0:8181->80/tcp   ng-cont1
PS C:\Users\BS960>
PS C:\Users\BS960>
PS C:\Users\BS960> docker ps -a
CONTAINER ID   IMAGE                COMMAND                  CREATED         STATUS                      PORTS                  NAMES
a1d64148dc88   imranmadbar/ubuntu   "/bin/bash system-mo…"   3 minutes ago   Up 3 minutes                                       distracted_booth
6f6219c38dc3   nginx                "/docker-entrypoint.…"   41 hours ago    Exited (255) 18 hours ago   0.0.0.0:8283->80/tcp   ng-cont4
94a2f4108ee3   nginx                "/docker-entrypoint.…"   41 hours ago    Exited (255) 18 hours ago   0.0.0.0:8282->80/tcp   ng-cont3
71b297529e38   nginx                "/docker-entrypoint.…"   41 hours ago    Up About a minute           0.0.0.0:8181->80/tcp   ng-cont1
PS C:\Users\BS960>
```

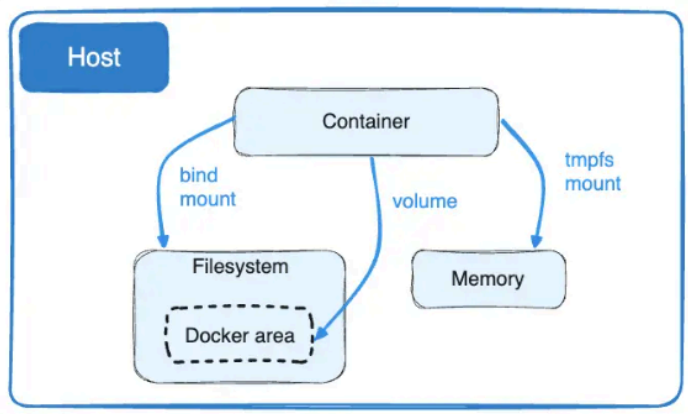
$docker containe inspect containerId

```
Windows PowerShell
PS C:\Users\BS960> docker inspect a1d64148dc88
[
    {
        "Id": "a1d64148dc88afbc7b0af53340b6e4de76d5158befb71eaf4954b1e064dc369f",
        "Created": "2023-11-29T10:49:52.914589514Z",
        "Path": "/bin/bash",
        "Args": [
            "system-monitor.sh"
        ],
        "State": {
            "Status": "exited",
            "Running": false,
            "Paused": false,
            "Restarting": false,
```

6) Storage and Volumes: By default all files created inside a container are stored on a writable container layer.

- The data doesn't persist when that container no longer exists.
- Tightly coupled to the host machine.
- Reduces performance.

To managing and persisting data in containers. Two way solved this problem: **Volume, Bind-mount.**

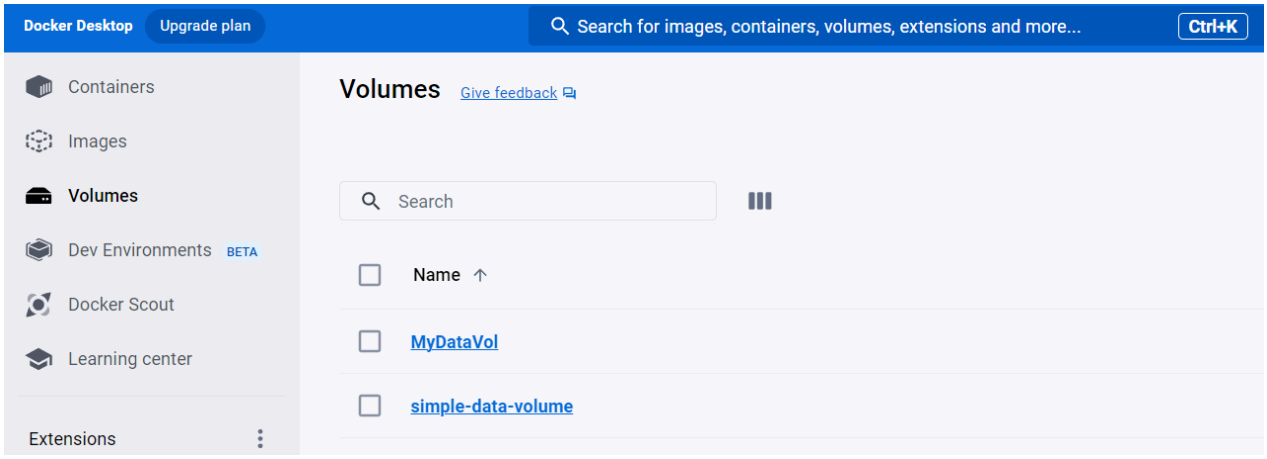


- Volumes are stored in a part of the host filesystem which is *managed by Docker* (`/var/lib/docker/volumes/` on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
- Bind mounts may be stored anywhere on the host system. They 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.

Storage drivers:

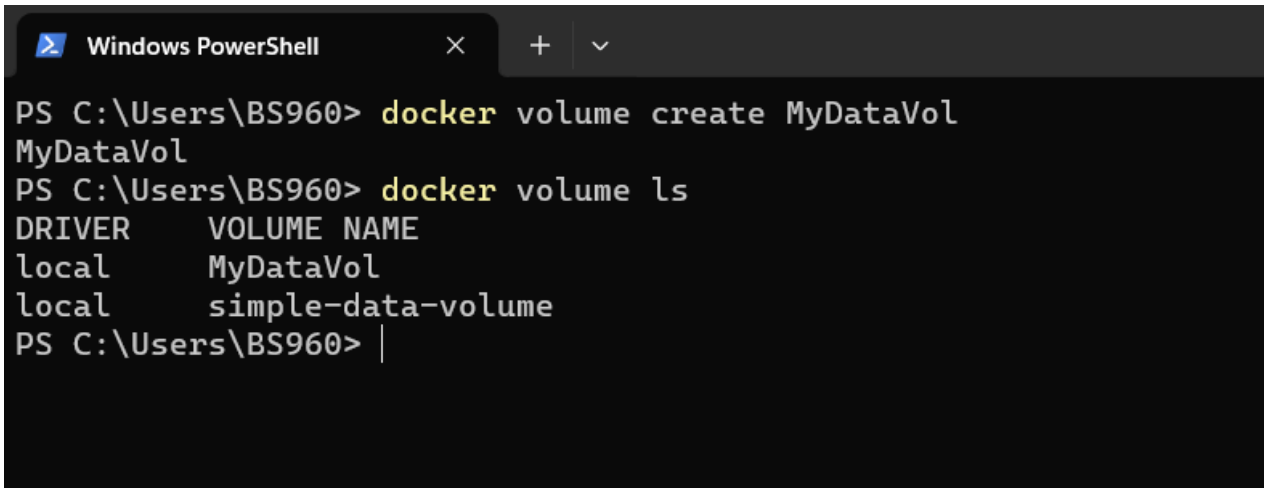
Docker uses storage drivers to store image layers, and to store data in the writable layer of a container.

Volumes: Created and managed by Docker. You can create a volume explicitly using the docker volume create command, or Docker can create a volume during container or service creation.



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.

- **Creation:** `docker volume create MyDataVol`
- **Listing all volumes:** `docker volume ls`
- **Listing volumes with a filter:** `docker volume ls -f name=data`
- **Inspecting a specific volume:** `docker volume inspect MyDataVol`
- **Removing a specific volume:** `docker volume rm MyDataVol`
- **Pruning unused volumes:** `docker volume prune`



Lets Make volume and tie it with a container:

$docker run --name my-ng -d -p 8080:80 -v nginx-data-vol:/usr/share/nginx/html nginx

Bind mounts: When you use a bind mount, a file or directory on the host machine is mounted into a container. Sharing data / files from the host machine to containers.

$docker run --name my-ng -d -p 8080:80 -v /opt/tempFile:/usr/share/nginx/html nginx
$docker run --name my-ng -d -p 8080:80 -v C:\myFile\tempFile:/usr/share/nginx/html nginx

Alternative: **--mount** option in Docker is an alternative to the **-v** to provides a more explicit way.

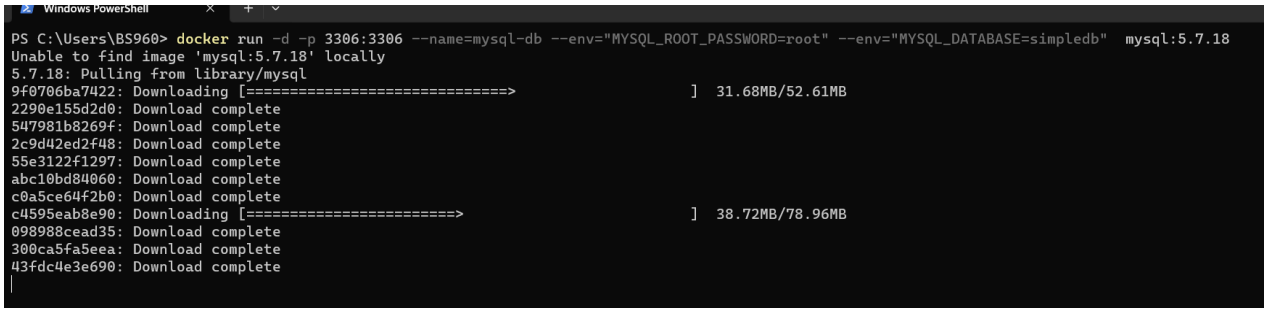
$docker run --name myng -d -p 8080:80 --mount type=volume,source=myvolume, target=/usr/share/nginx/html nginx

And

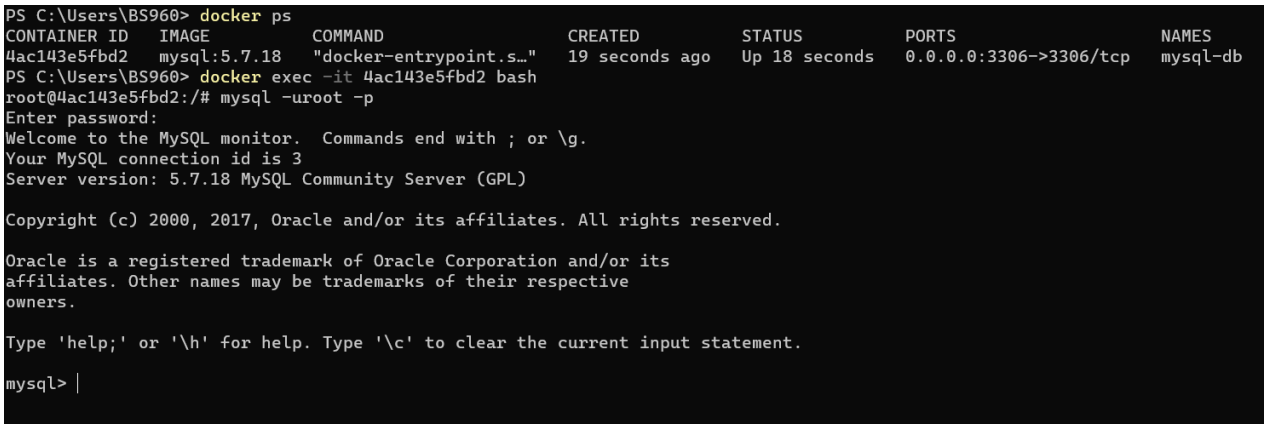
$docker run --name myng -d -p 8080:80 --mount type=bind,source=/opt/tempFile, target=/usr/share/nginx/html nginx

MySql example:

$docker run -d -p 3306:3306 --name=mysql-db --env="MYSQL_ROOT_PASSWORD=root" --env="MYSQL_DATABASE=simpledb" mysql:5.7.18



Vomume / bind-mounting with MySQL:



$docker run -d -p 3306:3306 --name=mysql-db --env="MYSQL_ROOT_PASSWORD=root" --env="MYSQL_DATABASE=simpledb" mysql:5.7.18
$docker ps
$docker exec -it containerId bash

$mysql -uroot -p

=>CREATE TABLE book (id INT PRIMARY KEY, name VARCHAR(255) NOT NULL);
=>INSERT INTO book (id, name) VALUES (1, 'The Great book'), (2, 'My Java'), (3, 'Python Book');
=>SELECT * FROM book;

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| simpledb           |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql> use simpledb;
Database changed
mysql> show tables;
Empty set (0.00 sec)

mysql> CREATE TABLE book (id INT PRIMARY KEY, name VARCHAR(255) NOT NULL);
Query OK, 0 rows affected (0.03 sec)

mysql> INSERT INTO book (id, name) VALUES (1, 'The Great book'), (2, 'My Java'), (3, 'Python Book');
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM book;
+----+----------------+
| id | name           |
+----+----------------+
|  1 | The Great book |
|  2 | My Java        |
|  3 | Python Book    |
+----+----------------+
3 rows in set (0.00 sec)

mysql>
```

