

# Introduction to Docker



Franco Arcieri

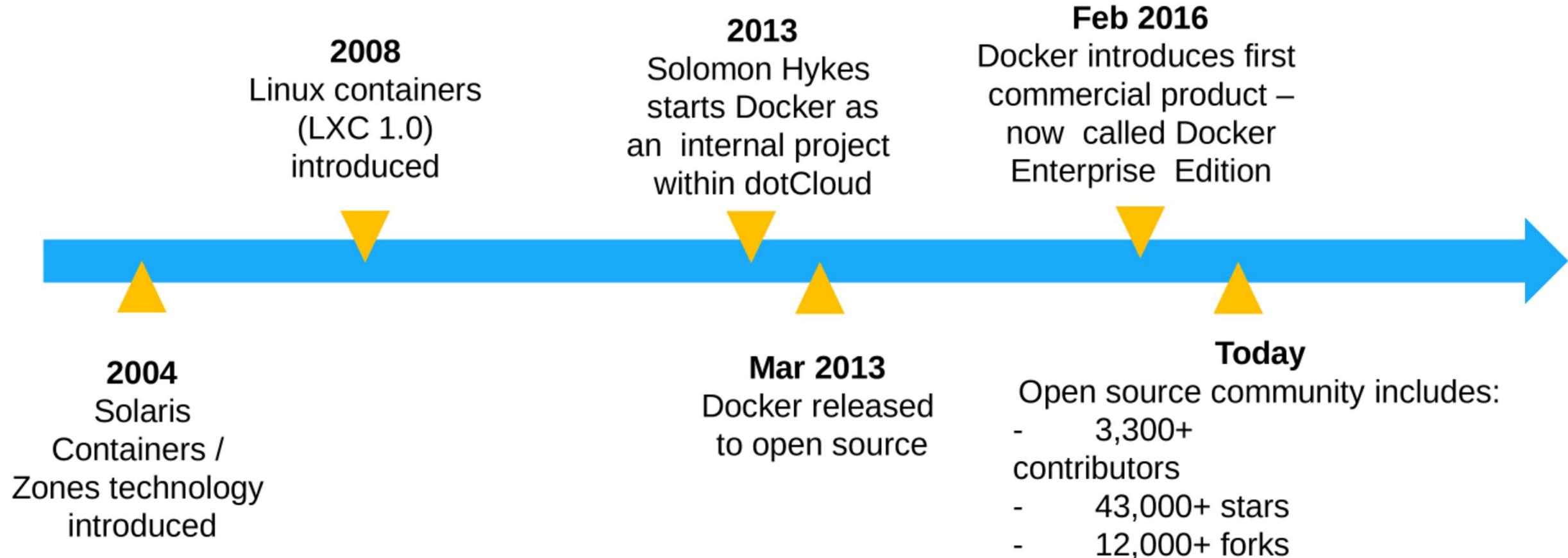
# L'esigenza di un container

- Gli sviluppatori lavorano nel proprio ambiente di sviluppo.
- Ad esempio voi state sviluppando:
  - Ubuntu 22.04
  - Processore Intel (sse, sse3, sse4, avx2, ..aes, ...)
  - Ext4 (indirizzamento a 64 bit, uso di file sparsificati, uso di link logici e fisici, ...)
  - Versione di java: es: la 17
  - Kernel 6.5
  - .....

# L'esigenza di un container

- Un'applicazione sviluppata nel vostro ambiente di sviluppo, sicuramente potrà essere messa in esecuzione su un ambiente «target» identico al vostro ambiente di sviluppo ma:
  - Funzionerebbe con una JRE 9?
  - Funzionerebbe sotto ubuntu 18.04 oppure 24.04?
  - Funzionerebbe su un Apple MAC?
  - Funzionerebbe sotto windows 7, 8, 10, 11?

