The University Of Queensland
AUSTRALIA
CREATE CHANGE

Cloud Computing (INFS3208)

Lecture 3: Docker I: Container, Docker Commands, and Dockerfile

Lecturer: Dr Heming Du

School of Electrical Engineering and Computer Science

Faculty of Engineering, Architecture and Information Technology

The University of Queensland

Re-cap Lecture 2

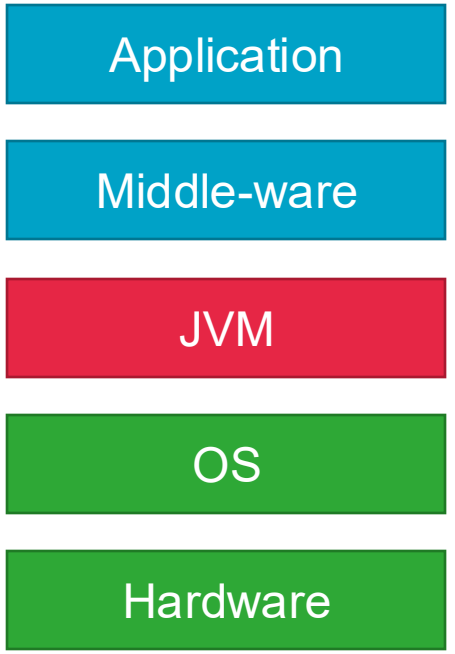
- Networking and Virtual Private Cloud
- Load Balancing
  - What & Why Load Balancing
  - Algorithms
  - LB in Cloud Architecture
  - LB in Distributed Systems
  - LB in Network Communications
  - LB in Cloud Product

Outline

- Container
- What is Docker
- Basic concepts in Docker
  - Images
  - Layer architecture
  - Containers
  - Registry
- Docker Commands
- Containerisation and Dockerfile
  - Dockerfile instructions

Revisit: Virtual Machines (VMs)

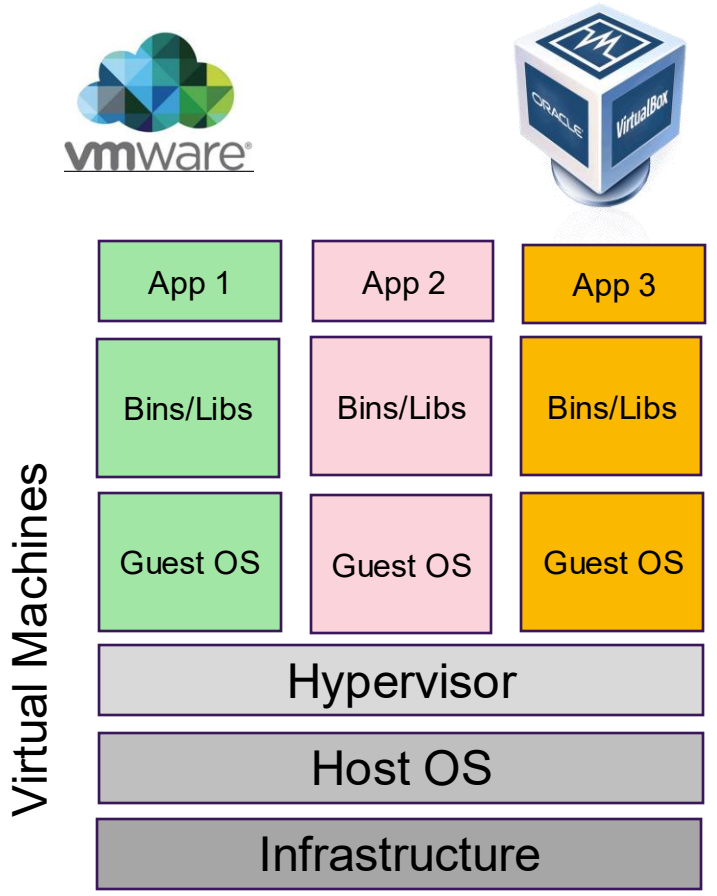
- A virtual machine (VM) is an **emulation of a computer** system.
- Virtual machines are based on **computer architectures** and provide functionality of a physical computer: software, specialised hardware, or a combination.
- **System virtual machines**: provide a substitute for a real machine.
- **Process virtual machines**
  - are designed to execute computer programs in a platform-independent environment.
  - provide a high-level abstraction (a high-level programming language abstraction)
  - Example: Java virtual machine (JVM)



https://blog.netapp.com/blogs/containers-vs-vms/
https://en.wikipedia.org/wiki/Virtual_machine

Virtual Machines (VM)

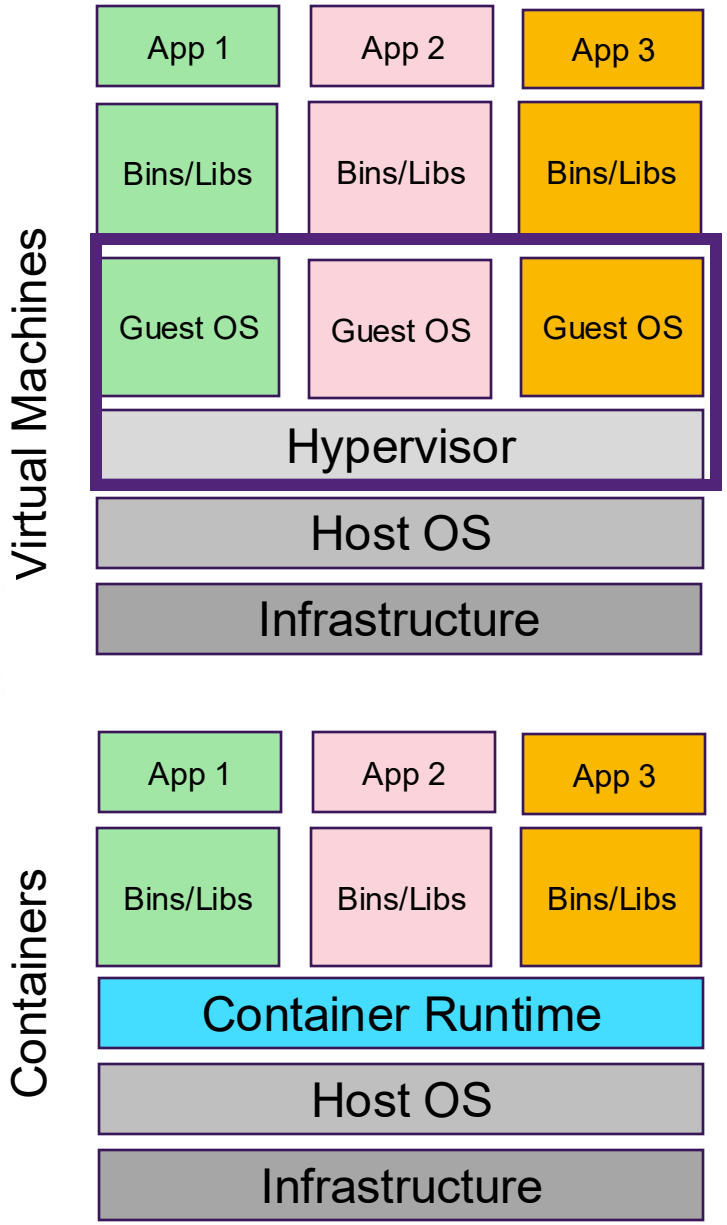
- VM could simulate almost anything:
  - Hardware (vCPU, vRAM, vHDisk, vGPU, vNetwork, etc.)
  - Software (Operating System: Ubuntu, CentOS, Windows Server)
- Environment **Isolation** and **Consistency**:
  - Isolation: VMs are **isolated** envs and **independent** to the host system (No affect).
  - Consistency: dev can create a **consistent** env. (Code MUST behaves consistently).
- But…
  - Each VM requires its own dedicated OS; Extra hardware and software overhead
  - Performance of VM is not very satisfied (slow to boot up)
  - Migration between physical servers is not always smooth.



Can we boot up more VMs for scalability?

Containers

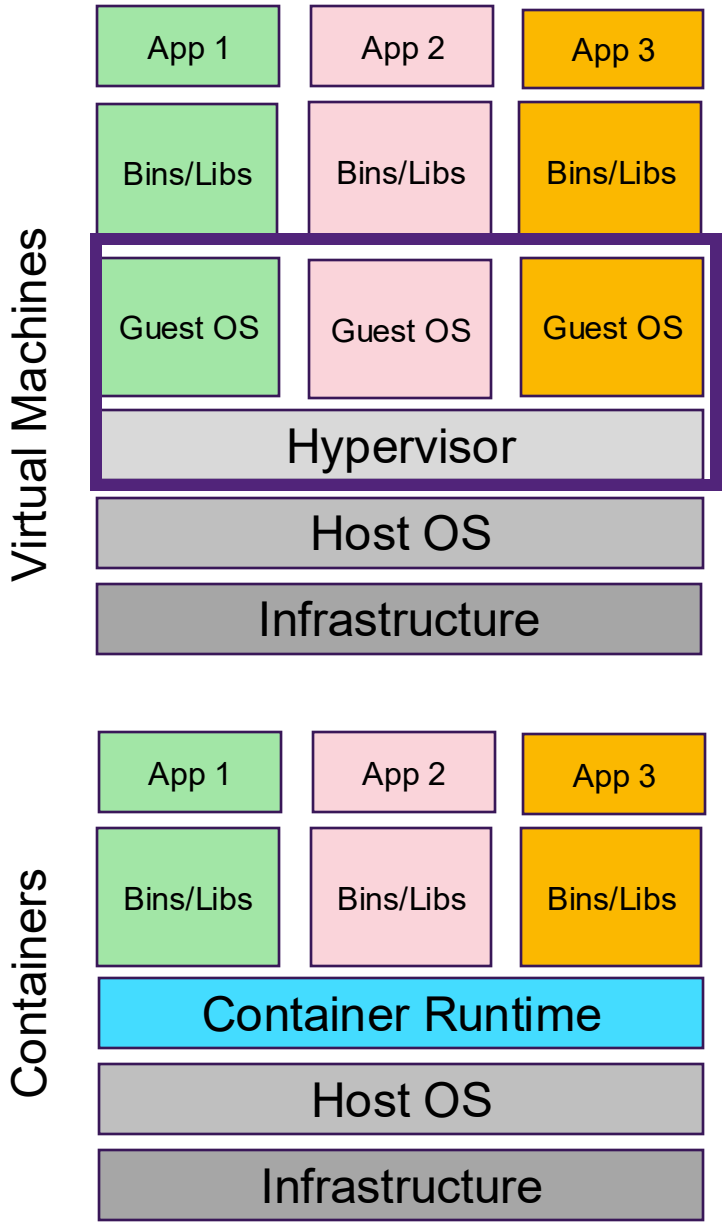
- **Containers** offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run.
- Completely isolated environments: services, network interfaces, mounts
- Multiple containers running atop the **OS kernel** directly – far more lightweight:
  - share the OS kernel
  - start much faster (in seconds)
  - use a fraction of the memory (compared to booting an entire OS)
  - one physical machine/server can run much more containers than VMs
- Running Linux Containers on Windows (not easy but possible)
  - Windows Subsystem for Linux (WSL)
  - Docker Desktop (helps you build, share, and run containers on Mac and Windows as you do on Linux)



https://cloud.google.com/containers/
https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/linux-containers

The History of Containers

- Before container:
  - Chroot system call in 1979 and FreeBSD jails in 2000.
  - Cgroups (2006, 2007) and namespaces (2002)
  - Linux Containers (LXC) in 2008
- Containers virtualise at the operating system level
- Containerisation provides a clean separation of concerns:
  - developers focus on their application logic and dependencies
  - IT operations teams focus on deployment and management without bothering with application details (e.g. specific software versions and apps configurations).

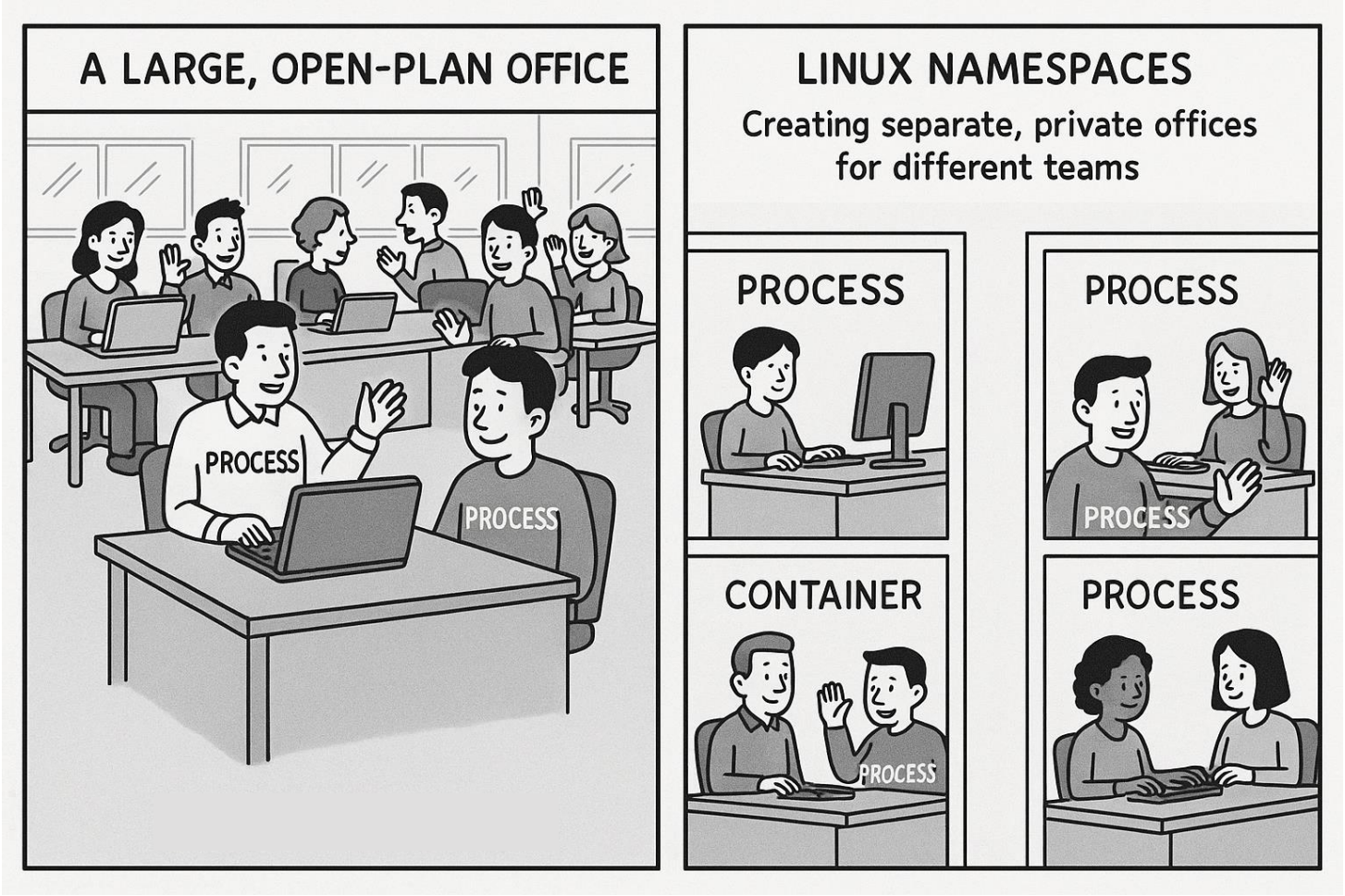


https://www.redhat.com/en/blog/history-containers
https://cloud.google.com/containers/

Linux Namespaces: Isolation of System Resources

- Namespaces are a feature of the Linux kernel that partition kernel resources such that one set of processes sees one set of resources, while another set of processes sees a different set.