Volume and Bind-mount command:

Volume Mount:

$docker run --name mysqldb -d -e MYSQL_ROOT_PASSWORD=root --env="MYSQL_DATABASE=simpledb" -v mysql_data:/var/lib/mysql mysql:8.2.0

Bind-Mount:

$docker run --name mysqldb -d -e MYSQL_ROOT_PASSWORD=root --env="MYSQL_DATABASE=simpledb" -v C:\Z_MY_COMPUTER\yt\DOCKER\resource-file\storage-volumes:/var/lib/mysql mysql:8.2.0

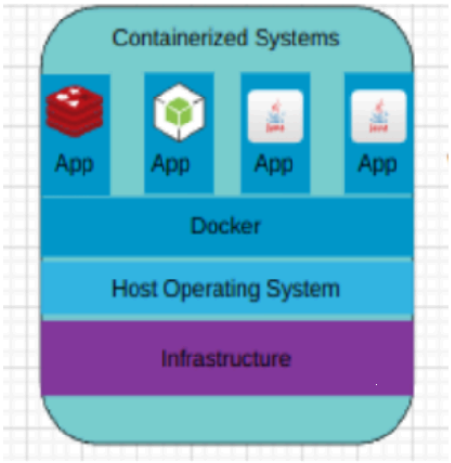
$docker run --name mysqldb -d -e MYSQL_ROOT_PASSWORD=root --env="MYSQL_DATABASE=simpledb" --mount type=bind,src=C:/Z_MY_COMPUTER/yt/DOCKER/resource-file/storage-volumes/mysql-vol,target=/var/lib/mysql mysql:8.2.0

```
PS C:\Users\BS960> docker run --name mysqldb -d -e MYSQL_ROOT_PASSWORD=root --env="MYSQL_DATABASE=simpledb" -v C:\Z_MY_COMPUTER\yt\DOCKER\resource-file\stor
age-volumes:/var/lib/mysql mysql
7741d5b154f756e9791de20a7abbb0b2fda942df3e560aeccda4834d34b3b3e2
PS C:\Users\BS960> docker exec -it 774 bash
bash-4.4# mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.2.0 MySQL Community Server - GPL

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| simpledb           |
| sys                |
+--------------------+
5 rows in set (0.01 sec)

mysql> use simpledb;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+--------------------+
| Tables_in_simpledb |
+--------------------+
```

7) Networking and Security:

Docker provides various networking options to facilitate communication between containers, external world and the security.

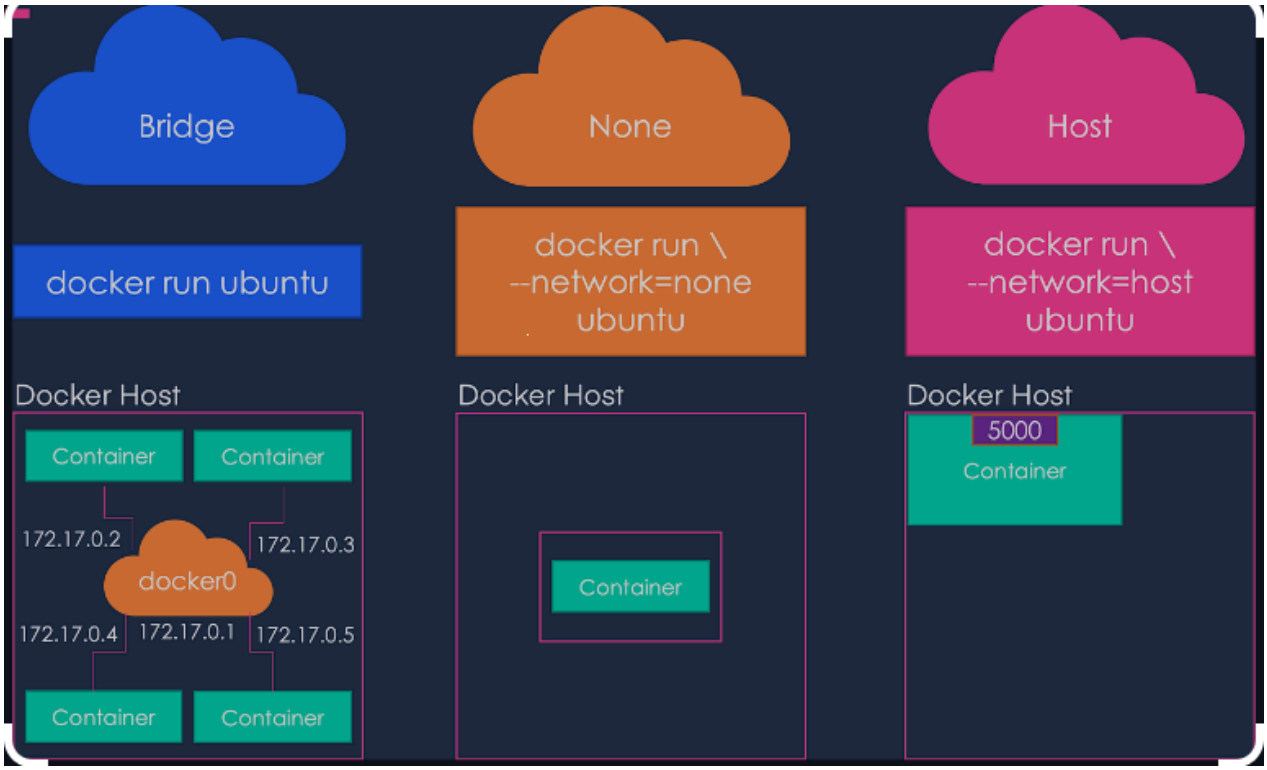
1. `docker network ls`
   - Lists all networks on the host.
2. `docker inspect bridge`
   - Displays details about the default bridge network.
3. `docker run --network`
   - Connects a container to a specific network during creation.
4. `docker network connect`
   - Connects an existing container to a network.
5. `docker network disconnect`
   - Disconnects a container from a network.
6. `docker exec 484bc9a71fea ifconfig`
   - Shows network configuration inside a running container.
7. `docker network inspect netName`
   - Displays detailed information about a network.
8. `docker network rm`
   - Removes a custom network.
9. `docker network prune`
   - Removes unused networks on the host.




```
PS C:\Users\BS960> docker inspect bridge
[
    {
        "Name": "bridge",
        "Id": "47dbe964b50bc35888eb68353d4502b93d31f9308537ce32ab231b34845dac14",
        "Created": "2023-12-01T09:08:23.819203685Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
```

