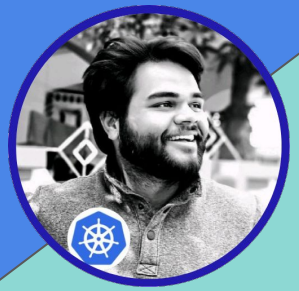


NINAD SAMUDRE

CLOUD ENGINEER AND TRAINER

CERTIFIED kubernetes APPLICATION DEVELOPER

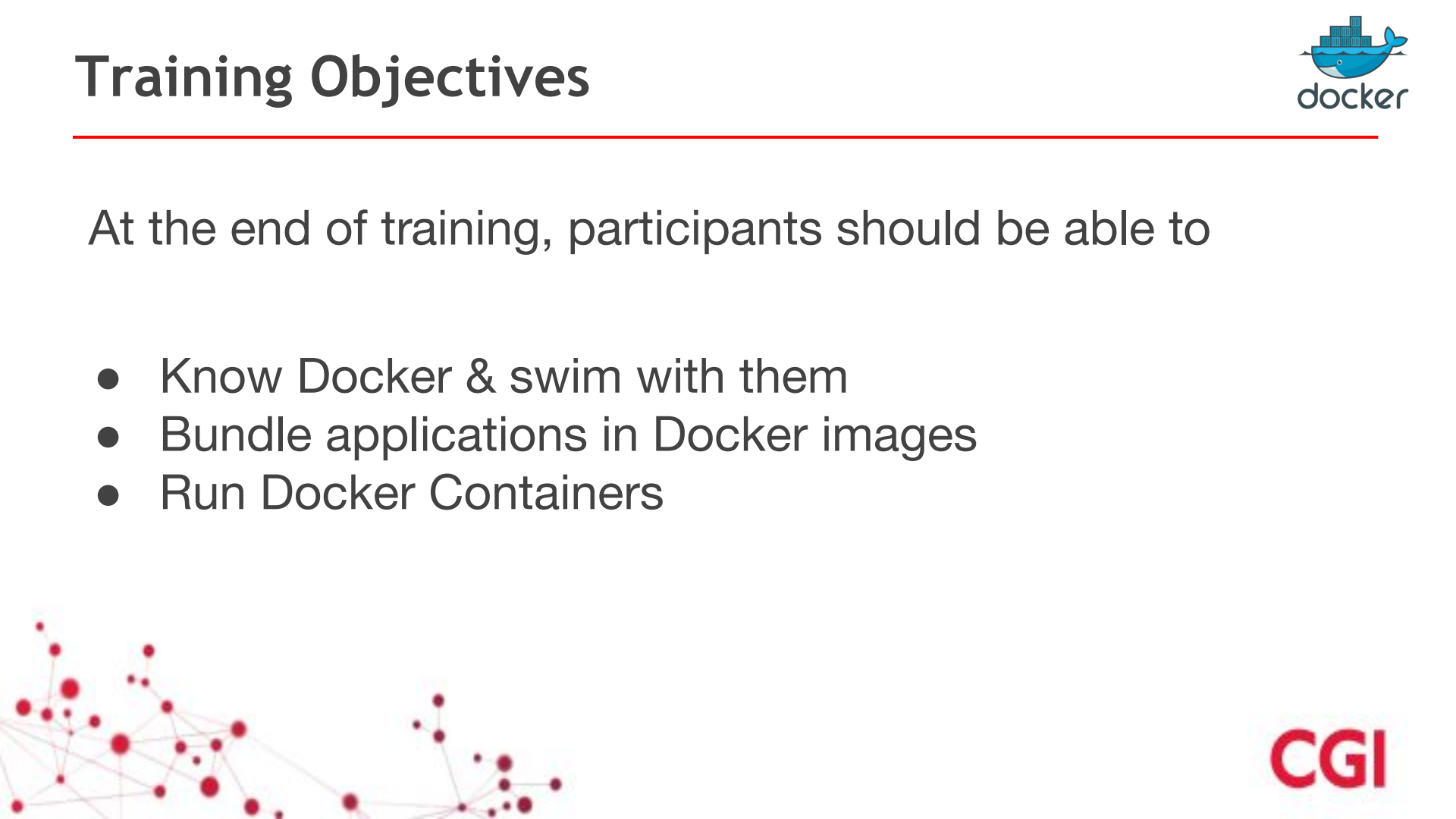
Docker and Kubernetes

docker

amazon web services

CGI

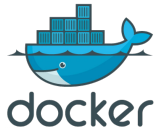
Training Objectives



At the end of training, participants should be able to

- Know Docker & swim with them
- Bundle applications in Docker images
- Run Docker Containers

Table of Contents



Module 1: Docker concepts and terms

- Intro
- Containerization vs Virtualization
- Terminologies in Docker world
- Docker Architecture
- Docker Machine
- Configuring Docker engine

Module 2: Docker Containers

- Creating containers
- Running containers
- Running containers in background
- Connecting containers
- Docker Images

Module 3: Provisioning Docker Image

- Introducing the Dockerfile
- Creating a Dockerfile
- Building images manually
- Building images using Continuous Integration tools
- Storing and retrieving Docker Images from Docker Hub
- Inspecting a Dockerfile from DockerHub

Module 4: Diving Deeper into Dockerfile

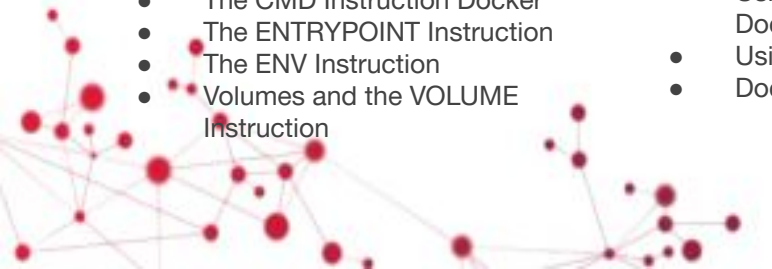
- The Build cache
- Dockerfile and Layers
- Building a Web Server Container
- The CMD Instruction Docker
- The ENTRYPOINT Instruction
- The ENV Instruction
- Volumes and the VOLUME Instruction

Module 5: Working with Registry

- Module Intro
- Creating a Public repo on Docker Hub
- Using our Public repo on Docker Hub,
- Using a Private Registry,
- Docker Hub Enterprise

Module 6: Docker Networking

- Module Intro
- The docker0 Bridge
- Virtual Ethernet Interfaces
- Network Configuration Files
- Exposing Ports
- Viewing Exposed Ports
- Linking Containers
- Lab Exercises