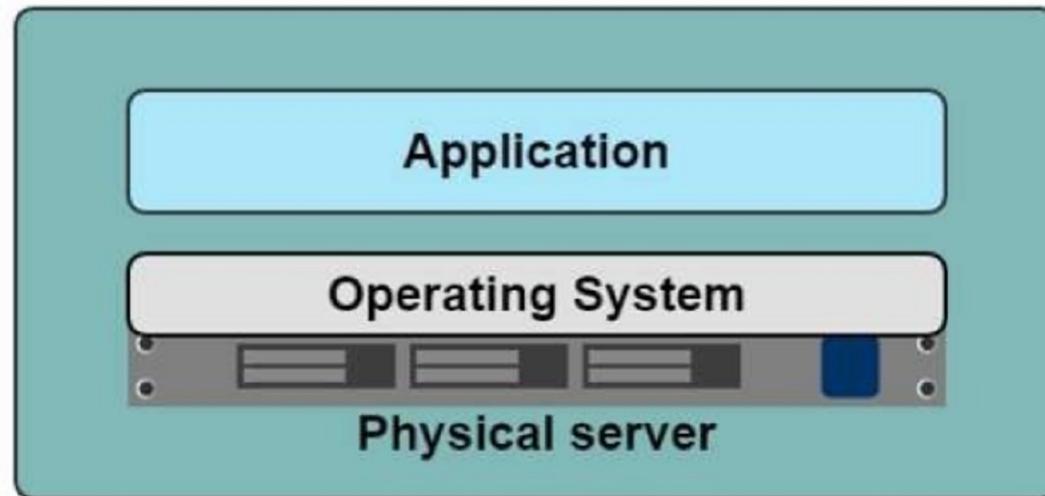
# History of Docker



# A History Lesson

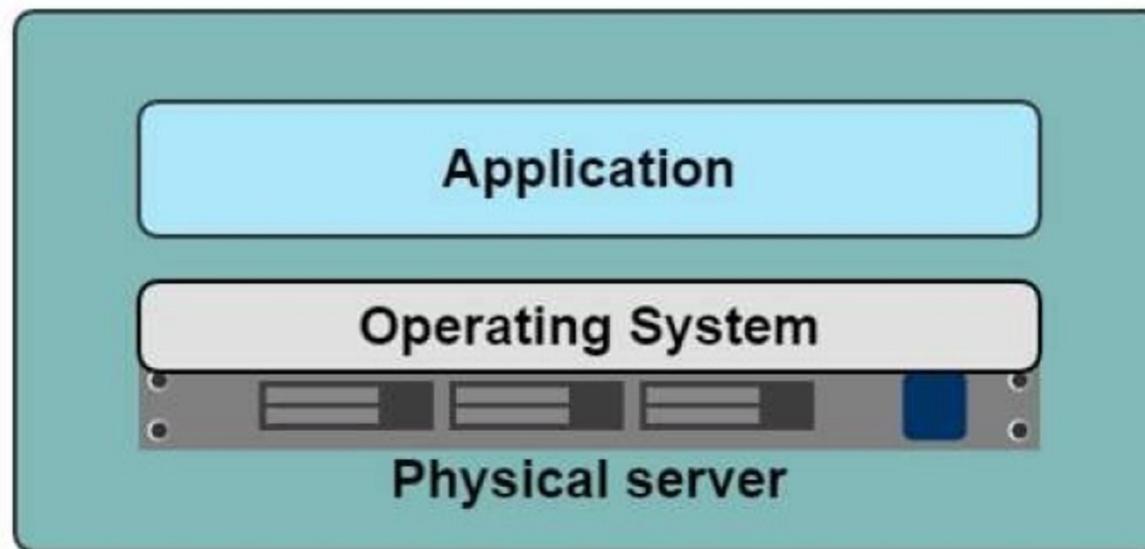
In the Dark Ages

**One application on one physical server**



# Historical limitations of application deployment

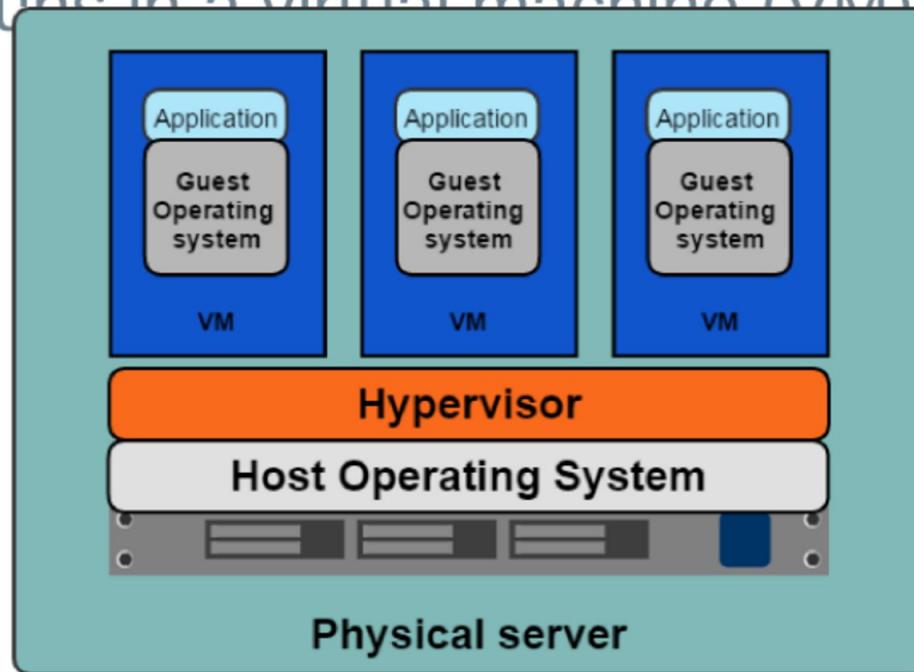
- Slow deployment times
- Huge costs
- Wasted resources
- Difficult to scale
- Difficult to migrate
- Vendor lock in



# A History Lesson

## Hypervisor-based Virtualization

- One physical server can contain multiple applications
- Each application runs in a virtual machine (VM)



# Benefits of VMs

- Better resource pooling
  - One physical machine divided into multiple virtual machines
- Easier to scale
- VMs in the cloud
  - Rapid elasticity
  - Pay as you go model

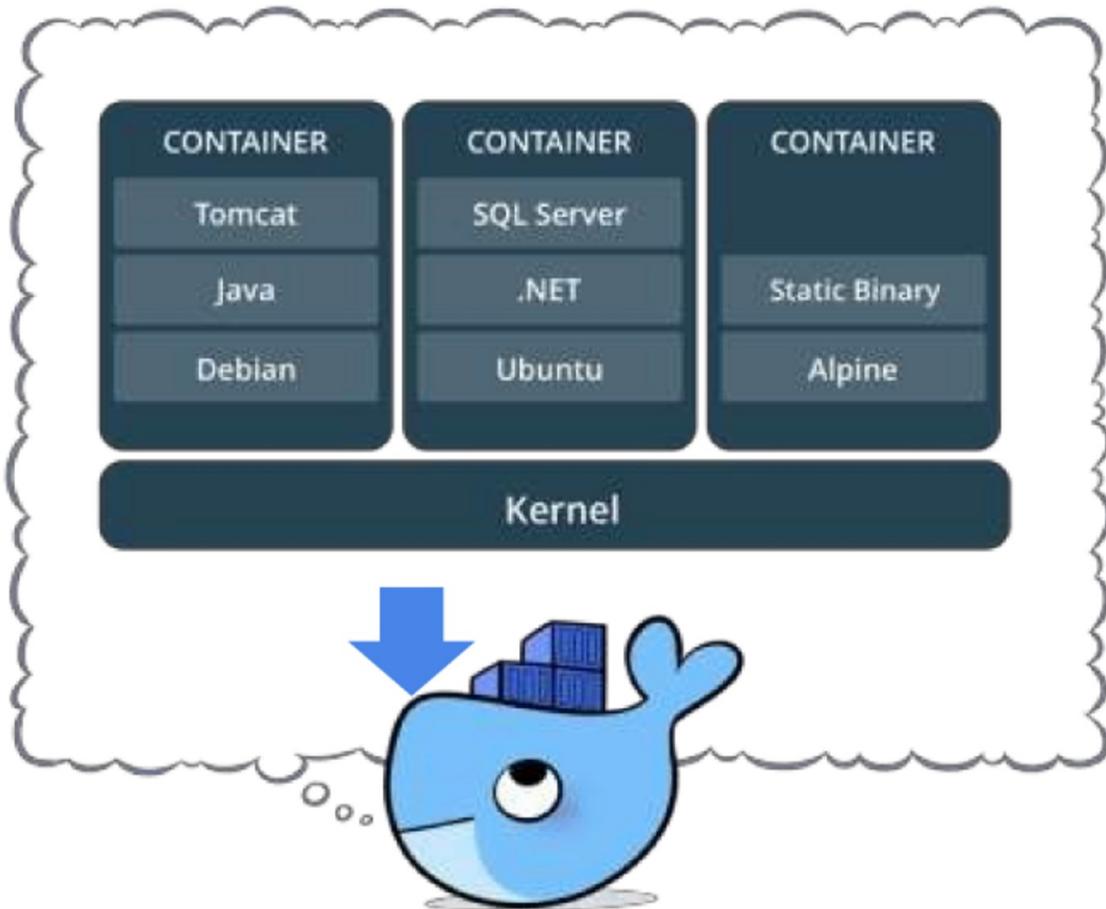


# Limitations of VMs

- Each VM stills requires
  - CPU allocation
  - Storage
  - RAM
  - An entire guest operating system
- The more VMs you run, the more resources you need
- Guest OS means wasted resources
- Application portability not guaranteed



