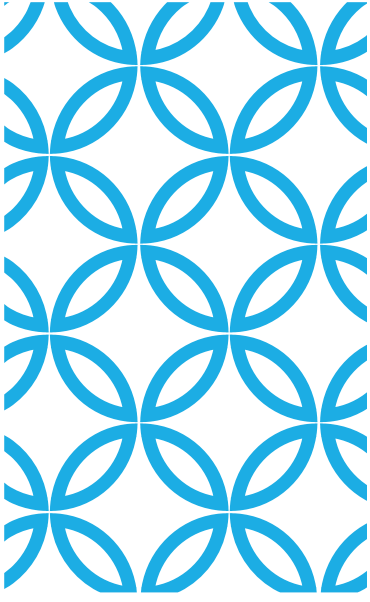
docker

TRAINING OUTLINE

- Introduction
- Docker Architecture
- Installing Docker
- Docker components
- Docker Commands
- Docker Images
- Managing docker Containers
- Dockerfile
- Docker volume
- Docker networking
- YAML Introduction
- Docker Compose
- Security Strategies
- The ecosystem & the future



DOCKER INTRODUCTION

DOCKER HISTORY

Docker was created by dotCloud and launch in March 13, 2013 at PyCon Lighting Talk

dotCloud was a PaaS platform company (now Docker Inc)

Solomon Hykes is the father of Docker

Hykes had a cofounder who's now at a partner company (mesosphere)

Hykes never liked Docker's name

INTRODUCTION

Docker is an open-source tool for the creation, deployment, and working of applications on a centralized platform.

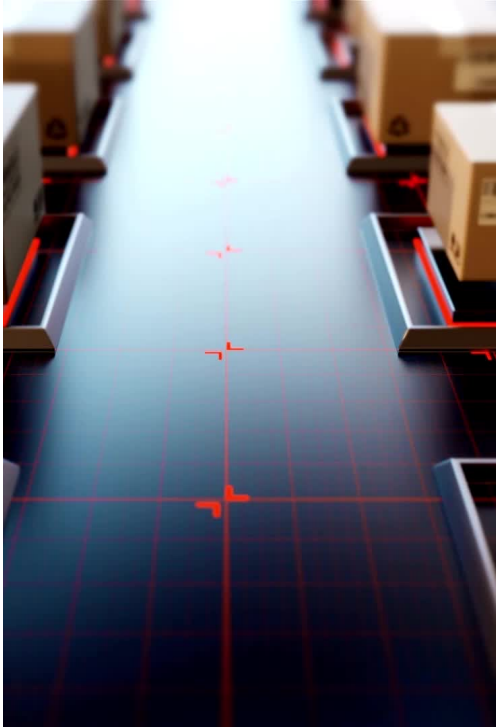
It brings together the kernel namespaces, cgroups, capabilities and all of that stuff into a product

Docker provides a very uniform and standard runtime

Docker is growing more than just a container runtime, becoming more of a platform (registry, clustering, orchestration, networking, etc)

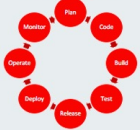
Features provided by Docker are as follows

- Docker is Fast and Easy configurable.
- Technical feasibility and increased productivity.
- Secure services through commands like secret inspect and secret create, etc.
- Provides application isolation and no container is dependent on any other.

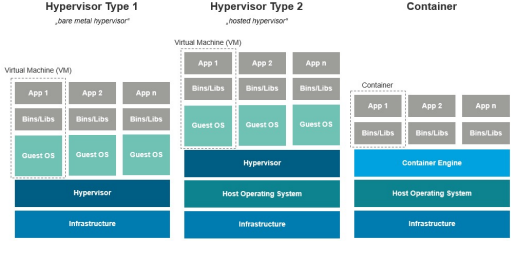


HISTORY AND MULTI-DIMENSIONAL EVOLUTION OF COMPUTING

Development Process	Application Architecture	Deployment and Packaging	Application Infrastructure
Waterfall	Monolithic	Physical Server	Datacenter
Agile	N-Tier	Virtual Servers	Hosted
DevOps	Microservices	Containers	Cloud



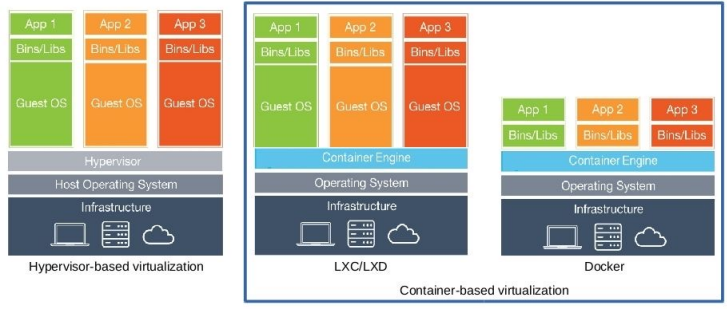
CONTAINER VS VIRTUAL MACHINE



- A VM has emulated hardware and hosts a whole Operating System (guest), which is separate from the host O.S.
- A container does not emulate any hardware, and shares the O.S. kernel with the host → less isolation, more efficiency

CONTAINER TECHNOLOGIES ON LINUX

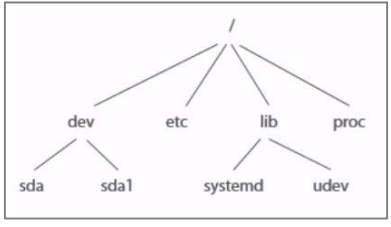
- Several light virtualization technologies are available for Linux
  - Is an operating-system-level virtualization method.
  - They build on cgroups, namespaces and other containment functionalities
  - LXC (Linux Containers) and Docker are the most popular products
  - Run multiple isolated Linux systems (containers) on a control host using a single Linux kernel.

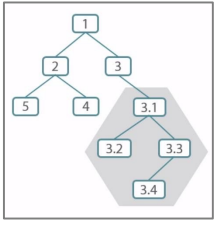


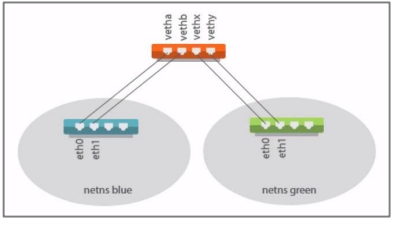
WHAT IS A LINUX CONTAINER (LXC)?

- LXC have Independent:
  - File System
  - Process Tree
  - Networking Stacks

- LXC are implemented using:
- namespaces → Isolation
- cgroups → Apply limits
- UFS capabilities → Manage privileges



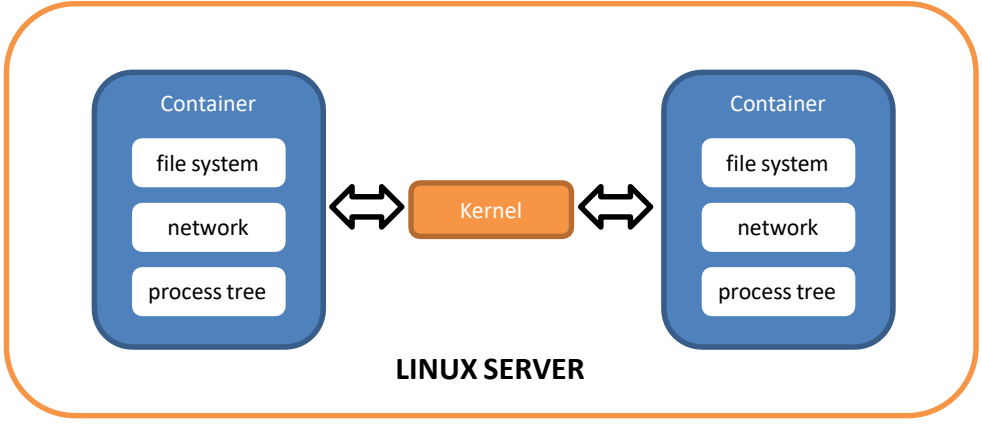




LXC VS DOCKER

Parameter	LXC	Docker
Developed by	LXC was created by IBM, Virtuozzo, Google and Eric Biederman.	Docker was created by Solomon Hykes in 2003.
Data Retrieval	LXC does not support data retrieval after it is processed.	Data retrieval is supported in Docker.
Usability	It is a multi-purpose solution for virtualization.	It is single purpose solution.
Platform	LXC is supported only on Linux platform.	Docker is platform dependent.
Virtualization	LXC provides us full system virtualization.	Docker provides application virtualization.
Cloud support	There is no need for cloud storage as Linux provides each feature.	The need of cloud storage is required for a sizeable ecosystem.
Popularity	Due to some constraints LXC is not much popular among the developers.	Docker is popular due to containers and it took containers to a next level.

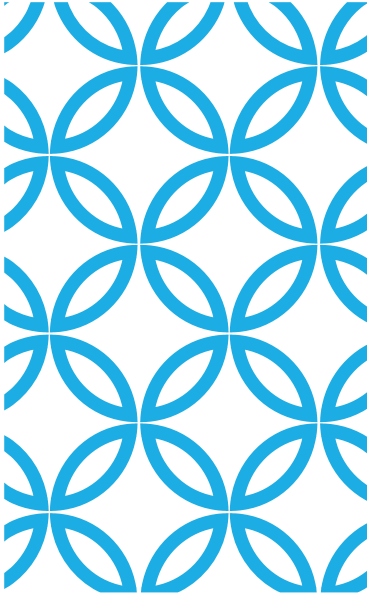
CONTAINER STRUCTURE



CONTAINERS ADVANTAGES

- More lightweight than VM's
- Containers consumes less CPU, less RAM and less diskspace.
- Every container shares a single common linux kernel in the host Containers are faster and more portable than VM's
- Provides a secure isolated runtime environment for each container

(No more operating system for each application)



DOCKER ARCHITECTURE

DOCKER OVERVIEW

(Shipping Yard)

(Manifests)

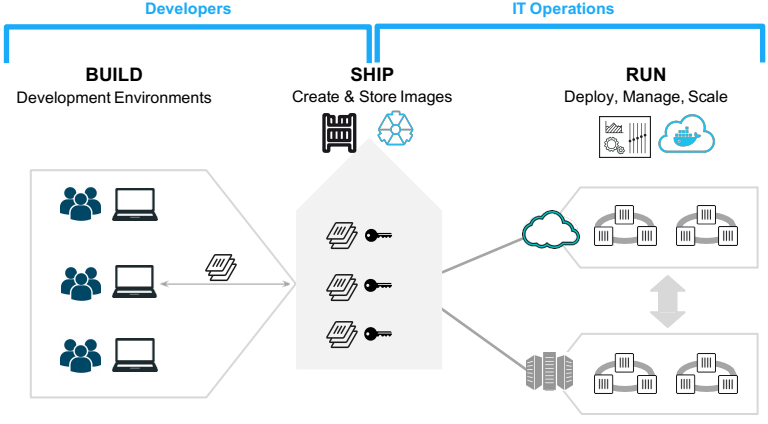
(Shipping Containers)

Shipping Yard: The infrastructure that's needed to import and export goods

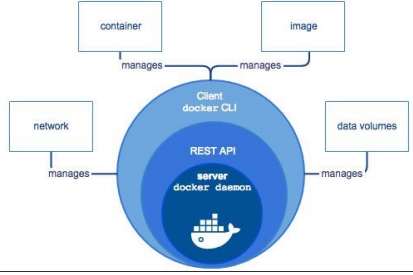
Manifests: List of the container content plus instructions on how to build it

Containers: A package with all the goods ready to be imported/exported

USING DOCKER: BUILD, SHIP, RUN WORKFLOW



DOCKER ARCHITECTURE



Client-server architecture to to manage images, containers, volumes and virtual networks

Client and server may run on different machines

- The Docker daemon

Receives and processes incoming Docker API requests.

- The Docker client

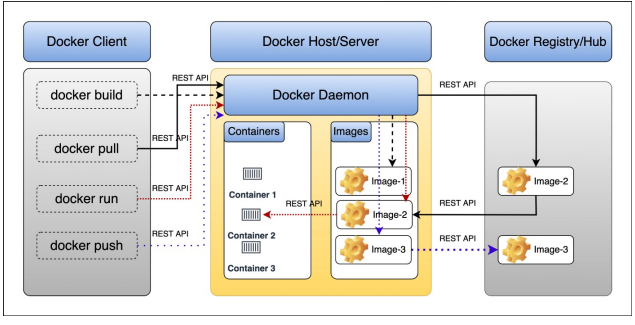
Command line tool - the docker binary.

Talks to the Docker daemon via the Docker API.

- Docker Hub Registry

Public image registry.

The Docker daemon talks to it via the registry API.