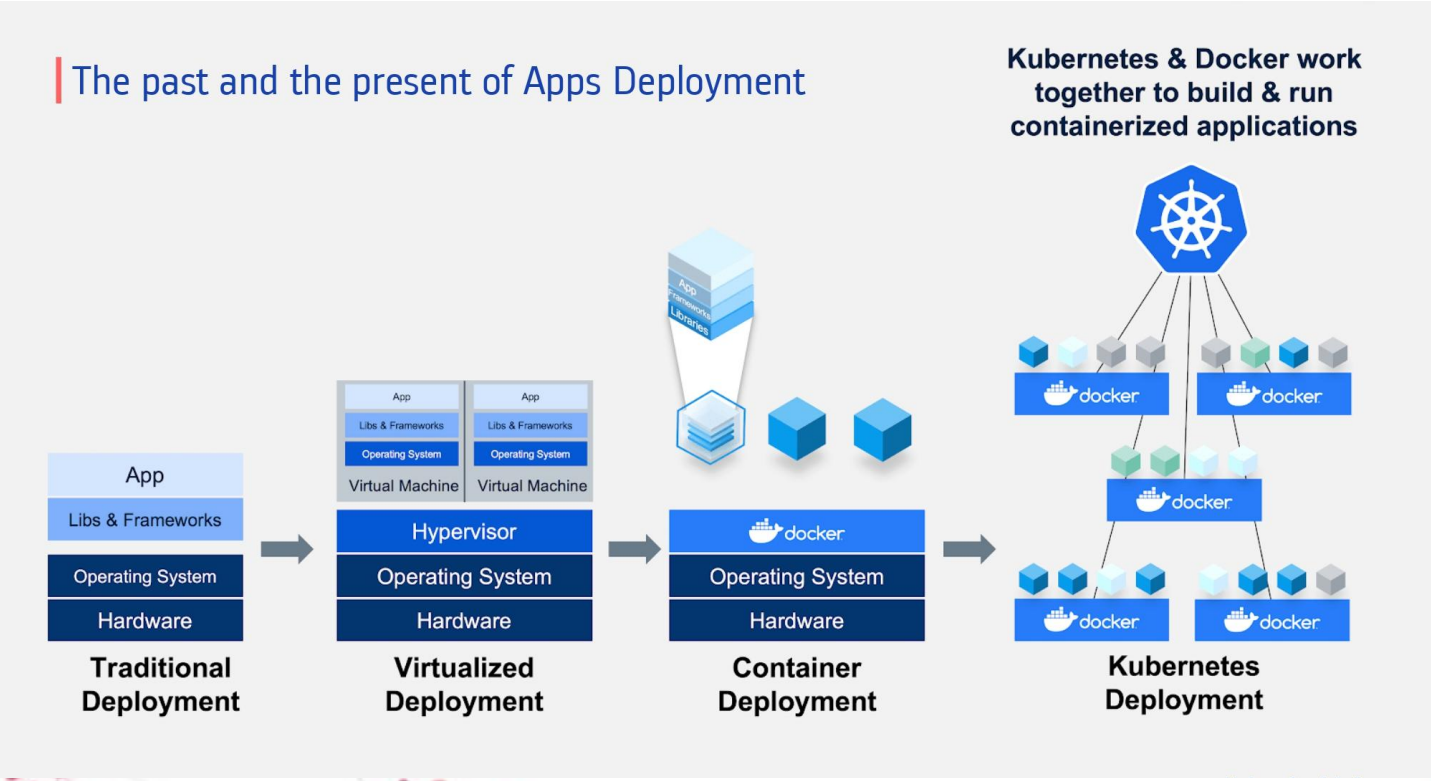


CGI

Module 1: Docker Concept & Terms

- What is container & Why?
- Container vs Virtual Machine
- Linux Containers & Docker
- Terminologies in Docker world
- Docker Architecture

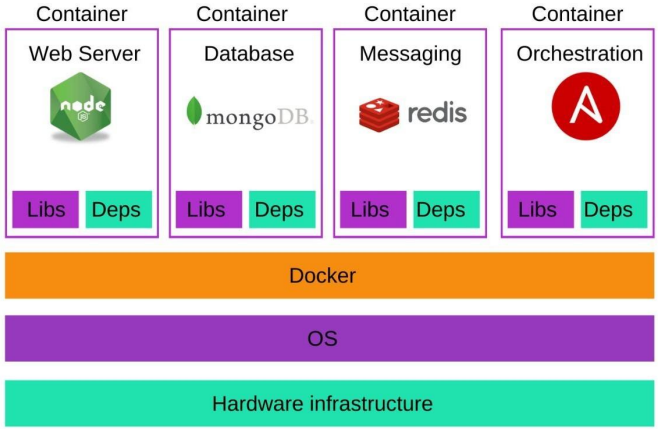
Container - Journey



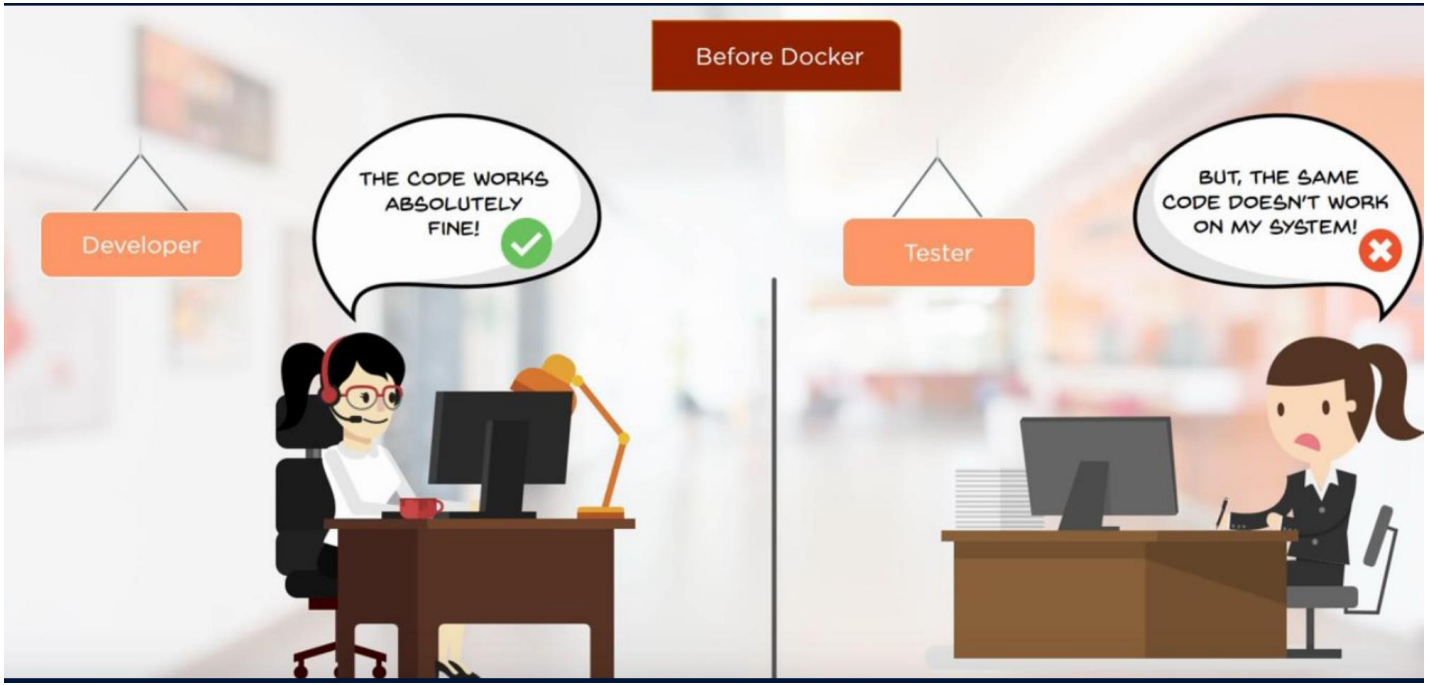
What is Docker and Container ?

Docker is a software development tool and a virtualization technology that makes it easy to develop, deploy, and manage applications by using containers.

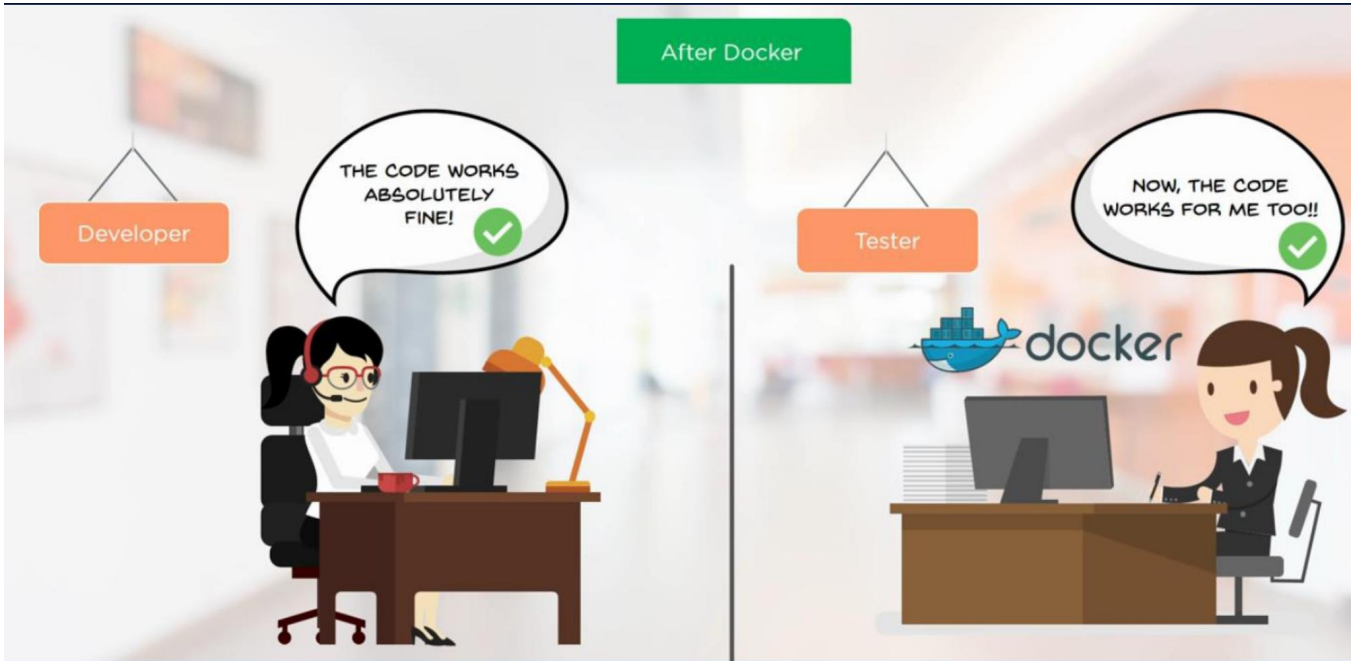
Container refers to a lightweight, stand-alone, executable package of a piece of software that contains all the libraries, configuration files, dependencies, and other necessary parts to operate the application.



Why Containers?



Why Containers?

