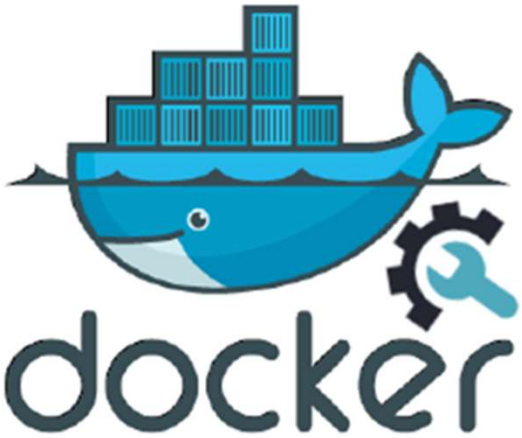
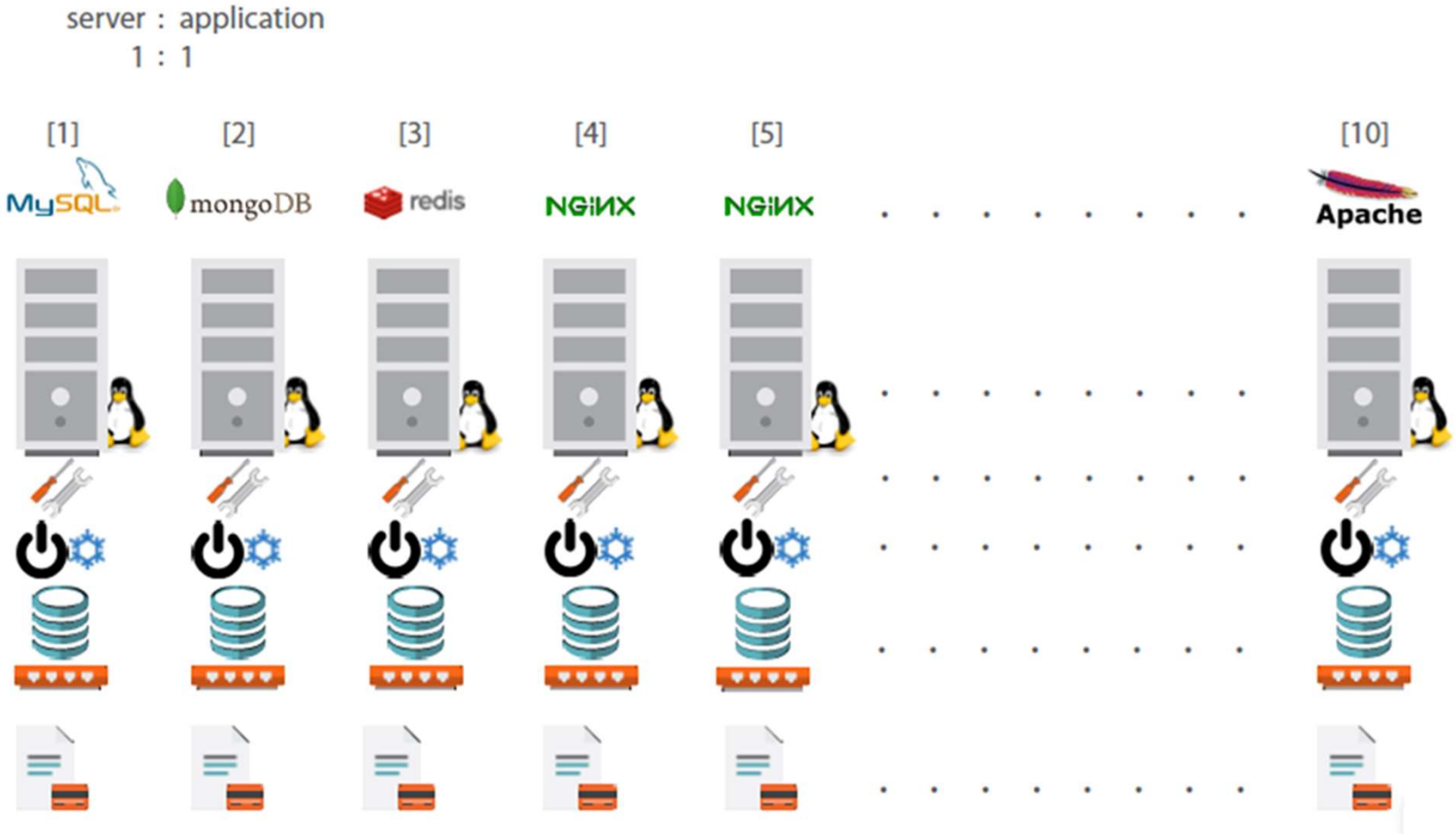
Docker – Day 1

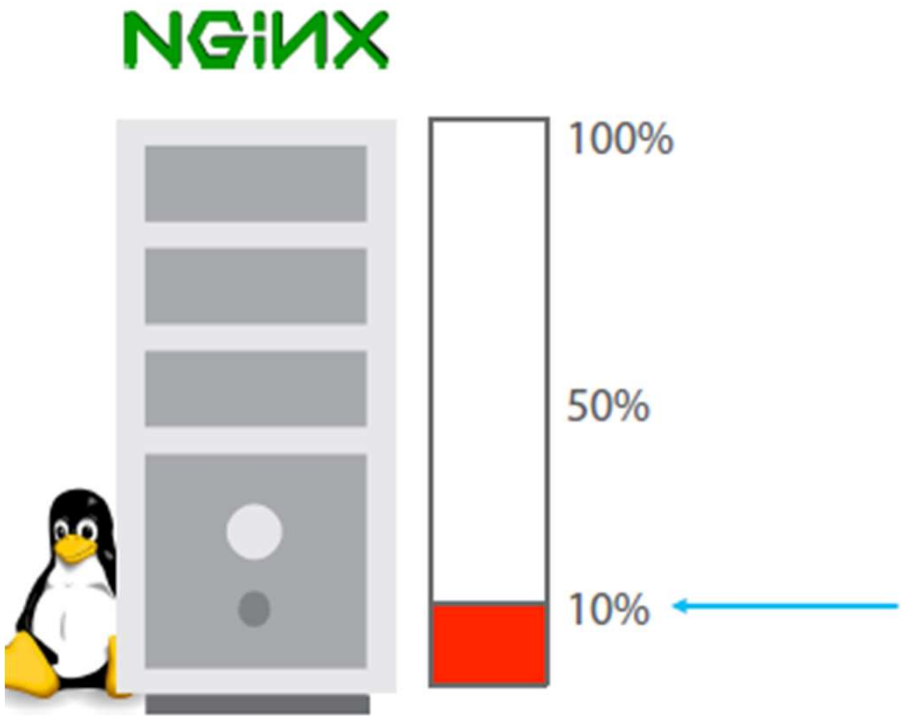
Docker



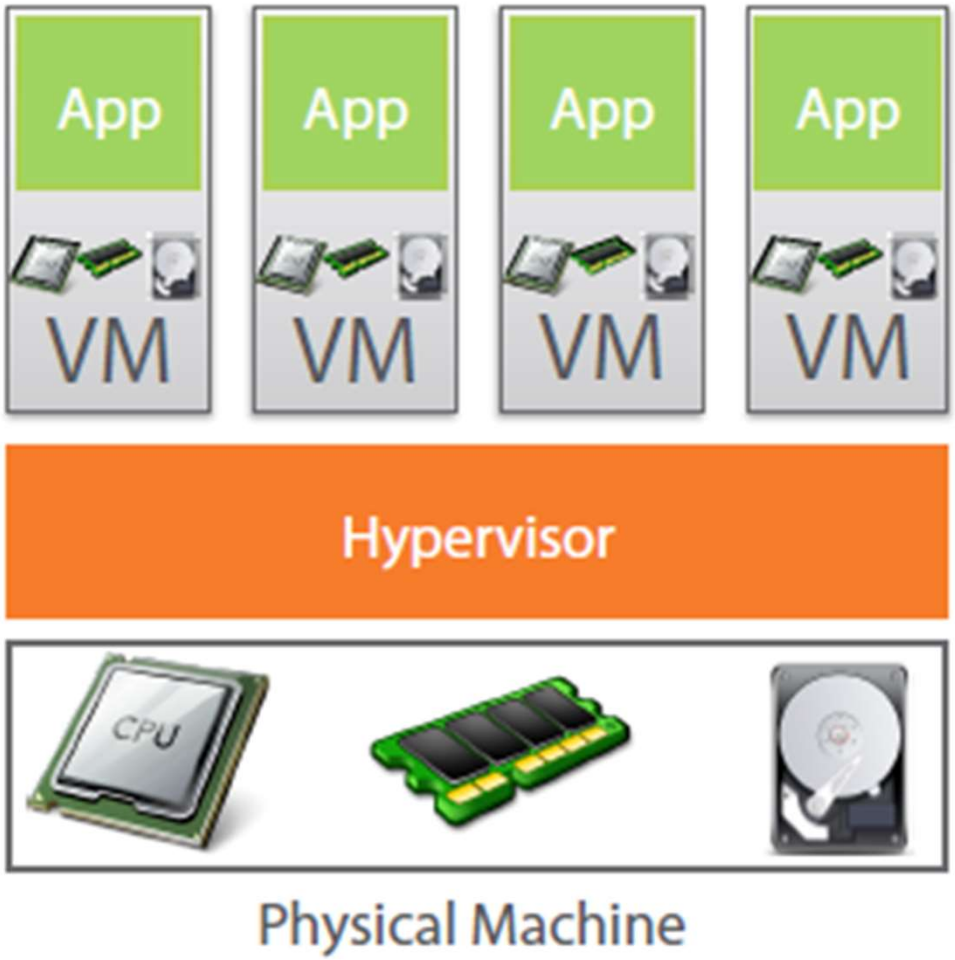
Traditional Deployment Architecture



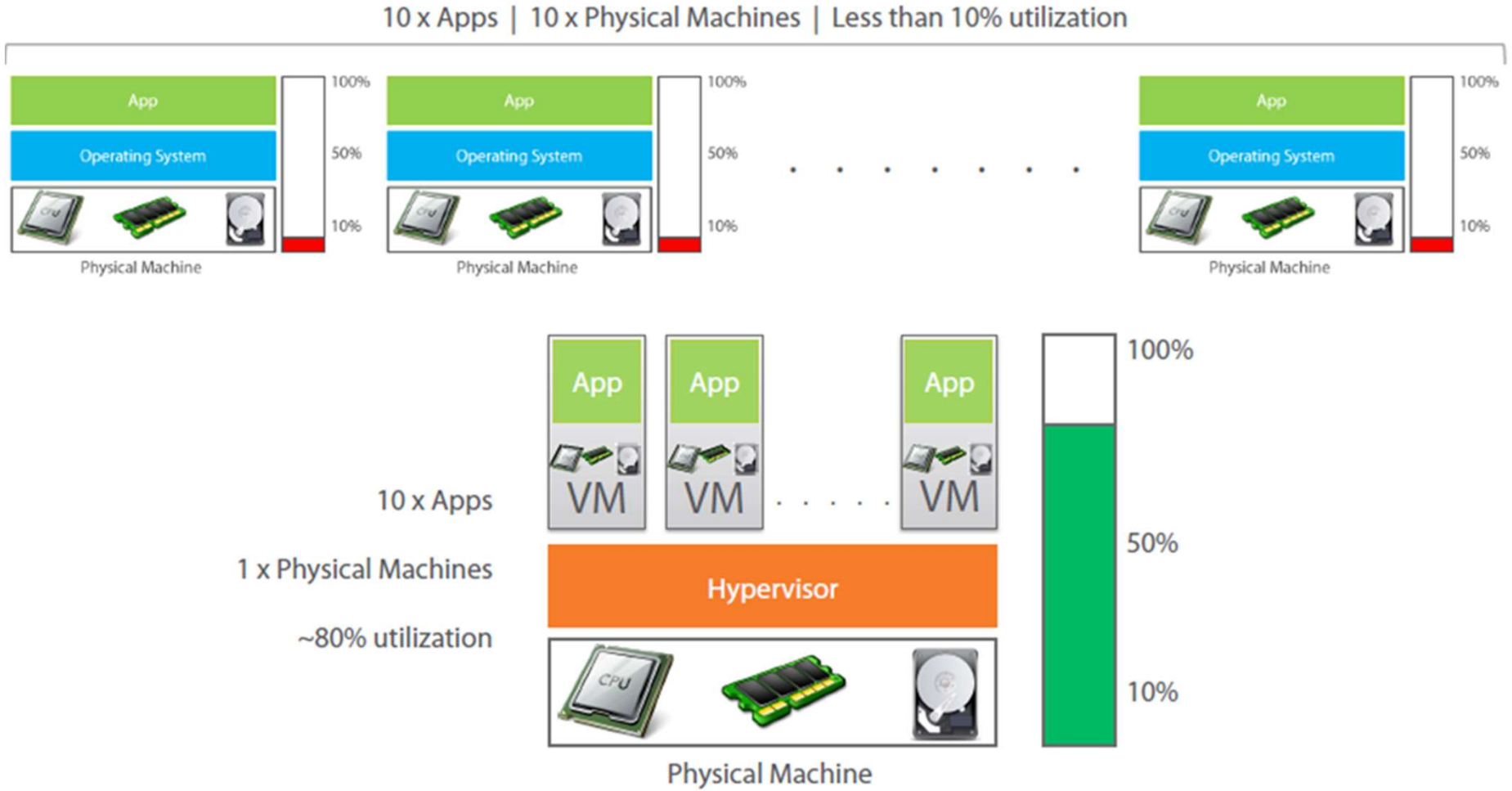
Less Utilization in Traditional Architecture



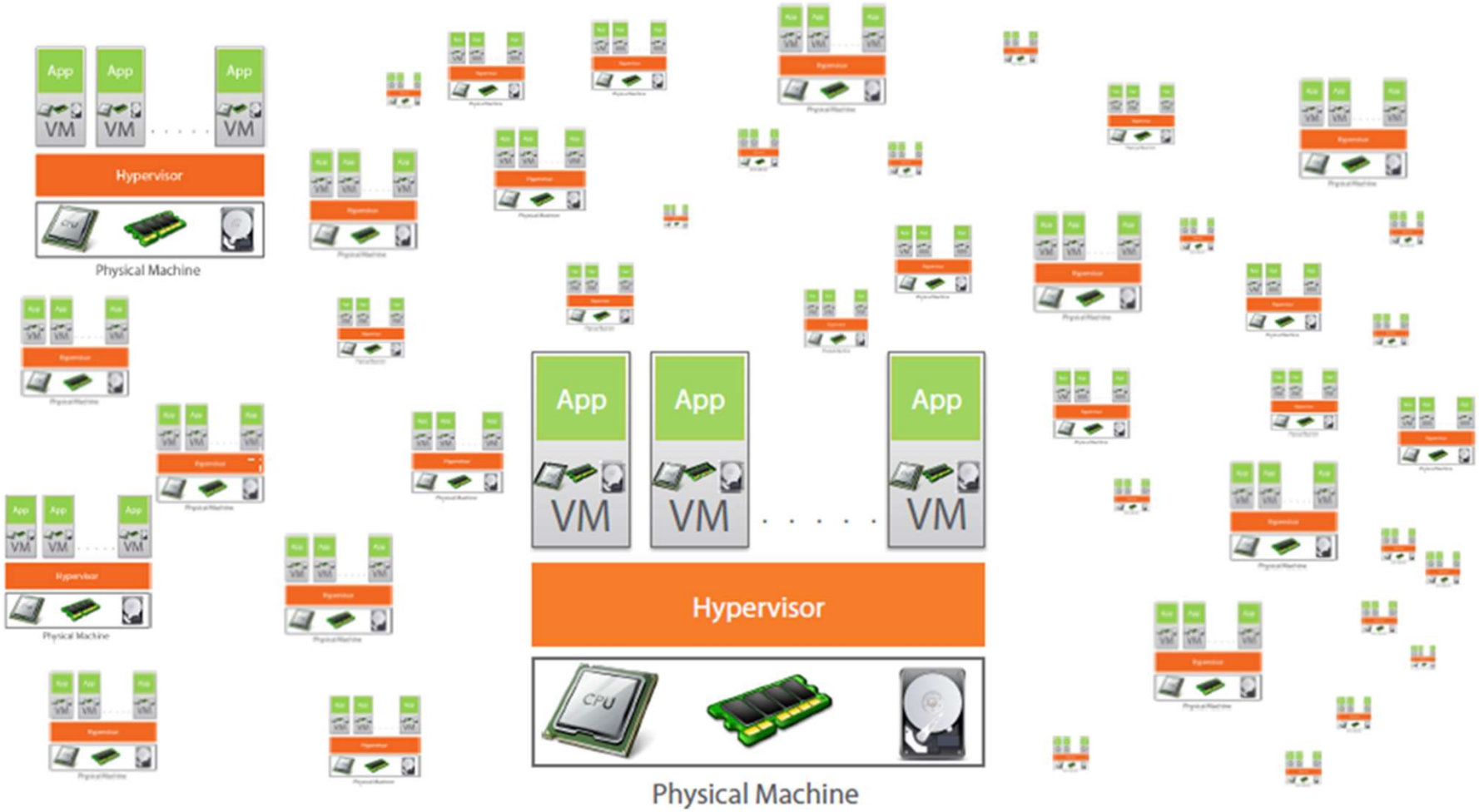
Virtual Machine to the Rescue



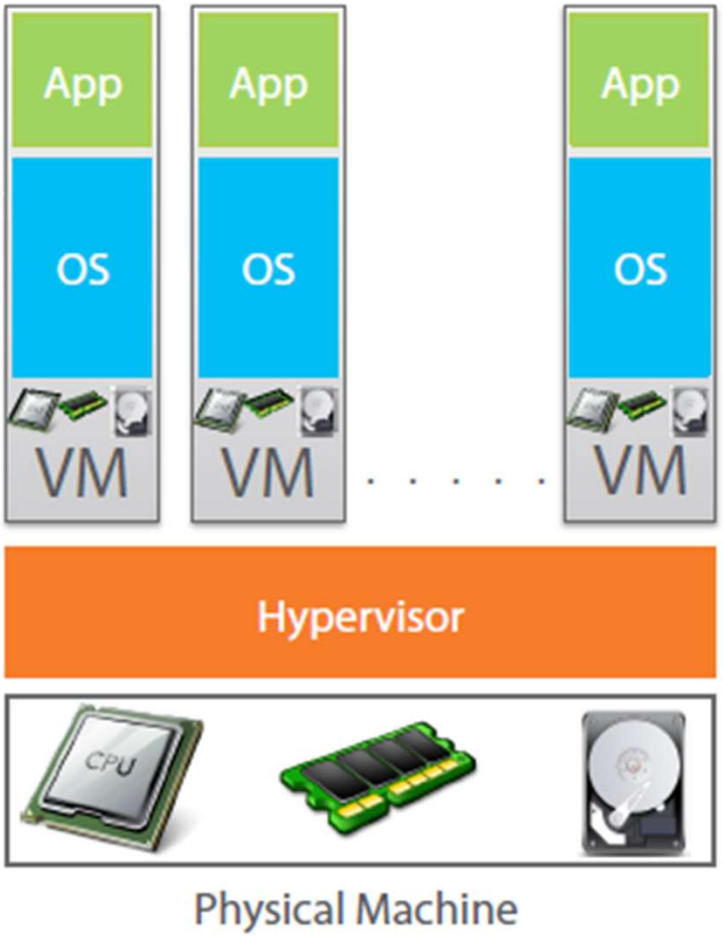
Virtual Machine provides better utilization



