/kahootali

DevOps - Week 4 - Docker Compose & Swarm

Muhammad Ali Kahoot
Dice Analytics

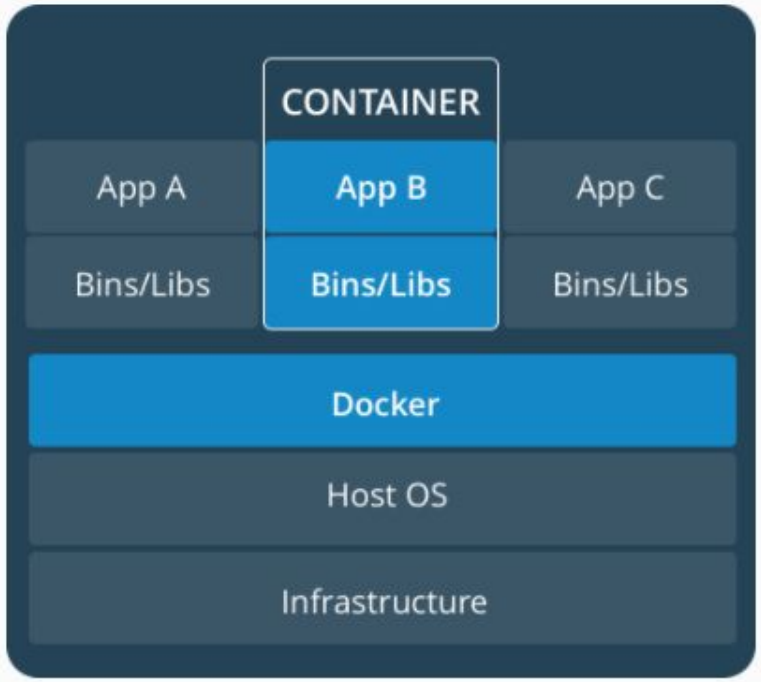
QUIZ

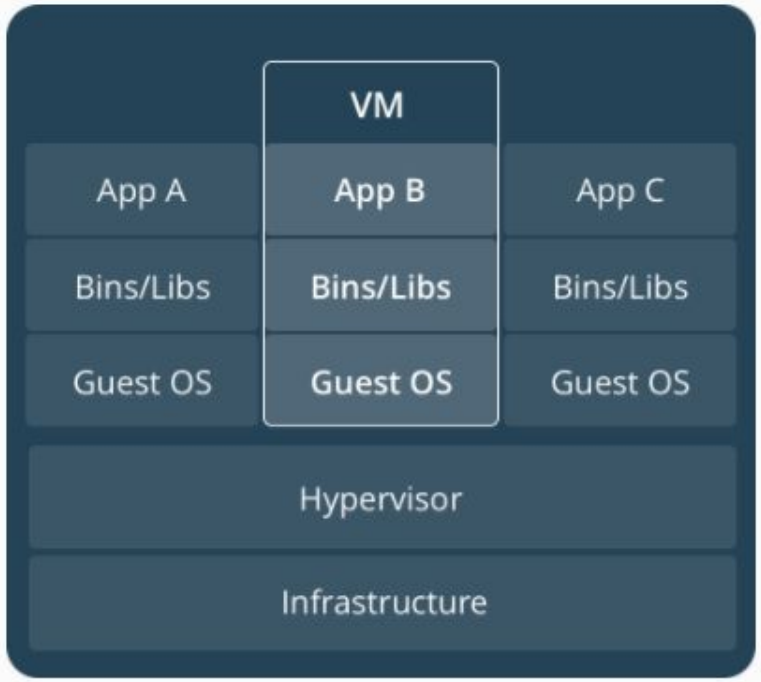
- Write differences between VMs & Containers
- How to persist data in Containers
- Why and how are networks used in Docker
- What are Docker Images
- Docker Images vs Containers
- What are layers in Docker Images
- RUN vs CMD vs ENTRYPOINT
- ENV vs ARG
- Best practices of creating an Image

