Full Stack Development

Containers, Microservices and UI

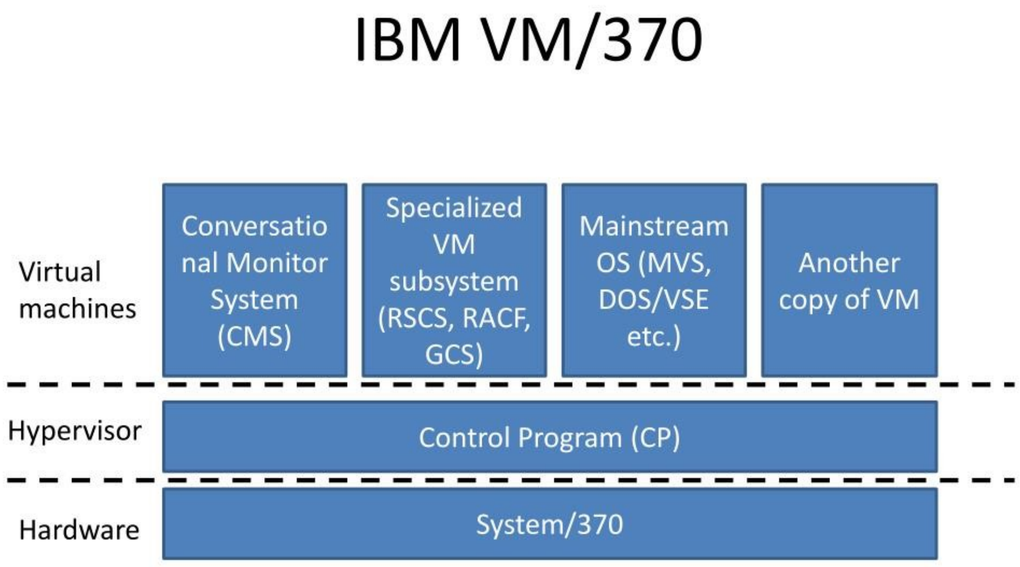
2. Docker Containers

Virtualization

- The massive increase in hardware computing power resulted software not being able to use all of the available resources
- In order to optimize use of hardware, several virtual machines could be run on the same hardware
- This virtualization model enabled the development of cloud computing
- Developers are completely insulated from the hardware
  - Code is written to run in a virtual environment
  - Virtual environments can be described in a declarative language like terraform or cloud formation in AWS for example

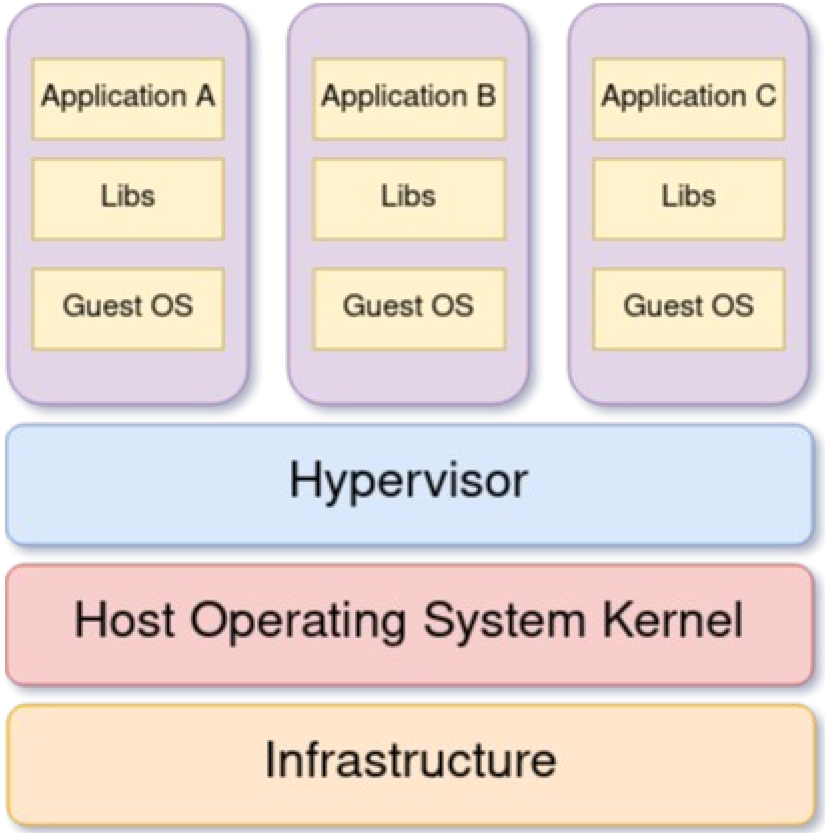
Virtualization Origins

- In the 1970s, IBM wanted to replace multiple hardware lines and associated operating systems with the 370
- This would have required clients to do massive rewrites of their code
- The VM operating system used a hypervisor to emulate the legacy hardware and software in a VM running on a 370
- This allowed for a smooth transition for clients from legacy systems to the 370


IBM VM/370
Virtual machines
Conversational Monitor System (CMS)
Specialized VM subsystem (RSCS, RACF, GCS)
Mainstream OS (MVS, DOS/VSE etc.)
Another copy of VM
Hypervisor
Control Program (CP)
Hardware
System/370

