DevOps Training

Week 03:
Containers, Docker

Muhammad Sohaib Ali

/msohaibali

Previous Weeks Recap

- What is DevOps
- VM Introduction & How to Setup one
- LINUX Basics
- Introduction to Git and GitHub

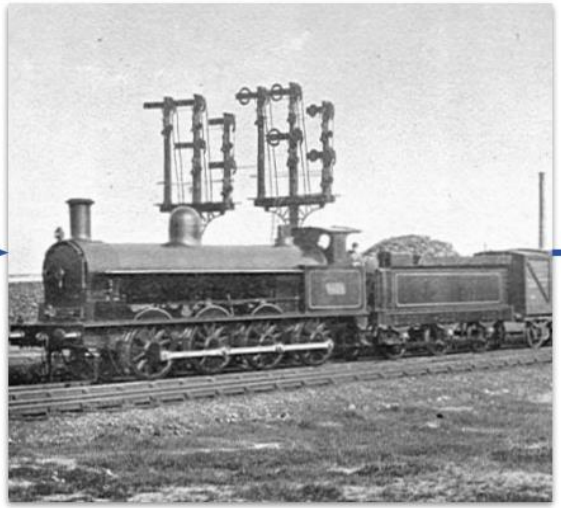
Agenda - Week 3

1. Understanding Containers
2. Docker Deep Dive

Introduction to Containers

Quick History of Shipping

Source: https://www.publicdomainpictures.net/en/view-image.php?image=275355



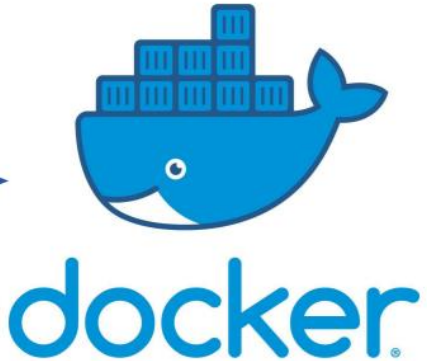
Source: https://en.wikipedia.org/wiki/Rail_freight_in_Great_Britain

Source: https://pxhere.com/en/photo/553345

Software = Shipping?

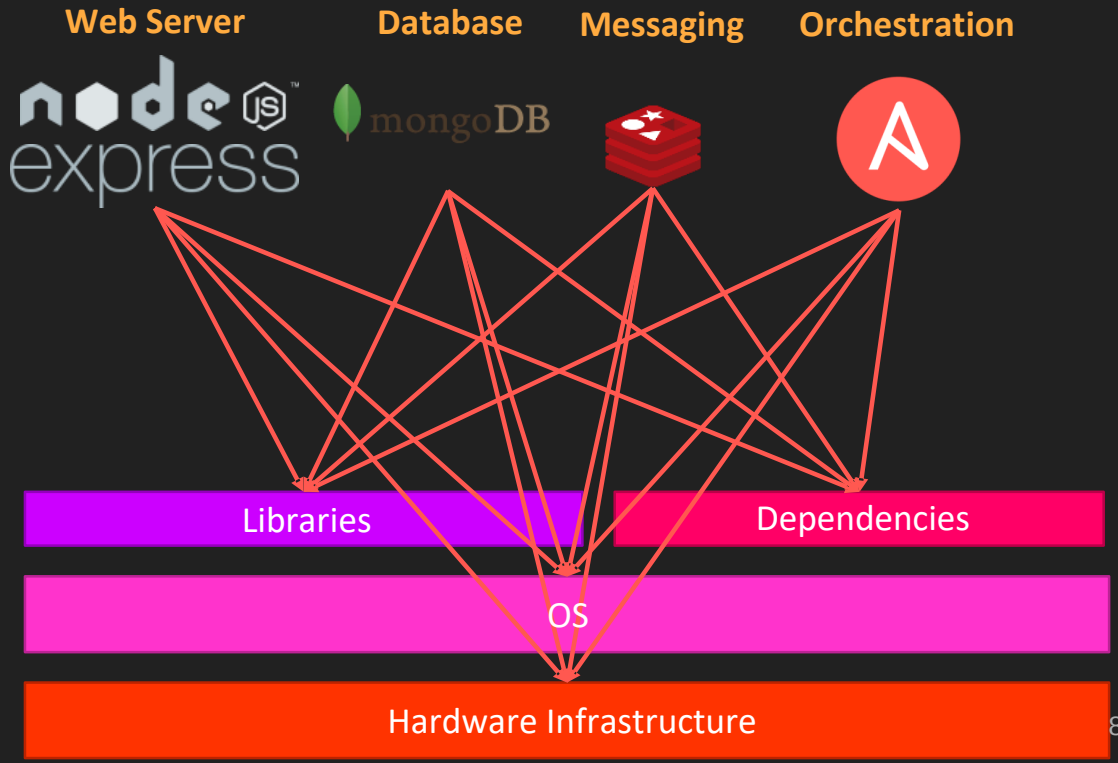
Shipping in Software

Source: https://www.usafe.af.mil/News/Photos/igphoto/2000887438/



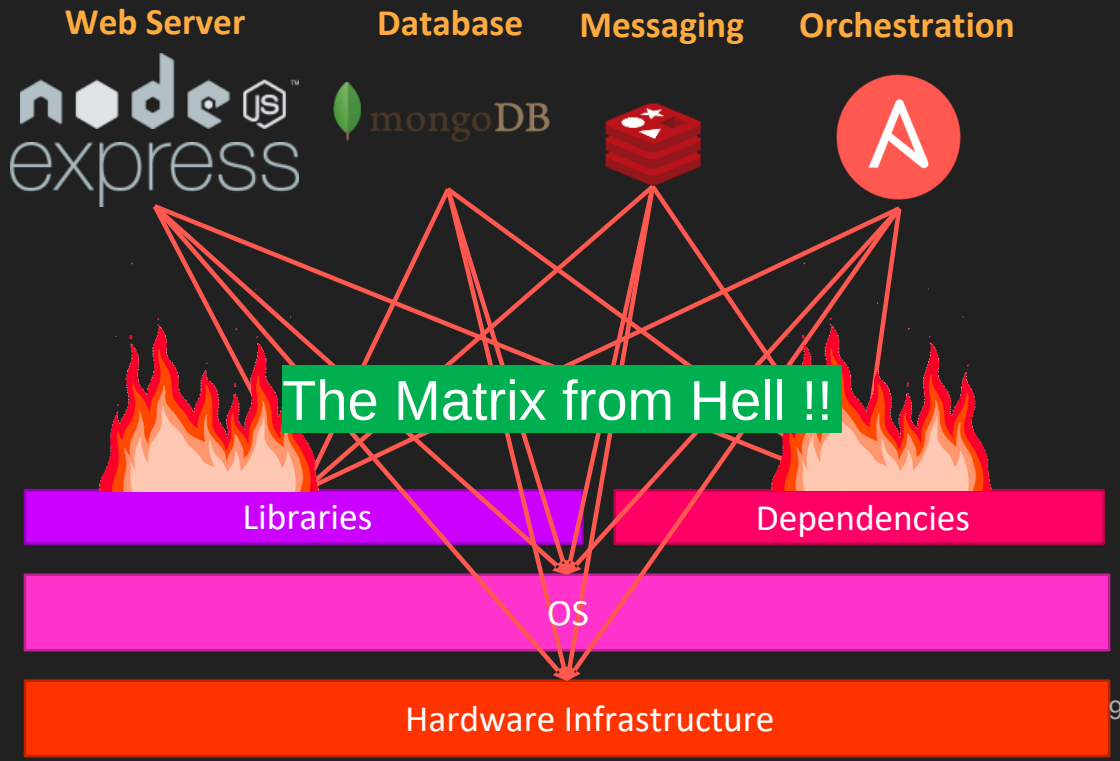
Shipping in Software: **Traditional Way**

- Compatibility/Dependency
- Long setup time
- Different Dev/Test/Prod environments



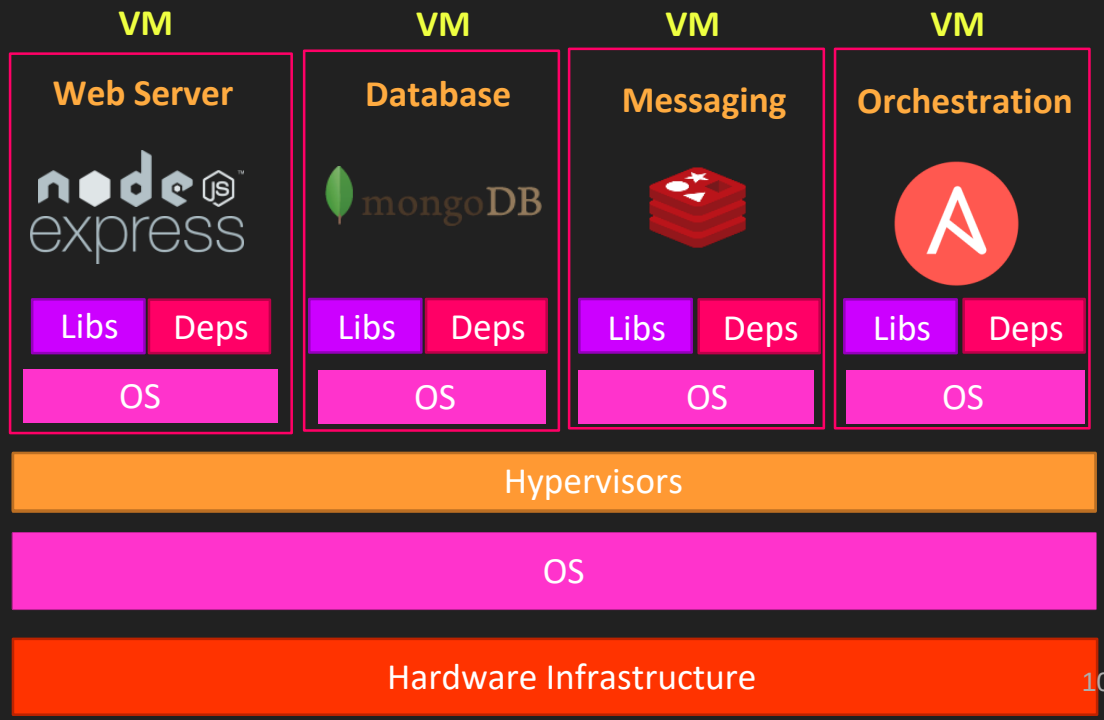
Shipping in Software: **Traditional Way**

- Compatibility/Dependency
- Long setup time
- Different Dev/Test/Prod environments



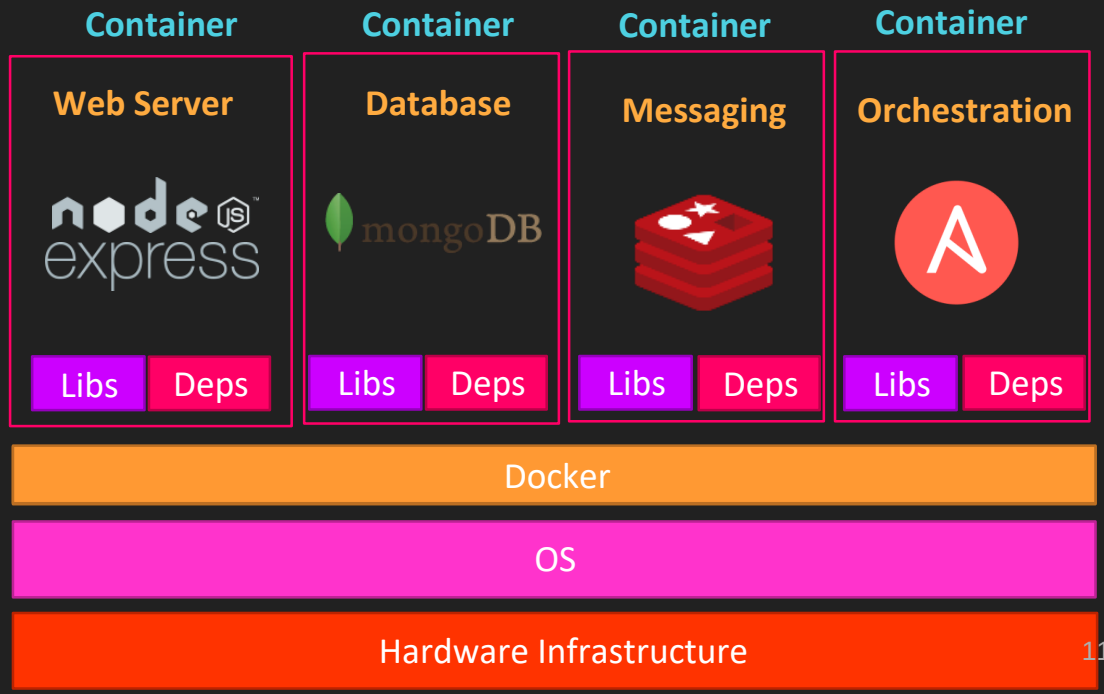
Shipping in Software: **Using VMs**

- Performance overhead
- Resource consumption
- Complexity
- Licensing costs
- Security concerns



Shipping in Software: **Using Containers**

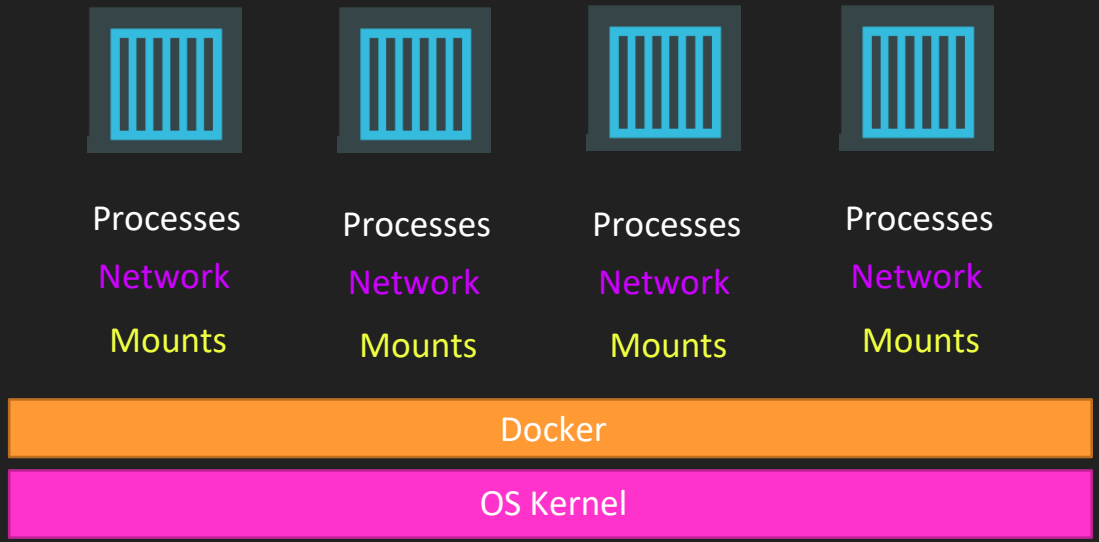
- Lightweight
- Portability
- Scalability
- Resource efficiency
- Isolation