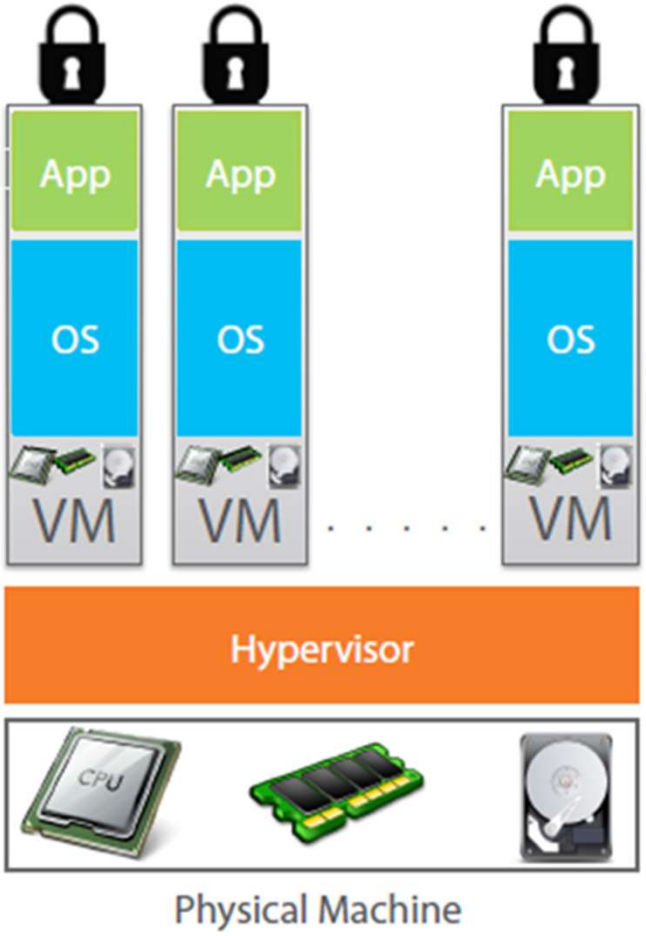
But Virtual Machine increases Licensing Cost



Each VM needs a separate OS

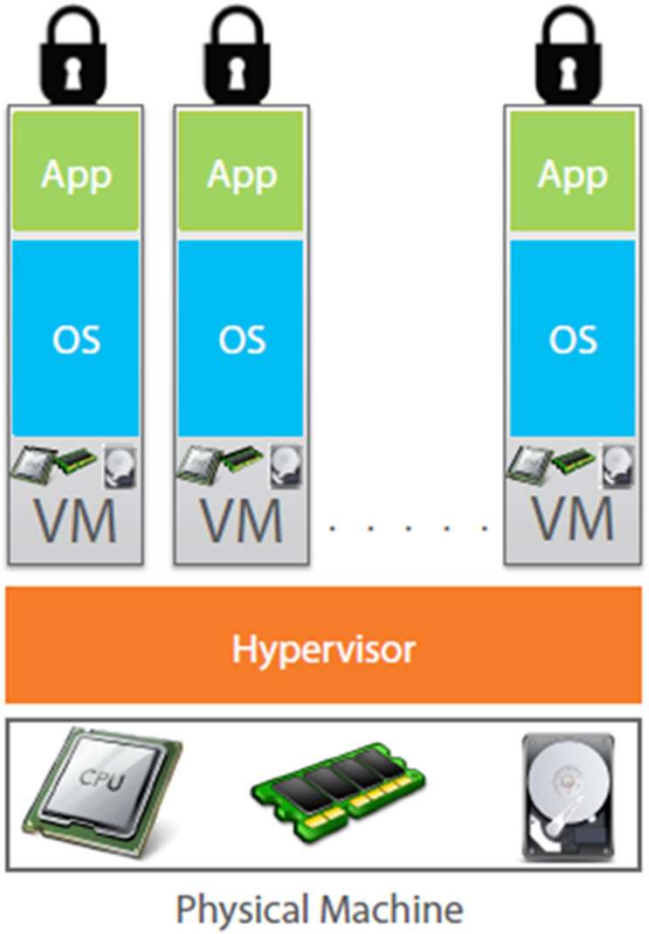


More OSes doesn’t increase Business Value



> OS != Business Value

OS takes most of the Resources

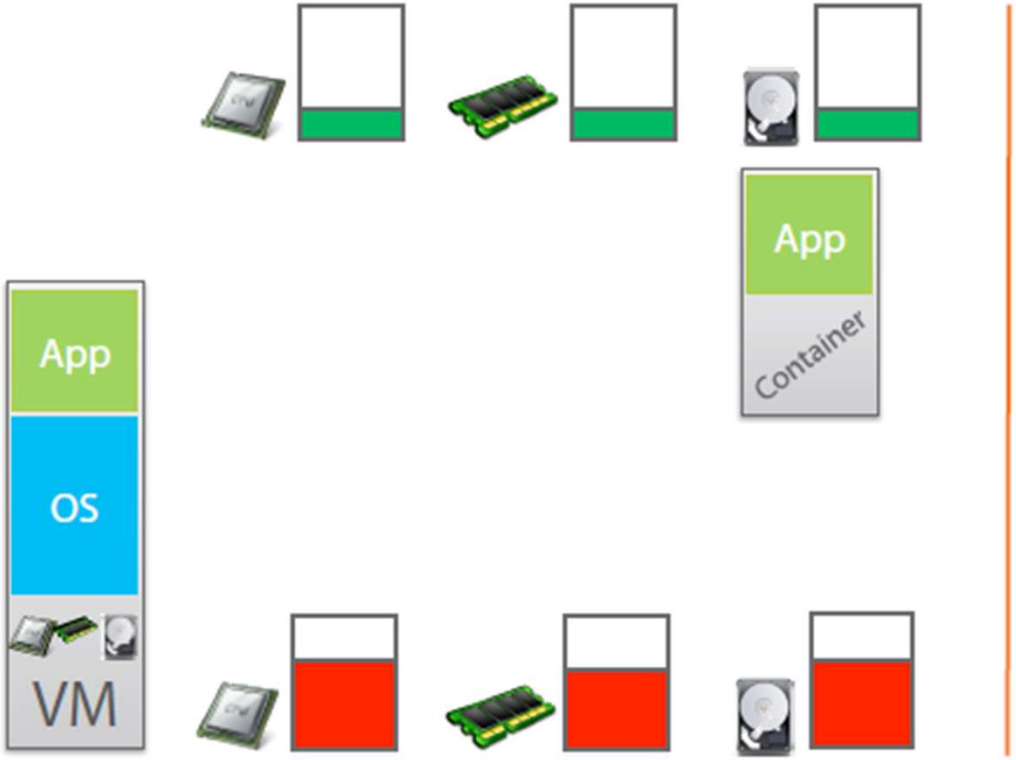


Why use separate OS for each App?

Containerization

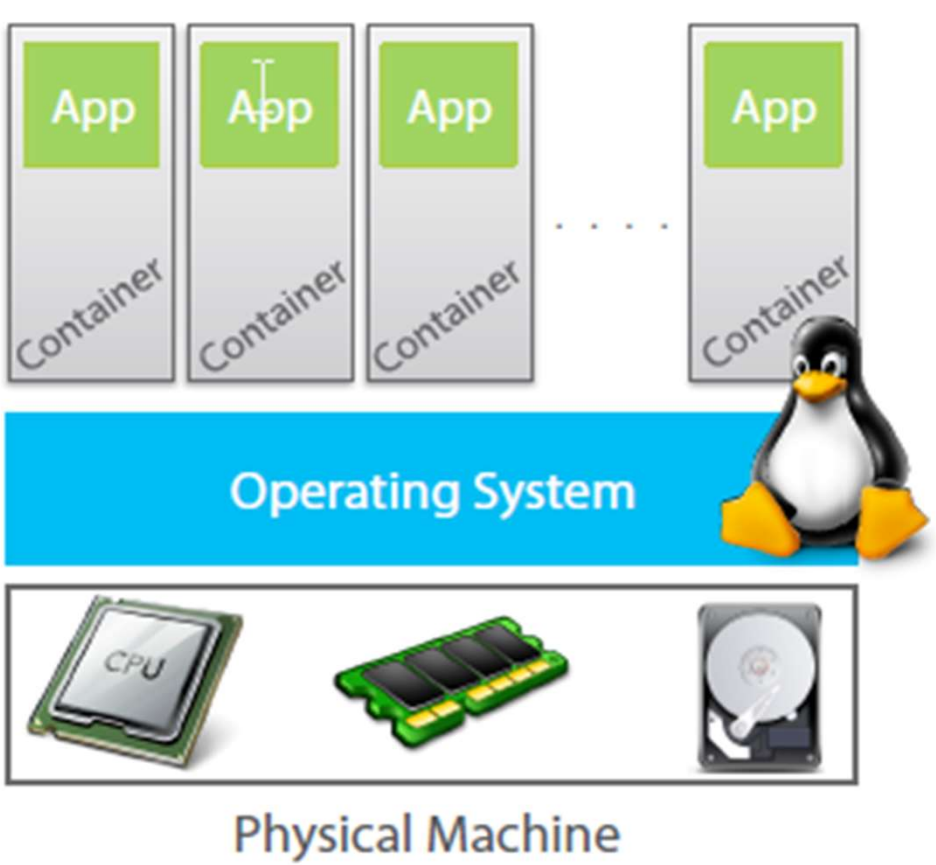
- Encapsulation of an application and its required environment.
- The process of packaging an application along with its required libraries, frameworks, and configuration files together so that it can be run in various computing environments efficiently.

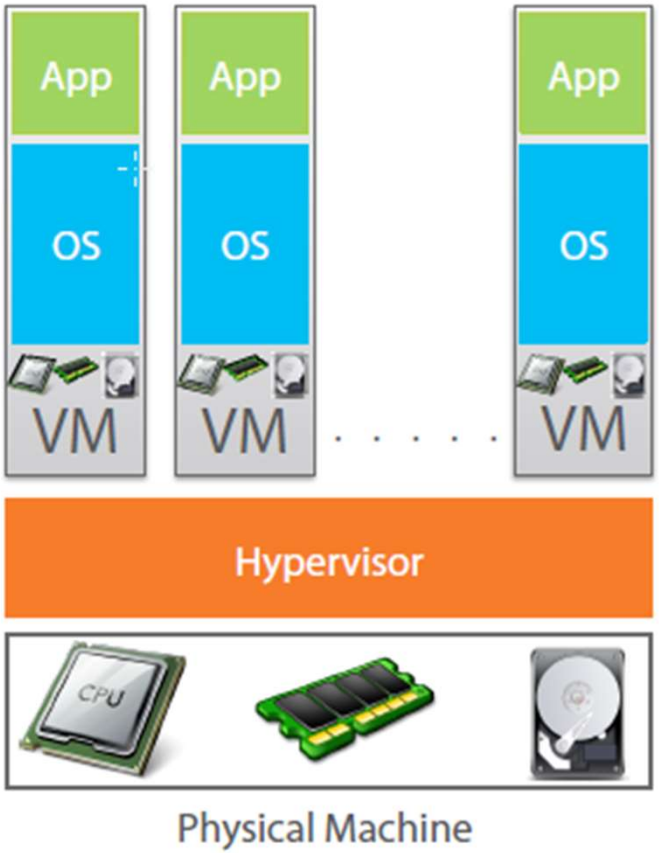
Containers to the Rescue



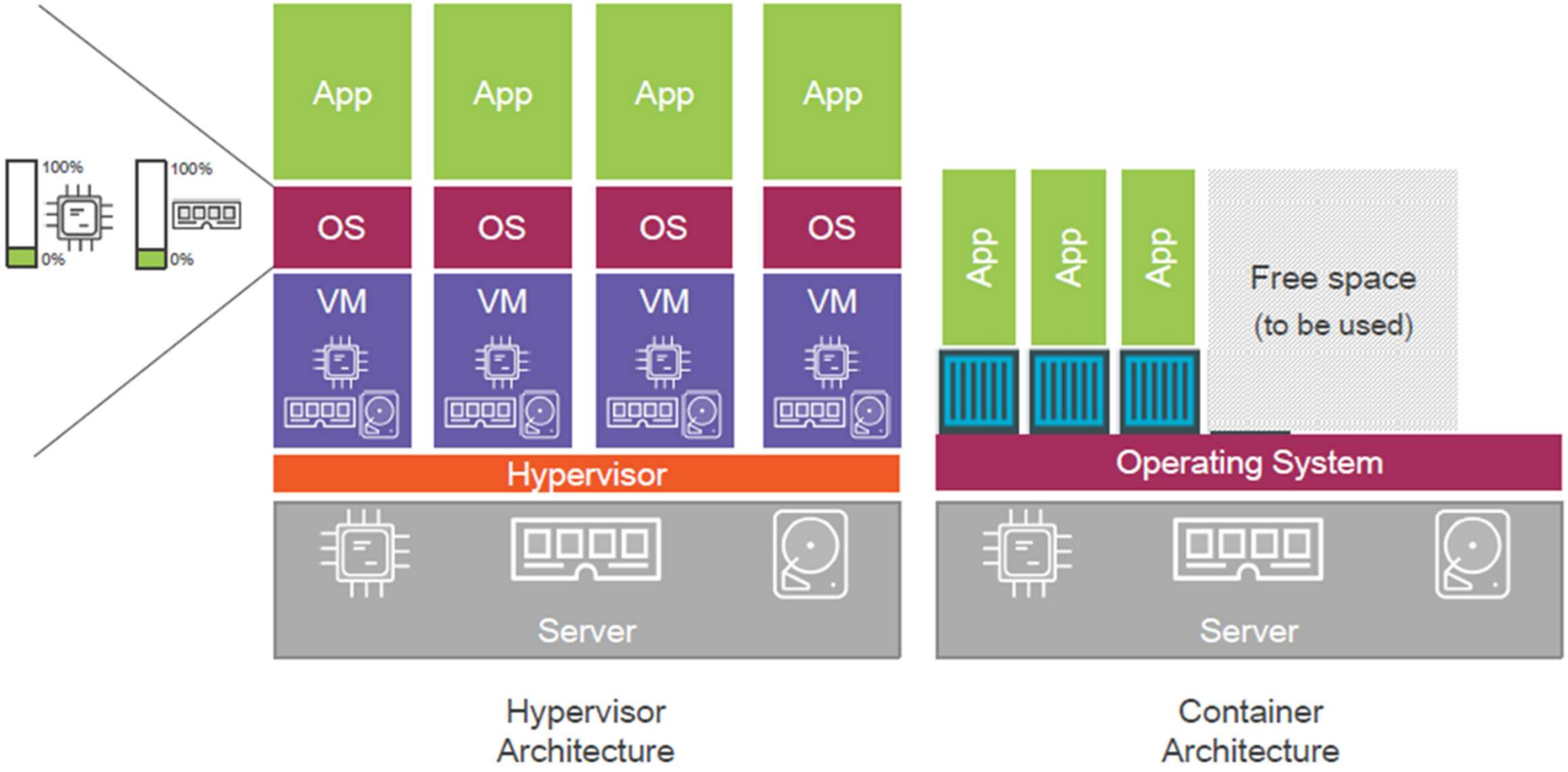
Containers are more lightweight than Virtual Machines