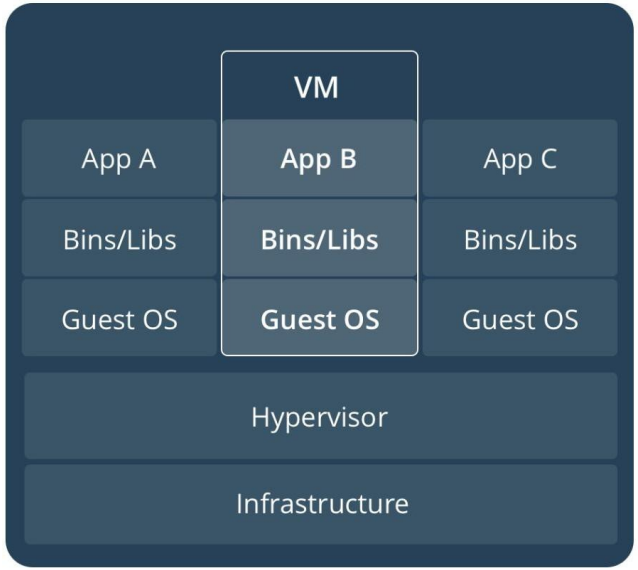


Why Containers?

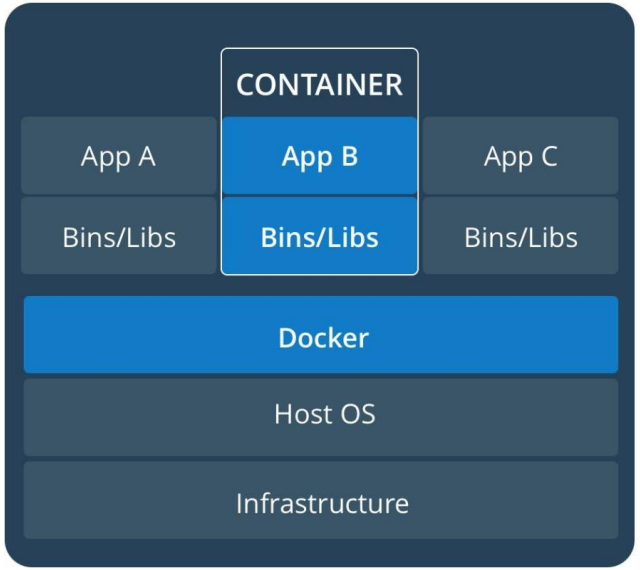
- **Flexible**: Even the most complex applications can be containerized.
- **Lightweight**: Containers leverage and share the host kernel.
- **Interchangeable:** You can deploy updates and upgrades on-the-fly.
- **Portable**: You can build locally, deploy to the cloud, and run anywhere.
- **Scalable**: You can increase and automatically distribute container replicas.
- Running more workload on the same hardware

Virtual Machines and Containers

Virtual Machine diagram



Container diagram



Virtual Machines and Containers

Virtual Machine	Docker Container
Hardware-level process isolation	OS level process isolation
Each VM has a separate OS	Each container can share OS
Boots in minutes	Boots in seconds
VMs are of few GBs	Containers are lightweight (KBs/MBs)
Ready-made VMs are difficult to find	Pre-built docker containers are easily available
VMs can move to new host easily	Containers are destroyed and re-created rather than moving
Creating VM takes a relatively longer time	Containers can be created in seconds
More resource usage	Less resource usage

Terminologies

Image Executable package that includes everything needed to run an application – the code, a runtime, libraries, environment variables, and configuration files

Container Runtime instance of an image–what the image becomes in memory when executed

Service a container but service codifies the way image runs -replicas, port, name etc

Swarm cluster of machines running docker containers

Registry storage and content delivery system, holding named Docker images, available in different tagged versions

Server Daemon creates and manages docker objects - images, containers, network, volumes, swarm etc

Docker Client CLI to communicate with server using Docker API

Docker REST API Communication contract between docker component (servers & clients)

Network Docker object holding the networking meta-data

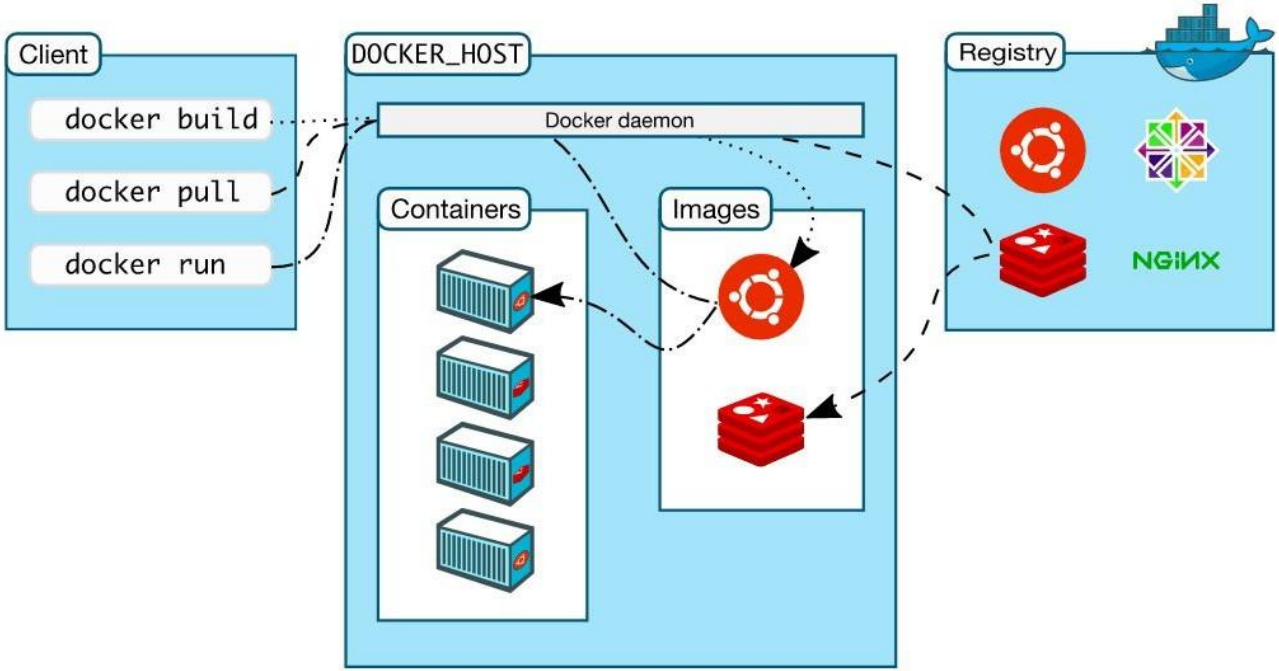
Node machine participating in Swarm

Volume Storage of persistence data generated and managed by Docker containers

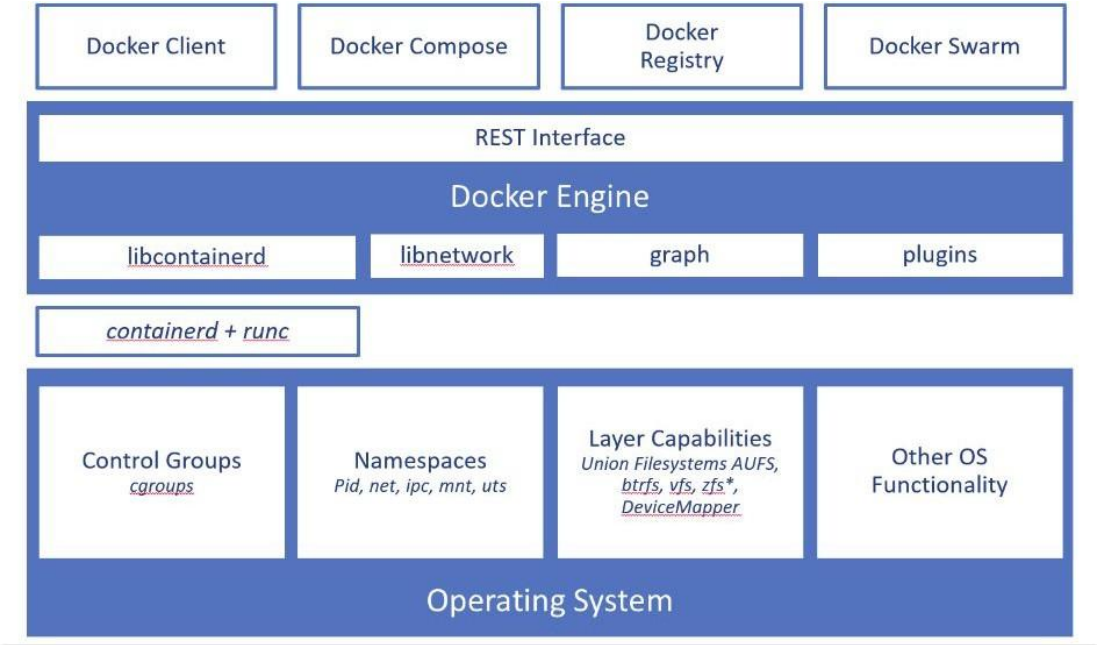
Docker Architecture

- Docker uses a client-server architecture.
- Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers.
- Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.
- For a virtual communication between CLI client and Docker daemon, a REST API is used