Definition and characteristics

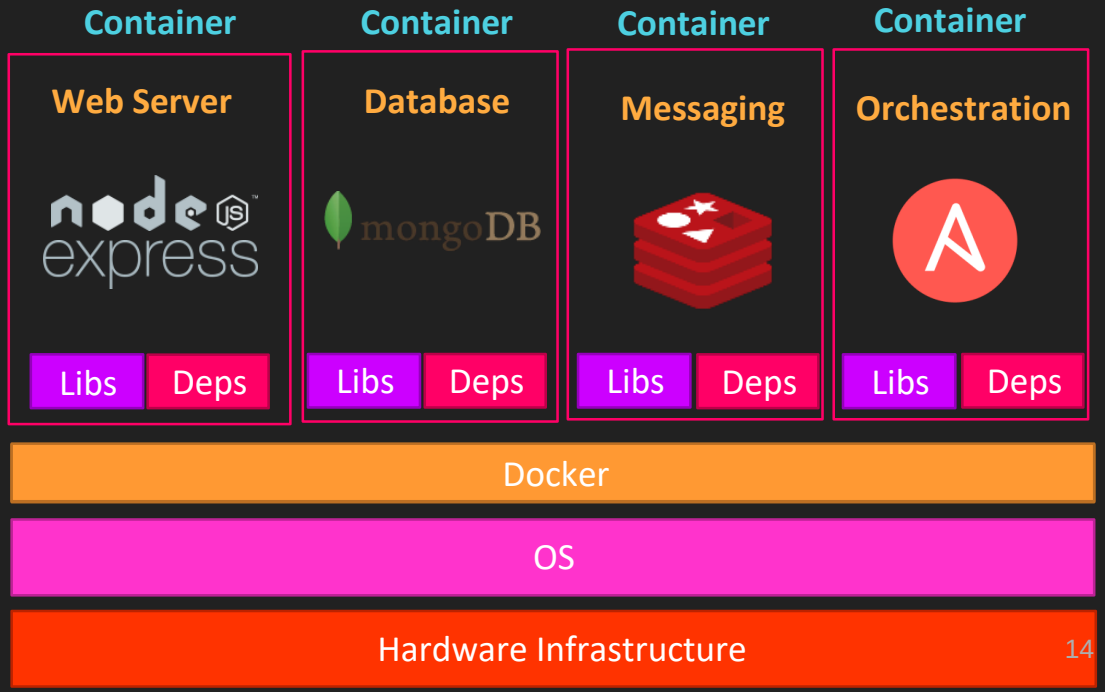
- Container:
  - A **self-contained** environment for running software that provides a **lightweight** and **portable** alternative to traditional virtual machines.
- It can **encapsulate** an application and all its dependencies
- Containers are **isolated** from each other and from the host operating system,
- Ensures that applications can run consistently across different computing environments.

What are containers?

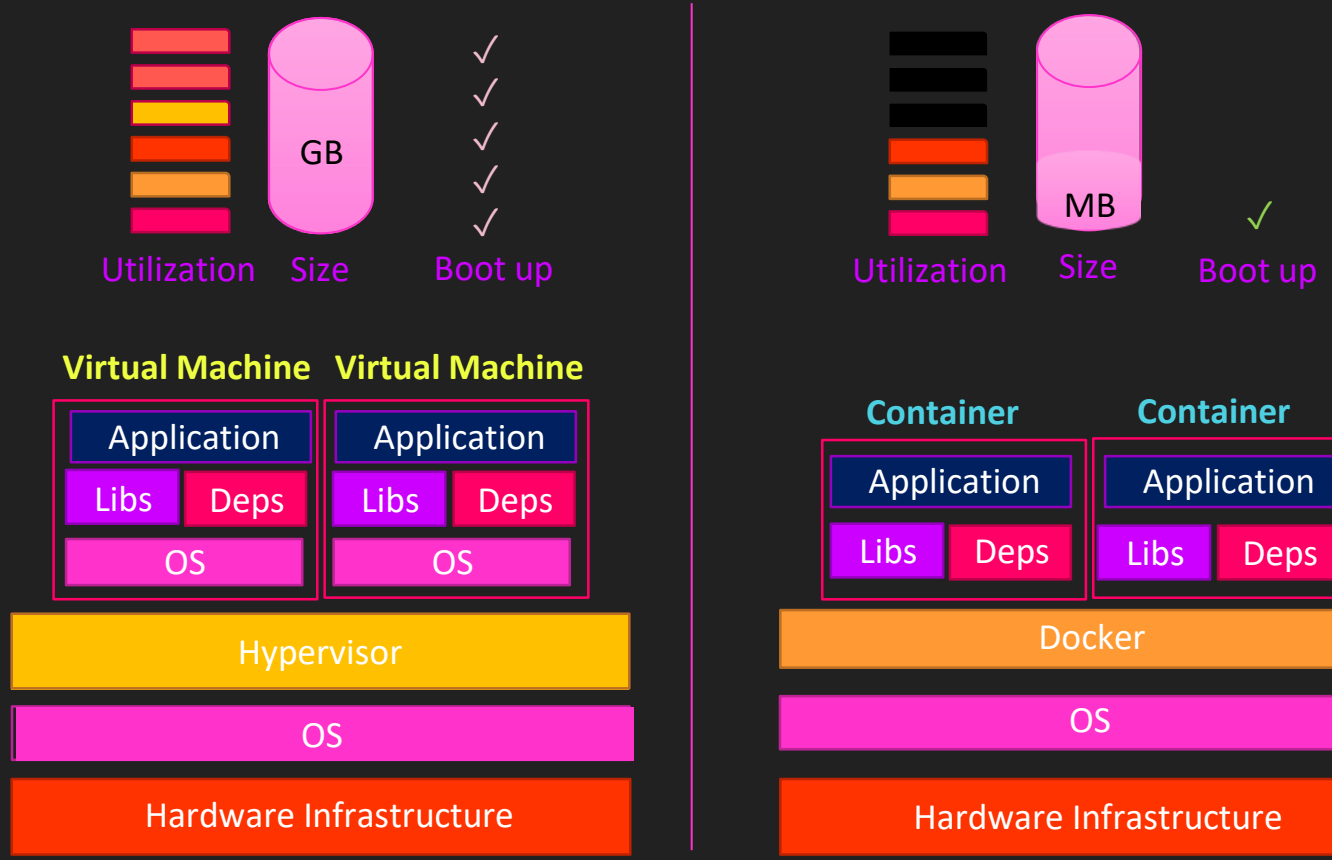


Definition and characteristics

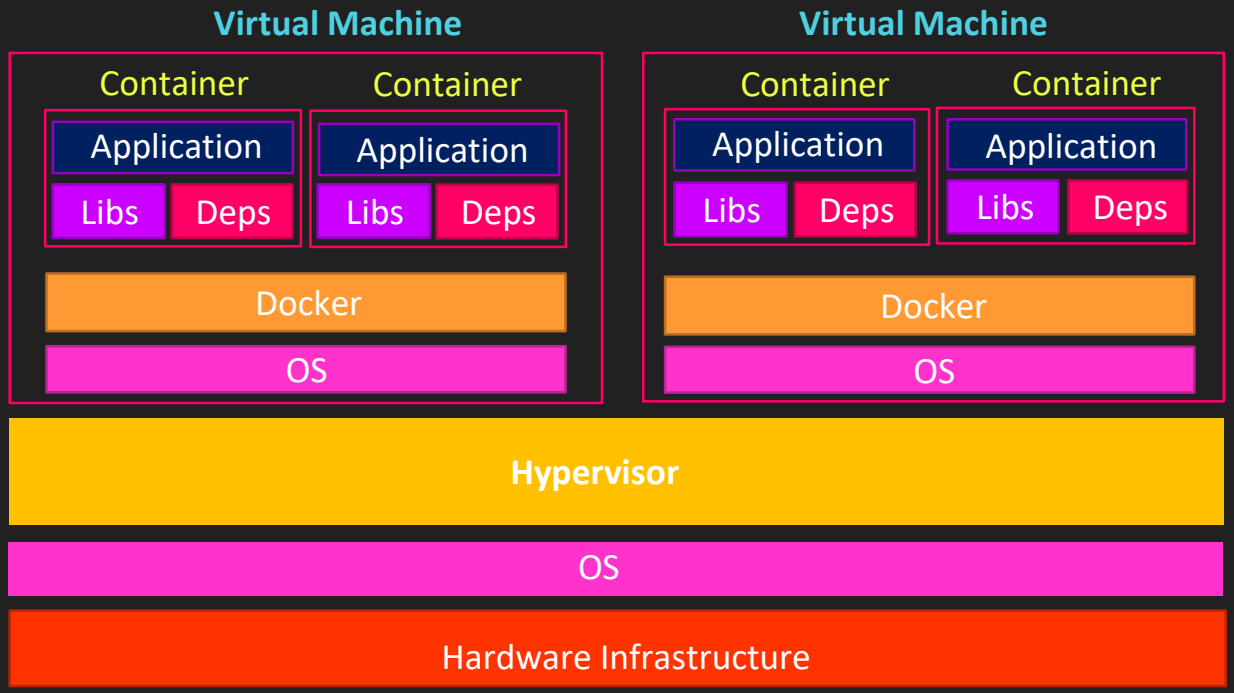
- Containers share the same host operating system kernel
- But they have their own file systems, processes, and networking.
- Run multiple instances of the same application on the **same host**, without interfering with each other or the host operating system.
- Containers are also **lightweight**



Containers vs Virtual Machines



Containers and Virtual Machines



Key Characteristics of Containers

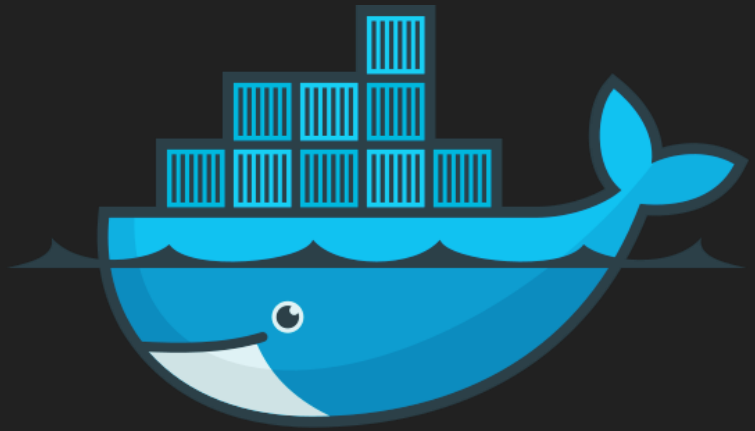
- Portability
- Isolation
- Lightweightness
- Efficiency
- Scalability

Containerization Technology

- Containers are created from **container images**
- Image contain all the necessary files and dependencies for an application.
- Containers are managed by container runtimes
- Container **orchestration** tools, such as Kubernetes, enable developers to manage and scale containers across multiple hosts.

Containerization Technology

- Containers are created using containerization technology
- There are several containerization technologies available, but the most popular and widely used one is Docker.



Containerization technology: **Docker**

- Open-source containerization platform
- Provides a simple and standardized way of creating container
- Docker also provides a centralized repository, called **Docker Hub**

Containerization Technology: **Docker**

- Docker uses a client-server architecture.
- The Docker daemon is responsible for managing containers and providing the necessary system services.
- The Docker client provides a command-line interface and a graphical user interface for interacting with the Docker daemon.
- We will explore docker engine architecture in more details soon!

Containerization Technology

- Other containerization technologies include LXC/LXD, rkt.
- LXC/LXD is a Linux-based containerization technology that provides a more traditional virtualization environment, with more overhead than Docker.
- Rkt is an alternative to Docker that provides a more secure and modular containerization platform.

Challenges and Limitations

- Security Risks if not properly secured and isolated from each other and host
- Can be complex to manage
- Compatibility issues

Future of Containerization

- The containerization ecosystem is rapidly evolving, with new technologies and tools emerging.
- Containers are becoming more integrated with other cloud-native technologies, such as serverless computing and edge computing.
- The future of containerization is closely tied to the larger trends in software development and deployment, such as cloud computing, microservices, and DevOps.

Docker Deep Dive

Let’s study docker in detail

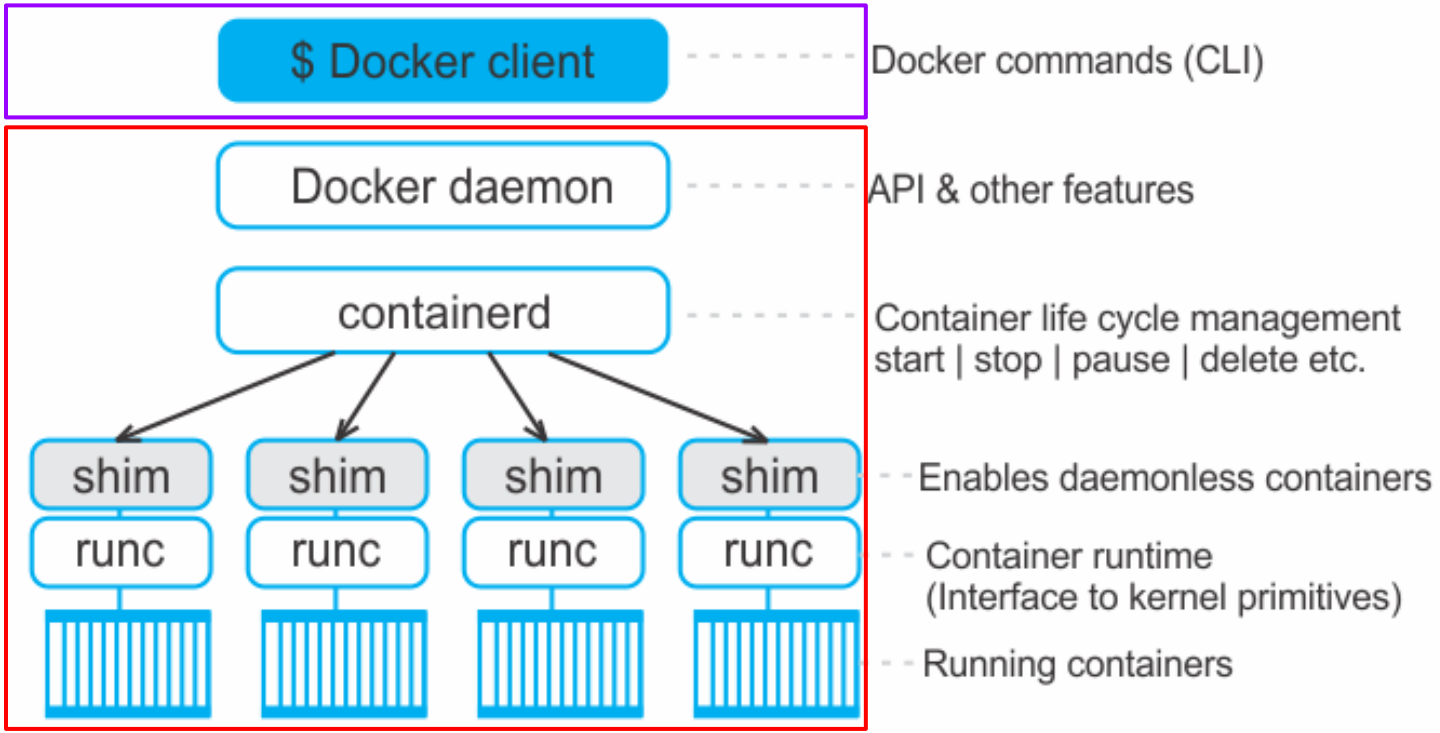
Introduction to Docker

- Open-source platform for building, shipping, and running applications in containers.
- First released in 2013 by Docker, Inc. and has since become one of the most widely used containerization platforms
- Docker simplifies the containerization process
- Docker is built on a modular architecture that allows developers to easily package and distribute applications as container images.
- These container images can be run on any system that supports Docker, regardless of the underlying hardware or operating system.
- Docker has gained popularity for its ease of use, portability, and flexibility. It has become an essential tool for developers, IT operations teams, and DevOps engineers who are looking to build, deploy, and manage modern applications with agility and efficiency.