Some common networking options include:

```
PS C:\Users\BS960> docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
8c6920e039fc   bridge    bridge    local
ce3d6a66b8ad   host      host      local
9df8d264fba9   none      null      local
```



Bridge: The default network mode in Docker, which creates a private internal network on the host system.

```
PS C:\Users\BS960> docker inspect bridge
[
    {
        "Name": "bridge",
        "Id": "47dbe964b50bc35888eb68353d4502b93d31f9308537ce32ab231b34845dac14",
        "Created": "2023-12-01T09:08:23.819203685Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
```

Create a bridge network:

$docker network create my_net
$docker network ls
$docker inspect netIdORname
$docker run --name=my-ng-service --network=my_net -d nginx

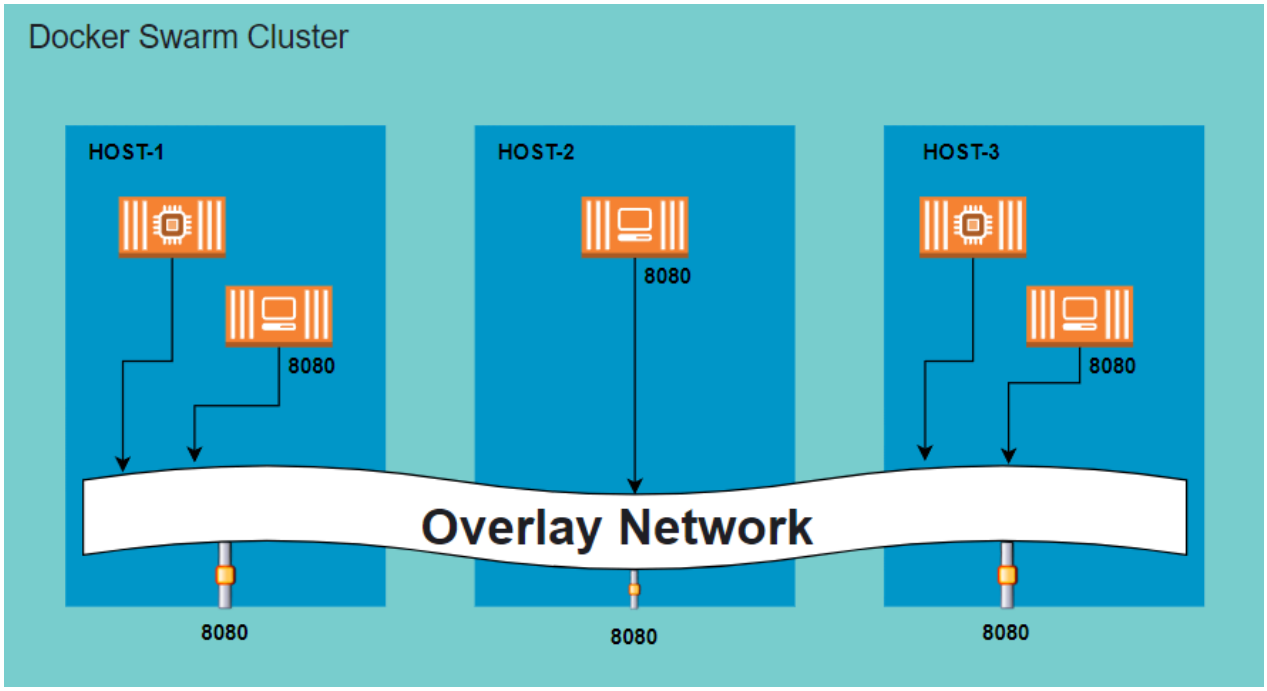
```
PS C:\Users\BS960> docker network create my-net
5a125a19fa46a7a32a788dbddd3a0205aa25c3d8813d1f8b843a31afd65ce875
PS C:\Users\BS960> docker inspect 5a125a19fa46a7a32a788dbddd3a0205aa25c3d8813d1f8b843a31afd65ce875
[
    {
        "Name": "my-net",
        "Id": "5a125a19fa46a7a32a788dbddd3a0205aa25c3d8813d1f8b843a31afd65ce875",
        "Created": "2023-12-01T10:36:18.935133503Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
```

Host: Containers share the host system's network stack.

> **Note**
>
> Given that the container does not have its own IP-address when using `host` mode networking, port-mapping doesn't take effect, and the `-p`, `--publish`, `-P`, and `--publish-all` option are ignored, producing a warning instead:
>
> ```
> WARNING: Published ports are discarded when using host network mode
> ```

Host networking is almost never necessary. In some very unusual cases -- if your service has thousands of ports it listens on, if you've measured the Docker NAT overhead to be significant with very-high-volume traffic -- it can get around some limitations of the Docker networking system. In almost all practical cases you should use the default (bridged) networking, and publish specific ports if you need them to be accessible from outside Docker space.

Overlay: Used for connecting containers across multiple Docker hosts. It's commonly used in Docker Swarm clusters.



Custom User-Defined Networks: You can create custom user-defined networks to group containers with specific requirements.

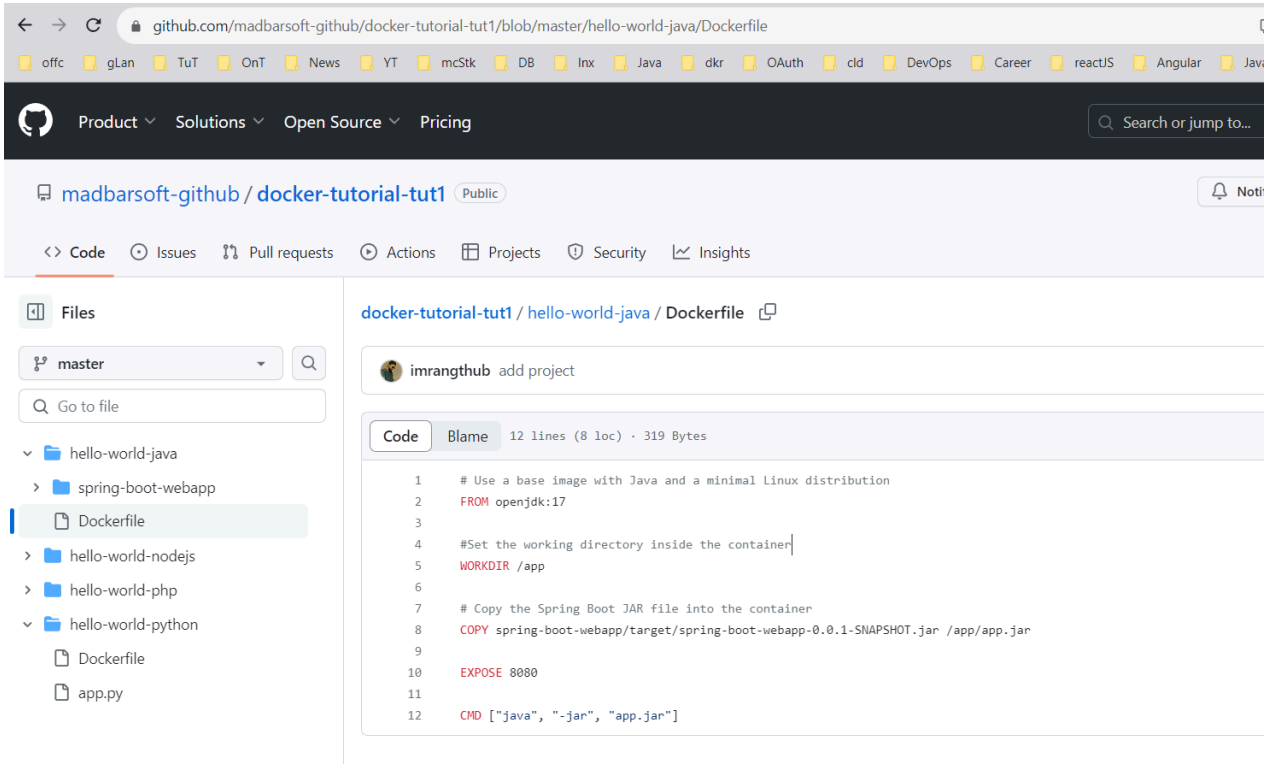
```
=>docker network create --driver bridge --subnet 182.18.0.1/24 --gateway 182.18.0.1 wp-mysql-network
Create a new network named wp-mysql-network using the bridge driver. Allocate subnet 182.18.0.1/24. Configure Gateway 182.18.0.1
```

