# Docker For DevOps

Lev Epshtein


# About me

**Lev Epshtein**

Technology enthusiast with 10 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.

DevOps, and Big Data instructor at NAYA.

Partner & Solution Architect Consultant at Opsguru.

**lev@opsguru.io**

# Docker Basics

# Objectives

- By the end of this module
  - You'll be familiar with Docker concepts & Base command
  - Configure Dockers using Dockerfile and Passing Properties To it
  - Run standalone Jar in docker
  - Operate docker registry (docker hub)
  - Build docker images with Maven
  - Docker Network, Docker compose

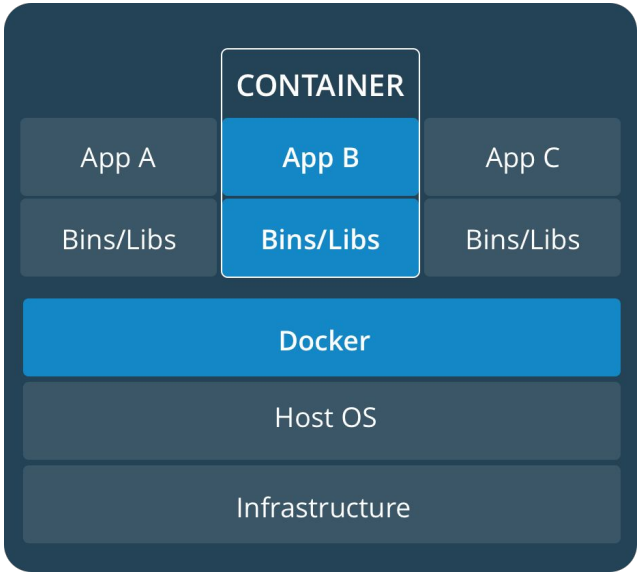

# Questions for you…

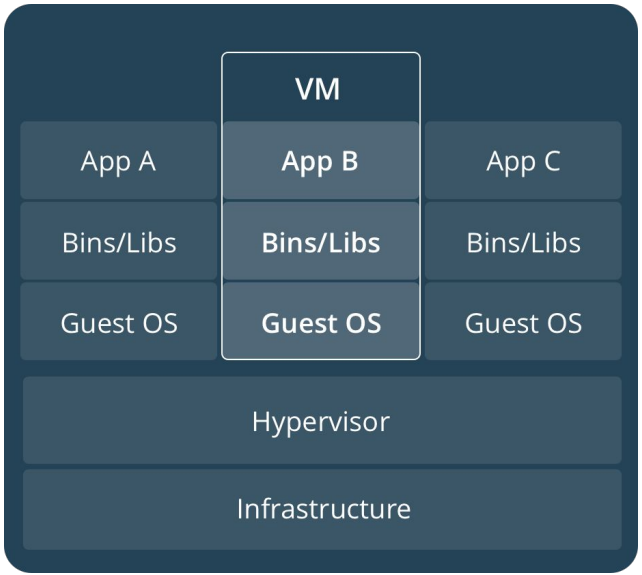
- What do you know about Docker?
- Who use docker for Development/QA/STG/PROD?
- Who tried and failed implementing Docker?

# What is Docker?

- Docker is a platform for developers and sysadmins to **build, share, and run** applications with containers.
- The use of containers to deploy applications is called *containerization*.
  - Flexible
  - Lightweight
  - Portable
  - Loosely coupled
  - Scalable
  - Secure

