Docker

Lev Epshtein

NAYA College

About me

Lev Epshtein

Technology enthusiast with 15 years of industry experience in DevOps and IT. Industry experience with back-end architecture.

Solutions architect with experience in big scale systems hosted on AWS/GCP, end-to-end DevOps automation process.

Cloud DevOps, and Big Data instructor at NAYA and JB.

Certified GCP trainer,

Partner & Solution Architect Consultant at Opsguru.

lev@opsguru.io

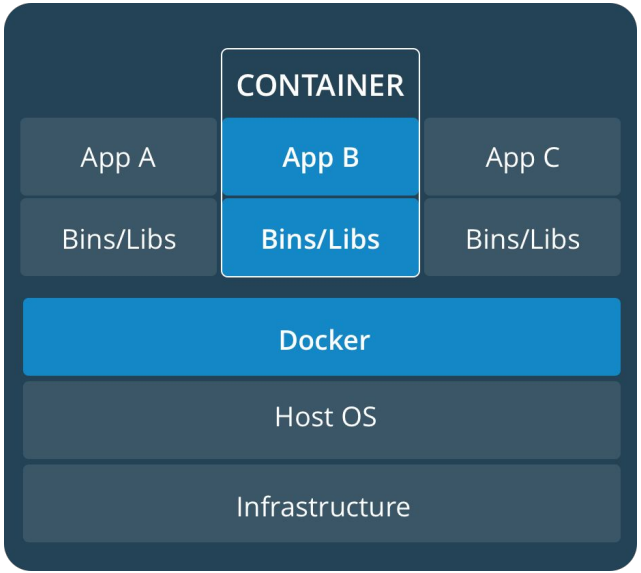
NAYA College

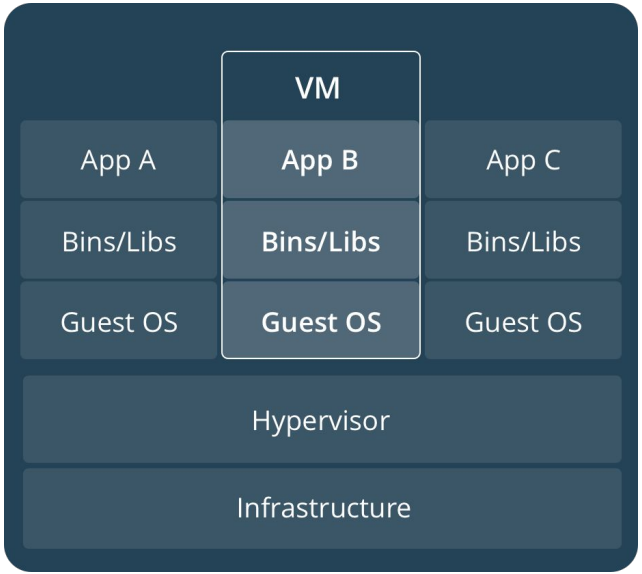
Docker

What is Docker?

- Docker is an open platform for developing, shipping, and running applications.
- Docker allows you to package an application with all of its dependencies into a standardized unit for software development.
- The use of containers to deploy applications is called ***containerization***.

Containers and Virtual machines





NAYA College

Docker Benefits Upon VMs

- **Small to tiny images** - Few hundred MB's for OS + Application (5MB for full OS - Alpine) VS. Gigabytes in VM's
- **Very small footprint on the host machine** (CPU, RAM Impact) as Docker only use what it required instead of building a complete Operating system per VM.
- **Containers use up only as many system resources as they need at a given time**. VMs usually require some resources to be permanently allocated before the virtual machine starts.
- **Direct hardware access.** Applications running inside virtual machines generally cannot access hardware like graphics cards on the host in order to speed processing. Containers Can (ex. Nvidia)
- **Microservice in nature and integrations** (API's) for whatever task required.
- **Portable, Fast** (Deployments , Migration , Restarts and Rollbacks) and **Secure**
- **Can run anywhere and everywhere**
- **Simplify DevOps**
- **Version controlled**
- **Open Source**

Common Use Case for Docker

- CI / CD
- Fast Scaling application layers for overcoming application performance limitations.
- For Sandboxed environments (Development, Testing , Debugging)
- Local development environment (no more “ It ran on my laptop…”)
- Infrastructure as a CODE made easy with docker
- Multi-Tier applications (Front End , Mid Tier (Biz Logic) , Data Tier) / Microservices
- Building PaaS , Saas