Containers vs VM

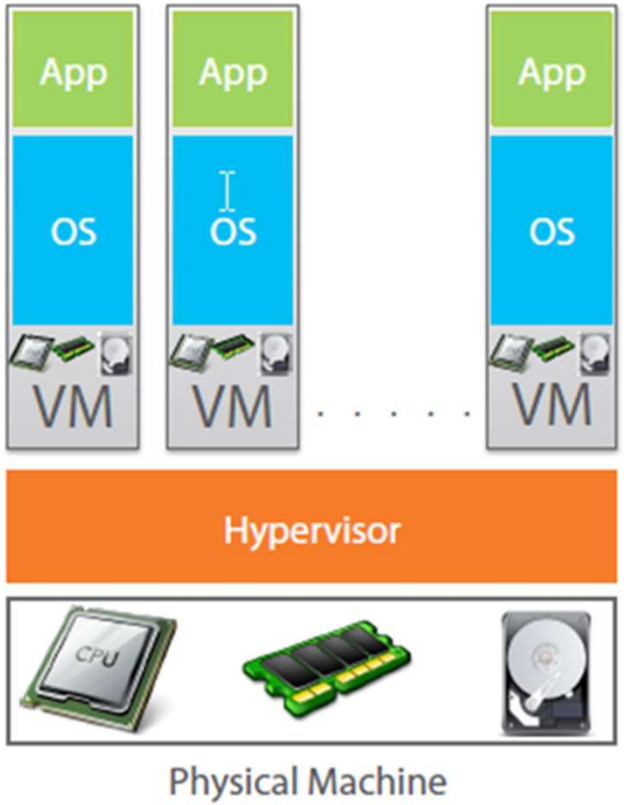


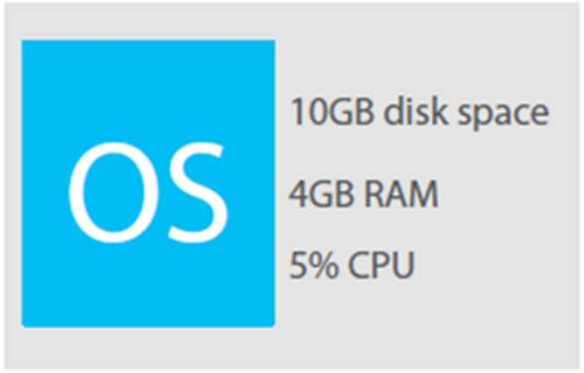


Containers vs VM



OS takes more resources and Licensing cost





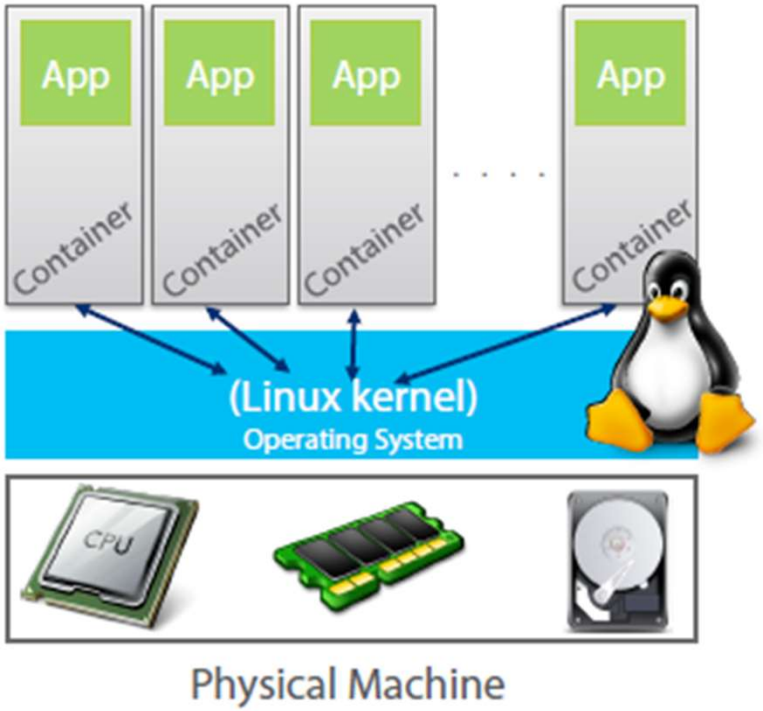
x 10 =

100GB disk space

40GB RAM

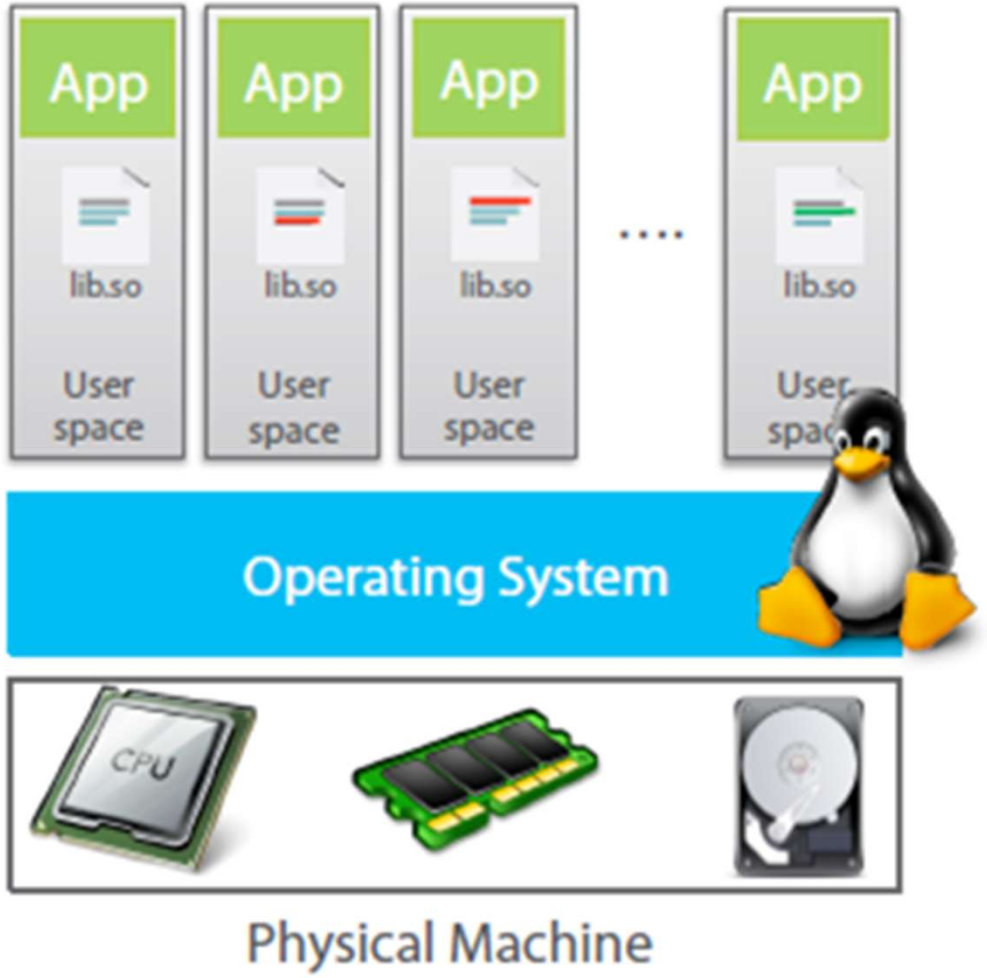
50% CPU

Containers takes less resources

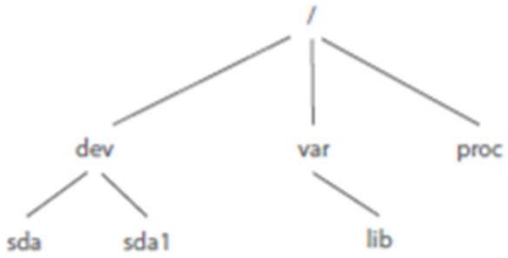


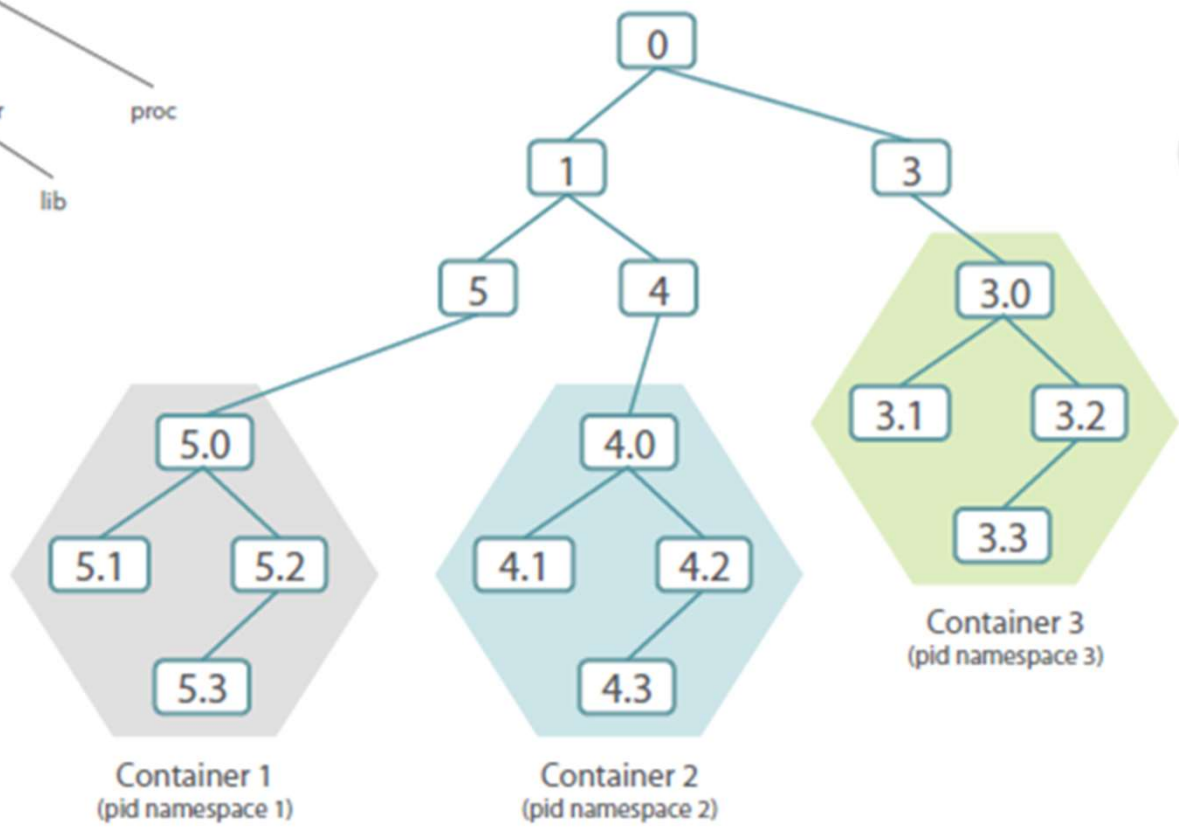
Containers consume less CPU, RAM and disk resource than Virtual Machines

How containers work?



How containers work?





What is Docker?

- Docker is an open-source project
  - that automates the deployment of applications inside software containers,
  - by providing an additional layer of abstraction and
  - automation of operating system–level virtualization on Linux.

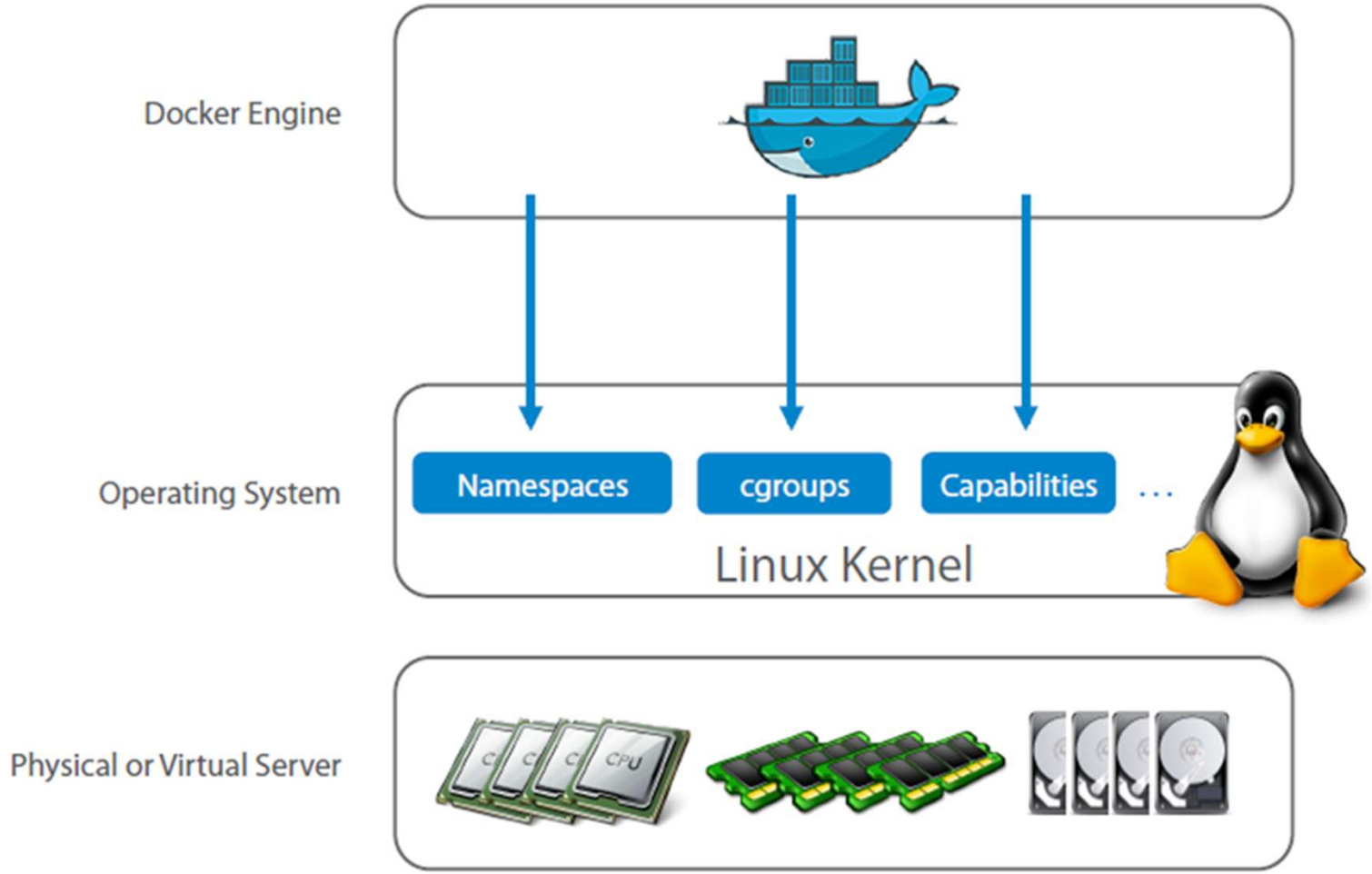
Practical

Practical Guide

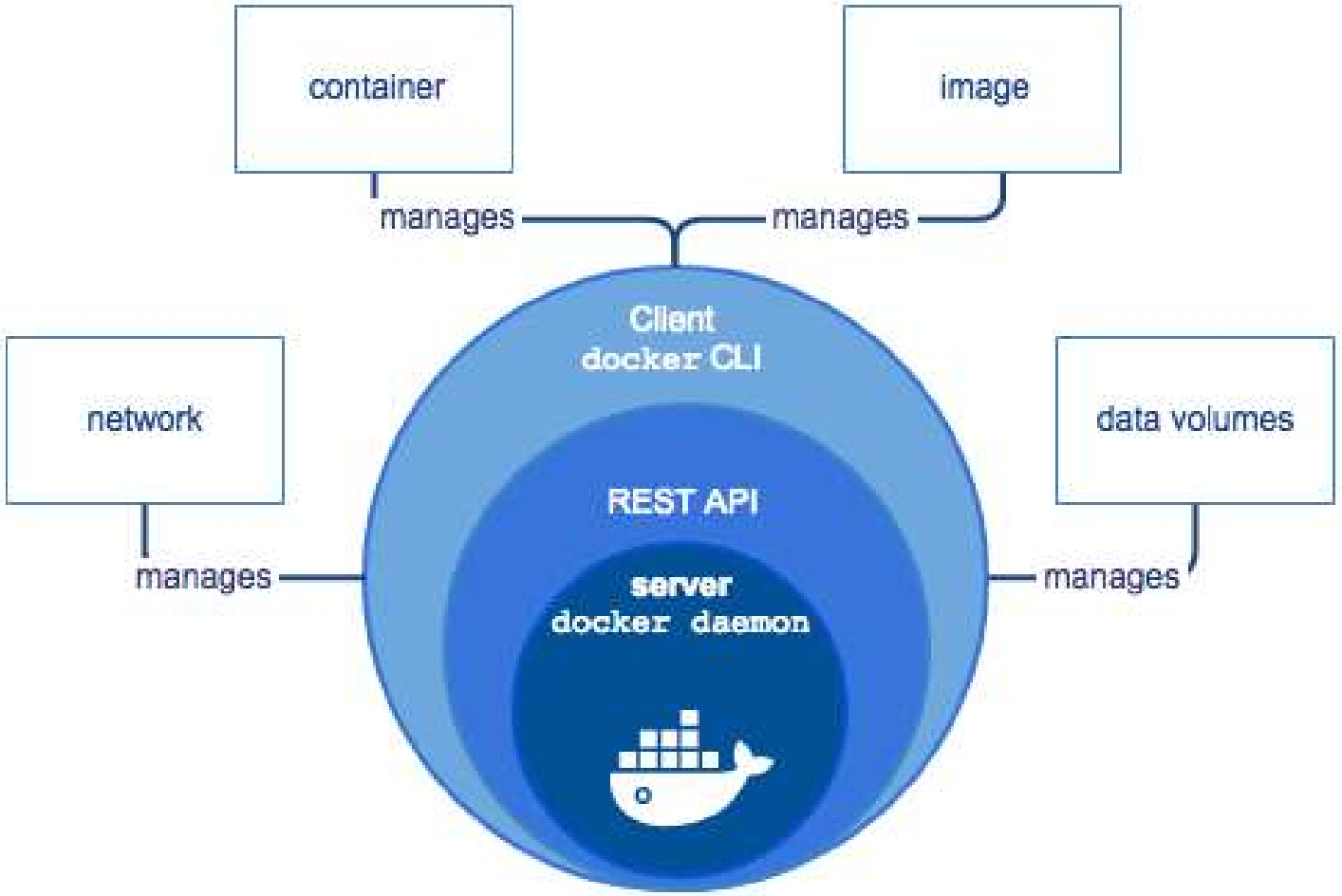
- Refer to the Practical Guide on below URL:
  - https://drive.google.com/open?id=1xcQk6enDDQvQls5V9X7MAtJbcGKl8Puk

- **Note:** We will get some hands-on for 30 minutes before moving on to next topic