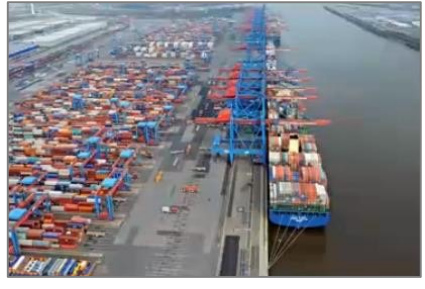


THE DOCKER ENGINE (DEAMON)

It's our Docker program that we install on each Docker host to provide us with a Docker environment and access to all Docker services.

Contains the application infrastructure and runtime dependencies (Standardized).

Run in the same way in your laptop, datacenter or in the cloud.



DOCKER IMAGES

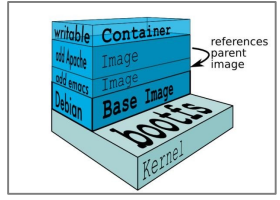
Are read only templates used to create containers

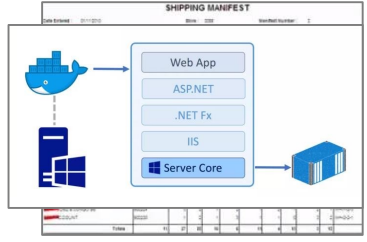
Images are comprised of multiple layers

The first image is called the “Base Image”

Intermediate layers increases reusability, decrease disk usage, and speed up the build

The higher layers win where there are conflicts





DOCKER CONTAINERS

We can think of them as running instances of an image.

Containers are created from images. Inside a container, it has all the binaries and dependencies needed to run the application.

Under the hood containers are Linux processes running in the Docker host

DOCKER REGISTRIES

Is a stateless, highly scalable server side application that stores and lets you distribute Docker images.

A docker registry contains multiple repositories (can be private or public)

You can pull/push images from repositories

Docker Hub is the default Docker Registry

SOME DOCKER VOCABULARY

Docker Image

The basis of a Docker container. Represents a full application

Docker Container

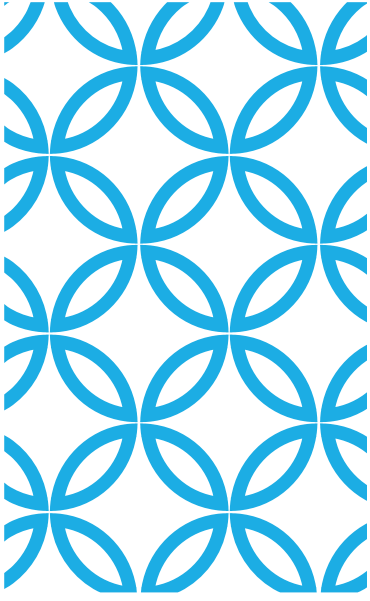
The standard unit in which the application service resides and executes

Docker Engine

Creates, ships and runs Docker containers deployable on a physical or virtual, host locally, in a datacenter or cloud service provider

Registry Service (Docker Hub(Public) or Docker Trusted Registry(Private))

Cloud or server based storage and distribution service for your images



INSTALLING DOCKER

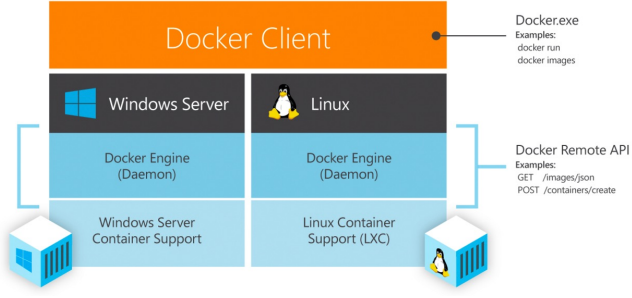
EDITIONS AND VERSIONS

Capabilities	Docker Engine - Community	Docker Engine - Enterprise	Docker Enterprise
Container engine and built in orchestration, networking, security	✓	✓	✓
Certified infrastructure, plugins and ISV containers		✓	✓
Image management			✓
Container app management			✓
Image security scanning			✓

The **Stable** version gives you reliable updates every quarter
The **Edge** version gives you new features every month

DOCKER – NATIVE SUPPORT

- We can run both Linux and Windows programs and executables in Docker containers.
- The Docker platform runs natively on Linux (on x86-64, ARM and many other CPU architectures) and on Windows (x86-64).
- Docker Inc. builds products that let you build and run containers on Linux, Windows and macOS.

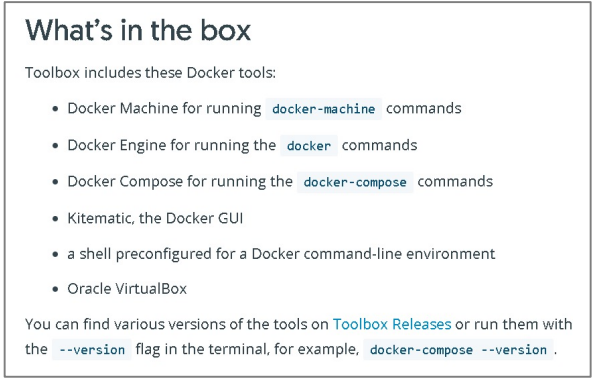


SUPPORTED PLATFORMS

Linux
- Native Application

Mac
- Native Application
- Docker Toolbox – Legacy

Windows
- Windows 10 (native)
- Windows Server 2016 (native)
- Other (Docker Toolbox)



What's in the box

Toolbox includes these Docker tools:

- Docker Machine for running `docker-machine` commands
- Docker Engine for running the `docker` commands
- Docker Compose for running the `docker-compose` commands
- Kitematic, the Docker GUI
- a shell preconfigured for a Docker command-line environment
- Oracle VirtualBox

You can find various versions of the tools on Toolbox Releases or run them with the `--version` flag in the terminal, for example, `docker-compose --version`.

DOCKER IN THE CLOUD

Google Cloud Platform

Amazon Web Services

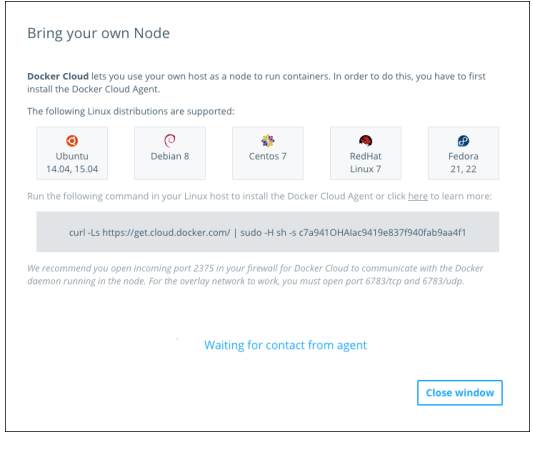
Microsoft Azure

IBM Cloud

Digital Ocean

Packet

Docker Cloud

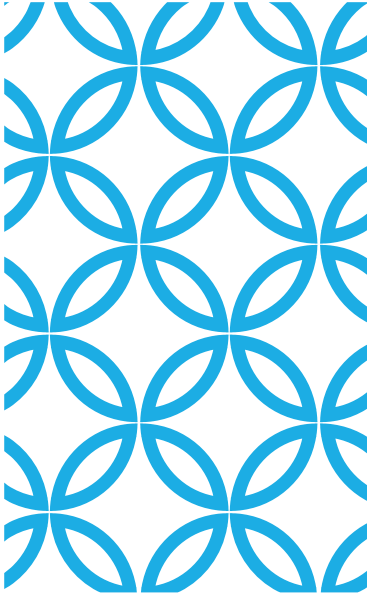


DOCKER EVERYWHERE







DOCKER COMPONENTS

DOCKER COMPONENTS (I)

Image

- An image is a portable template, accessible in read only mode, that has all the instructions necessary to create a Docker container
- Contains all the dependencies needed for a software component (code or binary, run-time, libraries, configuration files, environment variables, …)
- An image can be defined from a file system archive (tarball) Or it can be defined by extending a previous image with a list of instructions specified in a text file (known as *Dockerfile*)
- Images are made of layers, conceptually stacked on top of each other

Docker registry

- Database to efficiently store images
- Registries can be public (like DockerHub) or private to an organization

DOCKER COMPONENTS (II)

Container

- A Docker container is an executable instance of a Docker image
- Defined by the image and the configuration specified at creation time
- A container can be created, started, stopped, migrated, destroyed, connected to one or more virtual networks, associated to one or more data volumes ...
- The container is the unit of application development, testing, delivery and
- deployment, assuming that Docker is used as operating support
- Any modification to the file system visible to a container are not reflected on the image (image is read-only)
- It's possible to define to what extent a container is isolated from the host
- Access to the host file system and special devices, limitations on memory allocation and CPU utilization.

DOCKER COMPONENTS (III)

Network

- Virtual networks, implemented by means of *virtual switches* and iptables
- *Bridge* networks limit connectivity to the containers on a single host
- *Overlay* networks allow for containers connectivity among different hosts
  - *Typically using VXLAN encapsulation*

Volume

- A volume is a directory that can be associated to one or more containers
- Its lifespan is independent of the containers that use it
- Used to share storage across different containers, or anyway storage that can outlive the user container

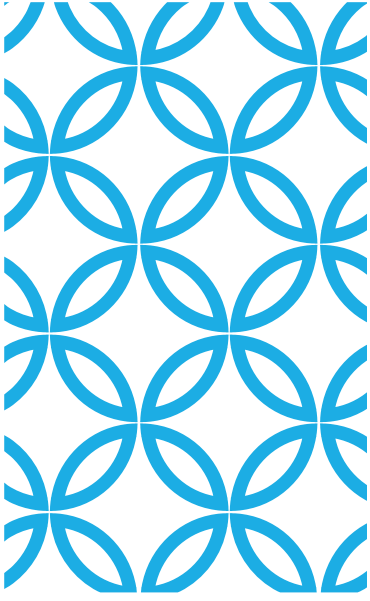
DOCKER COMPONENTS (IV)

Service

- A Docker *service* is a set of containers that are replicas of the same image, and which together provide a *load balanced* service
- *Services* are used to deploy containers "in production"
- A service can be scaled up or down depending on the input load

Stack

- A set of interdependent services that interact to implement a complete application:
  - Ex: A web application to share pictures could be made of
    1) a service for the storage and search of pictures;
    2) a service for the web interface for the users;
    3) a service to encode/decode pictures



DOCKER COMMANDS

THE DOCKER CLI

```
$ docker
Usage: docker [OPTIONS] COMMAND [ARG...]
       docker [ --help | -v | --version ]

A self-sufficient runtime for containers.

Options:
      --config string      Location of client config files (default "/root/.docker")
  -D, --debug              Enable debug mode
      --help               Print usage
  -H, --host value         Daemon socket(s) to connect to (default [])
  -l, --log-level string   Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (de
      --tls                Use TLS; implied by --tlsverify
      --tlscacert string   Trust certs signed only by this CA (default "/root/.docker/ca.pem
      --tlscert string     Path to TLS certificate file (default "/root/.docker/cert.pem")
      --tlskey string      Path to TLS key file (default "/root/.docker/key.pem")
      --tlsverify          Use TLS and verify the remote
  -v, --version            Print version information and quit
```

Docker commands reference :
https://docs.docker.com/engine/reference/commandline/cli/

$ DOCKER CREATE

```
$ docker create [ARGUMENTS] <image>
```

Create a new container based in an specific image

If the image is not found **locally**, it's pulled from the default **registry**

Each container have it's own Id

$ DOCKER RUN

```
$ docker run [ARGUMENTS] <image>
```

Create a new container based in an specific image and run it

If the image is not found locally, it’s pulled from the default registry

The container exits once the command running inside of it exits

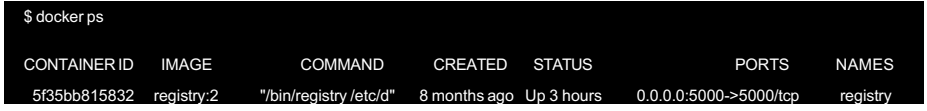
-it : interactive mode (default)

-d : detached mode

$ DOCKER PS

```
$ docker ps
```

Will list the running container in your host



“docker ps -a” command show all containers that have run in the past, but are not necessarily running now.

$ DOCKER IMAGES

```
$ docker images
```

Show all top level images, their repository and tags, and their size.

Intermediate layers are not shown by default but can be showed by adding ‘-a’ flag.