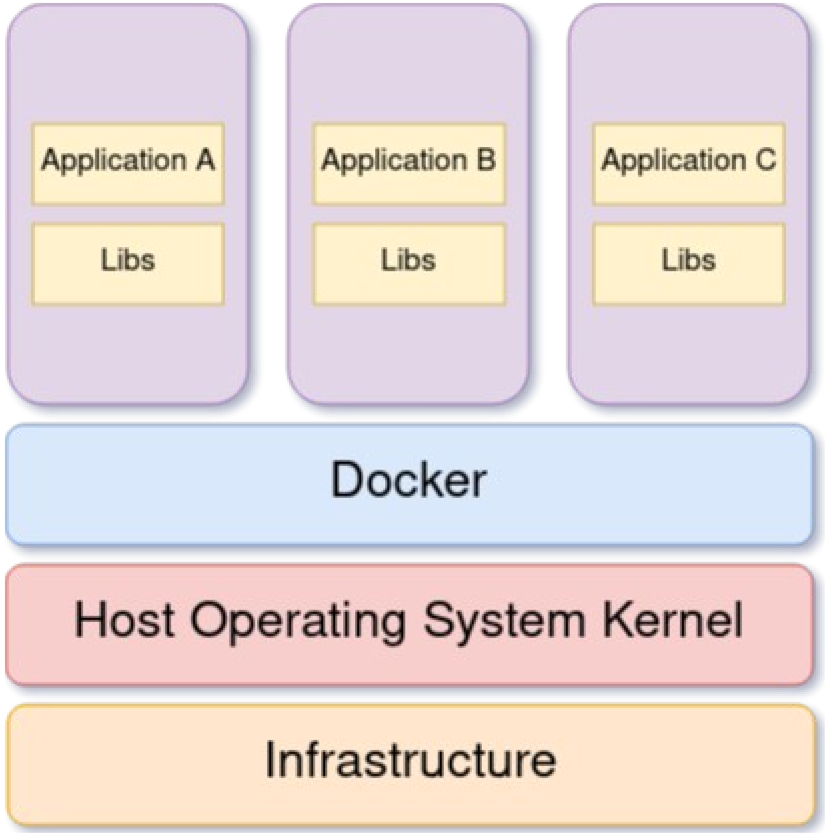
Virtual Machines

- Each VM is a full installation of a complete OS
  - The VM hard drive maps to a file or files on the host OS
  - The guest OS hardware calls are relayed to the host OS by the hypervisor
- VMs are slow to start and have a large footprint
  - Great for emulating a computer
  - Too heavyweight for running a small lightweight process



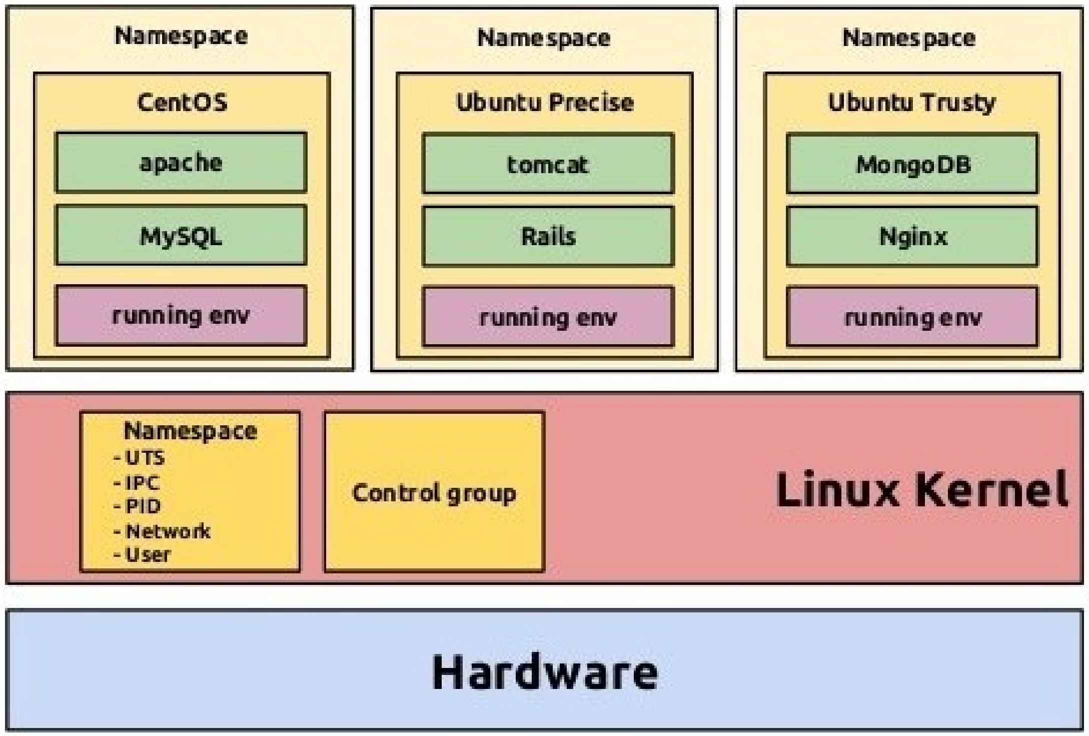
Containers

- A container is a lightweight process managed by the Docker engine
  - It has no persistent storage
  - Contains only what is needed to run the application
  - Small footprint
  - Fast start up and shutdown
- Based on Linux containers
  - Use specific features of the Linux kernel
  - Windows can run containers by using an embedded Linux VM
    - WLS - “Windows Linux subsystem”