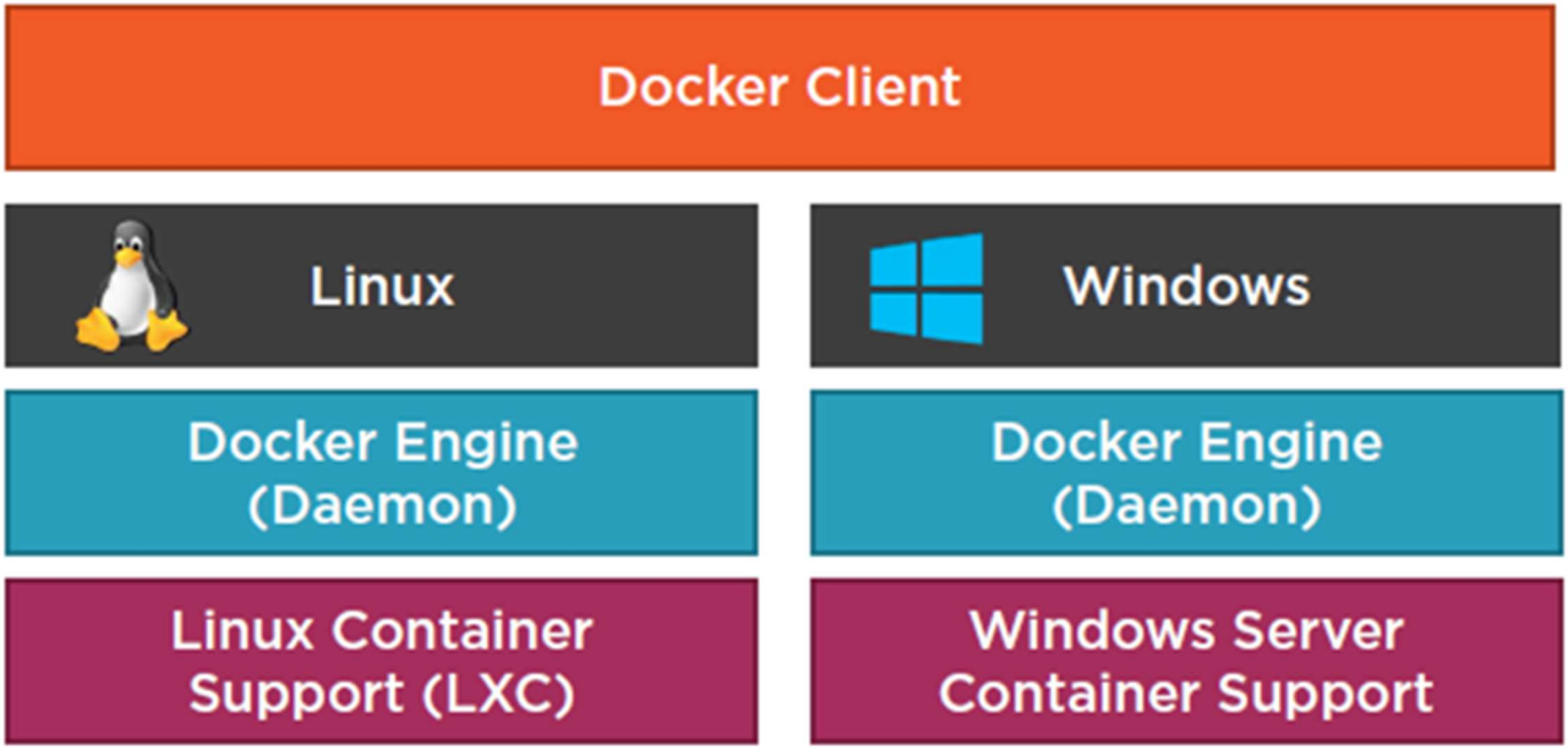
Docker Engine



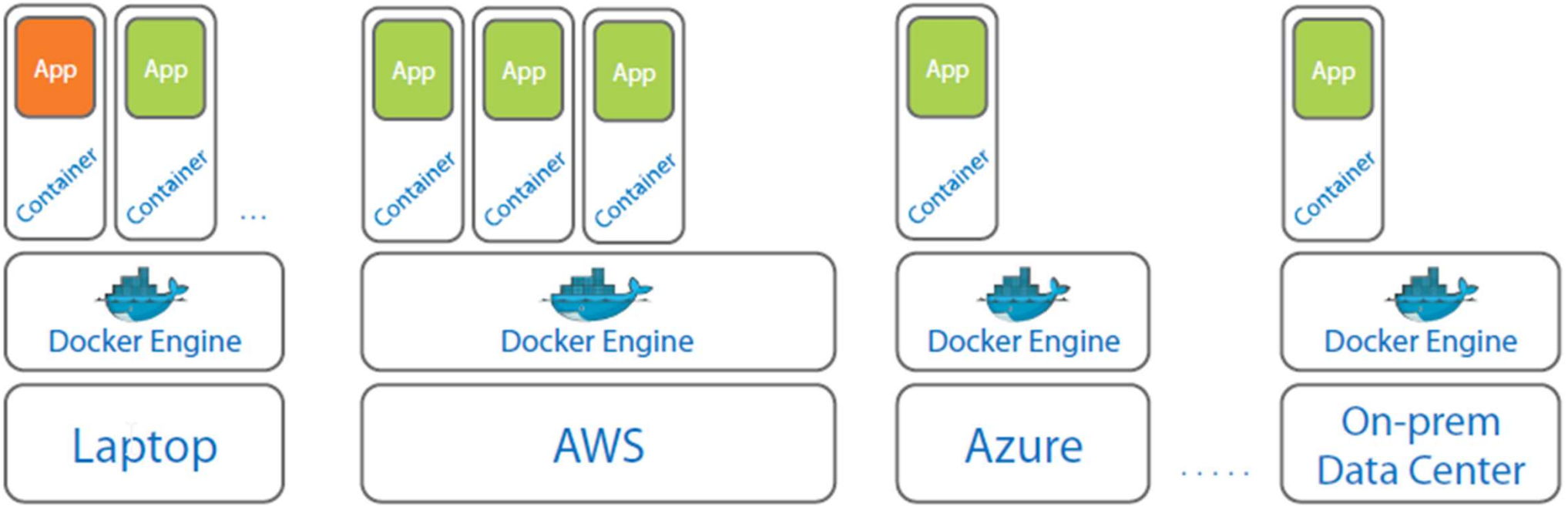
Docker Engine



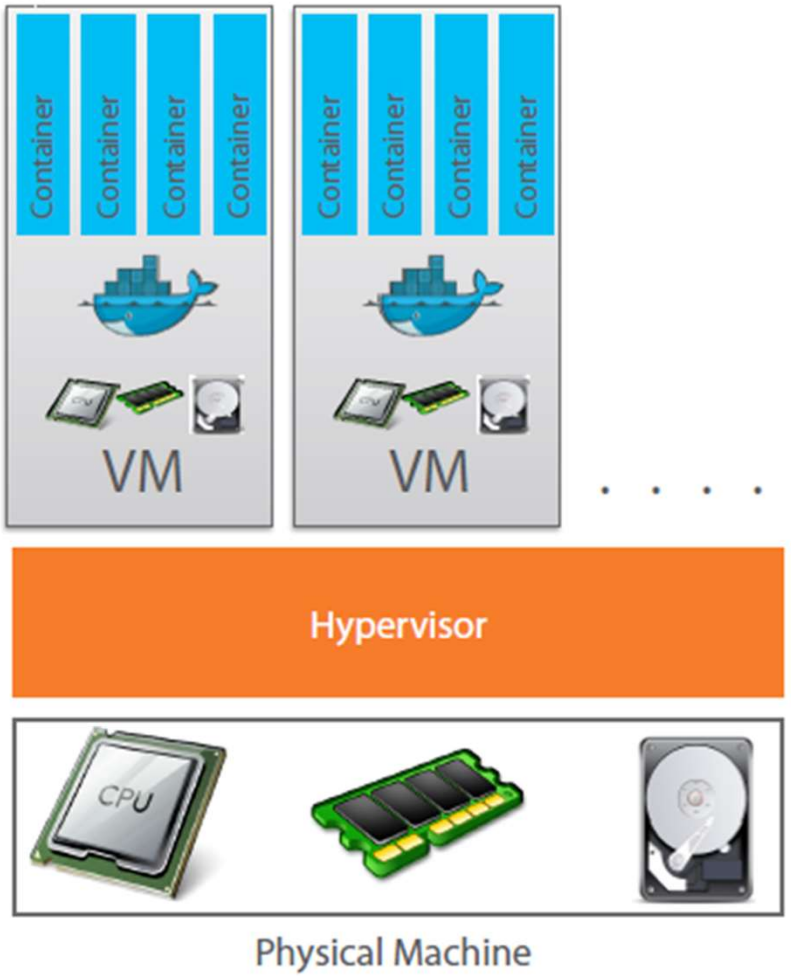
Where does Docker Run?

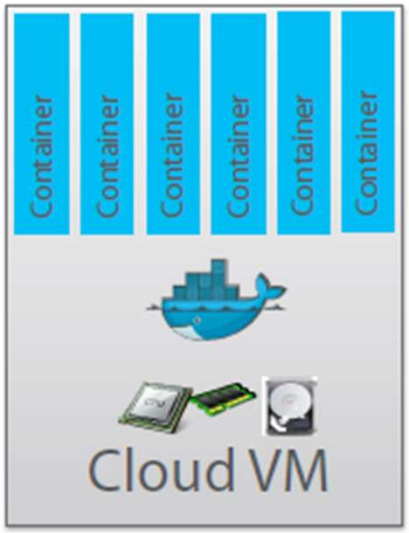


Docker can run anywhere



Docker on Physical Machine and also on Cloud

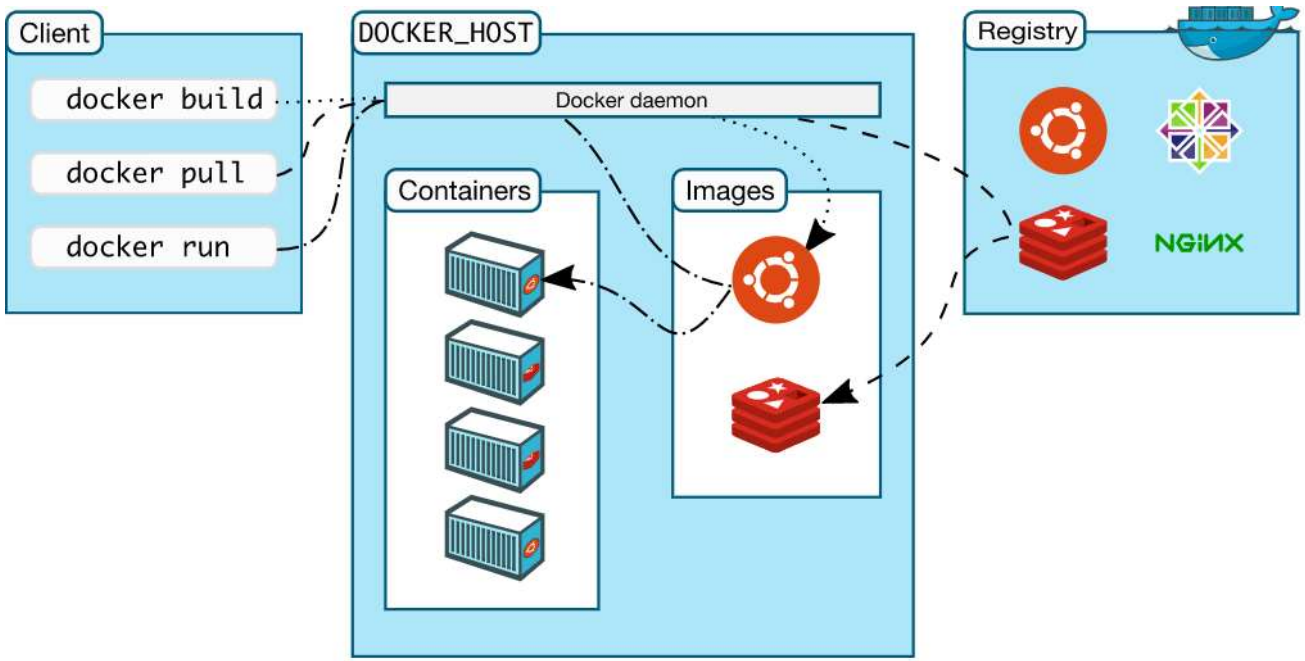




What can I use Docker for?

- Fast, consistent delivery of your applications
  - Continuous integration and continuous delivery (CI/CD) workflows.
  - Examples:
    - Developers write code and share their work with their colleagues using Docker containers.
    - Use Docker to push their applications into a test environment and execute tests.
    - When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.
    - When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.
- Responsive deployment and scaling
  - Dynamically manage workloads, scaling up or tearing down applications and services
- Running more workloads on the same hardware

Docker Architecture



- Docker uses a client-server architecture.
- Docker client talks to the Docker daemon
- The Docker client and daemon can run on the same system, or can connect a client to a remote Docker daemon.
- The Docker client and daemon communicate using a REST API

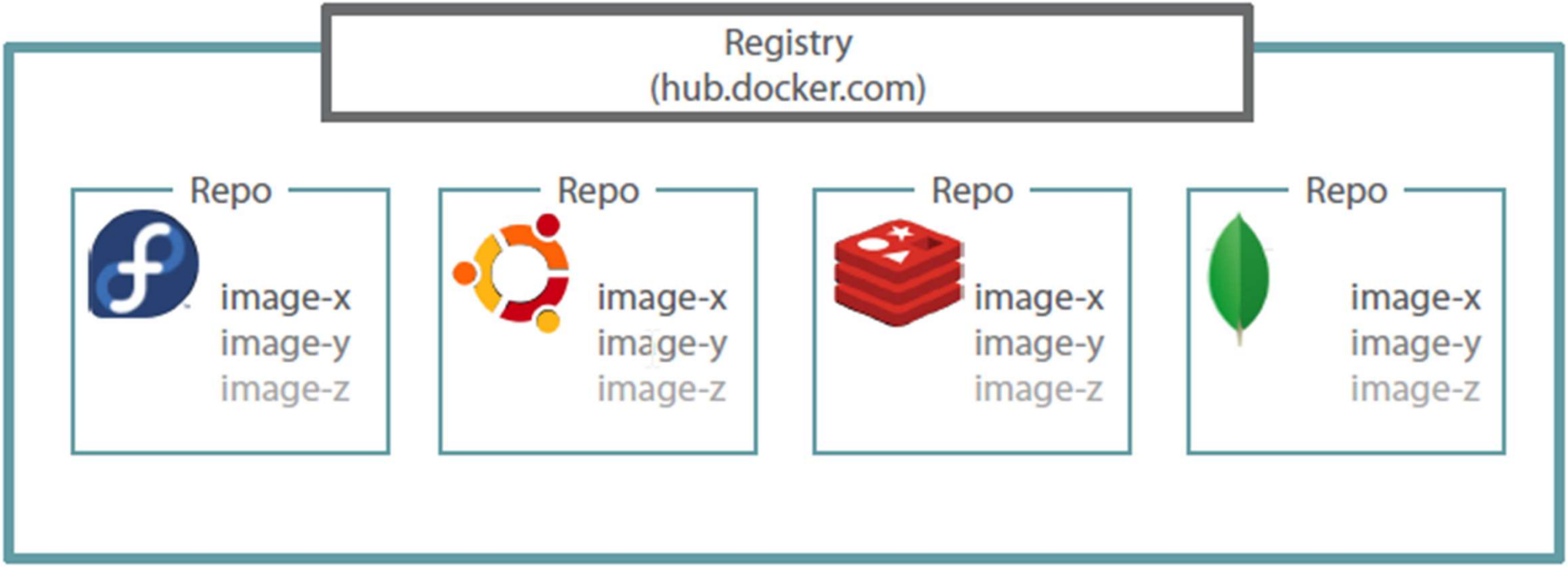
Image

- Persisted snapshot that can be run
- Common Docker Commands:
  - images: List all local images
  - run: Create a container from an image and execute a command in it
  - tag: Tag an image
  - pull: Download image from repository
  - rmi: Delete a local image

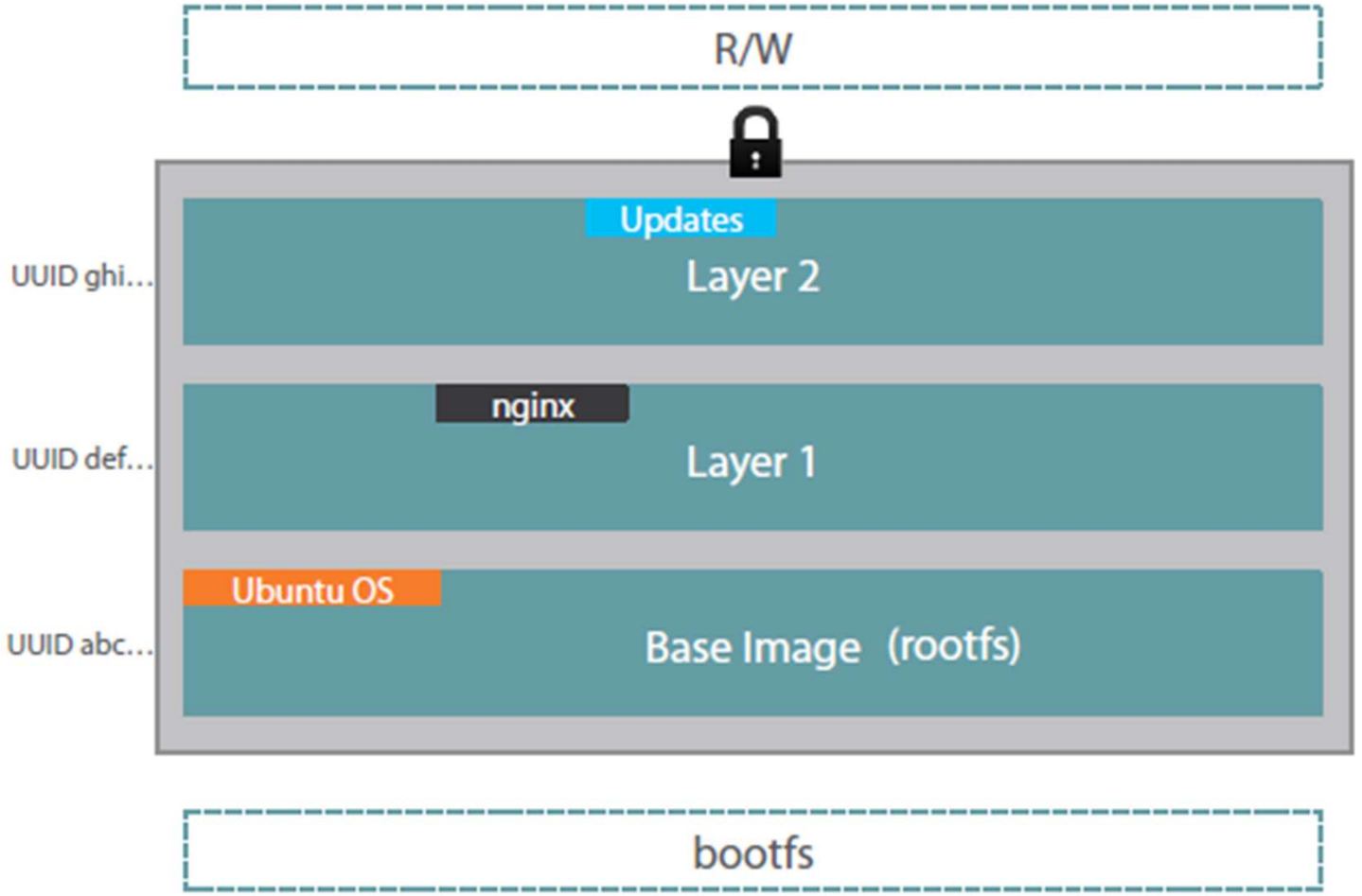
Container

- Runnable instance of an image
- Common Docker Commands
  - ps: List all running containers
  - ps –a: List all containers (incl. stopped)
  - top: Display processes of a container
  - start: Start a stopped container
  - stop: Stop a running container
  - pause: Pause all processes within a container
  - rm: Delete a container
  - commit: Create an image from a container