# What is a container?

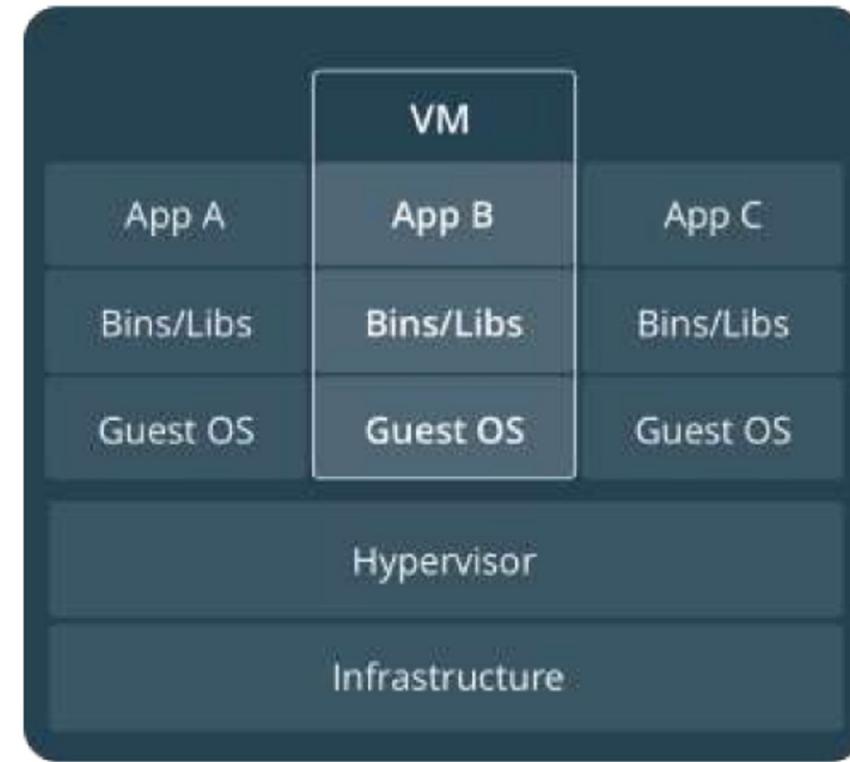


- Standardized packaging for software and dependencies
- Isolate apps from each other
- Share the same OS kernel
- Works with all major Linux and Windows Server

# Comparing Containers and VMs

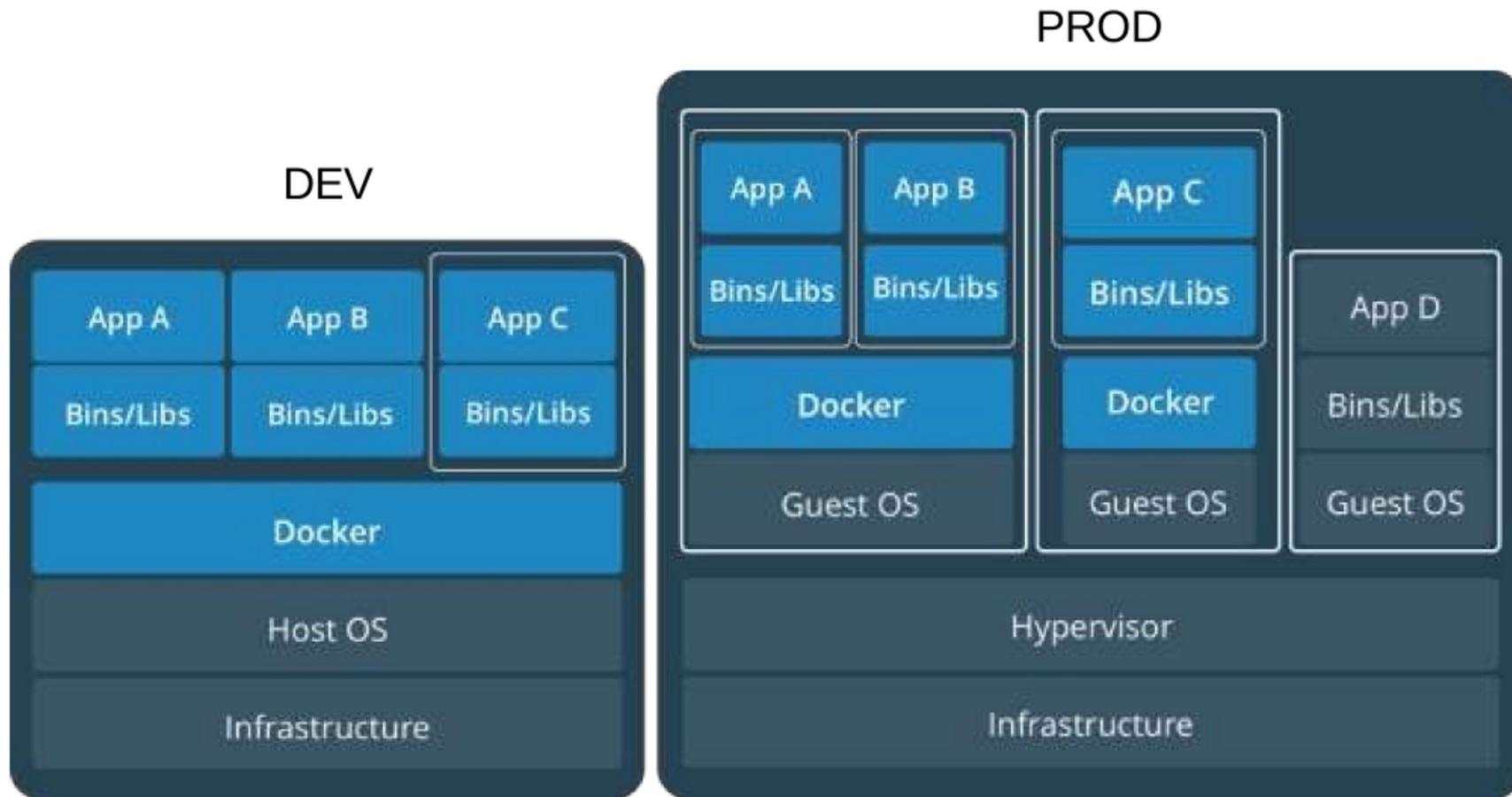


Containers are an app level construct



VMs are an infrastructure level construct to turn one machine into many servers

# Containers and VMs together



Containers and VMs together provide a tremendous amount of flexibility for IT to optimally deploy and manage apps.