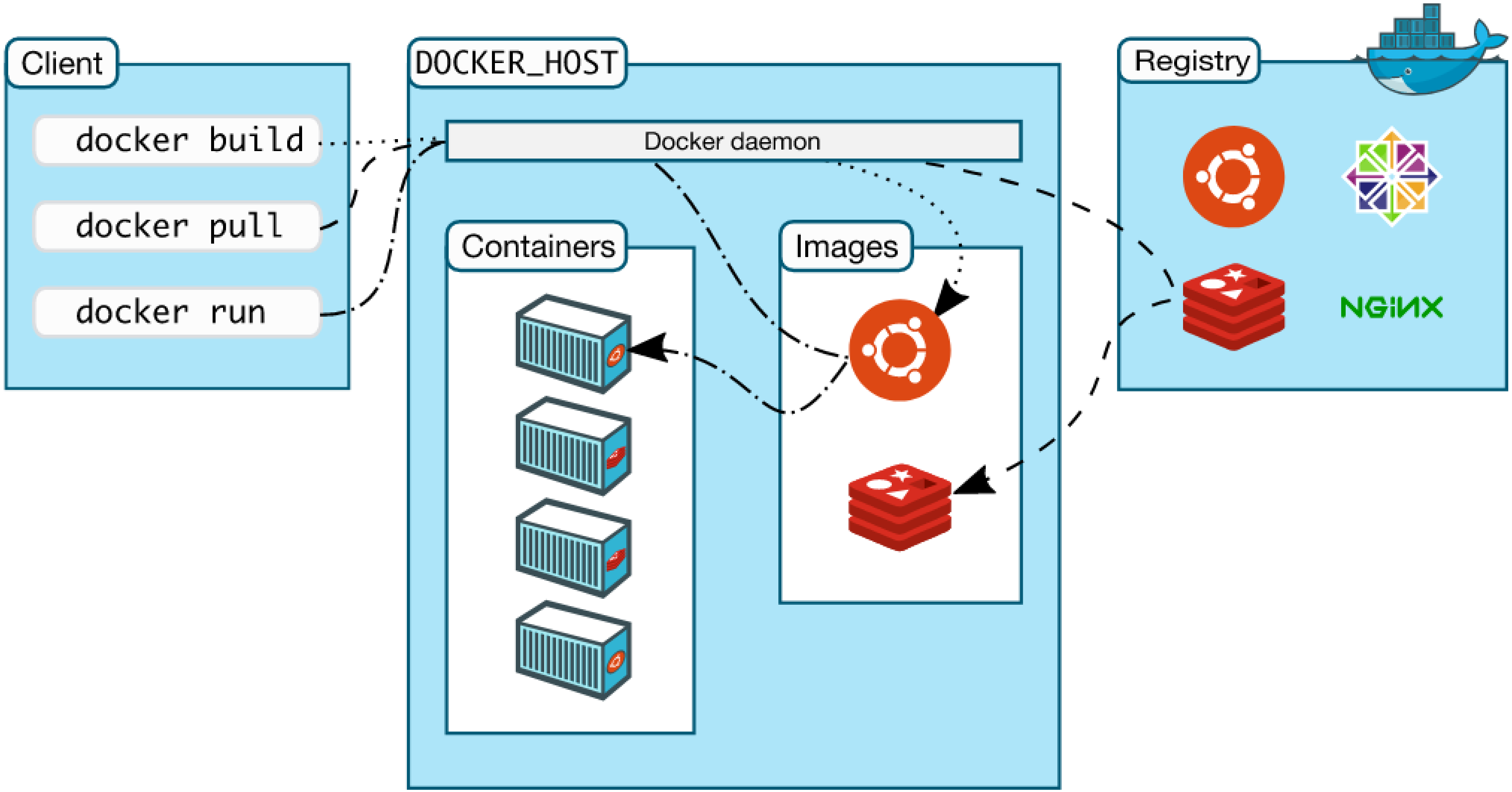


Implementation

- Linux containers run in isolated environments using Linux namespaces and control groups
- Provide resource limitation, prioritization, accounting, and control
- Hides the process space and resource information of each container from the others
- Docker is an implementation of Linux containers