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





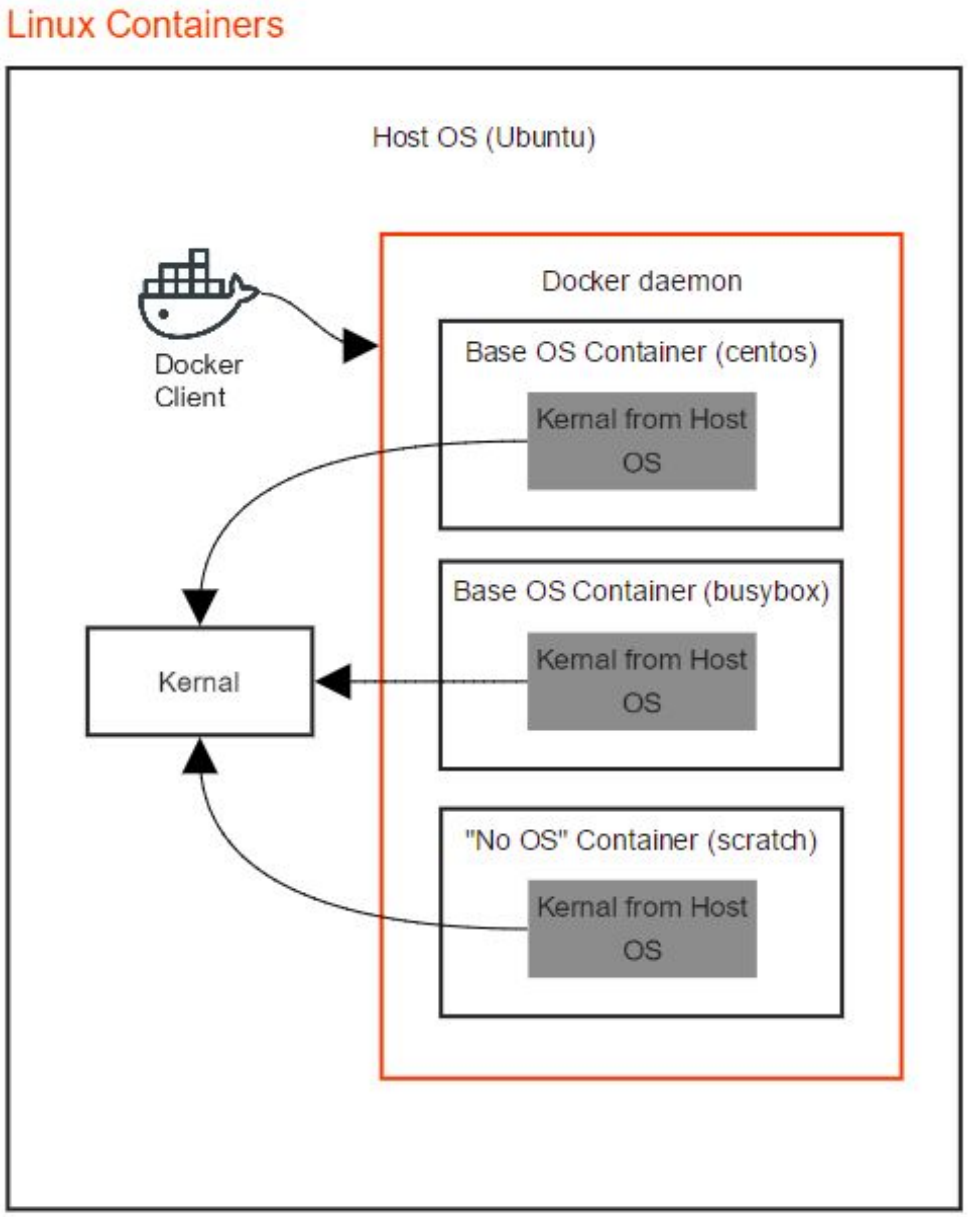
CONTAINERS

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.