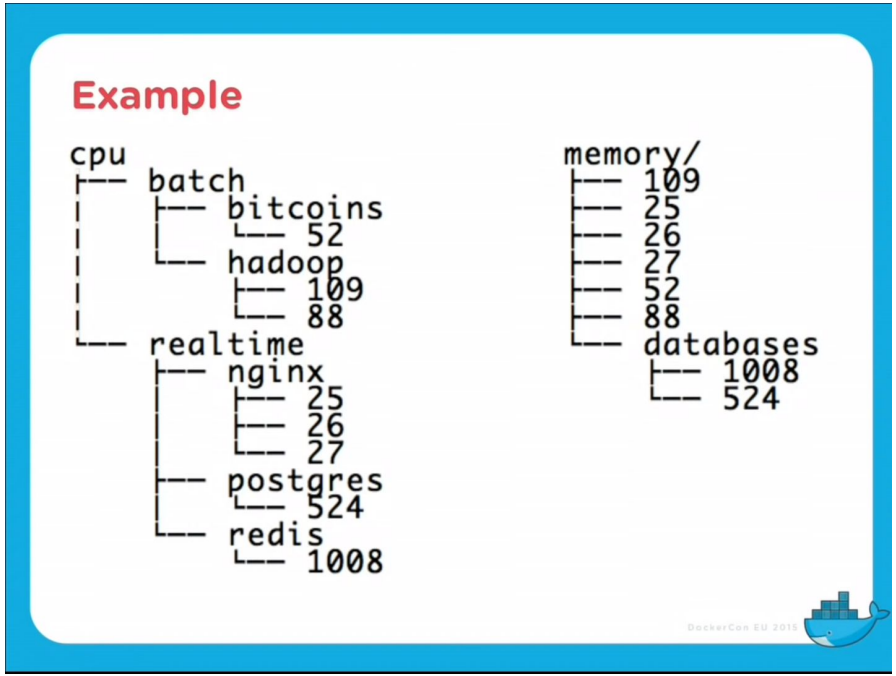
Docker Architecture

Under The Hood

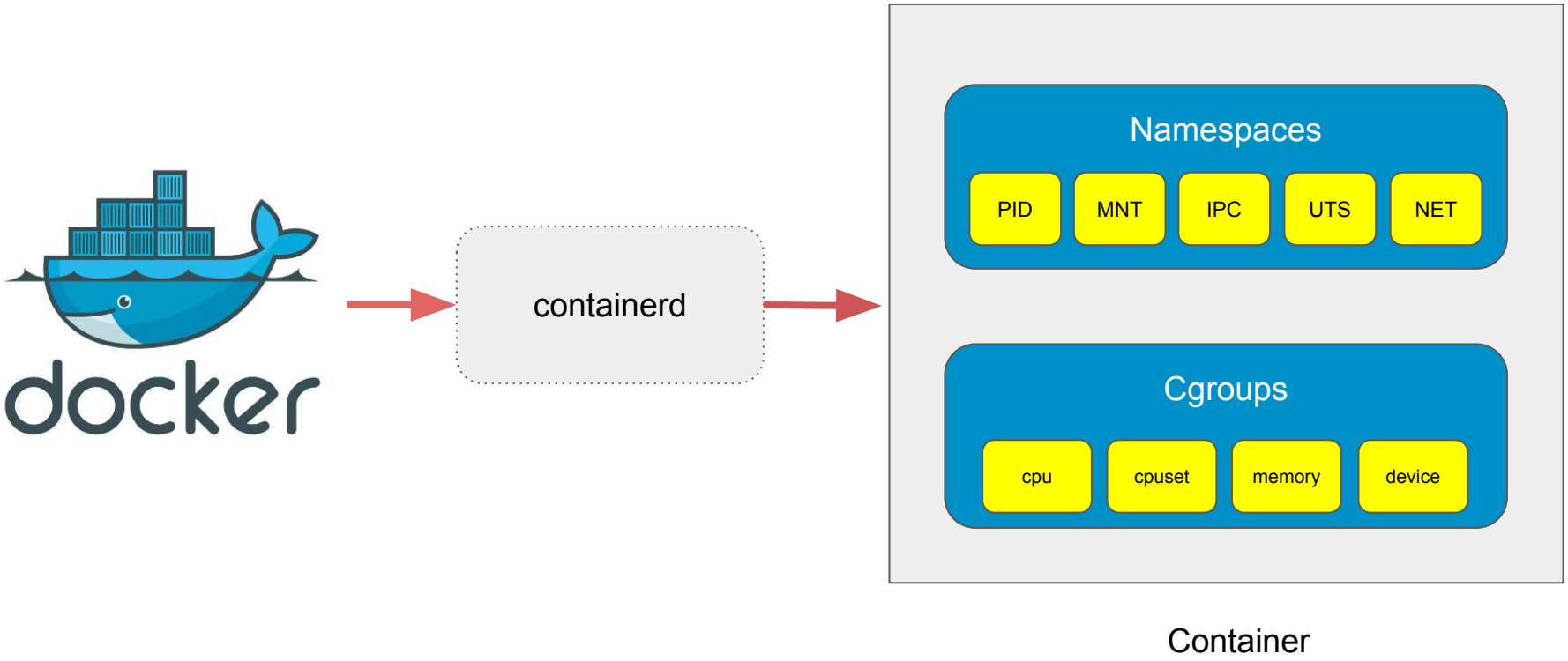
- Architecture: Linux X86-64
- Written in: GoLang (On March 13, 2014, with the release of version 0.9, Docker dropped LXC as the default execution environment which is an operating system level virtualization and replaced it with its own libcontainer library written in the Go programming language)
- Engine: Client - Server (Daemon) Architecture
- Namespace: Isolation of process in linux where one process cant “See” the other process
- Control Groups: Linux Kernel capability to limit and isolate the resource usage (CPU, RAM, disk I/O, network etc..) of a collection of process
- Container format: libcontainer - Go implementation for creating containers with namespaces, control groups and File system capabilities access control