Docker Architecture

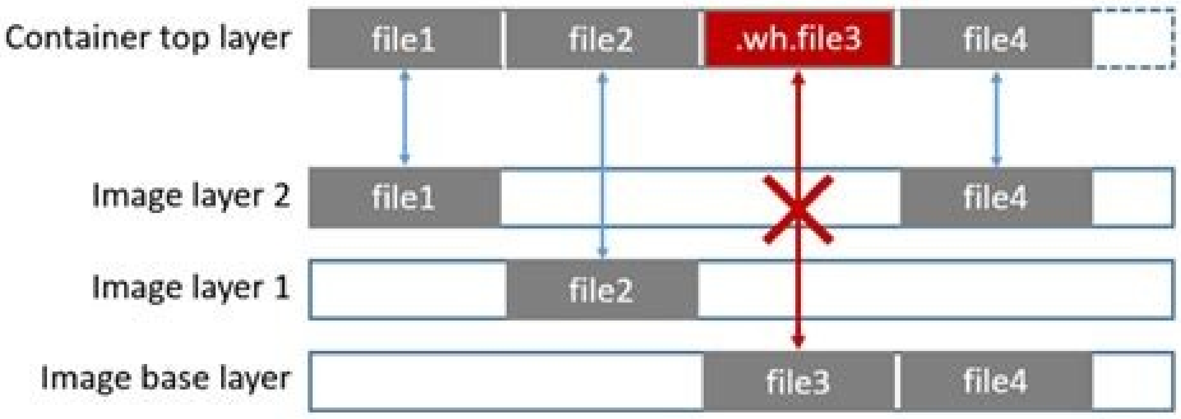


Docker Terminology

- Docker Daemon or Engine
  - The process that manages images and containers on the Docker host
  - The Docker CLI is used to request services from the Docker engine
- Docker Image
  - Analogous to an executable file - template for running a container
- Docker Container
  - A Docker image that is executing or has finished executing
  - Analogous to a process that is running an executable file
  - Multiple containers can be created and run from a single image
- Docker Registry
  - A version collection Docker images
  - Each set of versions for an image is referred to as an image repository

Docker Images

- Docker images are read only
  - Uniquely identified by hash codes
- Built-up in layers
  - Uses Linux union file system, also referred to an overlay file system
  - Each layer is immutable identified by a unique hash code
  - Layers are shared by images – only one copy of a layer exists



Docker container
(AUFS storage-driver demonstrating whiteout file)

Docker Registry

- The local registry cache
  - On the machine running Docker
  - Images pulled from other registries are cached here
  - This is the first registry searched for a requested image
- Docker Hub
  - Public repository maintained by Docker
  - Searched by default after the local registry
- Other registries
  - Docker can be configured to use other registries
  - Allows control over which images Docker pulls
  - Ensures only approved images are used by Docker installation

Docker Repository

- Images are versioned
  - A set of versioned images is called repository
  - A specific image is referenced by <image_name>:<version_tag>
  - The following are different versions of the Ubuntu image
    - ubuntu:18.04
    - ubuntu:20.04
  - If no version tag is specified, then the version defaults to “latest”
    - Pulling the image **ubuntu** is the same as pulling **ubuntu:lates**t
- Images are uniquely identified by their digest value
  - Tags are identifies that are added for convenience
  - Images do not have to have tags but a single image can have multiple tags
  - Images can only be deleted if they have zero or one tags