$ DOCKER RM

```
$ docker rm <container-id>
```

Delete a container

The container must be stopped in order to be removed

The flag “-f” can be used to remove running containers

You can remove all the containers at once using the command:

```
$ docker rm $(docker ps -a -q)
```

$ DOCKER RMI

```
$ docker rmi <image-id>
```

Delete the specified image

You can delete multiple images in the same commands passing them as arguments, for example:

$ docker rmi 18wj2 as83j a92k4

You can remove all the images at once using the command:

$ docker rmi $(docker images -q)

$ DOCKER ATTACH

```
$ docker attach <container-id>
```

Attaches to PID1 inside the container

To detach from the container use “Ctrl + P + Q”

Using “Ctrl + C” will stop the process in the container (and therefore stop the container itself)

$ DOCKER EXEC

```
$ docker exec <container-id> <tool>
```

Run a command (new process) in a running container

Useful when the PID1 is not a shell

You can use the flag -it to run the command interactively

$ DOCKER COMMIT

```
$ docker commit <container-id> <new-image-name>
```

Save container status creating a new image

By default, the container being committed and its processes will be paused while the image is committed

Use the flag “-p=false” to avoid this behavior

$ DOCKER SAVE

```
$ docker save –o <path/to/file.tar> <image-id>
```

- Save a container image in a file
- Useful to share containers without a container registry

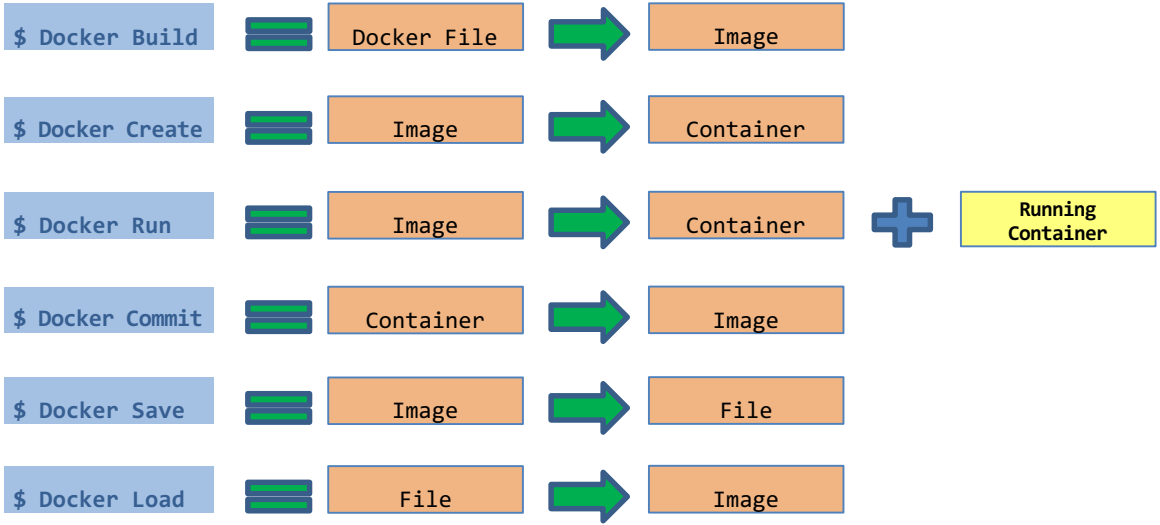
$ DOCKER LOAD

```
$ docker load -i <path/to/file.tar>
```

Load a container image from a file

Useful to share containers without a container registry

COMMANDS TRANSFORMATION



FIRST CONTAINER

- We used one of the smallest, simplest images available: busybox.
- busybox? The Swiss Army Knife of Embedded Linux. Coming in somewhere between 1 and 5 Mb in on-disk size (depending on the variant), BusyBox is a very good ingredient to craft space-efficient distributions
- BusyBox combines tiny versions of many common UNIX utilities into a single small executable
- We ran a single process and echo'ed hello world.

```
$ docker run busybox echo hello world
hello world
```

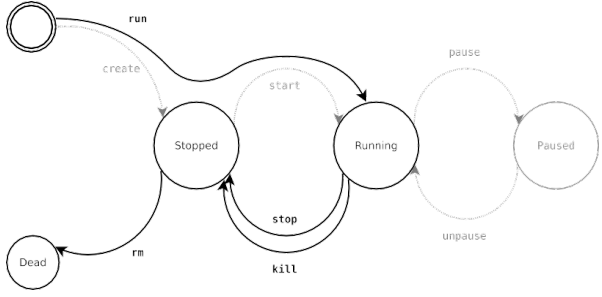
STOP CONTAINER

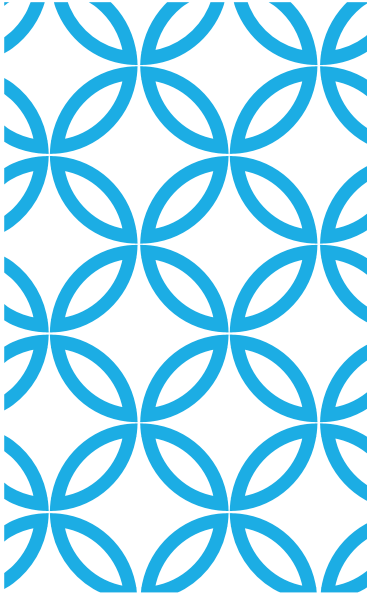
- There are two ways we can terminate our detached container.
  - Killing it using the docker kill command.
  - Stopping it using the docker stop command.
- The first one stops the container immediately, by using the KILL signal.
- The second one is more graceful. It sends a TERM signal, and after 10 seconds, if the container has not stopped, it sends KILL.
- Reminder: the KILL signal cannot be intercepted, and will forcibly terminate the container

LIFECYCLE OF DOCKER CONTAINER

There are different stages when we create a Docker container which is known as Docker Container Lifecycle. Some of the states are:

- **Created:** A container that has been created but not started
- **Running:** A container running with all its processes
- **Paused:** A container whose processes have been paused
- **Stopped:** A container whose processes have been stopped
- **Deleted:** A container in a dead state





DOCKER IMAGES

DOCKER IMAGES

• An image is a collection of files + some meta data.

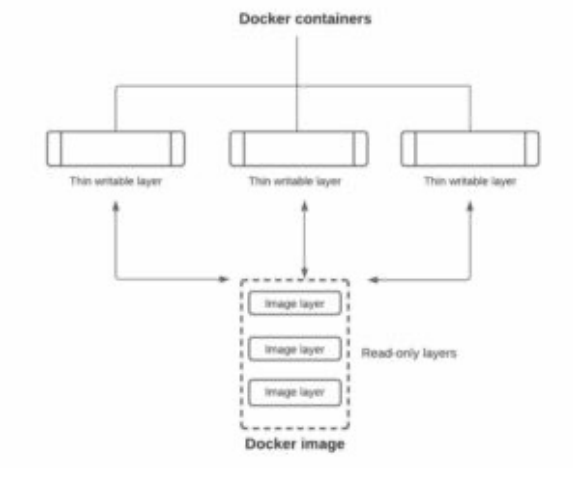
(Technically: those files form the root filesystem of a container.)

• Images are made of layers, conceptually stacked on top of each other.

• Each layer can add, change, and remove files.

• Images can share layers to optimize disk usage, transfer times, and memory use.

IMAGE VS CONTAINER



Docker Image	Docker Container
It's a container blueprint	It's an image instance
It's Immutable	Its Writable
It can exist without a container	A container must run an image to exist
Doesn't need computing resources to operate	Need computing resources to run containers as docker in Vms
It can be shared via a public or private	No need to share an already running entity
Created only once	Multiple containers can be created from same image

IMAGE CREATION

There are multiple ways to create new images.

- docker commit: creates a new layer (and a new image) from a container.
- docker build: performs a repeatable build sequence.
- docker import: loads a tarball into Docker, as a standalone base layer.

Import can be used for various hacks, but its main purpose is to bootstrap the creation of base images.

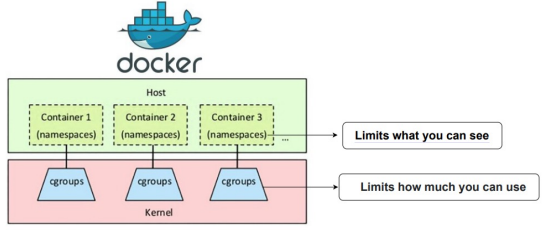
IMAGES NAMESPACES

There are three namespaces:

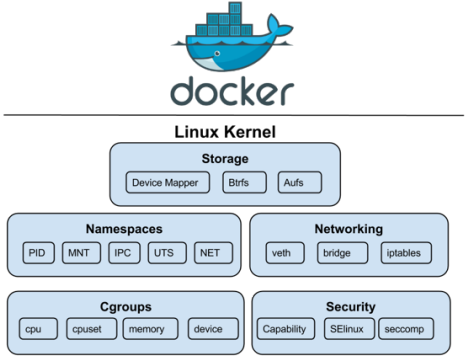
• Root-like - The root namespace is for official images. They are put there by Docker Inc., but they are generally authored and maintained by third parties

• User (and organizations) - The user namespace holds images for Docker Hub users and organizations.
(ex) - jpetazzo/clock

• Self-Hosted - This namespace holds images which are not hosted on Docker Hub, but on third party registries. They contain the hostname (or IP address), and optionally the port, of the registry
server.registry.example.com:5000/my-private-image
(ex) localhost:5000/wordpress



KERNEL NAMESPACES



The type of resources associated with that process is dependent on the kind of namespace provided for it. Let's look at the features of a few of them:

- Mount (MNT): Controls mount points. When new namespaces are created the current mounts are copied to a new namespace.
- Process ID (PID): Provides processes with process IDs from other namespaces.
- Interprocess Communication (IPC): Prevents processes in different IPC namespaces from forming SHM functions.
- Network (NET): Virtualizes network stack.
- UNIX Time Sharing (UTS): This allows a system to have different host and domain names for various processes.

SEARCH AND DOWNLOAD IMAGES

Searching for images

Searches your registry for images:

$ docker search zookeeper

- "Stars" indicate the popularity of the image.
- "Official" images are those in the root namespace.
- "Automated" images are built automatically by the Docker Hub.
- (This means that their build recipe is always available.)

- There are two ways to download images.
  - Explicitly, with docker pull.
  - Implicitly, when executing docker run and the image is not found locally.

```
Alexander@DESKTOP-90ATKET MINGW64 ~/Docker/Demo
$ docker pull nginx:latest
latest: Pulling from library/nginx
bc95e04b23c0: Pull complete
f3186e650f4e: Pull complete
9ac7d6621708: Pull complete
Digest: sha256:b81f317384d7388708a498555c28a7cce778a8f291d90021208b3eba3fe74887
Status: Downloaded newer image for nginx:latest
```

IMAGE AND TAGS

Images can have tags.

Tags define image variants.

- docker pull ubuntu **will refer to** ubuntu:latest.

The :latest tag can be updated frequently.

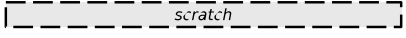
When using images, it is always best to be specific.

DOCKER IMAGE PULL: PULLS LAYERS

```
Alexander@DESKTOP-90ATKET MINGW64 ~/Docker/Demo
$ docker pull nginx:latest
latest: Pulling from library/nginx
bc95e04b23c0: Pull complete
f3186e650f4e: Pull complete
9ac7d6621708: Pull complete
Digest: sha256:b81f317384d7388708a498555c28a7cce778a8f291d90021208b3eba3fe74887
Status: Downloaded newer image for nginx:latest
```

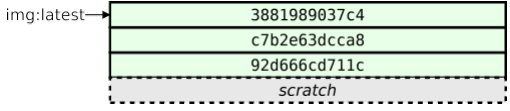
EXAMPLE: IMAGES & CONTAINERS

EXAMPLE: IMAGES & CONTAINERS



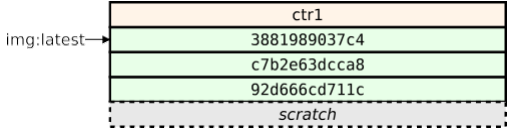
EXAMPLE: IMAGES & CONTAINERS

```
docker pull img
```



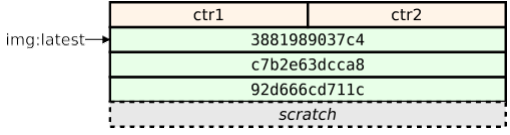
EXAMPLE: IMAGES & CONTAINERS

```
docker run --name ctr1 img
```



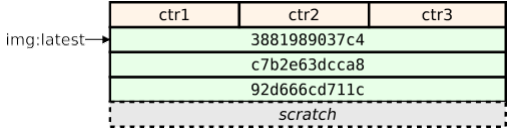
EXAMPLE: IMAGES & CONTAINERS

```
docker run --name ctr2 img
```



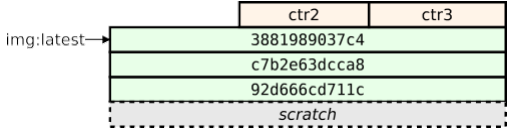
EXAMPLE: IMAGES & CONTAINERS

```
docker run --name ctr3 img
```



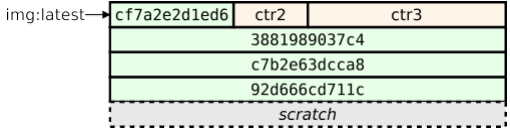
EXAMPLE: IMAGES & CONTAINERS

```
docker rm ctr1
```