# Containers and virtual machines

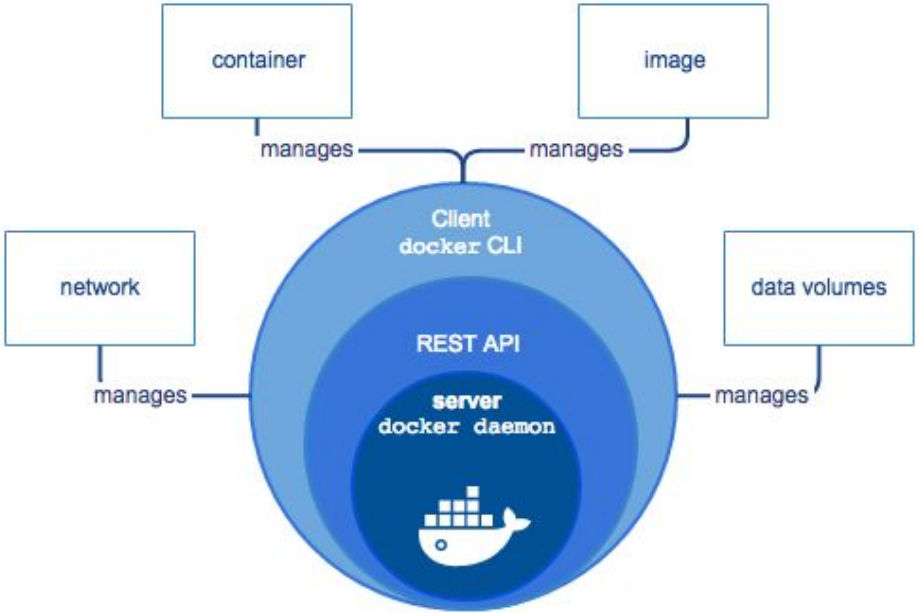


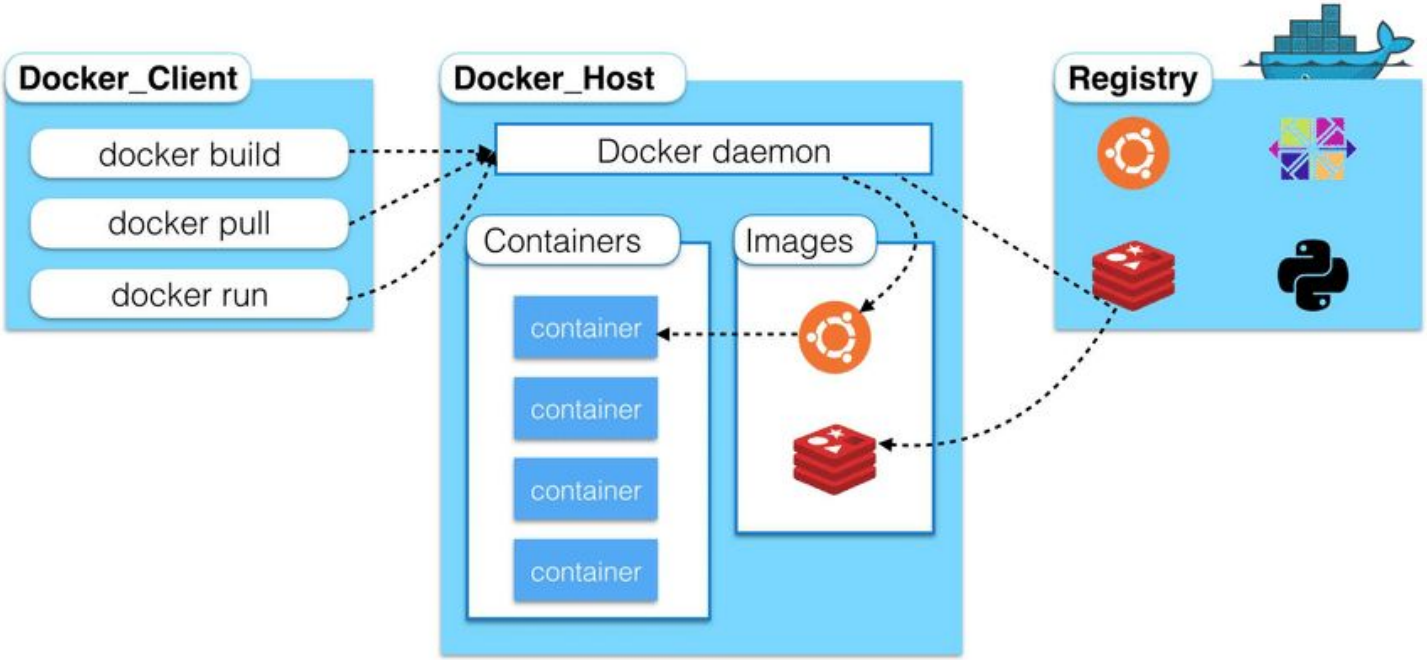
# Common Use Case for Docker

- Microservices / Multi-Tier application (Front End, Mid Tier, Data Tier)
- CI/CD
- Sandboxed environment
- Local Environment

# Docker Engine



# The Docker Architecture




# The underlying technology

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

- Namespaces
- Control groups

# Docker installation on RedHat

*> sudo yum update*

*> curl -sSL https://get.docker.com/ | sh*

*> sudo usermod -aG docker <User Name>*

*> sudo vi /usr/lib/systemd/system/docker.service*

**Add the following line to docker.service in the [Service] header.**

TimeoutStartSec=0

*> sudo service docker start*

Check by running: *> docker info*

# Docker installation on RedHat

Other OS installation you can find in documentation:

https://docs.docker.com/search/?q=installation

# Let’s Start

git clone https://github.com/ops-guru/nice-devops-supplementary.git

*[vagrant@localhost ~]$* ***docker run alpine echo "Hello World"***
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
89d9c30c1d48: Pull complete
Digest: sha256:c19173c5ada610a5989151111163d28a67368362762534d8a8121ce95cf2bd5a