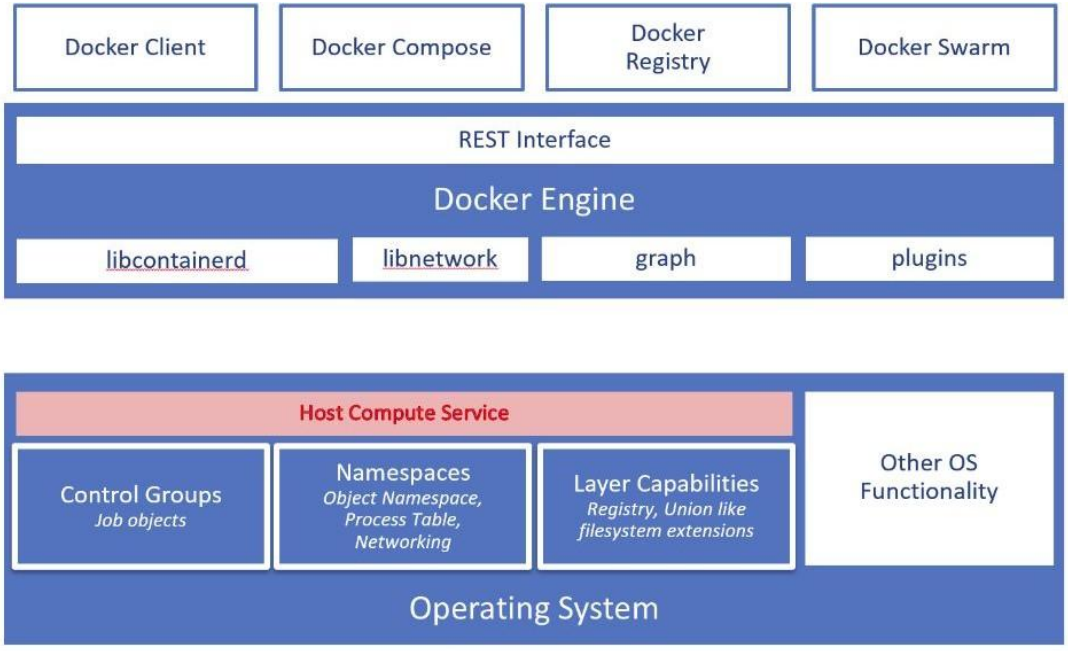
Docker Architecture



Docker Architecture - Linux



Docker Architecture - Windows



Docker Setup (Ubuntu)

```
sudo apt-get update

sudo apt-get remove docker docker-engine docker.io

sudo apt install docker.io

sudo groupadd docker

sudo usermod -aG docker $USER

sudo systemctl start docker

sudo systemctl enable docker
```

Module 2: Docker Containers

- Creating & Starting containers
- Running containers
- Docker Images
- Connecting containers

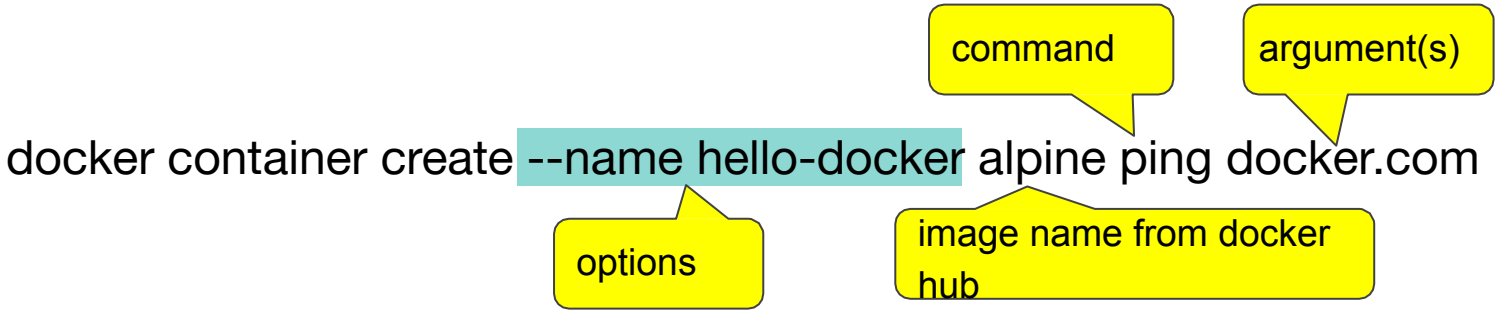
Creating containers

docker container create [OPTIONS] IMAGE [COMMAND] [ARG...]

Options:

--name string name of the container
--cpus decimal number of CPUs
--label list set metadata on a container
--memory bytes memory limit
--network string connect container to a network (default “default”)
--publish list publish container’s port to the host
--rm remove container when it exits
-i interactive Keep STDIN open if not attached
-t allocates psuedo-TTY

Creating containers - Examples



docker container create --name busy -it busybox

docker container create --name alpine -it alpine sh

docker container create --name hello -p 80:80 tutum/hello-world

Starting containers

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

Options:

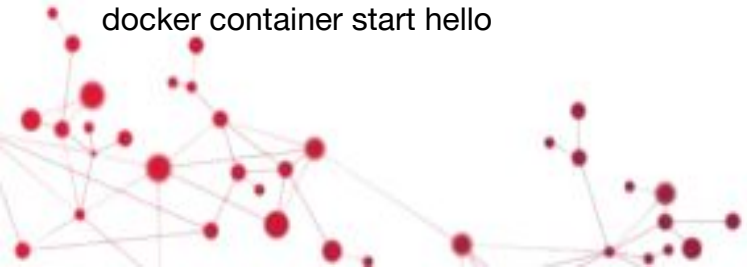
-i Attach container's STDIN
-a Attach container's STDOUT/STDERR and forward signals

Examples:

docker container start -ia busy

docker container start -ia alpine

docker container start hello

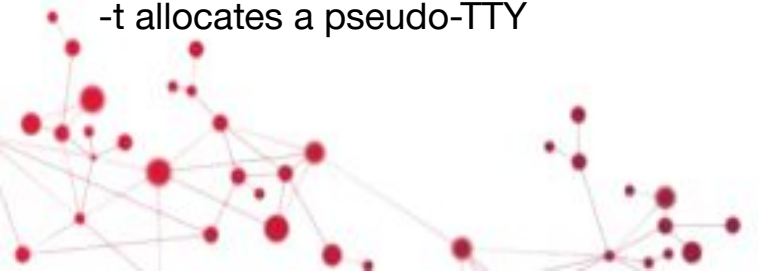


Running containers

docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]

Options:

--name string name of the container
--cpus decimal number of CPUs
--label list set metadata on a container
--memory bytes memory limit
--network string connect container to a network (default “default”)
--publish list publish container’s port to the host
--rm remove container when it exits
-i interactive mode
-t allocates a pseudo-TTY



Running containers - Examples

docker container run -p 80:80 tutum/hello-world (creates container with random name)

docker container run -p 80:80 nginx (connects to tty, Ctrl+C to exit)

docker run -p 81:80 nginx (shorthand command)

docker run --name ngx -p 80:80 -it nginx (interactive terminal, Ctrl+PQ to leave it running)

docker attach ngx

docker run -d ubuntu /bin/sh -c "while true; do echo current date and time is: $(date); sleep 10; done"

docker run -P --name nginx nginx (map exposed ports to random ports on the host {range 49153 and 65535})

docker run -d -p 8000-9000:80 nginx (maps port 80 to any random port between 8000 to 9000 on host)

docker run --restart always -p 80:80 -it nginx

Docker Images

- **Image** - Executable package that includes everything needed to run an application – the code, a runtime, libraries, environment variables, and configuration files

docker images
docker images nginx
docker images ubuntu
docker images ubuntu
docker rmi ubuntu
docker rmi $(docker images -q)

Module 3: Provisioning Docker Images

- Introducing the Dockerfile
- Building images manually / Examples...
- Storing and retrieving Docker Images from Docker Hub
- Building images using Continuous Integration tools
- Inspecting a Dockerfile from DockerHub