Containers Cgroups

Cgroup Hierarchy Example

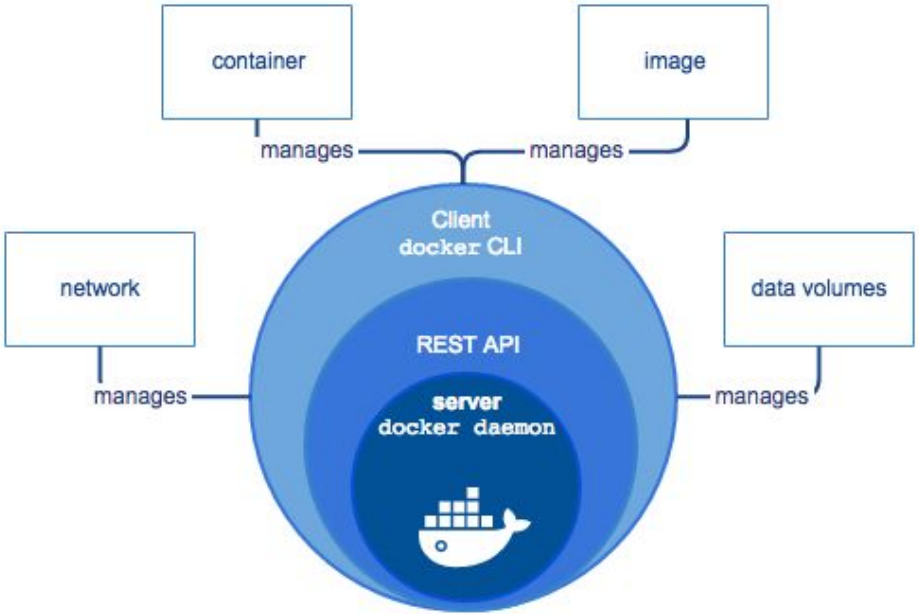


Deep into Container



Docker Architecture

Overview



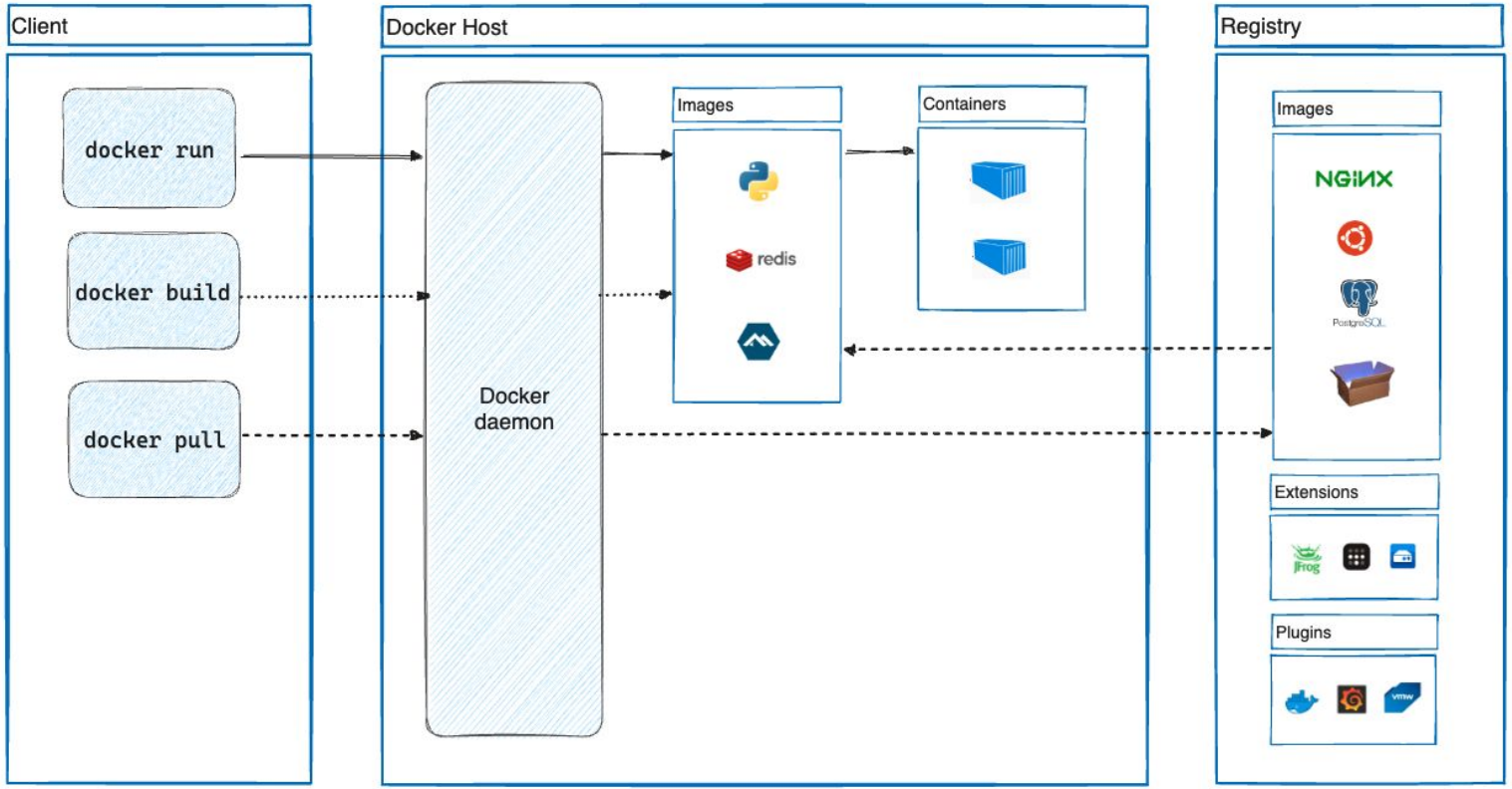
NAYA College

What is docker - Technical Aspect

Docker Architecture

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

The Docker Architecture



NAYA College

Docker Components

- Engine
- Daemon
- (Docker) Client
- Docker Registries
- Docker Objects
- Machine
- Compose

Docker Components

Engine

- A server which is a type of long-running program called a daemon process (the dockerd command).
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client (the docker command).

Docker Components

Daemon

- The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other **daemons** to manage Docker services.

Docker Components

Docker Client

- The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The **Docker client can communicate with more than one daemon.**

Docker Components

Docker Registries

- A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.
- When one use “docker pull / push / run” commands, the required images are pulled from the configured registry.

Docker Components

Docker Objects

1. **Images**
   a. Read Only template with instruction for creating a Docker Container. Often, an Image is based on another image with some additional customization.
   b. Self own images that are fully created by you using DockerFile with a simple syntax where every instruction control a different Layer in the image. Once a change is made to a specific layer, a rebuild of the image will change only the updated layers. This what makes images small, fast and lightweight in compared to other virtualization solutions

Docker Components

Docker Objects

1. **Containers**
   a. A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
   b. Container is defined by its image as a well as any configuration options we provide to it when created or when we start it

Docker Components

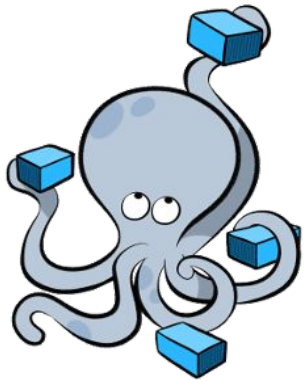
Docker Objects

1. **Services**
    a. Allow you to scale containers across multiple Docker daemons, which all work together as a swarm with multiple managers and workers. Each member of a swarm is a Docker daemon, and the daemons all communicate using the Docker API. A service allows you to define the desired state, such as the number of replicas of the service that must be available at any given time. By default, the service is load-balanced across all worker nodes. To the consumer, the Docker service appears to be a single application. Docker Engine supports swarm mode in Docker 1.12 and higher.

Docker Components

Docker Compose

A tool for defining and running complex applications with Docker (eg multi-container application ex. LAMP)
With a single file.



The underlying technology

Written in Go and takes advantage of several of the Linux kernel features:

- **Namespaces**
  - Provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.
- **Control groups (cgroups)**
  - Allow you to limit the access processes and containers have to system resources such as CPU, RAM, IOPS and network.

Let’s Start

git clone https://github.com/levep/ML-k8s.git

NAYA College

Docker Flow

```
> docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest sh
```

‘`docker run`’ will run the container

This will not restart an already running container, just create a new one

docker run [options] **IMAGE** [command] [arguments]

[options] modify the docker process for this container

IMAGE is the image to use

[command] is the command to run inside the container (entry point to hold the container running)

[arguments] are arguments for the command

NAYA College

Docker Flow

```
> docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest sh
```

‘`docker run`’ will run the container:

-i - Interactive mode

-t - Allocate pseudo TTY - or not Terminal will be available

-d - Run in the background (Daemon style)

--name - Give the container a name or let Docker to name it

-p [local port] : [container port] - Forward local port to the container port

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES
98debfed4458        alpine:latest       "sh"                Less than a second ago   Up 1 second         0.0.0.0:8080->80/tcp     dockerlearning
```

NAYA College

Docker Flow

```
> docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest sh
```

- Pulls the alpine:latest image from the registry (if not existed on our station)
  - Run “docker images” to see what images already downloaded / in use locally
- Creates new container
- Allocate FS and Mounts a read-write Layer
- Allocates network/bridge interface
- Set up an IP Address
- Executes a process that we specify (in this scenario - “sh” as alpine release doesn't have bash)
- Captures and provides application outputs

Common Docker Commands

```
> docker run -h MyDocker -i -t --rm debian /bin/bash
```

- docker run will run the container
  - -h - Container host name
  - --rm - Automatically remove the container when it exits
- docker exec/atatch

```
> docker exec -it <CONTAINER> ash
```

```
> docker attach <CONTAINER> ,.. (Ctrl P + Ctr Q to quit)
```

NAYA College

Common Docker commands

// General info
man docker // man docker-run
docker help // docker help run
docker info
docker version
docker network ls
docker volumes ls

// Images
docker images // docker [IMAGE_NAME]
docker pull [IMAGE] // docker push [IMAGE]

// Containers
docker run
docker ps // docker ps -a, docker ps -l
docker stop/start/restart [CONTAINER]
docker stats [CONTAINER]
docker top [CONTAINER]
docker port [CONTAINER]
docker inspect [CONTAINER]
docker inspect -f "{{ .State.StartedAt }}" [CONTAINER]
docker rm [CONTAINER]

NAYA College

Running Mysql On docker

1. Execute:

```
docker run -p 13306:3306 --name mysql-docker-local -eMYSQL_ROOT_PASSWORD=Password -d mysql:latest
```

1. ssh to running container and check mysql is running.

NAYA College

STOP AND REMOVE

CLEAN UP

Docker Volumes

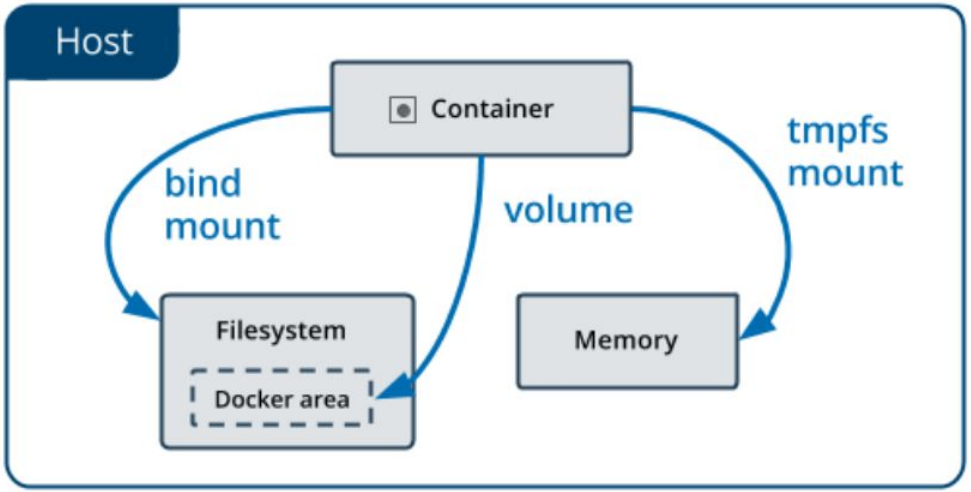
What is a Volume

Special type of directory in a container typically referred to as a “data volume”

- Can be shared and reused among one or many containers
- Updates to an image won’t affect a data volume
- Data volumes are persisted even after container deletion
- Volumes are OS agnostic. They can run on Linux and windows containers
- Volumes drivers allow us to store volumes on remote hosts or cloud providers.
- Volumes can be encrypted or to add other functionality
- A new volume content can be pre-populated by a container

Manage Data in Docker

An easy way to visualize the difference among **volumes**, **bind mounts**, and **tmpfs** mounts is to think about where the data lives on the Docker host.



NAYA College

Volumes - Demo

```
> docker run -dti --name alpine1 --mount target=/app alpine ash

> docker inspect alpine1

> docker stop alpine1 && docker rm alpine1
```

NAYA College

STOP AND REMOVE

CLEAN UP

Volumes - Demo

Creating a VOLUME managed by docker FS and share it with multiple containers

```
> docker volume create fs_shared

> docker volume ls

> docker run --rm -tdi --name alpine1 --mount source=fs_shared,target=/app alpine ash

> docker run --rm -tdi --name alpine2 --mount source=fs_shared,target=/app alpine ash

> docker run --rm -tdi --name alpine3 --mount source=fs_shared,target=/app alpine ash
```

NAYA College

Volumes - Lab

Attach to running containers, create files and verify files gets updated on all containers

TIPS:

- Use previous slides to create volume and container that running with mount to that volume.
- Disconnect sequence: **Ctrl+p+Ctrl+q**

STOP AND REMOVE

CLEAN UP

BIND mounts

```
> docker run --rm -tdi -v `pwd`/source:/app alpine ash
```

```
> docker run --rm -tdi --mount type=bind,source=`pwd`/source,target=/app alpine ash
```

NAYA College

BIND mounts using -V or --MOUNT?

Both will provide the same outcome but as -v /--volume exists since day 1 in docker and --mount was introduced since docker 17.06 it became normal and easier to use --mount.

BIND MOUNTS - Lab

- **Create and manage bind mount:**
  - Create new host local project folder called “nice_docker” and cd into it
    - Create 2 alpine containers and share new local folder called source1 using --mount
    - Create 2 alpine containers and share new local folder called source2 using -v
    - What happened when you tried creating a shared host folder with --mount without first creating the folder manually ? and what happened when you were using -v
  - Inspect the new volumes and containers
  - Validate shared folder by creating files and make sure the exists on both containers
  - Stop all docker containers and Make sure containers got deleted

Running BootStrap app in a container - Lab

- **Hook SpringBoot Jar into a container:**
  - Create temp directory
  - Cd into your “temp” folder
    - Copy from your cloned git the demo artifact to ./source

      from solaredge-k8s-course/Docker/spring-boot-music/artifacts/spring-music.jar
  - Run **1 new container**
    - Name: web_api
    - Mount Using -v or --mount
      - temp/source
      - Target: /app
    - Image: **levep79/alpine-oraclejre8**
    - CMD: `java -jar -Dspring.profiles.active  /app/spring-music.jar`

NAYA College

Running BootStrap app in a container - Lab

Validate your work:

docker ps, docker log, docker attach

Try Browsing from your host browser: http://<machine ip>:8080

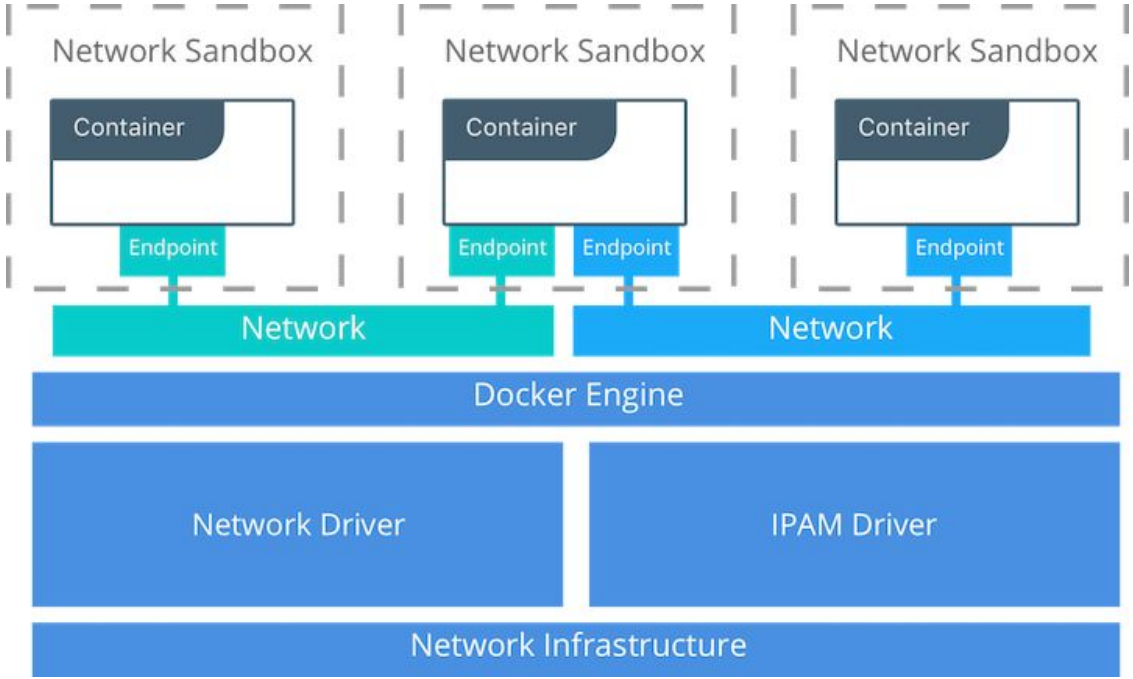
Did it worked?

What do you need to do to forward request to port 8080 and 8080 to your docker web_api?

Docker Networking

https://docs.docker.com/network/

The Container Networking Model - CNM



Network Native Network Driver

Host - With the **host** driver, a container uses the network stack of the host. There is no namespace separation, and all interfaces on the host can be used directly by the container.

Bridge - The basic and default driver which is used for standalone containers setup that need to communicate.

Overlay - Connect multiple docker daemons together and enable swarm (cluster) services to communicate with each other. This can be used to facilitate communication between swarm and standalone container or between two standalone containers on different docker daemons.

Network Native Network Driver

MACVLAN - Allow us to assign a MAC address to a container for making it appear as physical device on our network. Best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address. Macvlan usually to be used with legacy or HW required product that must have a MAC and being directly connected to the physical network to operate.

None - The none driver gives a container its own networking stack and network namespace but does not configure interfaces inside the container. Without additional configuration, the container is completely isolated from the host networking stack.

Networking - Lab