# Key Benefits of Docker Containers

## Speed

- No OS to boot = applications online in seconds

## Portability

- Less dependencies between process layers = ability to move between infrastructure

## Efficiency

- Less OS overhead
- Improved VM density

# Docker Basics



## Image

The basis of a Docker container. The content at rest.



## Container

The image when it is ‘running.’ The standard unit for app service



## Engine

The software that executes commands for containers. Networking and volumes are part of Engine. Can be clustered together.



## Registry

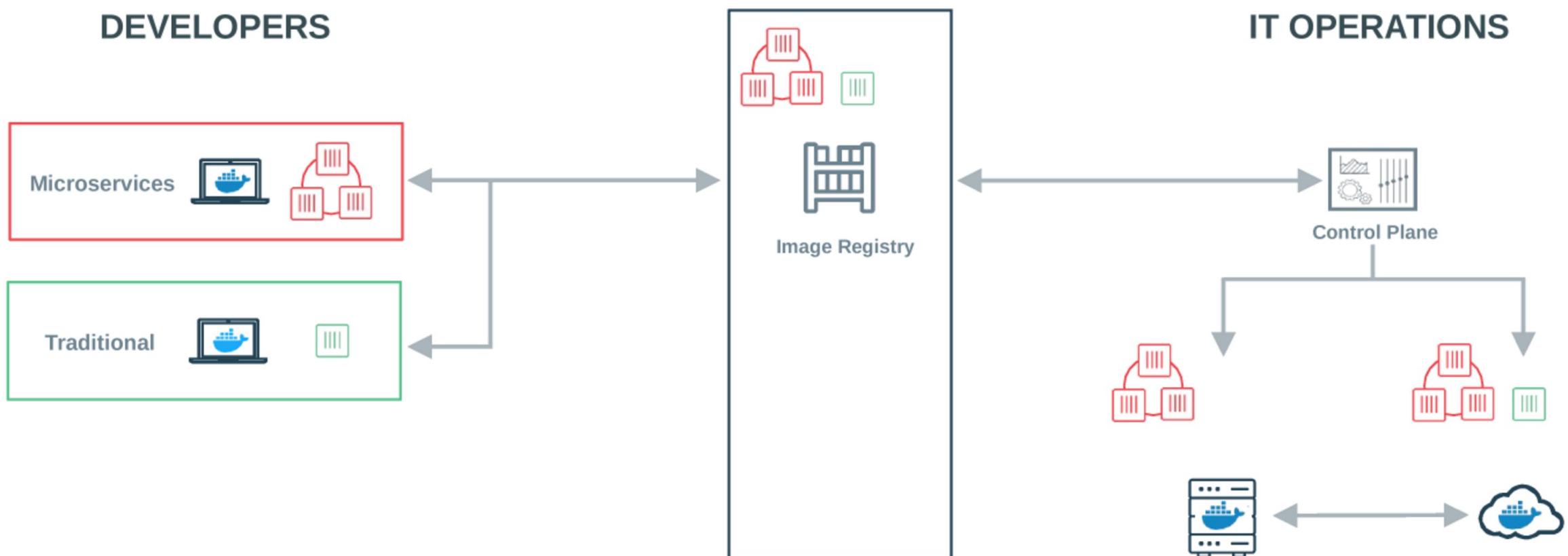
Stores, distributes and manages Docker images



## Control Plane

Management plane for container and cluster orchestration

# Building a Software Supply Chain



# Docker registry

A **Docker registry** is a storage and distribution system for named Docker images. The same image might have multiple different versions, identified by their tags.

A Docker registry is organized into Docker repositories , where a repository holds all the versions of a specific image.

The registry allows Docker users to pull images locally, as well as push new images to the registry (given adequate access permissions when applicable).

By default, the Docker engine interacts with DockerHub , Docker's public registry instance.

However, it is possible to run on-premise the open-source Docker registry/distribution, as well as a commercially supported version called Docker Trusted Registry .

# Run your first docker

# Docker run

One of the first and most important commands Docker users learn is the `docker run` command. This comes as no surprise since its primary function is to build and run containers.

There are many different ways to run a container. By adding attributes to the basic syntax, you can configure a container to run in detached mode, set a container name, mount a volume, and perform many more tasks.

*`docker run [OPTIONS] IMAGE [COMMAND] [ARG...]`*

# Docker run

```
> docker run hello-world
```

```
Unable to find image 'hello-world:latest' locally
```

```
latest: Pulling from library/hello-world
```

```
[...]
```

```
Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker!
```

```
This message shows that your installation appears to be working correctly.
```

```
[...]
```

# Semplici comandi docker

- Docker images
  - Lista le immagini che ho in locale
- Per eseguire un'immagine tomcat
  - `docker run -it --rm -p 8888:8080 tomcat:10.0`
    - `-it` => terminale interattivo
    - `--rm` => rimuovi il container al termine
    - `-p` => mappa una porta TCP dell'host su una porta TCP del container. 8888 sta sulla macchina host, 8080 è la porta di default di tomcat
  - Per «entrare» nell'ambiente di esecuzione, se è previsto,
    - `docker exec -it cec3a80a94d0 /bin/bash -l`
    - Dove l'id alfanumerico è quello che risulta da `docker ps`

# Come pubblicare un'app TOMCAT

- `docker run -it --rm -p 8888:8080 -v «path della cartella webapp creata da Eclipse»:/usr/local/tomcat/webapps/<nome pubblico dell'applicazione, definito da voi> tomcat:10.0`
- NB: se dovete pubblicare su un'altra porta, allora al posto di 8888 mettere la porta che più vi aggrada

# Docker-compose

- Al posto di mettere le opzioni e i valori sulla riga di comando di docker, è possibile scrivere un file (docker-compose.yaml) che contiene tutti i parametri che intendete utilizzare e docker, tramite il comando docker-compose up, esegue tutte le immagini che avete indicato nel file

# Build your own docker

# Docker file basics

A [Dockerfile](#) is a simple text file that contains a list of commands that the Docker client calls while creating an image.

It's a simple way to automate the image creation process.

The [commands](#) you write in a Dockerfile are *almost* identical to their equivalent Linux commands: this means you don't really have to learn new syntax to create your own dockerfiles.

# Docker file directives

## FROM

The from directive is used to set base image for the subsequent instructions. A Dockerfile must have FROM directive with valid image name as the first instruction.

*FROM ubuntu:20.04*

## RUN

Using RUN directing ,you can run any command to image during build time. For example you can install required packages during the build of image.

*RUN apt-get update*

*RUN apt-get install -y apache2 automake build-essential curl*

# Docker file directives

## COPY

The COPY directive used for copying files and directories from host system to the image during build.