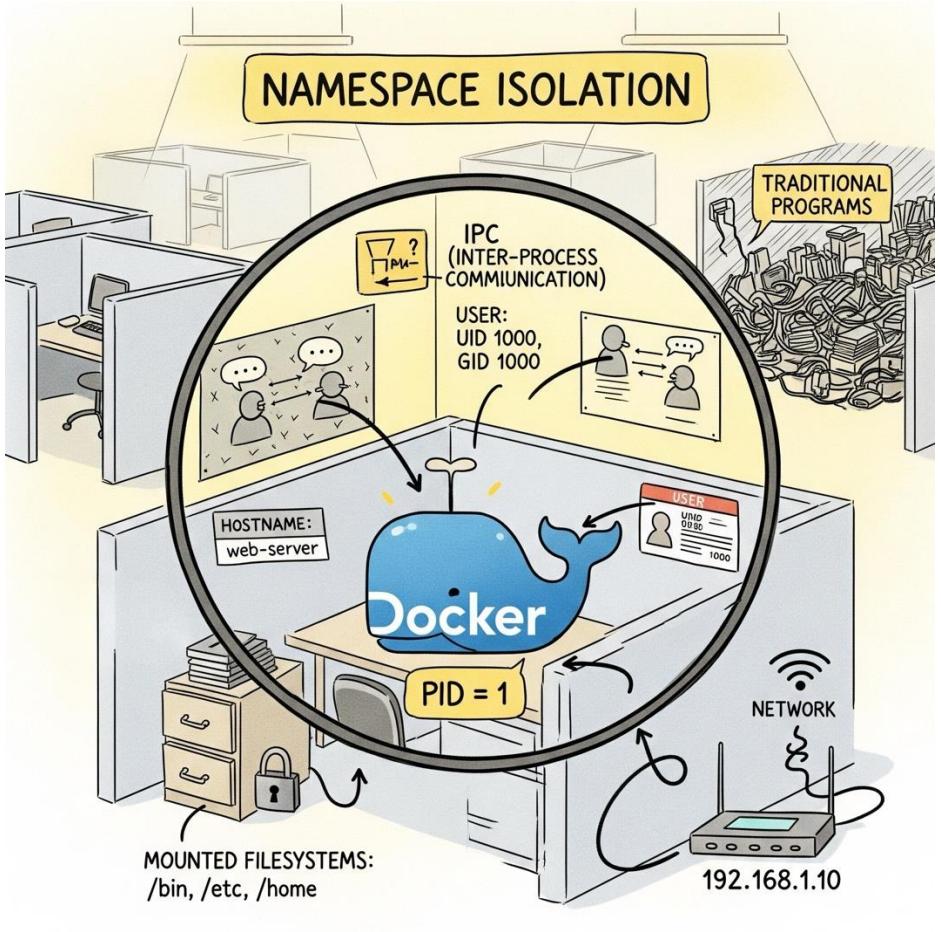
- Namespaces provides a process with its own isolated view of the system.



Linux Namespaces: Isolation of System Resources

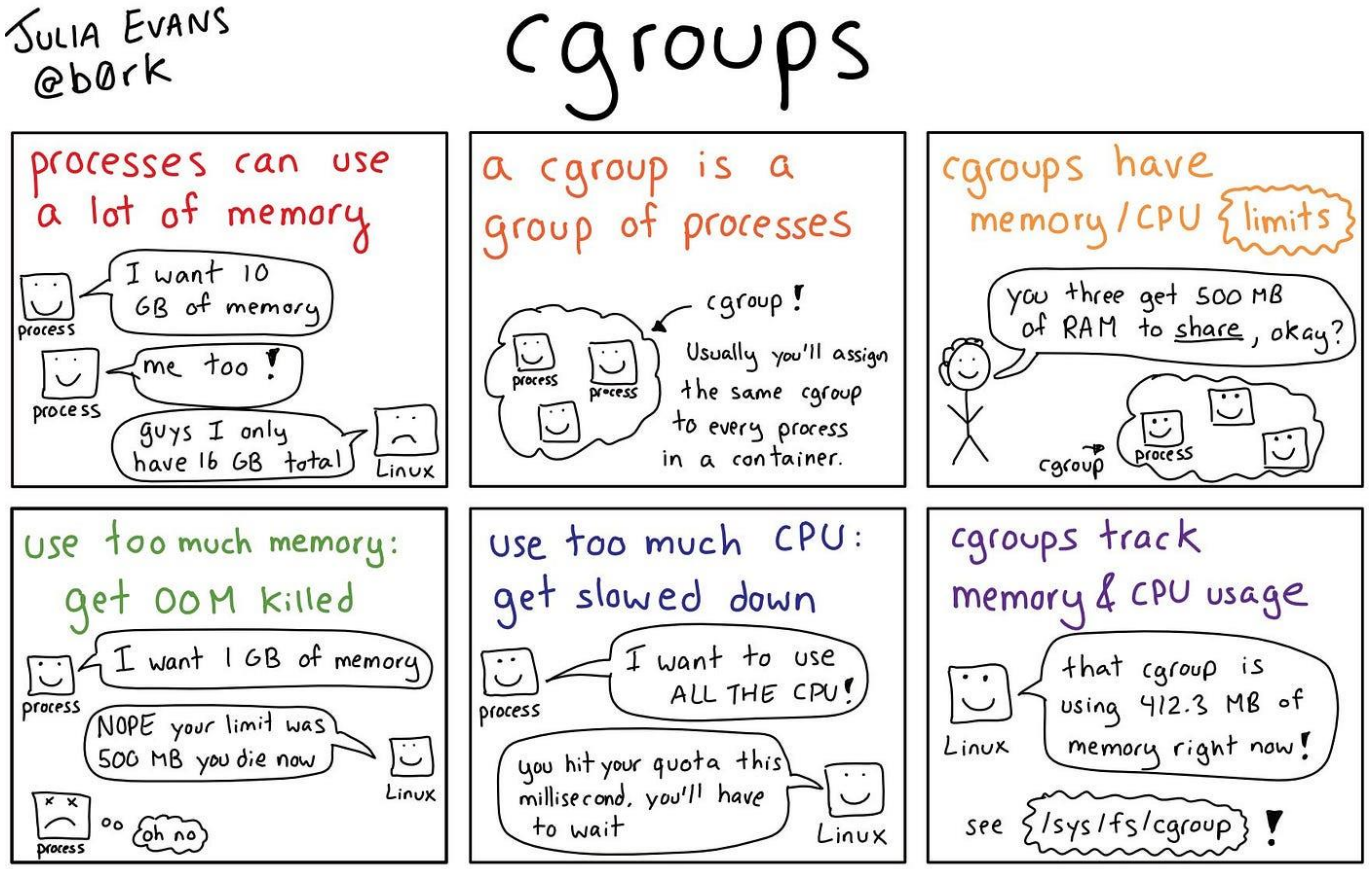
Namespaces are utilized to give a container the illusion that it's running on its own, separate machine. Key namespaces used by containers include:

- PID (Process ID): Isolates the process tree. Inside a container, the main process has a PID of 1, but on the host machine, it has a different PID.
- NET (Network): Provides each container with its own network stack (IP address, routing tables, etc.).
- MNT (Mount): Gives each container its own filesystem.
- UTS (UNIX Time-sharing System): Allows each container to have its own hostname.
- IPC (Inter-Process Communication): Isolates IPC resources.
- USER: Isolates user and group IDs.



Managing Resources with Control Groups (Cgroups)

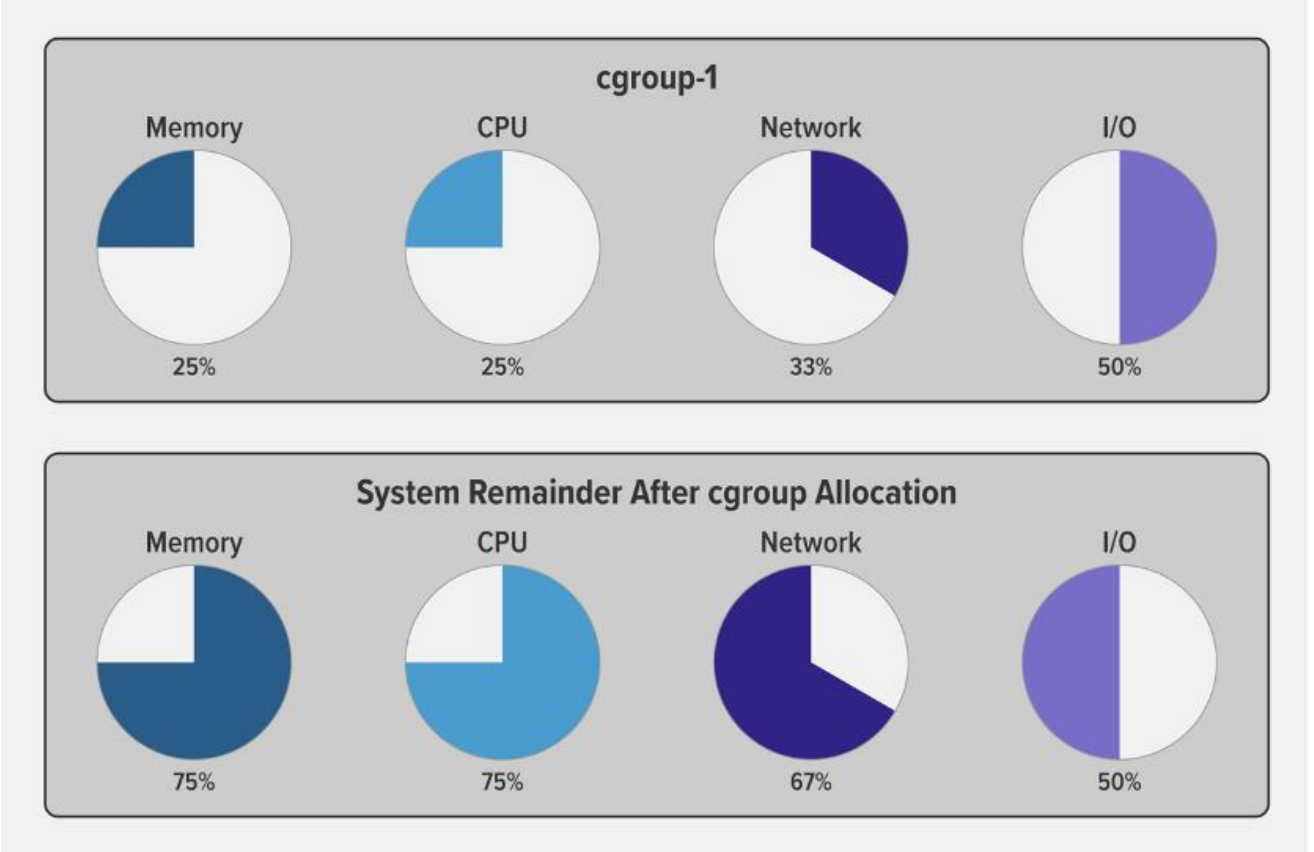
- Control Groups (cgroups) are a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.
- While namespaces handle what a container can see, cgroups control what a container can use.
- Cgroups prevent a single container from consuming all of the host's resources and impacting other containers or the host system itself.



https://medium.com/@boutnaru/linux-cgroups-control-groups-part-1-358c636ffde0

Managing Resources with Control Groups (Cgroups)

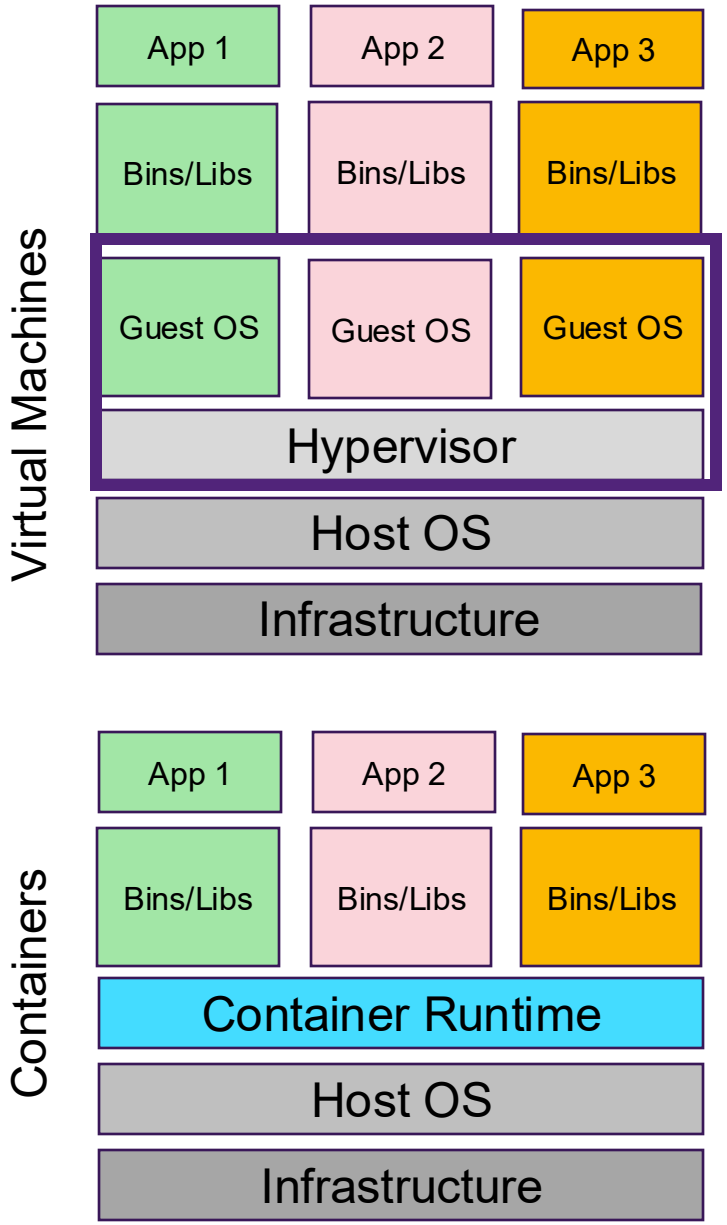
- Resource Limiting: You can set hard limits on the amount of memory or CPU a container can use.
- Prioritization: You can allocate more or fewer resources to certain containers.
- Accounting: Cgroups monitor and report on the resource usage of containers.
- Control: You can freeze and resume all processes within a cgroup.



https://blog.nginx.org/blog/what-are-namespaces-cgroups-how-do-they-work

The History of Containers

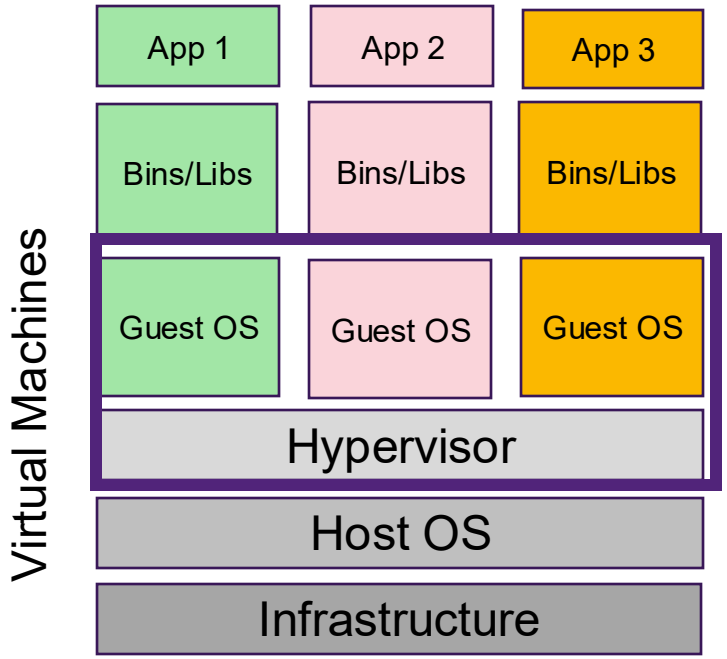
- Before container:
  - Chroot system call in 1979 and FreeBSD jails in 2000.
  - Cgroups (2006, 2007) and namespaces (2002)
  - Linux Containers (LXC) in 2008
- Containers virtualise at the operating system level
- Containerisation provides a clean separation of concerns:
  - developers focus on their application logic and dependencies
  - IT operations teams focus on deployment and management without bothering with application details (e.g. specific software versions and apps configurations).