```
> docker network ls
```

```
> docker run --rm -tdi --name alpine1 alpine ash
```

```
> docker run --rm -tdi --name alpine2 alpine ash
```

1. Check that the containers are actually running
2. Inspect the network and see what containers are connected to it using

   docker network inspect bridge
3. Connect to one of the Alpine containers using **docker attach or exec** and ping the other container with IP and than with it’s name. What happened ?

NAYA College

Networking - Lab

```
> docker network create dmz
> docker network inspect dmz
> docker run -tdi --rm --name network_test --network dmz alpine ash
> docker inspect network_test
```

NAYA College

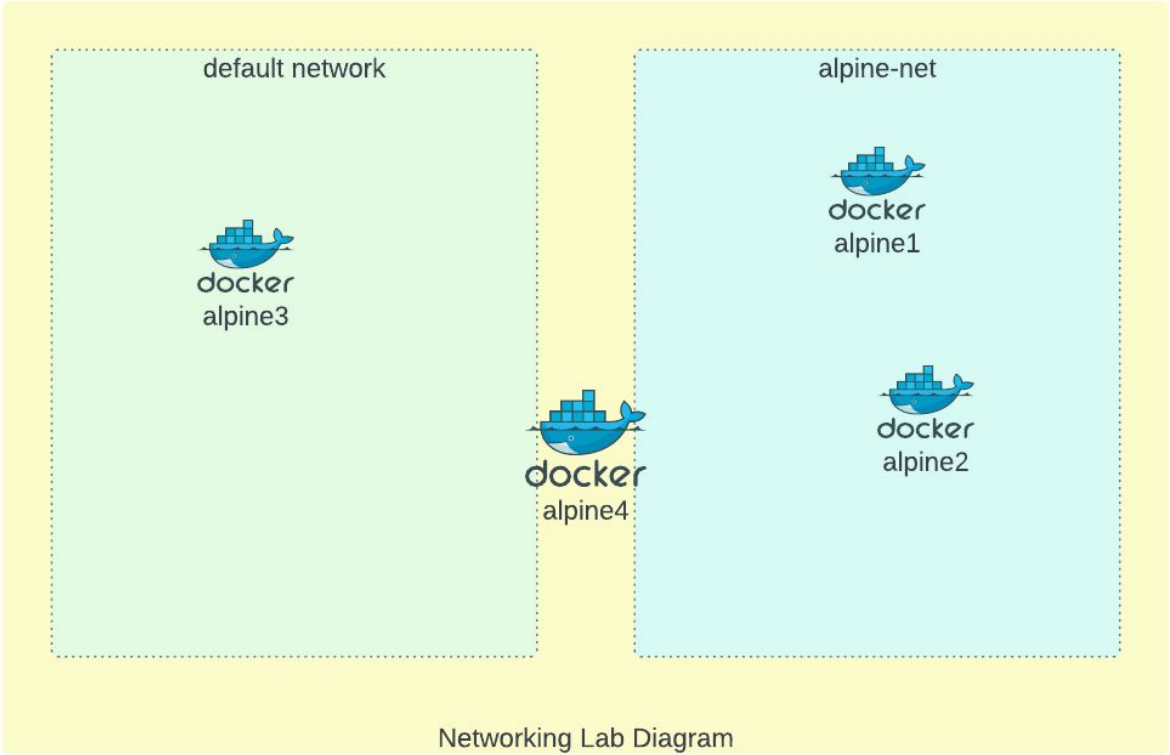
Networking - Lab

1. Delete the previous containers (stop and then remove)
2. Create a newly user Defined network bridge named “**alpine-net”** and verify creation with **network ls** and than **Inspect** the network to see that no containers are connected
3. Create 4 new alpine containers with -dit and --network to the following network configuration
    a. First two to: alpine-net
    b. 3rd one to the **default bridge**
    c. 4th one to **alpine-net &** to the **bridge** network **(trickey…)**

    Tip: *network connect*...
4. Inspect Network bridge and user defined network

NAYA College

Lab Architecture Diagram



Networking - Lab

5. Connect to alpine1 and try pinging to alpine1,2,3,4 with IP and DNS - What happened ?
6. Connect to alpine4 and try pinging to alpine1,2,3,4 with IP and DNS - What happened ?
7. Why?
8. Stop all containers , Remove them and delete the user defined network you created

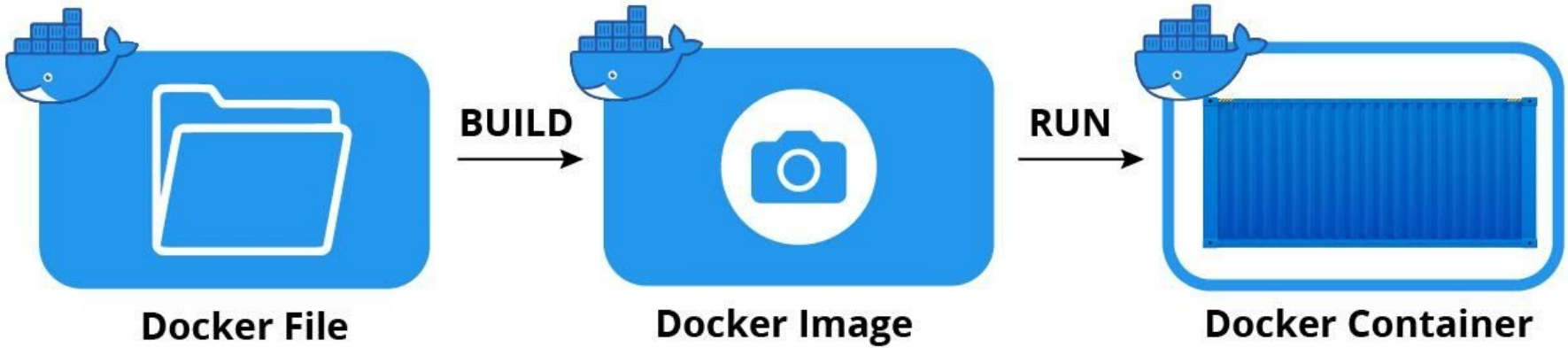
Docker Hub & Registry

Local registry

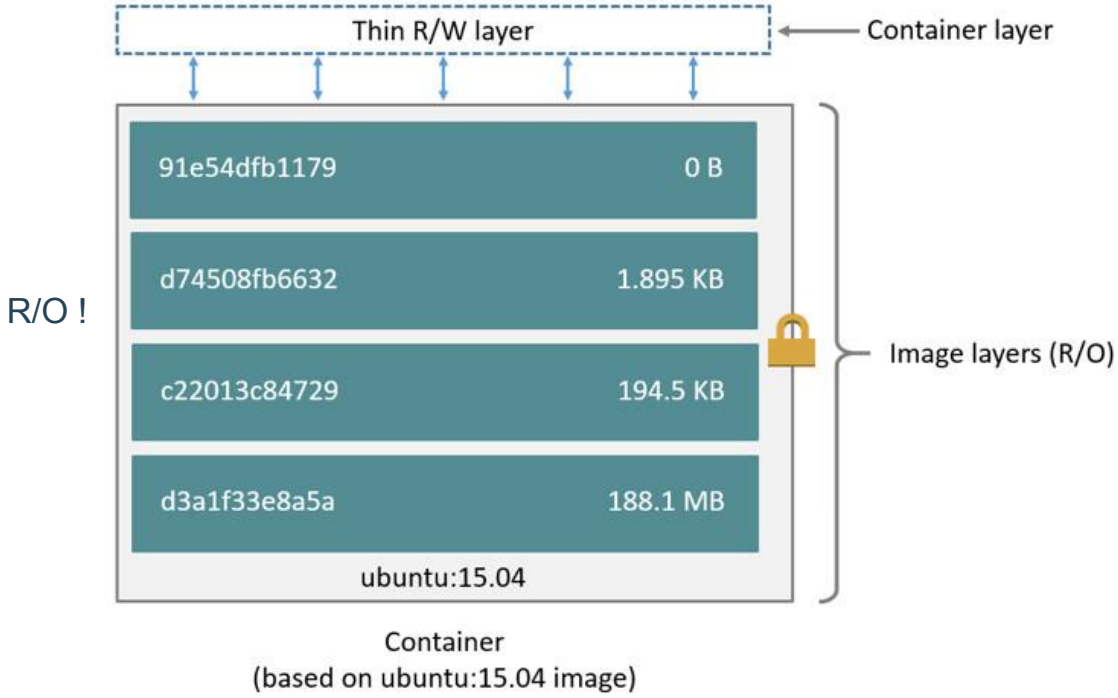
```
> docker run -d -p 5000:5000 --name registry registry:2.7

