/kahootali

# DevOps - Week 3 - Docker

Muhammad Ali Kahoot

DevOps Course By M. Ali Kahoot - Dice Analytics

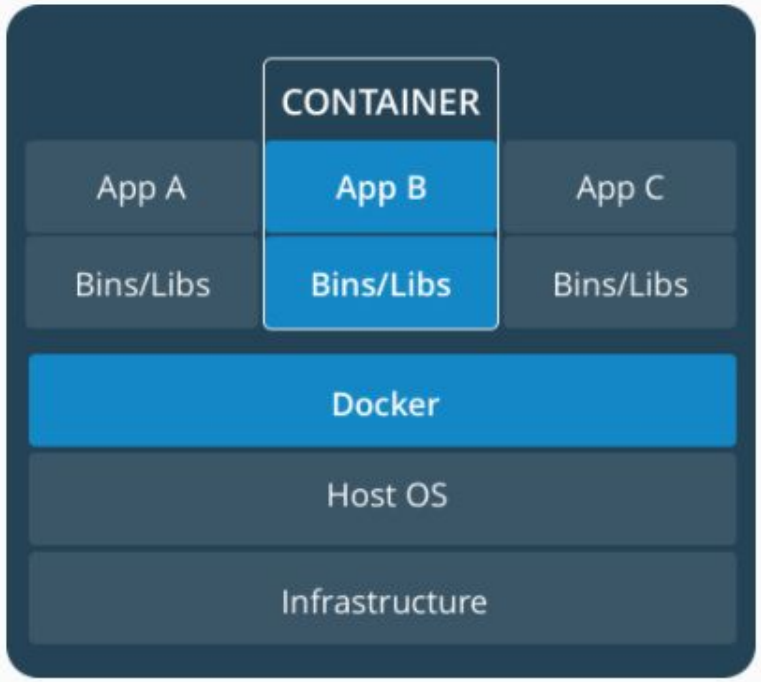
# QUIZ

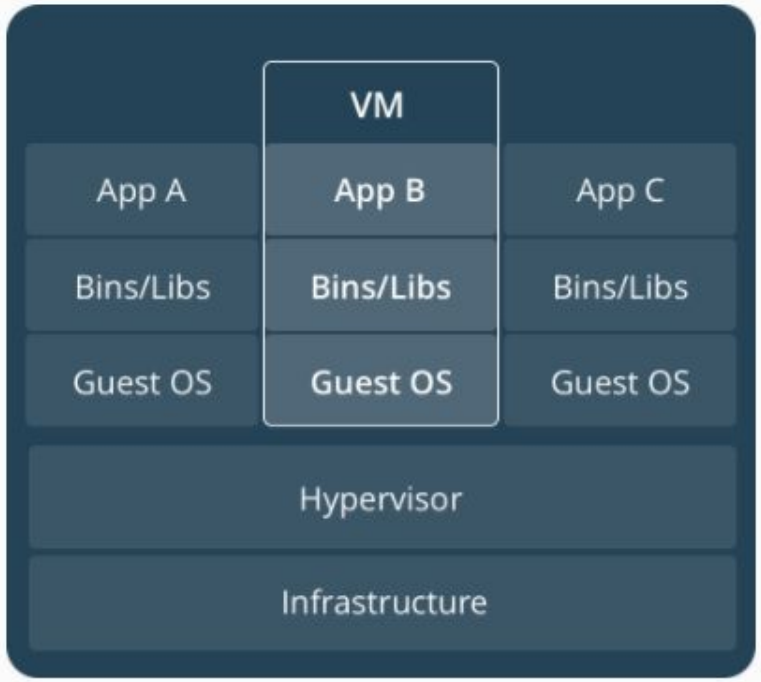
- Write differences between VMs & Containers
- Explain Docker Architecture
- Write command to run containers in background and map local port 8080 to containers 90 port
- Write command to run containers in foreground
- Write command to see logs of container
- Write command to pull image

# Disclaimer

These slides are made with a lot of effort, so it is a humble request not to share it with any one or reproduce in any way.

# Containers vs VM




### CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.

### VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.



Source: https://www.docker.com/what-container#/package_software

# Containers vs VM

**VMs**

- Hardware level virtualization i.e. abstraction of physical hardware
- Has own Hardware & OS
- Each VM has a full copy of an operating system + application + binaries + libraries
- can take up to tens of GBs.
- VMs are isolated, apps are not
- Complete OS, Static Compute, Static Memory, High Resource Usage

**Containers**

- OS level virtualization i.e. abstraction at the app layer (code + dependencies)
- Share hardware, host OS kernel but can have own OS
- take up less space (typically tens to hundreds of MBs in size)
- containers are isolated, so are the apps
- Container Isolation, Shared Kernel, Burstable Compute, Burstable Memory, Low Resource Usage