For example the first commands will copy all the files from hosts html/ directory /var/www/html image directory.

Second command will copy all files with extension .conf to /etc/apache2/sites-available/ directory.

```
COPY html/* /var/www/html/  
COPY *.conf /etc/apache2/sites-available/
```

## WORKDIR

The WORKDIR directive used to sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD commands during build.

```
WORKDIR /opt
```

# Docker file directives

## CMD

The CMD directive is used to run the service or software contained by your image, along with any arguments during the launching the container. CMD uses following basic syntax

```
CMD ["executable", "param1", "param2"]  
CMD ["executable", "param1", "param2"]
```

For example, to start Apache service during launch of container, Use the following command.

```
CMD ["apachectl", "-D", "FOREGROUND"]
```

## EXPOSE

The EXPOSE directive indicates the ports on which a container will listen for the connections. After that you can bind host system port with container and use them.

```
EXPOSE 80
```

```
EXPOSE 443
```

# Docker file directives

## ENV

The ENV directive is used to set environment variable for specific service of container.

```
ENV PATH=$PATH:/usr/local/pgsql/bin/
```

```
ENV PG_MAJOR=9.6.0
```

## VOLUME

The VOLUME directive creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

```
VOLUME ["/data"]
```

# Sample docker file

Given this Dockerfile:

```
FROM alpine  
CMD ["echo", "Hello Tor Vergata!"]
```

Build and run it:

```
docker build -t hello .  
docker run --rm hello
```

This will output:

Hello Tor Vergata!

# Sample docker file

```
FROM nginx:latest  
RUN touch /testfile  
COPY ./index.html /usr/share/nginx/html/index.html
```

# Docker build / push

Use Docker build to build your image locally

*docker build -t <registry>/<image name>:<tag> .*

And Docker push to publish your image on registry

*docker push <registry>/<image name>:<tag>*

# Data persistence

# Data persistence

Docker containers provide you with a writable layer on top to make changes to your running container.

**But these changes are bound to the container's lifecycle:** If the container is deleted (not stopped), you lose your changes.

Let's take a hypothetical scenario where you are running a database in a container without any data persistence configured.

You create some tables and add some rows to them: but, if some reason, you need to delete this container, as soon as the container is deleted all your tables and their corresponding data get lost.

Docker provides us with a couple of solutions to persist your data even if the container is deleted.

The two possible ways to persist your data are:

- **Bind Mounts**
- **Volumes**

# Bind mounts

**Bind mounts** have been around since the early days of Docker.

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

The file or directory is referenced by its absolute path on the host machine.

By contrast, when you use a volume, a new directory is created within Docker's storage directory on the host machine, and Docker manages that directory's contents.

The file or directory does not need to exist on the Docker host already.

It is created on demand if it does not yet exist.

Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available.

If you are developing new Docker applications, consider using named volumes instead.

# Docker volumes

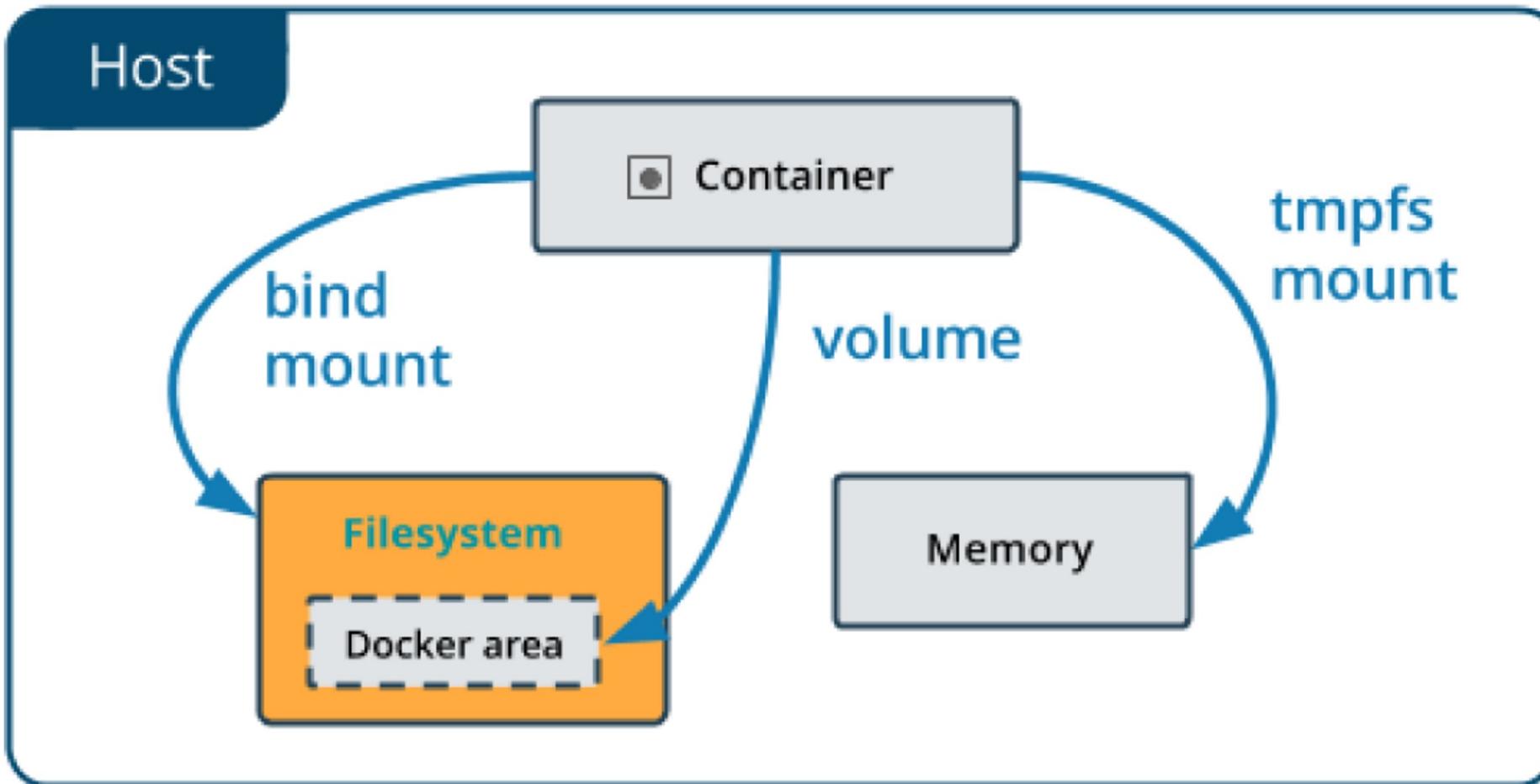
Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.