Different ways to create images

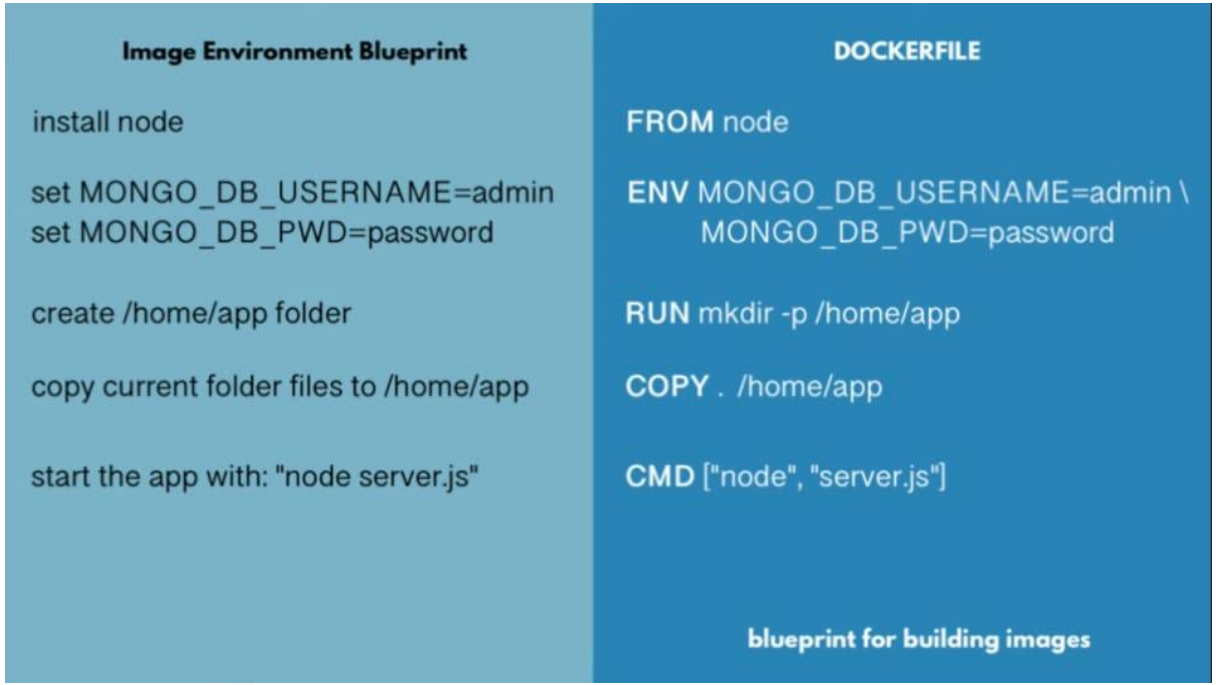
docker commit	Build an image from a container
docker build	Create an image from a Dockerfile by executing the build steps given in the file
docker import	Create a base image by importing from a tarball. [import is mainly used for creating base-images; first two options are widely used]

`Dockerfile is a text document that contains`

- `a set of instructions required to assemble the app (image) and/ run it`

CGI

Introducing the Dockerfile



Introducing the Dockerfile

- **ENV** - to set environment variables
- **EXPOSE** - to expose ports
- **FROM** - base image
- **LABEL** - to add metadata to image
- **HEALTHCHECK** - to check if container is running
- **USER** - to set user and group
- **VOLUME** - to specify mount point from external host
- **WORKDIR** - workdir to run any of the commands

CGI

Introducing the Dockerfile

- ARG - variable used during build time
- CMD - to provide defaults to executing container
- RUN - to execute commands in new layer
- COPY - Copy file,dir or remote url to image
- ADD - Copy file,dir or remote url to image
- ENTRYPOINT - to configure container as executable
- MAINTAINER - the image maintainer

RUN COPY ADD instructions create new layers in the image stack - refer layering section

Building Images (Alpine ping)

cat Dockerfile

```
FROM alpine:latest
MAINTAINER ninad@gmail.com
CMD ping google.com
```

Build

- `docker build -t myalpine .`

Run

- `docker run myalpine`

Building Images (Ubuntu with utilities)

cat Dockerfile

```
FROM ubuntu:letest
MAINTAINER ninad@gmail.com
RUN apt-get update && apt-get install -y tree && apt-get install -y telnet && apt-get install -y curl
```

Build

- `docker build -t myubuntu .`

Run

- `docker run -it myubuntu`

Building Images (hello-world)

git clone https://github.com/NinjaCloud/tutum-hello-world-rebuild

```
FROM nginx:1.19.6-alpine
RUN apk --update add php-fpm
RUN mkdir -p /tmp/nginx && echo "clear_env = no" >> /etc/php7/php-fpm.conf
ADD www /www
ADD nginx.conf /etc/nginx/
COPY entrypoint.sh .
RUN chmod +x entrypoint.sh
ENTRYPOINT [ "./entrypoint.sh" ]
```

Build

- `docker build -t hello-world .`

Run

- `docker run -it hello-world`

curl -o logo.png https://www.docker.com/wp-content/uploads/2022/03/vertical-logo-monochromatic.png

Building Images (Java-Program)

cat Dockerfile

```
FROM java:latest
COPY . /usr/src/
WORKDIR /usr/src/
RUN javac hello.java
CMD ["java", "hello"]
```

vi hello.java

```
class hello {
    public static void main(String []args) {
        System.out.println("Hey Ninad");
    }
}
```

Build

- `docker build -t mycode-java .`

Run

- `docker run -it mycode-java`

Docker Hub - store & retrieve

https://hub.docker.com (register and create login)

- docker tag alpine ninjacloud05/alpine:ninad
- docker push ninjacloud05/alpine:ninad
- docker pull ninjacloud05/alpine:ninad

docker login -u "myusername" -p "mypassword" docker.io
docker push myusername/myimage:0.0.1

Module 4: Diving deeper - Dockerfile

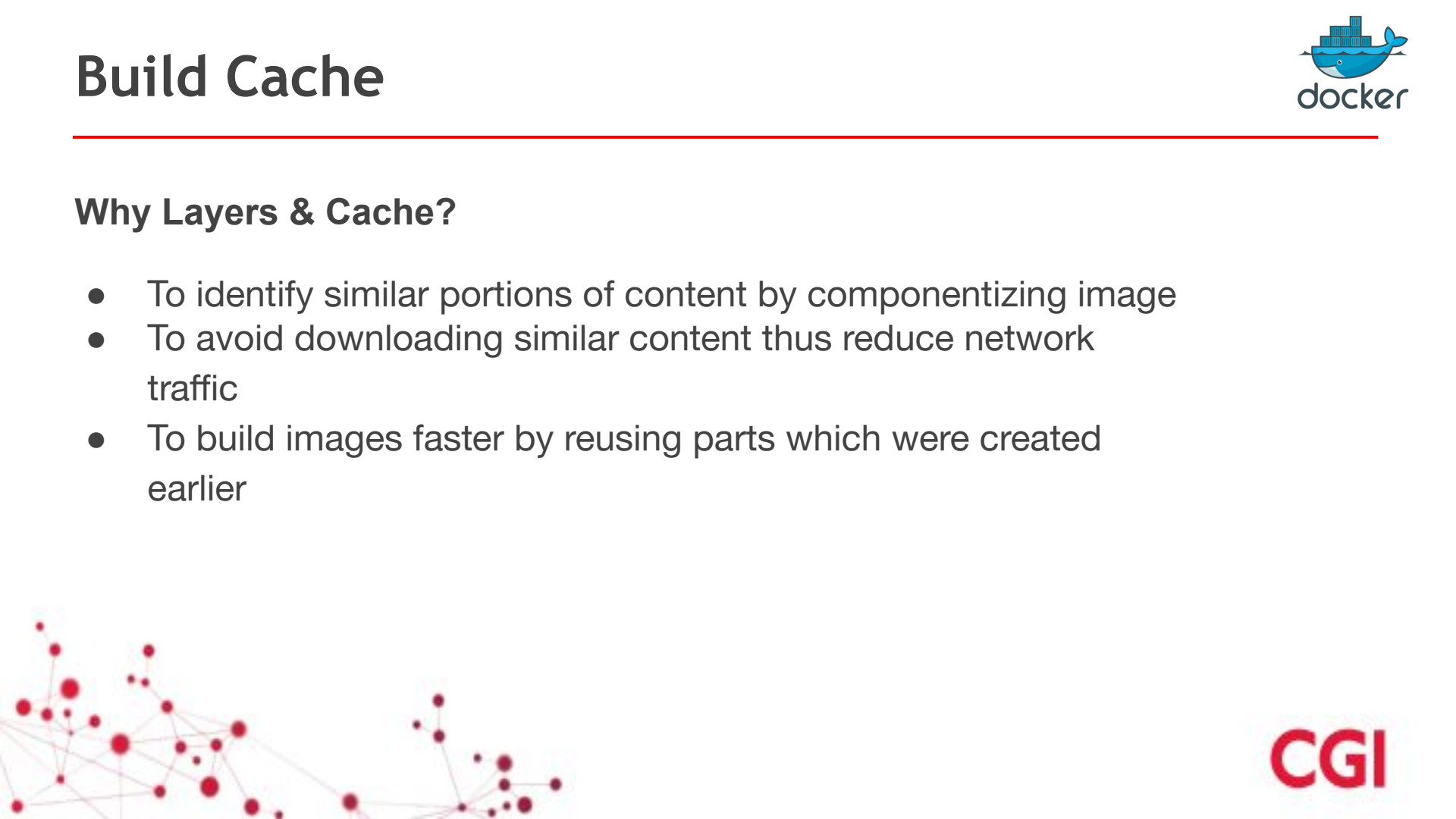
- Dockerfile and Layers
- The Build cache
- The ENTRYPOINT Instruction
- The CMD Instruction Docker
- The ENV Instruction
- Volumes and the VOLUME Instruction