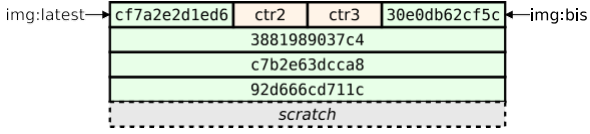
EXAMPLE: IMAGES & CONTAINERS

```
docker commit ctr2 img
```



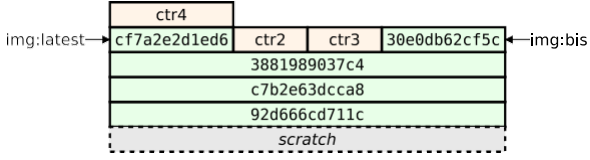
EXAMPLE: IMAGES & CONTAINERS

```
docker commit ctr3 img:bis
```



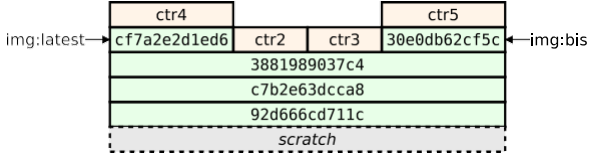
EXAMPLE: IMAGES & CONTAINERS

```
docker run --name ctr4 img
```



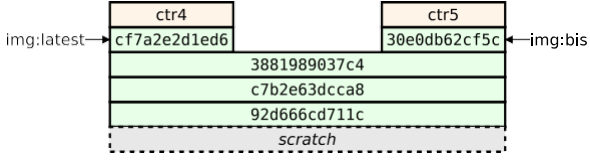
EXAMPLE: IMAGES & CONTAINERS

```
docker run --name ctr5 img:bis
```



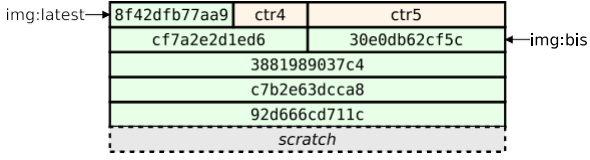
EXAMPLE: IMAGES & CONTAINERS

```
docker rm ctr2 ctr3
```



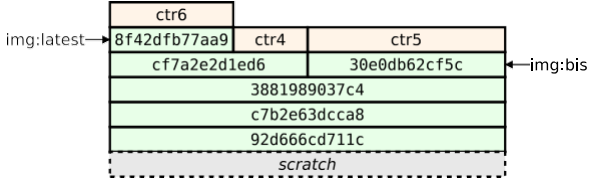
EXAMPLE: IMAGES & CONTAINERS

```
docker commit ctr4 img
```



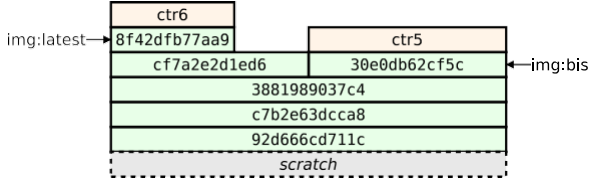
EXAMPLE: IMAGES & CONTAINERS

```
docker run --name ctr6 img
```



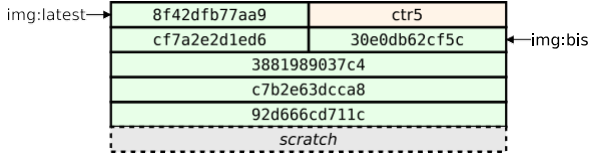
EXAMPLE: IMAGES & CONTAINERS

```
docker rm ctr4
```



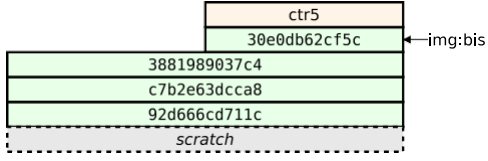
EXAMPLE: IMAGES & CONTAINERS

```
docker rm ctr6
```



EXAMPLE: IMAGES & CONTAINERS

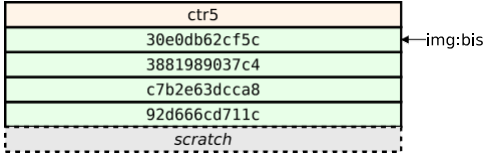
```
docker rmi img
```



EXAMPLE: IMAGES & CONTAINERS

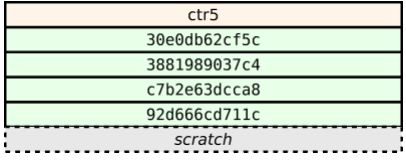
```
docker rmi img:bis
```

Error: image img:bis is reference by ctr5



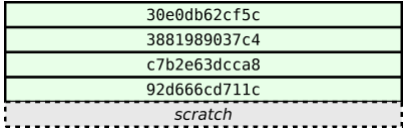
EXAMPLE: IMAGES & CONTAINERS

```
docker rmi -f img:bis
```



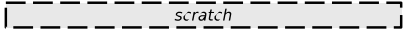
EXAMPLE: IMAGES & CONTAINERS

```
docker rm ctr5
```



EXAMPLE: IMAGES & CONTAINERS

```
docker rmi 30e0
```



STRUCTURE OF A DOCKER IMAGE

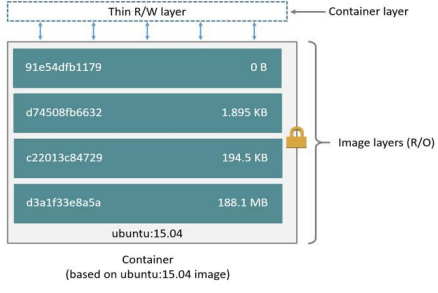
An image is a stack of layers (onion-like structure)

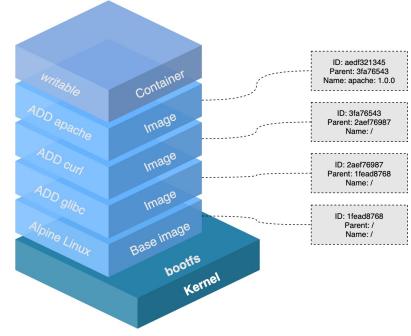
Each instruction in the Dockerfile adds a layer

- Each layer stores only the difference w.r.t. the previous layers.

A read/write container layer gets created on containers creation

```
# This is a Dockerfile
FROM ubuntu:15.10
COPY . /app
RUN make /app
CMD python /app/app.py
```

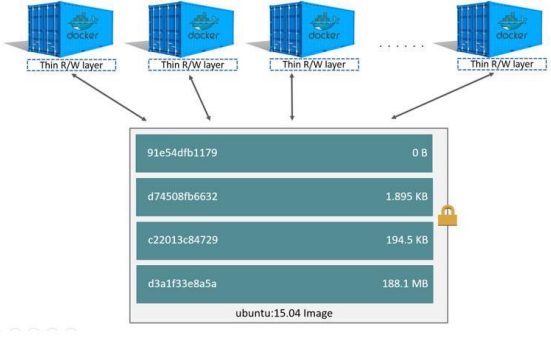




SHARING LAYERS

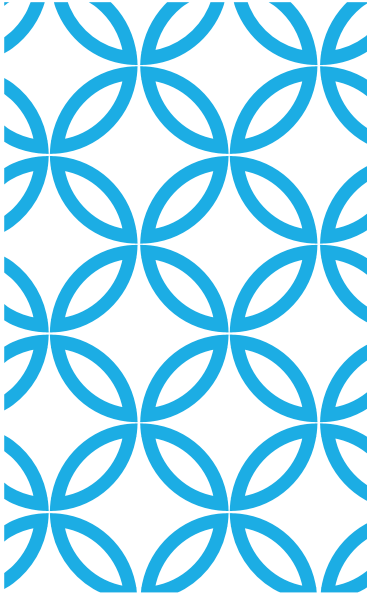
When a container is running, any modification to its disk are reflected to the container layer.

All the other layers are read-only and can be shared among many container



BASIC DOCKER COMMANDS

```
$ docker image pull node:latest
$ docker image ls
$ docker container run –d –p 5000:5000 –-name node node:latest
$ docker container ps
$ docker container stop node(or <container id>)
$ docker container rm node (or <container id>)
$ docker image rmi (or <image id>)
$ docker build –t node:2.0 .
$ docker image push node:2.0
$ docker --help
```



MANAGING DOCKER CONTAINERS

$ DOCKER STOP

```
$ docker stop <container-id>
```

Terminate the container PID1 process and make sure that all other processes on the system are cleaned up as gracefully as possible

$ DOCKER KILL

```
$ docker kill <container-id>
```

‘Kill’ the container PID1 process

Brutal stoppage

$ DOCKER START

```
$ docker start <container-id>
```

Start a container from an image

$ DOCKER RESTART

```
$ docker restart <container-id>
```

Restart a running container

$ DOCKER PAUSE\UNPAUSE

```
$ docker <pause\unpause> <container-id>
```

Pause - Pauses a running container, "freezing" it in place.

Unpause - Unpause a “frozen container.

$ DOCKER INFO

```
$ docker info
```

Display system-wide information

The number of images shown is the number of unique images. The same image tagged under different names is counted only once

It is Docker's – 'Systeminfo' command in Windows & 'lshw' in Linux

$ DOCKER STATS

```
$ docker stats <container-id>
```

Display a live stream of container(s) resource usage statistics

```
$ docker stats

CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
3540224d937f        ubuntu              0.00%               1008KiB / 1.934GiB    0.05%               1.04kB / 0B         0B / 0B             1
```

$ DOCKER TOP

```
$ docker top <container-id>
```

Display the running processes of a container

Note it shows the view of processes and PIDs from outside the container (from our docker host, within the wider namespace of the docker host process tree)

If we want to see the PIDs inside the container we can jump into the container and check (docker attach + ps -ef)

$ DOCKER CP

```
$ docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH
```

```
$ docker cp [OPTIONS] SRC_PATH CONTAINER:DEST_PATH
```

Copy files/folders between a container and the local filesystem

$ DOCKER HISTORY

```
$ docker history <image-id>
```

Show the history of an image

When using the --format option, the history command will either output the data exactly as the template declares or, when using the table directive, will include column headers as well.

$ DOCKER INSPECT

```
$ docker inspect <container-id>
```

Return low-level information on Docker objects

By default, docker inspect will render results in a JSON array.

$ DOCKER LOGS

```
$ docker logs <container-id>
```

Fetch the logs of a container (output)

The docker logs --follow command will continue streaming the new output from the container's STDOUT and STDERR

The docker logs --details command will add on extra attributes, such as environment variables and labels, provided to --log-opt when creating the container.

$ DOCKER DIFF

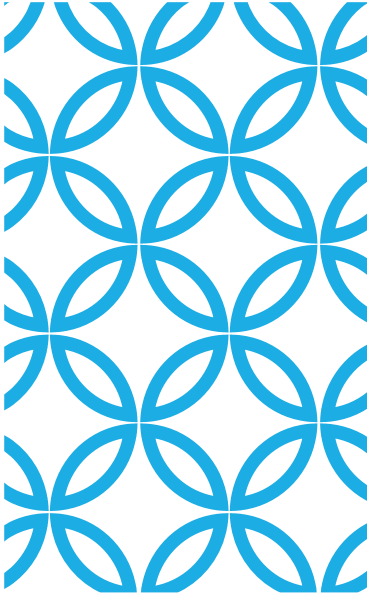
```
$ docker diff <container-id>
```

Inspect changes to files or directories on a container’s filesystem

Symbol	Description
A	A file or directory was added
D	A file or directory was deleted
C	A file or directory was changed

CONTAINER MANAGEMENT COMMANDS

command	description
docker create image [command] docker run image [command]	create the container = create + start
docker rename container new name docker update container	rename the container update the container config
docker start container... docker stop container... docker kill container... docker restart container... .	start the container graceful[2] stop kill (SIGKILL) the container = stop + start
docker pause container... docker unpause container...	suspend the container resume the container
docker rm [-f[3]] container...	destroy the container



DOCKERFILE

DOCKERFILE

```
FROM ubuntu:14.04
RUN \
  apt-get update && \
  apt-get -y install apache2

VOLUME /myvol
ADD index.html /var/www/html/index.html

EXPOSE 80

CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

- Dockerfile (with capital D)
- It's comprised of instructions that define how to build an image
- These instructions get read one at time, from top to bottom
- When we build images from Dockerfiles, any other files and directories in the same directory as the Dockerfile were going to get included in the build (build context).
- A text file that contains a recipe to build an image
- An image should be a well-defined component and contain only the software actually needed for a well-defined task

DOCKERFILE

```
# Start from an official image with the Python
runtime
FROM python:2.7-slim
# Set the container current working directory (PWD) to “/chess”
WORKDIR /chess
# Copy files from current host directory to the /chess directory in the container
ADD . /chess
# Install some packages
RUN apt-get update && apt-get install -y libcgroup acl
# Flag that the software inside this image listens on port 9000
EXPOSE 9000
# Define an
environment
variable  ENV
PLAYER Ghost
# Specify the command to be run inside the container when it’s started
CMD ["python", "./chess.py"]
```

Dockerfile