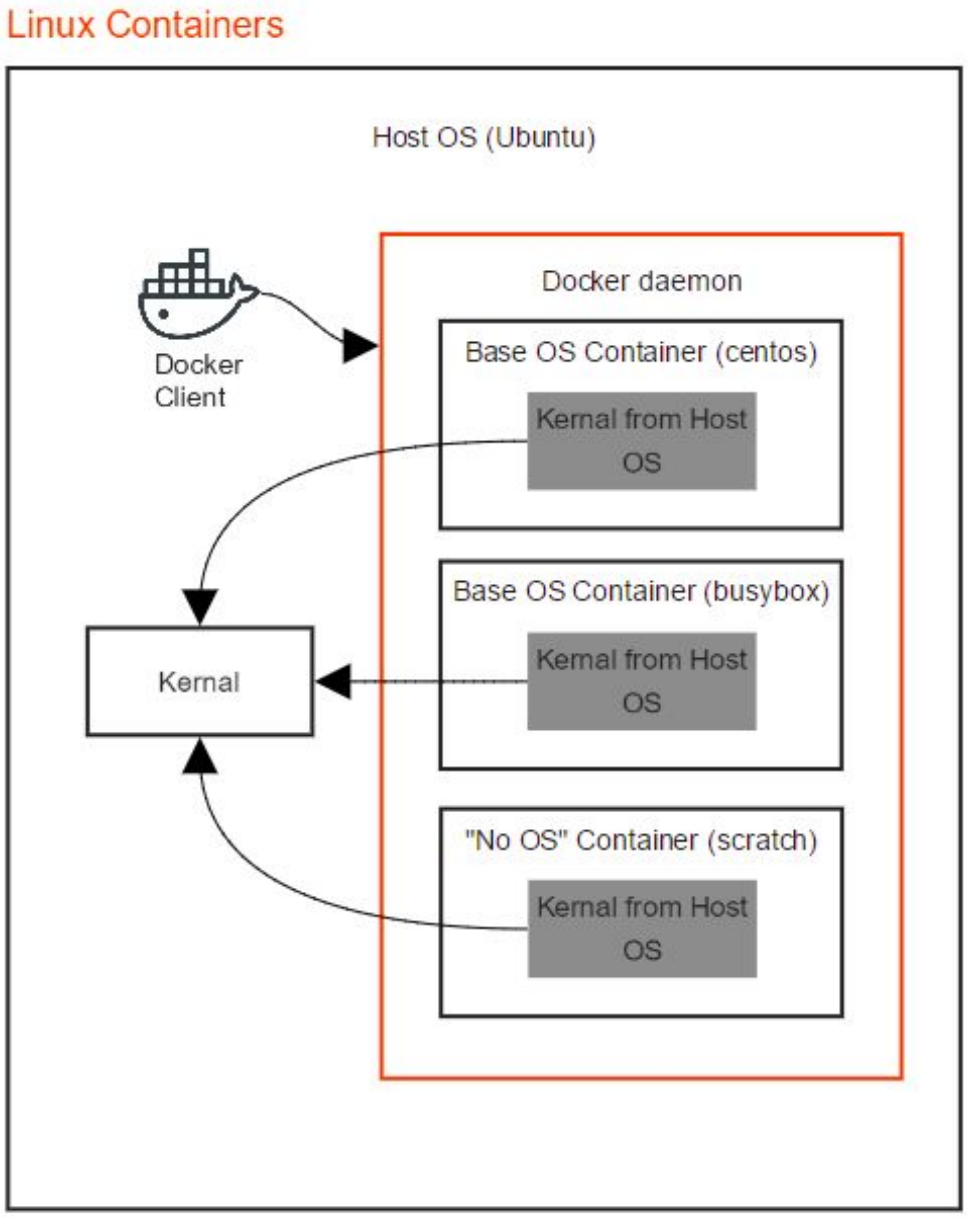
# Containers vs VM

**VMs**

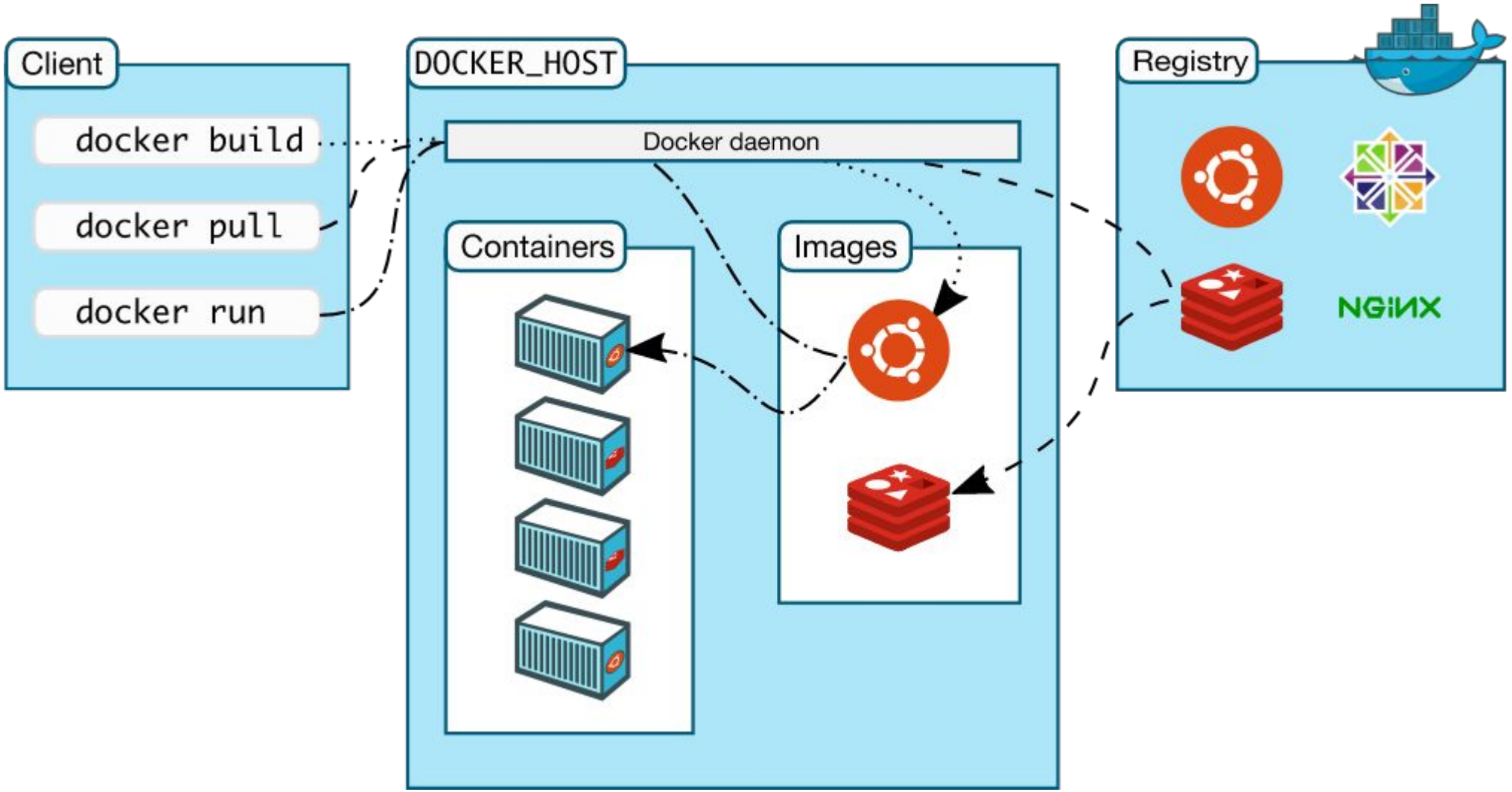
- Ops were responsible for creating VMs, installing Software Dependencies, then installing Software which might not work due to some compatibility issues
- Dev responsible for Software Development and running on local machine vs Ops running the Software on VM with newly installed Libraries
- Works on my machine issue

**Containers**

- Ops are responsible for VM creation and installing Docker only
- Dev writes code and tests in local container based on the same image
- Same image is deployed in Stage, Prod
- Ideally no "WORKS ON MY MACHINE" issue
- Process level isolation but relatively less secure


Linux Containers
Host OS (Ubuntu)
Docker daemon
Docker Client
Base OS Container (centos)
Kernal from Host OS
Base OS Container (busybox)
Kernal from Host OS
Kernal
"No OS" Container (scratch)
Kernal from Host OS

# Docker Architecture



# DOCKER COMMANDS

```
docker container run -d --publish 8081:80 --name nginx-cont nginx
docker container run -d -p 8082:80 --name apache-cont httpd
docker container run -it ubuntu
docker container run -it centos:7
docker container ps
docker container prune
docker images
docker image pull centos:8
docker container stop nginx-cont
docker container rm nginx-cont
docker container rm apache-cont -f
```

# DOCKER NETWORKS

Each container connected to a private virtual network - "bridge"

Each virtual network routes through NAT firewall on host - IP

All containers on a virtual network can talk to each other

Best practice is to create a new virtual network for each application

Can attach containers to more than one virtual network - (or none)

Skip virtual networks and use host IP (--net=host)