Docker Engine Architecture

CLIENT

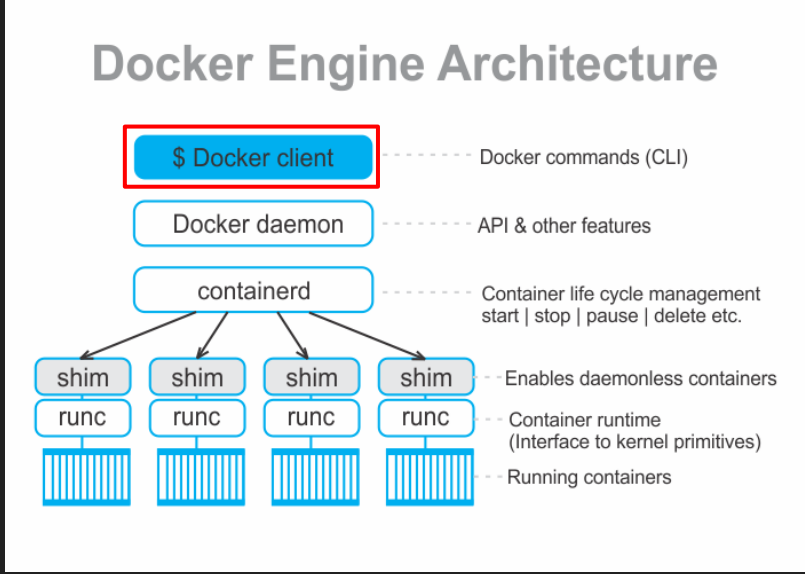
SERVER



CLIENT and SERVER can be on same system/machine or on separate. Usually they are on same machine.

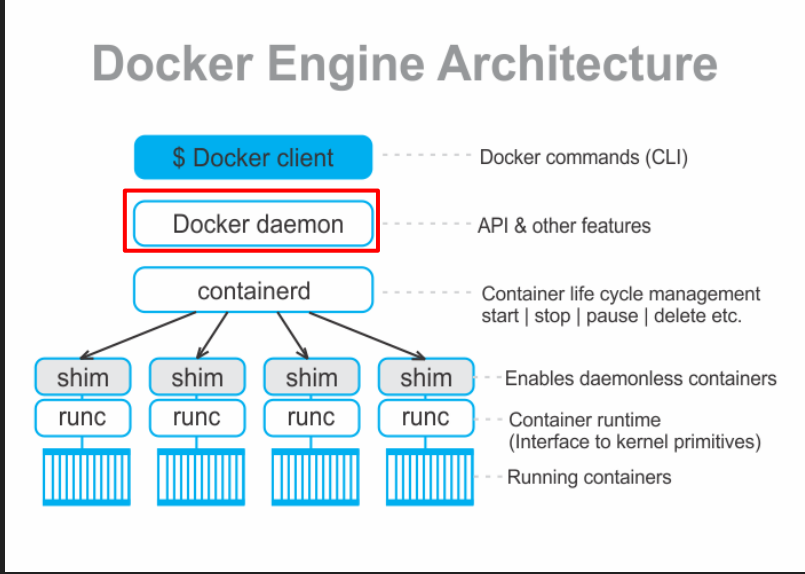
Docker Client

- A command-line tool that interacts with the Docker daemon and provides a user interface for managing containers.
- The Docker client sends API requests to the Docker daemon and receives responses from the daemon.
- A user interacts with docker engine with docker client (CLI: Command Line Interface)



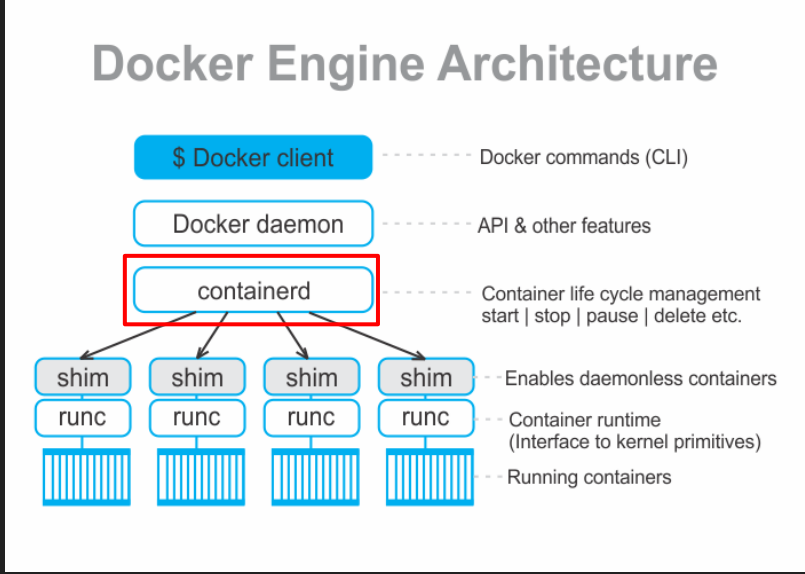
Docker Daemon/Engine

- The Docker daemon, also known as the Docker Engine, is a background service that runs on the host machine and manages containers.
- The Docker daemon listens for Docker API requests and is responsible for managing containers, images, and networks, and it uses a low-level container runtime service to interact with the Linux kernel and run containers.



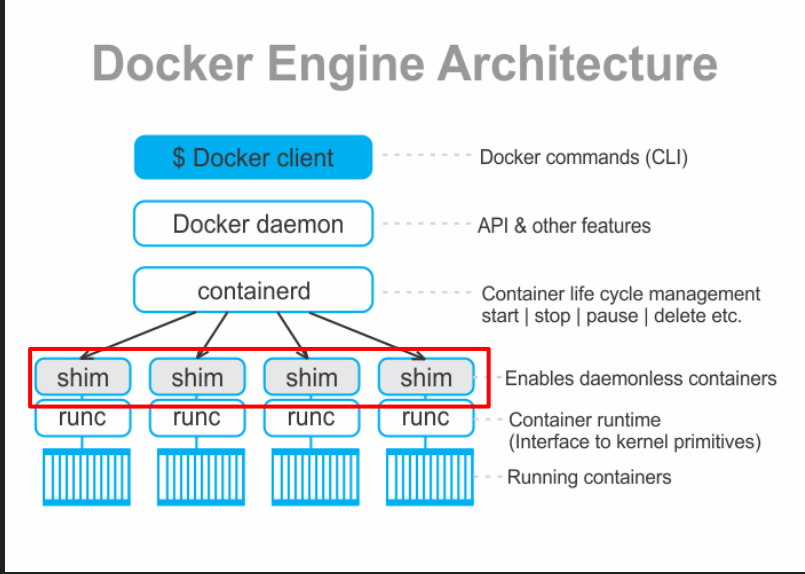
Containerd

- Containerd is an open-source project that provides a container runtime service
- Container runtime: Containerd provides a low-level container runtime service that interacts with the Linux kernel to run containers.
- The container runtime service is responsible for creating and managing container processes, isolating container resources, and managing container networks.



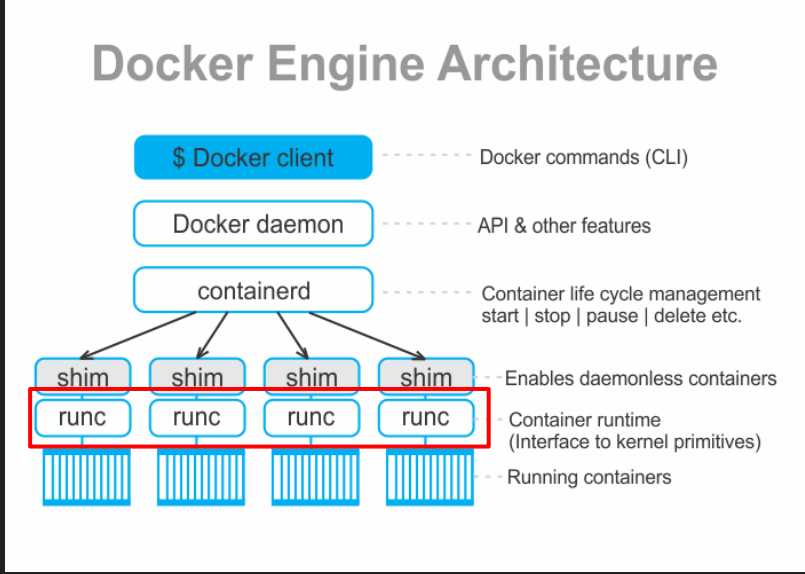
Shim

- The shim is a lightweight component that acts as a bridge between the Docker daemon and the container runtime service (i.e., containerd).
- The shim was introduced in Docker 1.11 to improve the security and reliability of Docker Engine by isolating the container process from the Docker daemon and providing a more robust communication channel between the Docker daemon and the container runtime service.
- The shim is responsible for managing the container process lifecycle, including starting and stopping the container process.



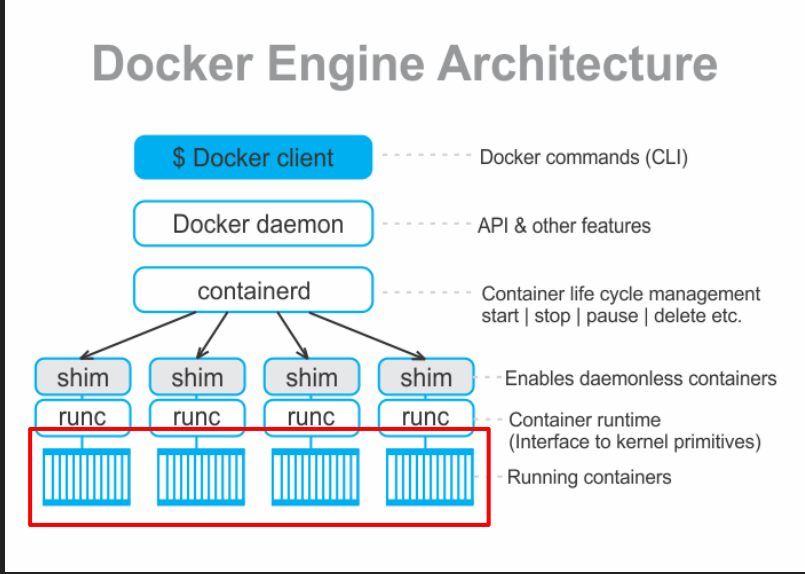
Runc

- Runc is a command-line tool that provides a container runtime implementation compliant with the Open Container Initiative (OCI) specification.
- Runc is the default container runtime implementation used by containerd for running containers in Docker Engine.
- Runc is responsible for starting and managing the container processes, managing container filesystems, and managing container networking.
- Runc uses the Linux namespaces and cgroups features to provide process isolation and resource management for container processes.



Containers

- A container is a self-contained environment for running software that provides a lightweight and portable alternative to traditional virtual machines.
- A container can encapsulate an application and all its dependencies, including libraries, binaries, and configurations, in a single package.
- Containers are isolated from each other and from the host operating system, which ensures that applications can run consistently across different computing environments.
- ALREADY EXPLAINED IN PREVIOUS LECTURES.



d o c k e r

G ett i n g S ta r te d

Docker Editions

Community Edition

Enterprise Edition

Docker Editions

- Docker Community Edition (CE):
  - Open-source software package
  - Free to use
  - Supported by a large community of contributors
  - Ideal for small teams or individual developers
  - Includes basic features for creating and managing containers, such as Docker engine, Swarm mode, and Docker Compose
  - Community-supported
- Docker Enterprise Edition (EE):
  - Commercial software package
  - Provides additional features and support for enterprise-grade applications
  - Includes all the features of Docker CE
  - Provides enterprise-grade support and training from Docker
  - Includes additional features such as secure container orchestration, advanced networking capabilities, and integrated security
  - Designed for larger organizations with more complex deployments
  - Better integration with other enterprise systems and platforms, such as Kubernetes and cloud providers.