Status: Downloaded newer image for alpine:latest
Hello World
*[vagrant@localhost ~]$* ***docker run -i -t alpine ash***
***/ # echo "Hello from container"***
Hello from container
***/ # cat /etc/os-release***
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.10.3
PRETTY_NAME="Alpine Linux v3.10"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
***/ # exit***

# Docker Basic Commands

***> docker run -i -t -d --name nicedocker -p 8080:80 alpine ash***

- docker run will run the container
  - -i  - Interactive mode
  - -t  - Allocate pseudo  TTY - or not Terminal will be available
  - -d  - Run in background (Daemon style)
  - --name  - Give the container a name or let Docker to name it
  - -p [local port]:[container [port]  - Forward local to the container port

# Docker Basic Commands

***> docker run -h niceDocker -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)***

# Common Docker commands

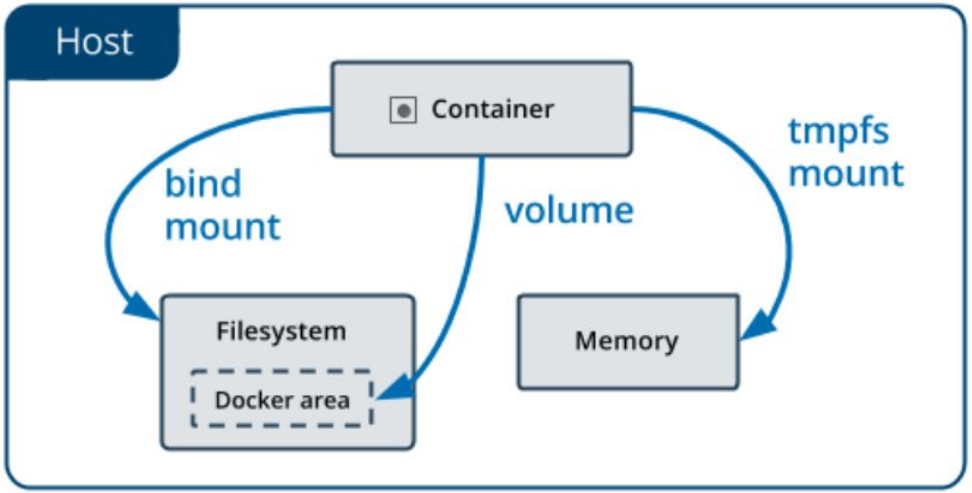
```
// General info
man docker // man docker-run
docker help // docker help run
docker info
docker version
docker network ls
docker volumes ls

// 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]
```