```
# Create image based on the official Node 6 image from dockerhub
FROM node:latest

# Create a directory where our app will be placed
RUN mkdir -p /usr/src/app

# Change directory so that our commands run inside this new directory
WORKDIR /usr/src/app

# Copy dependency definitions
COPY package.json /usr/src/app

# Install dependecies
RUN npm install

# Get all the code needed to run the app
COPY . /usr/src/app

# Expose the port the app runs in
EXPOSE 4200

# Serve the app
CMD ["npm", "start"]
```

DOCKERFILE COMMANDS

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

$ DOCKER BUILD

```
$ docker build –t name:tag .
```

```
$ docker build github.com/creack/docker-firefox
```

Create a new Docker image following the Dockerfile instructions

You can specify a Dockerfile in the filesystem or build an image from a Dockerfile stored in GitHub

Tags are used to manage image versions

DOCKERFILE – INSTRUCTIONS

FROM

```
FROM ubuntu:15.04
```

Sets the Base Image for subsequent instructions

The image can be any valid image (usually a container start by pulling an image from the Docker Hub)

DOCKERFILE – INSTRUCTIONS

LABEL

Add metadata to the docker image

Used to indicate things like the image maintainer, version, description, etc

```
FROM  ubuntu:15.04
LABEL maintainer="ck@bootlas.com"
LABEL  version="1.0"
LABEL description="my image description"
```

DOCKERFILE – INSTRUCTIONS

RUN

Used to run commands against our images that we're building

Every run instruction adds a layer to our image

Run commands are the image "build steps"

```
FROM  ubuntu:15.04
LABEL maintainer="ck@bootlas.com"
RUN apt-get update
RUN  apt-get install -y apache2
RUN apt-get install -y vim
RUN apt-get install -y apache2-utils
```

DOCKERFILE – INSTRUCTIONS

CMD

Is the command executed anytime we launch a container from this image

This command can be overridden in the run command

Two types of sintax:

Shell: echo "Hello World" Exec:

["echo", "Hello World"]

```
FROM ubuntu:15.04
LABEL maintainer="ck@bootlas.com"
RUN apt-get update
RUN apt-get install -y apache2
RUN apt-get install -y vim
RUN apt-get install -y apache2-utils
CMD ["echo","Hello World!"]
```

BUILD CACHE

When we build a new image the docker deamon iterates through our Dockerfile executing each instruction.

As each instruction gets executed, the deamon checks to see whether or not it's got an image for that instruction already in its build cache

The build cache store each instruction + linked image

(Change the docker file invalidates the build cache)

```
vagrant@ubuntudocker:~/docker-sample$ sudo docker build -t ubuntu-apache .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
 ---> d0955f21bf24
Step 1 : RUN apt-get update && apt-get install -y apache2 curl
 ---> Using cache
 ---> ffc5220d73a0
Step 2 : RUN echo "Hello Docker World" > /var/www/html/index.html
 ---> Using cache
 ---> 413aed5cf9a1
Step 3 : EXPOSE 80
 ---> Using cache
 ---> 064caef795a1
Step 4 : RUN echo /usr/sbin/apachectl start >> /etc/bash.bashrc
 ---> Running in c68d11d3d88c
 ---> fdc15a2f286a
Removing intermediate container c68d11d3d88c
Successfully built fdc15a2f286a
vagrant@ubuntudocker:~/docker-sample$
```

DOCKERFILE – INSTRUCTIONS

EXPOSE

The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime.

You can expose one port number and publish it externally under another number.

```
FROM  ubuntu:15.04
LABEL  maintainer="ck@bootlabs.com"
RUN apt-get update &&  apt-get install -y  \
            apache2 \
            vim \
            apache2-utils
EXPOSE  80
CMD  ["apache2ctl","-D","FOREGROUND"]
```

DOCKERFILE – INSTRUCTIONS

ENTRYPOINT

Is the better method of specifying the default app to run inside of a container

Anything we do specify at the end of the docker run command at runtime (or CMD instruction) get interpreted as arguments to the entrypoint instruction

```
FROM ubuntu:15.04
LABEL maintainer="ck@bootlabs.com"
RUN apt-get update && apt-get install -y \
        apache2 \
        vim \
        apache2-utils
EXPOSE 80
ENTRYPOINT ["echo"]
```

DOCKERFILE – INSTRUCTIONS

ENV

Used to assign environment variables

Environment variables can be overridden in the docker run command using **-e "var=value"**

```
FROM  ubuntu:15.04
LABEL  maintainer="ck@bootlas.com"
RUN apt-get update &&  apt-get install -y  \
            apache2 \
            vim \
            apache2-utils
ENV  var1=val1
EXPOSE  80
ENTRYPOINT  echo $var1
```

DOCKERFILE – INSTRUCTIONS

ARG

The ARG instruction defines variables used at build-time

Arguments can be passed in the docker build command using the flag - **--build-arg <varname>=<value>**

```
FROM  ubuntu:15.04
LABEL  maintainer="ck@bootlas.com"
RUN apt-get update &&  apt-get install -y  \
            apache2 \
            vim \
            apache2-utils
ENV var1=val1
ARG var2=val2
RUN  echo $var2
EXPOSE  80
ENTRYPOINT  echo $var1
```

DOCKERFILE – INSTRUCTIONS

COPY

Copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>

Each <src> may contain wildcards

```
FROM ubuntu:15.04
LABEL maintainer="ck@bootlas.com"
RUN apt-get update && apt-get install -y \
        apache2 \
        vim \
        apache2-utils
COPY hom* /mydir
ENV var1=val1
ARG var2=val2
RUN echo $var2
EXPOSE 80
ENTRYPOINT ["echo"]
```

DOCKERFILE – INSTRUCTIONS

ADD

Similar than Copy but:

- ADD allows <src> to be an URL
- If the <src> parameter of ADD is an archive in a recognized compression format or a URL, it will be unpacked

```
FROM  ubuntu:15.04
LABEL  maintainer="ck@bootlas.com"
RUN apt-get update &&  apt-get install -y  \
            apache2 \
            vim \
            apache2-utils
ADD  test.tar.gz  /mydir
ENV var1=val1
ARG var2=val2
RUN  echo $var2
EXPOSE  80
ENTRYPOINT  ["echo"]
```

DOCKERFILE – INSTRUCTIONS

WORKDIR

Used to set the working directory inside the container

```
FROM  ubuntu:15.04
LABEL  maintainer="ck@bootlas.com"
RUN apt-get update &&  apt-get install -y  \
            apache2 \
            vim \
            apache2-utils
ADD  test.tar.gz /mydir
ENV var1=val1 var2=val2
ARG var2=val2
RUN  echo $var2
EXPOSE  80
WORKDIR /mydir
ENTRYPOINT  ["echo"]
```

DOCKERFILE COMMON MISTAKES

Using "latest" tag in base images

Using external services during the build

Adding EXPOSE and VAR's at the top of your Dockerfile

Add the app directory at the beginning of the Dockerfile

Wrong multiple FROM statements

Multiple services running in the same container

DOCKERFILE BEST PRACTICES

Use a .dockerignore file

Containers should be immutable & ephemeral (no data inside)

Minimize the number of layers / Consolidate instructions

Avoid installing unnecessary packages

Sort multi-line arguments

Use Build cache

Understand CMD and ENTRYPOINT

DOCKERIGNORE FILE

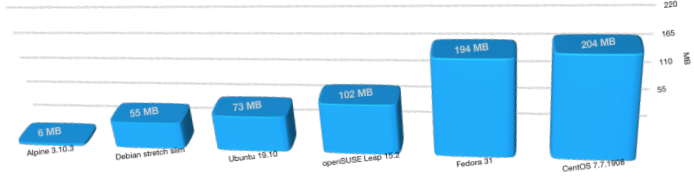
Use a .dockerignore file

The directory where you issue the docker build command is called the build context. Docker will send all of the files and directories in your build directory to the Docker daemon as part of the build context. If we have stuff in your directory that is not needed by our build, we'll have an unnecessarily larger build context that results in a larger image size.

We can remedy this situation by adding a .dockerignore file that works similarly to .gitignore. You can specify the list of folders and files that should be ignored in the build context.

OPTIMAL BASE IMAGE

- A base image forms the very first layer of the final Docker image. It is extremely important to choose an optimal base image.
- The image can be any valid image — it is especially easy to start by pulling an image from the Public Repositories."
- Obviously, there is a ton of different base images to choose from, each one with its own perks and features. Choosing an image which provides just enough of the tools and the environment we need for our application to run is of vital importance when it comes to the final size of your own Docker image.



THE ALPINE BASE IMAGE

Alpine is a vert tiny linux image which "weights" only 3.6 MB !

DISTRIBUTION	VERSION	SIZE
Debian	Jessie	123MB
CentOS	7	193MB
Fedora	25	231MB
Ubuntu	16.04	118MB
Alpine	3.6	3.98MB

Suppose your organization has millions pulls from Docker-hub each year, this pull size difference can save to your company approx. 400K$ a year!

THE ALPINE BASE IMAGE

Debian

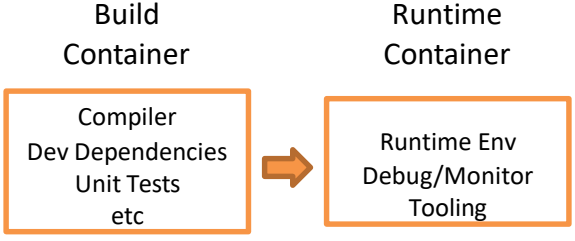
```
time docker run --rm debian sh -c "apt-get update && apt-get install curl"

real    0m27.928s
user    0m0.019s
sys     0m0.077s
```

Alpine

```
time docker run --rm alpine sh -c "apk update && apk add curl"

real    0m5.698s
user    0m0.008s
sys     0m0.037s
```

DOCKER BUILDER PATTERN

Pattern used to create small images



Benefits:

- Size
- Performance
- Security

STRATEGIES TO BUILD OPTIMAL DOCKER IMAGES

- A Docker image describes the instructions for running a container. Each of these instructions, contained in the Dockerfile, create a layer. The set of these layers is useful to speed up the build operations and the image transfer. This saves a lot of disk space when many images share the same layers. The 1st Dockerfile creates an image with 10 layers. However, if we structure these statements carefully, we can drastically reduce the image size.
- For instance, the same Dockerfile above can be re-written as shown in 2nd Dockerfile

```
FROM debian:stable

WORKDIR /var/www

RUN apt-get update

RUN apt-get -y --no-install-recommends install curl

RUN apt-get -y --no-install-recommends install ca-certificates

RUN curl https://raw.githubusercontent.com/gadiener/docker-images-size-
benchmark/master/main.go -o main.go

RUN apt-get purge -y curl

RUN apt-get purge -y ca-certificates

RUN apt-get autoremove -y

RUN apt-get clean

RUN rm -rf /var/lib/apt/lists/*
```

```
FROM debian:stable

WORKDIR /var/www

RUN apt-get update &amp;amp;&amp;amp; \
    apt-get -y --no-install-recommends install curl \
    ca-certificates && \
    curl https://raw.githubusercontent.com/gadiener/docker-images-size-
benchmark/master/main.go -o main.go &&  \
    apt-get purge -y curl \
    ca-certificates && \
    apt-get autoremove -y && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

DOCKER BUILDER PATTERN

700 MB
392 Vulnerabilities

```
FROM go:onbuild
WORKDIR /app
ADD . /app
RUN cd /app && go build -o goapp
EXPOSE 8080
ENTRYPOINT ./goapp
```

VS

12 MB
3 Vulnerabilities

```
FROM golang:alpine AS build-env
WORKDIR /app
ADD . /app
RUN cd /app && go build -o goapp

FROM alpine
RUN apk update && apk add ca-
certificates && rm -rf /var/cache/apk/*
WORKDIR /app
COPY --from=build-env /app/goapp /app

EXPOSE 8080
ENTRYPOINT ./goapp
```

WHAT IS A BETTER APPROACH?

(A)

```
FROM centos:7

RUN yum update -y
RUN yum install -y sudo
RUN yum install -y git
RUN yum clean all
```

470 MB

(B)

```
FROM centos:7

RUN yum update -y \
 && yum install -y \
 sudo \
 git \
 && yum clean all
```

265 MB

B) In *"A"* many layers were added for nothing. Also the *"yum clean all"* command is meant to reduce the size of the image but it actually does the opposite by adding a new layer

WHAT IS A BETTER APPROACH?