While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker.

Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container.

# Bind mounts vs volumes



# Docker volumes

> *docker volume create my-volume*

> *docker run -d --name test -v my-volume:/app nginx:latest*

# Docker networking

# Docker network basics

**Docker Networking** is used to connect docker container with each other and with the outside world.

Docker uses **CNM (Container Network Model)** for networking.

This model standardizes the steps required to provide networking for containers using multiple network drivers.

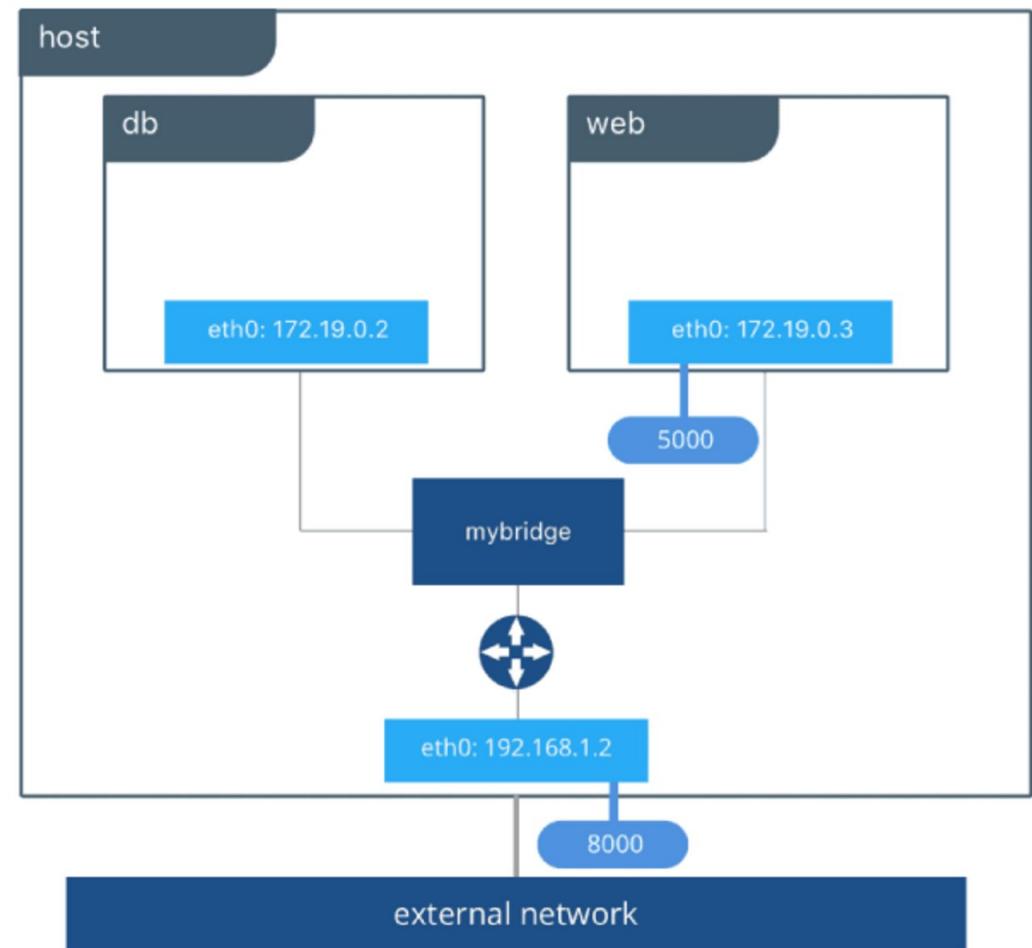
# Bridge networking

Bridge network is a default network created automatically when you deploy a container.

Bridge network uses a software bridge that allows containers connected to the same bridge network to communicate.

Bridge networks are used on containers that are running on the same **Docker daemon host**.

The bridge network creates a private internal isolated network to the host so containers on this network can communicate.



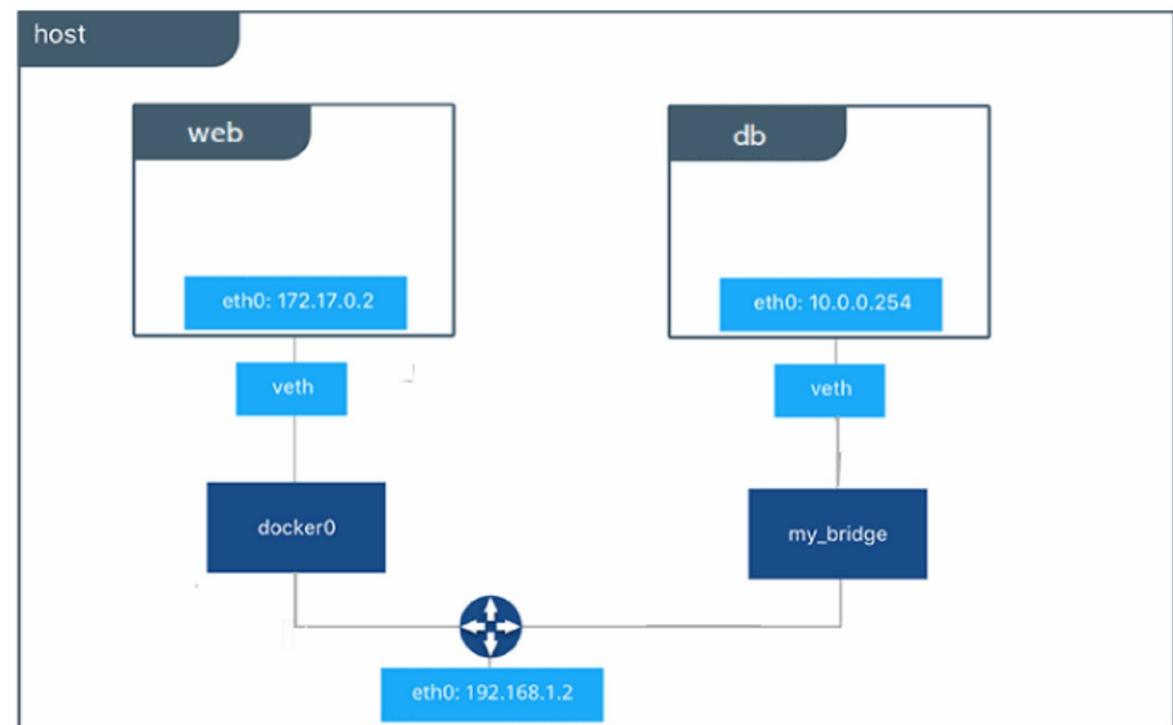
# Host networking

This takes out any network isolation between the docker host and the docker containers.