Community Edition

Linux

MAC

Windows

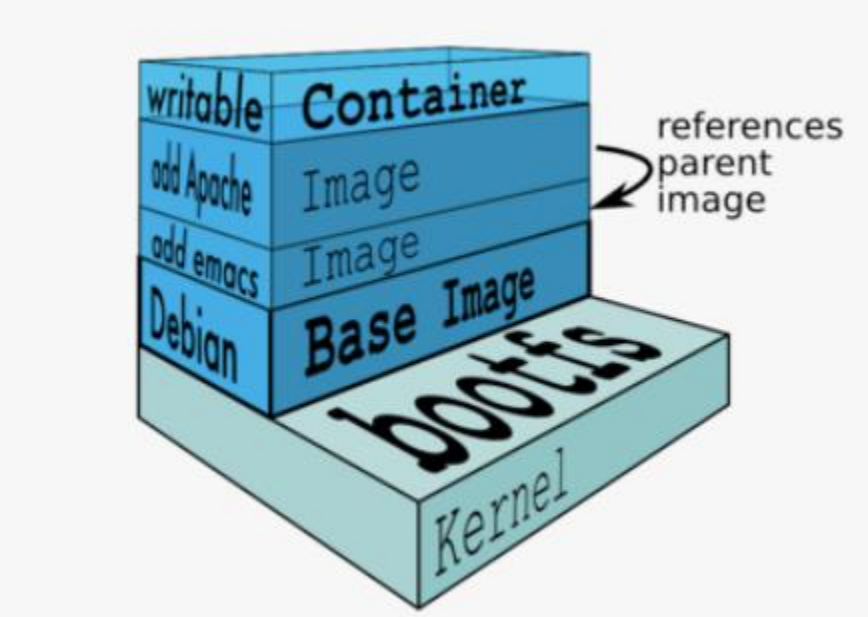
Cloud

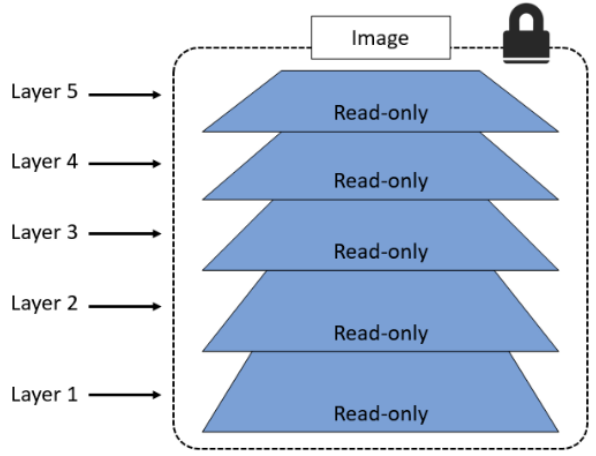
LAB: Installing docker engine and cli

- Go to https://docs.docker.com/engine/install/ubuntu/
- Follow installation instructions
- Use docker login to link to remote container registry

What is a Docker Image?

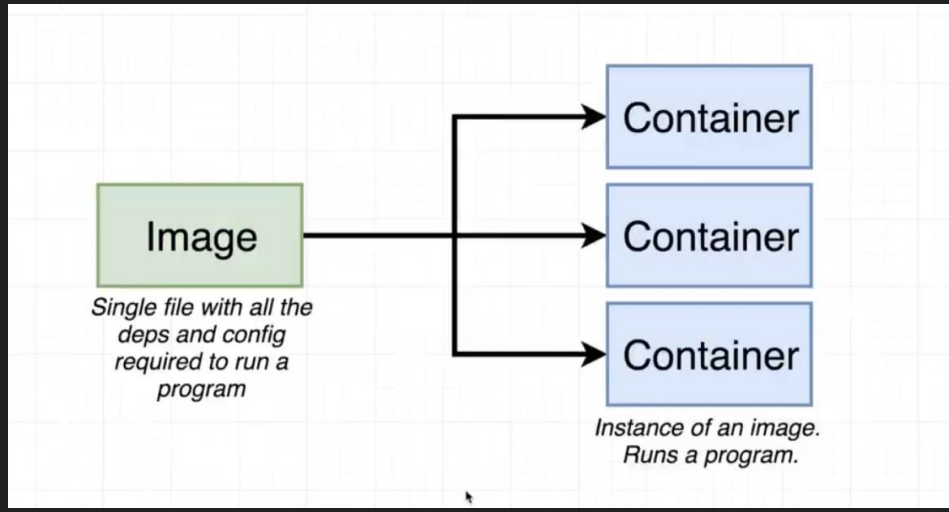
- An image is a collection of files + some meta data.
- Images are made of layers, conceptually stacked on top of each other.
- Each layer can add, change, and remove files.
- Multiple Images can share layers to optimize disk usage, transfer times, and
- memory use.





Differences between containers and images

- An image is a read-only filesystem.
- A container is an encapsulated set of processes running in a read-write copy of that filesystem.
- To optimize container boot time, copy-on-write is used instead of regular copy.
- docker run starts a container from a given image.
- Let's give a couple of metaphors to illustrate those concepts.



Analogy

- Image as stencils
- Images are like templates or stencils that you can create containers from.

Docker container

- Containers are “instance” of an image mean a container is created from the docker image.
- A Docker container is a lightweight, standalone executable package that contains everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings.
- It is created by running a Docker image, which starts a process inside a container that is isolated from the host system and other containers.
- Containers can be started, stopped, and deleted on demand, and they are designed to be highly portable and easily deployable.

Dockerfile

- A Dockerfile is a text file that contains a series of instructions used to build a Docker image.
- These instructions define the environment and configuration required for running an application in a container.
- The Dockerfile typically includes a base image, instructions for installing and configuring dependencies, setting environment variables, and defining the commands to run when the container is launched.
- The Dockerfile is used as a blueprint for building the Docker image, which can then be used to create and run Docker containers.

Dockerfile

- A Dockerfile is a build recipe for a Docker image.
- It contains a series of instructions telling Docker how an image is constructed.
- The docker build command builds an image from a Dockerfile.

Writing our first Dockerfile

Dockerfile

```
INSTRUCTION ARGUMENT
```

Dockerfile

```
FROM Ubuntu

RUN apt-get update

RUN pip install flask


COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

Start from a base OS or another image

Install all dependencies

Copy source code

Specify Entrypoint

What does the ubuntu docker base image contains?

- Here are the key components of an Ubuntu Docker image:
  - A minimal Ubuntu operating system, usually based on a specific release (e.g. 20.04, 18.04)
  - The Ubuntu distribution's package manager, APT, and related tools for installing and managing software packages
  - Basic system utilities and libraries required for running applications
  - Any additional software packages that have been installed by the image maintainer
  - Configuration files for the Ubuntu system and any installed software
  - Any data or assets that are included in the image, such as example files or scripts
  - Note that while the Ubuntu Docker image does not contain a full kernel, it is able to use the host system's kernel through the Docker runtime, which allows containers to run with a high degree of isolation from the host system.

Writing our first Dockerfile

Dockerfile

```
FROM Ubuntu

RUN apt-get update


RUN pip install flask


COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

1. OS - Ubuntu
2. Update apt repo
3. Install dependencies using apt
4. Install Python dependencies using pip
5. Copy source code to /opt folder
6. Run the web server using “flask” command

Building image

Dockerfile

```
FROM Ubuntu

RUN apt-get update


RUN pip install flask


COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

1. OS - Ubuntu
2. Update apt repo
3. Install dependencies using apt
4. Install Python dependencies using pip
5. Copy source code to /opt folder
6. Run the web server using “flask” command

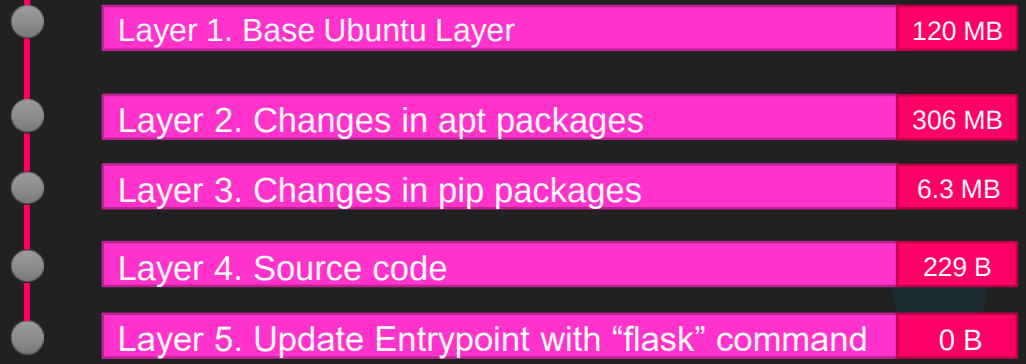
```
docker build Dockerfile –t my-custom-app
```

Layered Architecture

Dockerfile

```
FROM Ubuntu

RUN apt-get update && apt-get –y install python

RUN pip install flask flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

```
docker build -f Dockerfile –t my-custom-app .
```




```
root@osboxes:/root/simple-webapp-docker # docker history mmumshad/simple-webapp
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
1a45ba829f10        About an hour ago   /bin/sh -c #(nop)  ENTRYPOINT ["/bin/sh" "...   0B
37d37ed8fe99        About an hour ago   /bin/sh -c #(nop) COPY file:29b92853d73898...   229B
d6aaebf8ded0        About an hour ago   /bin/sh -c pip install flask flask-mysql        6.39MB
e4c055538e60        About an hour ago   /bin/sh -c apt-get update && apt-get insta...   306MB
ccc7a11d65b1        2 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>           2 weeks ago         /bin/sh -c mkdir -p /run/systemd && echo '...   7B
<missing>           2 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\...   2.76kB
<missing>           2 weeks ago         /bin/sh -c rm -rf /var/lib/apt/lists/*          0B
<missing>           2 weeks ago         /bin/sh -c set -xe   && echo '#!/bin/sh' >...   745B
<missing>           2 weeks ago         /bin/sh -c #(nop) ADD file:39d3593ea220e68...   120MB
```

Some common commands used in a Dockerfile

Command	Use Case
FROM	Specifies the base image to use for the Dockerfile.
MAINTAINER	Specifies the name and email address of the person maintaining the Dockerfile.
RUN	Runs a command in the shell or in the container.
CMD	Specifies the default command to run when the container starts.
EXPOSE	Specifies the ports to expose from the container.
ENV	Sets environment variables in the container.
ADD	Copies files and directories from the build context into the container.
COPY	Copies files and directories from the build context into the container.
ENTRYPOINT	Specifies the command to run when the container starts, and allows additional arguments to be passed to the command.
VOLUME	Creates a mount point for a volume to be shared between the container and the host.
USER	Sets the user or UID to run the container.
WORKDIR	Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, or ADD commands.

Docker build image output/logs

```
root@osboxes:/root/simple-webapp-docker # docker build .
Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM ubuntu
 ---> ccc7a11d65b1
Step 2/5 : RUN apt-get update && apt-get install -y python python-setuptools python-dev
 ---> Running in a7840dbfad17
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Get:4 http://security.ubuntu.com/ubuntu xenial-security/universe Sources [46.3 kB]
Get:5 http://archive.ubuntu.com/ubuntu xenial-backports InRelease [102 kB]
Get:6 http://security.ubuntu.com/ubuntu xenial-security/main amd64 Packages [440 kB]
Step 3/5 : RUN pip install flask flask-mysql
 ---> Running in a4a6c9190ba3
Collecting flask
  Downloading Flask-0.12.2-py2.py3-none-any.whl (83kB)
Collecting flask-mysql
  Downloading Flask_MySQL-1.4.0-py2.py3-none-any.whl
Removing intermediate container a4a6c9190ba3
Step 4/5 : COPY app.py /opt/
 ---> e7cdab17e782
Removing intermediate container faaaaf63c512
Step 5/5 : ENTRYPOINT FLASK_APP=/opt/app.py flask run --host=0.0.0.0
 ---> Running in d452c574a8bb
 ---> 9f27c36920bc
Removing intermediate container d452c574a8bb
Successfully built 9f27c36920bc
```

The caching system

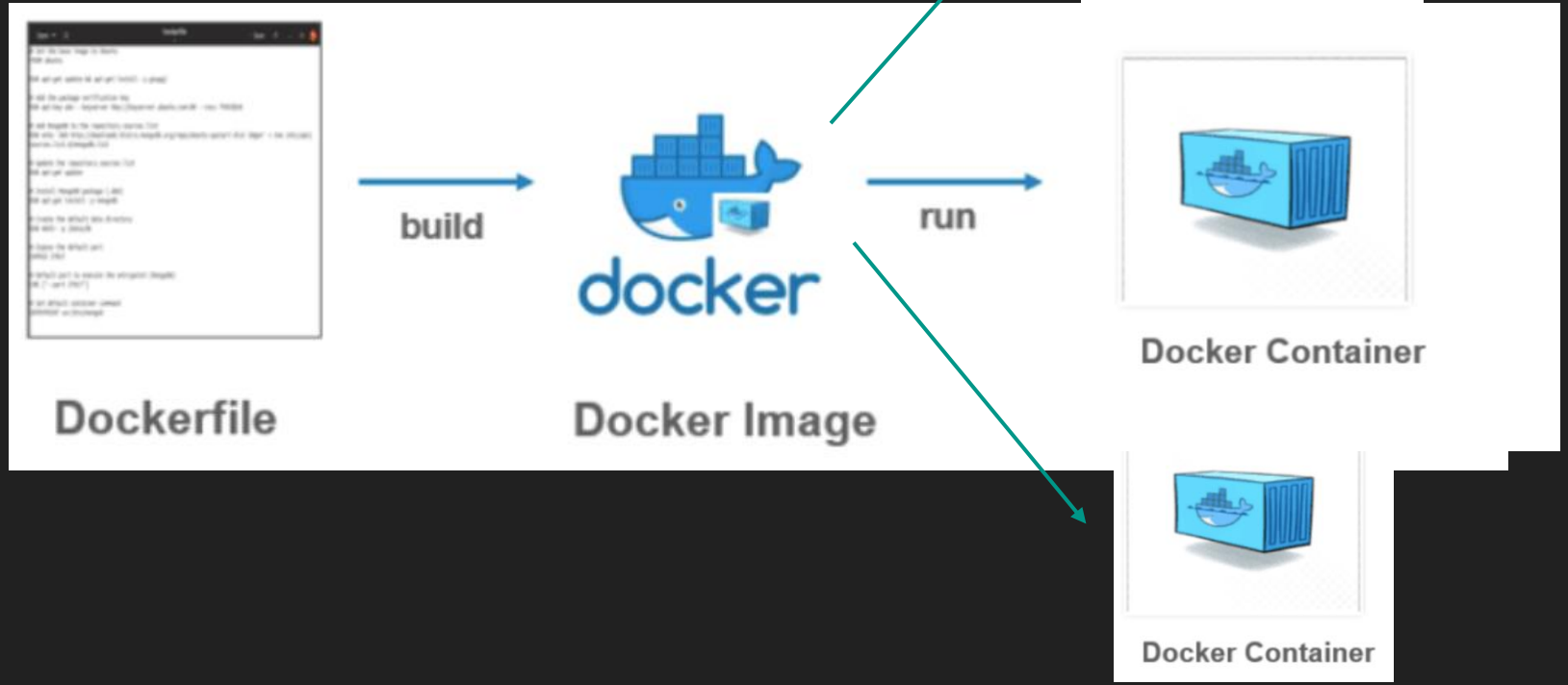
- If you run the same build again, it will be instantaneous.
- Why?
  - After each build step, Docker takes a snapshot of the resulting image.
  - Before executing a step, Docker checks if it has already built the same sequence.
- You can force a rebuild with docker build --no-cache ....