Docker Registry



Layers in Images



Docker Toolbox

- Docker Toolbox is for older Mac and Windows systems that do not meet the requirements of Docker Desktop.
- Docker Toolbox installs the
  - Oracle VirtualBox
  - Docker Engine
    - For running the docker commands
  - Docker Client,
    - Connects with the docker engine
  - Docker Machine,
    - For running docker-machine commands
  - Docker Compose
    - for running the docker-compose commands
  - Kitematic
    - Docker GUI

Install Docker Toolbox for Windows

- To run Docker, your machine must have a 64-bit operating system running Windows 7 or higher.
- Additionally, you must make sure that virtualization is enabled on your machine.
- Download from:
  - https://github.com/docker/toolbox/releases
- Install the downloaded setup and follow on screen instructions

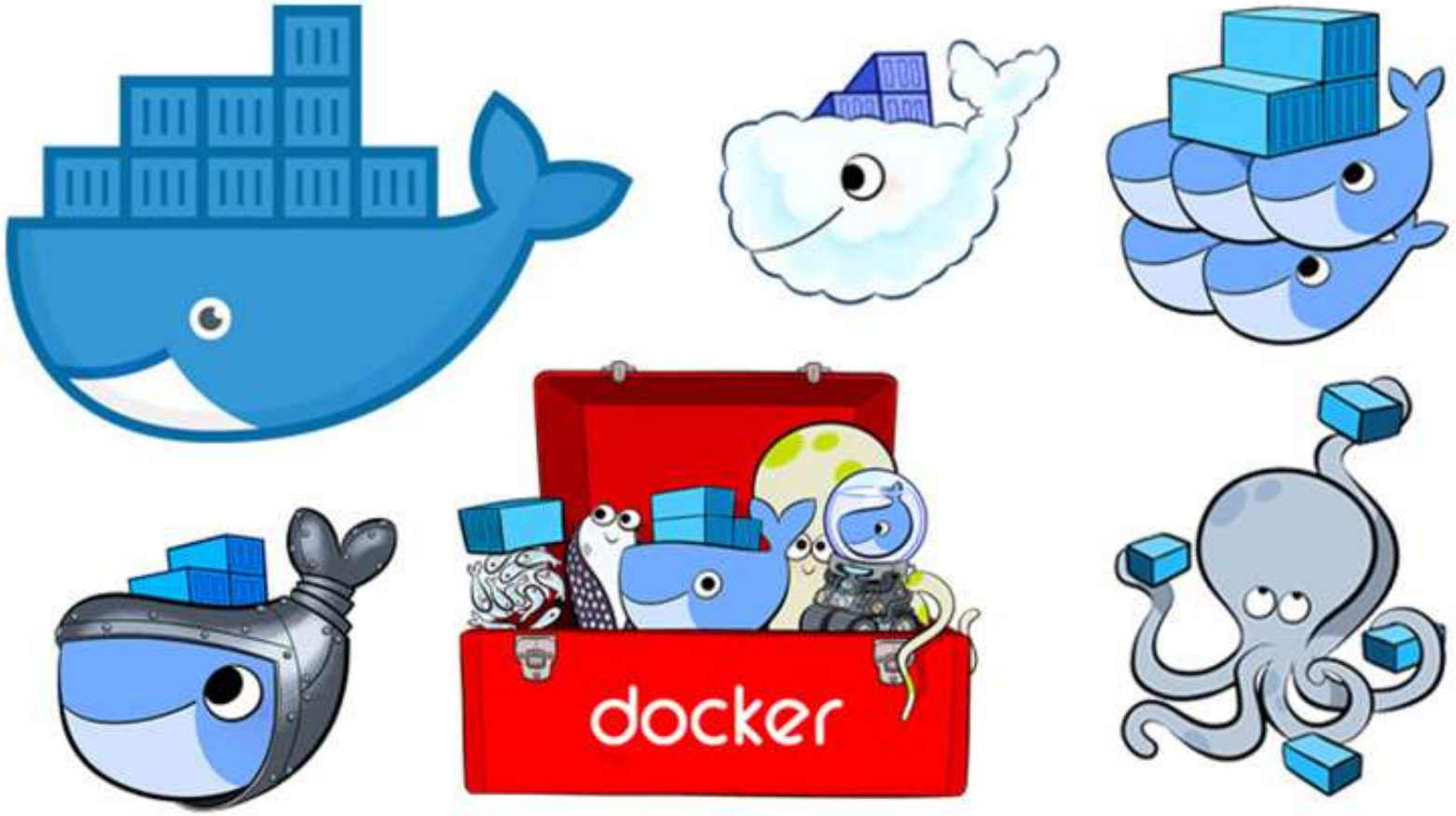
Install Docker Toolbox for Ubuntu

- Refer to our Docker Commands Guide:
  - https://drive.google.com/open?id=1xcQk6enDDQvQls5V9X7MAtJbcGKl8Puk

Hands-On

- We need to do the below hands-on:
  - Install and configure Docker
  - Deploy a Ubuntu 14.04 Server
  - ssh to Ubuntu server
  - Install Docker engine on Ubuntu 14.04
  - Validate docker engine is successfully installed
  - Launch a docker container
  - Login to container
  - Work in a container
  - List containers
  - Pause a container
  - Un-pause a container
  - Delete container
- Refer to the command guide for instructions

Docker Ecosystem

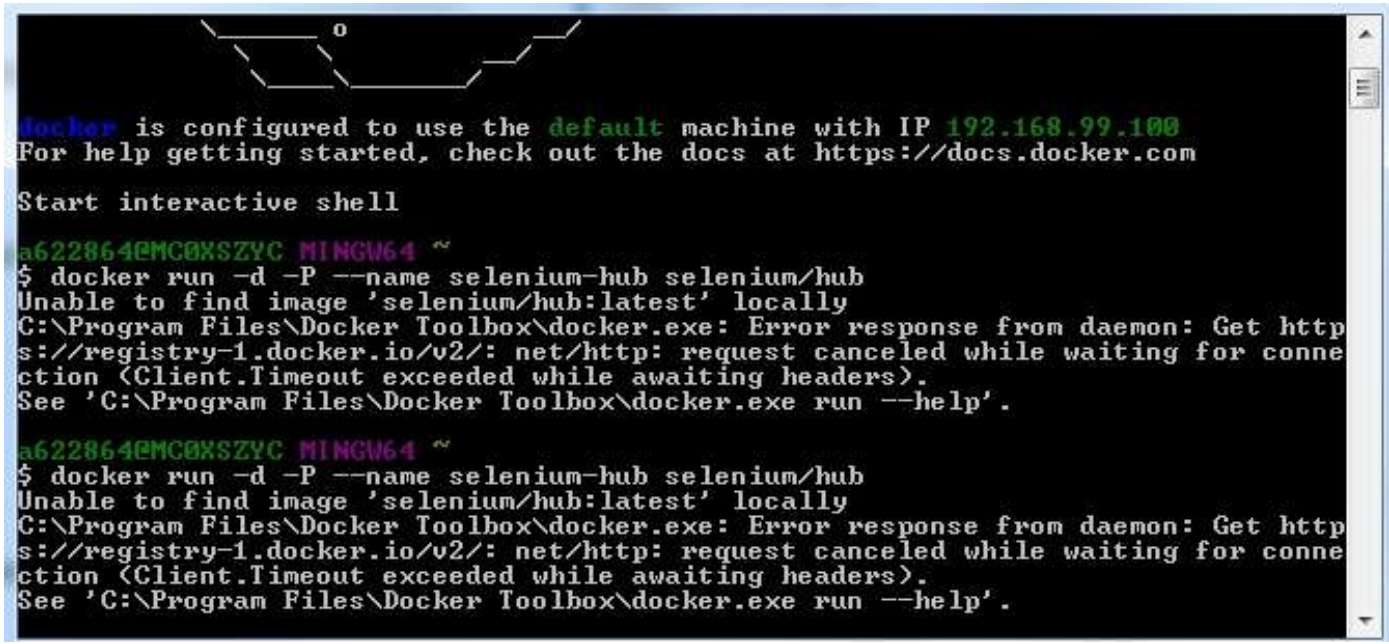


Docker Daemon

- Docker uses a client-server architecture.
- The Docker Daemon runs on your host operating system.
- This as a service that runs in the background.
- It is the brains of the operation when it comes to managing Docker containers.
- Docker client talks to the Docker daemon

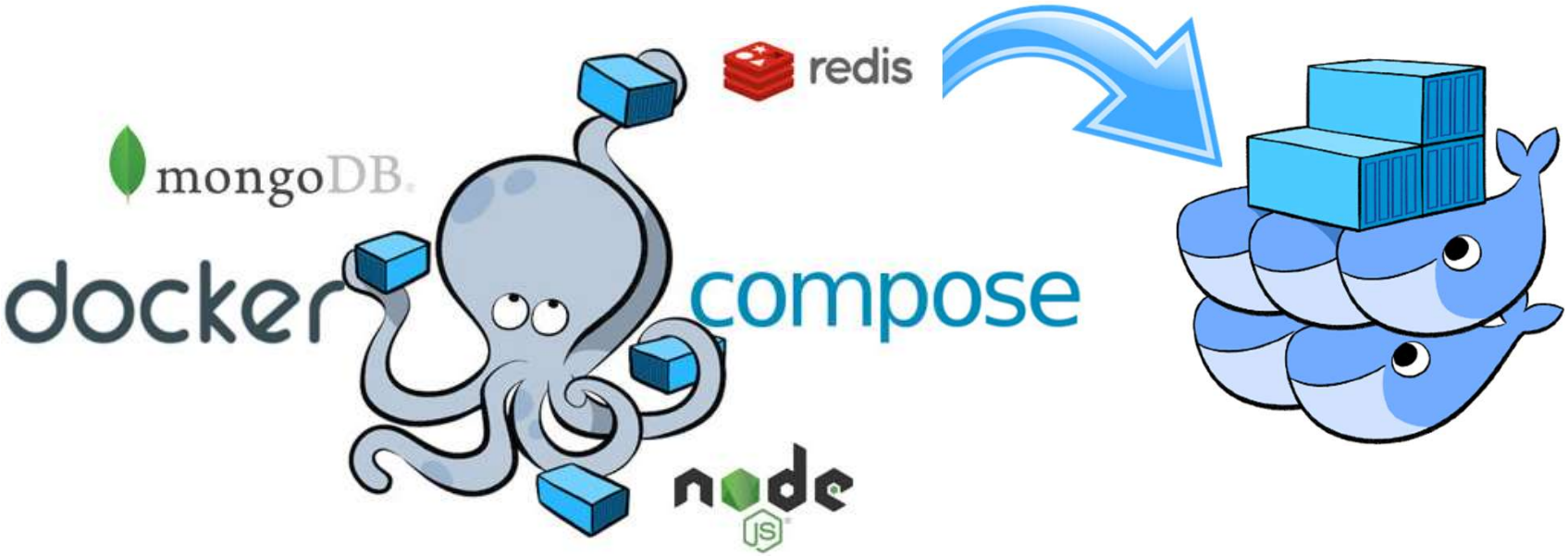
Docker CLI

- To interact with the Docker Daemon.
- The Docker Daemon exposes an API and the Docker CLI consumes that API.



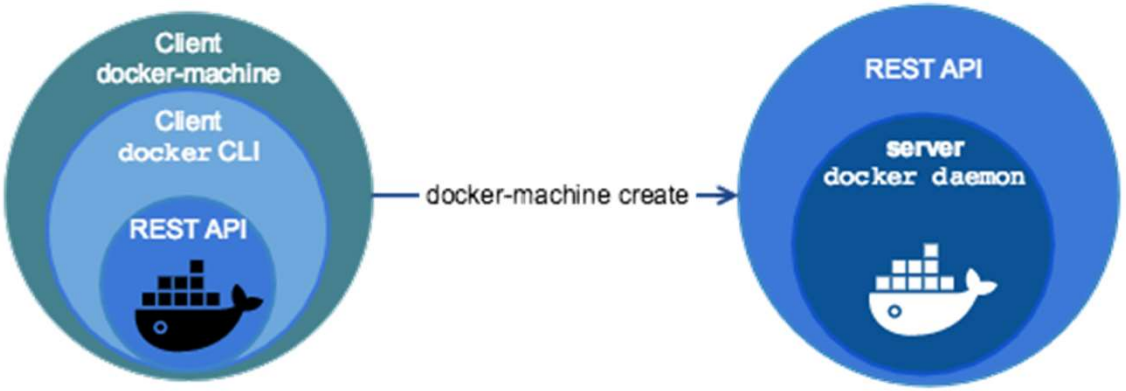
Docker Compose

- It lets you declare Docker CLI commands in the form of YAML



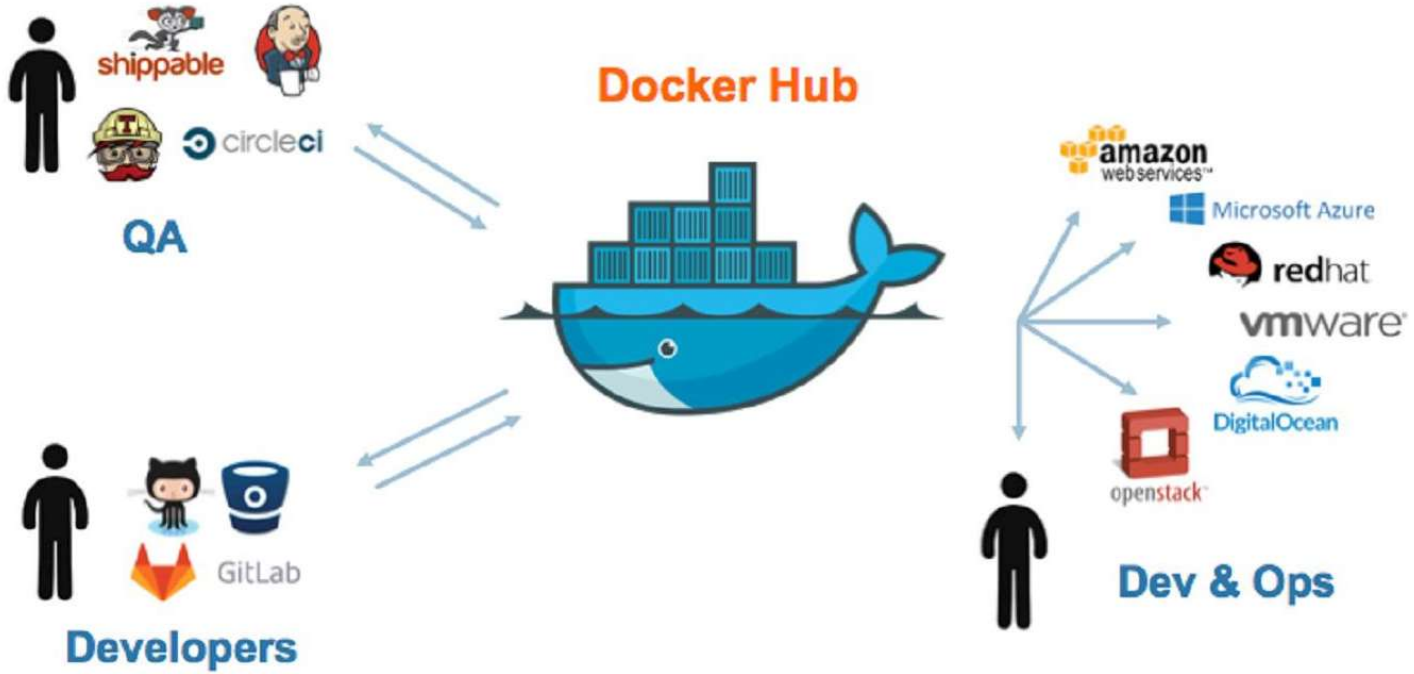
Docker Machine

- A command line tool that helps to create servers and then install Docker Engine on them.



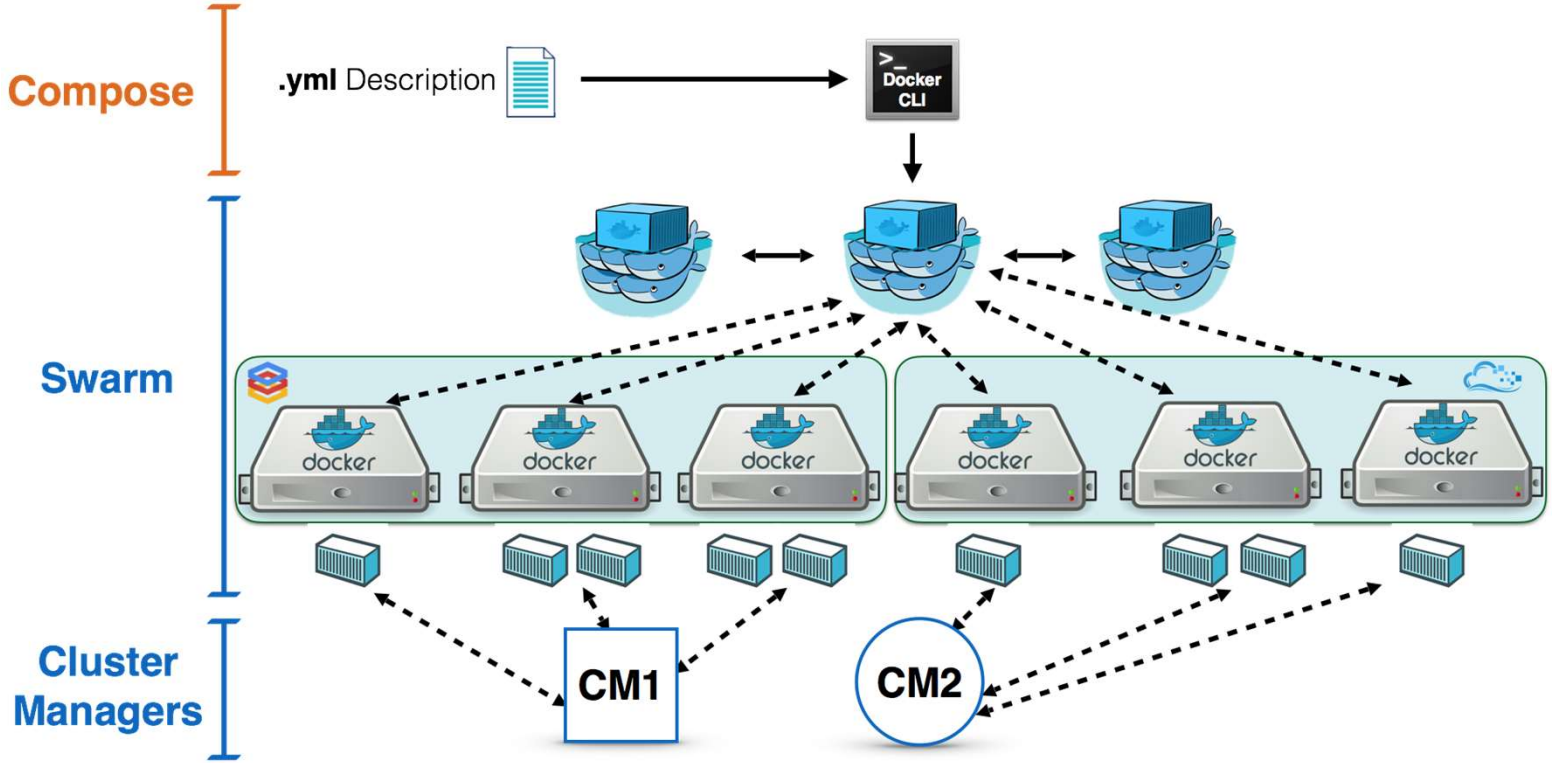
Docker Hub

- Are websites you can visit to find Docker images and even store your own (both publicly and privately).
- https://hub.docker.com/



Docker Swarm

- To manage a cluster of servers which can house 1 or more services.