# DOCKER NETWORKS

- Intercommunication never leaves host
- All externally exposed ports closed by default
- You must manually expose via -p, which is better default - security
- Containers shouldn't rely on IP's for inter-communication
- DNS is the key to easy inter-container communication
- DNS for friendly names is built-in if you use custom - networks

# DOCKER NETWORKS

Types of Network Drivers

- Bridge (A virtual Network)
- Overlay (When defining multiple nodes)
- Host (Use host network to run apps, publish will not work)
- None (No IP Address)

# DOCKER NETWORKS COMMANDS

Show networks: `docker network ls`

Inspect a network: `docker network inspect`

Create a network: `docker network create --driver`

Attach a network to container: `docker network connect` or `docker container run --net <network-name> <container-name/id>`

Detach a network from container: `docker network disconnect`

# LAB - NETWORKS - BRIDGE

- Create Network
- List Network
- Intercommunication of containers
- Delete Containers
- Delete Network

# LAB - NETWORKS - BRIDGE

- Intercommunication of containers

Run 2 containers and try to ping them

```
docker container run --rm --name centos1 -it centos
```

Open another terminal on your system and run second container:

```
docker container run --rm --name centos2 -it centos
```

# LAB - NETWORKS - BRIDGE

- Create Network

```
docker network create new-network
```

By default bridge driver type is used

- List Network

List all virtual networks:

```
docker network ls
```

# LAB - NETWORKS - BRIDGE

- Intercommunication of containers

Run 2 containers and try to ping them

```
docker container run --rm --net new-network --name centos1 -it
centos
```

Open another terminal on your system and run second container:

```
docker container run --rm --net new-network --name centos2 -it
centos
```

# LAB - NETWORKS - BRIDGE

- Intercommunication of containers

Access centos1 from shell of centos2 using ping:

```
ping centos1
```

Go back to shell of centos1 and run the command:

```
ping centos2
```

# LAB - NETWORKS - BRIDGE

- Delete Containers

Delete centos1 and centos2 containers:

```
docker container rm centos1 centos2
```

- Delete Network

```
docker network rm new-network
```

# LAB - NETWORKS - HOST

Run container with host network

```
docker container run --net host nginx
```

Access your local machine, and try to access port 80, either open in browser `localhost:80` or open a terminal and run

```
wget localhost:80
```

And see content of index.html and you will see nginx content

# PERSISTENT DATA

Containers are immutable and ephemeral

Containers gets deleted, all changes done to that container gets deleted

What about databases, or unique data?

Docker gives us a feature "persistent data" to resolve this

Two ways:

- Bind Mounts: link container path to host path
- Volumes: make special location outside of container UFS

# PERSISTENT DATA: BIND MOUNTING

Maps a host file or directory to a container file or directory

Basically just two locations pointing to the same file(s)

Bypasses Union File System

```
run -v /Users/stuff:/path/container (mac/linux)
run -v //c/Users/stuff:/path/container (windows)
```

# LAB: BIND MOUNTING

We will setup apache server and bind mount document root i.e. /usr/local/apache2/htdocs/ to your host directory Run an apache server container:

```
docker run -it --name web-server -p 8080:80 -v
/host/path/website/:/usr/local/apache2/htdocs/ httpd:2.4
```

Create a html page on host path named index.html

```
vi /host/path/website/index.html
```

# LAB: BIND MOUNTING

Add following content

```
!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8"> <title>Learn Docker</title> </head>
<body>
<h1>Learn Docker With Us</h1>
</body>
</html>
```

# LAB: BIND MOUNTING

Next point your browser to localhost:8080/index.html and you should be presented with the page we created

Cleanup Container:

```
docker container stop web-server
```

```
docker container rm web-server
```

Go to the host directory that you mounted and list files, you should see index.html still there

```
cd /host/path/website/
ls
```

# Persistent Data: Volumes

- Override with docker run -v /path/in/container
- Bypasses Union File System and stores in alt location on host
- Connect to none, one, or multiple containers at once
- By default they only have a unique ID, but you can assign name then it's a "named volume"

# LAB - PERSISTENT DATA

1. Upgrade postgres using Named Volume
2. Setup Apache Server using Bind Mounting

# LAB: VOLUME

We will persist data in Postgres and then delete and recreate again with new version and data will be persisted

Create named volume:

```
docker volume create psql
```

List volumes:

```
docker volume ls
```

# LAB: VOLUME

Run postgres version 13 container:

```
docker container run --rm -it --net psql --name postgres1 -e
POSTGRES_PASSWORD=testpwd  -v psql:/var/lib/postgresql/data
postgres:13
```

Check logs and verify if postgres is configured, now stop & remove the container

```
docker container stop postgres1
docker container rm postgres1
```

# LAB: VOLUME

Now run postgres version 13.4 container:

```
docker container run --rm -it --net psql --name postgres2 -e
POSTGRES_PASSWORD=testpwd  -v psql:/var/lib/postgresql/data
postgres:13.4
```

Check logs and see the db should be already configured, now stop & remove the container & volume

```
$ docker container stop postgres2
$ docker container rm postgres2
$ docker volume rm psql
```

# Docker Images

- Images are made up of series of layers and are immutable.
- New layer is built on every application update.
- Can be created from scratch (custom image).
- Can be downloaded via any registry (e.g. Docker Hub).
- Every image extends from a base image. (centos, ubuntu..)
- Every instruction of a Dockerfile is converted to a layer.
- Images are essentially a snapshot of a container that are then used to base containers upon.