> docker ps

>  docker tag mysql localhost:5000/mysql

> docker push localhost:5000/mysql
```

NAYA College

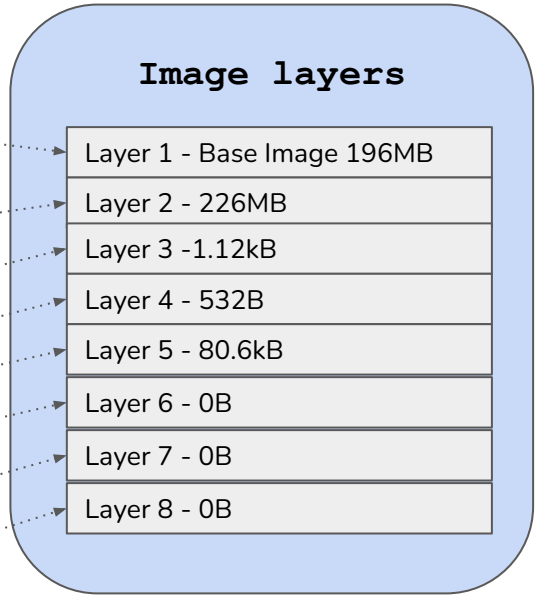
Docker Images


BUILD
RUN
Docker File
Docker Image
Docker Container

Layered FS


Thin R/W layer
Container layer
91e54dfb1179
0 B
d74508fb6632
1.895 KB
c22013c84729
194.5 KB
d3a1f33e8a5a
188.1 MB
ubuntu:15.04
Image layers (R/O)
R/O !
Container
(based on ubuntu:15.04 image)

```
#
# Ubuntu Dockerfile
#
# https://github.com/dockerfile/ubuntu
#

# Pull base image.
FROM ubuntu:14.04

# Install.
RUN \
  sed -i 's/# \(.*multiverse$\)/\1/g' /etc/apt/sources.list && \
  apt-get update && \
  apt-get -y upgrade && \
  apt-get install -y build-essential && \
  apt-get install -y software-properties-common && \
  apt-get install -y byobu curl git htop man unzip vim wget && \
  rm -rf /var/lib/apt/lists/*

# Add files.
ADD root/.bashrc /root/.bashrc
ADD root/.gitconfig /root/.gitconfig
ADD root/.scripts /root/.scripts

# Set environment variables.
ENV HOME /root

# Define working directory.
WORKDIR /root

# Define default command.
CMD ["bash"]
```



Command	Overview
FROM	Specify base image
RUN	Execute specified command
ENTRYPOINT	Specify the command to execute the container
CMD	Specify the command at the time of container execution (can be overwritten)
COPY	Simple copy of files / directories from host machine to container image
ADD	COPY + unzip / download from URL (**not recommended**)
ENV	Add environment variables
EXPOSE	Open designated port
WORKDIR	Change current directory
MAINTAINER	**deprecated**
	now `LABEL maintainer="maintainer@example.com"`should be specified as

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

CMD and ENTRYPOINT

- **CMD** sets default command and/or parameters, which can be overwritten from command line when docker container runs.

- **ENTRYPOINT** configures a container that will run as an executable.

Shell form

```
CMD echo "Hello world"
```

```
ENTRYPOINT echo "Hello world"
```

Exec form

```
CMD ["/bin/echo", "Hello world"]
```

```
ENTRYPOINT ["/bin/echo", "Hello world"]
```

NAYA College

how CMD and ENTRYPOINT interact

- Dockerfile should specify at least one of **CMD** or **ENTRYPOINT** commands.
- **ENTRYPOINT** should be defined when using the container as an executable.
- **CMD** should be used as a way of defining default arguments for an **ENTRYPOINT** command or for executing an ad-hoc command in a container.
- **CMD** will be overridden when running the container with alternative arguments.

ADD and COPY

- COPY will work for most of the cases.

- ADD has all capabilities of COPY and has the following additional features:

  Allows tar file auto-extraction in the image, for example,

  ```
  ADD app.tar.gz /opt/var/myapp
  ```

  Allows files to be downloaded from a remote URL.

ENV

Environment variables (declared with the **ENV** statement) can also be used in certain instructions as variables to be interpreted by the Dockerfile.