Dockerfile & Layers

```
ubuntu@ip-172-31-31-236:~$ docker images springio/*
REPOSITORY    TAG      IMAGE ID       CREATED        SIZE
springio/gs-spring-boot-docker  latest   3a7a85f42b64   6 months ago                181MB

ubuntu@ip-172-31-31-236:~$ docker history 3a7a85f42b64
IMAGE  CREATED         CREATED BY    SIZE     COMMENT
3a7a85f42b64   6 months ago    /bin/sh -c #(nop) ENTRYPOINT ["sh" "-c" "... 0B
<missing>      6 months ago    /bin/sh -c #(nop) ENV JAVA_OPTS=       0B
<missing>      6 months ago    /bin/sh -c #(nop) ADD file:2f6c6463d5fd2c4...  14.4MB
<missing>      6 months ago    /bin/sh -c #(nop) VOLUME [/tmp]            0B
<missing>      6 months ago    /bin/sh -c apk add --no-cache --virtual=bu...  156MB
<missing>      6 months ago    /bin/sh -c #(nop) ENV JAVA_VERSION=8 JAVA... 0B
<missing>      7 months ago    /bin/sh -c #(nop) ENV LANG=C.UTF-8    0B
<missing>      7 months ago    /bin/sh -c ALPINE_GLIBC_BASE_URL="https://... 6.7MB
<missing>      7 months ago    /bin/sh -c #(nop) CMD ["/bin/sh"]          0B
<missing>      7 months ago    /bin/sh -c #(nop) ADD file:4583e12bf5caec4...  3.97MB
```

Build Cache



Why Layers & Cache?

- To identify similar portions of content by componentizing image
- To avoid downloading similar content thus reduce network traffic
- To build images faster by reusing parts which were created earlier

ENTRYPOINT and CMD

```
FROM Ubuntu

CMD sleep 5
```

CMD command param1

CMD ["command", "param1"]

CMD sleep 5

CMD ["sleep", "5"] ✓

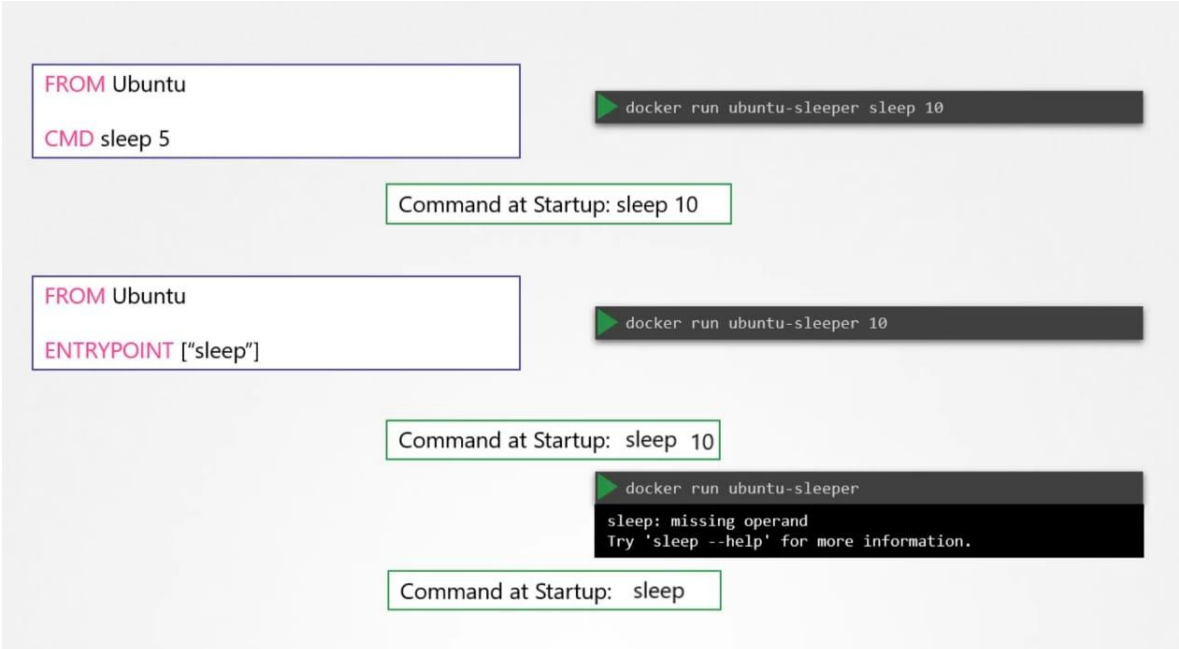
CMD ["sleep 5"] ✗

```
docker build -t ubuntu-sleeper .
```

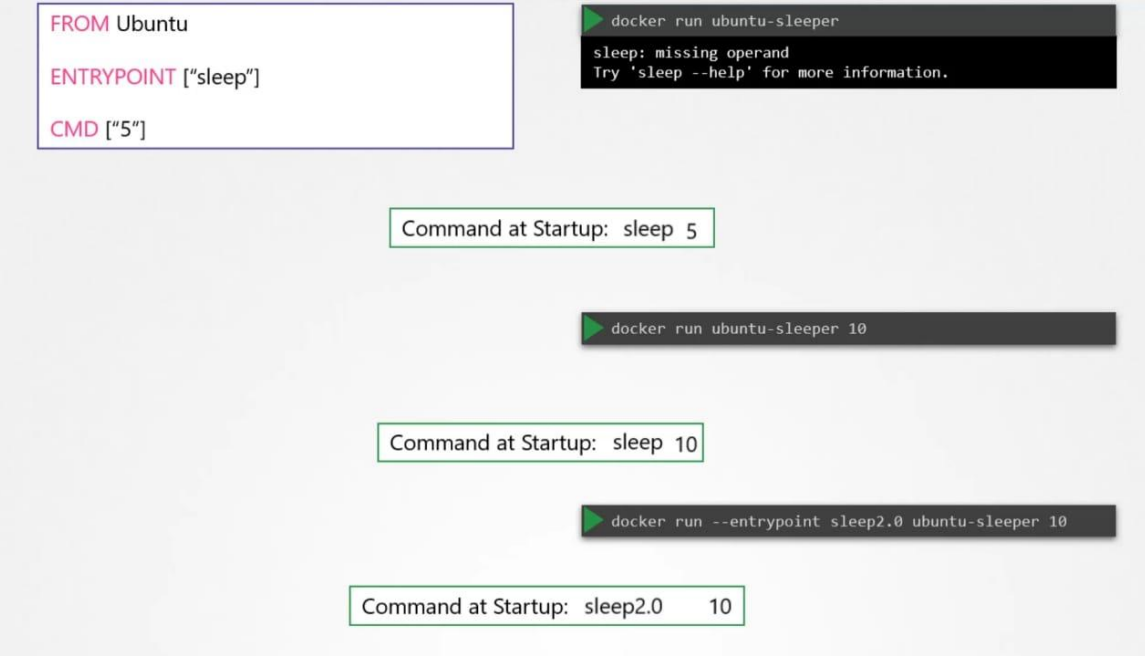
```
docker run ubuntu-sleeper
```

5

ENTRYPOINT and CMD

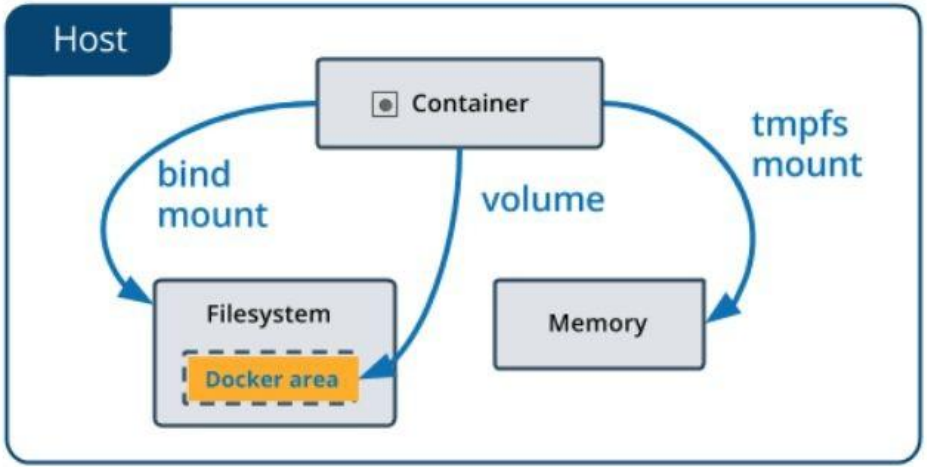


ENTRYPOINT and CMD



The VOLUME - Data Persistence

Storage of persistence data generated by managed by Docker containers



Commands:

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

VOLUME - Examples

Examples (volume): Persist data in a container's writeable layer

- docker run -d --name devtest --mount source=app,target=/app nginx:latest

Examples (bind volume): a file or directory on the *host machine* is mounted into a container. Performant but not-reliable

- docker run -d -it --name devtest --mount type=**bind**,source="$(pwd)",target=/app nginx:latest

Examples (tmfs volume): For temporary sensitive data to be kept only in memory

- docker run -d -it --name tmptest --mount type=**tmpfs**,destination=/app nginx:latest

VOLUME - preferred way

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers allow you to store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- A new volume's contents can be pre-populated by a container.