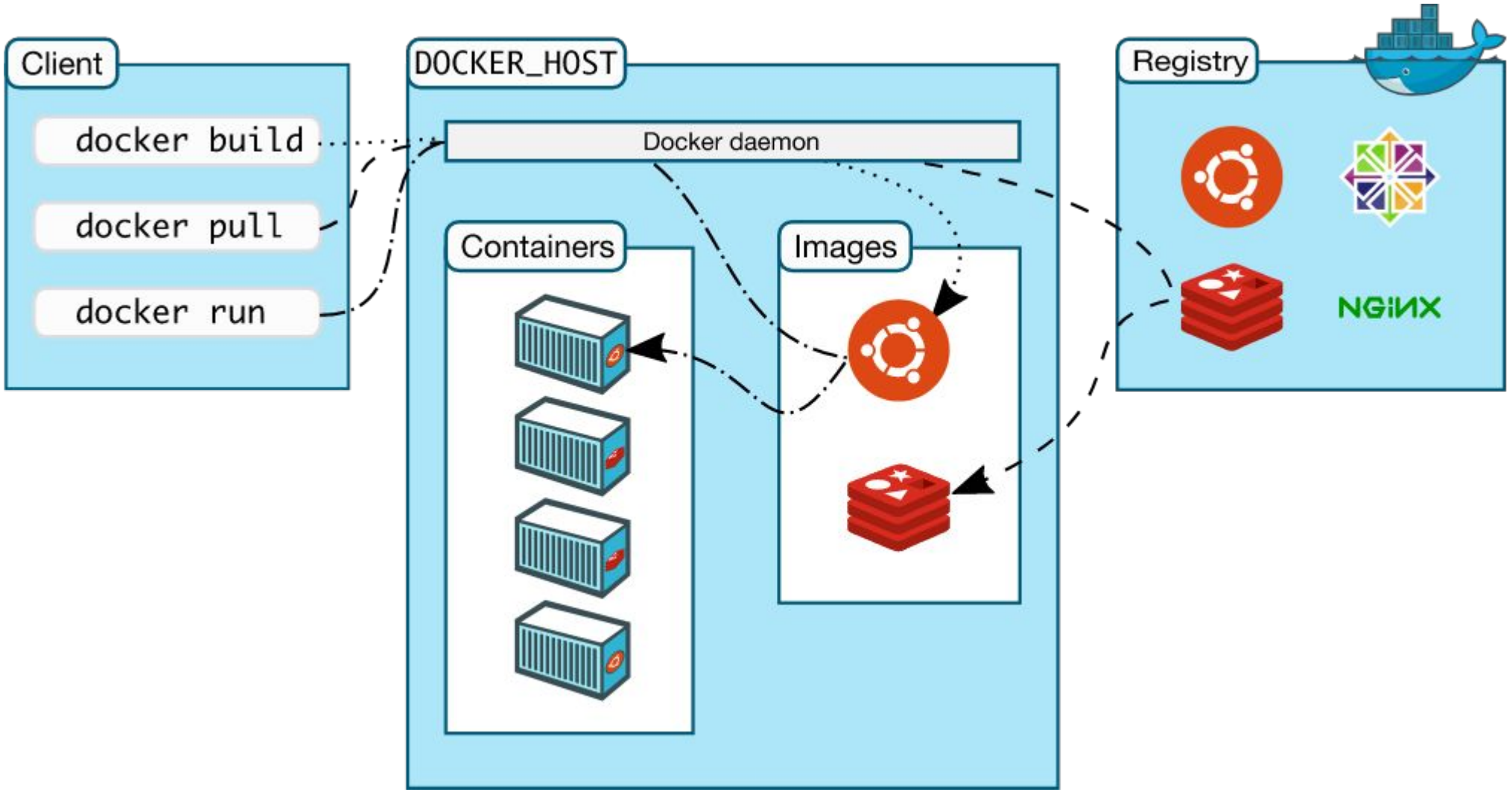
VIRTUAL MACHINES

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

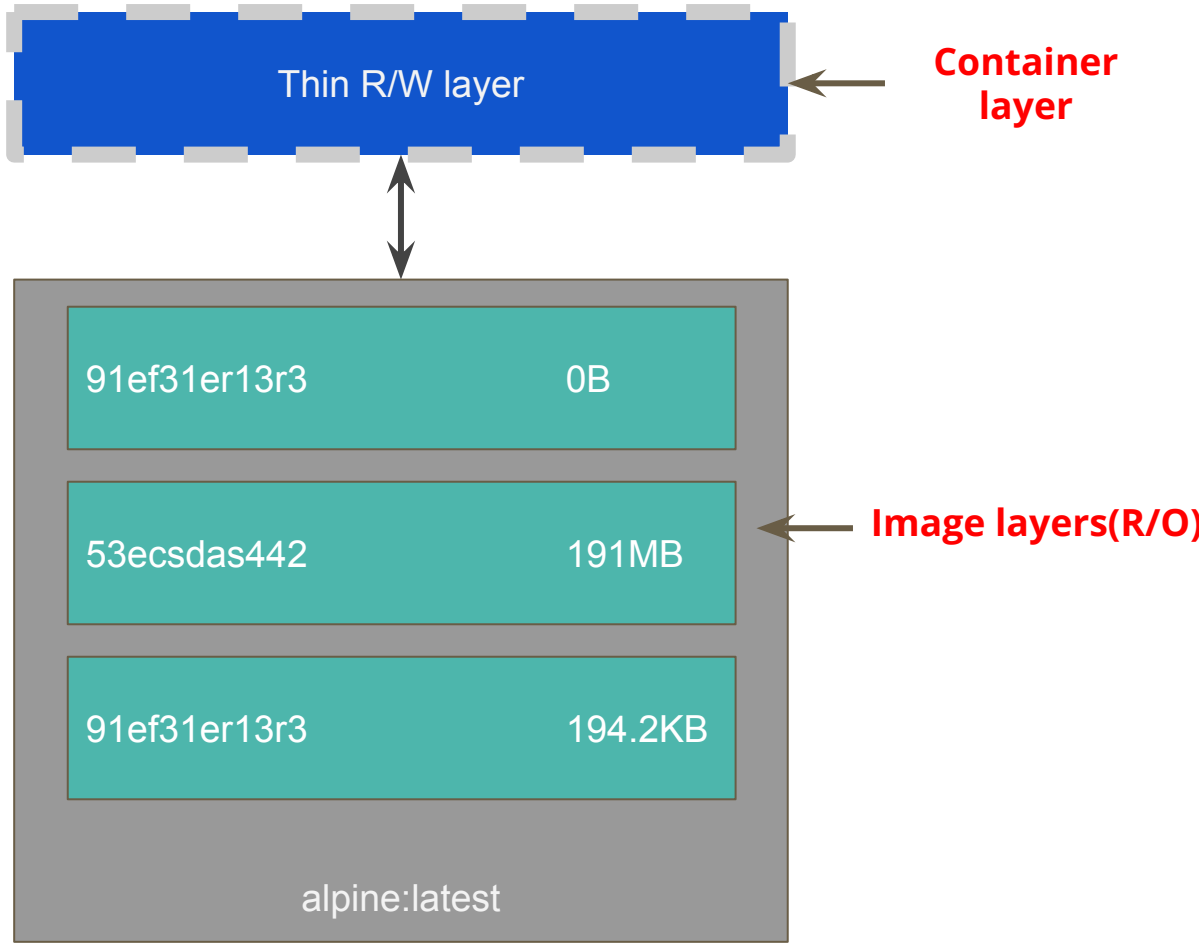

Linux Containers
Host OS (Ubuntu)
Docker daemon
Docker Client
Base OS Container (centos)
Kernal from Host OS
Base OS Container (busybox)
Kernal from Host OS
Kernal
"No OS" Container (scratch)
Kernal from Host OS