Running the container

```
docker run nginx
```

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
fc7181108d40: Already exists
d2e987ca2267: Pull complete
0b760b431b11: Pull complete

sha256:96fb261b66270b900ea5a2c17a26abbfabe95506e73c3a3c65869a6dbe83223a
Status: Downloaded newer image for nginx:latest
```

Process summary



d o c k e r

i m a ge s

What am I containerizing?

How to create my own image?

Dockerfile

```
FROM Ubuntu

RUN apt-get update


RUN pip install flask


COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

1. OS - Ubuntu
2. Update apt repo
3. Install dependencies using apt
4. Install Python dependencies using pip
5. Copy source code to /opt folder
6. Run the web server using “flask” command

```
docker build Dockerfile –tmy-custom-app .
```

Dockerfile

Dockerfile

```
INSTRUCTION ARGUMENT
```

Dockerfile

```
FROM Ubuntu

RUN apt-get update

RUN pip install flask

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

Start from a base OS or another image

Install all dependencies

Copy source code

Specify Entrypoint

Layered architecture

Dockerfile

```
FROM Ubuntu

RUN apt-get update && apt-get –y install python

RUN pip install flask flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

```
docker build Dockerfile –t mmumshad/my-custom-app
```

- Layer 1. Base Ubuntu Layer — 120 MB
- Layer 2. Changes in apt packages — 306 MB
- Layer 3. Changes in pip packages — 6.3 MB
- Layer 4. Source code — 229 B
- Layer 5. Update Entrypoint with “flask” command — 0 B

```
root@osboxes:/root/simple-webapp-docker # docker history mmumshad/simple-webapp
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
1a45ba829f10        About an hour ago   /bin/sh -c #(nop)  ENTRYPOINT ["/bin/sh" "...   0B
37d37ed8fe99        About an hour ago   /bin/sh -c #(nop) COPY file:29b92853d73898...   229B
d6aaebf8ded0        About an hour ago   /bin/sh -c pip install flask flask-mysql        6.39MB
e4c055538e60        About an hour ago   /bin/sh -c apt-get update && apt-get insta...   306MB
ccc7a11d65b1        2 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>           2 weeks ago         /bin/sh -c mkdir -p /run/systemd && echo '...   7B
<missing>           2 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\...   2.76kB
<missing>           2 weeks ago         /bin/sh -c rm -rf /var/lib/apt/lists/*          0B
<missing>           2 weeks ago         /bin/sh -c set -xe   && echo '#!/bin/sh' >...   745B
<missing>           2 weeks ago         /bin/sh -c #(nop) ADD file:39d3593ea220e68...   120MB
```

Docker build output

```
root@osboxes:/root/simple-webapp-docker # docker build .
Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM ubuntu
 ---> ccc7a11d65b1
Step 2/5 : RUN apt-get update && apt-get install -y python python-setuptools python-dev
 ---> Running in a7840dbfad17
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Get:4 http://security.ubuntu.com/ubuntu xenial-security/universe Sources [46.3 kB]
Get:5 http://archive.ubuntu.com/ubuntu xenial-backports InRelease [102 kB]
Get:6 http://security.ubuntu.com/ubuntu xenial-security/main amd64 Packages [440 kB]
Step 3/5 : RUN pip install flask flask-mysql
 ---> Running in a4a6c9190ba3
Collecting flask
  Downloading Flask-0.12.2-py2.py3-none-any.whl (83kB)
Collecting flask-mysql
  Downloading Flask_MySQL-1.4.0-py2.py3-none-any.whl
Removing intermediate container a4a6c9190ba3
Step 4/5 : COPY app.py /opt/
 ---> e7cdab17e782
Removing intermediate container faaaaf63c512
Step 5/5 : ENTRYPOINT FLASK_APP=/opt/app.py flask run --host=0.0.0.0
 ---> Running in d452c574a8bb
 ---> 9f27c36920bc
Removing intermediate container d452c574a8bb
Successfully built 9f27c36920bc
```

What can you containerize?



Containerize Everything!!!

docker
commands

images – List images

```
docker images
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
nginx               latest              f68d6e55e065        4 days ago          109MB
redis               latest              4760dc956b2d        15 months ago       107MB
ubuntu              latest              f975c5035748        16 months ago       112MB
alpine              latest              3fd9065eaf02        18 months ago       4.14MB
```

rmi – Remove images

```
docker rmi nginx
```

```
Untagged: nginx:latest
Untagged: nginx@sha256:96fb261b66270b900ea5a2c17a26abbfabe95506e73c3a3c65869a6dbe83223a
Deleted: sha256:f68d6e55e06520f152403e6d96d0de5c9790a89b4cfc99f4626f68146fa1dbdc
Deleted: sha256:1b0c768769e2bb66e74a205317ba531473781a78b77feef8ea6fd7be7f4044e1
Deleted: sha256:34138fb60020a180e512485fb96fd42e286fb0d86cf1fa2506b11ff6b945b03f
Deleted: sha256:cf5b3c6798f77b1f78bf4e297b27cfa5b6caa982f04caeb5de7d13c255fd7a1e
```

! Delete all dependent containers to remove image

pull – download an image

```
docker run nginx
```

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
fc7181108d40: Already exists
d2e987ca2267: Pull complete
0b760b431b11: Pull complete

sha256:96fb261b66270b900ea5a2c17a26abbfabe95506e73c3a3c65869a6dbe83223a
Status: Downloaded newer image for nginx:latest
```

```
docker pull nginx
```

```
Using default tag: latest

fc7181108d40: Pull complete
d2e987ca2267: Pull complete
0b760b431b11: Pull complete

sha256:96fb261b66270b900ea5a2c17a26abbfabe95506e73c3a3c65869a6dbe83223a
Status: Downloaded newer image for nginx:latest
```

run – start a container

```
docker run nginx
```

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
fc7181108d40: Already exists
d2e987ca2267: Pull complete
0b760b431b11: Pull complete

sha256:96fb261b66270b900ea5a2c17a26abbfabe95506e73c3a3c65869a6dbe83223a
Status: Downloaded newer image for nginx:latest
```

ps – list containers

```
docker ps
```

```
CONTAINER ID  IMAGE   COMMAND                  CREATED             STATUS          PORTS       NAMES
796856ac413d  nginx   "nginx -g 'daemon of…"   7 seconds ago       Up 6 seconds    80/tcp      silly_sammet
```

```
docker ps -a
```

```
CONTAINER ID   IMAGE    COMMAND                  CREATED             STATUS                      NAMES
796856ac413d   nginx    "nginx -g 'daemon of…"   7 seconds ago       Up 6 seconds                silly_sammet
cff8ac918a2f   redis    "docker-entrypoint.s…"   6 seconds ago       Exited (0) 3 seconds ago    relaxed_aryabhata
```

STOP – stop a container

```
docker ps
```

```
CONTAINER ID  IMAGE   COMMAND                  CREATED             STATUS          PORTS       NAMES
796856ac413d  nginx   "nginx -g 'daemon of…"   7 seconds ago       Up 6 seconds    80/tcp      silly_sammet
```

```
docker stop silly_sammet
```

```
silly_sammet
```

```
docker ps -a
```

```
CONTAINER ID   IMAGE     COMMAND                  CREATED             STATUS                      NAMES
796856ac413d   nginx     "nginx -g 'daemon of…"   7 seconds ago       Exited (0) 3 seconds ago    silly_sammet
cff8ac918a2f   redis     "docker-entrypoint.s…"   6 seconds ago       Exited (0) 3 seconds ago    relaxed_aryabhata
```

rm – Remove a container

```
docker rm silly_sammet
```

```
silly_sammet
```

```
docker ps -a
```

```
CONTAINER ID   IMAGE     COMMAND                  CREATED              STATUS                          NAMES
cff8ac918a2f   redis     "docker-entrypoint.s…"   6 seconds ago        Exited (0) 3 seconds ago        relaxed_aryabhata
```

```
docker run ubuntu
```

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
```

```
docker ps -a
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS
45aacca36850        ubuntu              "/bin/bash"         43 seconds ago      Exited (0) 41 seconds ago
```

Append a command

```
docker run ubuntu
```

```
docker run ubuntu sleep 5
```

Exec – execute a command in container

```
docker ps -a
```

```
CONTAINER ID        IMAGE     COMMAND      CREATED             STATUS                      NAMES
```

```
docker exec distracted_mcclintock cat /etc/hosts
```

```
127.0.0.1       localhost

fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes

172.18.0.2      538d037f94a7
```

Run – attach and detach

```
docker run kodekloud/simple-
```

```
This is a sample web application that displays a colored background.
* Serving Flask app "app" (lazy loading)
```

```
docker run -d kodekloud/simple-
```

```
a043d40f85fefa414254e4775f9336ea59e19e5cf597af5c554e0a35a1631118
```

```
docker attach a043d
```

docker run

run – tag

```
docker run redis
```

```
Using default tag: latest

f5d23c7fed46: Pull complete
Status: Downloaded newer image for redis:latest

1:C 31 Jul 2019 09:02:32.624 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
1:C 31 Jul 2019 09:02:32.624 # Redis version=5.0.5, bits=64, commit=00000000, modified=0, pid=1, just started
1:M 31 Jul 2019 09:02:32.626 # Server initialized
```

```
docker run redis:4.0
```

TAG

```
Unable to find image 'redis:4.0' locally
4.0: Pulling from library/redis

Status: Downloaded newer image for redis:4.0

1:C 31 Jul 09:02:56.527 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
1:C 31 Jul 09:02:56.527 # Redis version=4.0.14, bits=64, commit=00000000, modified=0, pid=1, just started
1:M 31 Jul 09:02:56.530 # Server initialized
```

Docker ports

- Allows you to map a container's internal network ports to a host's external ports.
- Enables communication between containers and applications running on the host system.
- Port mapping is specified when running a container using the -p flag or the --publish flag.
- The -p flag takes the format **<host port>:<container port>** where <host port> is the port on the host system and <container port> is the port on the container.
- CAUTION: Ensure that the ports you're mapping are not already in use by other applications on the host system.

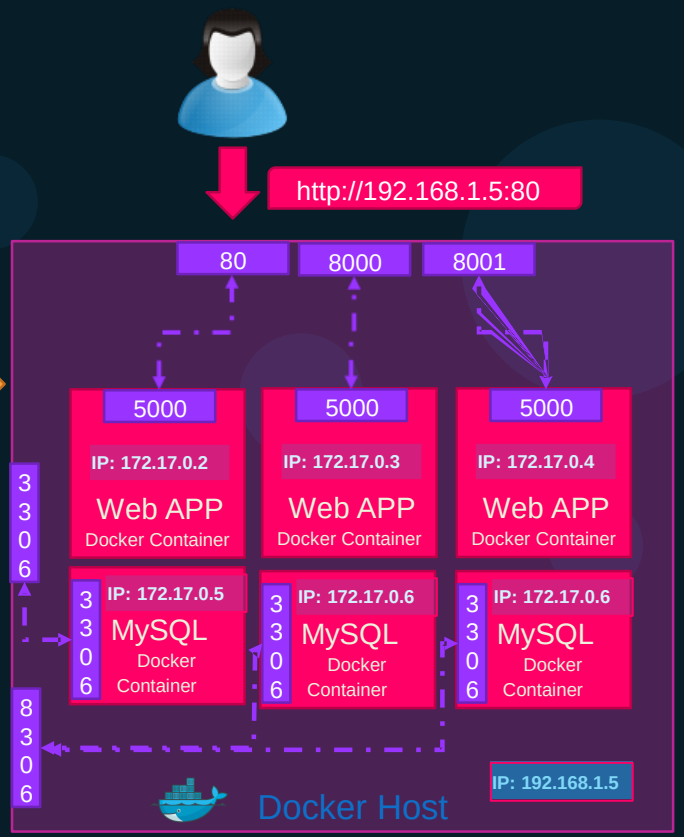
run – PORT mapping

```
docker run kodekloud/webapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

http://172.17.0.2:5000 → Internal IP

```
docker run -p 80:5000 simple-webapp
docker run -p 8000:5000 simple-webapp
docker run -p 8001:5000 simple-webapp
docker run -p 3306:3306 mysql
docker run -p 8306:3306 mysql
docker run -p 8306:3306 mysql
```

```
root@osboxes:/root # docker run -p 8306:3306 -e MYSQL_ROOT_PASSWORD=pass mysql
docker: Error response from daemon: driver failed programming external connectivity on endpoint boring_bhabha (
5079d342b7e8ee11c71d46): Bind for 0.0.0.0:8306 failed: port is already allocated.
```

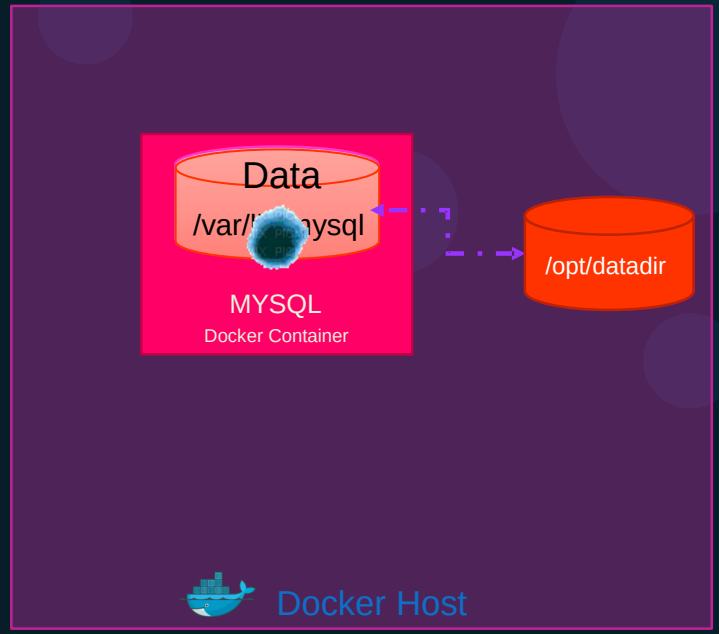


RUN – Volume mapping

```
docker run mysql
```

```
docker stop mysql
docker rm mysql
```

```
docker run -v /opt/datadir:/var/lib/mysql mysql
```



Docker Volume

- Enable data persistence and sharing between Docker containers and the host system.
- Docker volumes are separate from container file systems and are mounted into containers at specific mount points.
- Benefits:
  - Easy data sharing
  - Data persistence
  - Performance

Use Cases

- Data storage for databases
- Sharing configuration files
- Persisting log data

Docker Volume Types

1. Host-mounted Volumes
2. Named Volumes

Host-mounted Volumes

- Host-mounted volumes allow a container to access the filesystem of the host system.
- Example:
  - $ docker run -v /path/on/host:/path/in/container image_name
- In this example, the /path/on/host directory on the host system is mounted as /path/in/container in the container. The image_name is the name of the Docker image you want to run.