$docker network create --subnet=192.168.1.0/24 --gateway 192.168.1.1 my_bridge_net
$docker run --name=my-ng-service --network=my_bridge_net -d nginx

```
PS C:\Users\BS960> docker inspect my_bridge_network2
[
    {
        "Name": "my_bridge_network2",
        "Id": "716a59055f6a3880d8a019361635b2d57eff3b60d27115980c54be1ae0e8dec9
        "Created": "2023-12-01T09:25:05.318167849Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "192.168.2.0/24",
                    "Gateway": "192.168.2.1"
                }
            ]
        },
```

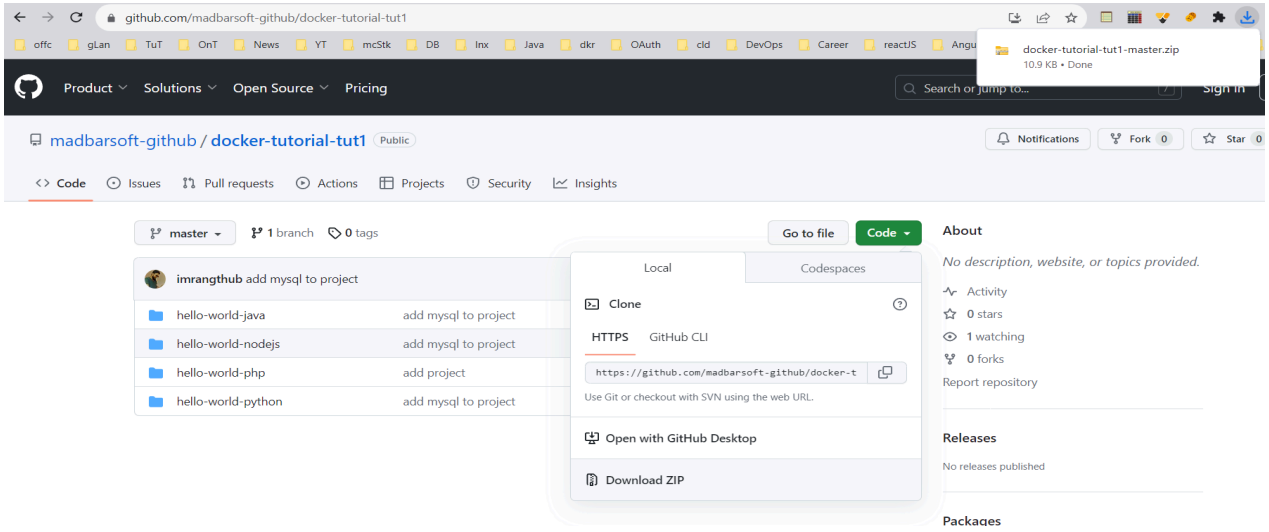
8) Simple Example Project

Project source GitHub Link: https://github.com/madbarsoft-github/docker-tutorial-tut1



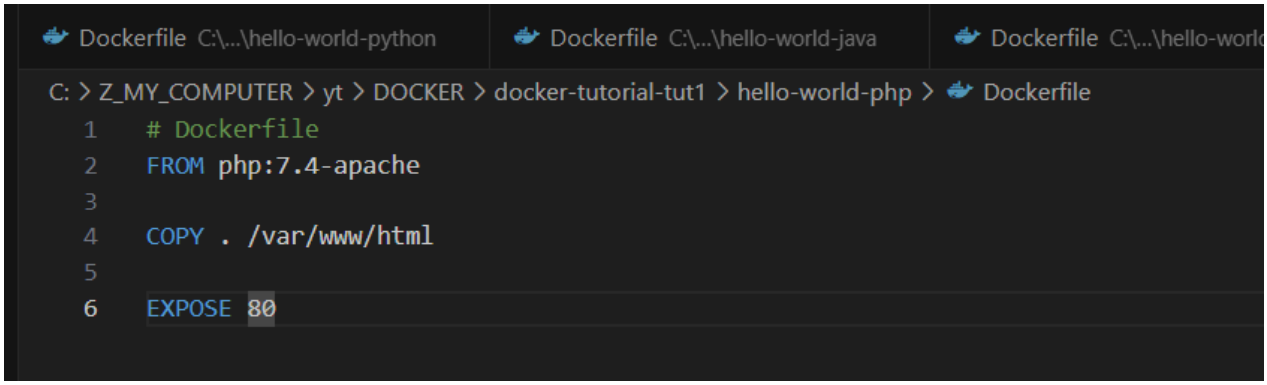
Download project from get using git pull or .zip file

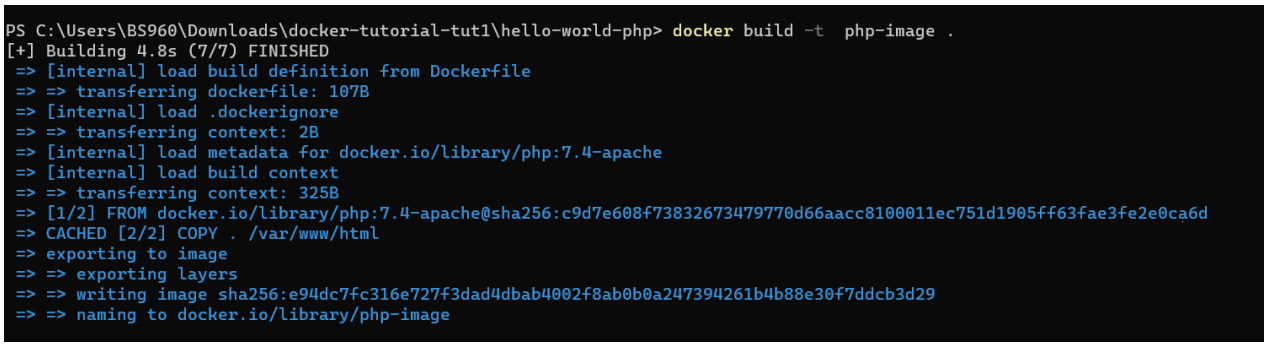
$git clone https://github.com/madbarsoft-github/docker-tutorial-tut1.git



Run PHP Project:

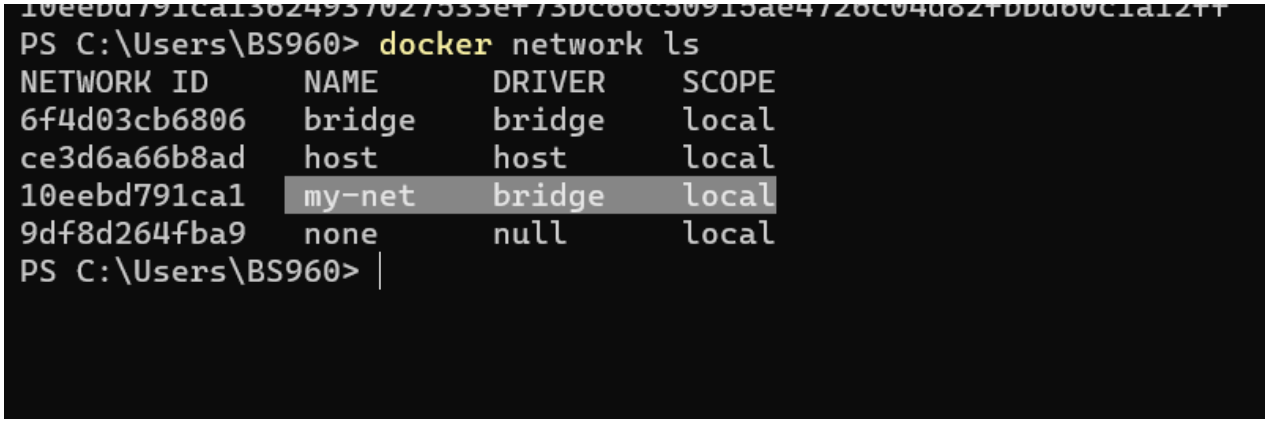
$docker build -t php-image .
$docker run --name php-app -d -p8080:80 php-image