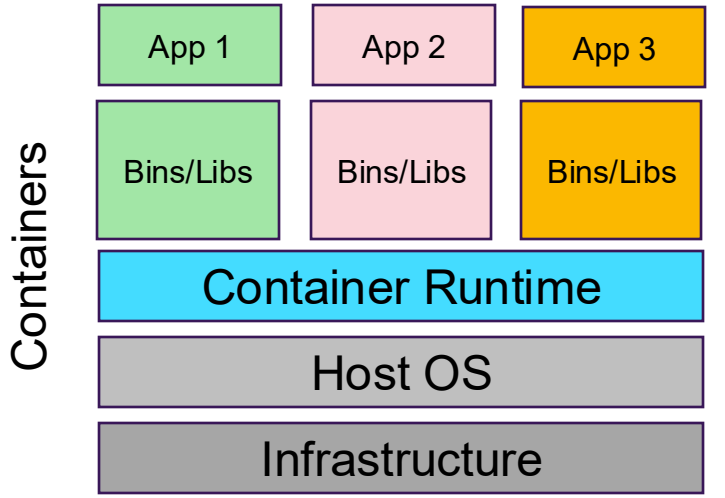


https://www.redhat.com/en/blog/history-containers
https://cloud.google.com/containers/

Why Containers?

- **Run Anywhere:**
  - Containers are able to run virtually anywhere, greatly easing development and deployment:
    - on Linux, Windows, and Mac operating systems;
    - on virtual machines;
    - on a developer's machine;
    - in data centres on-premises;
    - in the public cloud.
- **Isolation:**
  - Containers virtualise hardware at the OS-level:
    - CPU,
    - memory, storage,
    - network resources.
  - Containers provide developers with a sandboxed view of the OS logically isolated from other applications.





https://cloud.google.com/containers/

Containers vs. Virtual Machines

Features	Containers	Virtual Machines
Isolation	**Process-level** isolation using namespaces and cgroups	**Full OS-level** isolation with a separate guest OS
Resource Efficiency	**High**, due to sharing of the host OS kernel	**Lower**, as each VM runs a full OS
Startup Time	**Very fast** (seconds or less)	**Slower** (minutes)
Performance	**Near-native** performance	**Overhead** from hypervisor and full OS
Storage	Uses **layered file systems** (e.g., overlayfs)	Requires dedicated disk space for each VM
Deployment	**Lightweight** and quick to deploy	**Heavier** and slower to deploy
Scalability	**Easily scalable**	**Less flexible**, more resource-intensive to scale
Portability	**High portability** (consistent environments)	**Moderate portability** (more dependencies)
Security	**Less secure** than VMs (shared kernel vulnerabilities)	**More secure** (isolated OS kernel)
Use Cases	Microservices, DevOps, CI/CD, lightweight apps	Legacy applications, full OS environments, high-security isolation
Management Tools	Docker, Kubernetes	VMware, Hyper-V, VirtualBox
Overhead	**Minimal** overhead (shares host kernel)	**Significant** overhead (runs separate OS for each VM)

Outline

- Container
- What is Docker
- Basic concepts in Docker
  - Images
  - Layer architecture
  - Containers
  - Registry
- Docker Commands
- Containerisation and Dockerfile
  - Dokcerfile instructions

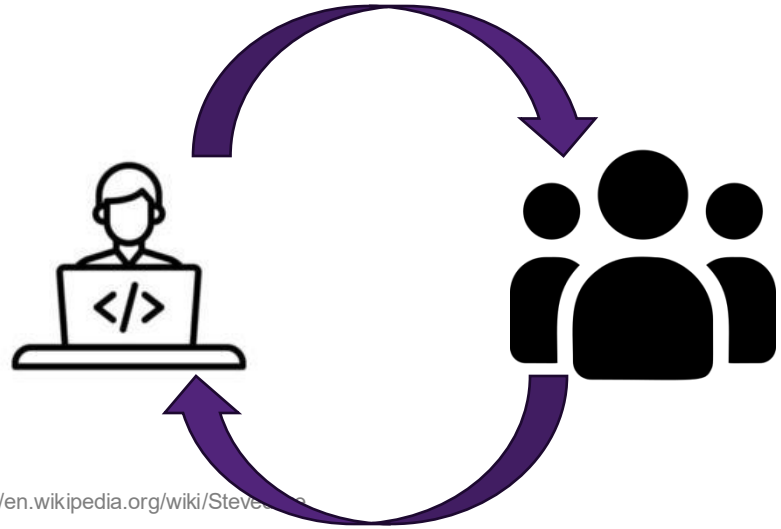
What is a “Docker”?

stevedore (/ˈstiːvɪˌdɔːr/), **longshoreman**, **docker**, or **dockworker**

https://en.wikipedia.org/wiki/Stevedore

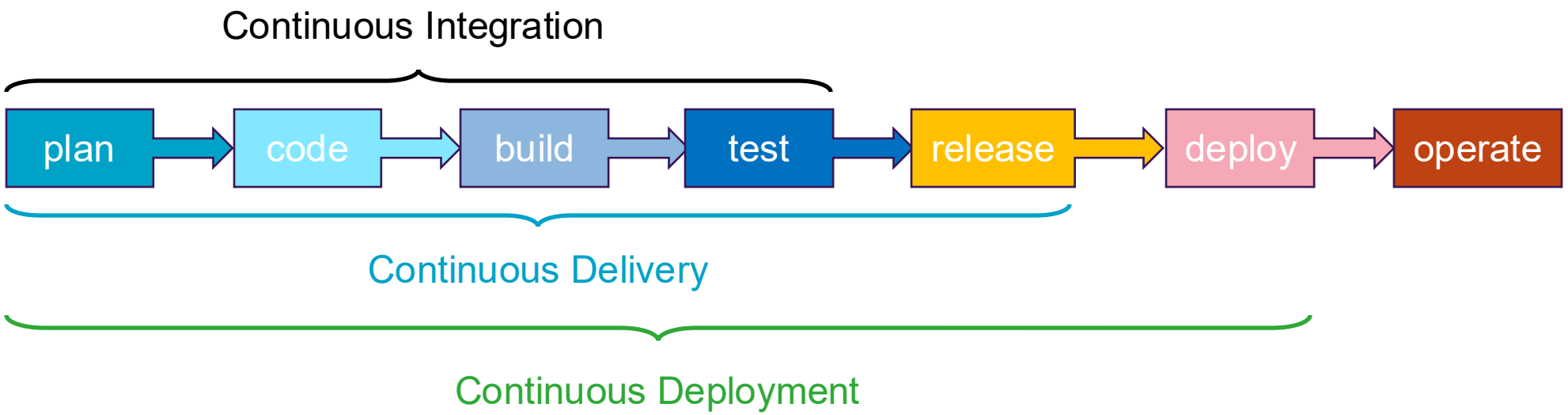
What is a “Docker”

- Cargo can be quickly packed at source of production
- Cargo can be quickly shipped from A to B
- Cargo can be quickly unpacked at destination
- Cargo is what it is after shipping



https://en.wikipedia.org/wiki/Steve

Modern Development Practices - DevOps



- Continuous Integration (CI): automated testing
- Continuous Delivery (CD): automated release
- Continuous Deployment: fully automated

Needs Isolation & Consistency

Lightweight

What is Docker

- Docker is
  - a job (literally),
  - a company providing solutions to containerisation (Google's LXD),
  - a new implementation of container technologies.
- Some definitions:
  - Docker is a container image that is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings.
  - Docker is an open platform for developing, shipping, and running applications (official documentation). Simply, Docker is a tool deploy applications in a sandbox (called containers) to run on the host operating system i.e. Linux, Windows, Mac OSX, etc.

https://thenewstack.io/solomon-hykes-departs-from-docker/

Docker founders Kamel Founadi, Solomon Hykes, Sebastien Pahl

https://www.docker.com/what-container

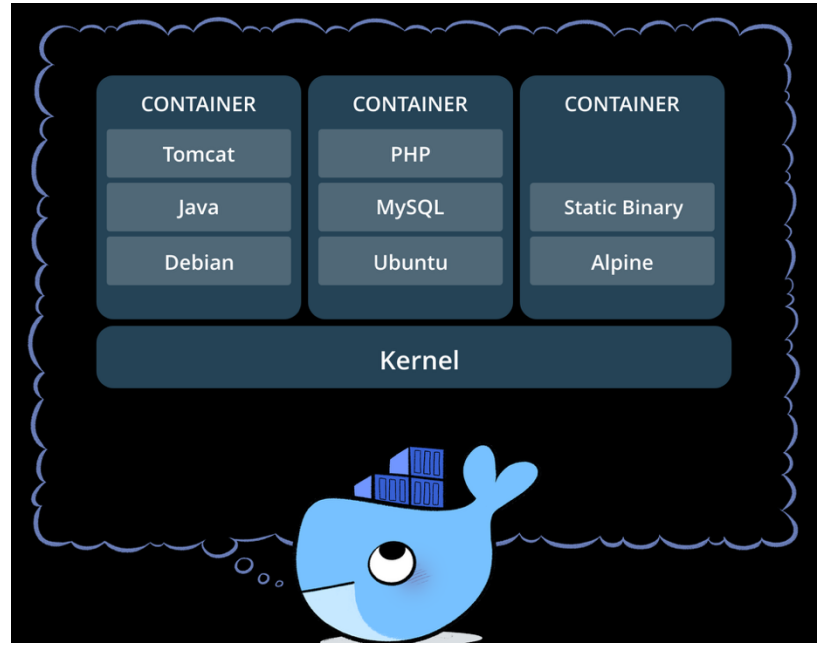
https://en.wikipedia.org/wiki/Docker_(software)

Outline

- Container
- What is Docker
- Basic concepts in Docker
  - Images
  - Containers
  - Registry
  - Layer architecture
- Docker Commands
- Containerisation and Dockerfile
  - Dokcerfile instructions

Basic Concepts - Docker Image

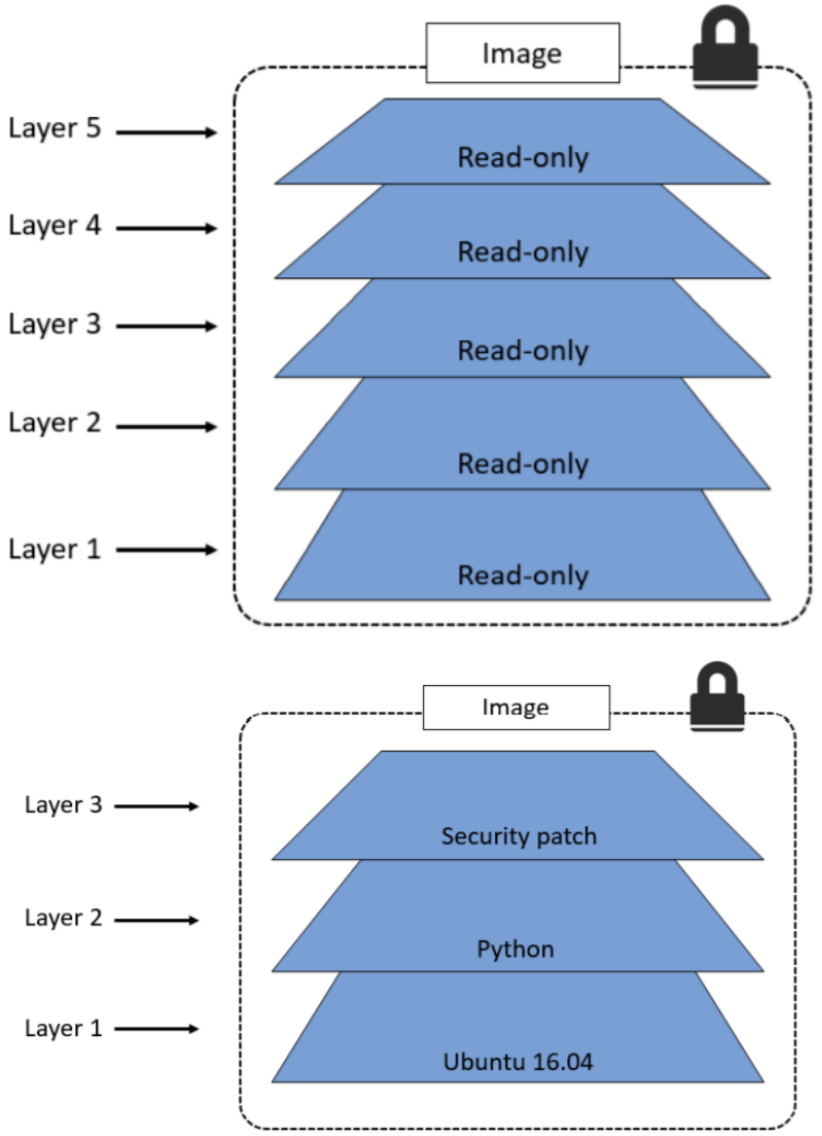
- Docker image is an object that contains
  - an OS filesystem
  - and one or more applications
- An image is a template that is readable only to build a container.
- An image is based on another image (base image) with some additional customisation
  - Example: an image that contains ubuntu 22.04 (LTS) operating system and apache web server
- Docker provides a very good mechanism to build images and can even pull the existing images.
- There are heaps of resources on Docker Hub.
- Many software is released as Docker images, instead of software packages.



https://cloud.google.com/containers/

Images and Layers

- With Union FS (Union Filesystems), Docker is designed as a layer-wise architecture.
- Union FS: create a union of (1) a list of lower directories – read only (2) upper directory – reads and writes
- Each image will not be changed (read-only) after it has been constructed – lower directories.
- Layer-wise architecture makes reuse and customization of images much easier.
  - Construct a new image based on the existing images by adding new layers on top of the current layers.
  - E.g. Ubuntu + Python + security patch