Module 5: Working with Registry

- Overview
- Creating a Public repo on Docker Hub
- Using our Public repo on Docker Hub
- Using a Private Registry
- Docker Enterprise
- Lab Exercises

Overview - Registry

Registry

Stateless, highly scalable server side application that stores and lets you distribute Docker images.

When to use

- tightly control where your images are being stored
- fully own your images distribution pipeline
- integrate image storage and distribution tightly into your in-house development workflow

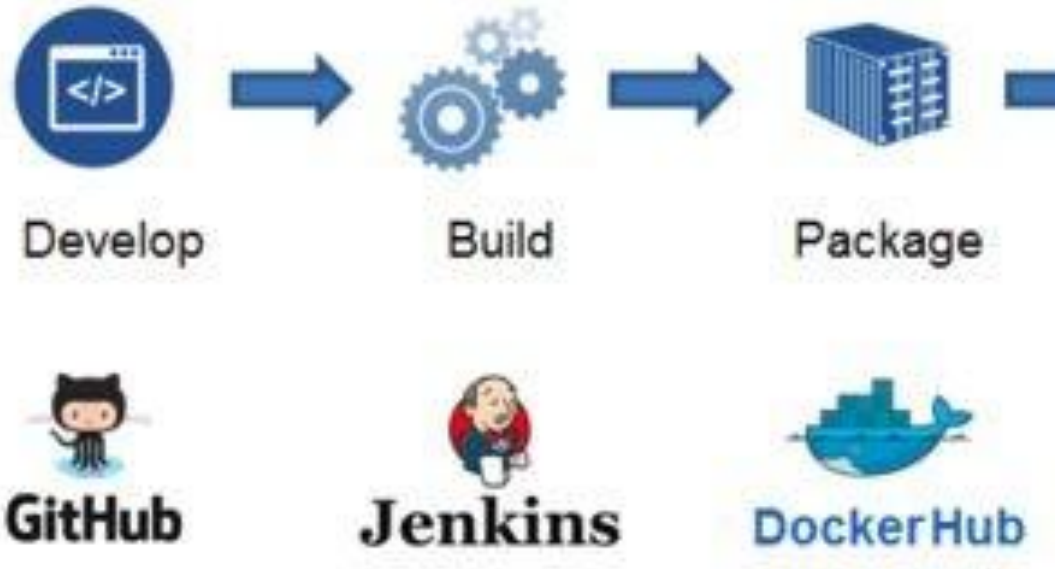
Registry Server

Run a local registry

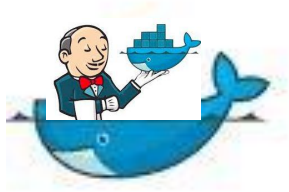
```
$ docker run -d -p 5000:5000 --restart=always --name registry registry:2
$ docker pull ubuntu:16.04
$ docker tag ubuntu:16.04 localhost:5000/my-ubuntu
$ docker push localhost:5000/my-ubuntu
$ docker image remove ubuntu:16.04
$ docker image remove localhost:5000/my-ubuntu
$ docker pull localhost:5000/my-ubuntu
$ docker container stop registry && docker container rm -v registry
```

CGI

Jenkins



Dockerizing dev workflow

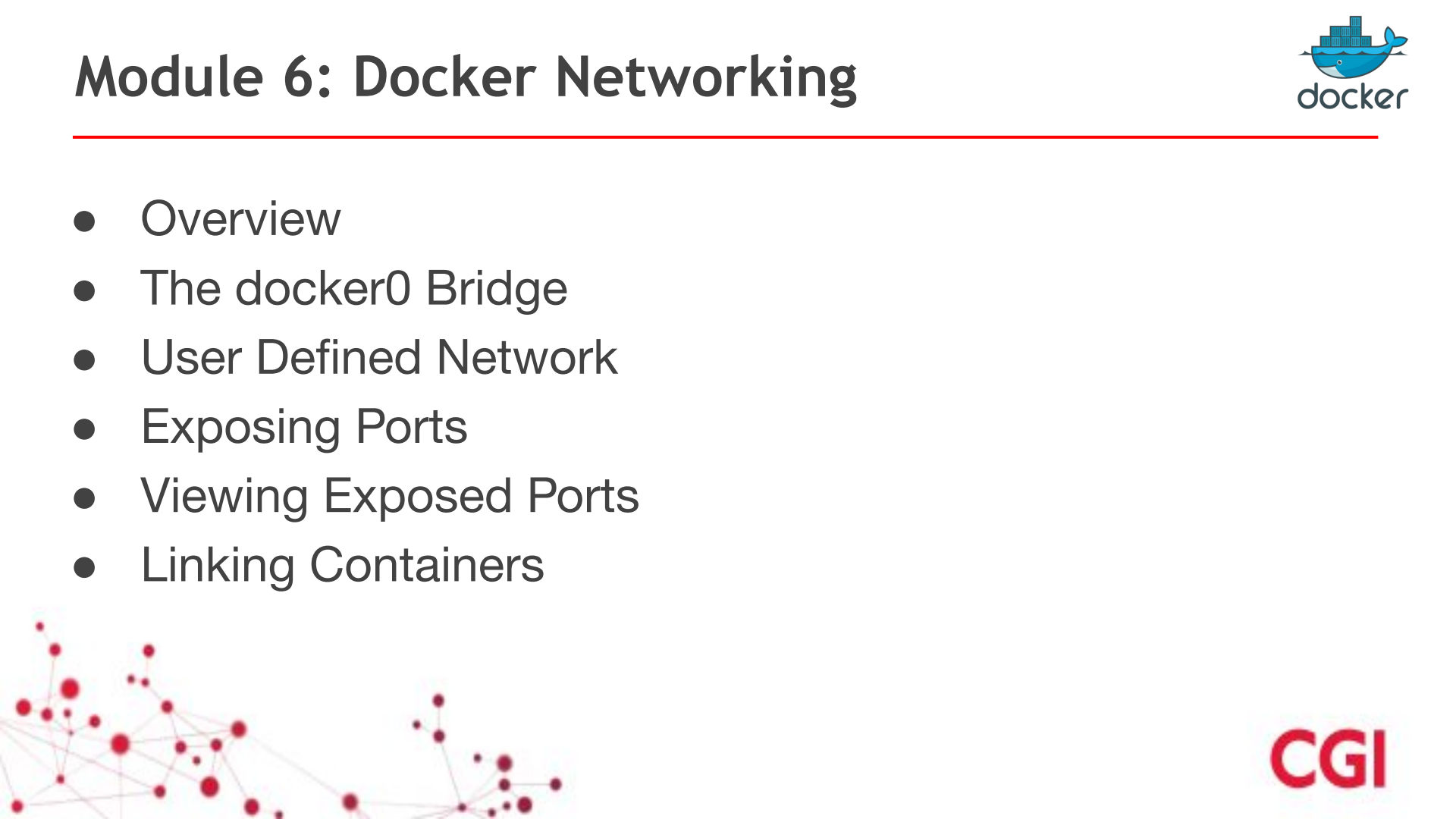


Private Docker Registry

```
docker run --name jenkins -u 0 -d -p 8080:8080 -v /var/run/docker.sock:/var/run/docker.sock -v $(which docker):$(which docker) jenkins/jenkins:lts
```

Notes:
Add docker pipeline jenkins plugin to work
Test project: https://github.com/NinjaCloud/nodejsappdocker.git
Add jenkins credential having ID **docker-hub-credentials** for docker hub push access

Module 6: Docker Networking



- Overview
- The docker0 Bridge
- User Defined Network
- Exposing Ports
- Viewing Exposed Ports
- Linking Containers

Overview - Networking

Defines how containers communicate with external world, amongst cluster members etc

Two types of networks:

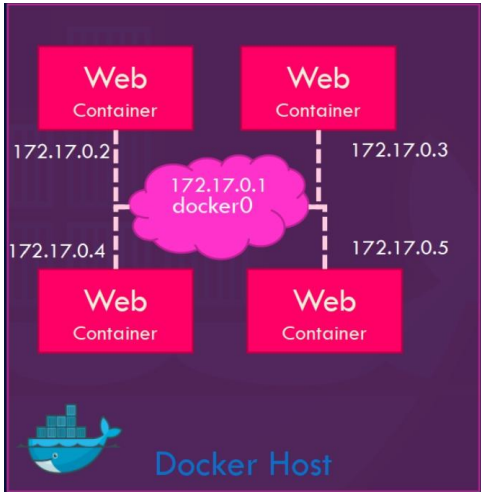
- Default
- Custom Defined

Default:

- Bridge - docker0 (docker created default network) Configurable
- Host - container on host network stack Not configurable
- None - container specific network stack (no network interface) Not configurable