Docker Architecture

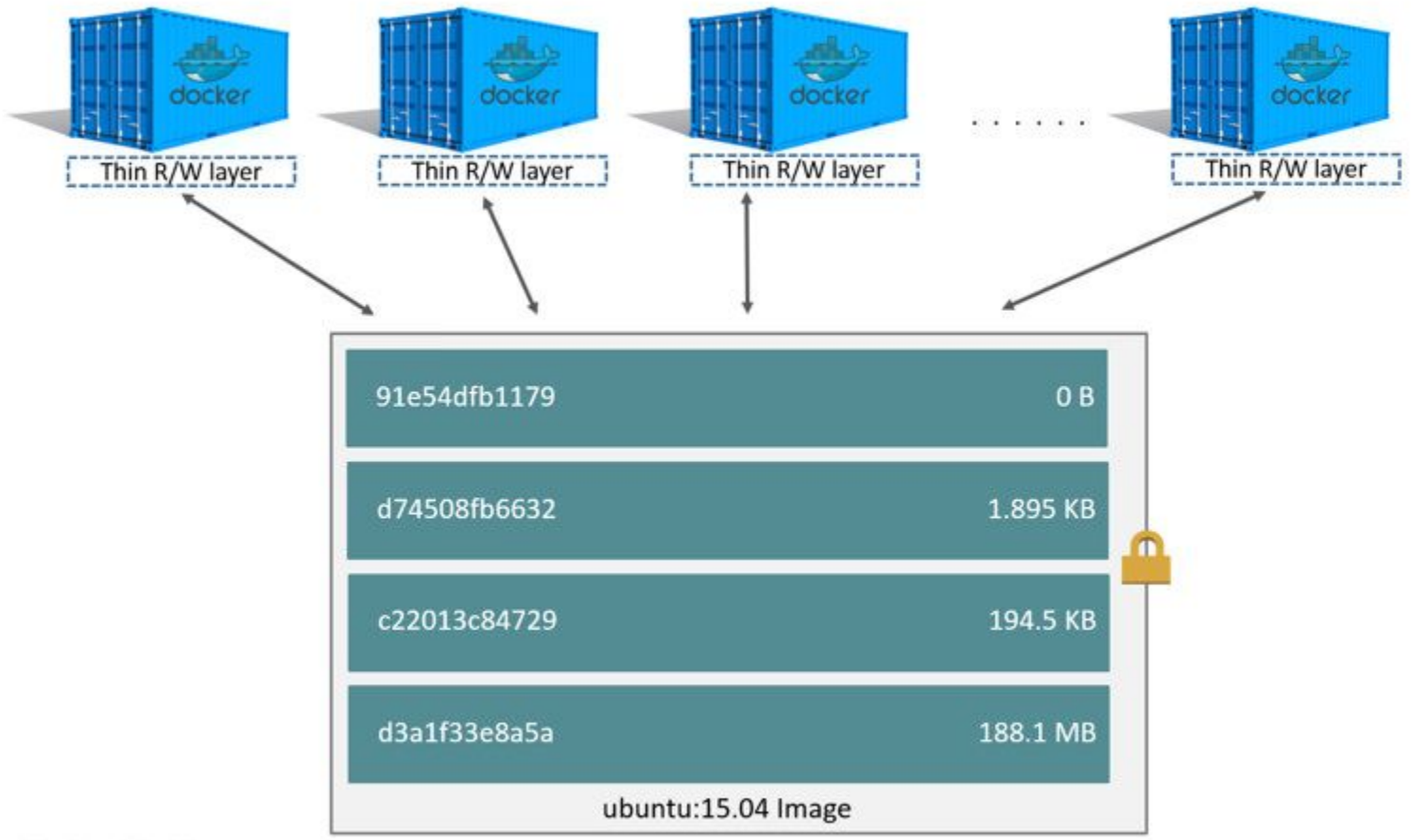


Container vs Image

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



Multiple Containers from Single Image

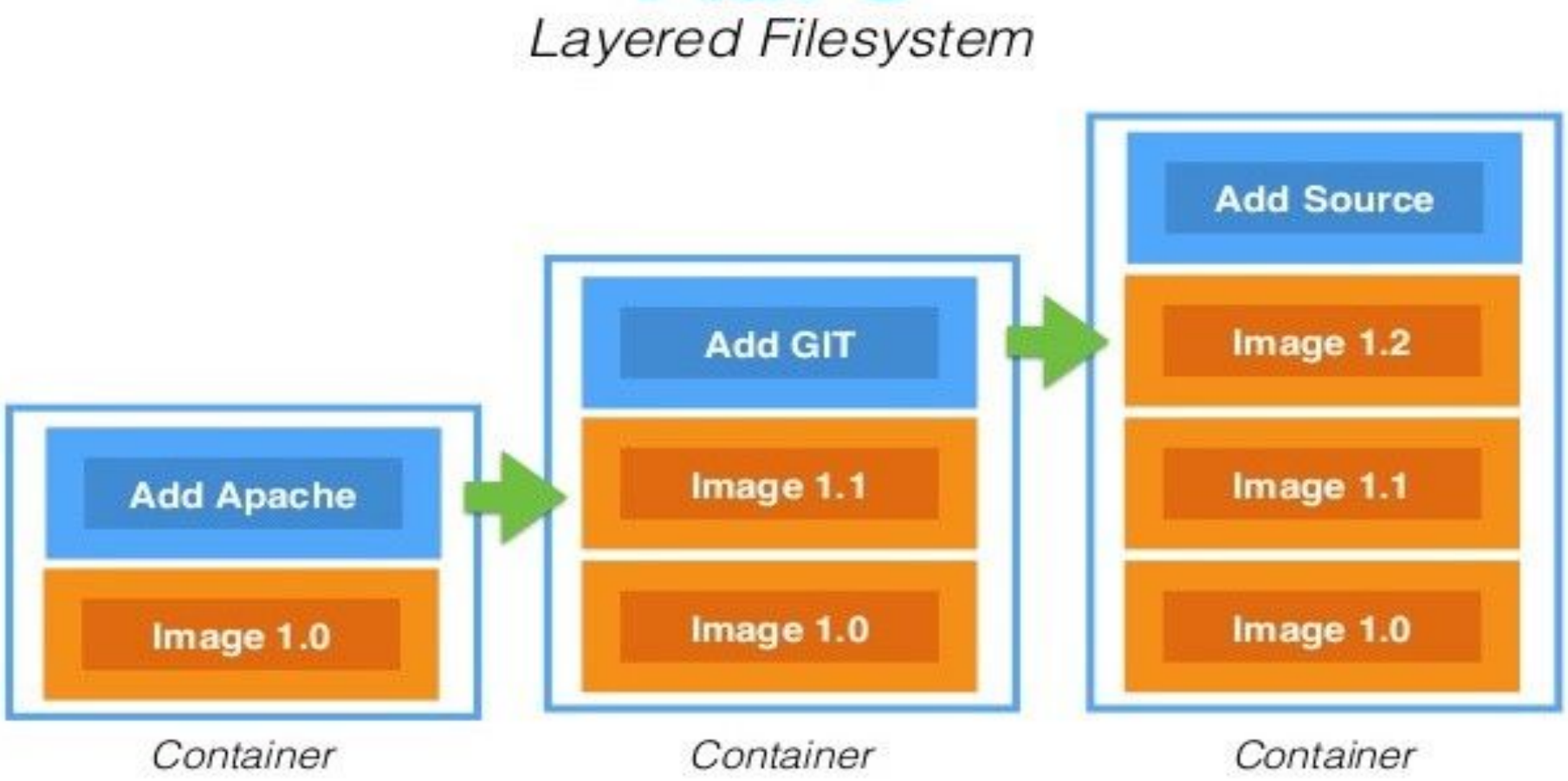


Container vs Image

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

Lab

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

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

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

Best Practices for creating an image

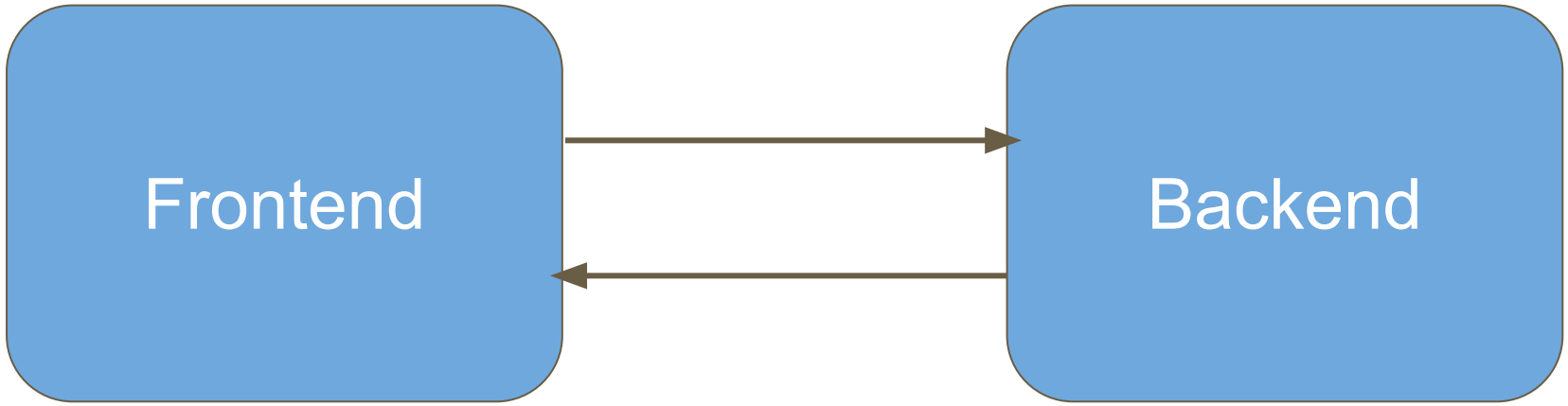
- Avoid including unnecessary files e.g copy all files
- Use .dockerignore
- Don’t install unnecessary packages
- Decouple applications
- Use official images when possible
- Minimize the number of layers
- Sort multi-line arguments
- Leverage build cache
- Use multi-stage builds

EXAMPLE: MYSQL

Go to https://hub.docker.com/_/mysql

and walkthrough the examples.

Dockerizing Microservices



Dockerizing Microservices

/hello
/instructor/id: Hello and call backend/ instructor/id
/student/id: Hello and call backend/student/id

/hello
/instructors
/instructor/id
/students
/student/id

Lab - Running Dockerized Microservices

- Dockerize Python Backend App
- Dockerize Node Frontend App
- Communicate with each other
- Create Network & Deploy apps in network

Lab - Running Dockerized Microservices

- Dockerize Python Backend App

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

- Dockerize Node Frontend App

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

Lab - Dockerize Python Backend App

```
cd ../microservices/backend-python/
docker build -t backend-python .
docker run -it --rm --name backend --init -p 9090:8080 backend-python
```

Go to browser and access

```
localhost:9090/hello
localhost:9090/instructors
localhost:9090/instructor/1 or 2
localhost:9090/students
localhost:9090/student/{id}
```

Now keep the backend running and we will try to run the frontend in other terminal that calls the backend through API.

Lab - Dockerize Node Frontend App

Move to microservices/frontend-node folder. Check the server.js file

```
cd ../frontend-node/
docker build -t frontend-node .
docker run -it --rm --name frontend -e BACKEND_URL=localhost:9090 --init
-p 9091:8080 frontend-node
```

Go to browser and access

`localhost:9091/hello` `localhost:9091/instructor/{id}` `localhost:9091/student/{id}`