Exploring Docker Registries

Demo

Working with Docker Images

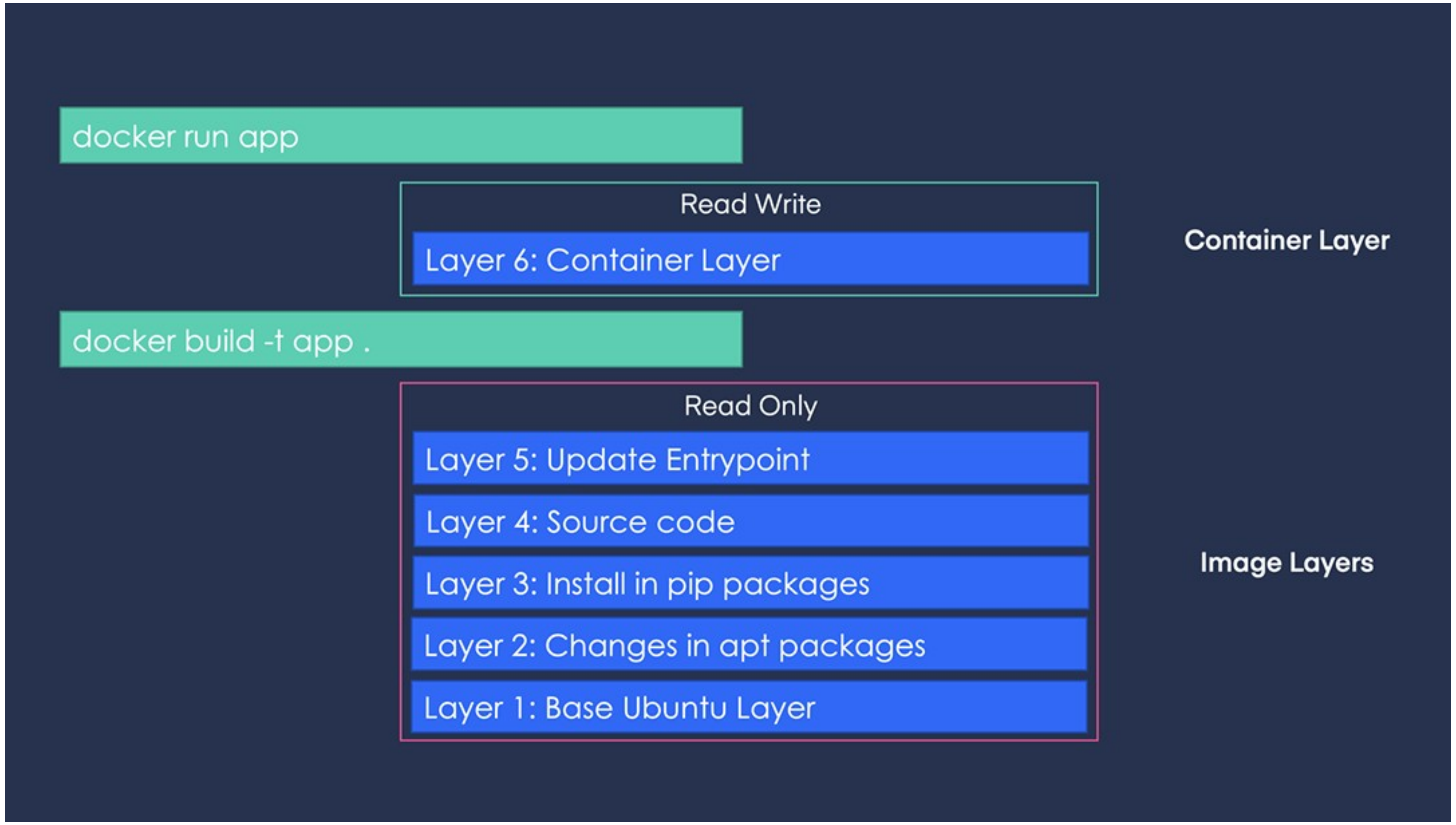
Demo

Docker Images

Lab Docker 1

Docker Containers

- Containers are running copies of a Docker image
- Containers have an additional write-able layer added to the image layers



Running Containers

- The **docker run** command starts a container based on an image
- The image contains a default command to run when it starts
  - Once the command completes, the container exits
  - Some containers, running a web server for example, do not exit
  - These have to be shut down with the **docker stop** command or **docker kill**
- Stopped containers are not destroyed but can be restarted
  - The command **docker start** restarts a stopped container but not exited containers
  - The command **docker create** creates a container but does not run it
  - The command **docker run** = **docker create** followed by **docker start**
- Specific commands inside a container can be executed
  - Inside an already running container with **docker exec <cmd>**
  - Or by starting up a container with **docker run <cmd>**

Running Containers

- Running containers have a hash id just like images
  - They also can have an optional name **docker run ubuntu --name zippy**
  - Containers are assigned default names otherwise
- There can be multiple containers created from a single image
- Commands used to work with containers
  - **docker ps** – lists all the running processes related to containers
  - **docker ps -a** – lists all of the running and exited processes related to containers
  - Using **docker container ls (-a)** gives exactly the same output
- Docker keeps logs of all activity in each container
  - We can access both a container’s logs and monitor its running processes

Running Containers

- Interactive terminal connections allow us to work within a container if the container supports a shell
  - To work with a shell in Ubuntu we could run **docker run -it ubuntu**
    - If we omit the **-it**, the shell will start up and immediately exit
  - We can override the default command in the image
    - Normally the nginx image starts a web server and does not exit
    - We can start a shell instead with the following command **docker run -it nginix bin/sh**
  - For a running container, we can execute a command using **docker exec -it <container id>**

Running Docker Containers

Demo

Running Docker Containers

Docker Lab 2

Docker Networks

- Docker engine runs a set of private networks
  - Each container gets an IP address on the docker network
  - If the container provides a service, it normally is exposed through a port on the container
  - The docker engine will map ports on the host networks to ports on containers
- Private docker networks
  - Allow containers to run without interfering with IP addresses or ports on the host system
  - Network types:
    - **Bridge**: the default - creates a private internal isolated network for containers
    - **Host**: allows containers to run on the host network – only implemented for Linux hosts
    - **Overlay**: allows containers on different hosts to communicate with each other
- Exposing ports
  - Services offered by container are specified by port numbers which are made available via port exposing
  - Specific ports to be expose can be defined in an image or a container