Named Volumes

- Named volumes are created and managed by Docker.
- Example:
  - $ docker run -v volume_name:/path/in/container image_name
- In this example, a named volume named **volume_name** is created and mounted at /path/in/container in the container.

Inspect Container

```
docker inspect blissful_hopper
```

```
[
    {
        "Id": "35505f7810d17291261a43391d4b6c0846594d415ce4f4d0a6ffbf9cc5109048",
        "Name": "/blissful_hopper",
        "Path": "python",

            "app.py"
        ],
        "State": {

            "Running": true,
        },

        "Mounts": [],
        "Config": {

                "python",
                "app.py"
            ],
        },
        "NetworkSettings": {..}
      }
]
```

Container Logs

```
docker logs blissful_hopper
```

```
 This is a sample web application that displays a colored background.
 A color can be specified in two ways.

 1. As a command line argument with --color as the argument. Accepts one of
red,green,blue,blue2,pink,darkblue

red,green,blue,blue2,pink,darkblue

 Note: Command line argument precedes over environment variable.


No command line argument or environment variable. Picking a Random Color =blue
 * Serving Flask app "app" (lazy loading)
 * Environment: production

   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

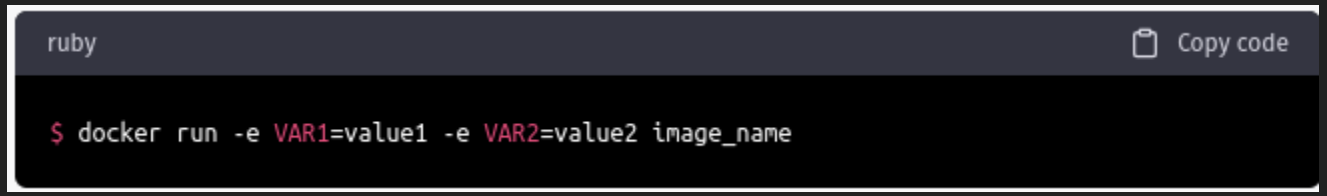
docker environment variables

Environment variables

- Environment variables are commonly used in Docker to configure applications and services running inside containers.

Setting up Environment variables

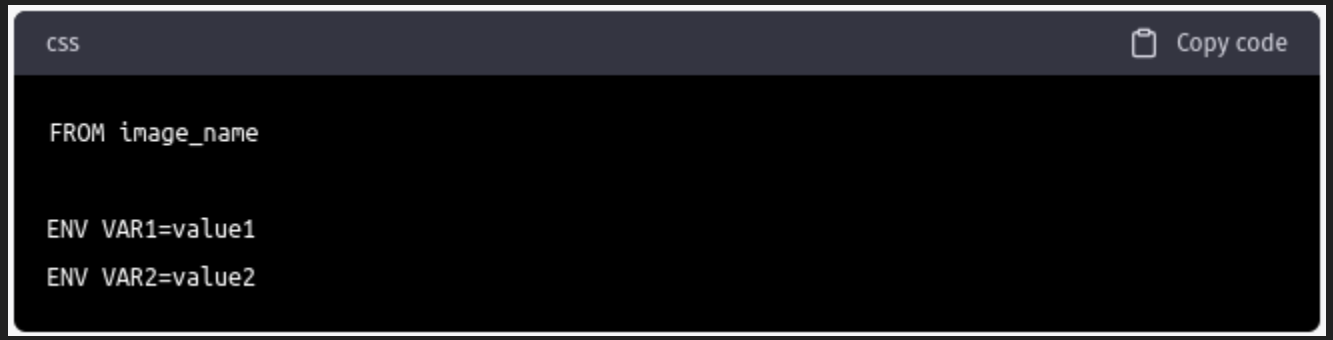
- Using the -e option in the docker run command: This option allows you to pass one or more environment variables to the container when it is started. Here's an example:



- In this example, the VAR1 and VAR2 environment variables are set to value1 and value2, respectively.

Setting up Environment variables

- Using a Dockerfile: You can set environment variables in a Dockerfile using the ENV instruction. Here's an example:



- In this example, the VAR1 and VAR2 environment variables are set to value1 and value2, respectively.

Setting up Environment variables

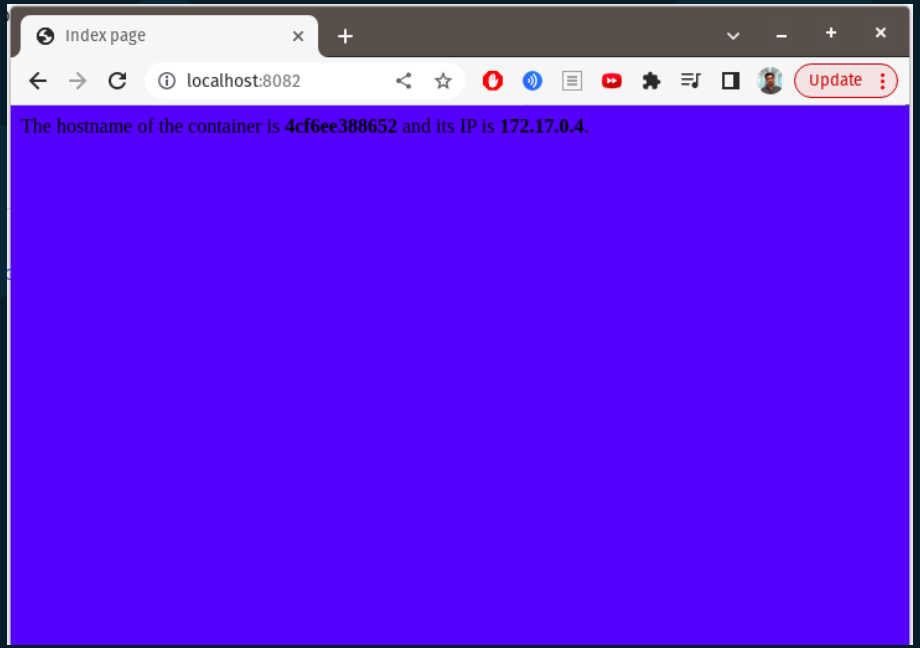
- Using an environment file: You can create an environment file containing environment variable definitions, and then use the --env-file option in the docker run command to pass the variables to the container. Here's an example:

- In this example, the VAR1 and VAR2 environment variables are set to value1 and value2, respectively, using the myenvfile file.

Environment Variables

app.py

```python
from flask import Flask,render_template
import socket

app = Flask(__name__)

@app.route("/")
def index():
    try:
        host_name = socket.gethostname()
        host_ip = socket.gethostbyname(host_name)
        return render_template('index.html', hostname=host_name,
            ip=host_ip, color="blue")
    except:
        return render_template('error.html')


if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8080)
```




```
python app.py
```

Environment Variables

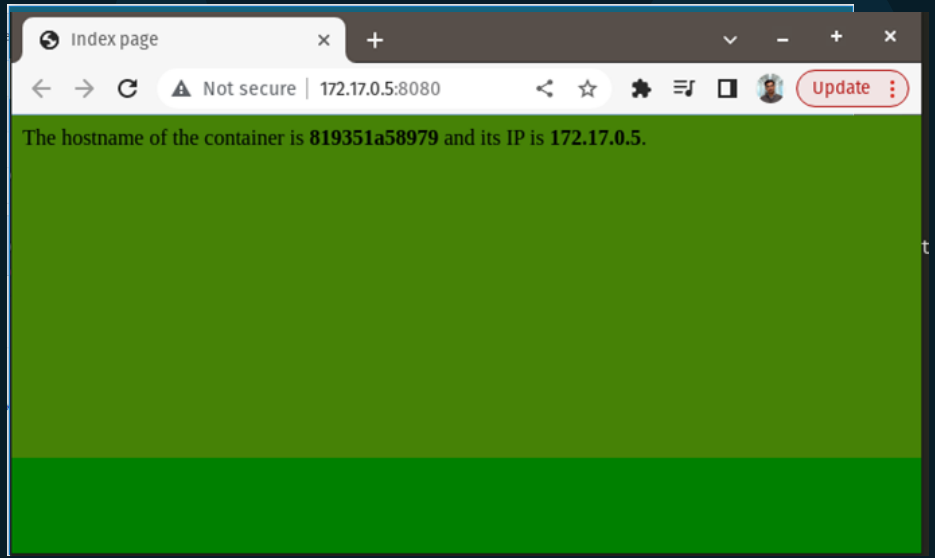
app.py

```python
from flask import Flask,render_template
import socket
import os

app = Flask(__name__)

color = os.environ.get('APP_COLOR')

@app.route("/")
def index():
    try:
        host_name = socket.gethostname()
        host_ip = socket.gethostbyname(host_name)
        return render_template('index.html', hostname=host_name,
            ip=host_ip, color=color)
    except:
        return render_template('error.html')


if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8080)

```

Environment Variables

app.py

```python
from flask import Flask,render_template
import socket
import os

app = Flask(__name__)

color = os.environ.get('APP_COLOR')

@app.route("/")
def index():
    try:
        host_name = socket.gethostname()
        host_ip = socket.gethostbyname(host_name)
        return render_template('index.html', hostname=host_name,
            ip=host_ip, color=color)
    except:
        return render_template('error.html')


if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8080)

```

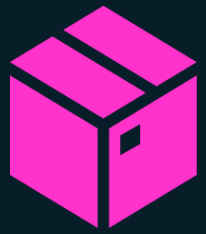


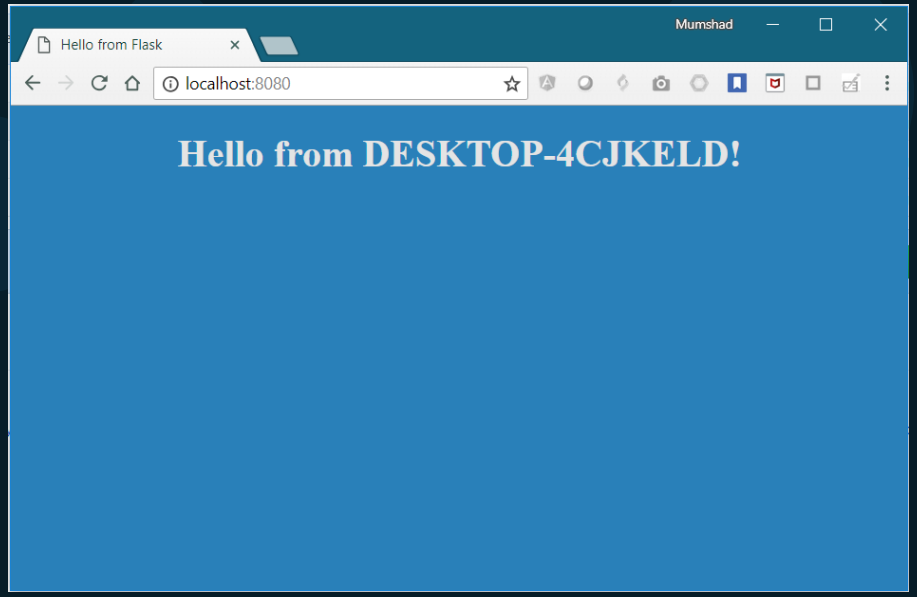

```
export APP_COLOR=green; python app.py
```

ENV Variables in Docker

app.py

```python
import os
from flask import Flask

app = Flask(__name__)
color = os.environ.get('APP_COLOR')


@app.route("/")
def main():
  print(color)

  return
  render_template('hello.html',
  color=color)

if __name__ == "__main__":
  app.run(host="0.0.0.0", port="8080")
```

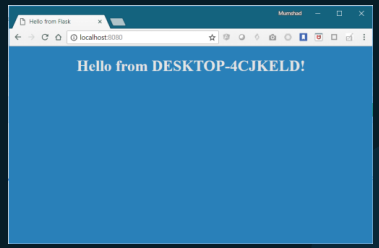




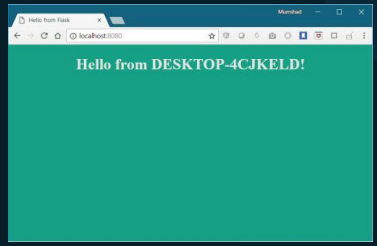

```
docker run -e APP_COLOR=blue simple-webapp-color
```

ENV Variables in Docker

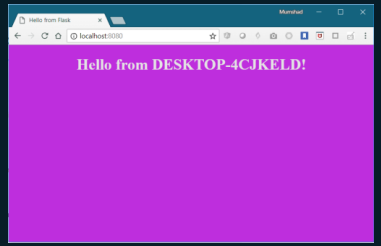
```
docker run -e APP_COLOR=blue     simple-webapp-color
```




```
docker run  -e APP_COLOR=green simple-webapp-color
```




```
docker run  -e APP_COLOR=pink simple-webapp-color
```



Inspect Environment Variable

```
docker inspect blissful_hopper
```

```
[
    {
        "Id": "35505f7810d17291261a43391d4b6c0846594d415ce4f4d0a6ffbf9cc5109048",
        "State": {

            "Running": true,
        },

        "Mounts": [],
        "Config": {
            "Env": [
                "APP_COLOR=blue",

                "GPG_KEY=0D96DF4D4110E5C43FBFB17F2D347EA6AA65421D",
                "PYTHON_VERSION=3.6.6",

            ],
            "Entrypoint": [
                "python",

            ],
      }
]
```

docker

CMD vs ENTRYPOINT

Environment variables

- CMD and ENTRYPOINT are two important Dockerfile instructions that are used to define the default command to be executed when a container is started.
- While both of these instructions are used for similar purposes, they have some important differences that are important to understand.

CMD instruction

- The CMD instruction is used to define the default command that should be run when a container is started.
- You can specify the command in the form of a string or an array.
- You can override the default command by specifying a new command when you start the container using the docker run command.

CMD instruction

vbnet Copy code

```
FROM ubuntu:latest

# Set the default command to run when the container starts
CMD ["echo", "Hello, World!"]
```

Overriding CMD instruction

bash Copy code

```
docker run myapp echo "Hello, Docker!"
```

ENTRYPOINT instruction

- The ENTRYPOINT instruction is used to define the default command that should be run when a container is started, just like CMD.
- However, the command specified with ENTRYPOINT is not easily overridden by the docker run command.

- You can use the CMD instruction to provide additional arguments or options to the command specified with ENTRYPOINT.

ENTRYPOINT instruction

vbnet Copy code