Docker Swarm

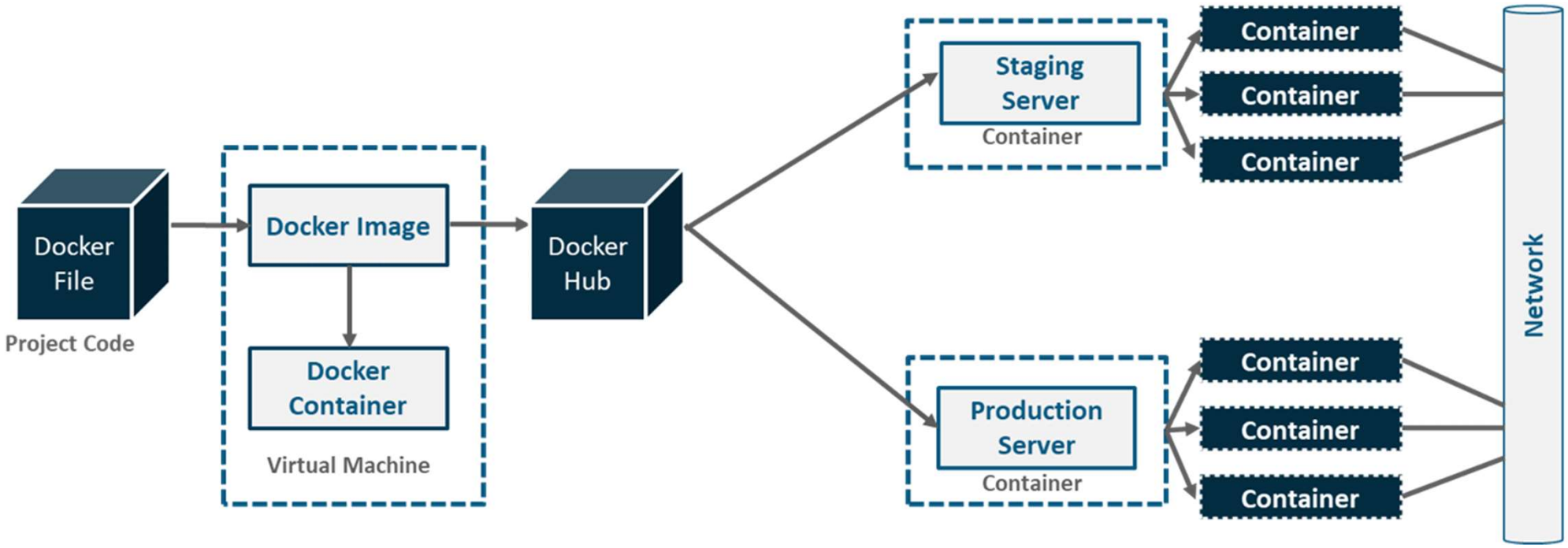
- A group of machines
- Have been configured to join together in a cluster.
- Activities are controlled by a swarm manager.
- Is a container orchestration tool
- Allows the user to manage multiple containers deployed across multiple host machines.
- Benefit: High Availability


thanks

Day 2

Docker Networking

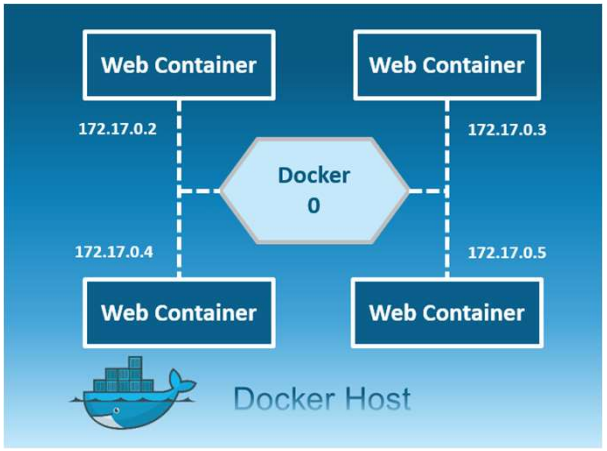
- Before I deep dive into Docker Networking let me show you the workflow of Docker.

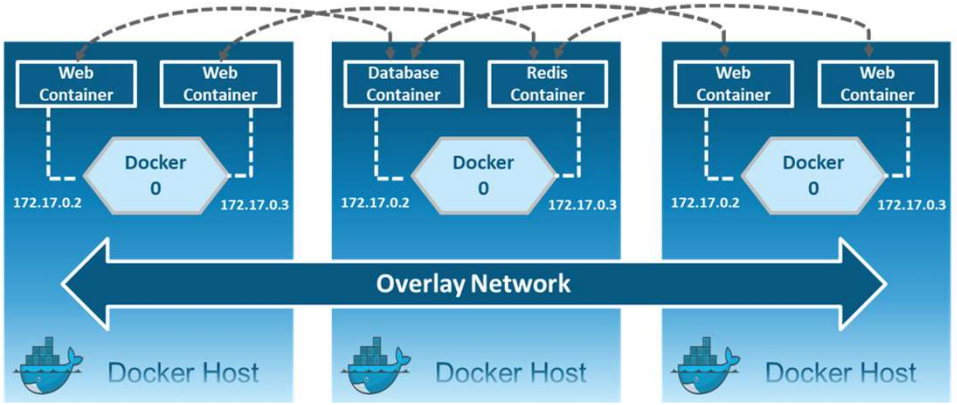


Goals of Docker Networking

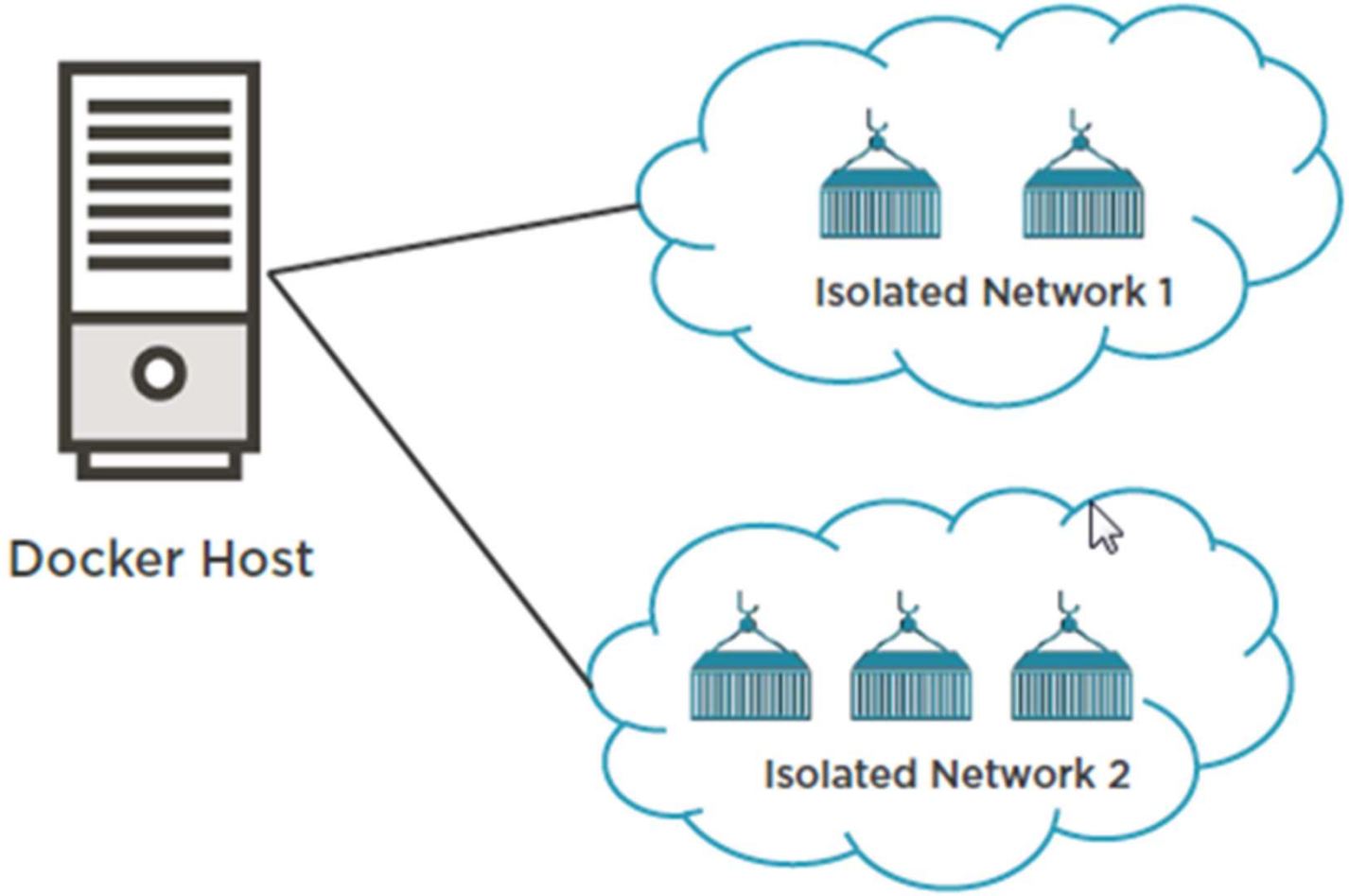
Network Drivers

- Bridge
  - Private default internal network created by docker on the host
  - All containers get an internal IP address
  - These containers can access each other, using this internal IP
- Host
  - This driver removes the network isolation between the docker host and the docker containers to use the host's networking directly
- None
  - Overlay





Understanding Container Networks



Use the default bridge network

- docker network ls
  - NETWORK ID NAME DRIVER SCOPE
  - 17e324f45964 bridge bridge local
  - 6ed54d316334 host host local
  - 7092879f2cc8 none null local
- docker run -dit --name alpine1 alpine ash
- docker run -dit --name alpine2 alpine ash
  - Because you have not specified any --network flags, the containers connect to the default bridge network
- docker container ls / docker ps
  - Check that both containers are actually started
- docker network inspect bridge
  - Inspect the bridge network to see what containers are connected to it.

Use the default bridge network

- docker attach alpine1
  - Use the docker attach command to connect to alpine1.
- ip addr show
- ping google.com
  - Access google
- ping -c 2 172.17.0.3
  - ping the second container
- ping -c 2 alpine2
  - Ping by hostname

- Remove the containers you created
  - Docker stop alpine1; docker rm alpine1
  - Docker stop alpine2; docker rm alpine2

Use user-defined bridge networks

- docker network create --driver bridge alpine-net
  - Create the alpine-net network
- docker network ls
- docker network inspect alpine-net
- Now lets create 4 containers and attach those to the network
- docker run -dit --name alpine1 --network alpine-net alpine ash
- docker run -dit --name alpine2 --network alpine-net alpine ash
- docker run -dit --name alpine3 alpine ash # Will be connected to bridge
- docker run -dit --name alpine4 --network alpine-net alpine ash
- docker network connect bridge alpine4 # Connected to 2 networks
- docker container ls
- docker network inspect bridge
- docker network inspect alpine-net

Use user-defined bridge networks

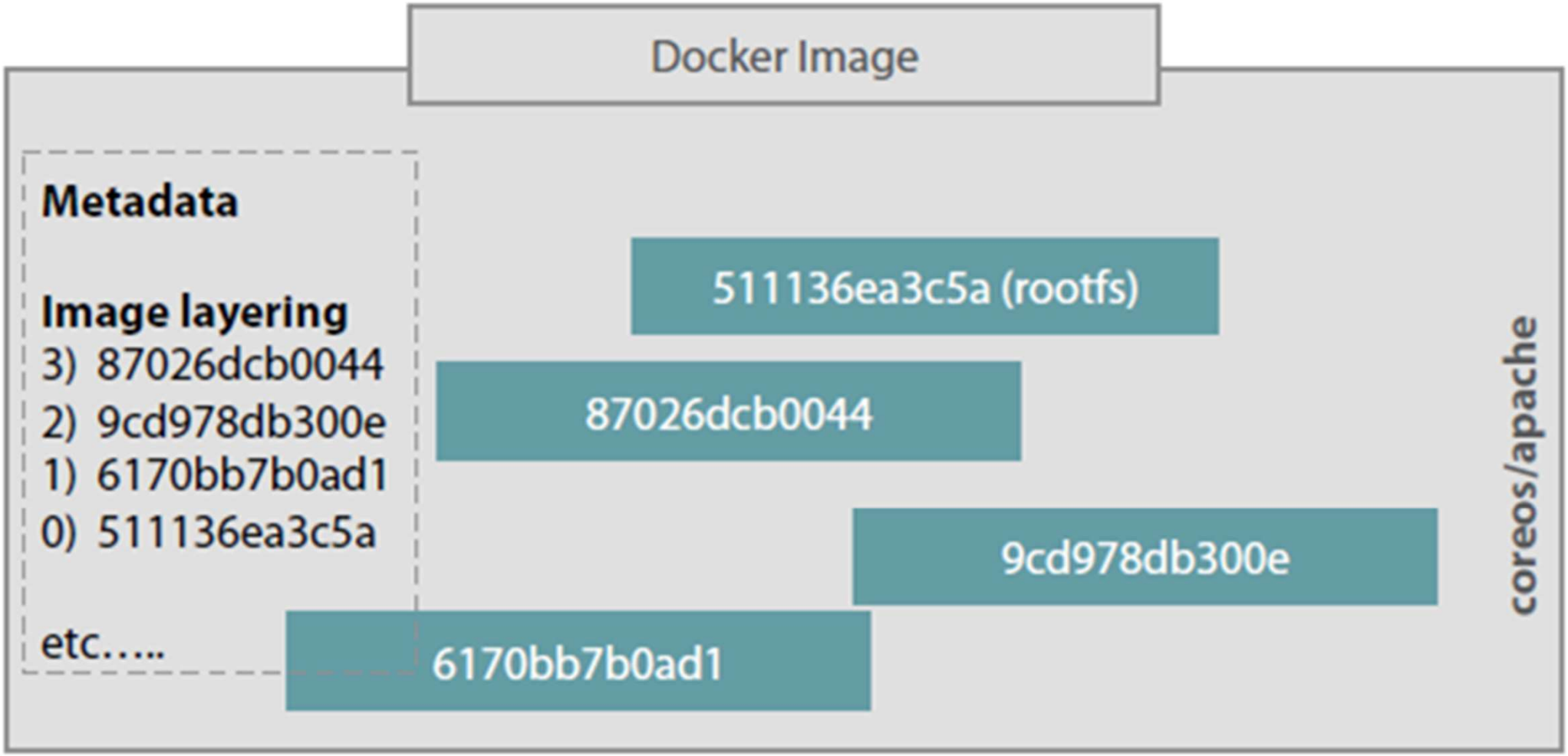
- Now let’s ping the containers from each container to another

- Finally Stop and remove all containers and the alpine-net network.
  - $ docker container stop alpine1 alpine2 alpine3 alpine4
  - $ docker container rm alpine1 alpine2 alpine3 alpine4
  - $ docker network rm alpine-net

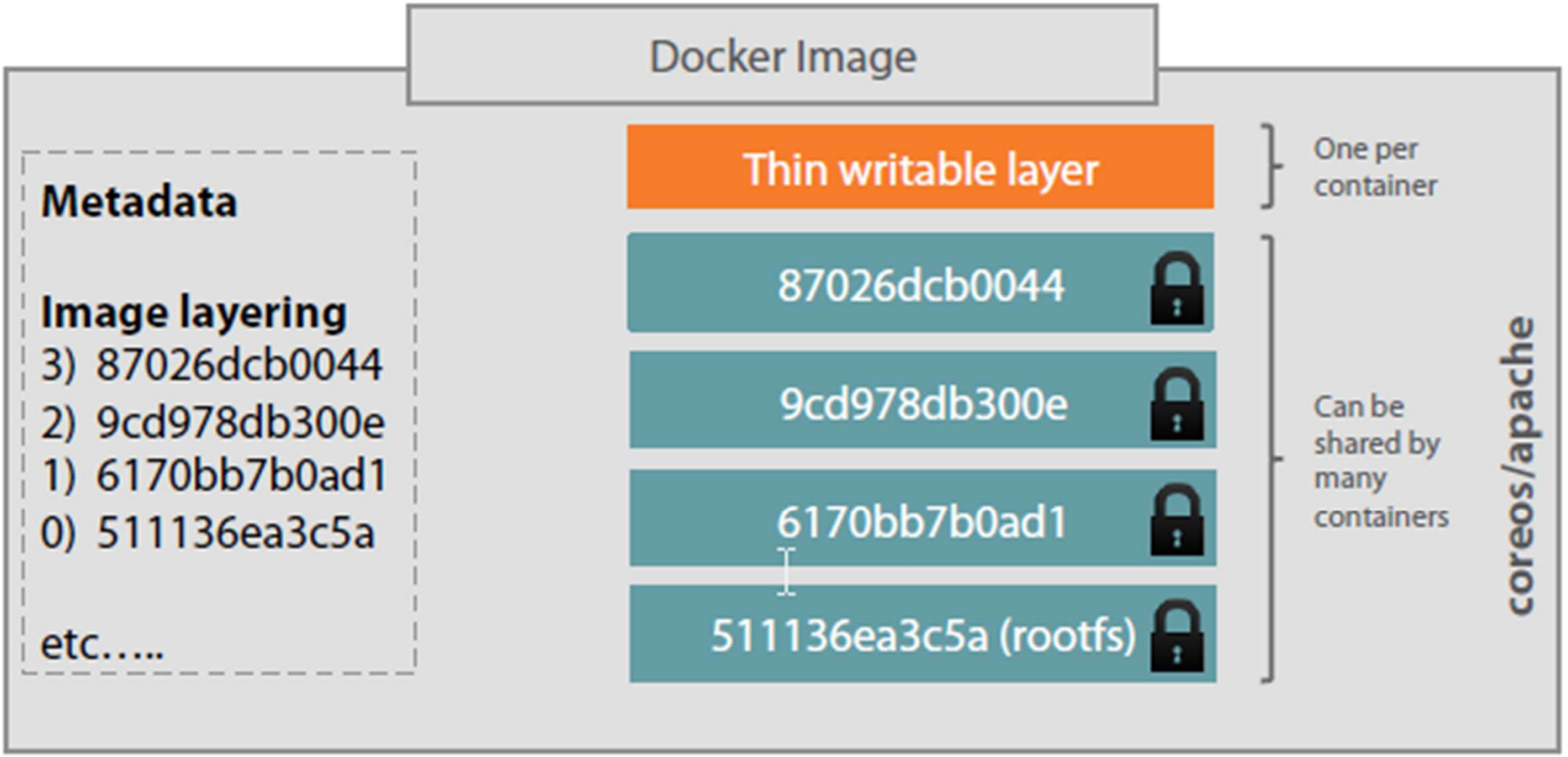
Networking using the host network

- docker run --rm -d --network host --name my_nginx nginx
- curl localhost:80
- sudo netstat -tulpn | grep :80
  - Verify which process is bound to port 80, using the netstat command
- docker container stop my_nginx