Docker Networking

Demo

Docker Networking

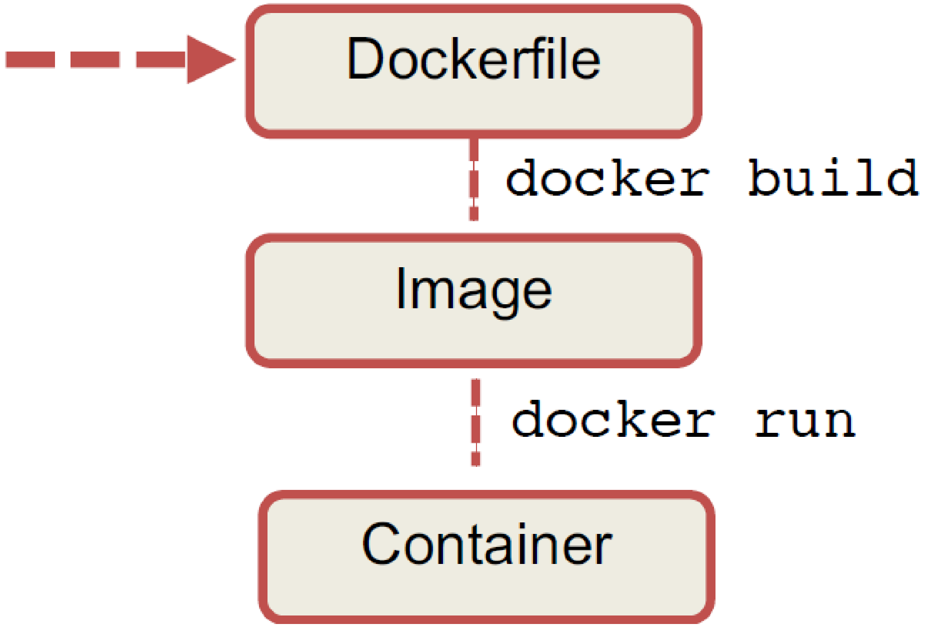
Docker Lab 3

Building Images with Dockerfile

- Docker can build images automatically by executing instructions in a Dockerfile
  - For example, to build an image with Python three installed on a Debian Linux container

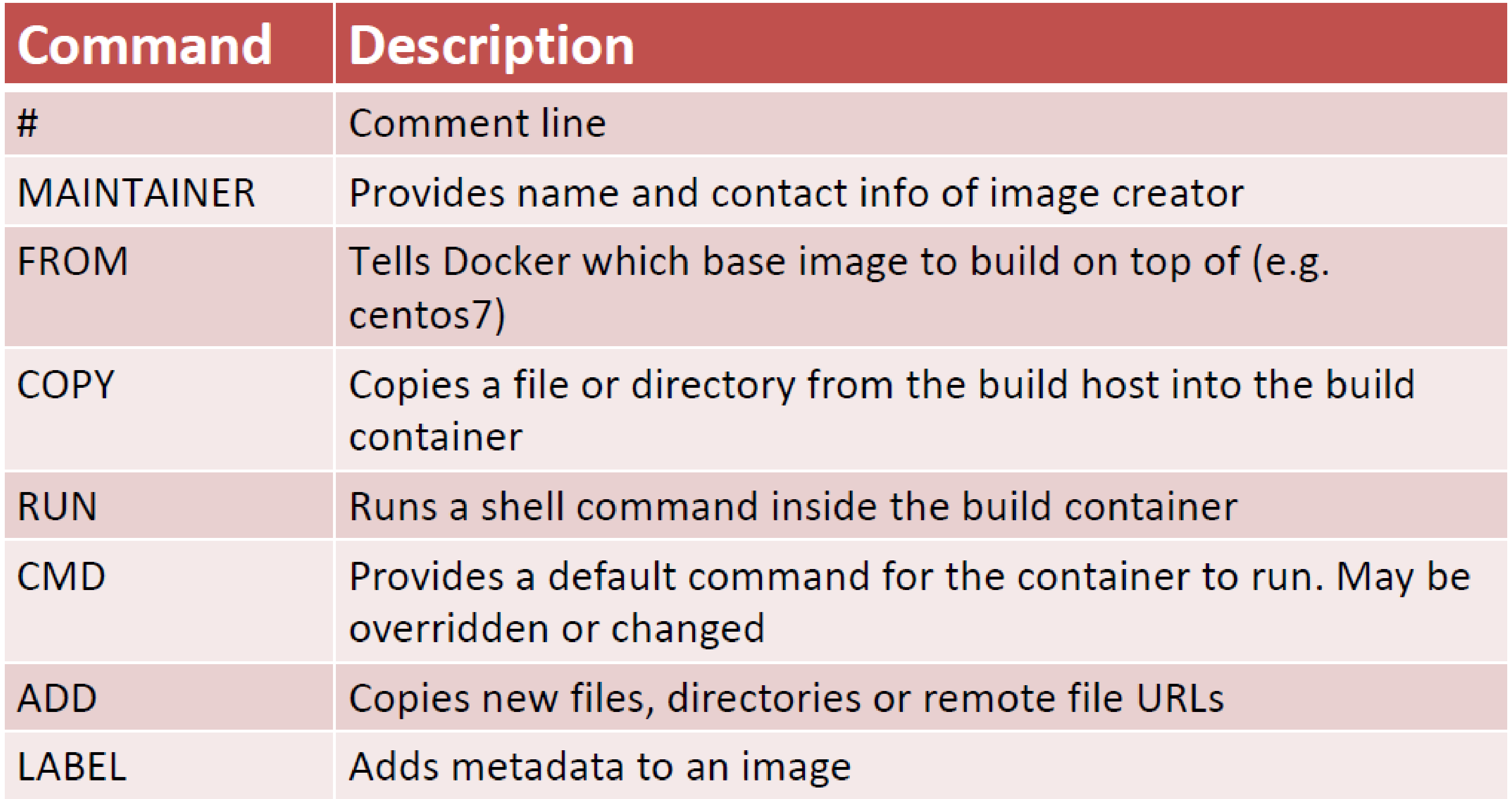
Dockerfile

```
FROM debian:bullseye

RUN apt-get update –y

RUN apt-get install
python3
```