# Docker Images

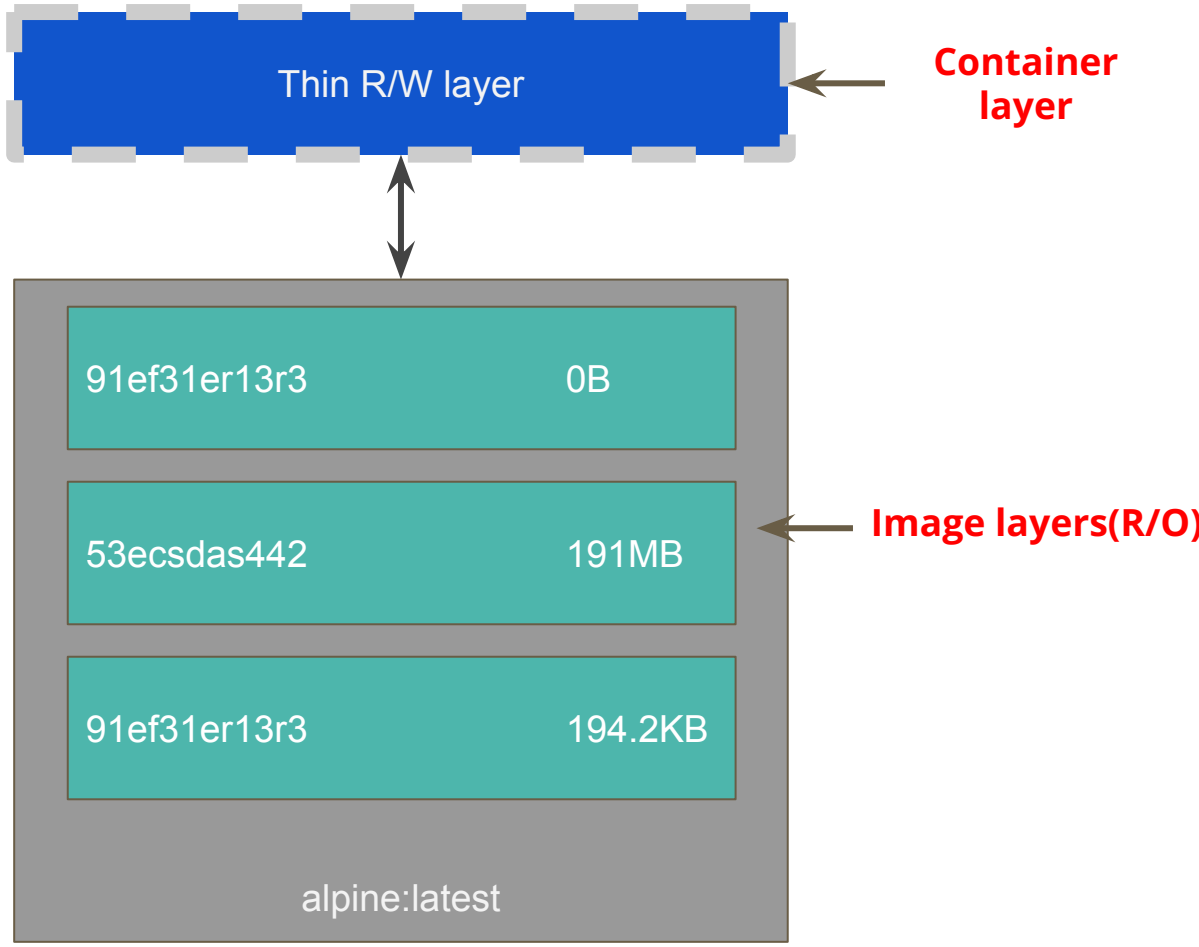
- An image is simply a runnable component with filesystem.
- Docker image is a file composed of multiple layer, used to execute code in docker container. It becomes one or more instances of that containers.
- Layers in docker images are just tar files, can be shared b/w images.

# Docker Containers

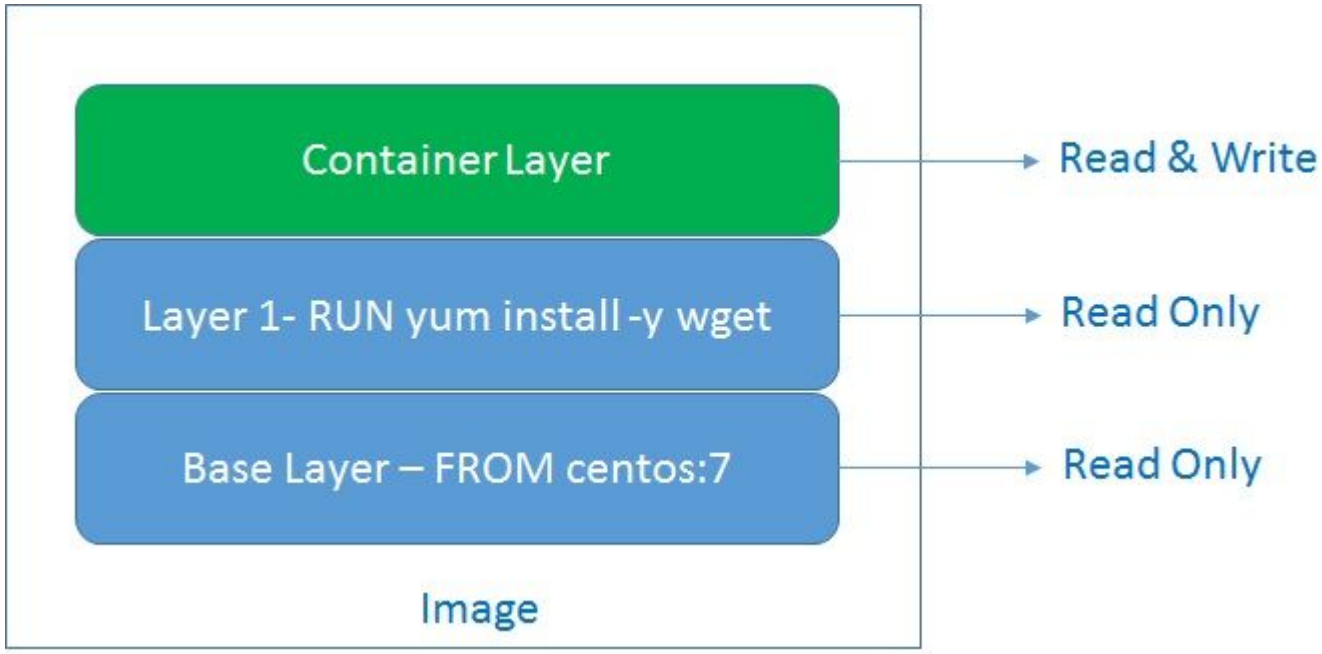
- Runnable instance of an image, created when we start an image with '**run**' command.
- When a container is created a thin read/write layer is put on top of image, Any changes made to the container is put on this read/write layer during the lifetime of the container.
- Can be started, stopped or even deleted (read/write layer is removed) at any time.
- We can start multiple containers of same image and each container runs in complete isolation which means each container maintains its own data safely on the top of the Docker Images as each has its own read/write layer
- You can enter in to container using docker client API.