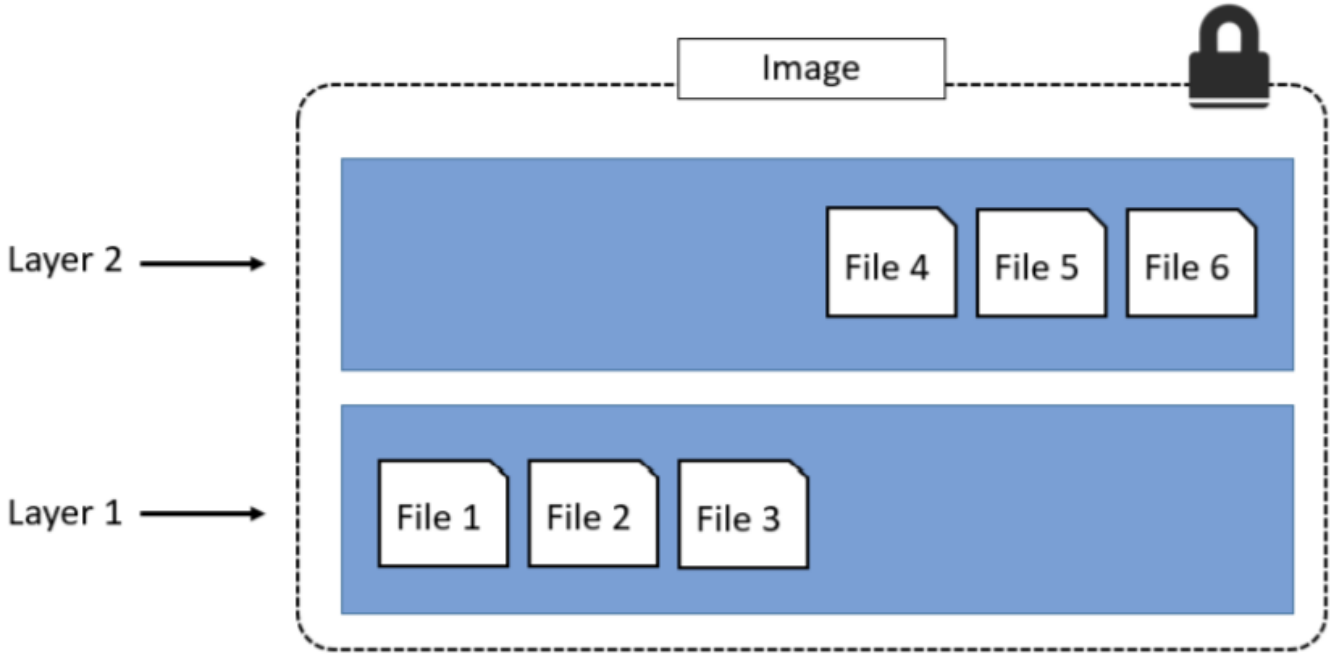


https://docs.docker.com/storage/storagedriver/

https://blog.netapp.com/blogs/containers-vs-vms/

Copy-on-write Strategy: Example of Updating Files

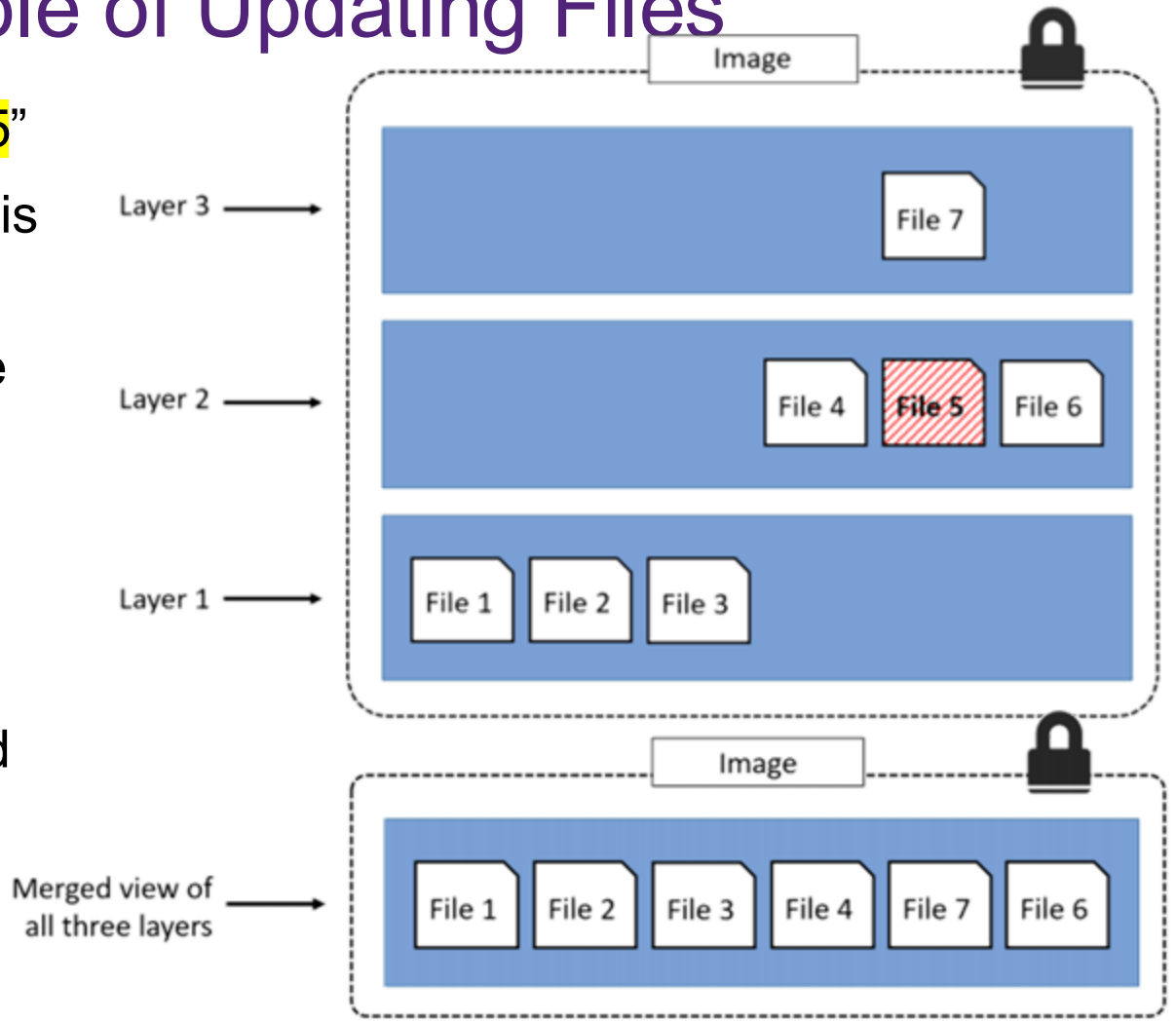
- Copy-on-write is a strategy of sharing and copying files for maximum efficiency.
- Example: after constructing two layers, three files in Layer 1 and three files in Layer 2.
- What will happen if we want to update “File 5” in Layer 2



https://docs.docker.com/storage/storagedriver/#the-copy-on-write-cow-strategy

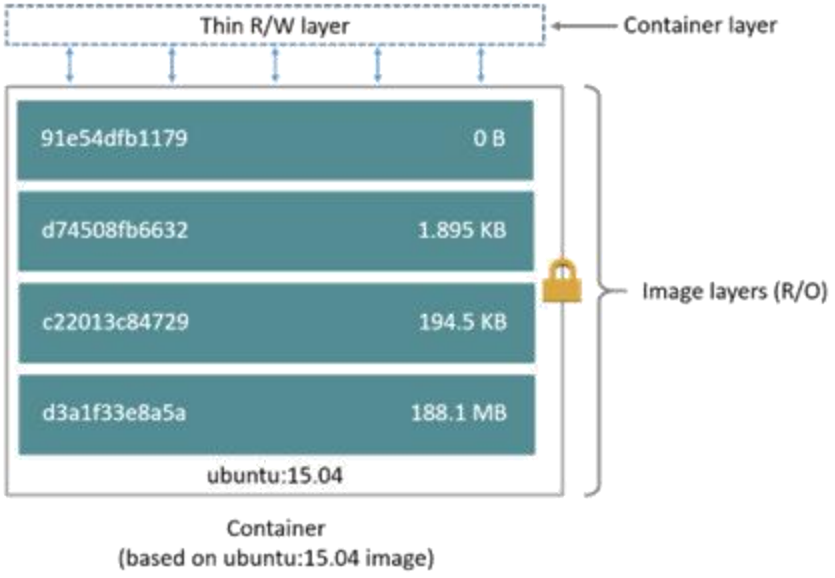
Copy-on-write Strategy: Example of Updating Files

- "File 7" in layer 3 is an updated version of "File 5"
- Due to read-only property of each layer, "File 5" is still stored in the image.
- "File 5" is updated as a new layer added into the image.
- Good practice:
  - only add required files or apps in the current layer
  - Temporary or redundant files must be removed from the current layer before constructing the next layer
  - Very important when writing dockerfiles.



Basic Concepts - Container

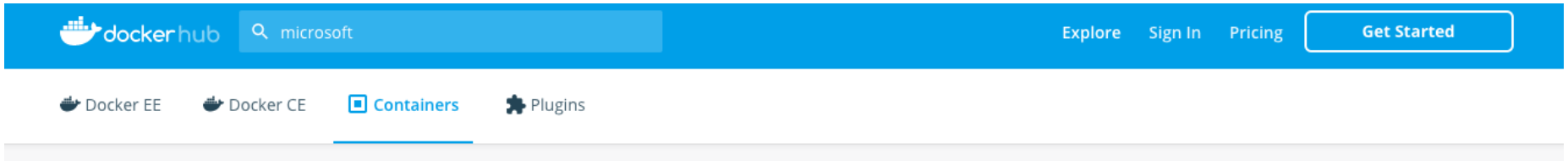
- Container: a running instance of an image
  - Class vs Instances (in Objective Oriented Programming)
  - E.g. House Blueprints vs House
- Container is a running process
  - Independent namespace
    - *E.g. variable $JAVA_HOME in Container A is different from $JAVE_HOME in Container B*
  - Independent root filesystem
    - *E.g. /usr/local in Container A is different from /user/local in Container B*
  - Independent networks
    - *E.g. 192.168.0.1 in Container A is different from 192.168.0.1 in Container B*
  - Independent user space
    - *E.g. user "John" in Container A is different from "John" in Container B, even user root is different.*



https://stackoverflow.com/questions/2885385/what-is-the-difference-between-an-instance-and-an-object

Basic Concepts - Docker Registry

- After releases of docker images, a registry is needed to store and manage images for sharing.
- Docker images are stored in a Docker registry:
  - **Public Docker registry**: Docker Hub (available to everyone)
  - **Private Docker registry:** company's hub (only available to the employees)
- Each Docker registry can have multiple repositories and each repository can have multiple tags. Each tag corresponds to an image:
  - <repository_name>:<tag>
  - Example: for ubuntu images – ubuntu:20.04 vs ubuntu:22.04
  - Tags can be ignored (ubuntu -> ubuntu:latest)
- Registry's name is normally represented by a path with two components (user/software)
  - Example: docker pull microsoft/cntk:2.7-cpu-python3.5

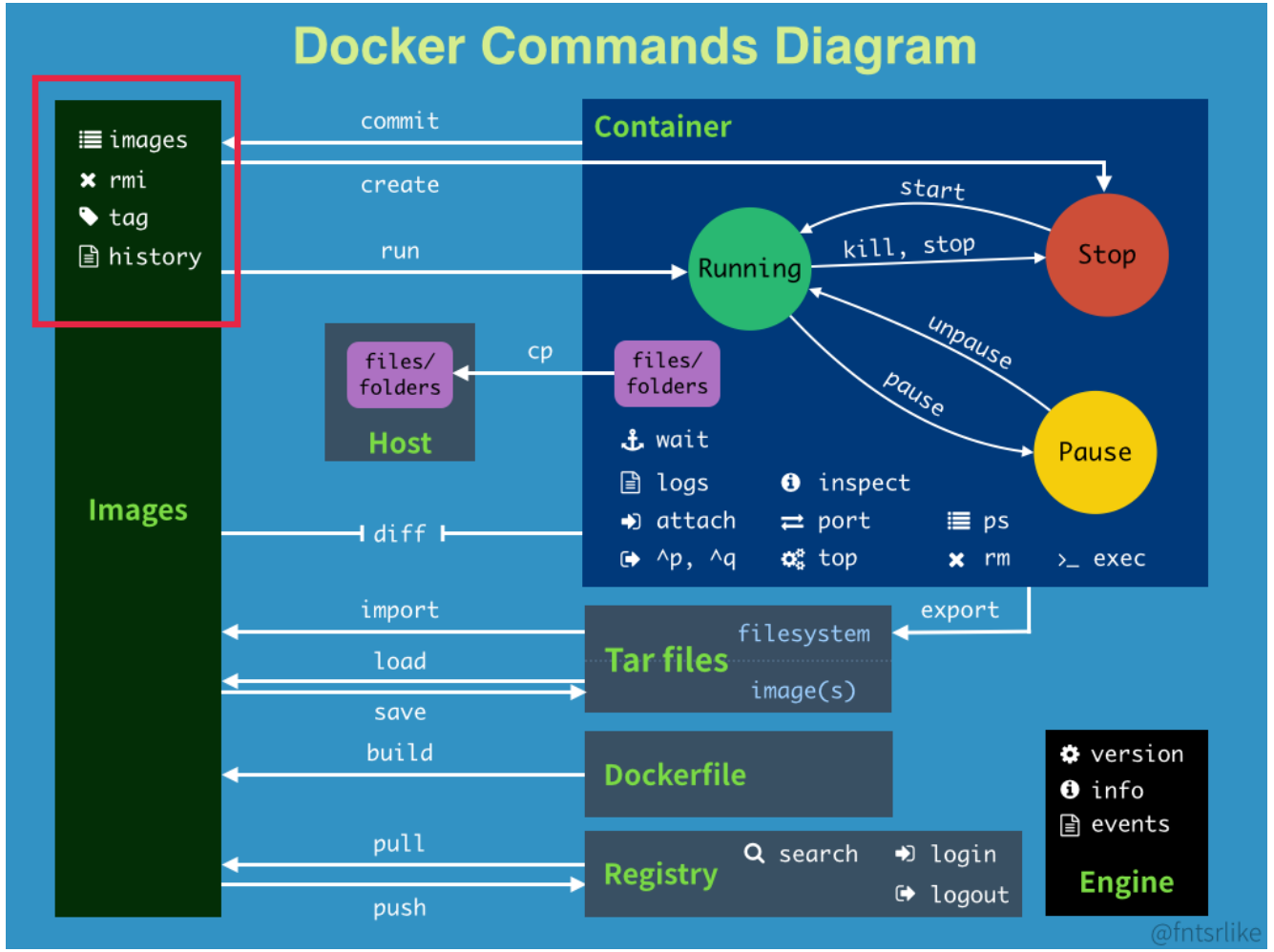


https://hub.docker.com/
https://cloud.google.com/container-registry/

Outline

- Container
- What is Docker
- Basic concepts in Docker
  - Images
  - Layer architecture
  - Containers
  - Registry
- Docker Commands
- Containerisation and Dockerfile
  - Dokcerfile instructions

Docker Command Diagram – Image management



- **List images**:

  docker images [OPTIONS] [REPOSITORY[:TAG]]
- **Remove one or more image**:

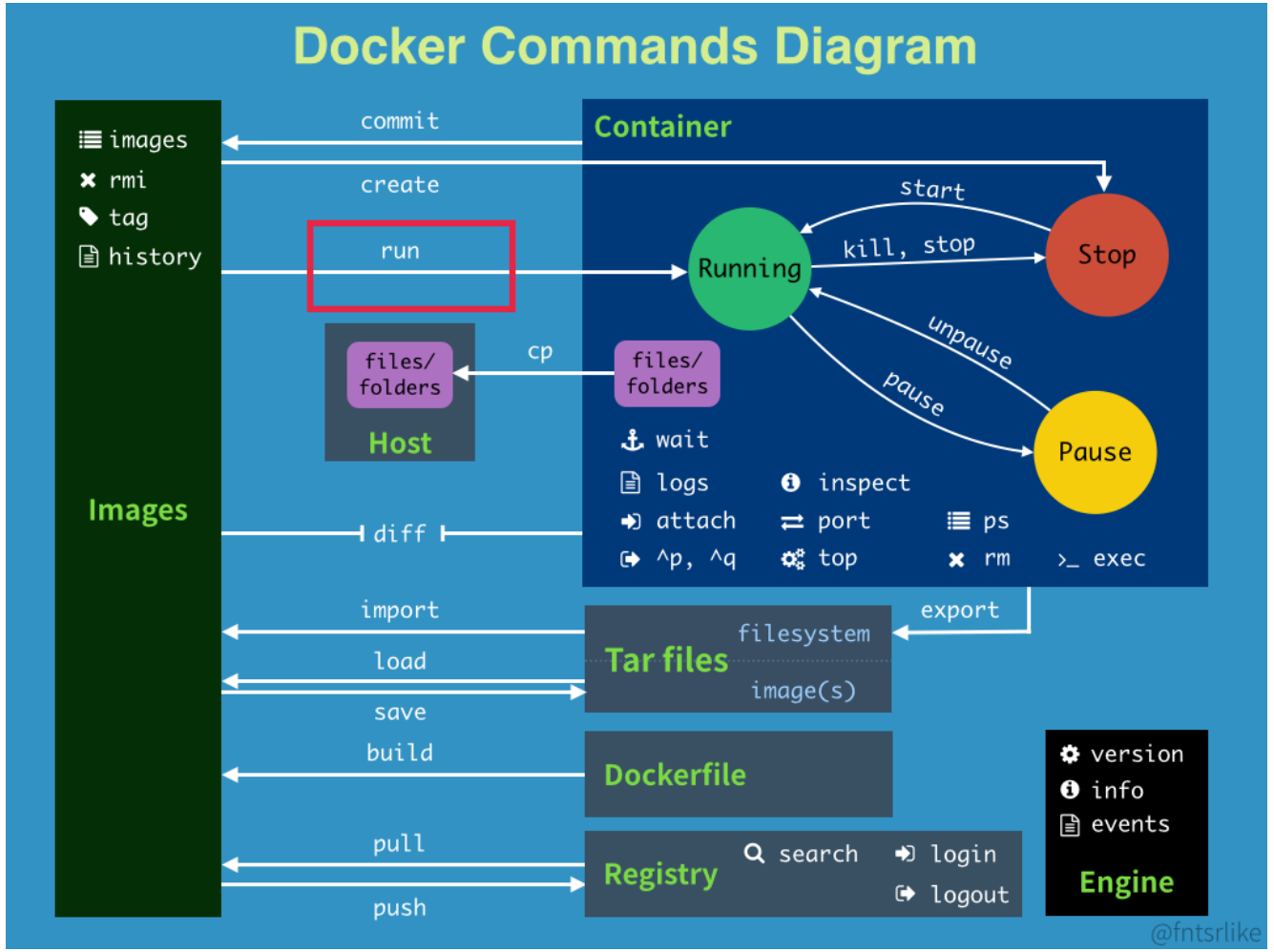
  docker rmi [OPTIONS] IMAGE [IMAGE...]
- **Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE**

  docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
- **Show the history of an image**

  docker history [OPTIONS] IMAGE

https://philipzheng.gitbook.io/docker_practice/appendix_command

Docker Command Diagram – Docker Run Images



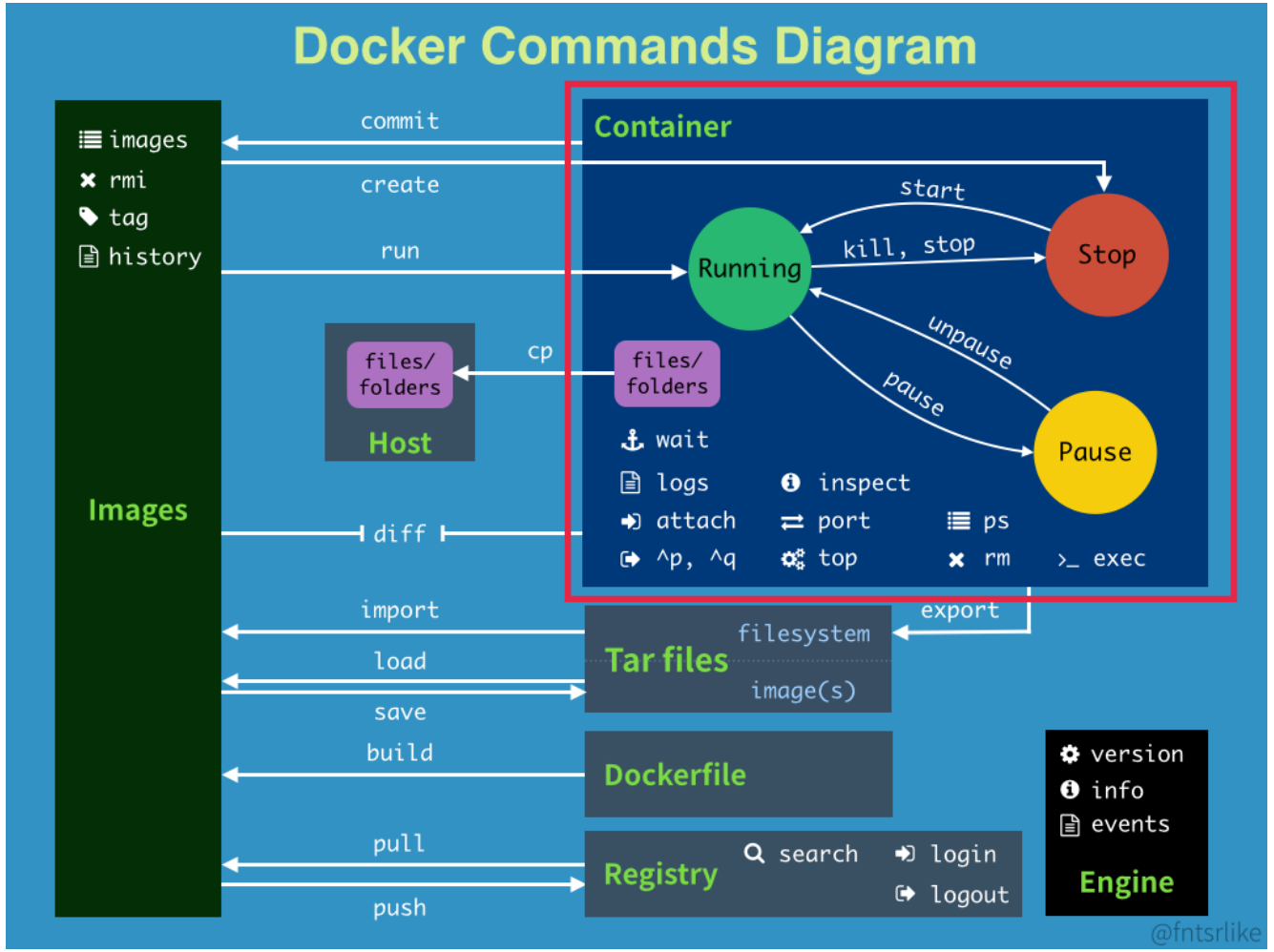
- **Run images**:

  docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]

  - Must specify an IMAGE to derive the container form
  - Foreground (-it) vs Detached (-d)
    - E.g. docker run -it alpine
    - E.g. docker run –d nginx
  - Name (--name)
    - E.g. docker run --name my-redis -d redis
  - EXPOSE (incoming ports –P or –p)
  - VOLUME (shared filesystems)
  - and more…

https://philipzheng.gitbook.io/docker_practice/appendix_command

Docker Command Diagram – Container management



- **Kill/Stop one or more running containers**:

  docker container kill/stop [OPTIONS] CONTAINER [CONTAINER...]

  Or

  docker kill/stop [OPTIONS] CONTAINER [CONTAINER...]

- **Start one or more stopped containers**:

  docker container start [OPTIONS] CONTAINER [CONTAINER...]

- **Pause/Unpause all processes within one or more containers**

  docker container pause/unpause [OPTIONS] CONTAINER [CONTAINER...]

- **Other container management commands**

  List containers: docker ps

  Attach containers: docker attach [OPTIONS] CONTAINER

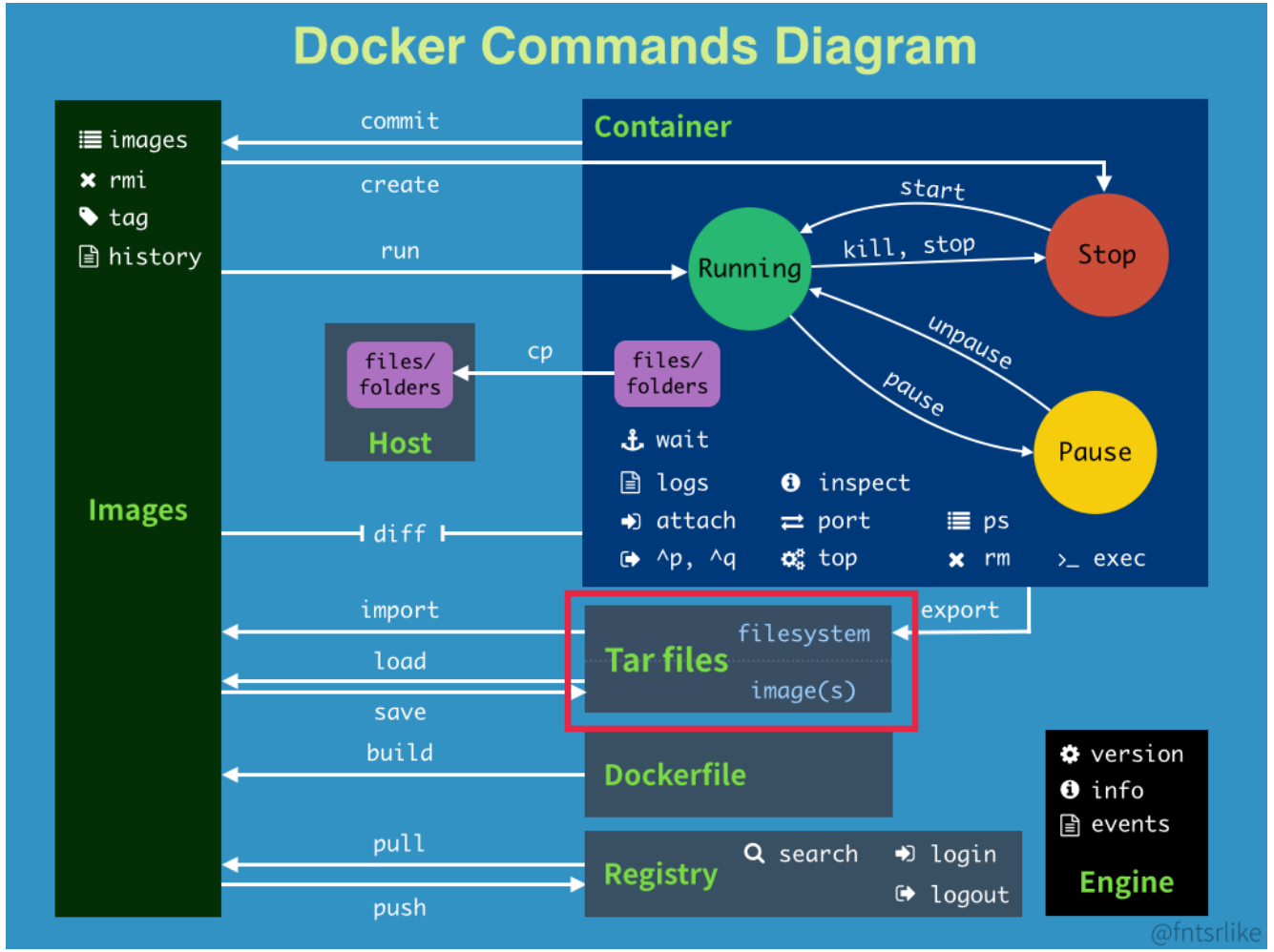
  Run a bash: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

  and more …

https://docs.docker.com/engine/reference/commandline/docker/

https://philipzheng.gitbook.io/docker_practice/appendix_command

Docker Command Diagram – Docker sharing



- **Export a container's filesystem as a tar archive**:

  docker export [OPTIONS] CONTAINER
- **Import the contents from a tarball to create a filesystem image:**

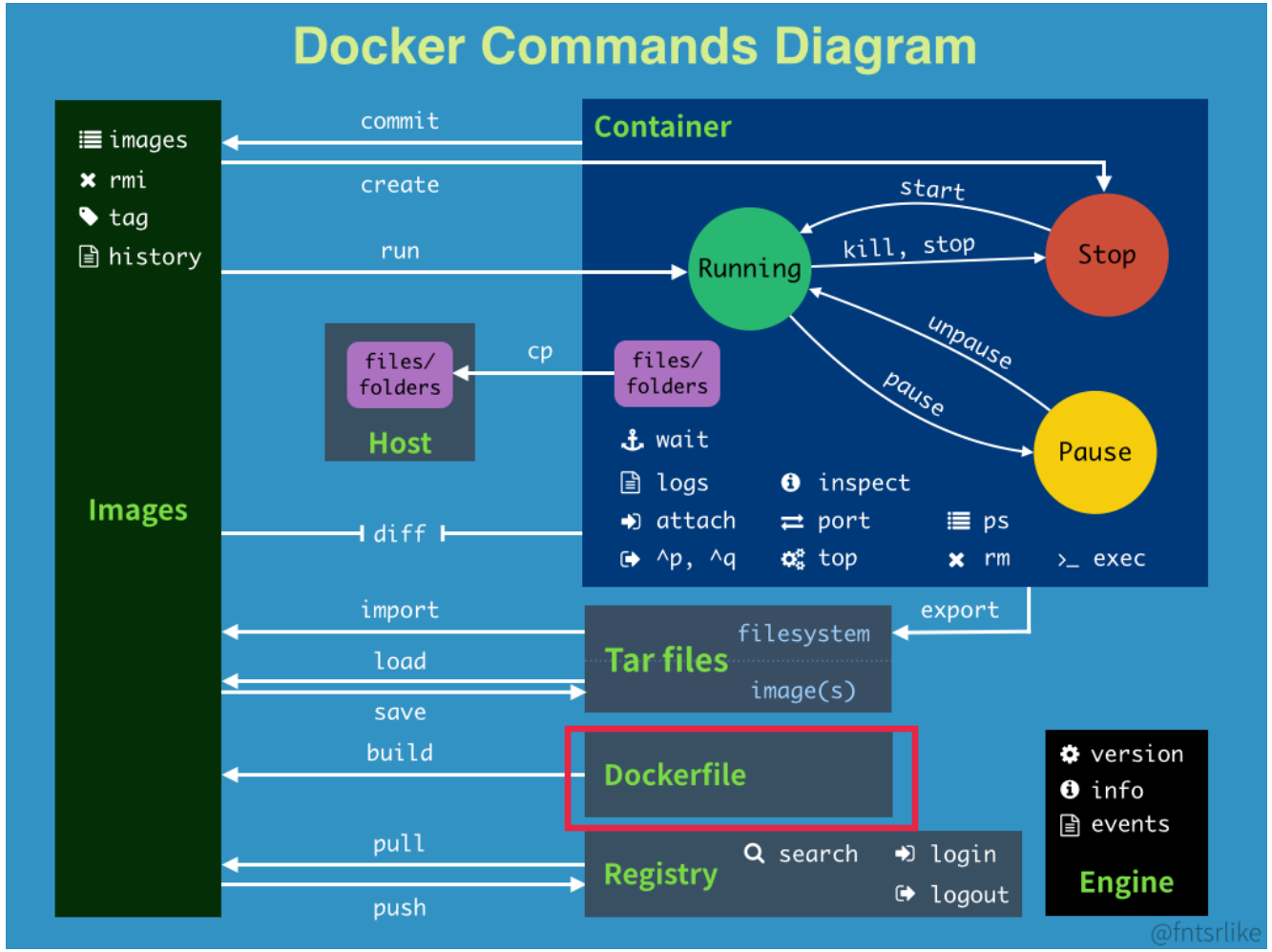
  docker import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]
- **Load an image from a tar archive or STDIN**:

  docker load [OPTIONS]
- **Save one or more images to a tar archive (streamed to STDOUT by default)**

  docker save [OPTIONS] IMAGE [IMAGE...]