# Docker Volumes

# 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.



# Volumes - Demo

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

> ***docker inspect alpine1***

> ***docker stop alpine1 && docker rm alpine1***

# 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***

# 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**

# 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:**
  - Cd into your “nice_docker” folder
    - Copy from your cloned git the demo artifact to ./source

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

# 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**

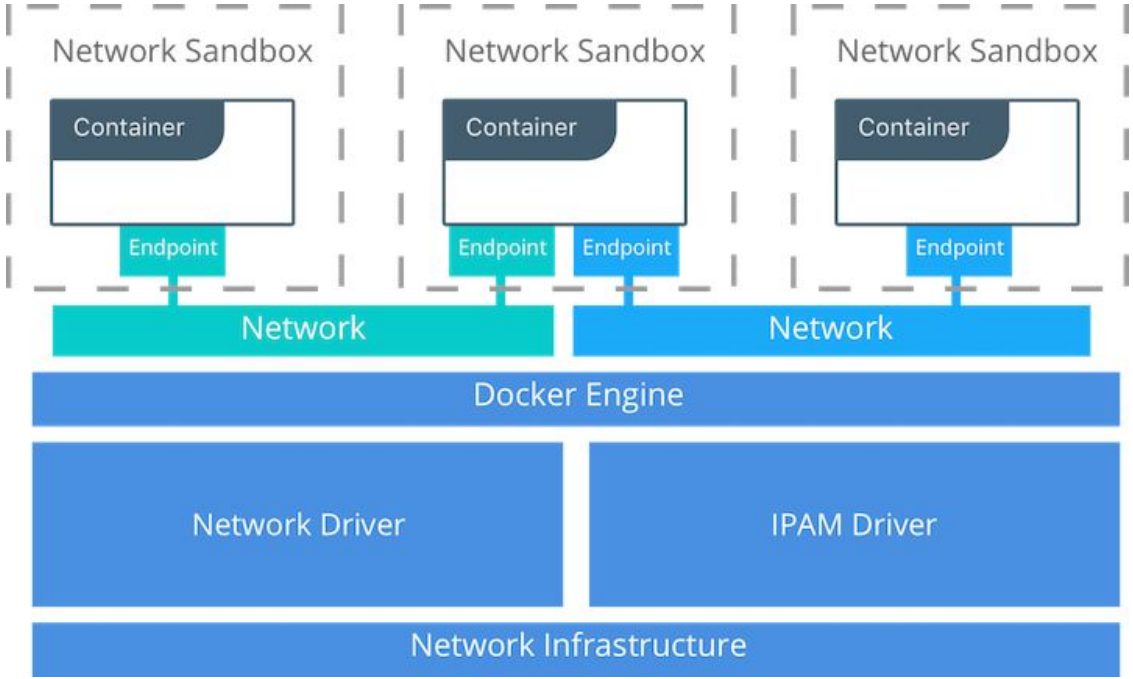
Did it worked?

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 ?

# 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***

# 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 **dmz** network **(trickey…)**

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

# Networking - Lab

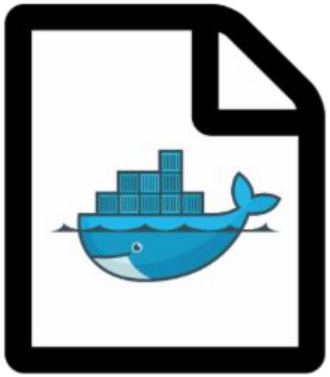
5. Connect to alpine1 and try pinging to alpine1,2,3,4 with IP and DNS - What happened ?
6. Connect to alpine4 and try pinging to alpine1,2,3,4 with IP and DNS - What happened ?
7. Why?
8. Stop all containers , Remove them and delete the user defined network you created