```
FROM busybox

ENV FOO=/bar

WORKDIR ${FOO}   # WORKDIR /bar

ADD . $FOO       # ADD . /bar

COPY \$FOO /quux # COPY $FOO /quux
```

NAYA College

USER

If a service can run without privileges, use **USER** to change to a non-root user. Start by creating the user and group in the Dockerfile with something like

```
RUN groupadd -r postgres && useradd --no-log-init -r -g postgres postgres
```

Avoid installing or using **sudo** as it has unpredictable TTY and signal-forwarding behavior that can cause problems. If you absolutely need functionality similar to **sudo**, such as **initializing the daemon as root but running it as non-root**, consider using “**gosu**”.

Dockerfile - LAB

1. Make a new folder in your project directory called course_dockerfile
2. Copy spring-music.jar from solaredge-k8s-course/Docker/spring-boot-music/artifacts to a new folder course_dockerfile/artifacts
3. Create an empty dockerfile

SPEC

From: levep79/jdk-alpine:latest

Workdir /app

Copy: artifact to /app

Expose: 8080

CMD: java -jar -Dspring.profiles.active=none spring-music.jar

Build && Run image

NAYA College

BuildKit

BuildKit provides new functionality and improves your builds' performance. It also introduces support for handling more complex scenarios:

- Detect and skip executing unused build stages
- Parallelize building independent build stages
- Incrementally transfer only the changed files in your build context between builds
- Detect and skip transferring unused files in your build context
- Use Dockerfile frontend implementations with many new features
- Avoid side effects with rest of the API (intermediate images and containers)
- Prioritize your build cache for automatic pruning

https://docs.docker.com/develop/develop-images/build_enhancements/

Import and export images

Docker images can be saved using image save command to a .tar file:

```
> docker image save helloworld > helloworld.tar
```

These tar files can then be imported using load command:

```
> docker image load -i helloworld.tar
```

Multi-stage builds

With multi-stage builds, you use multiple FROM statements in your Dockerfile. Each FROM instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don't want in the final image.

Multi-stage builds Lab

Follow README.md:

solaredge-k8s-course/Docker/spring-boot-simple-app

Run the Docker daemon as a non-root user

https://docs.docker.com/engine/security/rootless/

Docker Scout

https://docs.docker.com/scout/

Dockerfile best practices

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

Develop with Docker

https://docs.docker.com/develop/

NAYA College

Dockerfile best practices

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

Develop with Docker

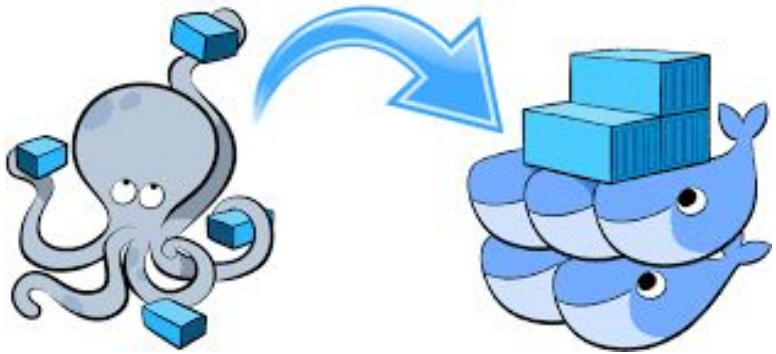
https://docs.docker.com/develop/

Docker Compose

https://docs.docker.com/compose/

Docker-compose

Docker compose manges our application lifecycle



NAYA College

Docker-compose

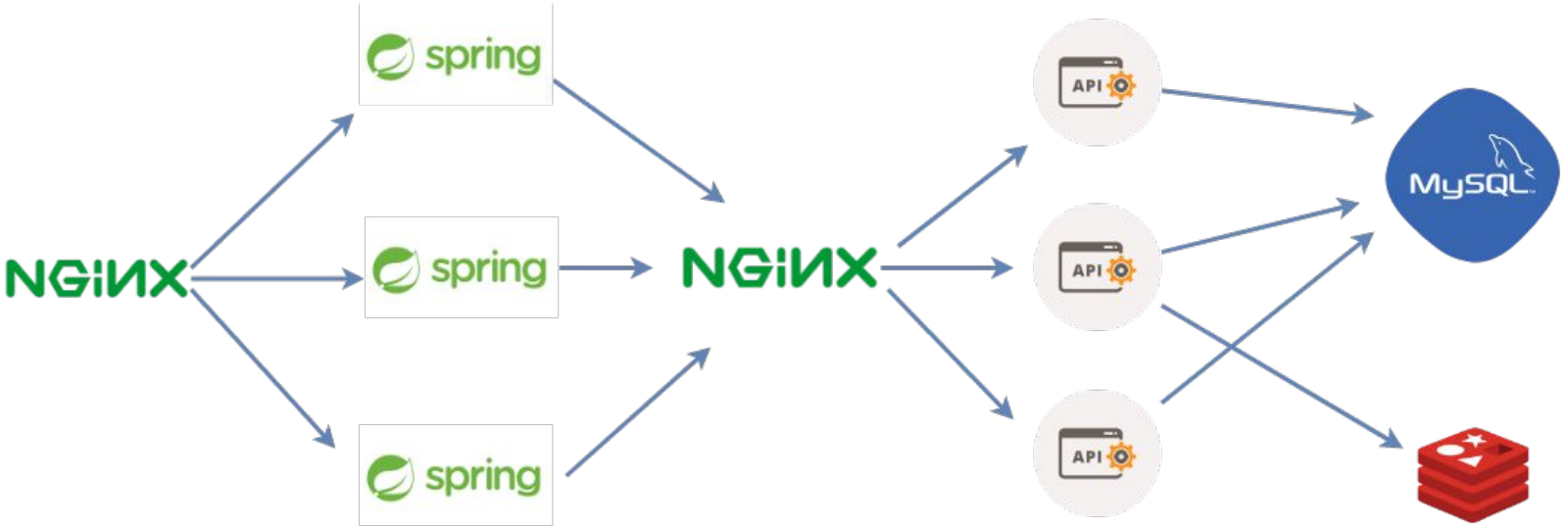
Compose is a tool for defining and running **multi-container Docker applications.** With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

It also has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Why do we need docker compose?

Imagine managing this manually



NAYA College

Why do we need docker compose?

Using docker-compose.yml we can define

- Networking
- Dependencies between services
- Environments
- What makes up a role and its components
- Manage each application services we got

Docker compose flow

Build Services
(dockerfiles)

Generate images

Start our services

Take them down

With Compose, you use a YAML file to configure your application's services

```yaml
version: "3.9"  # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

With a single command, you create and start all the services from your configuration.

```
$ docker-compose up
```

or daemon mode:

```
$ docker-compose up -d
```

Using Compose is basically three-step

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.