Create a bridge network:

$docker network ls
$docker network create --subnet=192.168.1.0/24 --gateway 192.168.1.1 my-net



Run a simple (Database) MySQL server:

$docker run --name mysqldb --network=my-net -d -e MYSQL_ROOT_PASSWORD=root --env="MYSQL_DATABASE=simpledb" -v mysql-data-vol:/var/lib/mysql mysql:8.2.0

MySQL Operation:

$docker exec -it containerId bash
$mysql -uroot -p
$show databases;
$use simpledb;

=>CREATE TABLE book (id INT PRIMARY KEY, name VARCHAR(255) NOT NULL);

=>INSERT INTO book (id, name) VALUES (1, 'The Great book');

=>SELECT * FROM book;

Run Python Project:

Dockerfile

C: > Z_MY_COMPUTER > yt > DOCKER > docker-tutorial-tut1 > hello-world-python > Dockerfile

```
# Use a base image with Python
FROM python:3.8-slim

# Set the working directory inside the container
WORKDIR /app

# Install Flask and mysql-connector-python
RUN pip install --no-cache-dir Flask mysql-connector-python


COPY app.py /app/

EXPOSE 80

CMD ["python", "app.py"]
```

$docker build -t python-mysql-image .

$docker run --name python-mysql-app --network=my-net -d -p 8080:80 -e MYSQL_HOST=mysqldb -e MYSQL_USER=root -e MYSQL_PASSWORD=root -e MYSQL_DATABASE=simpledb python-mysql-image

```
PS C:\Users\BS960\Downloads\docker-tutorial-tut1\hello-world-python> docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED          STATUS          PORTS                    NAMES
184fdeca0e2b   python-mysql-image   "python app.py"          46 seconds ago   Up 45 seconds   0.0.0.0:8080->80/tcp     python-mysql-app
8d6308e596af   mysql:8.2.0          "docker-entrypoint.s…"   19 minutes ago   Up 19 minutes   3306/tcp, 33060/tcp      mysqldb
PS C:\Users\BS960\Downloads\docker-tutorial-tut1\hello-world-python> docker run --name python-mysql-app --network=my-net -d -p 8080:80 -e MYSQL_HOST=mysqldb
 -e MYSQL_USER=root -e MYSQL_PASSWORD=root -e MYSQL_DATABASE=simpledb python-mysql-image
```

Run Java (BackEnd) Project:

Dockerfile C:\...\hello-world-python | Dockerfile C:\...\hello-world-java

C: > Z_MY_COMPUTER > yt > DOCKER > docker-tutorial-tut1 > hello-world-java > Dockerfile

```
1  # Use a base image with Java and a minimal Linux distribution
2  FROM openjdk:17
3
4  #Set the working directory inside the container
5  WORKDIR /app
6
7  # Copy the Spring Boot JAR file into the container
8  COPY spring-boot-webapp/target/spring-boot-webapp-0.0.1-SNAPSHOT.jar /app/app.jar
9
10 EXPOSE 8080
11
12 CMD ["java", "-jar", "app.jar"]
```

$docker build -t spring-mysql-image .

$docker run --name spring-mysql-app --network=my-net -d -p 8080:8080 -e MYSQL_HOST=mysqldb -e MYSQL_USER=root -e MYSQL_PASSWORD=root -e MYSQL_DATABASE=simpledb spring-mysql-image

```
PS C:\Users\BS960\Downloads\docker-tutorial-tut1\hello-world-java> docker ps
CONTAINER ID   IMAGE               COMMAND                  CREATED          STATUS          PORTS                    NAMES
ddbff6c95f91   spring-mysql-image  "java -jar app.jar"      26 seconds ago   Up 25 seconds   0.0.0.0:8080->8080/tcp   spring-mysql-app
8d6308e596af   mysql:8.2.0         "docker-entrypoint.s…"   32 minutes ago   Up 32 minutes   3306/tcp, 33060/tcp      mysqldb
PS C:\Users\BS960\Downloads\docker-tutorial-tut1\hello-world-java>
```

Run NodeJS (FrontEnd) Project:

Dockerfile C:\...\hello-world-python | Dockerfile C:\...\hello-world-java | Dockerfile C:\...\hello-world-nodejs

C: > Z_MY_COMPUTER > yt > DOCKER > docker-tutorial-tut1 > hello-world-nodejs > Dockerfile

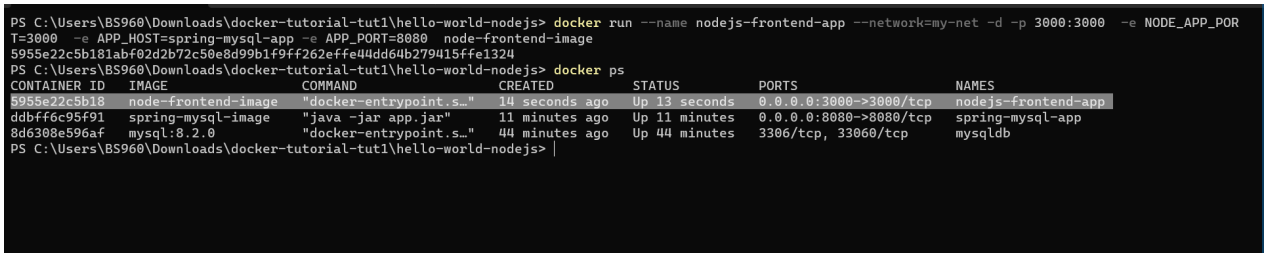
```
1  #Use an official Node.js parent image
2  FROM node:14
3
4  #Set the working directory in the container
5  WORKDIR /app
6
7  #Copy package.json to the working directory
8  COPY package*.json ./
9
10 #Install app dependencies
11 RUN npm install
12
13 # Copy the current directory contents into the container at /app
14 COPY . .
15
16 # Expose port 3000
17 EXPOSE 3000
18
19 # Run the application
20 CMD ["npm", "start"]
```

$docker build -t node-frontend-image .

```
PS C:\Users\BS960\Downloads\docker-tutorial-tut1\hello-world-nodejs> docker build -t  node-frontend-image .
[+] Building 9.3s (10/10) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 535B
 => [internal] load metadata for docker.io/library/node:14
 => [1/5] FROM docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa
 => [internal] load build context
 => => transferring context: 2.29kB
 => CACHED [2/5] WORKDIR /app
 => [3/5] COPY package*.json ./
 => [4/5] RUN npm install
 => [5/5] COPY . .
 => exporting to image
 => => exporting layers
 => => writing image sha256:62762b95f45451544d89c3183f3d198a48de3288a2389eb9a6cef8a931ff9a85
 => => naming to docker.io/library/node-frontend-image
```

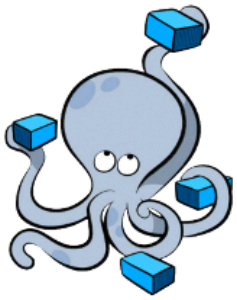
```
PS C:\Users\BS960\Downloads\docker-tutorial-tut1\hello-world-nodejs> docker images
REPOSITORY              TAG       IMAGE ID       CREATED          SIZE
node-frontend-image     latest    62762b95f454   10 seconds ago   917MB
spring-mysql-image      latest    2d74217cd55f   10 minutes ago   520MB
python-mysql-image      latest    25002b9605fa   28 minutes ago   246MB
php-image               latest    e94dc7fc316e   32 hours ago     453MB
```

$docker run --name nodejs-frontend-app --network=my-net -d -p 3000:3000 -e NODE_APP_PORT=3000 -e APP_HOST=spring-mysql-app -e APP_PORT=8080 node-frontend-image



9) Docker Compose

release v2.23.3 | go.dev docs | ci passing | go report A+ | codecov 57% | openssf scorecard 6.7



Docker Compose is a tool for running multi-container applications on Docker defined using the Compose file format. A Compose file is used to define how one or more containers that make up your application are configured. Once you have a Compose file, you can create and start your application with a single command: `docker compose up`.

$docker-compose up
Builds, (re)creates, starts, and attaches to containers defined in the docker-compose.yml file.

$docker-compose down
Stops and removes containers created by up.

$docker-compose ps
Lists containers.

$docker-compose logs
Shows output from containers.