(A)

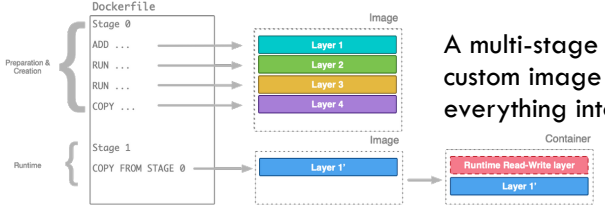
```
FROM python:3.6
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["ap.py"]
```

(B)

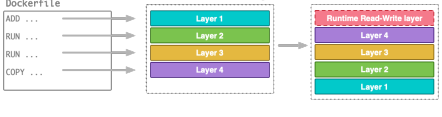
```
FROM python:3.6
WORKDIR /app
COPY requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt
COPY . /app
ENTRYPOINT ["python"]
CMD ["ap.py"]
```

B) The COPY . /app command will invalidate the cache as soon as any file in the current directory is updated.

MULTI-STAGE BUILDS



A multi-stage build allows image builders to leave custom image build scripts behind and integrate everything into the well-known Dockerfile format.



In high-level terms, we can think of a multi-stage build as merging multiple Dockerfiles together, or simply a Dockerfile with multiple FROMs.

INSPECTING THE IMAGE

Once we create a Docker image, we should always inspect the Docker image to check each layer and check how optimized the image is. Using the Docker history command is one such way.

```
$ docker history cfssl/cfssl
IMAGE          CREATED        CREATED BY                                      SIZE      COMMENT
fb08fbb7bb6d   15 months ago  /bin/sh -c #(nop)  CMD ["--help"]                0B
<missing>      15 months ago  /bin/sh -c #(nop)  ENTRYPOINT ["cfssl"]         0B
<missing>      15 months ago  /bin/sh -c #(nop)  EXPOSE 8888                  0B
<missing>      15 months ago  /bin/sh -c go get github.com/cloudflare/cfss...  280MB
<missing>      15 months ago  /bin/sh -c #(nop) COPY dir:a17f6a12df322554f...  20.6MB
<missing>      15 months ago  /bin/sh -c #(nop) WORKDIR /go/src/github.com...  0B
<missing>      15 months ago  /bin/sh -c #(nop)  ENV USER=root                 0B
<missing>      2 years ago    /bin/sh -c #(nop) COPY file:ea7c9f4702f94a0d...  2.48kB
<missing>      2 years ago    /bin/sh -c #(nop) WORKDIR /go                   0B
<missing>      2 years ago    /bin/sh -c mkdir -p "$GOPATH/src" "$GOPATH/b...  0B
```

DOCKER ATTACHED VS DETACHED MODE

By default, Docker runs the container in attached mode. In the attached mode, Docker can start the process in the container and attach the console to the process's standard input, standard output, and standard error.

- *docker run --name springbootappcontainer -p 8080:8080 springbootapp*

Detached mode, started by the option --detach or –d flag in docker run command, means that a Docker container runs in the background of your terminal. It does not receive input or display output. Using detached mode also allows you to close the opened terminal session without stopping the container.

- *docker run **-d** --name springbootappcontainer -p 8080:8080 springbootapp*

To reattach to a detached container, use docker attach command.

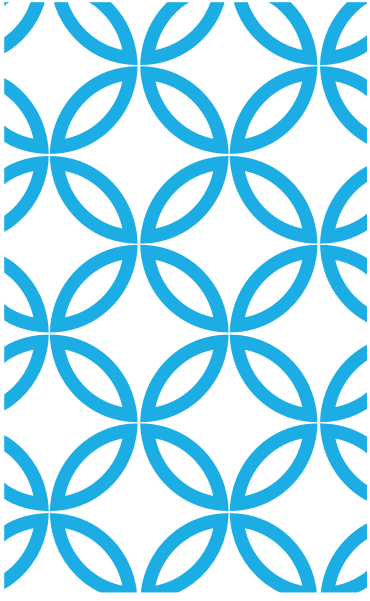
- *docker attach <nameofcontainer>*

OR

- *docker attach <dockerid>*

DOCKER IMAGES

Demo



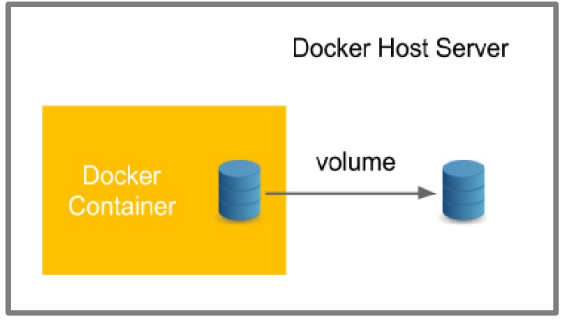
DOCKER VOLUMES

VOLUMES

When a container dies all the data it has created (logs, database records, etc) dies with it (and remember containers are ephemeral).

Volumes are external storage areas used to store data produced by a Docker container.

Volumes can be located on the Docker host or even on remote machines



WHY USE VOLUMES

Mount local source code into a running container

docker container run -v $(pwd):/usr/src/app/ myapp

Improve performance

—As directory structures get complicated traversing the tree can slow system performance

Data persistence

DOCKER VOLUMES

By default volumes are not deleted when the container is stopped.

Data volumes can be shared across containers.

Volumes could be mounted in read-only mode.

--volume: Create a file or directory if it doesn't exist on the Docker host

--mount: Does not automatically create it for you, but generates an error

VOLUMES ARE SPECIAL DIRECTORIES

Volumes can be declared in two different ways.

- Within a Dockerfile, with a VOLUME instruction.
  - VOLUME /var/lib/postgresql
- On the command-line, with the -v flag for docker run.
  - $ docker run -d -v /var/lib/postgresql training/postgresql

In both cases, /var/lib/postgresql (inside the container) will be a volume.

DOCKER VOLUMES - SYNTAX

```
$ docker run [OPTIONS] -v "volume_name:/container/path" [IMAGE]
```

```
$ docker run [OPTIONS] -v "/host/path:/container/path" [IMAGE]
```

```
$ docker run [OPT] --mount "type=bind,source=host/path,target=contain/path" [IMAGE]
```

DOCKER VOLUMES – DOCKERFILE

VOLUME

Create a new volume with any data that exists at the specified location within the base image.

Anything after the VOLUME instruction will not be able to make changes to that volume.

```
FROM microsoft/iis
RUN powershell -NoProfile -Command
Remove-Item -Recurse
C:\inetpub\wwwroot\*
WORKDIR /inetpub/wwwroot
COPY . .
VOLUME c:/inetpub/wwwroot
EXPOSE 80
```

DOCKER VOLUMES – MANAGING VOLUMES

Create a volume:

```
$ docker volume create volume-name
```

```
$ docker volume create demo-volume
demo-volume
```

List volumes:

```
$ docker volume ls
```

```
$docker volume ls
DRIVER          VOLUME NAME
local           demo-volume
local           ed702f0a2c8b6ceb56...
local           my_volume
```

DOCKER VOLUMES – MANAGING VOLUMES

Inspect a volume:

```
$ docker volume inspect volume-name
```

```
$ docker volume inspect demo-volume
[
  {
      "CreatedAt": "2018-06-07T23:39:20+03:00",
      "Driver": "local",
      "Labels": {},
      "Mountpoint":
"/var/lib/docker/volumes/demo-volume/_data",
      "Name": "demo-volume",
      "Options": {},
      "Scope": "local"
  }
]
```

Remove a volume:

```
$ docker volume rm volume-name
```

```
$ docker volume rm demo-volume
demo-volume
```

DOCKER VOLUMES – MANAGING VOLUMES

Remove all unused local volumes:

```
$ docker volume prune
```

```
$ docker volume prune
demo-volume-1
Demo-volume-2
```

DOCKER VOLUMES

Volumes mount a directory on the host into the container at a specific location

- Can be used to share (and persist) data between containers
  - Directory persists after the container is deleted
  - Unless you explicitly delete it
- Can be created in a Dockerfile or via CLI

SHARED VOLUMES

We can start a container with exactly the same volumes as another one.

The new container will have the same volumes, in the same directories.

They will contain exactly the same thing, and remain in sync.

Under the hood, they are actually the same directories on the host anyway.

This is done using the --volumes-from flag for docker run.

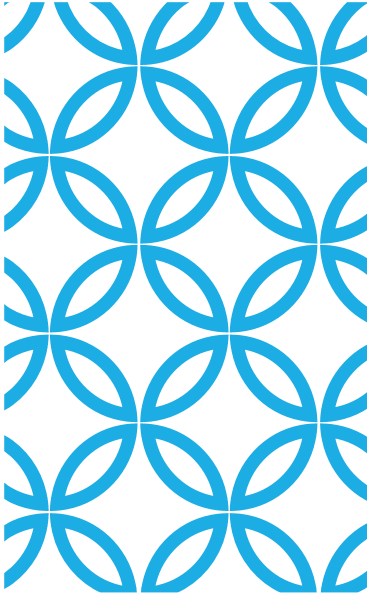
- $ docker run -it --name alpha -v /var/log ubuntu bash

In another terminal, let's start another container with the same volume.

- $ docker run --volumes-from alpha ubuntu cat /var/log/now

DOCKER VOLUMES

Demo



DOCKER NETWORKING

DOCKER NETWORKING

Docker includes support for networking containers through the use of network drivers.

By default, the container is assigned an IP address for every Docker network it connects to (the Docker daemon acts as a DHCP server).

By default, a container inherits the DNS settings of the Docker daemon (can be overridden on a per-container basis).

CONTAINER NETWORK MODEL (CNM)

The CNM defines sandboxes, endpoints, and networks

Libnetwork is Docker's implementation of the CNM

Libnetwork is extensible via pluggable drivers

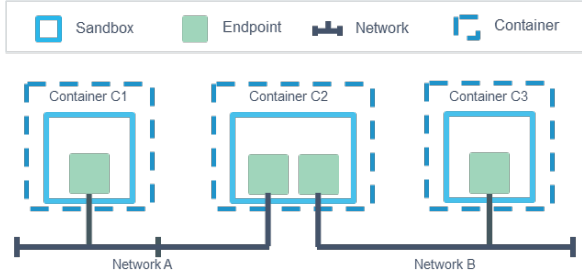
Drivers allow Libnetwork to support many network technologies

Libnetwork is cross-platform and open-source

A CNM has mainly built on 5 objects: Network Controller, Driver, Network, Endpoint, and Sandbox.

The CNM is an open-source container networking specification contributed to the community by Docker, Inc.

The CNM and Libnetwork simplify container networking and improve application portability



CONTAINER NETWORK MODEL OBJECTS

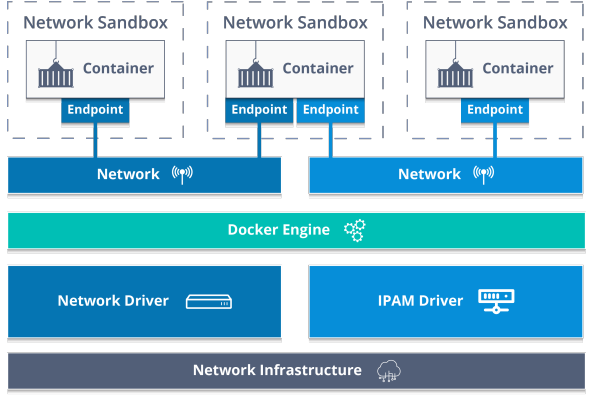
Network Controller: Provides the entry-point into Libnetwork that exposes simple APIs for Docker Engine to allocate and manage networks. Network Controller enables users to attach a particular driver to a given network.

Driver: Owns the network and is responsible for managing the network by having multiple drivers participating to satisfy various use-cases and deployment scenarios.

Network: Provides connectivity between a group of endpoints that belong to the same network and isolate from the rest. When a network is created or updated, the corresponding Driver will be notified of the event.

Endpoint: Provides the connectivity for services exposed by a container in a network with other services provided by other containers in the network. An endpoint represents a service and not necessarily a particular container, Endpoint has a global scope within a cluster as well.

Sandbox: Created when users request to create an endpoint on a network. A Sandbox can have multiple endpoints attached to different networks representing container's network configuration such as IP-address, MAC-address, routes, DNS.



Architecture of Container Networking Model

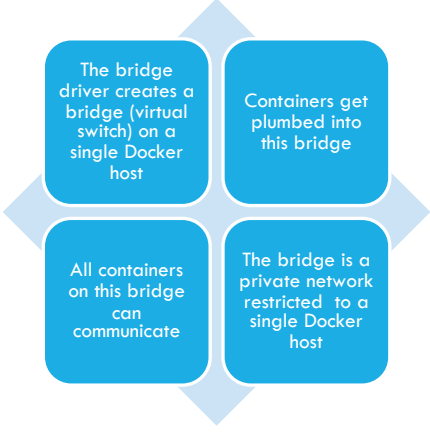
NETWORK DRIVERS

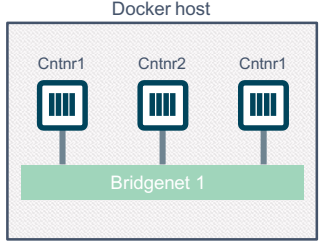
bridge: Allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network.

host: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly.

overlay: Creates a distributed network among multiple Docker hosts. (available only using swarm mode)

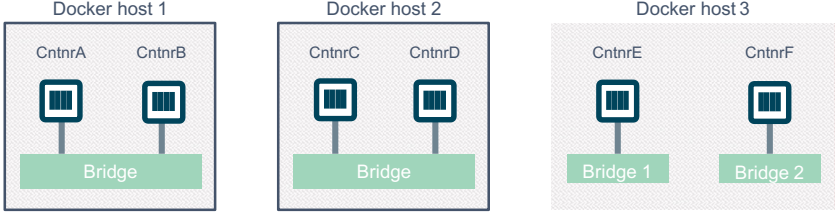
WHAT IS DOCKER BRIDGE NETWORKING?





docker network create -d bridge --name bridgenet1

WHAT IS DOCKER BRIDGE NETWORKING?