https://philipzheng.gitbook.io/docker_practice/appendix_command

Docker Command Diagram – Build image from dockerfile

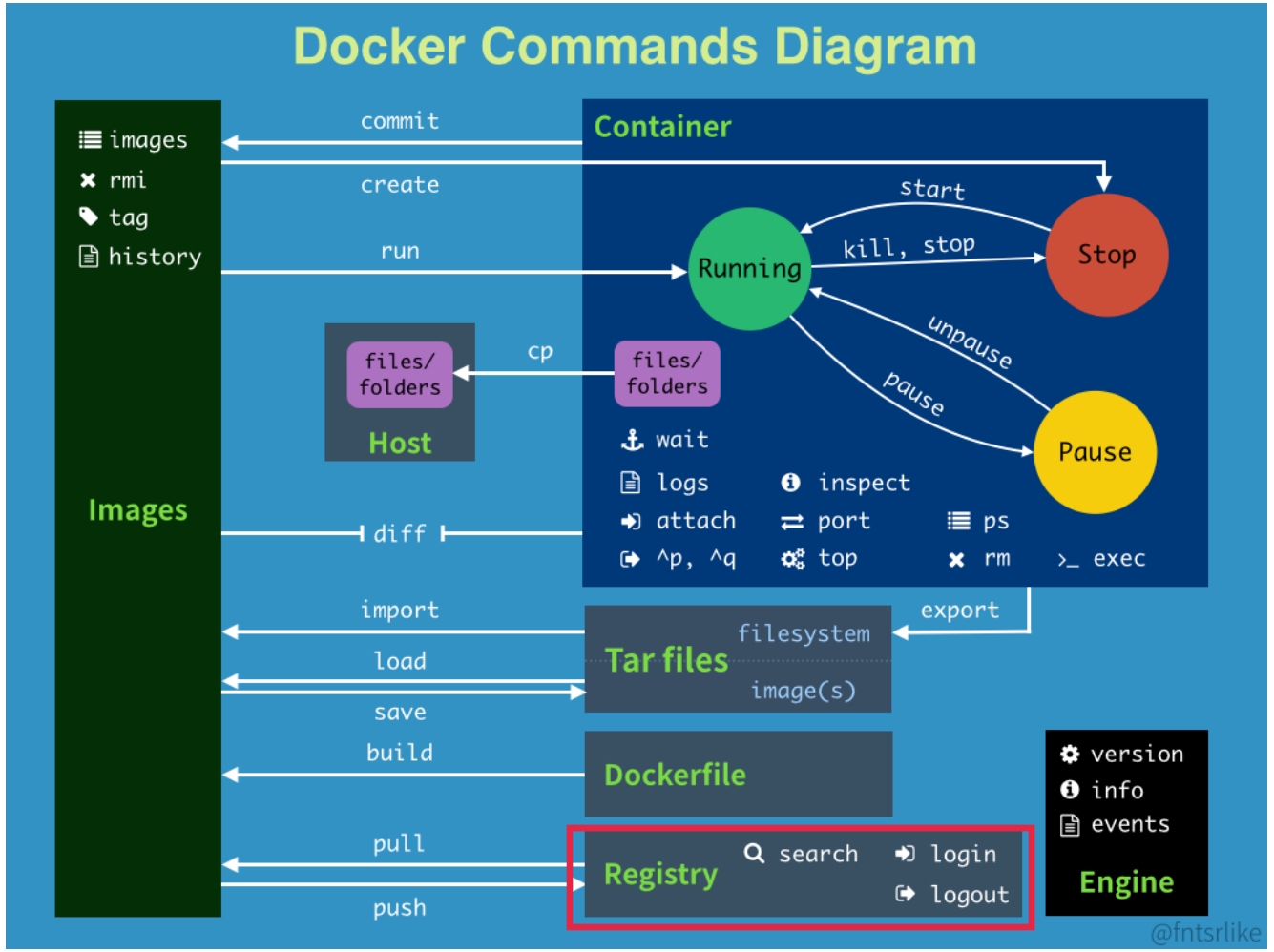


- **Build an image from a Dockerfile**:

  docker build [OPTIONS] PATH | URL | -

https://philipzheng.gitbook.io/docker_practice/appendix_command

Docker Command Diagram – Interact with Registry



- **Pull an image or a repository from a registry**:
  docker pull [OPTIONS] NAME[:TAG|@DIGEST]
- **Push an image or a repository to a registry**:
  docker push [OPTIONS] NAME[:TAG]
- **Search the Docker Hub for images:**
  docker search [OPTIONS] TERM

https://philipzheng.gitbook.io/docker_practice/appendix_command

Outline

- Container
- What is Docker
- Basic concepts in Docker
  - Images
  - Layer architecture
  - Containers
  - Registry
- Docker Commands
- Containerisation and Dockerfile
  - Dokcerfile instructions

Containerizing an app

The process of taking an application and configuring it to run as a container is called "containerizing" or "Dockerizing".

Containers are all about apps! In particular, they're about making apps simple to build, ship, and run.

One sharing example (with docker commands):

- Use docker commands to construct an image layer by layer
- Use docker commit to make changes when necessary
- Use docker pull/push to share with others via public/private registry

Any problems that may concern the other users?

- Size of image/container
- Contents of image/container(malicious software)

A simple solution: an "audit" file that lists all contents in this image

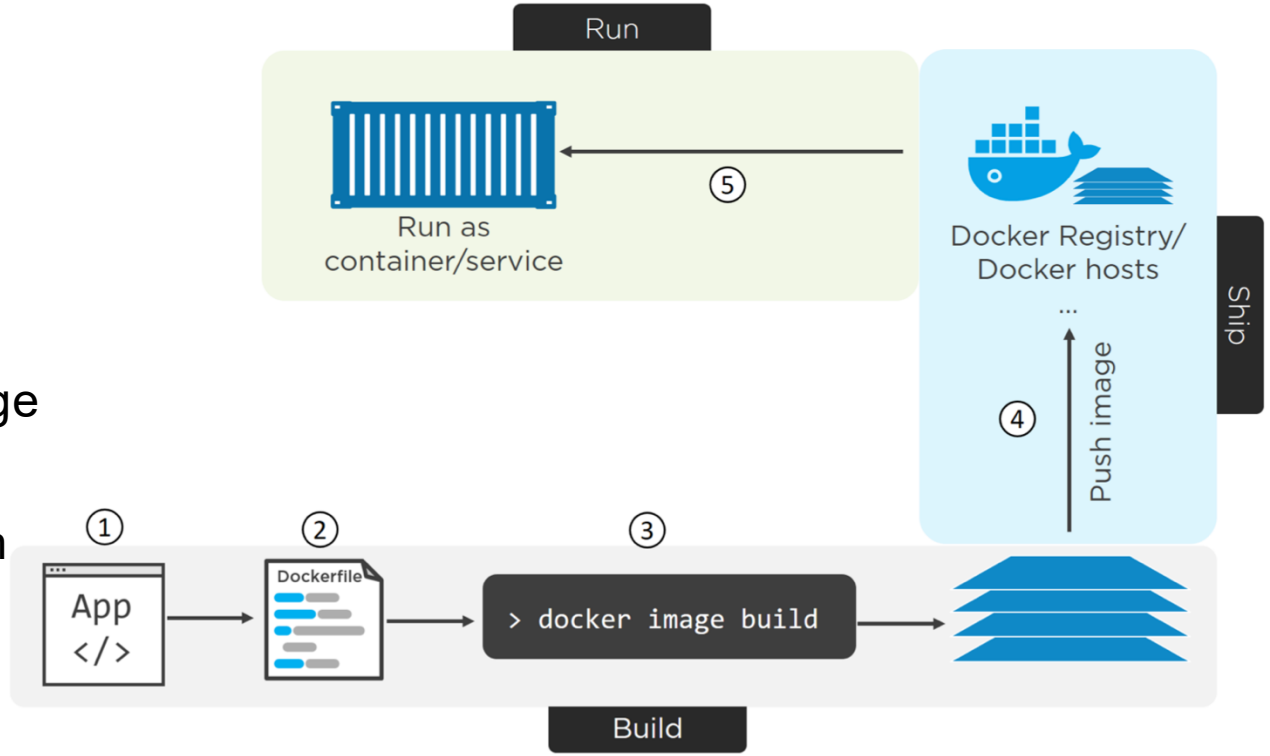
https://blog.netapp.com/blogs/containers-vs-vms/

https://www.abc.net.au/news/2023-08-14/mushroom-poisoning-leongatha-erin-patterson-police-statement/102725876

Containerizing Steps

The process of containerizing an app:

1. Start with your application code.
2. Create a Dockerfile that describes your app, its dependencies, and how to run it.
3. Feed this Dockerfile into the docker image build command.
4. Docker builds your application into a Docker image and push image to Docker Registry
5. Your teammates and you can remotely/locally run the image as a container



https://blog.netapp.com/blogs/containers-vs-vms/

Dockerfile

- is a text file that defines the **environment** inside the container
- is a collection of instructions and commands (blueprint)
- clearly tells what are contained in the image.
- Docker can build images automatically by reading the instructions from a Dockerfile
  - Usage example: docker build <dockerfile>
- Docker image is in form of dockerfile

```
1   # This dockerfile uses the ubuntu image
2   # VERSION 2 - EDITION 1
3   # Author: docker_user
4   # Command format: Instruction [arguments / command] ..
5
6   # Base image to use, this must be set as the first line
7   FROM ubuntu
8
9   # Maintainer: docker_user <docker_user at email.com>
    (@docker_user)
10  MAINTAINER docker_user docker_user@email.com
11
12  # Commands to update the image
13  RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main
    universe" >> /etc/apt/sources.list
14  RUN apt-get update && apt-get install -y nginx
15  RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf
16
17  # Commands when creating a new container
18  CMD /usr/sbin/nginx
```

https://blog.netapp.com/blogs/containers-vs-vms/

Launch a Nginx container with Docker CLI commands

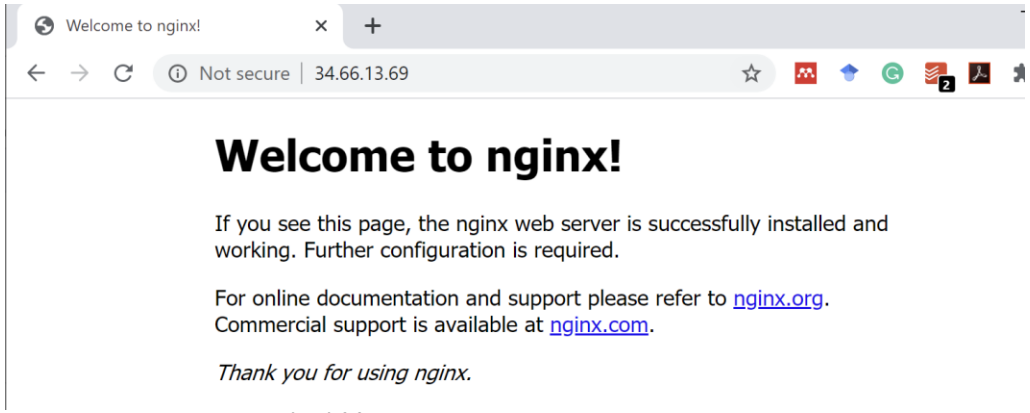
```
uqteaching@instance-1:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my_ubuntu           1.1                 499e9fad3b9c        3 days ago          64.2MB
my_ubuntu           latest              499e9fad3b9c        3 days ago          64.2MB
wordpress           latest              ee2256095234        8 days ago          543MB
mariadb             latest              7c1c380b0dd8        12 days ago         407MB
ubuntu              latest              adafef2e596e        12 days ago         73.9MB
ubuntu              18.04               d27b9ffc5667        12 days ago         64.2MB
hello-world         latest              bf756fb1ae65        6 months ago        13.3kB
uqteaching@instance-1:~$
```

```
uqteaching@instance-1:~$ docker run -it -d -p 80:80 --name ubuntu_nginx ubuntu
1d28fafcac6b5820e7be1d294b047b40866b454b2c6b6a4048fd35ef2be29a18
uqteaching@instance-1:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
               NAMES
1d28fafcac6b        ubuntu              "/bin/bash"         2 seconds ago       Up 1 second         0.0.0.
0:80->80/tcp   ubuntu_nginx
uqteaching@instance-1:~$
```

```
uqteaching@instance-1:~$ docker exec -it 1d28 bash
```

```
root@1d28fafcac6b:/# apt-get update && apt-get install -y nginx
```

```
root@1d28fafcac6b:/# service nginx start
```



Launch a Nginx container with a Dockerfile

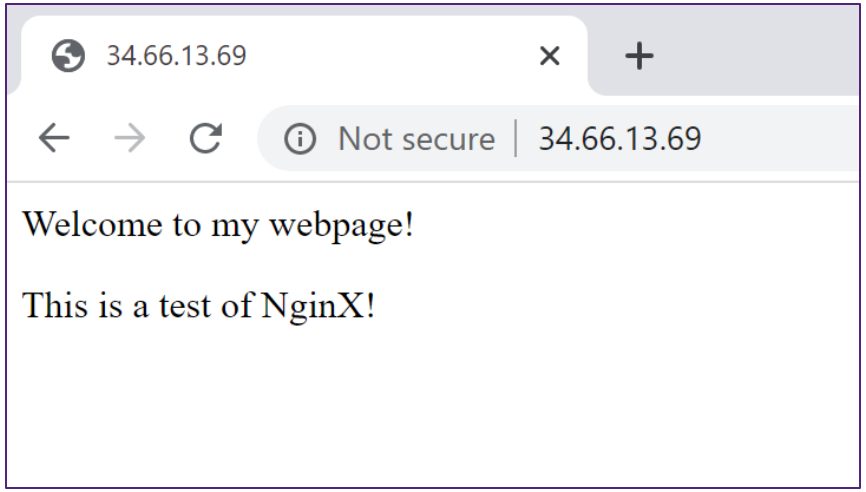
```
1  FROM ubuntu
2  LABEL maintainer="sen.wang@uq.edu.au"
3
4  # Install Nginx.
5  RUN apt-get -yqq update
6  RUN apt-get -yqq install nginx
7  COPY website /var/www/html
8  WORKDIR /var/www/html
9  # Expose ports.
10 EXPOSE 80
11 CMD ["nginx", "-g", "daemon off;"]
```

dockerfile

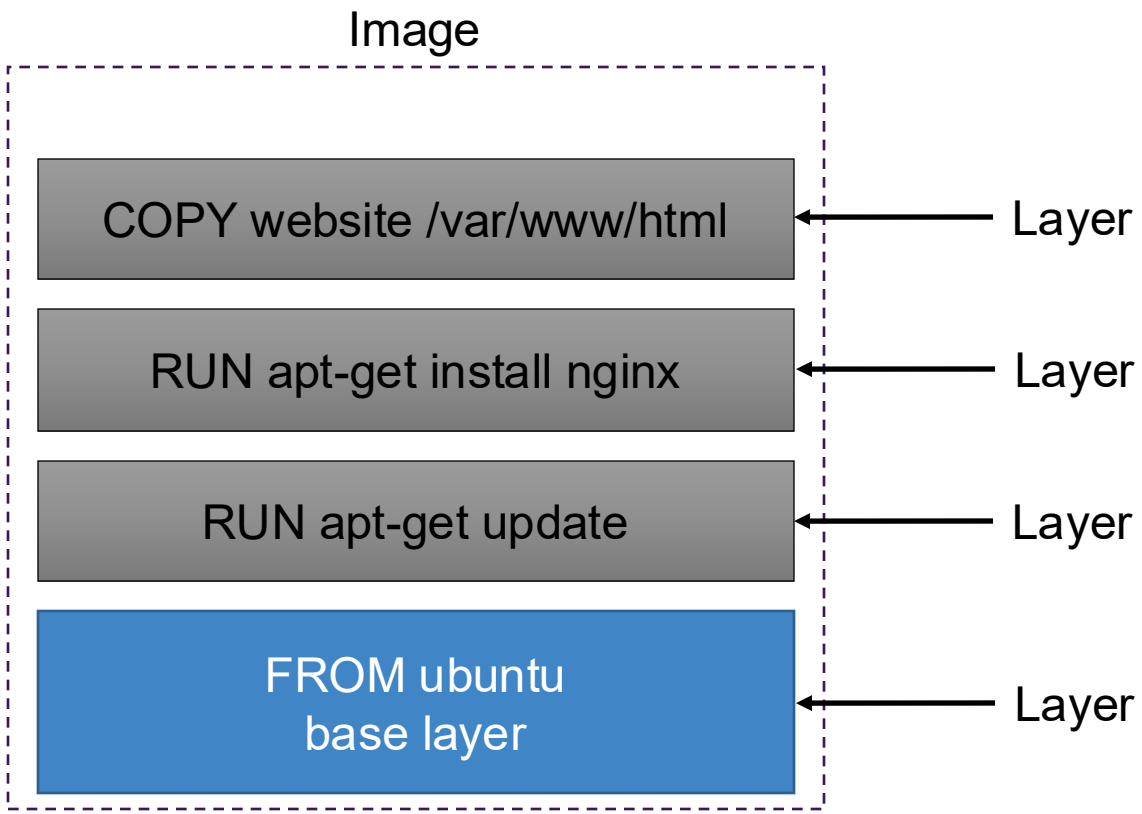
```
<h>Welcome to my webpage!</h>
<p>This is a test of NginX!</p>
~
```