This will not work. Check the code of Frontend trying to call the backend and see the issue

Lab - Dockerize Node Frontend App

As containers cannot access your machines local applications. Press Ctrl + C. Lets connect to container's shell and try curl.

```
docker run -it --rm --name frontend --init -p 9091:8080 frontend-node sh
curl localhost:9090/instructors
```

But if you check the browser on localhost:9090/hello, the backend app is working. So one container cannot access other containers and host's other processes by default.

Lab - Creating Network and attaching applications

Exit from previous containers. Create a new Docker Network.

```
docker network create application
```

Run backend again and attach to network

```
docker run -it --rm --name backend --network application --init -p
9090:8080 backend-python
```

Change frontend env var from **localhost:9090** to **backend:8080**, so the url will become

```
http://backend:8080/instructor/${id}
```

Lab - Creating Network and attaching applications

Now run the container again and join network

```
docker run -it --rm --name frontend -e BACKEND_URL=backend:8080 --network
application --init -p 9091:8080 frontend-app
```

Now access in browser

```
localhost:9091/hello
localhost:9091/instructor/{id}    id=1/2
localhost:9091/student{id}        id=1/2/3/4/5
```

RESTART POLICY

Create a folder Restarts and create a shell script crash.sh with following content

```
#/bin/bash

sleep 30
exit 1
```

After 30 seconds the container will send a non-zero exit status i.e. a failed response. Create a Dockerfile

```
FROM ubuntu

ADD crash.sh /
CMD /bin/bash /crash.sh
```

RESTART POLICY

Build the image

```
docker build -t restart .
docker run -d --name test-restart restart
```

Check the status of container

```
docker container ls
```

And then check after 30 seconds

```
docker container ls -a
docker container rm test-restart
```

RESTART POLICIES

There are 4 restart policy to handle this situation

- No: Never restart the container (Default behavior)
- On-failure: Restart only in case of Failure (Non-zero Exit code)
- Always: Always Restart even in success or Failure, even when node restarts
- Unless-stopped: Same as always, but will not restart containers that were manually stopped

```
docker rm test-restart
docker run -d --name test-restart --restart always restart
```