Dockerfile

- The Dockerfile is a text file
  - contains the instructions that you would execute on the command line to create an image
  - Docker provides a set of standard instructions to be used in the Dockerfile

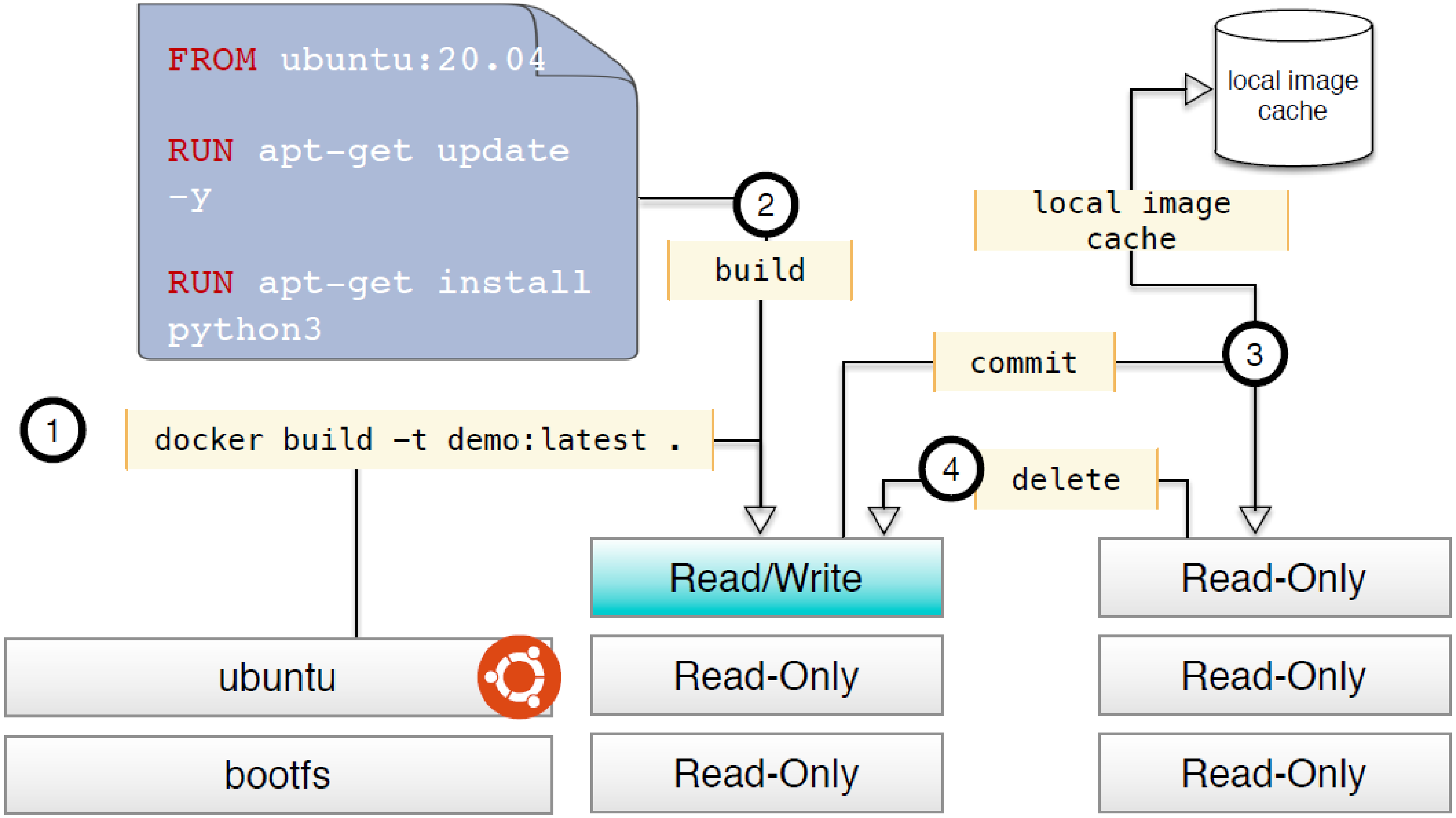


Command	Description
#	Comment line
MAINTAINER	Provides name and contact info of image creator
FROM	Tells Docker which base image to build on top of (e.g. centos7)
COPY	Copies a file or directory from the build host into the build container
RUN	Runs a shell command inside the build container
CMD	Provides a default command for the container to run. May be overridden or changed
ADD	Copies new files, directories or remote file URLs
LABEL	Adds metadata to an image

The Build Process

- Docker builds an image by running a series of containers
  - The FROM base image becomes the first layer in the new image
  - The base image is run in a temporary container
  - The first directive, RUN for example, executes and results written to the container’s writeable layer
  - The container is committed to a new temporary image
  - The writeable layer now a new image layer
  - This new temporary image is run in a new container, the next directive executes
  - The container is committed to a new temporary image
  - And this continues until the whole Dockerfile is executed