Layers in Docker Image



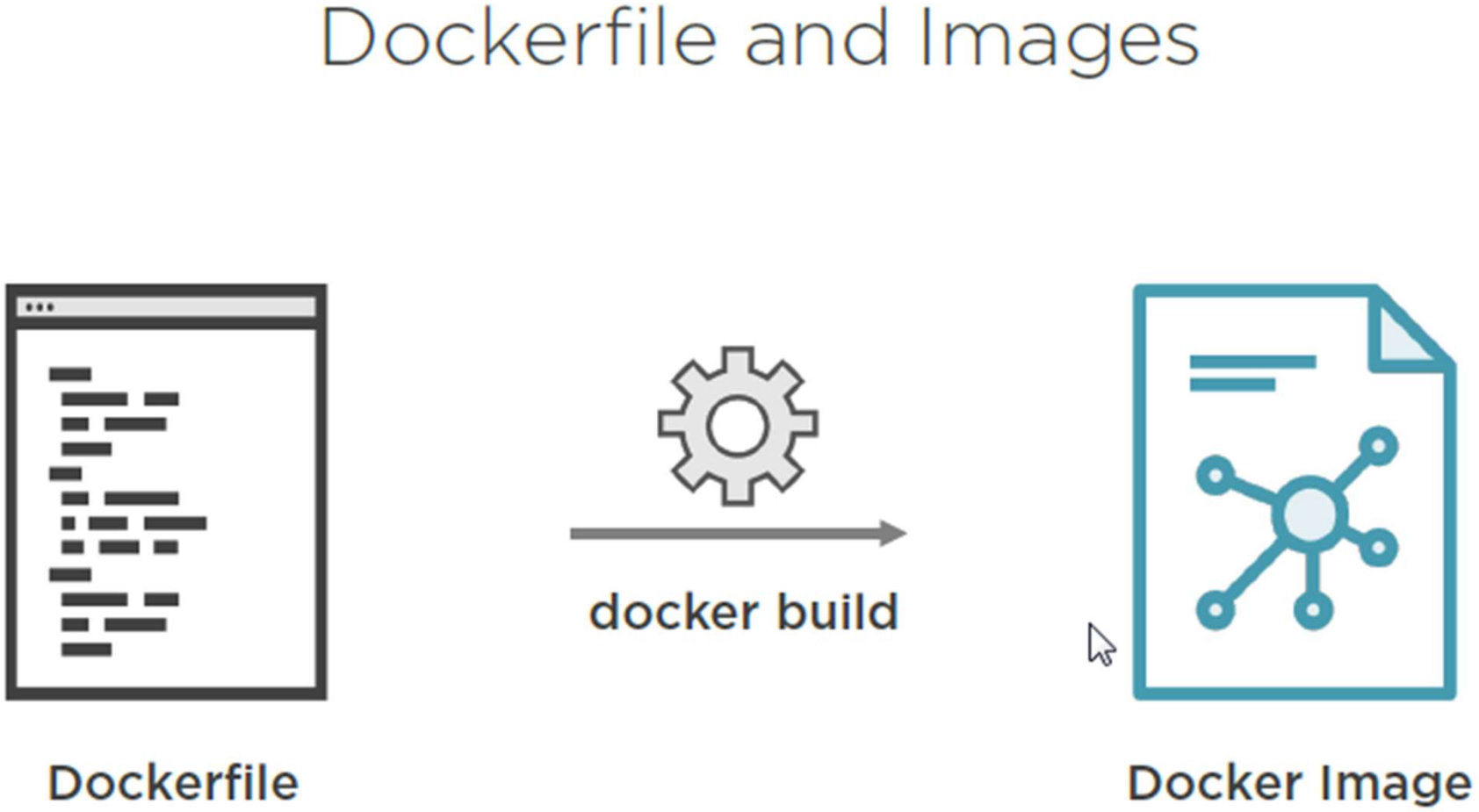
Layers in Docker Image



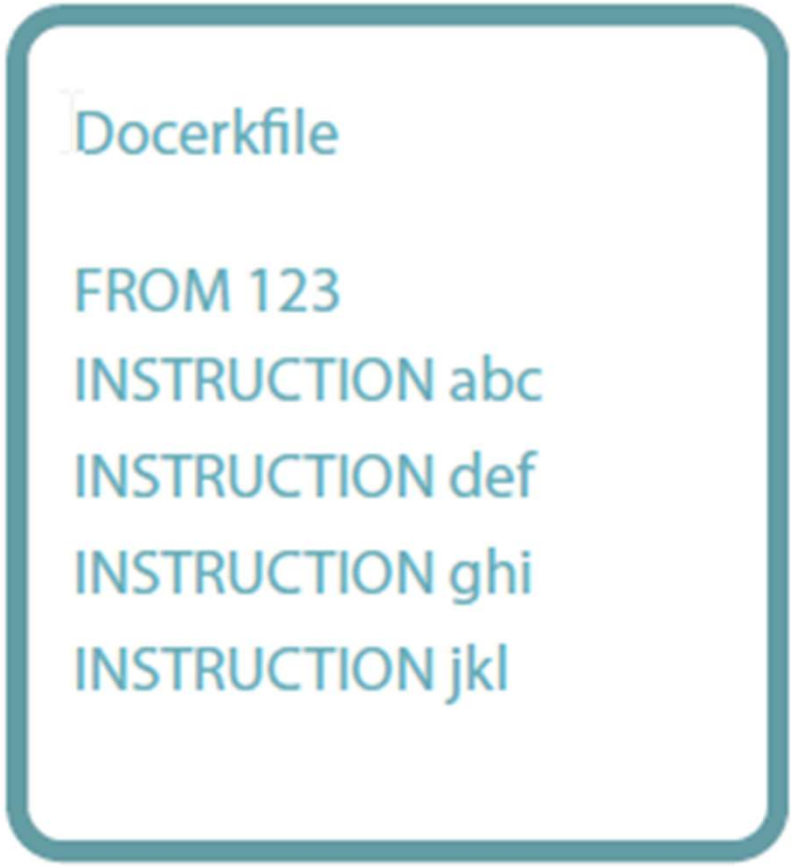
Create Dockerized Application

- We can dockerize our application using dockerfile
  - Dockerfile Create images automatically using a build script: «Dockerfile»
  - It Can be versioned in a version control system like Git
  - Docker Hub can automatically build images based on dockerfiles on Github
- This is a basic Dockerfile we need to dockerize a node application
  - FROM node:4-onbuild
  - RUN mkdir /app
  - COPY . /app/
  - WORKDIR /app
  - RUN npm install
  - EXPOSE 8234
  - CMD ["npm", "start"]

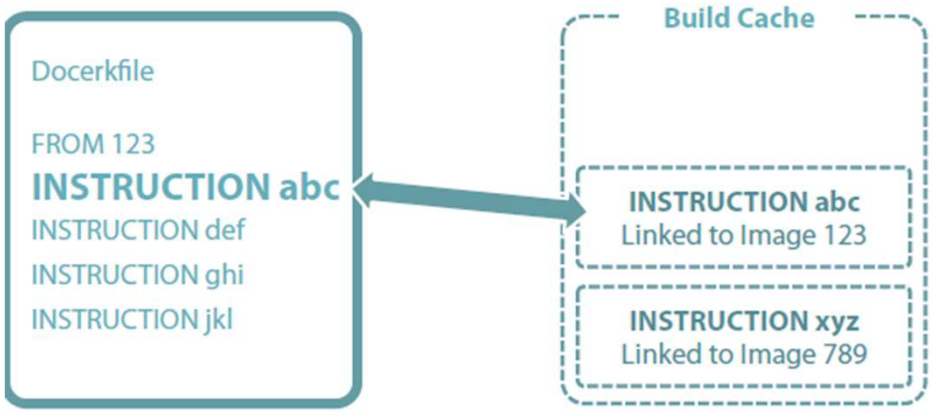
Dockerfile


Dockerfile and Images
docker build
Dockerfile
Docker Image

Dockerfile Template



Dockerfile Build Cache



FROM node:4-onbuild

- Pulls/downloads a base image from docker hub which is a public hub for docker images.
- For running a node appication you need to install node in your system

RUN mkdir /app

- In this command we make create an empty directory which will be our workin directory with the code files.

COPY . /app/

- Copies all files in current directory to the newly created app directory.
- Your Dockerfile should be in the parent directory of your project.

WORKDIR /app

- To switch from current directory to the app directory where we will run our application.

RUN npm install

- This npm command is related to node application.
- When we copied all dependencies, our main file - package.json would have been copied.
- So running above command installs all dependencies from the file and creates a node_modules folder with mentioned node packages.

EXPOSE 8234

- This command is to expose a port we want our docker image to run on.

CMD ["npm", "start"]

- This is a command line operation to run a node application.
- It may differ based on projects.

Build Image

- Now once we have our Dockerfile ready lets build an image out of it.
- Assuming you all have docker installed on your system lets follow some simple steps:-
  - Navigate to directory containing Dockerfile.
  - Run the following command on your terminal:-
    - docker build -t myimage .
- docker images
- docker run -p 8234:8234 'your image name'

Dockerfile for java project

- FROM openjdk:8-jdk-alpine
- RUN mkdir /apollo-services
- COPY . /app
- WORKDIR /app
- RUN ./gradlew raml-generate
- RUN ./gradlew clean build
- WORKDIR /app/service-impl/build/libs
- EXPOSE 8080
- ENTRYPOINT ["java","-jar","main.jar"]

Publish Port

- docker run –t –p 8080:80 ubuntu
  - Map container port 80 to host port 8080

Docker Hub

- Public repository of Docker images
  - https://hub.docker.com/
  - docker search [term]

- Use my own registry
  - To pull from your own registry, substitute the host and port to your own:
    - docker login localhost:8080
    - docker pull localhost:8080/test-image

Resource Usage

- docker top [container id]
- docker stats [container id]
- docker inspect [container id]

- docker stats –all

Clean Up

- docker stop $(docker ps -a -q) #stop ALL containers
- docker rm -f $(docker ps -a -q) # remove ALL containers

Advanced Docker commands

- docker save [OPTIONS] IMAGE [IMAGE…]
  - – Saving image to tar archives
  - Ex: docker save alpine > alpine.tar
  - Ex: docker save --output alpine.tar alpine

- docker search [OPTIONS] TERM
  - – Searching docker hub for images
  - Ex: docker search textbox

- docker stats [OPTIONS] [CONTAINER…]
  - – Displaying resource usage statistics
  - docker stats <containerid>

Advanced Docker commands

Command	Description
docker system df	Show docker disk usage
docker system events	Get real-time events from the server
docker system info	Display system-wide information
docker system prune	Remove unused data

Advanced Docker commands

- docker tag SOURCE_IMAGE[: TAG] TARGET_IMAGE[: TAG]
  - – Creating a target image referring to a source image
  - Ex : docker tag 0e5574283393 fedora/httpd:version1.0
- docker logout
- docker version
  - List info about your Docker Client and Server versions.

- docker container inspect my_container
  - — See lots of info about a container

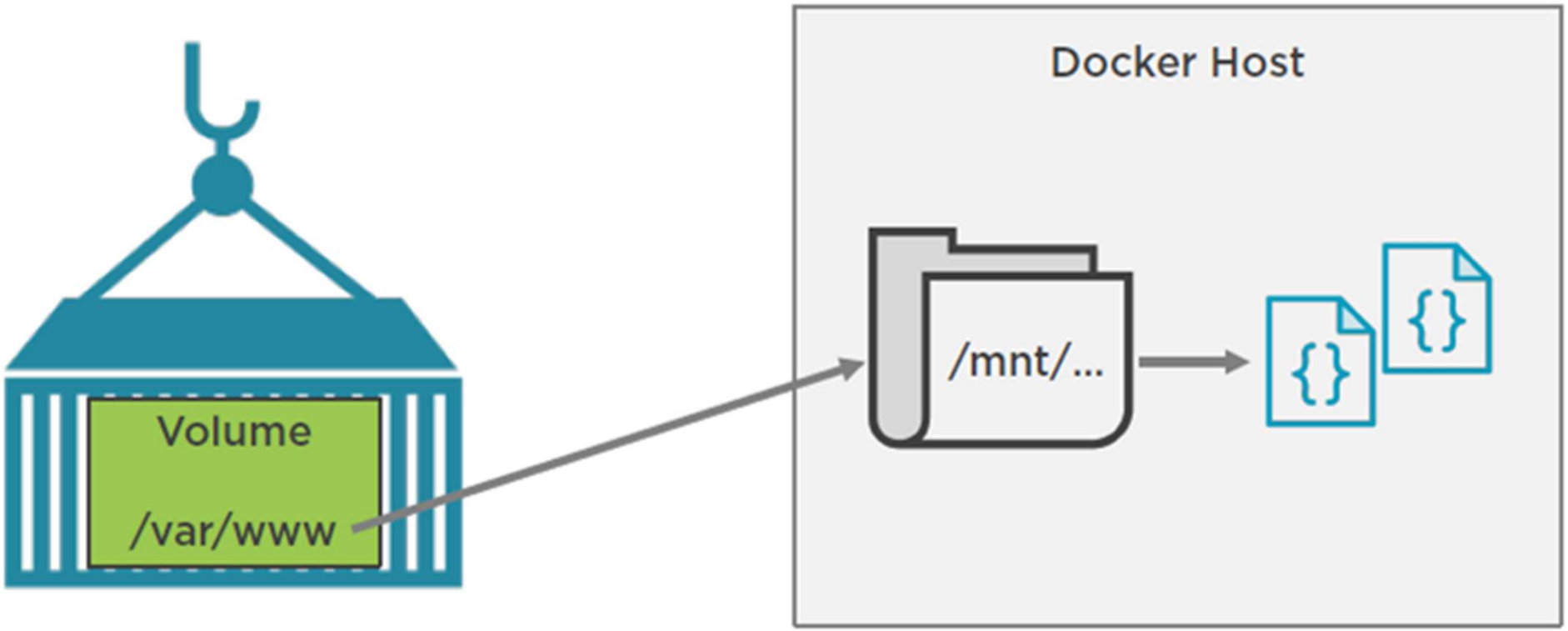
Advanced Docker commands

- docker container logs my_container
- docker container kill my_container
  - — Stop one or more running containers abruptly.
  - It’s like pulling the plug on the TV.
  - Prefer stop in most situations.
- docker container kill $(docker ps -q)
  - — Kill all running containers
- docker container rm my_container
  - — Delete one or more containers.
- docker container rm $(docker ps -a -q)
  - — Delete all containers that are not running.
- docker image rm $(docker images -a -q)
  - — Delete all images.

Persistence Storage

- By default all files created inside a container are stored on a writable container layer.
- This means:
  - The data doesn’t persist when that container is removed
- Docker uses volumes to persist data on host file system.

Docker Volume



Create and manage volumes

- docker volume create my-vol
- docker volume ls
- docker volume inspect my-vol
- docker volume rm my-vol

Start a container with a volume

- docker run -d --name devtest -v myvol2:/app nginx:latest
  - mounts the volume myvol2 into /app/ in the container
- docker inspect devtest
  - To verify that the volume was created and mounted correctly.