# Container vs Image

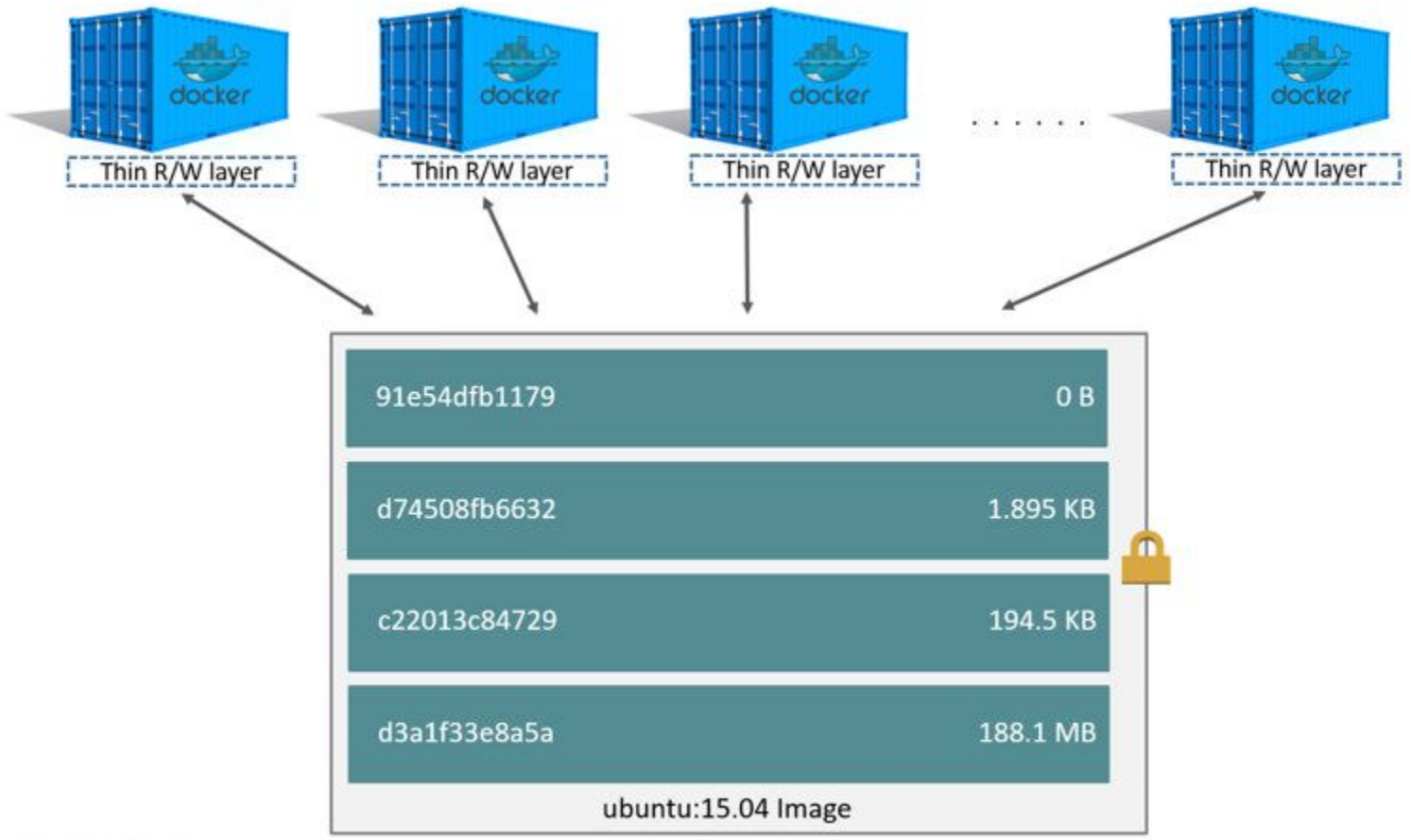
- Image: A packaged form of an application
- Container: Running form of an Image



# Layers for Image



# Multiple Containers from Single Image

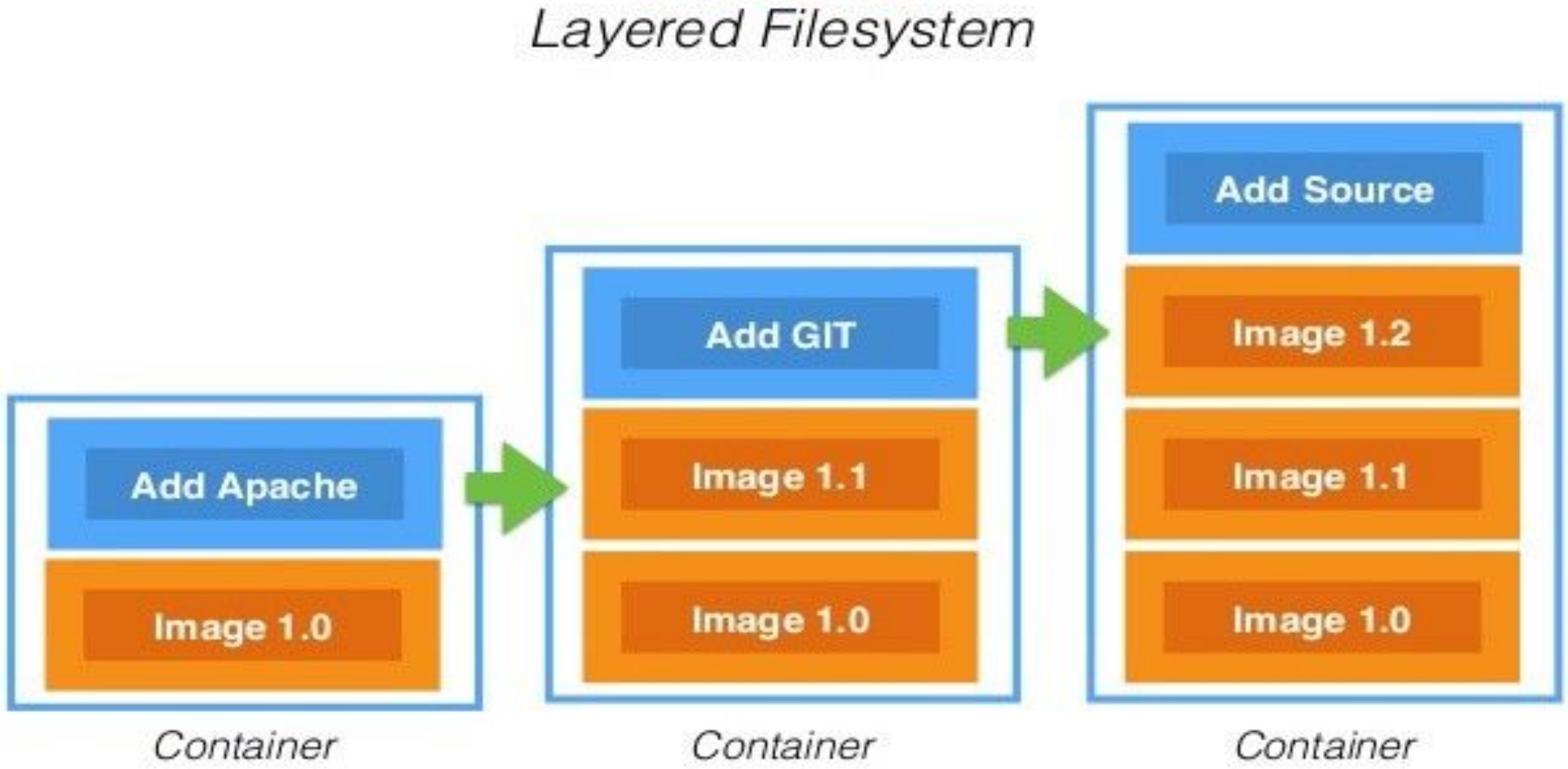


# Container vs Image

## Image


## Container

# Images based on other Images



# Docker Registry

- A central place to Store and Distribute docker images
- Can be publicly or privately hosted.
- Docker images can be pulled/pushed to these registries.
- Some popular hosted registries are
  - Docker Hub
  - AWS ECR
  - Google CR
  - Azure CR

# Docker Hub

- Cloud registry service for sharing application and automating workflows.
- **Features**
  - Unlimited Public and one free Private Docker repositories.
  - Official repositories.
  - Teams and organizations
  - Automated builds
  - Webhooks

# Lab

- Create a Docker Hub account.
  - https://hub.docker.com/signup
- Sign in using
  - **$ docker login**

# Dockerfile

- By default docker looks for a file name '**Dockerfile**' with no extension.
- A Dockerfile is text file that has all the instructions needed to build a docker image.
- It has a simple set of commands like CMD, FROM, RUN,ADD, ONBUILD, VOLUME, etc.
-

# Dockerfile

- Here's how it works
  - Create a Dockerfile
  - Build image using Dockerfile
  - Run a container using the image
  - Push to Docker Registry
  - Team can pull that image and test on their machine
  - Can deploy the image on sandbox environment
  - Test on sandbox
  - Deploy on Prod

# Dockerfile Commands



- **FROM:** Defines the base image.