# Docker Images

# Dockerfile

| Command | Purpose | Example |
|---|---|---|
| FROM | First non-comment instruction in Dockerfile | FROM ubuntu |
| COPY | Copies mulitple source files from the context to the file system of the container at the specified path | COPY .bash_profile /home |
| ENV | Sets the environment variable | ENV HOSTNAME=test |
| RUN | Executes a command | RUN apt-get update |
| CMD | Defaults for an executing container | CMD ["/bin/echo", "hello world"] |
| EXPOSE | Informs the network ports that the container will listen on | EXPOSE 8093 |

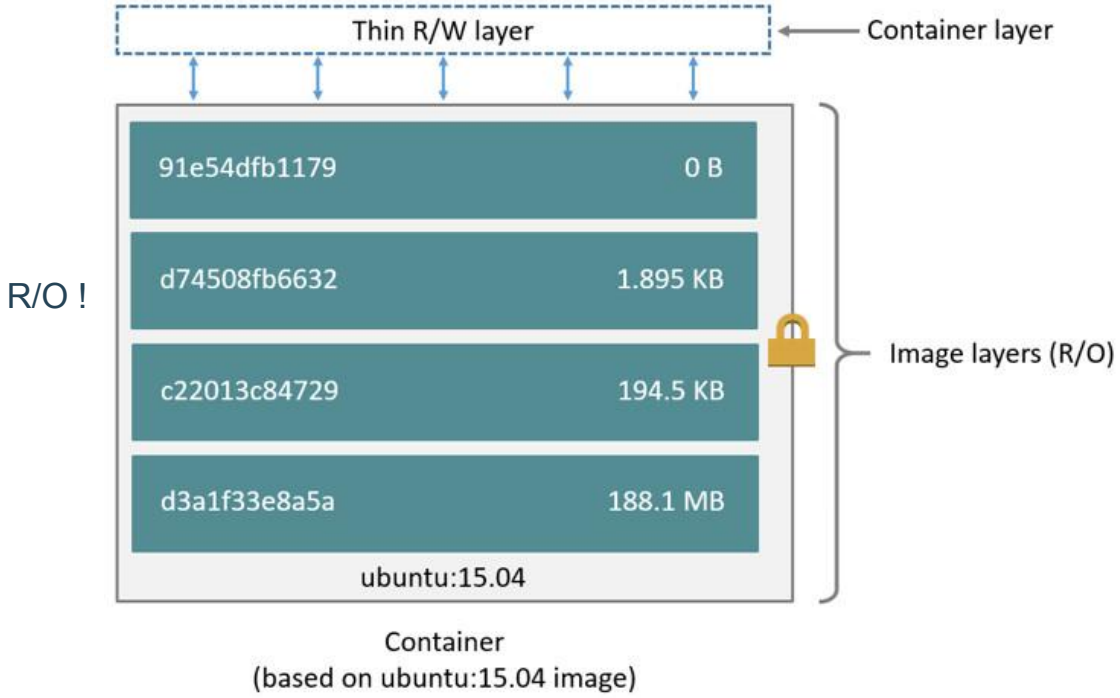
# Dockerfile



Dockerfile

| BUILD | Boot | Run |
|---|---|---|
| FROM | WORKDIR | CMD |
| | USER | ENV |
| COPY | | EXPOSE |
| ADD | | VOLUME |
| RUN | | ENTRYPOINT |
| ONBUILD | | |
| .dockerignore | | |

# 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)

# Difference between CMD and ENTRYPOINT

- CMD will work for most of the cases.
- ENTRYPOINT allows to override the entry point to some other command, and even customize it.