Index.html

```
uqteaching@instance-1:~/lecture_demo/l3/d1$ docker build -t df_ubuntu_nginx .
uqteaching@instance-1:~/lecture_demo/l3/d1$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
df_ubuntu_nginx     latest              ad1a43b49204        9 seconds ago       155MB
ubuntu_nginx        latest              23924d7573f2        About an hour ago   155MB
my_ubuntu           1.1                 499e9fad3b9c        3 days ago          64.2MB
my_ubuntu           latest              499e9fad3b9c        3 days ago          64.2MB
nginx               latest              0901fa9da894        8 days ago          132MB
wordpress           latest              ee2256095234        8 days ago          543MB
mariadb             latest              7c1c380b0dd8        12 days ago         407MB
ubuntu              latest              adafef2e596e        12 days ago         73.9MB
ubuntu              18.04               d27b9ffc5667        12 days ago         64.2MB
debian              jessie              a3590c0e9ff9        5 weeks ago         129MB
hello-world         latest              bf756fb1ae65        6 months ago        13.3kB
uqteaching@instance-1:~/lecture_demo/l3/d1$ docker run -it -d -p 80:80 df_ubuntu_nginx
212a95bf535cb528e4c5a8c2d63ab96d6efe95006f9516fca42234255dde1c17
```



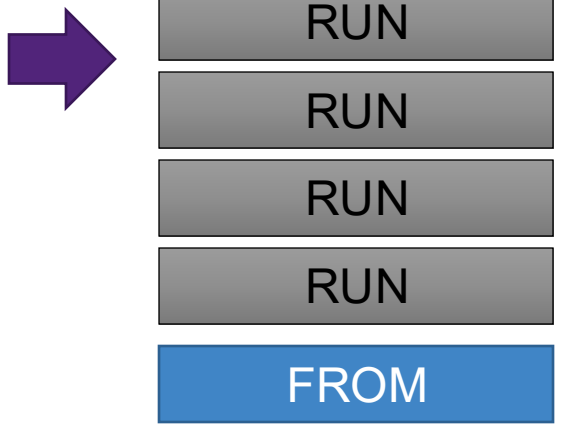
Union FS in Image Build




```
1  FROM ubuntu
2  LABEL maintainer="sen.wang@uq.edu.au"
3
4  # Install Nginx.
5  RUN apt-get -yqq update
6  RUN apt-get -yqq install nginx
7  COPY website /var/www/html
8  WORKDIR /var/www/html
9  # Expose ports.
10 EXPOSE 80
11 CMD ["nginx", "-g", "daemon off;"]
```

A Dockerfile Example

```
1  FROM debian
2
3  RUN apt-get update
4  RUN apt-get install -y gcc libc6-dev make wget
5  RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz"
6  RUN mkdir -p /usr/src/redis
7  RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1
8  RUN make -C /usr/src/redis
9  RUN make -C /usr/src/redis install
```



- We have a base image as the first layer, “RUN” will add SEVEN different layers:
- Union FS has maximum layer number, so you cannot write as many lines as you write a shell script (e.g. > 128 lines/layers)
- There are many libs/apps that are used as temporary tools, resulting in a big and redundant image.

A Dockerfile Example

```
1  FROM debian
2
3  RUN buildDeps='gcc libc6-dev make wget' \
4      && apt-get update \
5      && apt-get install -y $buildDeps \
6      && wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz" \
7      && mkdir -p /usr/src/redis \
8      && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
9      && make -C /usr/src/redis \
10     && make -C /usr/src/redis install \
11     && rm -rf /var/lib/apt/lists/* \
12     && rm redis.tar.gz \
13     && rm -r /usr/src/redis \
14     && apt-get purge -y --auto-remove $buildDeps
```

RUN

FROM

Dockerfile: FROM

Syntax:

FROM [--platform=<platform>] <image> [AS <name>]

Example:

FROM ubuntu:22.04

- FROM instruction initializes a new build stage and sets the Base Image for subsequent instructions.
- As such, a valid Dockerfile must start with a FROM instruction. ARG is the only instruction that may precede FROM in the Dockerfile.
  - ARG CODE_VERSION=latest
  - FROM base:${CODE_VERSION}
- The image can be any valid image
  - e.g. ubuntu/busybox on Docker Hub.
  - scratch: an explicitly empty image, especially for building images "FROM scratch"

Dockerfile: RUN

RUN has two forms:

- RUN <command>
  - Shell commands (by default is /bin/sh -c on Linux)
  - E.g. RUN apt-get update
- RUN ["executable", "param1", "param2"] (**exec form**)
  - The *exec* form is parsed as a JSON array, which means that you must use double-quotes (“) around words not single-quotes (‘).
  - E.g. RUN [“/bin/bash”, “-c”, “echo hello”]

To avoid redundant layers, it’s recommend to concatenate all commands with “**&&**” or “**;**” and separate them with “****”, which is a line continuation character.

```
1  FROM debian
2
3  RUN buildDeps='gcc libc6-dev make wget' \
4      && apt-get update \
5      && apt-get install -y $buildDeps \
6      && wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz" \
7      && mkdir -p /usr/src/redis \
8      && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
9      && make -C /usr/src/redis \
10     && make -C /usr/src/redis install \
11     && rm -rf /var/lib/apt/lists/* \
12     && rm redis.tar.gz \
13     && rm -r /usr/src/redis \
14     && apt-get purge -y --auto-remove $buildDeps
```

Dockerfile: COPY

COPY has two forms:

- COPY [--chown=<user>:<group>] <src>... <dest>
- COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]

<src> may contain wildcards and matching will be done using Go's filepath.Match rules.

- COPY website/*.html /var/www/html
  - Copy all html files in website to the destination folder.
- COPY website/in?ex.html /var/www/html
  - Copy any file named as in?ex.html to the destination folder. ? Is replaced with any single character, e.g. index.html or in_ex.html.

Option **"--chown=<user>:<group>"** can change user and group information in destination folder (Not working on Windows-based containers).

```
1  FROM ubuntu
2  LABEL maintainer="sen.wang@uq.edu.au"
3
4  # Install Nginx.
5  RUN apt-get -yqq update
6  RUN apt-get -yqq install nginx
7  COPY website /var/www/html
8  WORKDIR /var/www/html
9  # Expose ports.
10 EXPOSE 80
11 CMD ["nginx", "-g", "daemon off;"]
```

NOTE:
'*' matches any sequence of non-Separator characters
'?' matches any single non-Separator character

Dockerfile: ADD

ADD has two forms:

- ADD [--chown=<user>:<group>] <src>... <dest>
- ADD [--chown=<user>:<group>] ["<src>",... "<dest>"]

ADD is similar with COPY, but more powerful.

- ADD <src_URL> <dest>
  - If source is a remote file URL, ADD can download it into <dest> folder and set its permissions of 600 by default.
  - 600 (rw-,---,---): Owner(6-110)/Group(0)/Others(0)
  - If source is compressed file (tar/gzip/bzip2/gz), ADD can download it and uncompress it into destination folder.

In Best Practices for Writing Dockerfiles [1], COPY is preferred than ADD due to its semantic transparency.

[1] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Dockerfile: VOLUME

VOLUME [“PATH”] or VOLUME <PATH>

- creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.
- the value can be
  - either a JSON array, e.g. VOLUME ["/var/log/"],
  - or a plain string with multiple arguments, e.g. VOLUME /var/log.
- The docker run command initializes the newly created volume with any data that exists at the specified location within the base image.

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

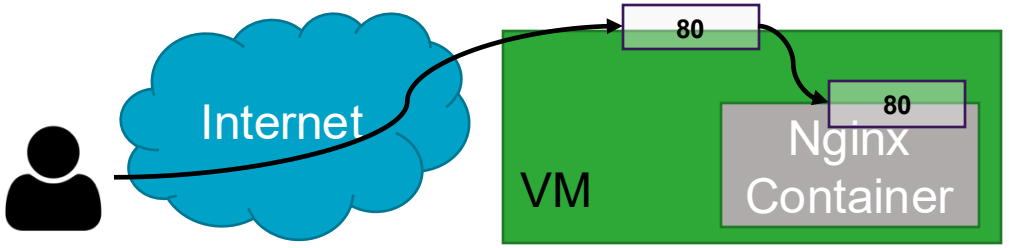
- docker run -it -d -v $PWD/mydata:/myvol [image_name]

Dockerfile: EXPOSE

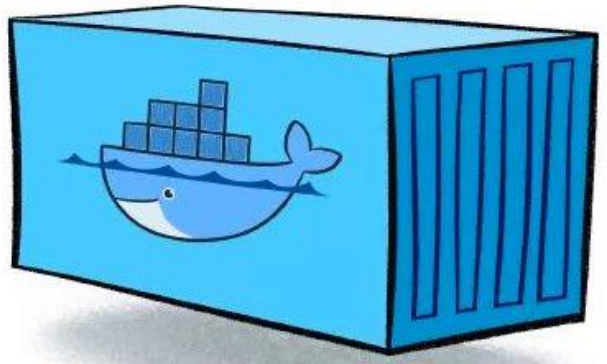
EXPOSE <port> [<port>/<protocol>...]

- informs Docker that the container listens on the specified network ports at runtime.
- can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.
  - EXPOSE 80/tcp
  - EXPOSE 80/udp
- In dockerfile, EXPOSE 80 is declaring the container will be using PORT 80, but it won't actually open the port.
- Use docker run -p <host_port>:<container_port> to map ports.

```
uqteaching@instance-1:~/lecture_demo/13/d1$ docker run -it -d -p 80:80 df_ubuntu_nginx
212a95bf535cb528e4c5a8c2d63ab96d6efe95006f9516fca42234255dde1c17
```



TCP: 80, 443, etc.



```
1  FROM ubuntu
2  LABEL maintainer="sen.wang@uq.edu.au"
3
4  # Install Nginx.
5  RUN apt-get -yqq update
6  RUN apt-get -yqq install nginx
7  COPY website /var/www/html
8  WORKDIR /var/www/html
9  # Expose ports.
10 EXPOSE 80
11 CMD ["nginx", "-g", "daemon off;"]
```

Dockerfile: CMD

The CMD instruction has three forms:

- CMD ["executable","param1","param2"] (exec form, this is the preferred form)
- CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
- CMD command param1 param2 (shell form)

The main purpose of a CMD is to provide defaults for an executing container.

- For example, the default CMD of ubuntu container is "/bin/bash"
- CMD ["sh", "-c", "echo /etc/os-release"]

There can only be ONE CMD instruction in a Dockerfile.

- If you list more than one CMD then only the last CMD will take effect.

```
1  FROM ubuntu
2  LABEL maintainer="sen.wang@uq.edu.au"
3
4  # Install Nginx.
5  RUN apt-get -yqq update
6  RUN apt-get -yqq install nginx
7  COPY website /var/www/html
8  WORKDIR /var/www/html
9  # Expose ports.
10 EXPOSE 80
11 CMD ["nginx", "-g", "daemon off;"]
```

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol

CMD echo $USER
CMD ["sh", "-c","echo $PWD"]
~
uqteaching@instance-1:~/lecture_demo/l3/d3$ docker run -it cmd_test
/
uqteaching@instance-1:~/lecture_demo/l3/d3$
```

Dockerfile: CMD

```
1  FROM ubuntu
2  LABEL maintainer="sen.wang@uq.edu.au"
3
4  # Install Nginx.
5  RUN apt-get -yqq update
6  RUN apt-get -yqq install nginx
7  COPY website /var/www/html
8  WORKDIR /var/www/html
9  # Expose ports.
10 EXPOSE 80
11 CMD ["nginx", "-g", "daemon off;"]
```

- The base image is ubuntu and its default executable is “/bin/bash” (main process)
- CMD [“nginx",“-g",“daemon off;"] set the default executable of this container is nginx and set it running in the frontend
- Why not to use “CMD service nginx start”?
  - Improper understanding the essence of container (it’s a process not a VM)
  - Container does not exist after finishing its main process.

Dockerfile: ENTRYPOINT

An ENTRYPOINT allows you to configure a container that will run as an executable.

ENTRYPOINT has two forms:

- ENTRYPOINT ["executable", "param1", "param2"] (exec form - preferred)
- ENTRYPOINT command param1 param2 (The shell form)

```
FROM ubuntu

ENTRYPOINT ["echo", "Hello"]
```

```
uqteaching@instance-1:~/lecture_demo/l3/d4$ docker run -it entry_test
Hello
uqteaching@instance-1:~/lecture_demo/l3/d4$ docker run -it entry_test Docker
Hello Docker
uqteaching@instance-1:~/lecture_demo/l3/d4$ docker run -it entry_test -setting Docker
Hello -setting Docker
```

When using both ENTRYPPOINT and CMD in one dockerfile,

- CMD will become the additional parameter -> ENTRYPOINT “CMD”
- **CMD can be overridden** by command line arguments provided when docker container runs.