RESTART POLICIES

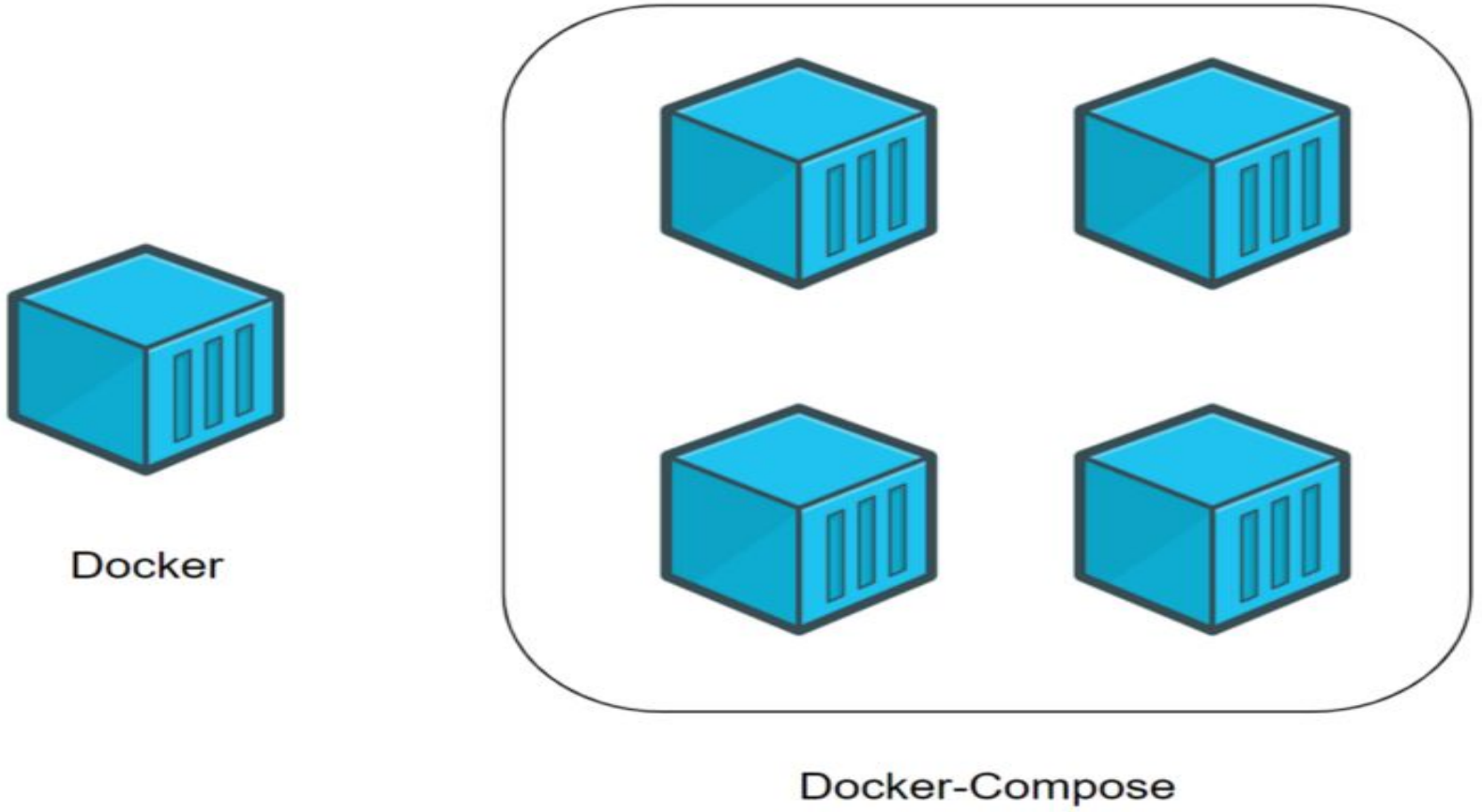
We will try with On-Failure now. Change the script to

```
#/bin/bash

sleep 30
exit 0
```

Run following commands

```
docker container rm test-restart

docker build -t restart .
docker run -d --name test-restart --restart on-failure:5 restart
```

DOCKER COMPOSE



DOCKER COMPOSE

- Tool for defining and running multi-container applications
- Compose contains information how to build and deploy containers
- You write a docker-compose.yml file to configure your applications services
- Single command to create and start all the services from - your configuration
- It is a YAML file

BENEFITS OF COMPOSE

- Multiple isolated environments on a single host
- Preserve volume data when containers are created
- Only recreate containers that have changed
- Variables and moving a composition between environments
- Infrastructure as Code
- Portability

DOCKER COMPOSE - SYNTAX

- Version: Species the Compose file syntax version
- Services: In Docker a service is the name for a - "Container in production".
- Networks: This section is used to configure networking - for your application.
- Volumes: Mounts a linked path on the host machine that - can be used by the container.

DOCKER COMPOSE - SERVICES

- Image: Sets the image that will be used to run the - container.
- Build: Can be used instead of image. Specifies the - location of the Dockerfile that will be used.
- Restart: Tells the container to restart if the system - restarts.
- Environment: Define environment variables to be passed - in to the Docker run

DOCKER COMPOSE - SERVICES

- Volumes: Mounts a linked path on the host machine that - can be used
- Depends_on: Sets a service as a dependency for the - current block-defined container
- Port: Maps a port from the container to the host in the - following manner

DOCKER COMPOSE - SERVICES

```yaml
version: '3'

services:

  cache:

    image: redis:alpine

  db:
    image: mysql:5.7
```

DOCKER COMPOSE - NETWORKS

```
version: '3'
services:
  cache:
    image: redis:alpine
    networks:
    - appnet
  db:
    image: mysql:5.7
    networks:
    - appnet
networks:
  appnet:
    driver: bridge
```

DOCKER COMPOSE - VOLUMES

```
version: '3'
services:
  mysql:
    image: mysql
    container_name: mysql
    volumes:
    - mysql:/var/lib/mysql
volumes:
  mysql:
```

DOCKER COMPOSE - COMMANDS

Now compose has been made part of docker CLI, so can use docker compose or docker-compose

docker-compose up: Create and start containers
docker-compose ps: List containers
docker compose build: Build or rebuild services
docker-compose start: Start services
docker-compose stop: Stop services
docker-compose restart: Restart services
docker-compose down: Stop and remove containers, - networks, images
docker-compose pull: Pull service images
docker-compose scale: Set number of containers for a - service