For example:

*> docker container run -it ubuntu*

*> docker container run -it --entrypoint=/bin/cat ubuntu /etc/passwd*

# Difference between 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.

# 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***

# Dockerfile best practices

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

# Docker Compose

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

# Overview

- 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.
- with a single command, you create and start all the services from your configuration.
- Compose has traditionally been focused on development and testing workflows.
- Production-oriented features, see compose in production

# Using Compose is basically three-step

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker-compose up and Compose starts and runs your entire app.

```
version: '3'
services:
  web:
    build: .
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    - logvolume01:/var/log
    links:
    - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

# Demo 1

$ cd <>nice-devops-supplementary/Docker/docker-compose/compose-file-demo1

$ docker-compose up

$ Ctrl + c , docker-compose down

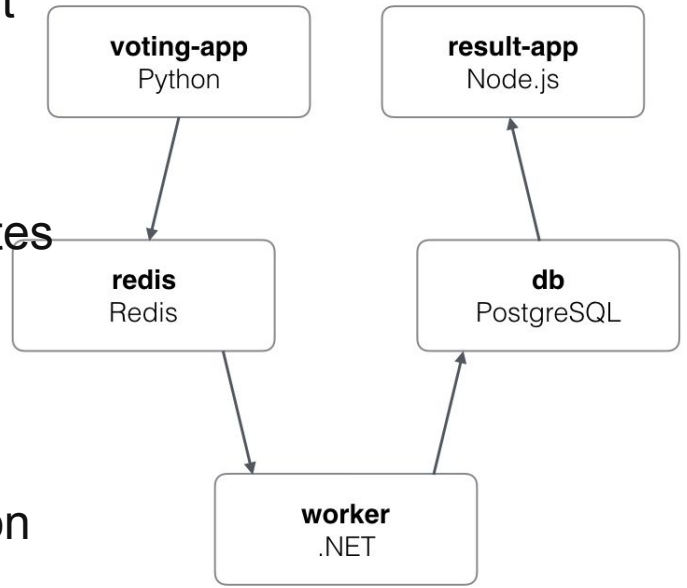
$ docker-compose up -d

$ docker-compose ps

$ docker-compose down

# 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.



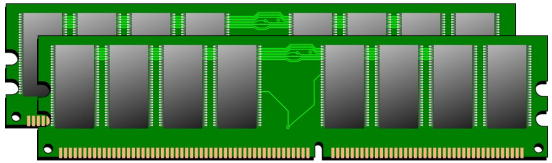
# Demo 2 - The Voting App

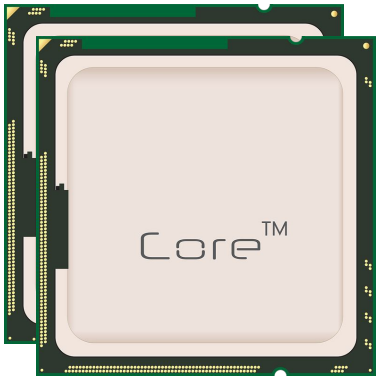
```
$ docker-compose up$ docker ps -a --format="table {{.Names}}\t{{.Image}}\t{{.Ports}}"
```

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

# Resources Limitation
# &
# Containers in Production

# Risk of running out of resources






```
docker run -it --cpus=".5" ubuntu /bin/bash
```


# Container in Production

## PROS

- Containers makes our applications "virtually look" the same on most infrastructures (Physical/Cloud/VM's)
- Containers makes our application runtime dependencies the developers's responsibility - **splendid**!
- Containers require the application developers to consider application state and persistence.
- Containers, once built, provide a (mostly) consistent behavior between dev, staging, and production environments. Immutable delivery mechanism out of the box
- Blazing fast scaling our ecosystem
- Delivery time - Days and hours becomes minutes / seconds
- Handoff - Developers <> Operators - Wall of confusion ? **breached**!


# Container in Production

CONS

- Containers do not make your applications more secure.
- Containers do not make your applications more scalable - **This is a common misconception**
- Containers do not make your applications more portable - **Shared / Common libraries in your code?**
- Network - NAT managed by Dockerd is not how we want to work in production

**All of the above becomes absolute once we move to K8S and Swarm (new Pros & cons but different...)**