$docker-compose exec
Runs a command in a running container.ac, below the part of dompose file section.

dokr_ref.txt M | Untitled-1 ●

```
1
2   version: '3'
3     ....
4   services:
5     ....
6     ....
7
8   networks:
9     ....
10  volumes:
11    ....
```

Docker Compose works by applying many rules declared within a single ***docker-compose.yml***

dokr_ref.txt M Untitled-1

```yaml
version: '3'


services:
  front-end :
    image: front-end-image
    build:
    context: /workspace/front-end
    dockerfile: /workspace/front-end/docker/Dockerfile
    ......

  back-end :
    image: back-end-image
    build: https://github.com/back-end/my-project.git
    ......

  mysql-db:
    image: mysql:latest
    ......


networks:
  my-net:
    driver: bridge

volumes:
  mysql-data:
    driver: local
```

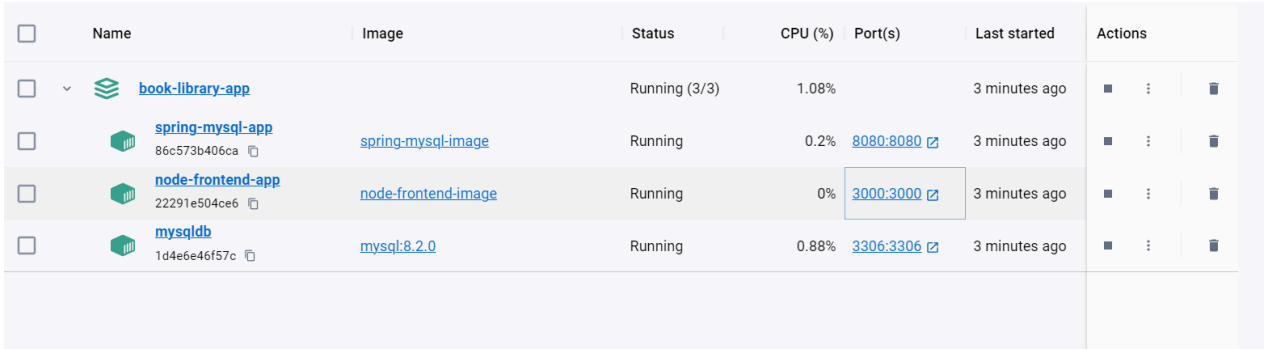
Windows PowerShell

```
PS C:\Z_MY_COMPUTER\yt\DOCKER\docker-tutorial-tut1\docker-compose> docker-compose up -d
[+] Building 0.0s (0/0)
[+] Running 3/3
 ✓ Container mysqldb             Started
 ✓ Container spring-mysql-app    Started
 ✓ Container node-frontend-app   Started
PS C:\Z_MY_COMPUTER\yt\DOCKER\docker-tutorial-tut1\docker-compose> docker-compose down
[+] Running 3/3
 ✓ Container node-frontend-app   Removed
 ✓ Container spring-mysql-app    Removed
 ✓ Container mysqldb             Removed
PS C:\Z_MY_COMPUTER\yt\DOCKER\docker-tutorial-tut1\docker-compose>
```

docker-compose.yml:

```yaml
version: '3.8'
name: book-li
services:
  node-frontend-service:
    image: node-frontend-image
    container_name: node-frontend-app
    environment:
      APP_HOST: spring-mysql-app
      APP_PORT: 8080
    ports:
      - "3000:3000"
    networks:
      - my-net
    depends_on:
      - spring-mysql-service
  spring-mysql-service:
    image: spring-mysql-image
    container_name: spring-mysql-app
    environment:
      MYSQL_HOST: mysqldb
      MYSQL_USER: root
      MYSQL_PASSWORD: root
      MYSQL_DATABASE: simpledb
    ports:
      - "8080:8080"
    networks:
      - my-net
    depends_on:
      - mysql-service
  mysql-service:
    image: mysql:8.2.0
    container_name: mysqldb
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: simpledb
    ports:
```

```
      - '3316:3306'
    networks:
      - my-net
    volumes:
      - mysql-data-vol:/var/lib/mysql
networks:
  my-net:
    driver: bridge
    name: my-net
    external: true
volumes:
  mysql-data-vol:
    name: mysql-data-vol
    external: true
```

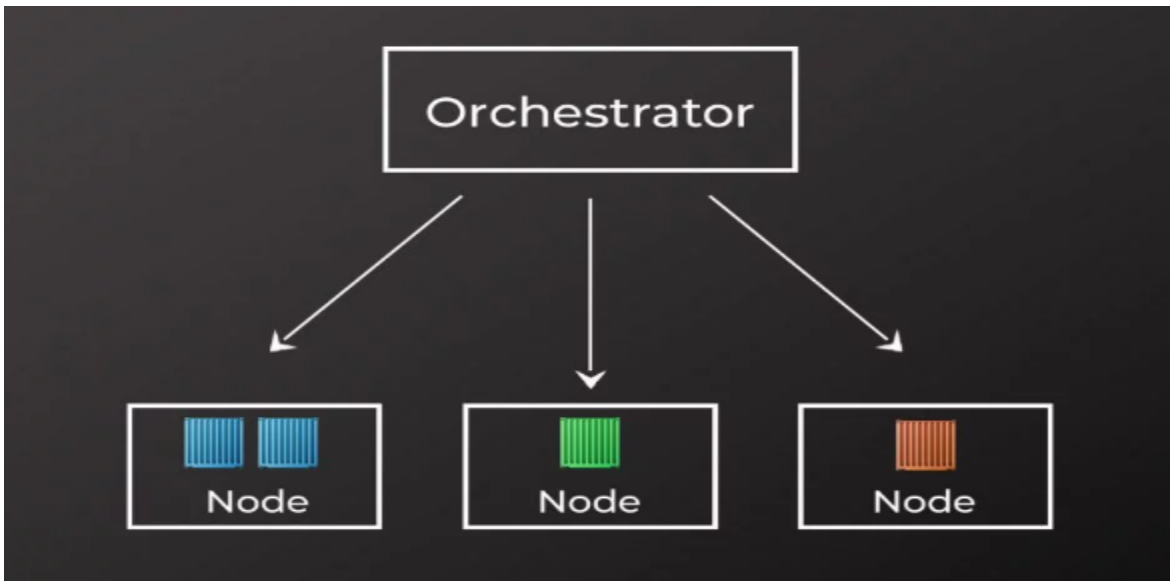


Git Link:

https://github.com/madbarsoft-github/docker-tutorial-tut1/blob/master/docker-compose.yml

10) Container Orchestration (Docker Swarms and Kubernetes)

Container orchestration is a big tools for managing and deploying containerized applications. It helps automate the deployment, scaling, and management of containerized applications.

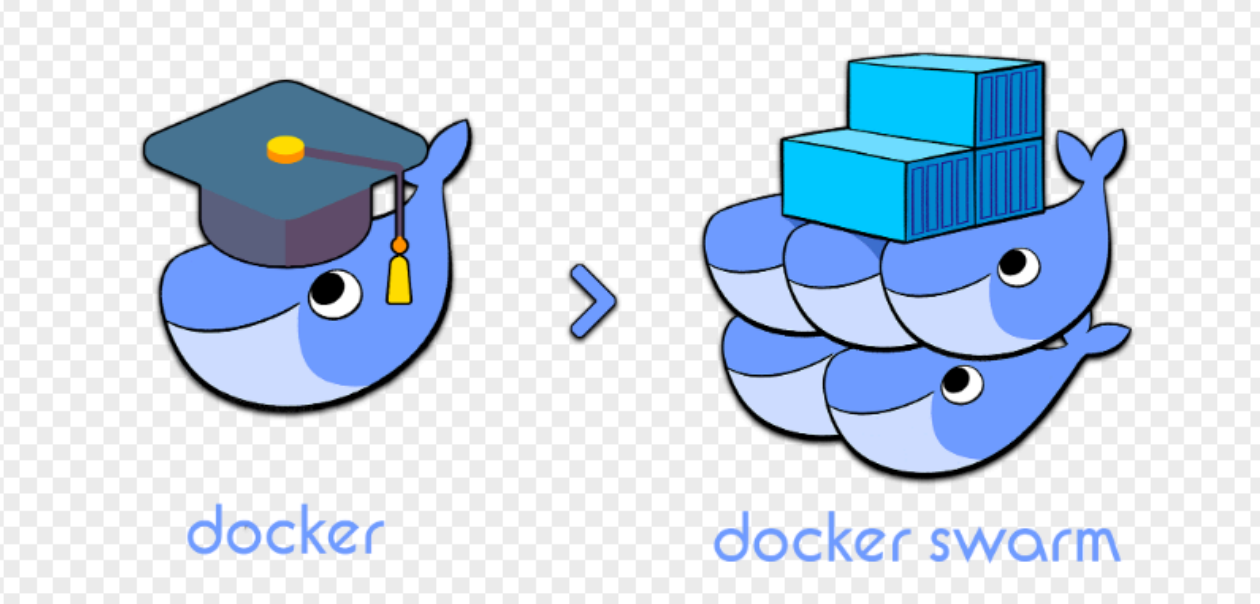


What is container orchestration used for: Use container orchestration to automate and manage tasks such as:

- Provisioning and deployment
- Configuration and scheduling
- Resource allocation
- Scaling or removing containers based on balancing workloads across your infrastructure
- Load balancing and traffic routing
- Keeping interactions between containers secure

What is a swarm ?