LAB - DOCKER COMPOSE COMMANDS

- Setup environment
- Create docker-compose file
- Create a compose service
- List containers by compose
- Stopping compose service
- Starting a compose service
- Restarting a compose service
- Deleting compose service

LAB - DOCKER COMPOSE COMMANDS

- Check files under docker-compose folder in the repo
- Check nginx/docker-compose.yml

LAB - DOCKER COMPOSE COMMANDS

- Run these commands to see what they do

```
docker-compose up -d
docker-compose ps
docker-compose stop
docker-compose start
docker-compose restart
docker-compose down
```

LAB - BUILD IMAGES WITH DOCKER COMPOSE

- Check **httpd** folder

```
version: '3.3'
services:
  web:
    build: .
    ports:
    - "8080:80"
  redis:
 image: "redis:alpine"
```

LAB - BUILD IMAGES WITH DOCKER COMPOSE

- Now with just one command, build the image and run the container

```
docker-compose up
```

- But this will not build after it, it will reuse the image despite change in code

```
docker-compose build
```

Lab: Wordpress

Check **Wordpress** folder to see working of an actual app

Lab: Env Vars in Compose file

Check **env-vars** folder to see how we can pass environment variables to containers through files

Lab: Network Restriction

Check **centos** folder to see how we can restrict containers to join networks

Lab: Microservices

- Now we will deploy the microservices we created using docker-compose
- Check **microservices** folder in parent directory
- Run

```
docker-compose build
docker-compose up
```