Host mode networking can be **useful to optimize performance**.

It does not require network address translation (NAT).

The host networking driver only works on Linux hosts, and is not supported on Docker Desktop for Mac, Docker Desktop for Windows, or Docker EE for Windows Server.



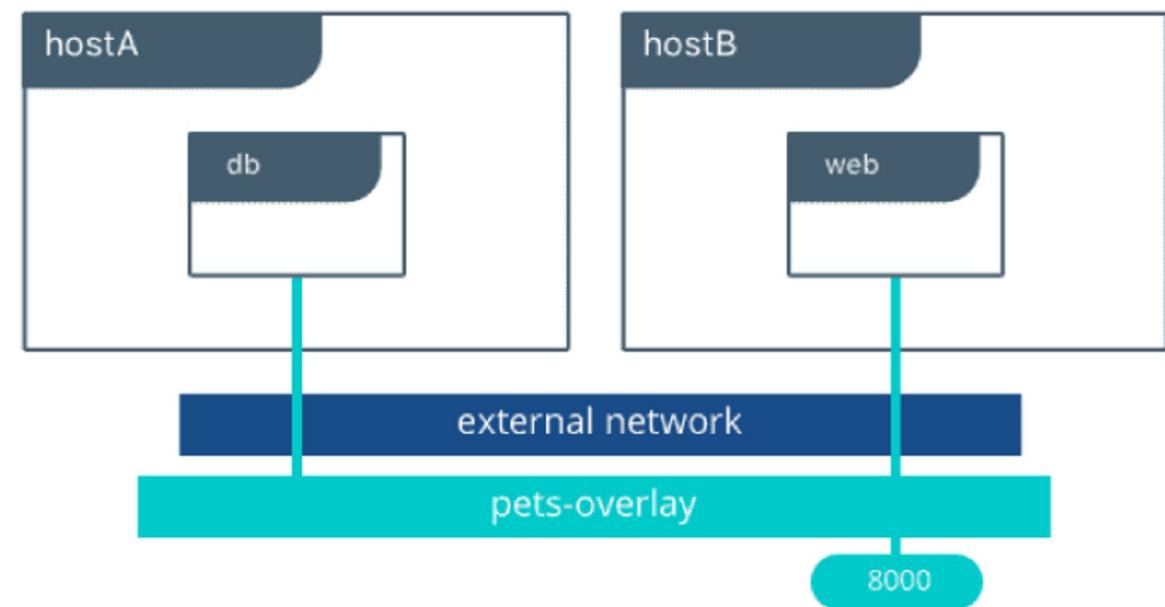
# Overlay networking

Overlay networking is used if container on node A wants to talk to node B then to make communication between them we use Overlay networking.

Overlay networking uses **VXLAN** to create an **Overlay network**.

This has the advantage of providing maximum portability across various cloud and on-premises networks.

By default, the Overlay network is encrypted with the **AES algorithm**.



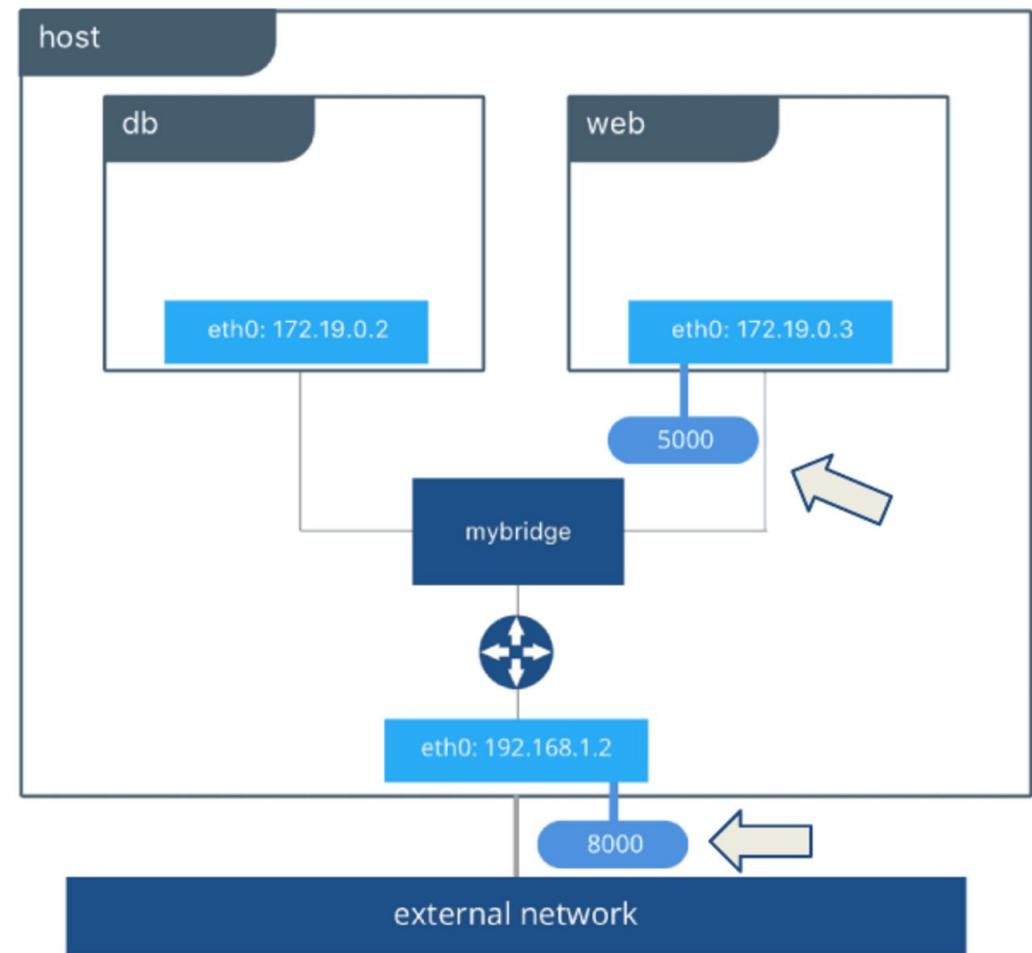
# Exposing ports

By default, when we create any containers it doesn't publish or expose the application ports running on the containers.

We can access these applications only within the docker host not through network systems.