Containers on different **bridge** networks cannot communicate

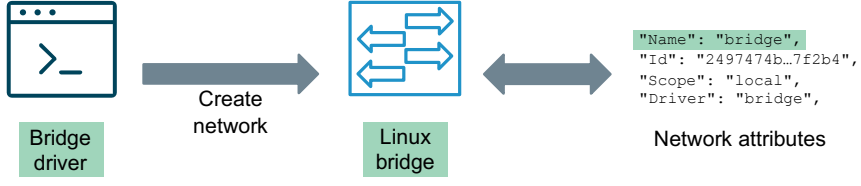
USE OF THE TERM “BRIDGE”

The bridge driver creates simple Linux bridges.

All Docker hosts have a pre-built network called “bridge”

- This was created by the bridge driver
- This is the default network that all new containers will be connected to (unless you specify a different network when the container is created)

You can create additional user-defined bridge networks



BRIDGE NETWORKING IN A BIT MORE DETAIL

The bridge created by the bridge driver for the pre-built bridge network is called docker0

Each container is connected to a bridge

network via a veth pair

Creates a private internal network (single-host)

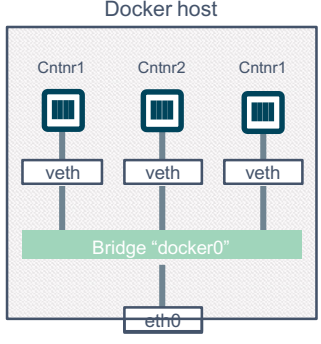
External access is via port mappings

on a host interface

There is a default bridge network called

bridge

Can create user-defined bridge networks



BRIDGE NETWORKING IN A BIT MORE DETAIL

The bridge created by the bridge driver for the pre-built bridge network is called docker0

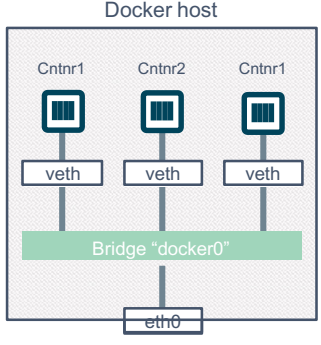
Each container is connected to a bridgenetwork via a veth pair

Creates a private internal network (single-host)

External access is via port mappings on a host interface

There is a default bridge network called bridge

Can create user-defined bridge networks



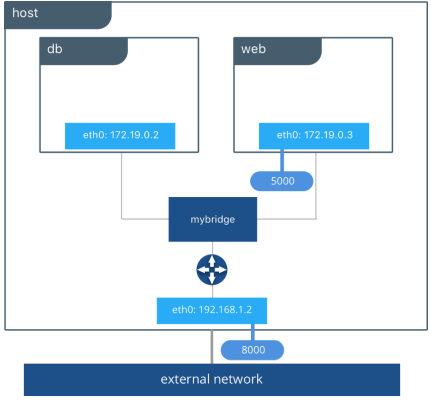
NETWORK DRIVERS

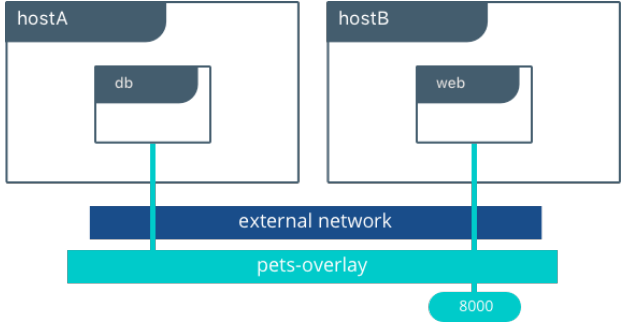
macvlan: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network.

none: disable all networking for the container.

Network Plugins: You can install and use third-party network plugins with Docker. (Develop or download from the Docker Store)

BRIDGE VS OVERLAY NETWORKS





DEFAULT NETWORKS

Every installation of the Docker Engine automatically includes three default networks:

```
$ docker network ls

NETWORK ID          NAME                DRIVER
18a2866682b8        none                null
c288470c46f6        host                host
7b369448dccb        bridge              bridge
```

Unless you tell it otherwise, Docker always launches your containers in the “bridge” network

MANAGING NETWORKS

Create a network:

```
$ docker network create -d bridge network-name
```

Delete a network:

```
$ docker network rm network-name
```

MANAGING NETWORKS

Run a container adding it to an specific network:

```
$ docker run [OPTIONS] --network=network-name [IMAGE]
```

Add running container to a network:

```
$ docker network connect network-name [CONTAINER]
```

MANAGING NETWORKS

Disconnect container from a network:

```
$ docker network disconnect network-name [CONTAINER]
```

Inspect networks:

```
$ docker network inspect network-name
```

MANAGING NETWORKS

Remove all unused networks:

```
$ docker network prune
```

DOCKER NETWORKING ON LINUX

The Linux kernel has extensive networking capabilities (TCP/IP stack, VXLAN, DNS…)

Docker networking utilizes many Linux kernel networking features

(network namespaces, bridges, iptables, veth pairs…)

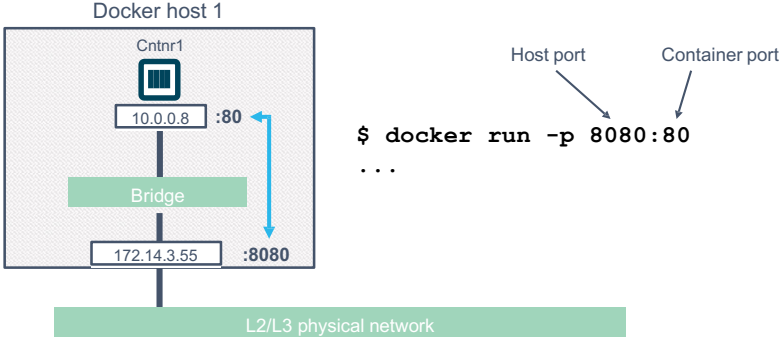
Linux bridges: L2 virtual switches implemented in the kernel

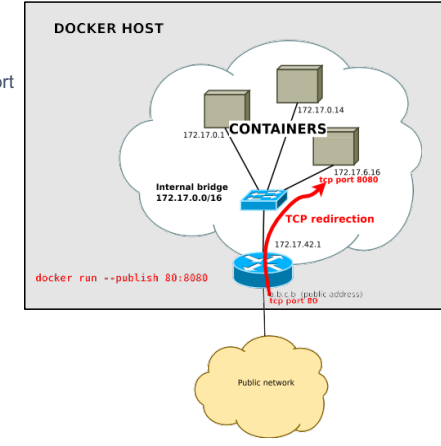
Network namespaces: Used for isolating container network stacks

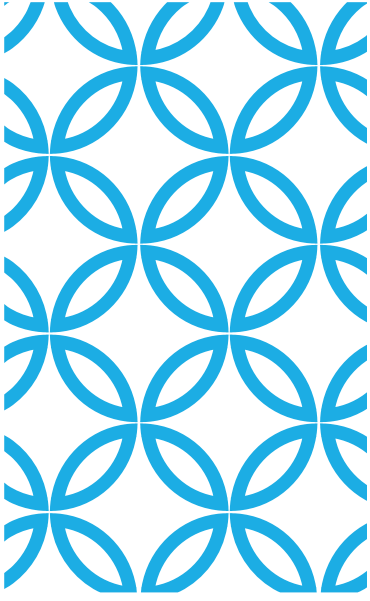
veth pairs: Connect containers to container networks

iptables: Used for port mapping, load balancing, network isolation…

DOCKER BRIDGE NETWORKING AND PORT MAPPING







YAML INTRODUCTION

YAML DEFINITION

YAML was created specifically for common use cases such as:

- Configuration files
- Log files
- Inter-process messaging
- Cross-language data sharing
- Object persistence
- Complex data structures

WHY YAML ?

- They are easily readable by humans. YAML files are expressive and extensible.
- They are easy to implement and use.
- They are easily portable between programming languages.
- They match the native data structures of agile languages.
- YAML files have a consistent model to support generic tools.
- They support one-pass processing.
- They are convenient to use, so you no longer need to add all of your parameters to the command line.
- You can perform maintenance. YAML files can be added to the source control to track the changes.
- They are flexible. You can create much complex structures using YAML than you can use on command line

RELATIONSHIP TO JSON AND XML

XML is a pioneer in many domains. XML was originally designed to be backwards compatible with the Standard Generalized Markup Language (SGML), which was designed to support structured documentation. Because of this, there are many design constraints with XML.

JSON's design goal is simplicity and universality, with its goal to generate and parse. It has reduced human readability, but its data can be processed easily by every modern programming environment.

YAML's design goals are human readability and a more complete information model. YAML is more complex to generate and parse, therefore it can be viewed as a natural superset of JSON. Every JSON file is also a valid YAML file.

XML	JSON	YAML

<Servers>
<Server>
<name>Server1</name>
<owner>John</owner>
<created>123456</created>
<status>active</status>
</Server>
</Servers>
	{
	Servers: [
	{
	name: Server1,
	owner: John,
	created: 123456,
	status: active
	}
	]
	}
		Servers:
		- name: Server1
		owner: John
		created: 123456
		status: active

YAML FILE STRUCTURE

The following are the building blocks of a YAML file:

- Key Value Pair — The basic type of entry in a YAML file is of a key value pair. After the Key and colon there is a space and then the value.
- Arrays/Lists — Lists would have a number of items listed under the name of the list. The elements of the list would start with a -. There can be a n of lists, however the indentation of various elements of the array matters a lot.
- Dictionary/Map — A more complex type of YAML file would be a Dictionary and Map.

Key Value Pair	Array/Lists	Dictionary/Map
Fruit: Apple Vegetable: Radish Liquid: Water Meat: Goat	Fruits: - Orange - Banana - Mango Vegetables: - Potato - Tomato - Carrot	Banana: Calories: 200 Fat: 0.5g Carbs: 30g Grapes: Calories: 100 Fat: 0.4g Carbs: 20g

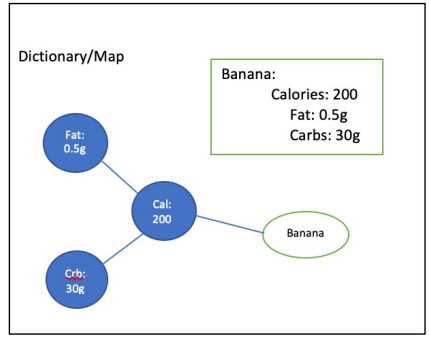
INDENTATION AND TABS

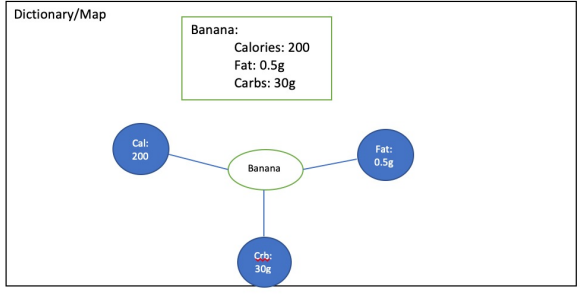
Consider the following diagrams, which has details about "Banana." There are 3 attributes:

Calories = 200

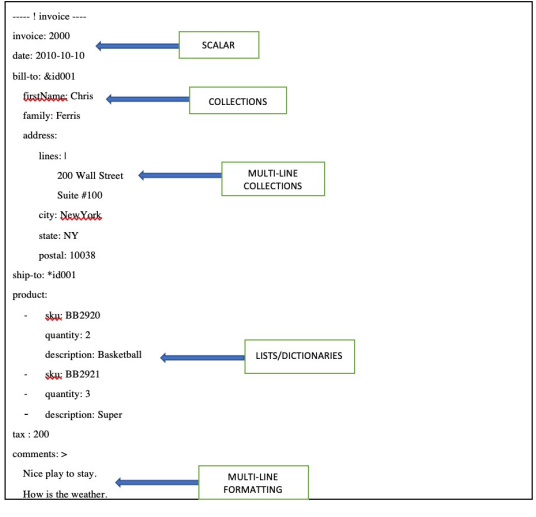
Fat = 0.5g

Carbs = 30g





YAML FILE STRUCTURE



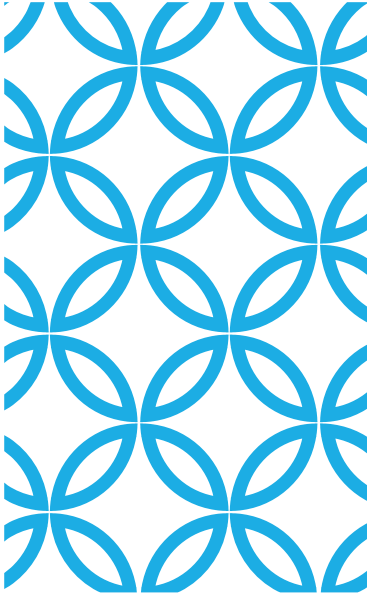
SCALAR — You can mention date in the format of "YYYY-MM-DD" and assign it to a date variable.