- docker container stop devtest
- docker container rm devtest
- docker volume rm myvol2

Use a read-only volume

- docker run -d --name=nginxtest -v nginx-vol:/usr/share/nginx/html:ro nginx:latest
- docker inspect nginx

- docker container stop nginxtest
- docker container rm nginxtest
- docker volume rm nginx-vol

Mount Volumes

- docker run –ti –v /hostLog:/log ubuntu

- Run second container: Volume can be shared
  - docker run –ti –volumesfrom firstContainerName ubuntu

Linked containers

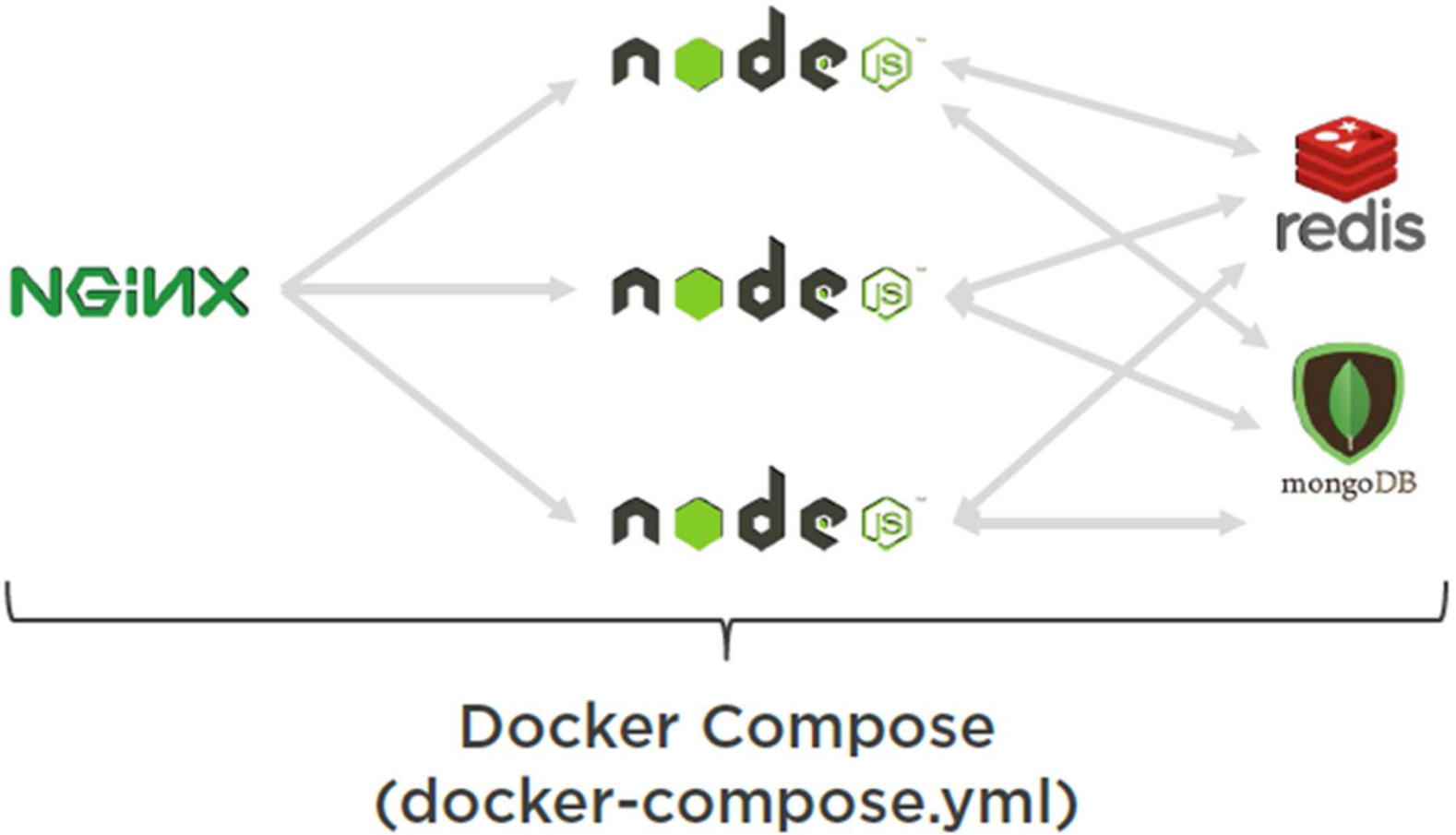
- docker network create mynetwork
- Deploy container-A (install python)
  - docker run -it --name u-python -p 8080:8080 -p 3286:3286 -d --network mynetwork ubuntu /bin/bash
- Deploy a linked container –B (install database server)
  - docker run -it --name u-sql -p 8081:8081 -p 3287:3287 -d --network mynetwork ubuntu /bin/bash
- Write python program on container-A to create table
- Run python program on container –A
- Login to container-B and validate table is created


thanks

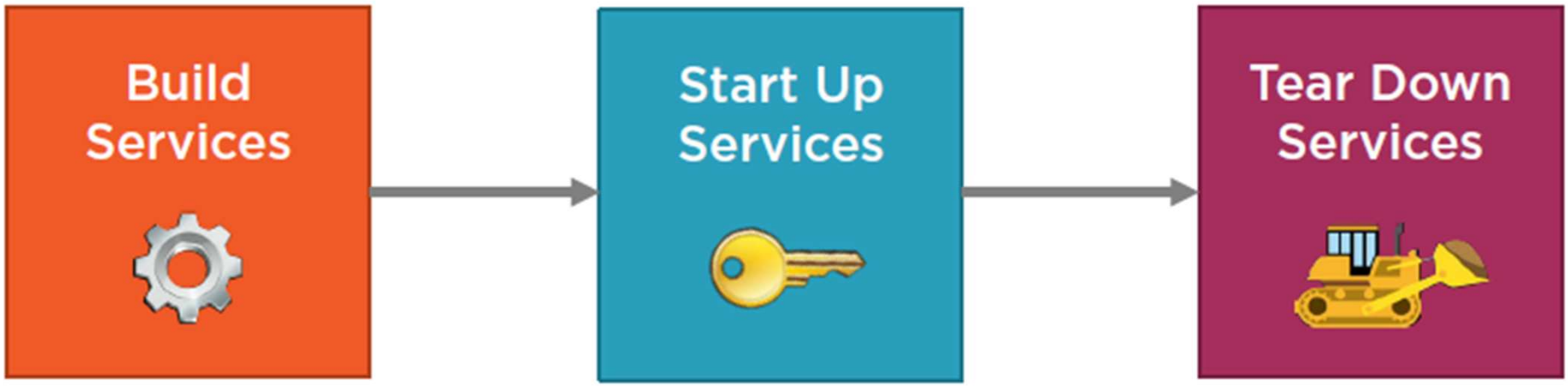
Docker Compose

- Manages the whole application lifecycle:
  - Start, stop and rebuild services
  - View the status of running services
  - Stream the log output of running services
  - Run a one-off command on a service

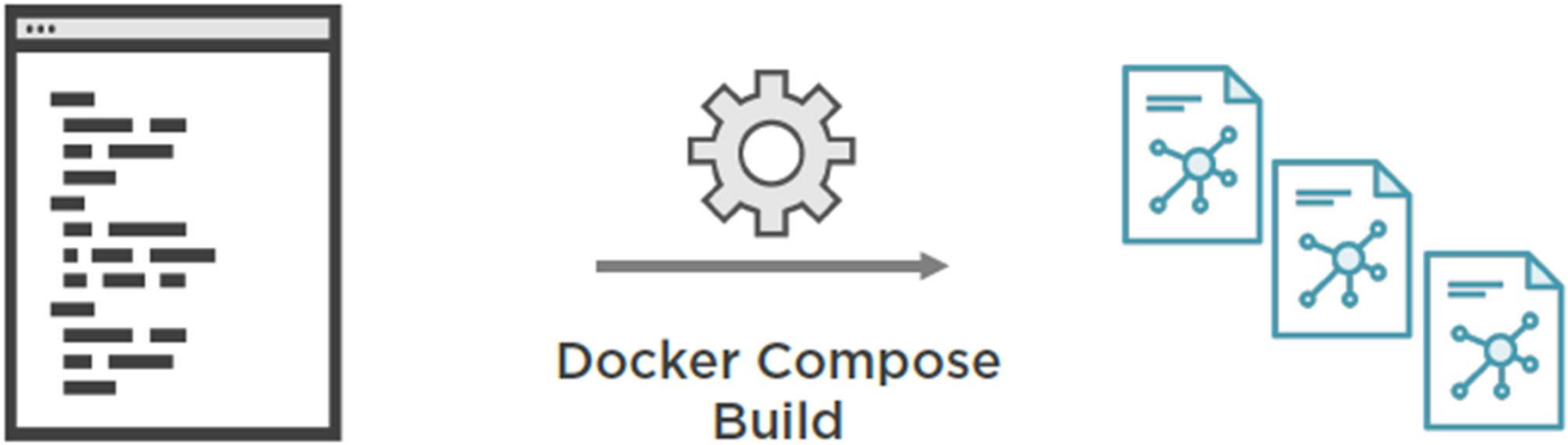
The need for Docker Compose



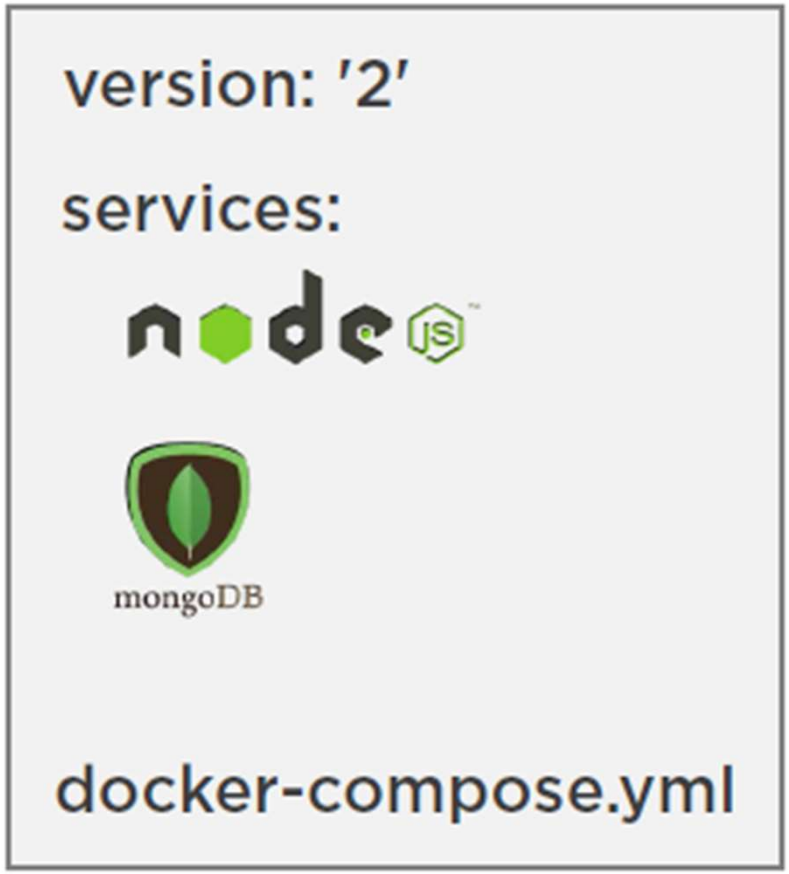
Docker Compose Workflow



The Role of the DockerCompose File



Docker Compose and Services



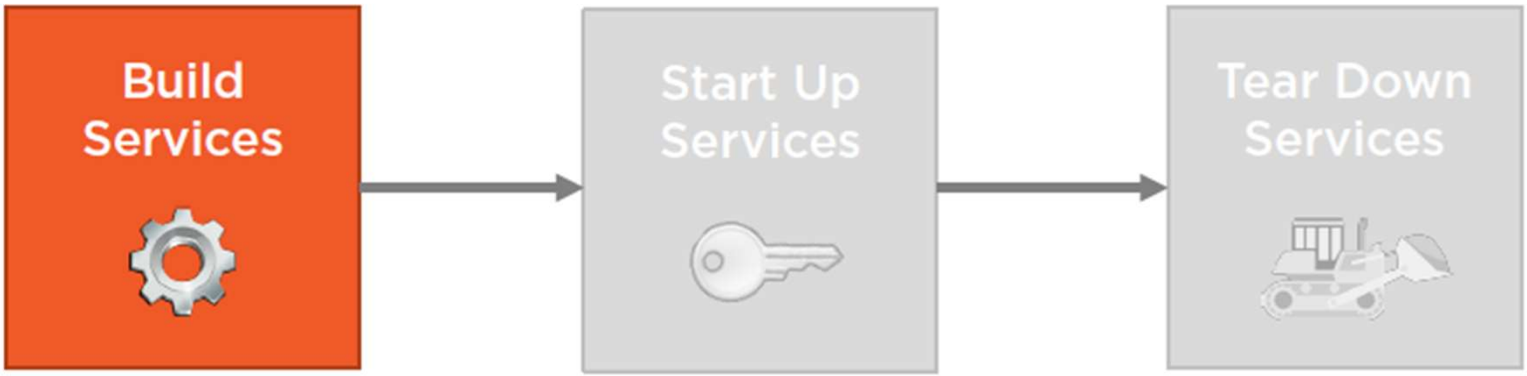
docker-compose.yml Example

- version: '2‘
- services:
  - node:
    - build:
      - context: .
      - dockerfile: node.dockerfile
    - networks:
      - -nodeapp-network
  - mongodb:
    - image: mongo
    - networks:
      - -nodeapp-network
- networks:
  - nodeapp-network
    - driver: bridge

Key Docker Compose Commands

- docker-compose build
- docker-compose up
- docker-compose down
- docker-compose logs
- docker-compose ps
- docker-compose stop
- docker-compose start
- docker-compose rm

Building Services



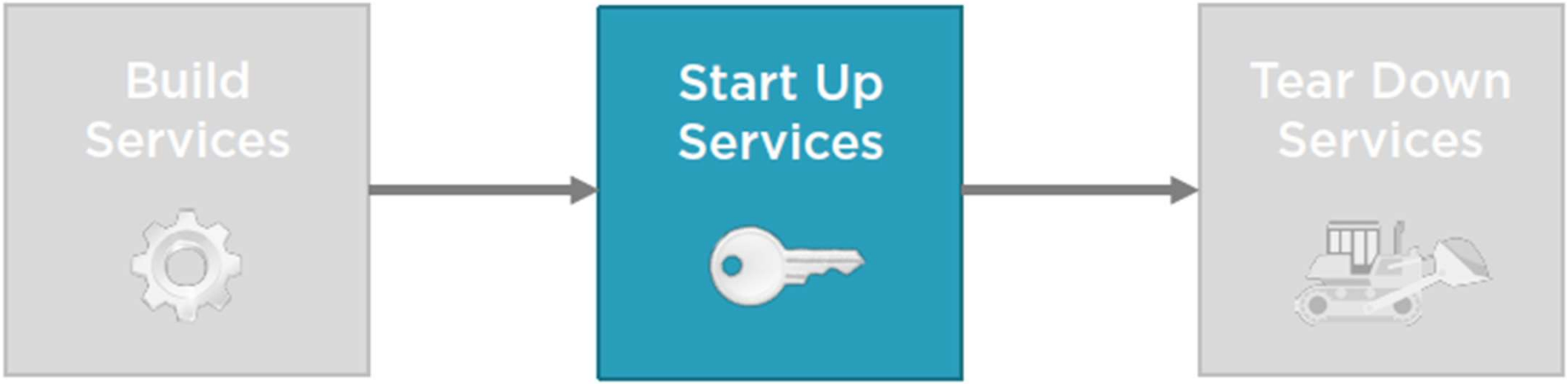

```
docker-compose build
```

Build or rebuild services defined in docker-compose.yml

Building Specific Services

```
docker-compose build mongo
```

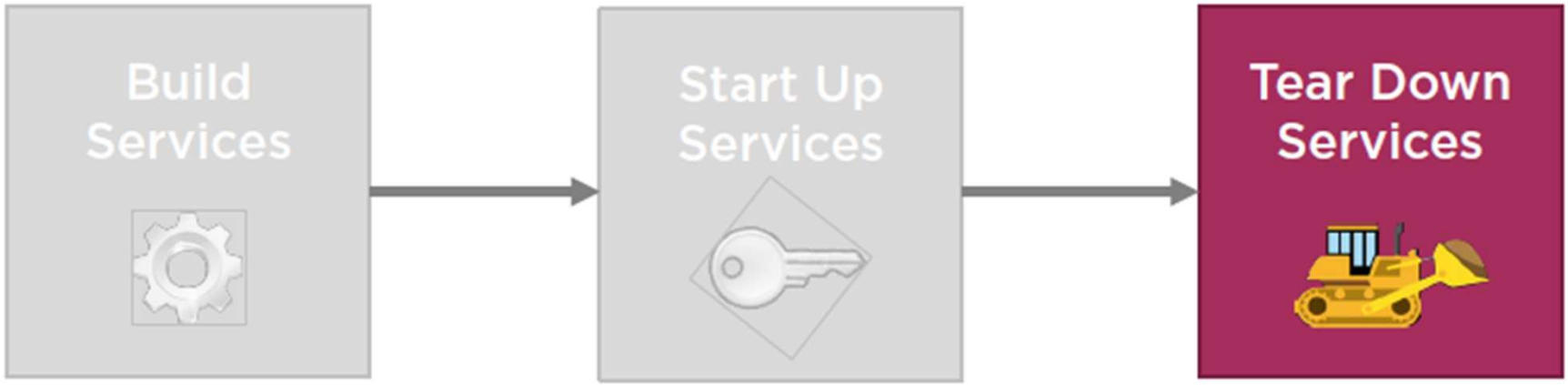
Starting Services Up




```
docker-compose up
```

Create and start the containers

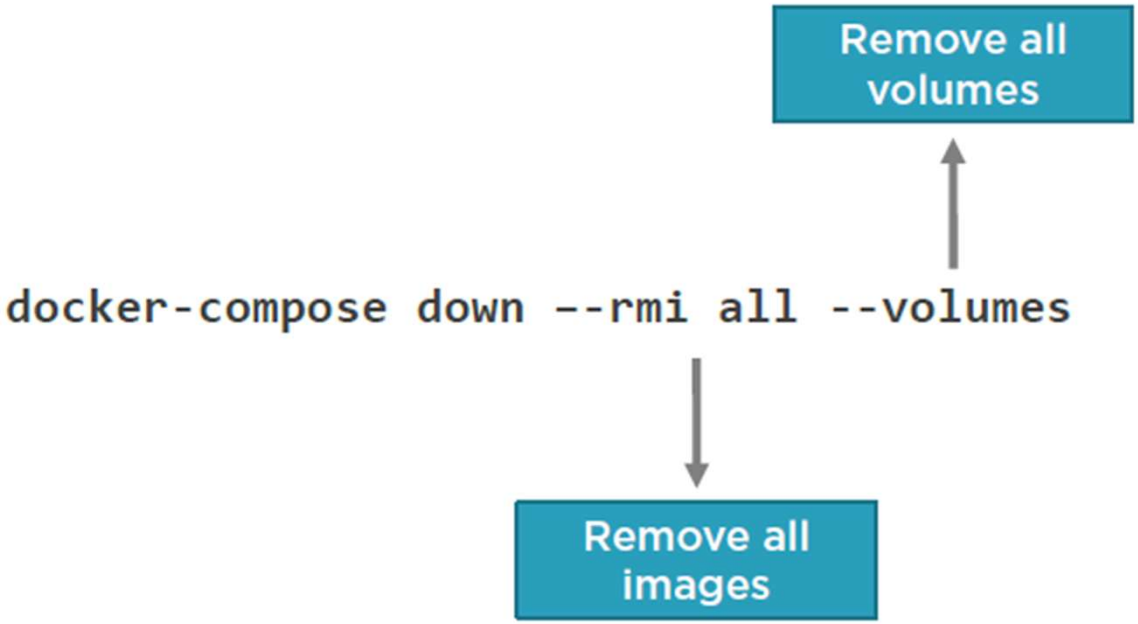
Tearing Down Services




```
docker-compose down
```

Take all of the containers down (stop and remove)

Stop and Remove Containers, Images, Volumes



Docker Use Cases

- Development Environment
- Environments for Integration Tests
- Quick evaluation of software
- Microservices
- Multi-Tenancy
- Unified execution environment
  - dev -> test -> prod (local, VM, cloud, ...)