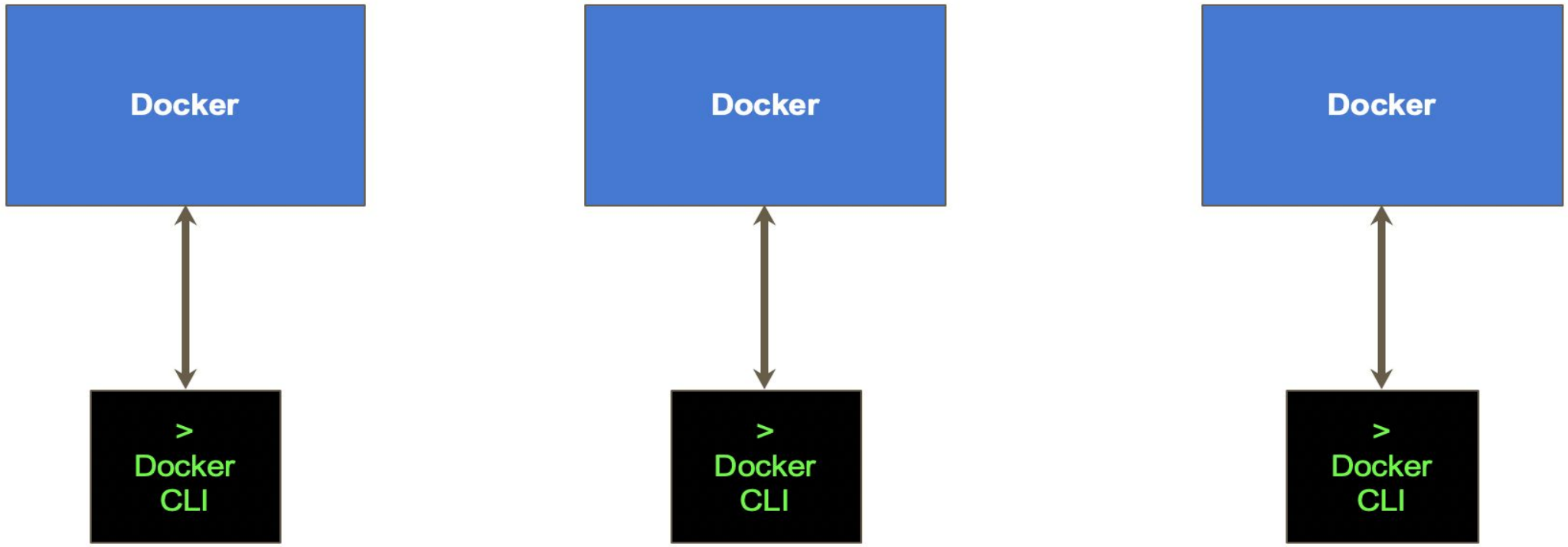
Lab: Scale Microservices

- Now we will scale the microservices
- Run

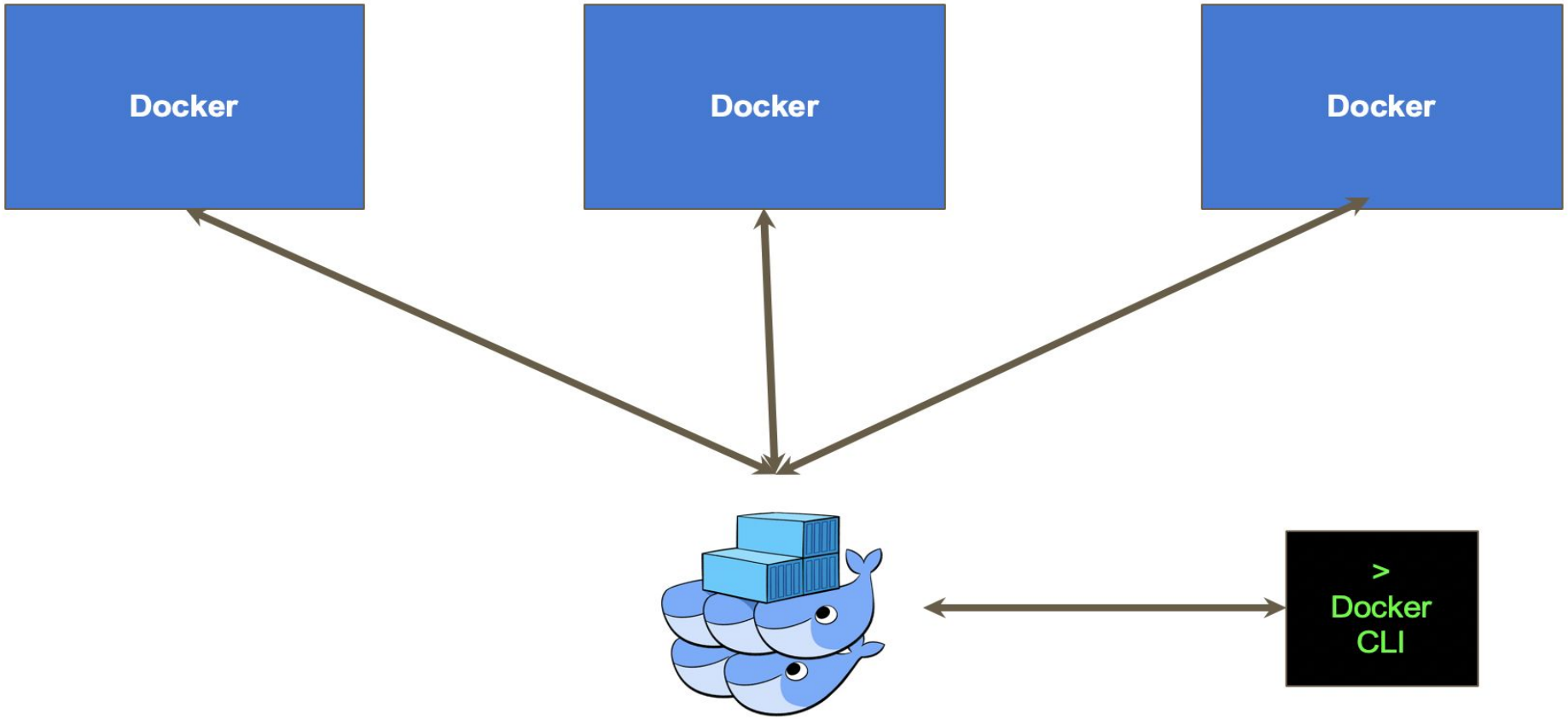
```
docker-compose scale backend=2
docker-compose scale frontend=2
```

- Frontend will give error

Docker Swarm



Docker Swarm



DOCKER SWARM

- Clustering and scheduling tool for containers
- Helps to manage cluster of docker nodes as a single - virtual system
- Manager and worker nodes
- Cluster of docker engine or nodes
- Two major components of Swarm
  - Cluster (can consist of one or more docker nodes)
  - Orchestration engine

DOCKER SWARM CONCEPTS

- Swarm: a swarm consists of multiple Docker hosts which - run in swarm mode and act as managers and workers

- Nodes: a swarm node is an individual Docker Engine - participating in the swarm

DOCKER SWARM CONCEPTS

- Task: the swarm manager distributes a specific number - of tasks among the nodes. A task carries a Docker - container and the commands to run inside the container

- Service: definition of the tasks to execute on the - manager or worker nodes.

DOCKER SWARM INIT

Run to initialize a swarm mode

```
docker swarm init
docker node ls
```

SWARM SERVICES

- A service is definition of tasks to execute on manager - or worker nodes
- Primary root of user interaction in Swarm
- We need to specify image, and which commands to execute - inside running container at time of creating a service

SWARM SERVICES COMMANDS

- create: Creates a new service
- inspect: Displays detailed information on one or more - services
- logs: Fetch logs of a services/task
- ls: list services
- ps: List tasks of one or more services
- rm : removes one or more services
- rollback: revert changes to a service conguration
- scale: scales one or multiple replicated services
- update: updates a service

LAB: SWARM SERVICES

- Run

```
docker service create -d --name nginx_service -p 8080:80 --replicas 2
nginx:latest
docker service ls
docker service scale nginx_service=3
```

SWARM STACK

- Used to manage a multi-service application
- Group of services that are deployed together.
- Method of using compose files to run an application

SWARM STACK COMMANDS

- deploy: Deploy a new stack or update an existing stack
- ls: List stacks
- ps: List the tasks in the stack
- rm: Remove one or more stacks
- services: List the services in the stack

LAB: DOCKER SWARM STACK

Go to folder docker-compose/wordpress and run

```
docker stack deploy --compose-file docker-compose.yml wordpress
docker service ls
```

This will deploy the wordpress stack in the docker swarm

LAB: DOCKER SWARM STACK

Remove the stack

```
docker stack rm wordpress
```

DOCKER COMPOSE & SWARM

	Run through Commands	Run through Container as Code
Single Host	Containers `docker container run`	Compose Services `docker-compose up`
Multiple Host(Swarm)	Services `docker service create`	Stack Services `docker stack deploy --compose-file`

SUMMARY

/kahootali

Running Microservices	Issues	Benefits
Containers	• Port conflict • Separate commands • Networks • Volumes • Difficult to remember Commands(deviate)	
Compose Services	• If deleted replica manually, does not recreate • Single Node Limitations(Node down/ node resources full)	• Network, Volumes shared in a single file • Run different containers at once • Define dependencies • Replicas • Infra(Container) as Code • Load Balance • Changed container recreate only
Swarm Services	• Difficult to remember Commands(deviate) • Specify Image name	• Replicas & Auto recreate replicas • Load Balancing • Distributed architecture • Can have Different Nodes • Can Scale based on Load
Swarm Stacks		• Same as Compose Services • Services as Code • Multi Node

Useful Resources

- A simple blog to understand yaml: https://javascript.plainenglish.io/everything-you-need-to-know-about-yaml-files-5423358cc5c9

- Understand Json & Yaml: https://kodekloud.com/courses/json-path-quiz/lectures/11338834

Before Next Class

- Download jenkins.war from http://mirrors.jenkins.io/war-stable/latest/jenkins.war

  on your OS where you are running docker
- Install java on your machine
- If using Ubuntu
  - sudo apt-get install openjdk-8-jdk -y
- Confirm by running
  - java -version
- If using MacOS
  - brew install jenkins-lts