Docker Swarm is native cluster managing toos and mode for docker and its offers limited feature. A swarm consists of multiple Docker hosts. A given Docker host can be a manager, a worker, or perform both roles.



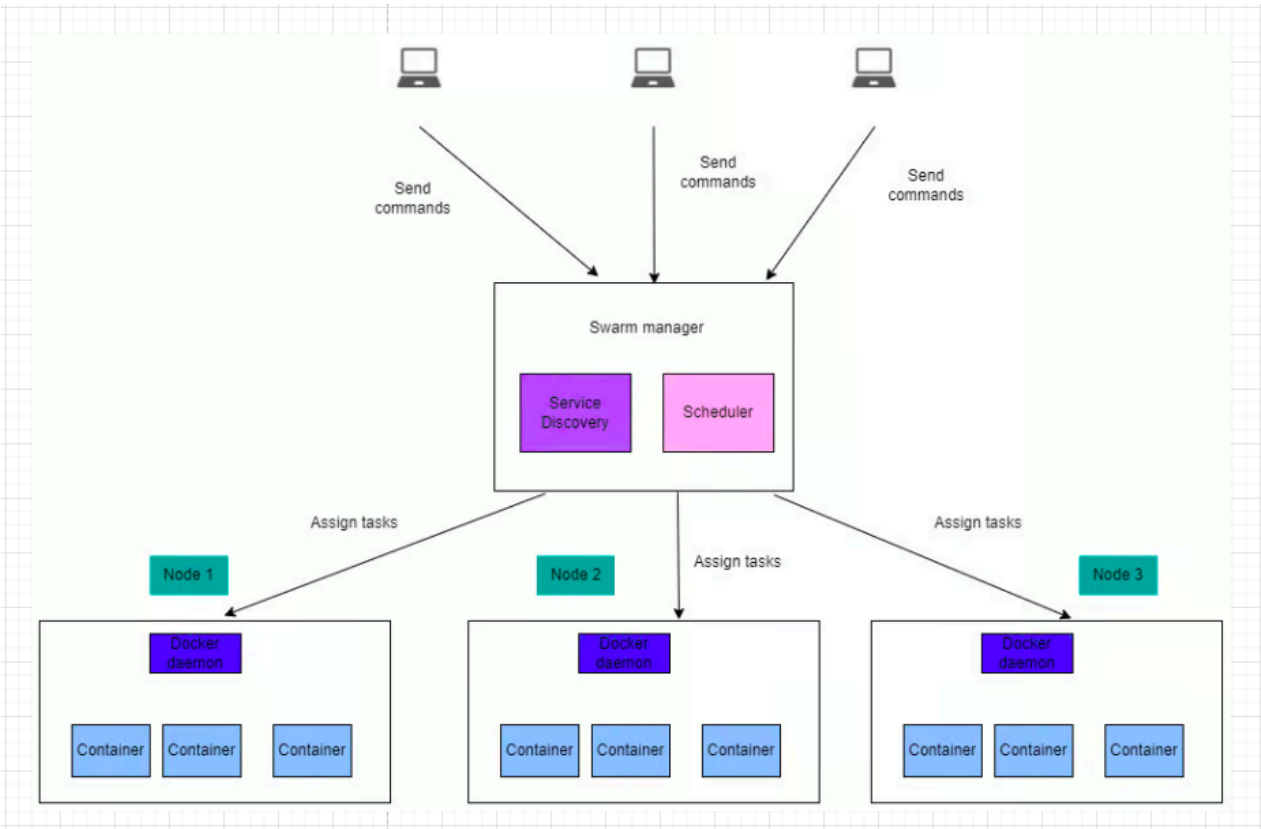
Nodes:

A node is an instance of the Docker engine participating in the swarm. You can run one or more nodes on a single physical computer or cloud server. here are two types of nodes in Docker Swarm:

Manager node. Maintains cluster management tasks

Worker node. Runs containers, executes tasks assigned by the Manager Node.

Docker Swarm component:



Swarm Mode:
Docker's native orchestration for clustering hosts into a single virtual environment.

Service:
A Service defines the desired state of an application in Docker Swarm, specifying parameters such as replicas, network configurations, and more.

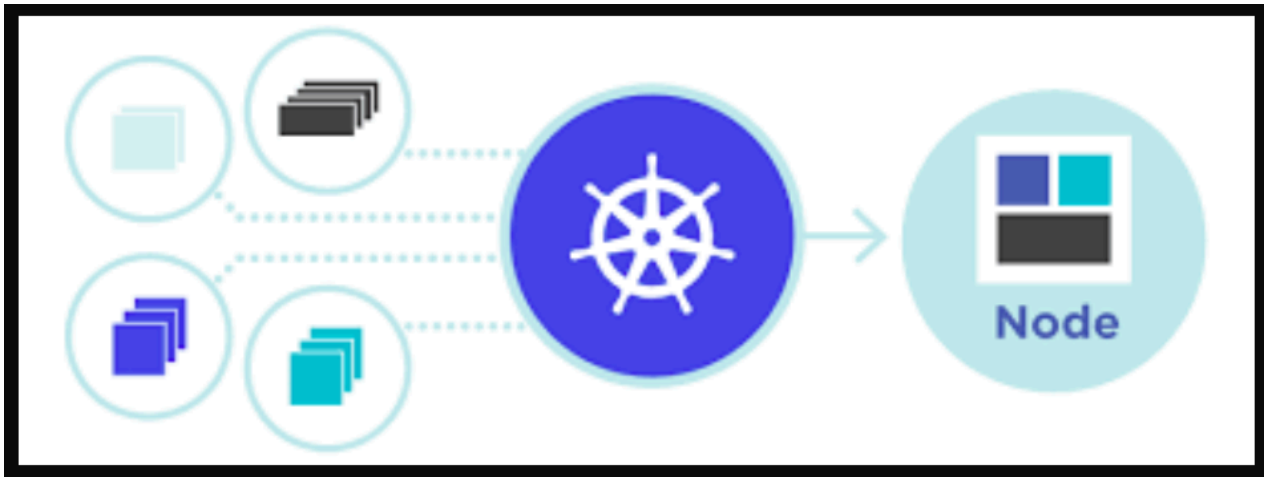
Task:
Smallest unit of work, represents a running container.

Overlay Network:
Facilitates seamless container communication on different nodes.

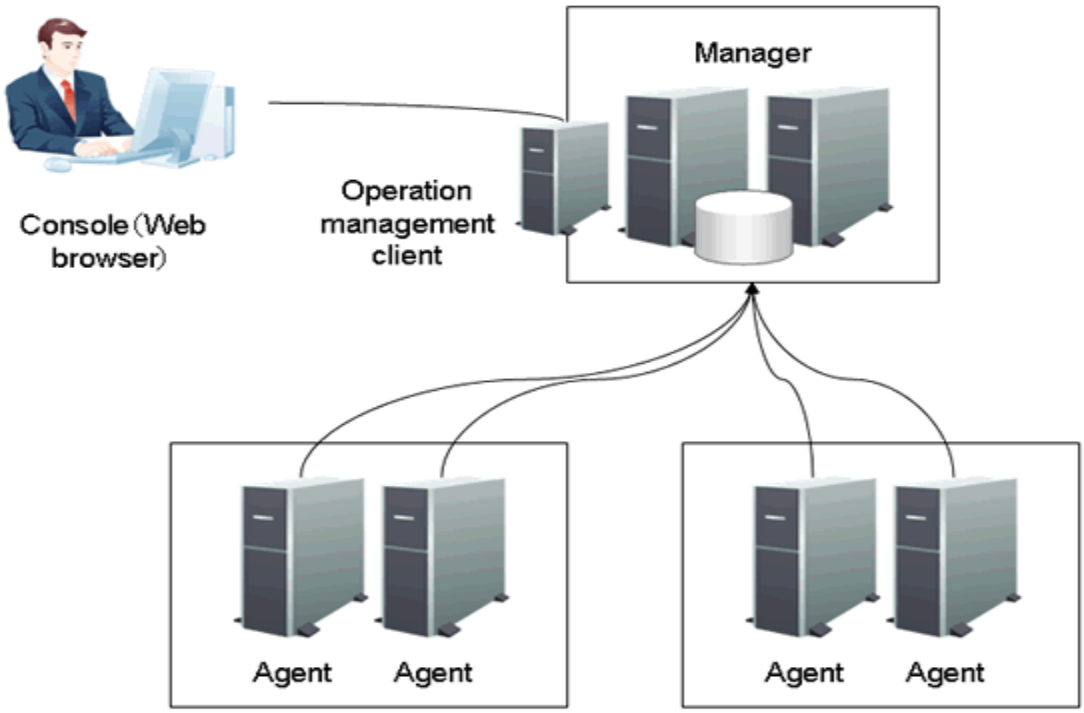
Load Balancing:
Distributes incoming traffic across containers for better performance.