```
FROM ubuntu:latest

# Set the default command to run when the container starts
ENTRYPOINT ["echo", "Hello,"]

# Provide additional arguments to the command
CMD ["World!"]
```

Feature	CMD	ENTRYPOINT
Purpose	Specifies the default command to run in a container	Configures the command and parameters that are always executed when the container is run
Override	Can be overridden with command-line arguments	Arguments provided at runtime are appended to the `ENTRYPOINT` instead of overriding it
Multiple	Can be specified multiple times	Can be specified multiple times, and the last `ENTRYPOINT` directive is used
Form	Shell form or Exec form	Exec form
Syntax	CMD ["executable","param1","param2"]	ENTRYPOINT ["executable","param1","param2"]
Execution	The command is executed as-is	The command is executed as the main process in the container, and additional arguments can be appended to it
Default	If no `CMD` is specified, the default is an empty command	If no `ENTRYPOINT` is specified, the default is `/bin/sh -c`
Best Practice	Prefer using `CMD` for defining the main command	Prefer using `ENTRYPOINT` to define a fixed command or application, with `CMD` used for providing optional arguments

d o c k e r

networking

Docker Networking

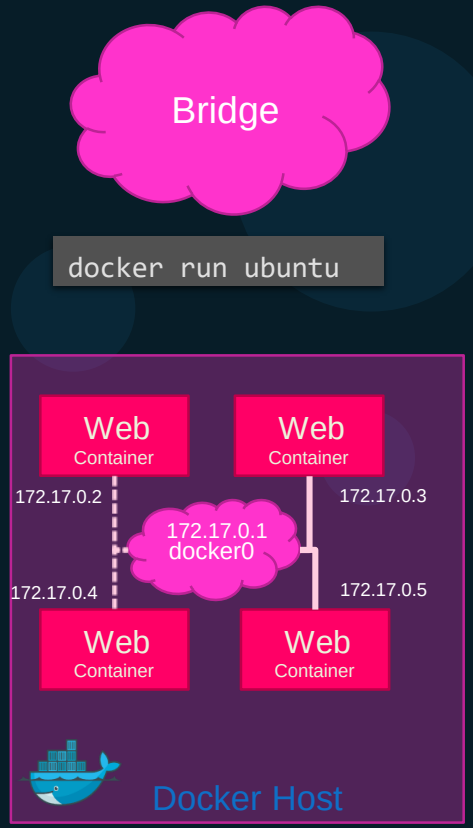
- Docker runs applications inside of containers, and these need to communicate over lots of different networks.
- This means Docker needs strong networking capabilities.
- Fortunately, Docker has solutions for container-to-container networks, as well as connecting to existing networks and VLANs.
- To create a smooth out-of-the-box experience, Docker ships with a set of native drivers that deal with the most common networking requirements.
- These include single-host bridge networks, multi-host overlays, and options for plugging into existing VLANs.

Docker Networking

- In Docker, each container has its own IP address and network stack, which means that it can be treated as a separate entity on the network.
- Docker networking can be categorized into three types:
  a. Bridge networking
  b. Host networking
  c. Overlay networking
- Will discuss Bridge and Host networking

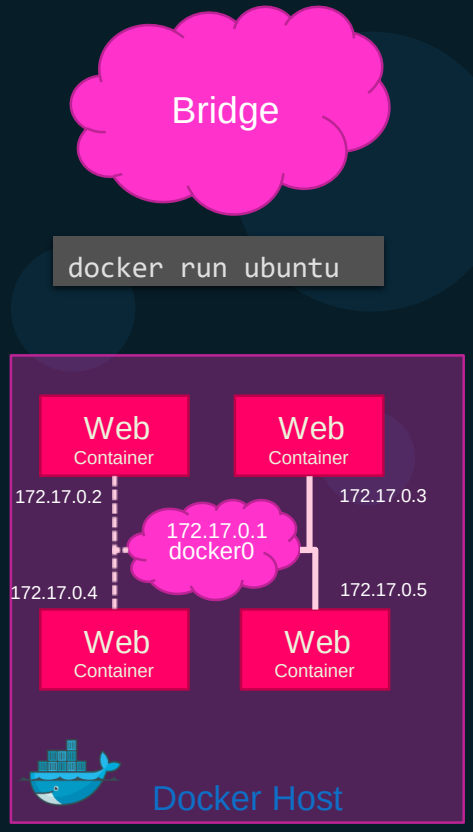
Bridge Networking:

- A bridge network is essentially a virtual switch that connects multiple containers together.
- Docker creates a new bridge network for each container by default, but you can also create your own custom bridge networks to group together specific containers.



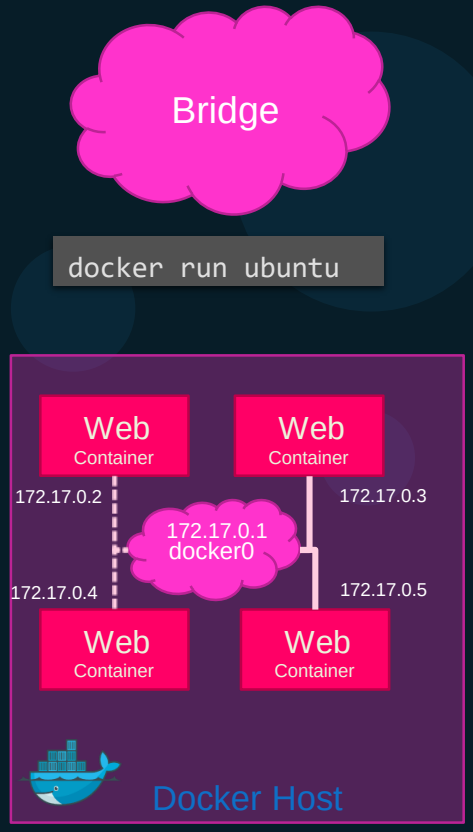
Bridge Networking:

- Bridge networking is the default networking mode in Docker, and it allows multiple containers to be connected to the same virtual network.
- When you run a Docker container, it is automatically connected to a default bridge network.
- Each container on this network has a unique IP address, and containers can communicate with each other using this IP address or by their container names if they are part of the same Docker network.



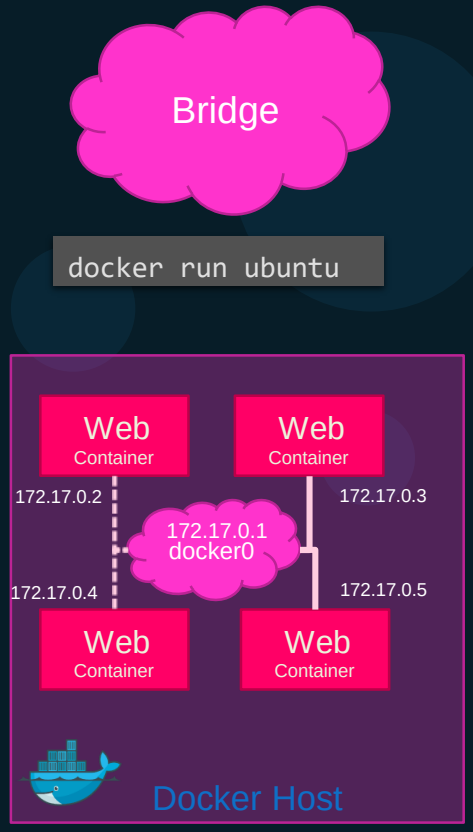
Bridge Networking:

- When a container is started on a bridge network, it is assigned an IP address from the range of IP addresses specified by the network.
- By default, the bridge network uses the subnet 172.17.0.0/16, and each container is assigned an IP address within this subnet.
- For example, if the first container on the network is assigned the IP address 172.17.0.2, the second container will be assigned 172.17.0.3, and so on.



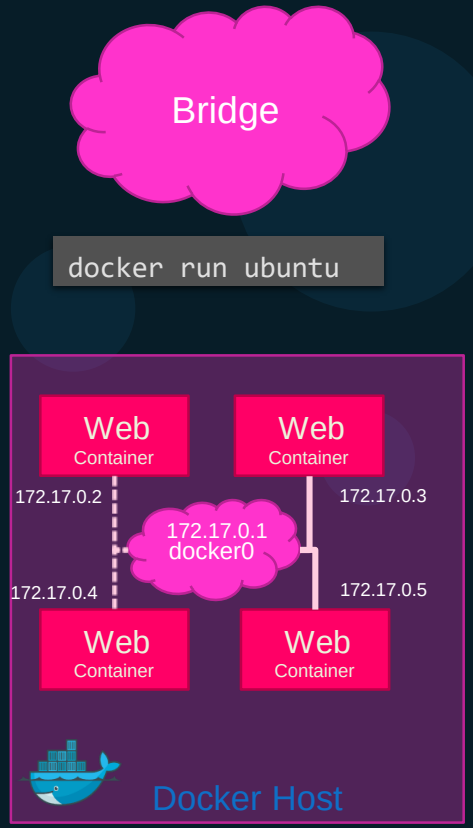
Bridge Networking:

- Containers on the same bridge network can communicate with each other using their IP addresses or container names.
- For example, if you have two containers named "web" and "db" on the same bridge network, the "web" container can connect to the "db" container using the hostname "db".



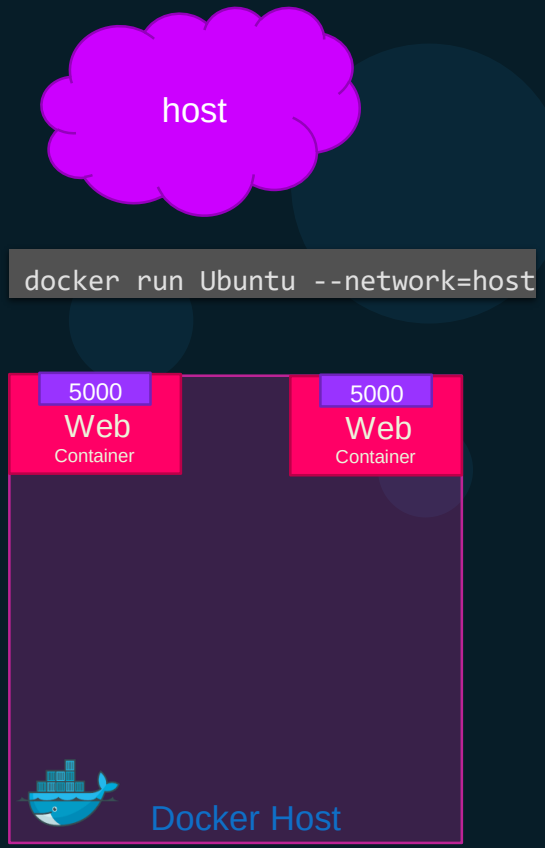
Bridge Networking:

- In addition to the default bridge network, Docker provides several other network drivers that can be used for different use cases, such as host networking and overlay networking.
- You can also create custom bridge networks and configure them with specific IP address ranges, DNS servers, and other options.



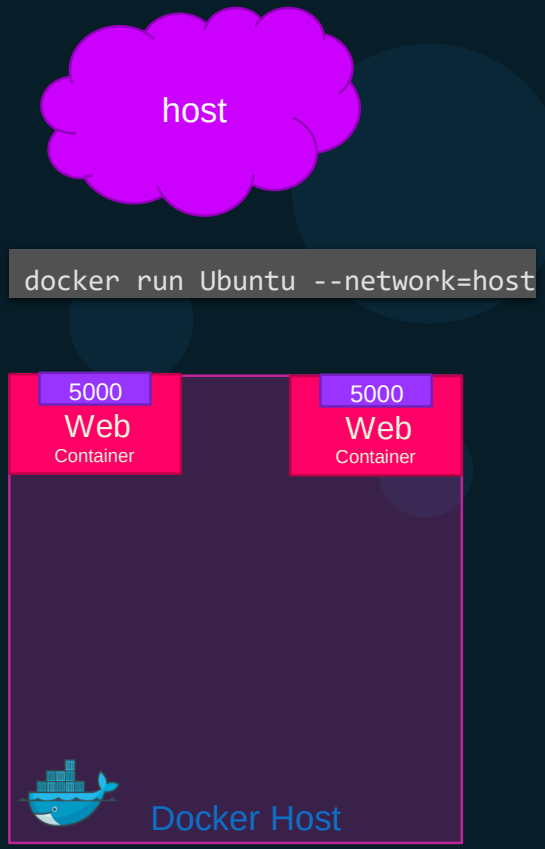
Host Networking:

- In host networking mode, Docker containers share the same network namespace as the host machine, meaning that they share the same IP address and network interface.
- This means that containers can communicate with other containers and external resources on the network using the host's network stack, without having to use port mapping or NAT.



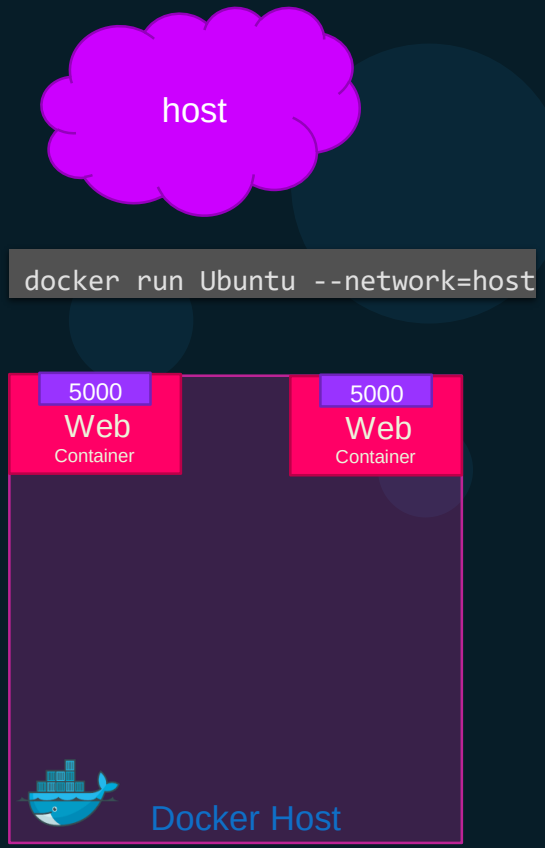
Host Networking:

- When you start a container in host networking mode, Docker doesn't create a new network namespace for the container.
- Instead, the container uses the same network namespace as the host, allowing it to communicate directly with other network devices and services on the host machine.
- This means that you can access services running on the host machine, such as a database or a web server, from inside a container using the same IP address and port as on the host.



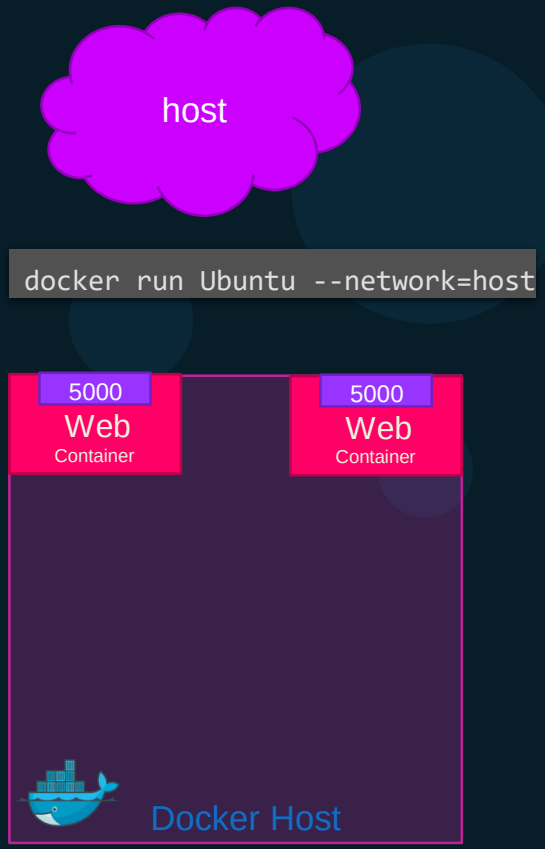
Host Networking:

- One advantage of host networking mode is that it can provide better performance and lower latency than bridge networking mode, since there is no overhead associated with NAT or port mapping.
- However, it also means that the container is exposed to the same network security risks and issues as the host machine, so it is important to ensure that the host machine is properly secured and isolated from potential threats.



Host Networking:

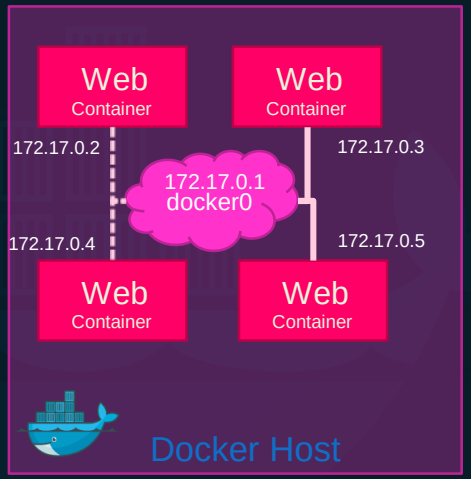
- Host networking mode can be useful in situations where you need to run a container that requires direct access to the host machine's network stack, such as when running network monitoring or troubleshooting tools.
- It can also be useful in cases where you need to run containers that require access to services running on the host machine.



Default networks

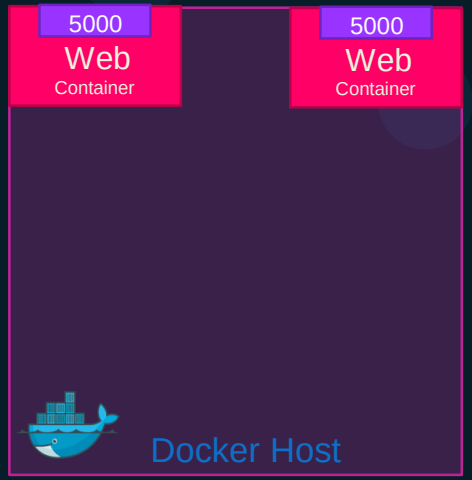
Bridge