The Build Process



```
FROM ubuntu:20.04

RUN apt-get update
-y

RUN apt-get install
python3
```

Dockerfile

Demo

Dockerfiles

Docker Lab 4

Docker Monitoring Tools

- Debugging containers can be problematic
- Docker has a number of monitoring tools that can be used
  - docker logs <container>: displays console output of the container
  - docker top <container>: lists all the processes running in a container
  - docker stats <container>: streams real time stats of containers
  - docker inspect <container>: displays detailed container configuration

Docker Monitoring Tools

Demo

Docker Volumes

- Volumes are where a directory on the host file system is mounted inside a container
  - Anything written to the volume remains on the host file system
  - Any other container can mount the volume and read what was written
- A wide range of file systems can be mounted
  - Amazon AWS buckets for example
- On Windows, there is not direct access to the underlying file since it is created in the Linux VM
  - However, deleting the volume will delete the underlying file

Docker Volumes

Demo

Docker Volumes

Docker Lab 5

Docker Compose

- Docker is written in GO
- Docker compose is a developer tool written in Python
- Docker compose is NOT intended to be a production deployment tool
  - It is designed to allow developers to quickly deploy an application using multiple docker containers with single command
  - And to shut it down with a single command
- The configuration of the application is in a docker-compose.yaml file
- Docker compose creates a new private network to run the app in
  - It shuts down the network when the app is shut down
  - Allows multiple instance of the application to run independently and concurrently

Docker Compose File

- The docker compose file specifies:
  - A set of images to be run called services
  - Any volumes that need to be created or previous volumes to be remounted
  - The configuration information such as ports exposed, etc.
  - Dependencies between the services
- Multiple copies of the application can run at the same time
  - Each instance runs in its own network address space

Docker Compose File

- The docker compose file specifies:
  - A set of images to be run called services
  - Any volumes that need to be created or previous volumes to be remounted
  - The configuration information such as ports exposed, etc.
  - Dependencies between the services
- Multiple copies of the application can run at the same time
  - Each instance runs in its own network address space

Docker Compose

Demo

Docker Compose

Docker Lab 6

Docker Registries

- Registries
  - Tags are of the form registry_host:port/repository_name:verson
  - Registry services expose the given port in order to manage image requests
  - We will using the registry docker image to provide a registry service on our local machine
  - The docker registry container does not store images, it processes pull and push requests
- The actual storage location of the images is normally specified
  - We are using a default of the local file system
  - But it can be a variety of storage mediums, like AWS S3 buckets for example
- The host running the registry container is referred to as the registry host
  - We can have any number of registries running, either on different hosts, or the same host but at different ports
  - In our lab, localhost will be our registry server which will be listening on port 5000
  - The registry container mounts the actual physical storage of the images as a docker volume

Docker Registries

- There are two default registries
- The local cache
  - This is not an actual registry, but where images pulled from different registries are cached
  - Also where our newly created images are stored
  - After creating a new image, it is a best practice to push it to a registry
- When we execute a command like “pull ubuntu”
  - Since there is no registry specified, Docker first checks the local cache
  - If there is no image in the local cache, Docker rewrites the pull request as
    - **Docker pull library/alpine:latest** where “library” is the default repo, in this case docker-hub
  - If there is a registry component to the image tag, that is used instead of the default

Docker Registry

- We can download and run the image for the registry executable with
  - **docker run -d -p 5000:5000 --restart=always --name registry registry:2**
- To push an image, we have to tag it with the appropriate registry info
  - **docker tag ubuntu:latest localhost:5000/my-ubuntu**
  - Notice our registry is the host and port, the name “registry” is just the docker container that processes the image requests into and out of the associated docker volume
- The we just push it to the registry
  - **docker push localhost:5000/my-ubuntu**
  - Now there exist two copies of the image, one in the cache and one in the registry
  - Normally we use a registry name that is self explanatory like:
    - **docker run -d -p 5000:5000 --restart=always --name devteam1 registry:2**
    - **docker run -d -p 5500:5000 --restart=always --name prod_qa registry:2**
  - But docker commands use the port and host, so docker sees these as localhost:5000 and localhost:5500

Docker Registry Tools

- Normally there is no GUI to a private registry
  - However there are web based tools that can be used
- The standard commands work on the cache
  - If you can’t execute a command line command on docker hub, you can’t do it on your registry
- The local cache is where we do our docker programming
- In brief
  - A registry is a docker volume
  - The registry container is a service that runs in a container that pulls and pushes images into and out of the volume
  - When we remove a registry, it is not enough to remove the registry container
  - We also have to remove the associated volume

Docker Registries

Demo

Docker Registries

Docker Lab 7

Questions?

Class Project Discussion

CMA CGM
CMAU 115887 0
22G1
CMAU 159369 3
22G1
TRLU US 308341 2
2210
GATU 082138 7
22G1
TTNU 168218 2
GESU 276340 7
22G1
DELMAS
DVRU 147046 5
22G1
DVRU 143916 1
22G1
XINU 108806 5
FCIU 268180 8
CLHU 309435 0
DVRU 151838 4
TRLU US 275847 1
2210
TRIU US 396740 3
2210
凯 航
CNC LINE
tex

End Module