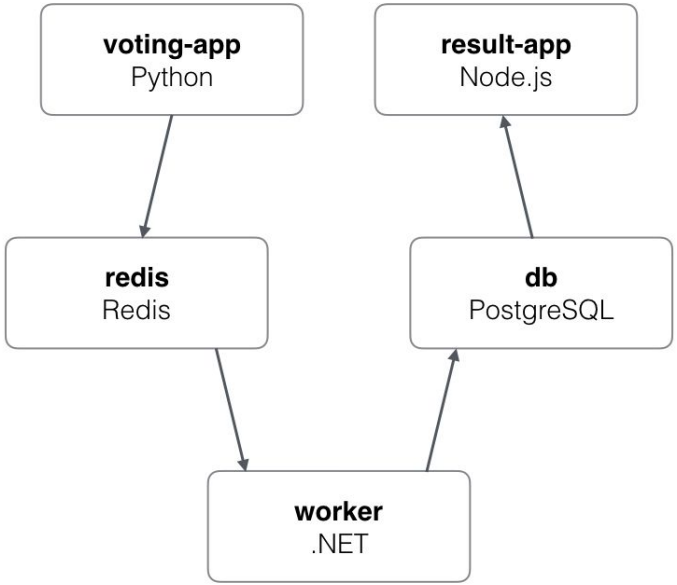
1. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.

1. Run docker-compose up and Compose starts and runs your entire app.

Demo

Demo 2 - The Voting App

1. **Voting-App**: Frontend of the application written in Python, used by users to cast their votes.
2. **Redis**: In-memory database, used as intermediate storage.
3. **Worker**: .Net service, used to fetch votes from Redis and store in Postres database.
4. **DB**: PostgreSQL database, used as database.
5. **Result-App**: Frontend of the application written in Node.js, displays the voting results.



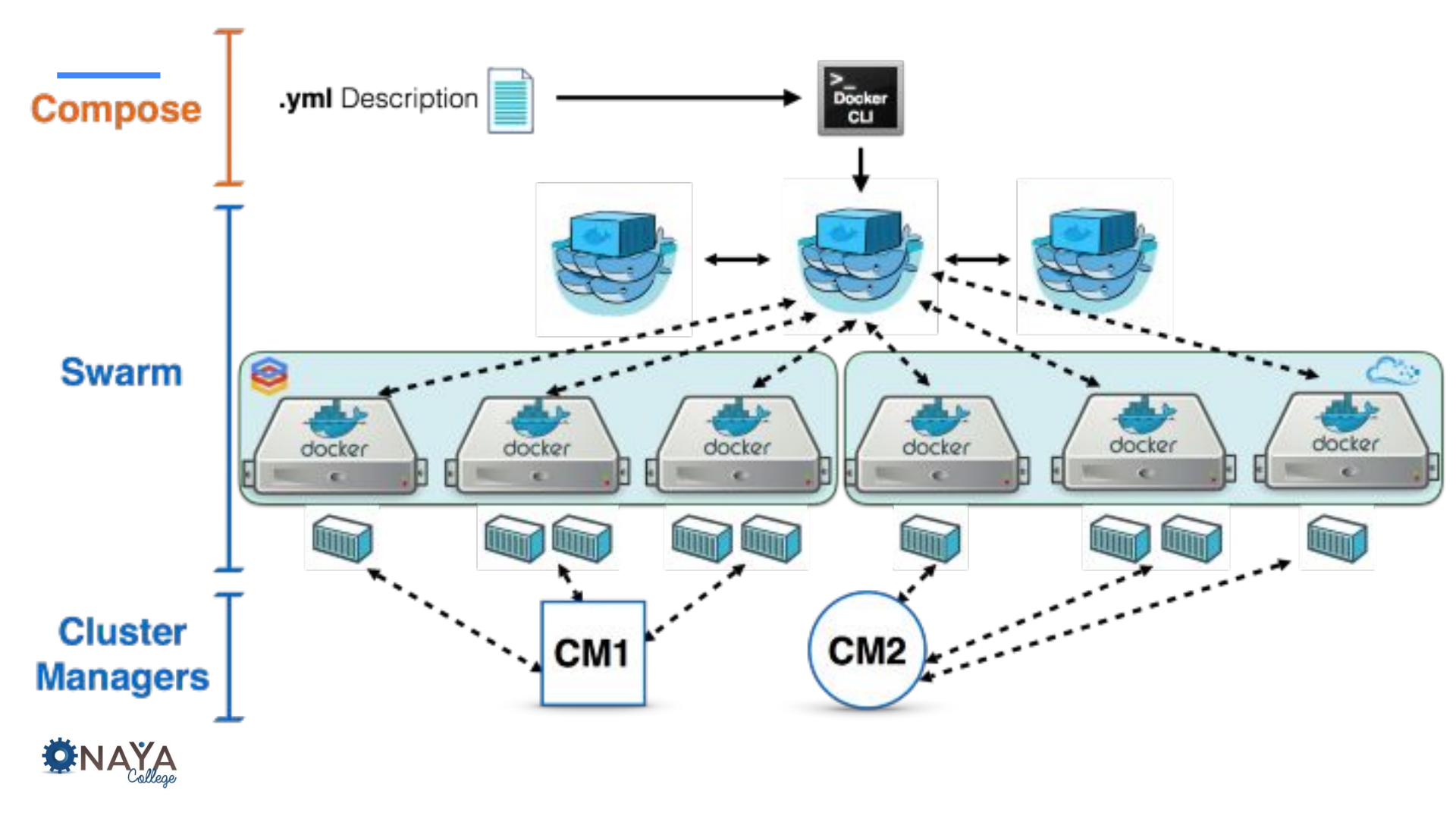
NAYA College

Demo 2 - The Voting App

```
> docker-compose up

> docker ps -a --format="table {{.Names}}\t{{.Image}}\t{{.Ports}}"

> docker-compose up -d
> docker-compose down
> docker-compose start
> docker-compose stop
> docker-compose build
> docker-compose logs -f db
> docker-compose up –scale worker=4
> docker-compose events
> docker-compose exec db bash
```

NAYA College

Docker Swarm


Compose
.yml Description
Docker CLI
Swarm
docker
docker
docker
docker
docker
docker
Cluster Managers
CM1
CM2
NAYA College