You can explicitly bind a port or group of ports from container to host using the `-p` flag.

```
docker run [...] -p 8000:5000 docker.io/httpd
```



# Docker cheatsheet



# Cheatsheet for Docker CLI

## Run a new Container

Start a new Container from an Image  
`docker run IMAGE`  
`docker run nginx`

...and assign it a name  
`docker run --name CONTAINER IMAGE`  
`docker run --name web nginx`

...and map a port  
`docker run -p HOSTPORT:CONTAINERPORT IMAGE`  
`docker run -p 8080:80 nginx`

...and map all ports  
`docker run -P IMAGE`  
`docker run -P nginx`

...and start container in background  
`docker run -d IMAGE`  
`docker run -d nginx`

...and assign it a hostname  
`docker run --hostname HOSTNAME IMAGE`  
`docker run --hostname srv nginx`

...and add a dns entry  
`docker run --add-host HOSTNAME:IP IMAGE`

...and map a local directory into the container  
`docker run -v HOSTDIR:TARGETDIR IMAGE`  
`docker run -v ~/usr/share/nginx/html nginx`

...but change the entrypoint  
`docker run -it --entrypoint EXECUTABLE IMAGE`  
`docker run -it --entrypoint bash nginx`

## Manage Containers

Show a list of running containers  
`docker ps`

Show a list of all containers  
`docker ps -a`

Delete a container  
`docker rm CONTAINER`  
`docker rm web`

Delete a running container  
`docker rm -f CONTAINER`  
`docker rm -f web`

Delete stopped containers  
`docker container prune`

Stop a running container  
`docker stop CONTAINER`  
`docker stop web`

Start a stopped container  
`docker start CONTAINER`  
`docker start web`

Copy a file from a container to the host  
`docker cp CONTAINER:SOURCE TARGET`  
`docker cp web:/index.html index.html`

Copy a file from the host to a container  
`docker cp TARGET CONTAINER:SOURCE`  
`docker cp index.html web:/index.html`

Start a shell inside a running container  
`docker exec -it CONTAINER EXECUTABLE`  
`docker exec -it web bash`

Rename a container  
`docker rename OLD_NAME NEW_NAME`  
`docker rename 096 web`

Create an image out of container  
`docker commit CONTAINER`  
`docker commit web`

## Manage Images

Download an image  
`docker pull IMAGE[:TAG]`  
`docker pull nginx`

Upload an image to a repository  
`docker push IMAGE`  
`docker push myimage:1.0`

Delete an image  
`docker rmi IMAGE`

Show a list of all Images  
`docker images`

Delete dangling images  
`docker image prune`

Delete all unused images  
`docker image prune -a`

Build an image from a Dockerfile  
`docker build DIRECTORY`  
`docker build .`

Tag an image  
`docker tag IMAGE NEWIMAGE`  
`docker tag ubuntu ubuntu:18.04`

Build and tag an image from a Dockerfile  
`docker build -t IMAGE DIRECTORY`  
`docker build -t myimage .`

Save an image to .tar file  
`docker save IMAGE > FILE`  
`docker save nginx > nginx.tar`

Load an image from a .tar file  
`docker load -i TARFILE`  
`docker load -i nginx.tar`

## Info & Stats

Show the logs of a container  
`docker logs CONTAINER`  
`docker logs web`

Show stats of running containers  
`docker stats`

Show processes of container  
`docker top CONTAINER`  
`docker top web`

Show installed docker version  
`docker version`

Get detailed info about an object  
`docker inspect NAME`  
`docker inspect nginx`

Show all modified files in container  
`docker diff CONTAINER`  
`docker diff web`

Show mapped ports of a container  
`docker port CONTAINER`  
`docker port web`

# Docker echosystems

# Docker compose

Docker Compose is a tool that was developed to help define and share multi-container applications.

With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

Each of the containers here run in isolation but can interact with each other when required.

Docker Compose files are very easy to write in a scripting language called YAML, which is an XML-based language that stands for Yet Another Markup Language.

```
version: "3.7"
```

```
services:
```

```
  app:
```

```
    image: node:12-alpine
```

```
    command: sh -c "yarn install && yarn run dev"
```

```
    ports:
```

```
      - 3000:3000
```

```
    working_dir: /app
```

```
    volumes:
```

```
      - ./:/app
```

```
    environment:
```

```
      MYSQL_HOST: mysql
```

```
      MYSQL_USER: root
```

```
      MYSQL_PASSWORD: secret
```

```
      MYSQL_DB: todos
```

```
mysql:
```

```
  image: mysql:5.7
```

```
  volumes:
```

```
    - mysql-data:/var/lib/mysql
```

```
  environment:
```

```
    MYSQL_ROOT_PASSWORD: secret
```

```
    MYSQL_DATABASE: todos
```

```
volumes:
```

```
  mysql-data:
```

# Docker orchestration

# Docker swarm

Docker swarm is a container orchestration tool, meaning that it allows the user to manage multiple containers deployed across multiple host machines.

A Docker Swarm is a group of either physical or virtual machines that are running the Docker application and that have been configured to join together in a cluster.

Once a group of machines have been clustered together, you can still run the Docker commands that you're used to, but they will now be carried out by the machines in your cluster.

The activities of the cluster are controlled by a swarm manager, and machines that have joined the cluster are referred to as nodes.

# Kubernetes

Kubernetes is an open source system to deploy, scale, and manage containerized applications.

It automates operational tasks of container management and includes built-in commands for deploying applications, rolling out changes to your applications, scaling your applications up and down to fit changing needs, monitoring your applications, and more.

Application developers, IT system administrators and DevOps engineers use Kubernetes to automatically deploy, scale, maintain, schedule and operate multiple application containers across clusters of nodes.

Containers run on top of a common shared operating system (OS) on host machines but are isolated from each other unless a user chooses to connect them.

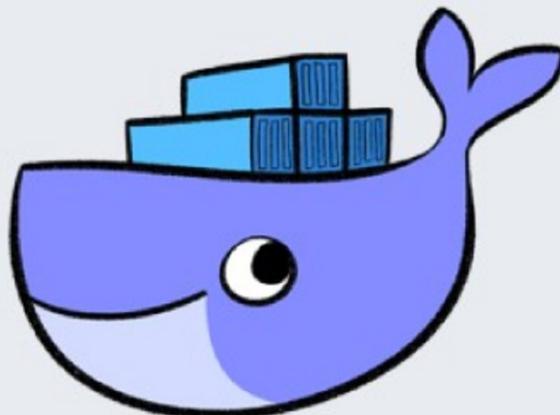
## Docker Swarm

- 1 No Auto Scaling
- 2 Good community
- 3 Easy to start a cluster
- 4 Limited to the Docker API's capabilities
- 5 Does not have as much experience with production deployments at scale

## Kubernetes

- 1 Auto Scaling
- 2 Great active community
- 3 Difficult to start a cluster
- 4 Can overcome constraints of Docker and Docker API
- 5 Deployed at scale more often among organizations

# Docker playground



## Play with Docker

A simple, interactive and fun playground to learn Docker

<https://www.docker.com/play-with-docker/>