- **RUN:** is an image build step, the state of container after a RUN command will be committed to docker image.
  - E.g. RUN apt update && apt upgrade

- **LABEL:** Any Labels/Metadata for the images

- **CMD:** Specifies what command needs to run by default while launching the container, can be overridden at time of container start.

- **ENTRYPOINT:** Specifies commands needed to be run at the start of the container, can't be overridden at the time of starting the container.

# Dockerfile Commands



- **WORKDIR:** Specifies working directory for RUN, CMD, COPY, ADD and ENTRYPOINT instructions.
- **EXPOSE**: exposes the local container port, on which container listens for connections.
- **ENV**: Sets the environment variable <key> to the value <value>
- **ARG**: 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

It's a best practice to set ENTRYPOINT as an image's main command and CMD to set default tags.

# Dockerfile commands (ADD vs COPY)

## ADD

- Lets you copy from multiple sources
  - Remote url
  - Can extract from source to destination directly

## COPY

- Lets you copy only from
  - Local file
  - Host machine where you're building images

# Lab: Add vs Copy

https://github.com/kahootali/docker-samples/tree/master/add-vs-copy

# Dockerfile commands (ENV vs ARG)

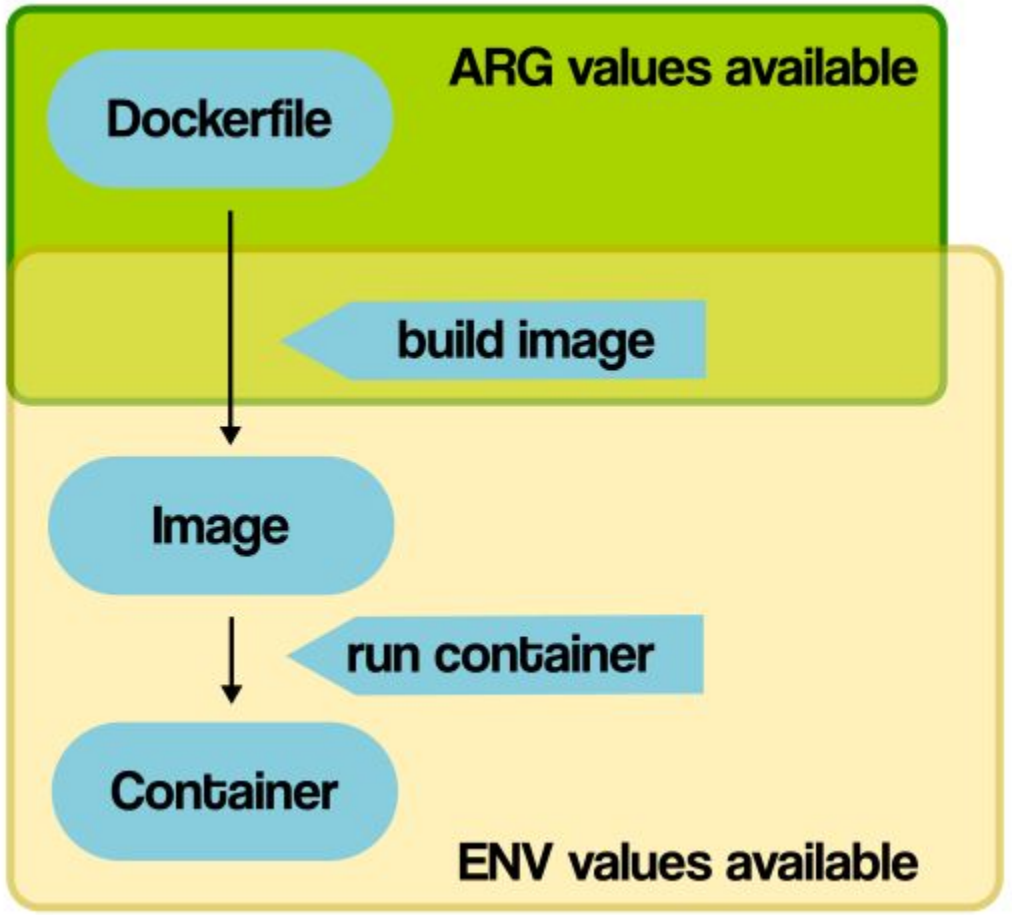
## ENV

- For default values for your future environment variables in your container
- Run time variable
- If not defined, so can be defined while running as well
- Used while docker run -e NAME=Ali Kahoot

## ARG

- For default values for variables during image build
- Build time variable
- A running container won't have access to an ARG variable value
- Used by docker build --build-arg NAME=Ali Kahoot

# ENV vs ARG


ARG values available
Dockerfile
build image
Image
run container
Container
ENV values available

# Lab: ENV vs ARG

https://github.com/kahootali/docker-samples/tree/master/arg-vs-env

# Dockerfile commands (ENTRYPOINT vs CMD vs RUN)

- RUN:  Build time command, if you want to run any command during the build of the image
- Entrypoint & CMD are runtime commands that are used to specify which command needs to run while running the container
- ENTRYPOINT: The executable to run while running the container
- CMD: If Entrypoint is not defined, CMD becomes the 1st command but if defined, CMD is passed as an argument to Entrypoint

# Dockerfile commands (ENTRYPOINT vs CMD)

If you want to run a command e.g. **echo Hello Ali Kahoot** for your ubuntu image.

- Entrypoint: “echo” CMD: ‘Hello Ali Kahoot’
- Entrypoint: ” CMD: echo Hello Ali Kahoot
- Entrypoint: echo Hello Ali Kahoot’ CMD: ”

# Lab: ENTRYPOINT vs CMD

https://github.com/kahootali/docker-samples/tree/master/entrypoint-vs-cmd

# Example DockerFile

## Dockerfile

```
FROM ubuntu:latest
COPY . /app
RUN make /app
CMD python /app/app.py
```

## Layers

**From** creates a layer from ubuntu docker image.

**COPY** add files from your Docker client's current directory.

**RUN** runs the command specified, here make builds your application with make

**CMD** specifies what command to be run within the container.

# Building custom Docker images

- Docker build command builds the docker image with specified tag using instructions defined in the Dockerfile and use local cache if image already exists.