```
docker run ubuntu
```



host

```
docker run Ubuntu --network=host
```



Overlay networking in Docker allows containers running on different hosts to communicate with each other seamlessly, even across multiple Docker Swarm nodes. Here's an explanation of overlay networking in Docker using bullet points:

- Overlay networking is a networking method provided by Docker for connecting containers across multiple Docker hosts or Swarm nodes.
- It enables container-to-container communication across different hosts by creating a virtual network overlay on top of the existing physical network infrastructure.
- Overlay networks are typically used in container orchestration scenarios, such as Docker Swarm, where containers are distributed across multiple nodes.
- To create an overlay network, you can use the Docker CLI or Docker Compose, specifying the driver as "overlay" and providing a unique network name.
- Once the overlay network is created, you can attach containers to the network by specifying the network name during container creation.
- Containers within the same overlay network can communicate with each other using their container names or service names, regardless of the underlying host or node they are running on.
- Overlay networking uses an encapsulation technique to encapsulate network traffic between containers in a secure and isolated manner.
- Docker Swarm manages the routing and load balancing of traffic between containers within the overlay network, ensuring efficient and reliable communication.
- Overlay networks support automatic service discovery and DNS resolution, allowing containers to refer to each other using their service names.
- Overlay networking in Docker provides features like multi-host networking, load balancing, and service discovery, making it suitable for building distributed and scalable applications.

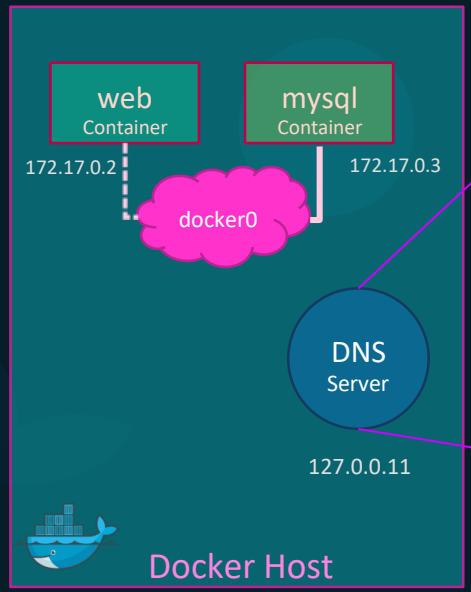
Inspect Network

```
docker inspect blissful_hopper
```

```
[
    {
        "Id": "35505f7810d17291261a43391d4b6c0846594d415ce4f4d0a6ffbf9cc5109048",
        "Name": "/blissful_hopper",
        "NetworkSettings": {

            "Gateway": "172.17.0.1",
            "IPAddress": "172.17.0.6",
            "MacAddress": "02:42:ac:11:00:06",

                "bridge": {

                    "IPAddress": "172.17.0.6",
                    "MacAddress": "02:42:ac:11:00:06",
                }
            }
    }
]
```

Embedded DNS

```
mysql.connect(  mysql       )
```

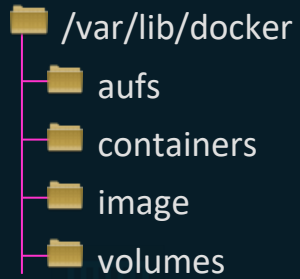


Host	IP
web	172.17.0.2
mysql	172.17.0.3

d o c k e r

sto ra ge

File system



Layered architecture

Dockerfile

```
FROM Ubuntu

RUN apt-get update && apt-get -y install python

RUN pip install flask flask-
mysql
COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask
run
```

```
docker build Dockerfile -t my-custom-app
```

Layer 1. Base Ubuntu Layer	120 MB
Layer 2. Changes in apt packages	306 MB
Layer 3. Changes in pip packages	6.3 MB
Layer 4. Source code	229 B
Layer 5. Update Entrypoint	0 B

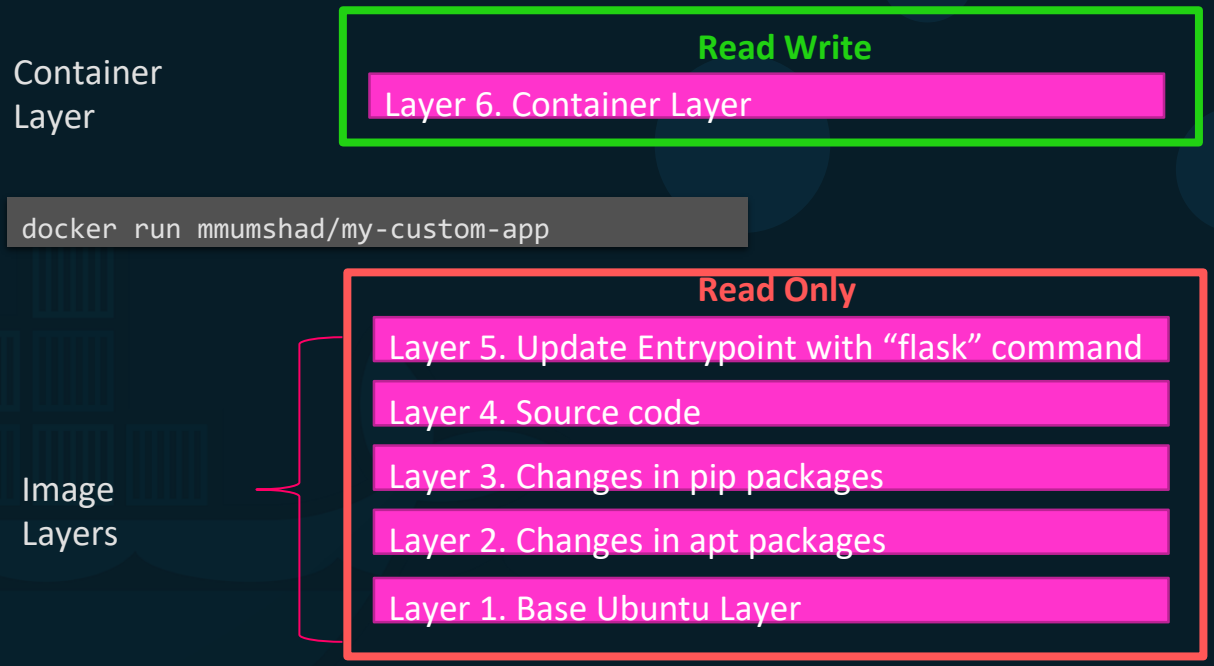
Dockerfile2

```
FROM Ubuntu

RUN apt-get update && apt-get -y install python

RUN pip install flask flask-
mysql
COPY app2.py /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app2.py flask
run
```

```
docker build Dockerfile2 -t my-custom-app-2
```

Layer 1. Base Ubuntu Layer	0 MB
Layer 2. Changes in apt packages	0 MB
Layer 3. Changes in pip packages	0 MB
Layer 4. Source code	229 B
Layer 5. Update Entrypoint	0 B

Layered architecture



COPY-ON-WRITE

Copy-on-write (CoW) is a storage optimization technique used in Docker to efficiently manage and store file system changes within containers. Here's an explanation of copy-on-write in Docker:

- In Docker, containers are created from images. Each image consists of one or more layers that represent the file system changes made during the image's creation.
- When a container is started from an image, Docker creates a thin writable layer called the container layer or the container's writeable layer.
- The container layer is created using the copy-on-write mechanism, which allows efficient use of disk space by sharing data across multiple containers.
- Copy-on-write works by maintaining a read-only base image layer and a separate container layer for each running container. The base image layer is shared among containers that use the same image.
- When a container modifies a file or directory within its container layer, Docker only writes the changed data to the container layer, while the unchanged data remains in the base image layer.
- As a result, the container layer only contains the differences or modifications made by the container, which helps reduce the storage space required for each container.
- Each container has its own isolated copy-on-write layer, allowing containers to make changes independently without affecting other containers or the base image.
- This copy-on-write mechanism enables fast container creation and startup times since Docker only needs to create the container layer and apply the necessary changes from the base image.
- Additionally, copy-on-write allows for efficient sharing of resources among containers and reduces disk I/O and storage requirements.

volumes

```
docker volume create data_volume
```

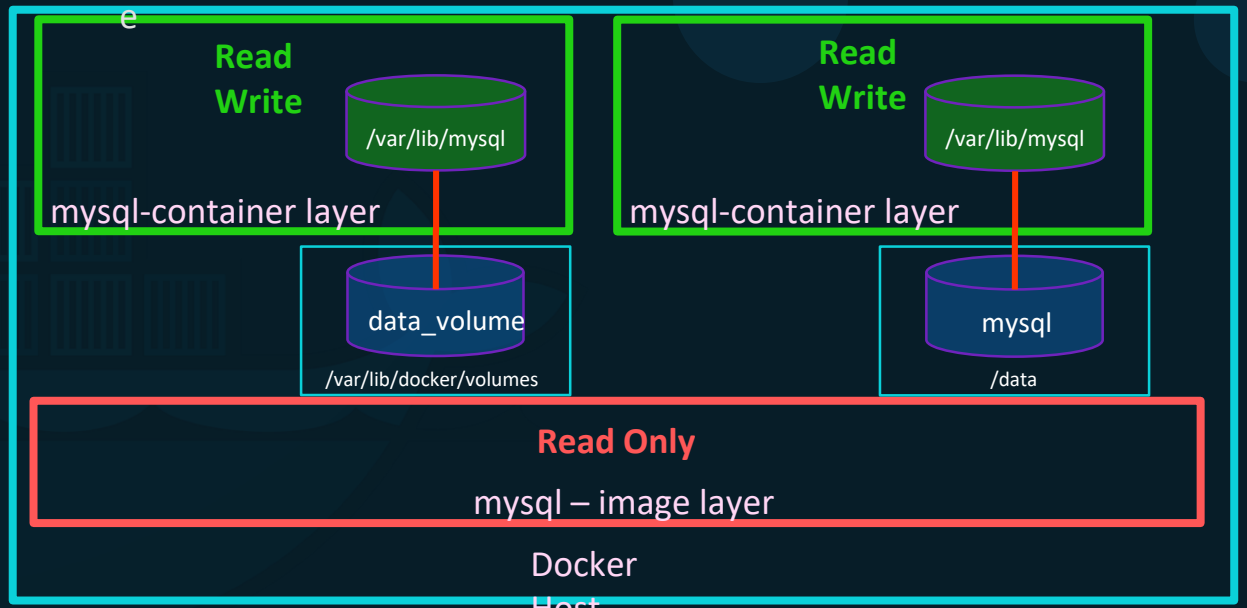
- /var/lib/docker
  - volumes
    - data_volume

```
docker run –v data_volume:/var/lib/mysql mysql
```

```
docker run –v data_volume2:/var/lib/mysql mysql
```

```
docker run –v /data/mysql:/var/lib/mysql mysql
```

```
docker run \
```




d o c k e r
registry

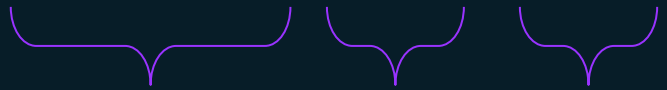
Image

```
docker run nginx
```

docker.i
Docker Hub

Image

```
image:  docker.io/nginx/nginx
```



Registry

User/ Account

Image/ Repository

```
gcr.io/ kubernetes-e2e-test-images/dnsutils
```

Private Registry

```
docker login private-registry.io
```

```
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to
https://hub.docker.com to create one.

Password:


Login Succeeded
```

```
docker run private-registry.io/apps/internal-app
```

LAB: Docker Registry

- Setting up docker registry
- Login
- Pushing and pulling image

The End