Kubernetes:

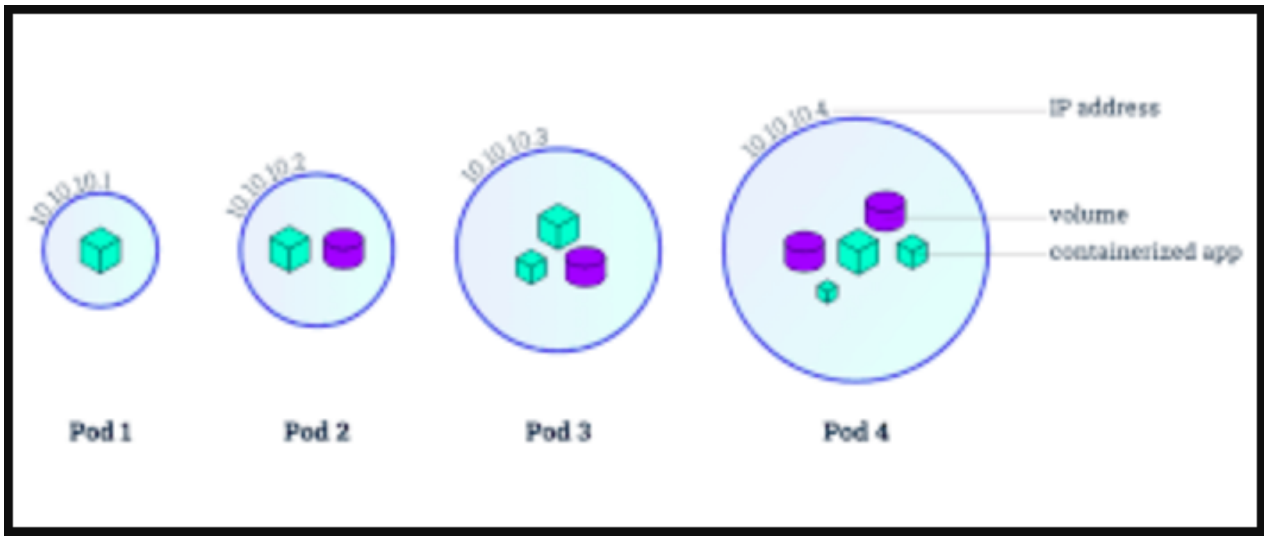
Kubernetes is an open-source container orchestration system for automating software deployment, scaling, and management. Originally designed by Google, the project is now maintained by the CNCF (Cloud Native Computing Foundation).



The kubectl command-line tool supports several different ways to create and manage Kubernetes objects.



Kubernetes Pod:

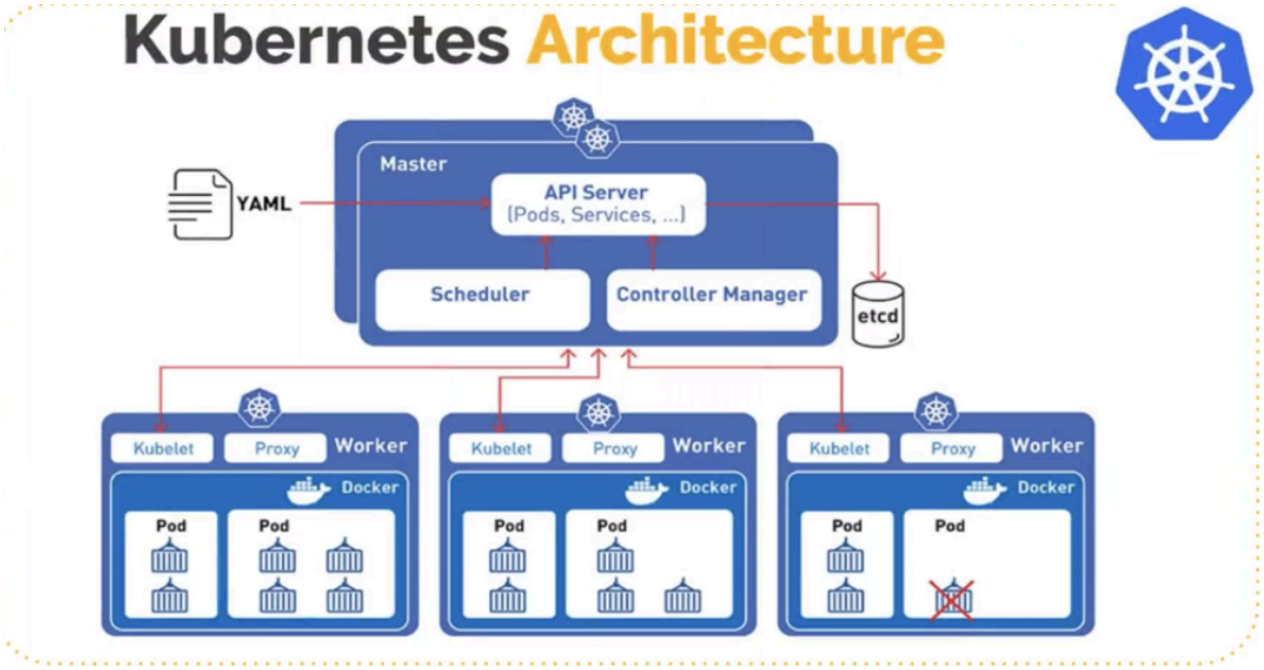


Imperative commands

```
$kubectl run my-nginx-pod --image=nginx:latest --port=80

$kubectl get pods
```

Declarative object configuration:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```



Kubernetes Components:

Master Node:
Controls the Kubernetes cluster, managing scheduling and orchestration.

Node:
Worker machines that run containers and execute tasks assigned by the Master Node.

Pod:
Smallest deployable unit, representing one or more containers that share resources.

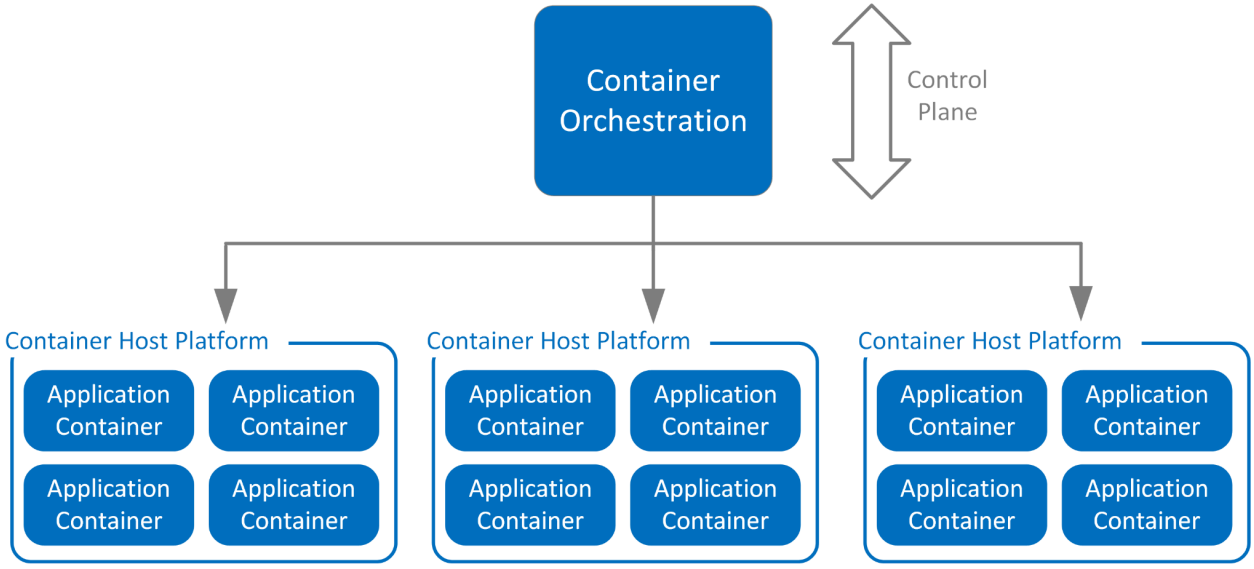
ReplicaSet:
Ensures a specified number of replicas (pods) are running at all times.

Service:
Enables network access to a set of pods, providing a stable endpoint.

Ingress and Kube-DNS:
Manages external access to services within the cluster. Provides DNS-based service discovery in the Kubernetes cluster.

ConfigMap and Secret:
ConfigMap Stores non-sensitive configuration data for pods and Secret stores sensitive information, such as passwords or API keys, for pods.

What is Next ?



Go For Container Orchestration ……