**Usage:** $ docker build –t <yourImageTagName>  <PathToDockerFile>

# Best Practices for creating an image

- Make the size of the image as low as possible
- Make the build time of the image as less as possible
- Make the image as secure as possible

# Best Practices for creating an image

- Avoid including unnecessary files e.g copy all files
- Use .dockerignore
- Don’t install unnecessary packages
- Decouple applications
- Use official images when possible
- Minimize the number of layers
- Sort multi-line arguments
- Leverage build cache
- Use multi-stage builds

# Lab - Best Practices for Dockerfile

- Create a simple python web server.
- Create a docker file
- Build an image from Dockerfile
- Check newly created image
- Run your container
- Test Application

# Lab - Best Practices for Dockerfile - Leverage Cache

- Create a simple python web server.
- Create a docker file

These files are present at

https://github.com/kahootali/docker-samples/tree/master/bad-practices-images/cached-layers

# Lab - Best Practices for Dockerfile - Leverage Cache

- Build an image from Dockerfile

```
docker build -t diceanalyticsweek3lab1 .
```

- Check newly created image

```
docker images
```

- Run your container

```
docker run -d --name diceanaylticsweek3lab -p 8082:8000 diceanaylticsweek3lab1
```

- Test Application

  Open localhost:8082 on your browser to test your application.

/kahootali

# Lab - Best Practices for Dockerfile - MultiStage Build

These files are present at

https://github.com/kahootali/docker-samples/tree/master/bad-practices-images/multi-stage-builds

# Docker Tags

- Conveys useful information about specific image version/variant.

- We can either assign tag during an image build

```
docker build -t kahootali/myImage:1.0 .
```

, or can explicitly use **'tag'** command

```
docker tag existingImage kahootali/myImage:1.0
```

- Alias to ID of your image, which helps easy identification of images.

# Build Images from Container

- Really handy when you're working out how an image should be constructed.
- You want an image, but not sure of the commands of how to build it
- You can just keep tweaking the container until it works like you want.
- docker container commit command is used for this purpose.

```
docker container commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

# Lab - Docker Images

- Run container and make changes
- Build image from a container.
- Tag newly created image.
- Login to your Docker Hub account.
- Push image to Docker Hub.

# Lab - Docker Images

- Run container and make changes

```
docker run -it ubuntu bash

cat <<EOF >> /home/file.txt

Name: Muhammad Ali Kahoot

Container: Diceanalyticslab

EOF
```

- Build image from a container

```
docker commit  <CONTAINER_ID>  image-from-container
```

# Lab - Docker Images

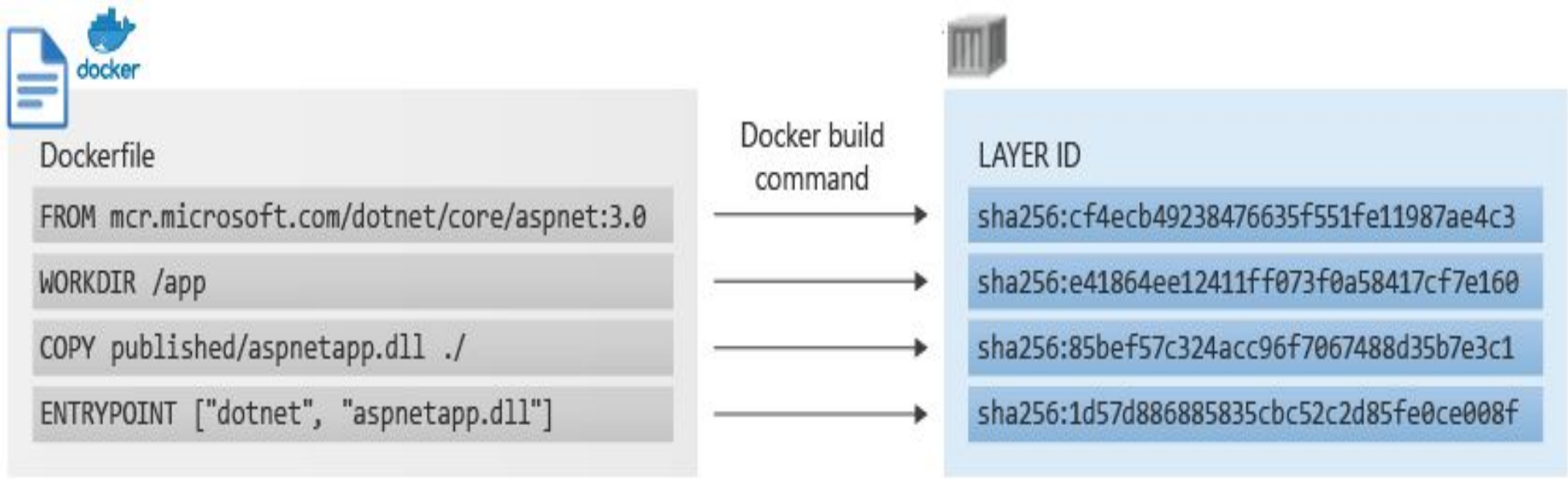
- Tag newly created image
  `docker tag <IMAGE_ID> <yourDockerHubUsername>/<repoName>:<TAG>`

  `E.g docker tag 4ffda6c9a8e3 kahootali/diceanalytics:v1.0`

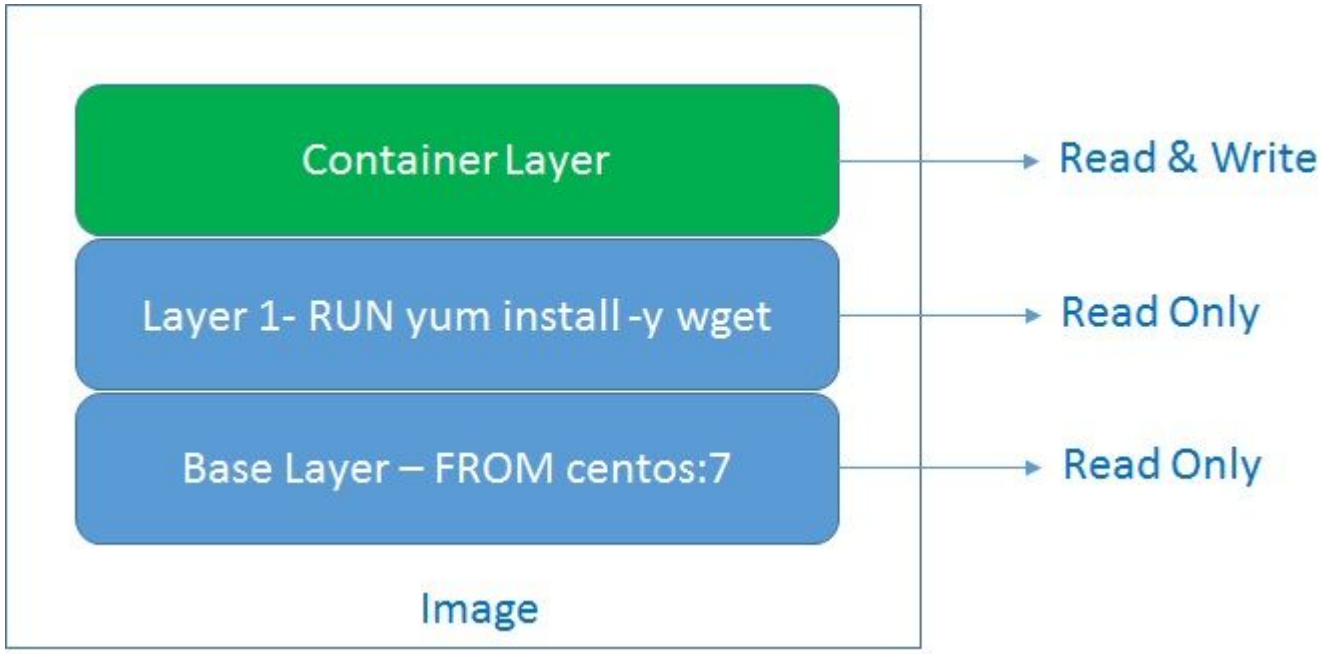
- Login to your Docker Hub account
  `docker login`

- Push image to Docker Hub
  `docker push <yourDockerHubUsername>/<repoName>:<TAG>`
  `docker push kahootali/diceanalytics:v1.0`

# Layers for Image



# Layers for Image



# Lab - Dockerizing Applications

We have an application written in different languages which we will dockerize. Almost all of them provide 2 endpoints

- /hello: Prints Hello World
- /count: Prints count of the endpoint visited

We have a sample repo with this applications in Dockerized form and we will be running them so first star & then fork the Repo

https://github.com/kahootali/docker-samples

# Lab - Dockerizing Applications

- Dockerize Golang App
- Dockerize Java App
- Dockerize Node App
- Dockerize Node Express App
- Dockerize Python App

# Lab - Dockerizing Go Application

- Dockerize Golang App

https://github.com/kahootali/docker-samples/tree/master/golang-app

# Lab - Dockerizing Go Application