The docker0 bridge

- In Bridge network, all containers get private internal IPs and they are isolated from host.
- Container inter-connectivity using IP addresses (no name resolution)
- For name resolution, legacy --link feature available for limited period
- Change default bridge to none using --network flag or daemon.json server config



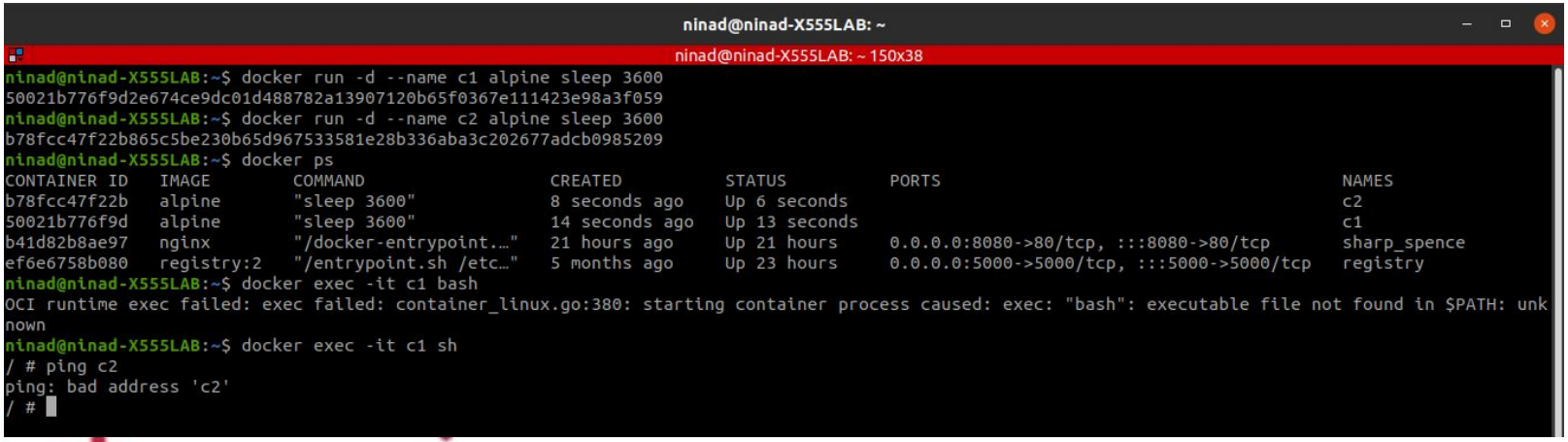
Create a bridge network: docker network create --driver bridge my-net

Attach a container to it: docker run -d --name web --net my-net nginx

Understanding DNS resolution in bridge network

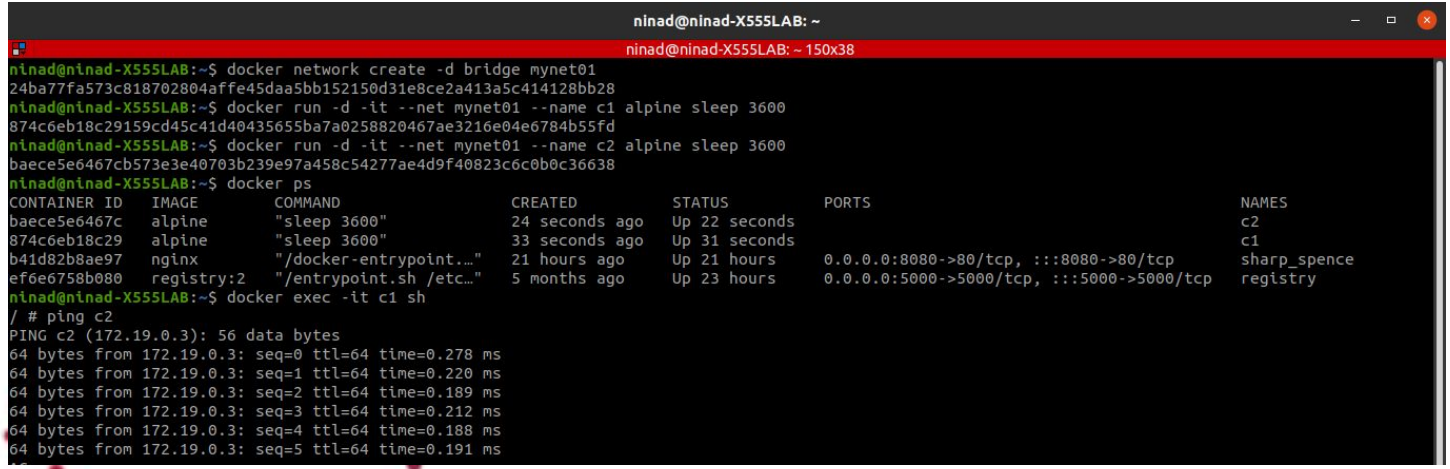
- When containers are run in default bridge network they cannot find each other using their container names.

- Simply put, DNS resolution through container names will not work under default bridge network



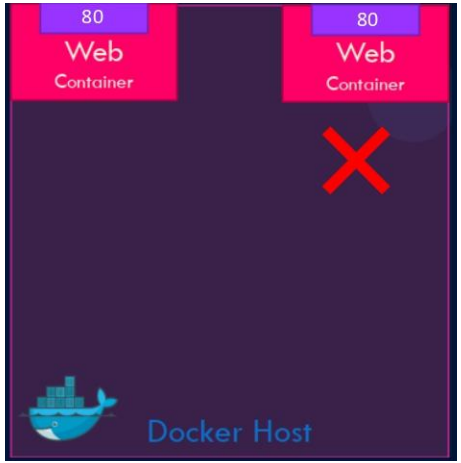
Understanding DNS resolution in bridge network

- Now a new bridge network is created and containers are attached to that network.
- In this case, containers find each other using their container names(DNS resolution through container names)



Docker Networking: Host

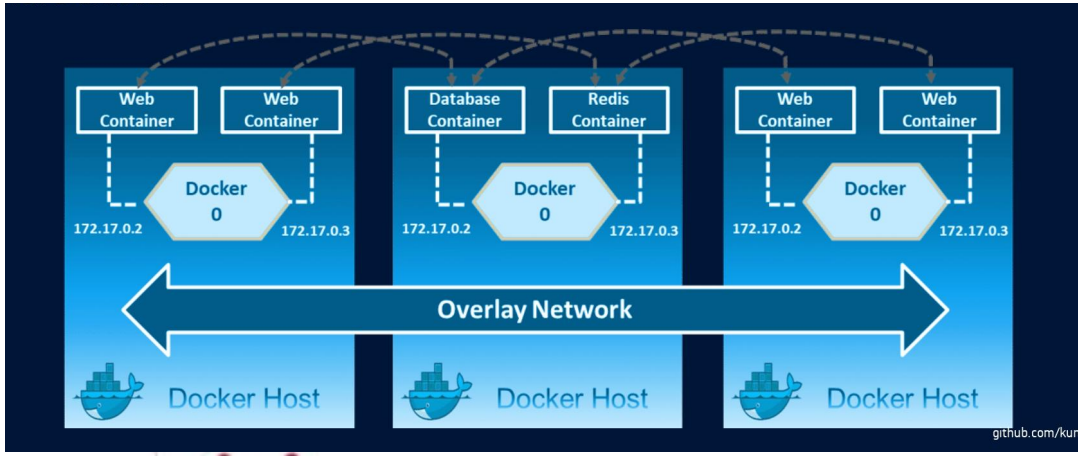
- In host network, all containers directly get connected to host.
- Multiple containers cannot run on same hosts because of port conflicts on host side



docker run -d --name web –net host nginx

Overlay Network

- Scope is swarm mode
- Bridge networks apply to containers running on the same Docker daemon host. For communication among containers running on different Docker daemon hosts, we should use an overlay network which spans across the entire cluster
- Uses NAT and port mapping (iptables)



Test Setup - 1

Test Setup:

Create custom network n1

- docker network create n1

Create two busybox containers attached to n1

- docker run -itd --name c1 --network n1 busybox
- docker run -itd --name c2 --network n1 busybox

Tests

- Log into c1 and ping c2 (should succeed)
  - docker exec -it c1 sh
  - ping c2
- Log into c2 and ping c1 (should succeed)
  - docker exec -it c2 sh
  - ping c1

Test Setup - 2

Prerequisites: Test Setup -1

Test Setup:
Remove network from both containers c1 & c2

- docker network disconnect n1 c1
- docker network disconnect n1 c2

Tests:

- Login into c1 and ping c2 (should fail)
  - docker exec -it c1 sh
  - ping c2
- Login into c1 and ping google.com (should fail)
  - docker exec -it c1 sh
  - ping google.com
- Run ifconfig on c1 to see interfaces (should see only loopback interface)
  - docker exec -it c1 sh
  - ifconfig
- Do the same on c2 (results should be similar)


docker
THANK YOU !!
CGI