```
FROM ubuntu

ENTRYPOINT ["echo", "Hello"]
CMD ["Docker World!"]
```

```
uqteaching@instance-1:~/lecture_demo/l3/d4$ docker run -it entry_cmd_test
Hello Docker World!
uqteaching@instance-1:~/lecture_demo/l3/d4$ docker run -it entry_cmd_test Cloud Comput
ing
Hello Cloud Computing
```

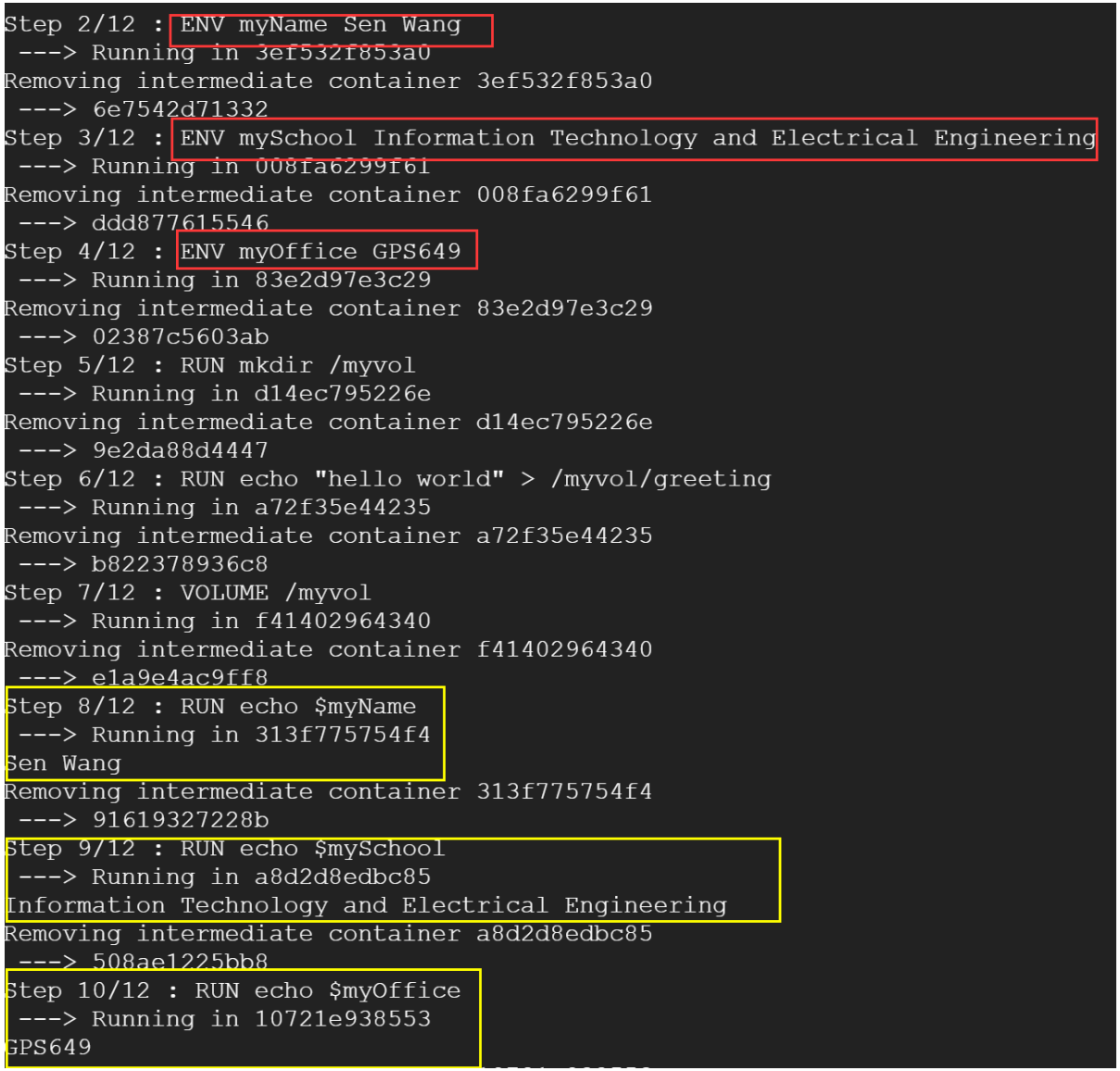
Dockerfile: ENV

The ENV instruction sets the environment variable <key> to the value <value>.

This value will be in the environment for all subsequent instructions in the build stage and can be replaced inline in many as well.

The ENV instruction has two forms:

- ENV <key> <value>
  - ENV myName Sen Wang
  - ENV mySchool EECS
  - ENV myOffice GPS649
- ENV <key>=<value> ...
  - ENV myName="Sen Wang" mySchool="EECS"



Dockerfile: ARG

The ARG instruction defines a variable that users can pass at build-time to the builder with the docker build command using the --build-arg <varname>=<value> flag.

ARG <name>[=<default value>]

```
1   FROM ubuntu
2
3   ARG user1=sen
4   ARG ver=1
```

Run "docker build --build-arg user1=wang" to build the dockerfile

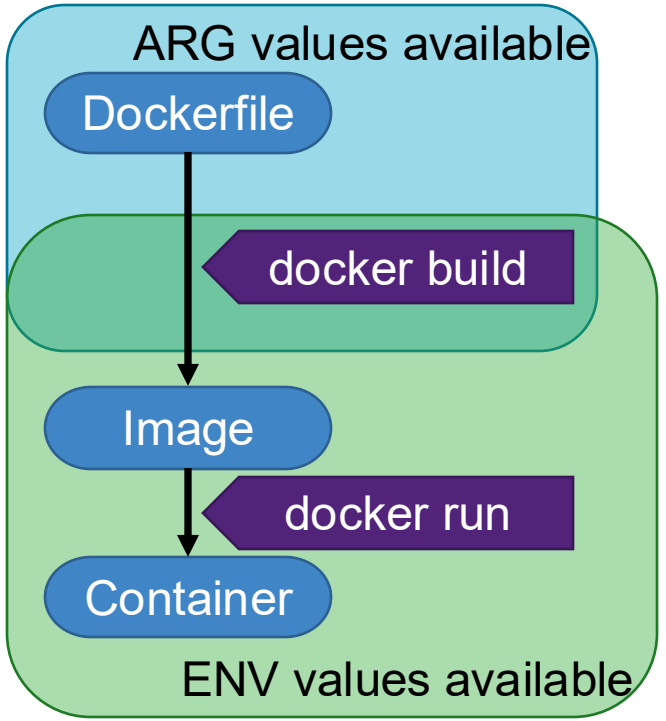
- user1 will be overridden as "wang" is passed to the first argument
- ver will be 1 as default.

Note: Do not use ARG to save your credentials (e.g. github keys, password) because build-time variable values are visible to any user of the image with the docker history command.

Dockerfile: ENV vs ARG

Both ENV and ARG can be used to define variables.

- Their life cycles are different.
  - ARG defines variables during build-time;
  - ENV defines variables during running container.
- ARG values can be changed when building the image
  - --build-arg can be used to over the ARG values
- ENV cannot be directly changed when building the image




```
FROM ubuntu

ARG VAR_A=5
ENV VAR_B $VAR_A

RUN echo $VAR_A
RUN echo $VAR_B
```

```
Sending build context to Docker daemon  2.048kB
Step 1/5 : FROM ubuntu
 ---> adafef2e596e
Step 2/5 : ARG VAR_A=5
 ---> Running in c46b303f6564
Removing intermediate container c46b303f6564
 ---> 42525016b12b
Step 3/5 : ENV VAR_B $VAR_A
 ---> Running in 59b797e3a0bd
Removing intermediate container 59b797e3a0bd
 ---> 1a3c85c8cf63
Step 4/5 : RUN echo $VAR_A
 ---> Running in 56589fa2dd52
10
Removing intermediate container 56589fa2dd52
 ---> ae645e07b22d
Step 5/5 : RUN echo $VAR_B
 ---> Running in bd9ea9da1811
10
```

```
docker build -t env_arg_test --build-arg VAR_A=10 .
```

https://vsupalov.com/docker-arg-vs-env/

Dockerfile: WORKDIR

The WORKDIR instruction sets the working directory (or current working directory)

WORKDIR /path/to/workdir

- If the WORKDIR doesn't exist, it will be created even if it's not used in any subsequent Dockerfile instruction.
- The WORKDIR instruction can be used multiple times in a Dockerfile.
- If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction.

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

```
/a/b/c
```

Why do we need WORKDIR?

```
RUN cd /app
RUN echo "hello" > world.txt
```

RUN echo "hello" > world.txt

RUN cd /app

Dockerfile: USER

The USER instruction sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile.

- USER <user>[:<group>] or USER <UID>[:<GID>]
- Similarly, USER just change the current user for the next layer.
- The <user> must be pre-defined, otherwise you cannot change.

```
RUN groupadd -r redis && useradd -r -g redis redis
USER redis
RUN [ "redis-server" ]
```

Building Solutions

Before Docker 17.05, there are two ways to build docker images:

- Pack everything into ONE Dockerfile:
  - Dockerfile could be lengthy and not easy to maintain
  - Too many layers result in fatty images and long time of deployment.
  - Code leaks risks
- Distribute to multiple Dockerfiles:
  - One dockerfile is to build the image
  - One dockerfile is to copy all the required libs/programs
  - One script to run dockerfiles
  - Can significantly reduce image size, but hard to maintain two dockerfiles synchronised.

Dockerfile.build

```
1  FROM golang:1.9-alpine
2  RUN apk --no-cache add git
3  WORKDIR /go/src/github.com/go/helloworld COPY app.go .
4  RUN go get -d -v github.com/go-sql-driver/mysql \
5  && CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
```

Dockerfile.copy

```
1  FROM alpine:latest
2  RUN apk --no-cache add ca-certificates WORKDIR /root/
3  COPY app .
4  CMD ["./app"]
```

Build.sh

```
1  #!/bin/sh
2  echo Building go/helloworld:build
3  docker build -t go/helloworld:build . -f Dockerfile.build
4  docker create --name extract go/helloworld:build
5  docker cp extract:/go/src/github.com/go/helloworld/app ./app docker rm -f extract
6  echo Building go/helloworld:2
7  docker build --no-cache -t go/helloworld:2 . -f Dockerfile.copy rm ./app
```

Multistage Builds

- Use multiple FROM statements in one Dockerfile
- Each FROM can use a different base image
- Each FROM indicates a new stage of the build
- Selectively copy artifacts from one stage to another
  - COPY --from=0 /go/src/xxx .

```
1   FROM golang:1.9-alpine
2   RUN apk --no-cache add git
3   WORKDIR /go/src/github.com/go/helloworld/
4   RUN go get -d -v github.com/go-sql-driver/mysql
5   COPY app.go .
6   RUN CGO_ENABLED=0 GOOS=linux go build -a
    -installsuffix cgo -o app .
7
8   FROM alpine:latest
9   RUN apk --no-cache add ca-certificates
10  WORKDIR /root/
11  COPY --from=0 /go/src/github.com/go/helloworld/app .
12  CMD ["./app"]
```

```
$ docker build -t go/helloworld:3 .
```

```
$ docker image ls

REPOSITORY          TAG    IMAGE ID          CREATED              SIZE
go/helloworld       3      d6911ed9c846      7 seconds ago        6.47MB
go/helloworld       2      f7cf3465432c      22 seconds ago       6.47MB
go/helloworld       1      f55d3e16affc      2 minutes ago        295MB
```

Multistage Build with one dockerfile

Multiple Dockerfiles with a script

Single lengthy dockerfile

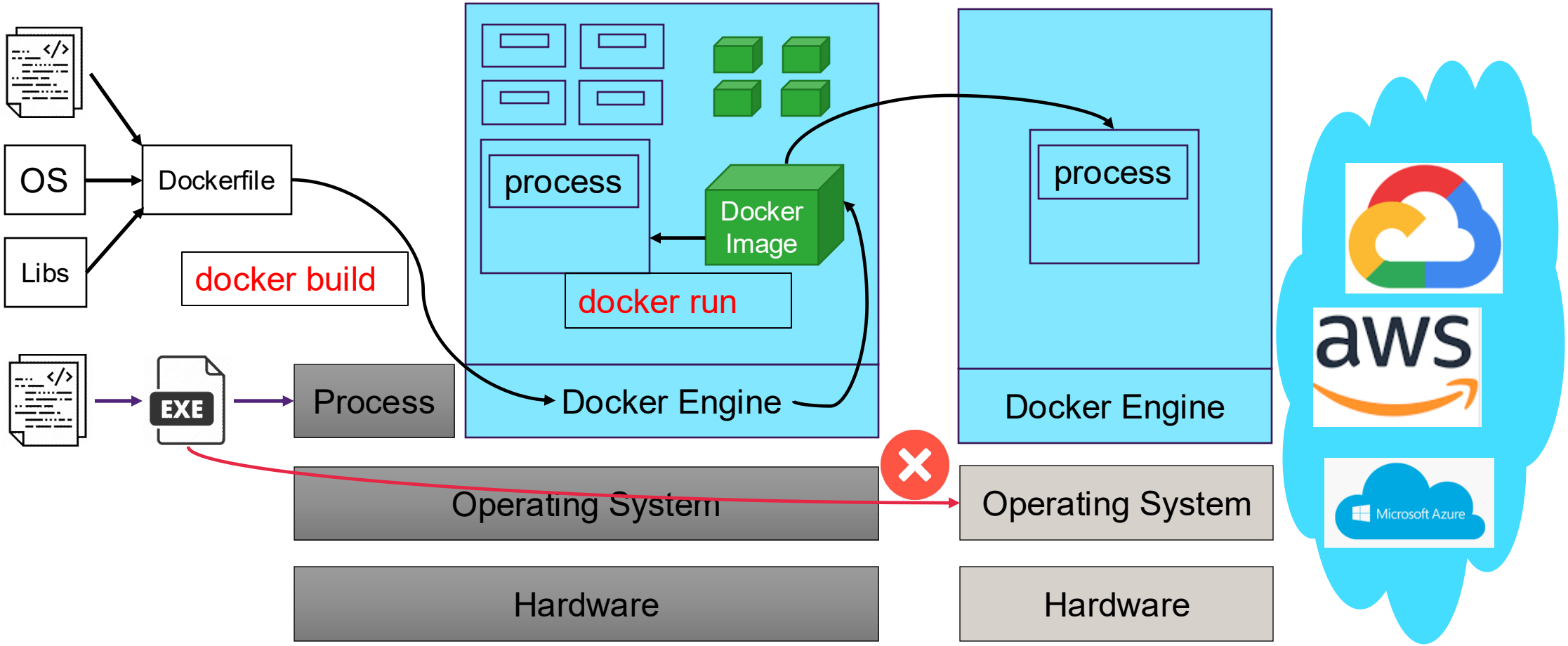
https://docs.docker.com/develop/develop-images/multistage-build/

Best practices for writing Dockerfiles

- Exclude with .dockerignore
  - like .gitignore
- Use multi-stage builds
- Don’t install unnecessary packages
- Decouple applications
- Minimize the number of layers
- Sort multi-line arguments
- …

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#exclude-with-dockerignore

Revisit: How does Dockerfile work in the Cloud?



Docker Tutorial - Part 1 - What is Docker, and Key Concepts, https://www.youtube.com/watch?v=T25Z4CUwYjE

Summary

- Container
- What is Docker
- Basic concepts in Docker
  - Images
  - Layer architecture
  - Containers
  - Registry
- Docker Commands
- Containerisation and Dockerfile
  - Dokckerfile instructions

Tutorial & Practical for Week 3

Tutorial 2 (Week 3)

1. What is a Virtual Private Cloud (VPC) network?
2. Please discuss the differences between VPCs in AWS and GCP?
3. What is Load Balancing? Please discuss the learned LB algorithms and compare them.
4. Please discuss the Layer 4 and Layer 7 load balancers.

Practical 2 (Week 3)

1. Based on the existing VM created on GCP, install and set up Docker environment
2. Docker commands for beginners
3. Set up a personal blog website with WordPress and MariaDB dockers on GCP.
4. Write a dockerfile to build your own Nginx image

Next (Week 4) Topic:

Docker II: Docker Compose and Docker Swarm