```
FROM golang:alpine as build
WORKDIR /app
COPY main.go ./
RUN go env -w GO111MODULE=auto
RUN go build main.go

FROM alpine
LABEL name="Golang Application" \
      maintainer="Ali Kahoot <kahoot.ali@outlook.com>" \
      summary="A Golang Sample application"
EXPOSE 8080
WORKDIR /app
COPY --from=build ./app/main ./
CMD ["./app/main"]
```

# Lab - Dockerizing Go Application

```
docker build -t golang-app .
docker run -it --rm --name golang-app --init -p 8080:8080 golang-app
```

Go to browser and access

```
localhost:8080/hello
localhost:8080/count
```

Now press CTRL+C to exit the container, Now we will try to override the CMD command.

# Lab - Dockerizing Go Application

```
docker run -it --rm --name golang-app --init -p 8080:8080 golang-app sh
```

Now you will see that the golang app didn't start rather a shell has started and you are attached to it. Now you can run any command that a shell can run. Now we will run the golang app from the shell. Run

```
./app
```

Access the browser again and application should be working. Now press CTRL+C to stop the application, But the container will still be running. Now press exit to exit from the container.

# Lab - Dockerizing Java Application

- Dockerize Java App

https://github.com/kahootali/docker-samples/tree/master/java-app

# Lab - Dockerizing Java Application

```
FROM maven:3.8.1 as build
COPY src /usr/src/app/src
COPY pom.xml /usr/src/app
RUN mvn -f /usr/src/app/pom.xml clean package -Dmaven.test.skip=true

FROM gcr.io/distroless/java:11
LABEL name="Java Application" \
     maintainer="Ali Kahoot <kahoot.ali@outlook.com>" \
     summary="A Java Spring Boot application"
WORKDIR /app
EXPOSE 8080
COPY --from=build /usr/src/app/target/*.jar artifacts/app.jar
CMD ["artifacts/app.jar"]
```

# Lab - Dockerizing Java Application

```
docker build -t java-app .
docker run -it --rm --name java-app --init -p 8081:8080 java-app
```

Go to browser and access

```
localhost:8081/hello
localhost:8081/count
```

We are using distroless image here, so it will not have a shell available so you can't exec into it, as the purpose is to have very small and more secure image.

# ENTRYPOINT vs CMD

As we are using the distroless image of Google so its ENTRYPOINT command is

```
java -jar
```

And in CMD we are passing ["artifacts/app.jar"] so it will be passed as an argument to ENTRYPOINT command. So thats how your application will run as the complete command will be

```
java -jar ["artifacts/app.jar"]
```

# ENTRYPOINT vs CMD

Now we will try to override the ENTRYPOINT & CMD and run again

```
docker run -it --rm --name java-app --init --entrypoint java -p 8081:8080
java-app -jar artifacts/app.jar
```

Now the application has started again but you can see this time -jar artifacts/app.jar(CMD) is passed as argument to java(ENTRYPOINT) and full command again becomes

```
java -jar ["artifacts/app.jar"]
```

# Lab - Dockerizing Python Application

- Dockerize Python App

https://github.com/kahootali/docker-samples/tree/master/python-app

# Lab - Dockerizing Python Application

```
FROM python:3.6-alpine
LABEL name="Python Application" \
      maintainer="Ali Kahoot <kahoot.ali@outlook.com>" \
      summary="A Sample Python application"
WORKDIR /app
EXPOSE 8080
RUN pip install flask
COPY app.py ./
CMD [ "python", "./app.py" ]
```

# Lab - Dockerizing Python Application

```
docker build -t python-app .
docker run -it --rm --name python-app --init -p 8082:8080 python-app
```

Go to browser and access

```
localhost:8082/hello
localhost:8082/count
```

# Lab - Dockerizing Node Application

- Dockerize Node App

https://github.com/kahootali/docker-samples/tree/master/node-app

# Lab - Dockerizing Node Application

```
FROM node:10-alpine
LABEL name="Node Application" \
      maintainer="Ali Kahoot <kahoot.ali@outlook.com>" \
      summary="A Node Sample application"
# Create app directory
WORKDIR /app
EXPOSE 8080
COPY index.js ./
CMD [ "node", "index.js" ]
```

# Lab - Dockerizing Node Application

```
docker build -t node-app .
docker run -it --rm --name node-app --init -p 8083:8080 node-app
```

Go to browser and access

```
localhost:8083/
```

# Lab - Dockerizing Node with Express Application

- Dockerize Node Express App

https://github.com/kahootali/docker-samples/tree/master/node-express-app

# Lab - Dockerizing Node Application

```
FROM node:10-alpine
LABEL name="Node Express Application" \
      maintainer="Ali Kahoot <kahoot.ali@outlook.com>" \
      summary="A Node Express application"
WORKDIR /app
EXPOSE 8080
COPY package*.json ./
RUN npm install
COPY server.js ./
CMD [ "npm", "start" ]
```

# Lab - Dockerizing Node Application

```
docker build -t node-express-app .
docker run -it --rm --name node-express-app --init -p 8084:8080

node-express-app
```

Go to browser and access

```
localhost:8084/hello

localhost:8084/count
```