COLLECTIONS — Suppose we mentioned the billing address and the shipping address is the same as the billing address. Then you make use of & followed by an ID in front of the billing address. If we need to copy the same address against the shipping address, then we use the * symbol along with the same ID used in the billing address. This helps redundancy of the data.

MULTI-LINE COLLECTIONS — Suppose we have an address line, which has multiple lines and need to maintain the format. Then use the | (vertical bar) symbol.

LISTS/DICTIONARIES — Lists and dictionaries are covered in the previous section.

MULTI-LINE FORMATTING — Suppose we have a long character string value and need to mention on multiple lines maintaining the formatting, then you make use of the symbol > as previously mentioned.



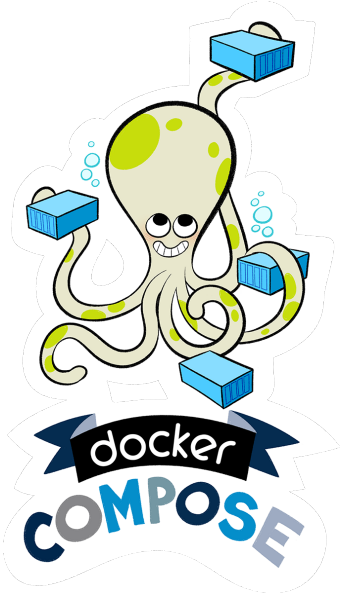
DOCKER COMPOSE

DOCKER COMPOSE

Docker Compose is a tool for defining and running complex applications with Docker.

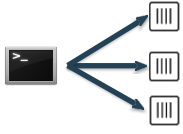
Define a multi-container application in a single file

Spin your application up in a single command

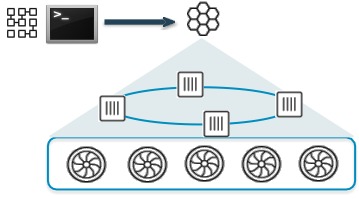


WHAT IS DOCKER COMPOSE ?

- Build and run one container at a time
- Manually connect containers together
- Must be careful with dependencies and start up order



- Define multi container app in compose.yml file
- Single command to deploy entire app
- Handles container dependencies
- Multiple isolated environments on a single host
- Preserve volume data when containers are created
- Only recreate containers that have changed
- Multiple compose files



INSTALLATION

- Windows / Mac - Docker Compose is included
- Linux:

```
sudo curl -L
https://github.com/docker/compose/releases/download/1.21.2/docker-
compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose
```

```
sudo chmod +x /usr/local/bin/docker-compose
```

Test the installation

```
docker-compose --version
```

BEFORE DOCKER COMPOSE

```
$ docker run -d -it --name redis redis
$ docker run -d -it --name postgres linhmtran168/postgres
$ docker run -d -it --name web \ -v
~/Dev/gitlab.com/linhmtran168/test-project:/var/www/html \ --link
postgres:db --link redis:redis linhmtran168/php-web
$ docker run -d -it -p 80:80 --name nginx \ --link web:web --
volumes-from web linhmtran168/php-nginx
$ docker run -d -it --name node --link web:web \ --volumes-from web
linhmtran168/gulp-bower
```

AFTER DOCKER COMPOSE

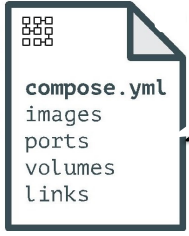
```
web:
  build: .
  links:
    - redis:redis
    - postgres:db
  volumes:
    - .:/var/www/html

  nginx:
  build: ../docker-php-nginx
  ports:
    - "80:80"
  links:
    - web:web
  volumes_from:
    - web
```

YML FILE

- Version
- Services
  - Build
  - Image
  - Environment
  - Ports
  - Volumes
- Volumes
- Networks

```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

SAMPLE YAML FILE




```
version: '2' # specify docker-compose version

# Define the services/containers to be run
services:
angular: # name of the first service
build: client # specify the directory of the Dockerfile
ports:
- "4200:4200" # specify port forewarding

express: #name of the second service
build: api # specify the directory of the Dockerfile
ports:
- "3977:3977" #specify ports forewarding

database: # name of the third service
image: mongo # specify image to build container from
ports:
- "27017:27017" # specify port forewarding
```

DIVE INTO THE YML FILE

Image / Build

image: jenkins/jenkins:lts
build: ./dir

```
version: '3'

services:
  jenkins:
    image: jenkins/jenkins:lts
    ports:
      - "8080:8080"
      - "50000:50000"
  myapp:
    build: ./dir
```

DIVE INTO THE YML FILE

Depends_on

```
web:
    build: .
    depends_on:
      - db
      - redis
```

```
version: '3'

services:
  web:
    build: .
    depends_on:
      - db
      - redis
```

DIVE INTO THE YML FILE

Restart

restart: "no"
restart: always
restart: on-failure
restart: unless-stopped

```
version: '3'

services:
  jenkins:
    image: jenkins/jenkins:lts
    ports:
      - "8080:8080"
      - "50000:50000"
    restart: alwayes
```

DIVE INTO THE YML FILE

Environment

MYSQL_USER: wordpress
MYSQL_PASSWORD: wordpress

Environment File

env_file: .env

```
version: '3'

services:
  jenkins:
    image: jenkins/jenkins:lts
    ports:
      - "8080:8080"
      - "50000:50000"
    restart: always
    environment:
        MYSQL_USER: wordpress
    env_file: .env
```

DIVE INTO THE YML FILE

Networks

```
web:
    networks:
        - mahindra


networks:
        mahindra:
        other:
```

```
version: '3'

services:
  jenkins:
    image: jenkins/jenkins:lts
    ports:
      - "8080:8080"
      - "50000:50000"
    restart: always
    environment:
        MYSQL_USER: wordpress
    networks:
        - mahindra


networks:
    mahindra:
    other:
```

DIVE INTO THE YML FILE

Volumes

volumes:

- "dbdata:/var/lib/postgresql/data"

volumes :

dbdata:

```
version: '3'

services:
  jenkins:
    image: jenkins/jenkins:lts
    ports:
      - "8080:8080"
      - "50000:50000"
    restart: always
    environment:
        MYSQL_USER: wordpress
    networks:
        - mahindra
    volumes:
        - dbdata:/var/lib/postgresql/data


networks:
    mahindra:
    other:

Volumes:
    dbdata:
```

DOCKER-COMPOSE COMMANDS

- Create and start all the containers listed in the “docker-compose.yml”

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

- List all the containers belong to the compose environment instance:

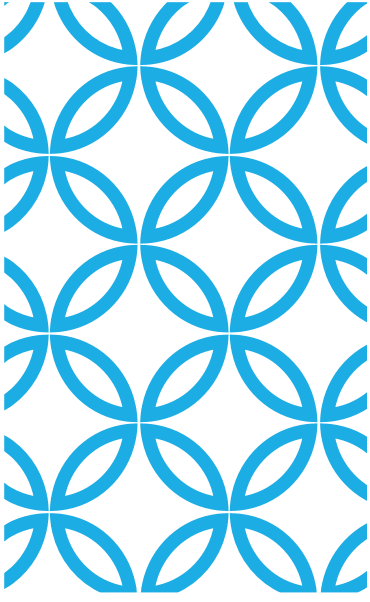
```
$ docker-compose ps
```

- Sets the number of containers:

```
$ docker-compose scale web=3
```

COMMANDS LIST

```
build    Build or rebuild services
help     Get help on a command
kill     Kill containers
logs     View output from containers
port     Print the public port for a port binding
ps       List containers
pull     Pulls service images
rm       Remove stopped containers
run      Run a one-off command
scale    Set number of containers for a service
start    Start services
stop     Stop services
restart  Restart services
up       Create and start containers
```



SECURITY STRATERGIES

SECURITY STRATEGIES

Docker containers are not really sandboxed from the host machine. They talk with the **same kernel**.
You may want to consider strategies to reduce the risks of privilege escalation.

Container/Host isolation

- run the container with an ordinary user (`docker run -u`)
- reduce root privileges (*capabilities*, *seccomp*, *apparmor*)
- configure a user namespace
- run the docker engine inside a VM

Container/Container isolation

- disable intercontainer communications (`--icc=false`)
- isolate containers in different networks

RUNNING CONTAINERS AS NORMAL USER

```
$ docker run -u USER ...
```

should be safe, but...

- setuid executables in the docker image
  → *should mount* `/var/lib/docker` *with* '`-o nosuid`'
- setuid executables in external volumes
  → *should mount all data volumes with* '`-o nosuid`'
- `/etc/passwd` in the docker image
  → *should use numeric ids: (*`docker run -u` *UID:GID)*

→ not easily enforceable if the image provider is malicious

REDUCED ROOT CAPABILITIES

- containers use a default set limited to 14 capabilities[16]:

  AUDIT_WRITE CHOWN NET_RAW SETPCAP DAC_OVERRIDE
  FSETID SETGID KILL NET_BIND_SERVICE
  FOWNER SETUID

  SYS_CHROOT MKNOD SETFCAP

- add additional capabilities: `docker run --cap-add=XXXXX ...`
- drop unnecessary capabilities: `docker run --cap-drop=XXXXX ...`
  → should use `--cap-drop=all` for most containers

REDUCED SYSCALL WHITELIST

seccomp-bpf == fine-grained acces control to kernel syscalls

- default built-in profile[17] whitelists only harmless syscalls[18]
- alternative configs:
  - disable seccomp (`--security-opt=seccomp:unconfined`)
  - provide a customised profile

```
$ docker run --rm debian date -s 2016-01-01
date: cannot set date: Operation not permitted
$ docker run --rm --cap-add sys_time debian date -s 2016-01-01
date: cannot set date: Operation not permitted
$ docker run --rm --security-opt seccomp:unconfined debian date -s 2016-01-01
date: cannot set date: Operation not permitted
$ docker run --rm --cap-add sys_time --security-opt seccomp:unconfined debian date -s 2016-01-01
```

USER NAMESPACES

since docker v1.10 but not enabled by default

- UIDs/GIDs inside the containers mapped to another range outside the container
- useful for:
  - preventing fs-based attacks *(eg: root user inside the container creates a setuid executable in an external volume)*
  - isolating docker users from each other *(one docker daemon for each user, with uids remapped to different ranges)*
- limits (as of v1.10)
  - global config only (daemon scope)
  - coarse mapping only (hardcoded range: 0..65535)

CONTAINER/CONTAINER ISOLATION

- by default all containers can connect to any other container (located in the same bridge)
  - run the daemon with `--icc=false`
    - all communications filtered by default
    - whitelist-based access with `--link`
      (only EXPOSEd ports will be whitelisted)
  - attach containers to different networks
- by default RAW sockets are enabled (allows ARP spoofing)[20]
  → use `docker run --cap-drop=NET_RAW`

OTHER SECURITY CONSIDERATIONS

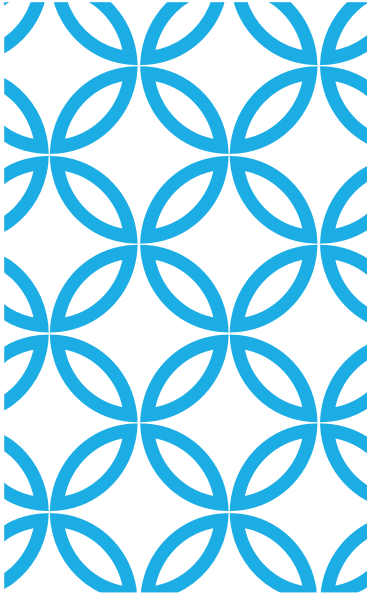
- images are immutable

  → need a process to apply automatic security upgrades, e.g:
  - apply upgrades & commit a new image
  - regenerate the image from the Dockerfile

- docker engine control == root on the host machine
  - give access to the docker socket only to trusted users

- avoid `docker run --privileged` (gives full root access)

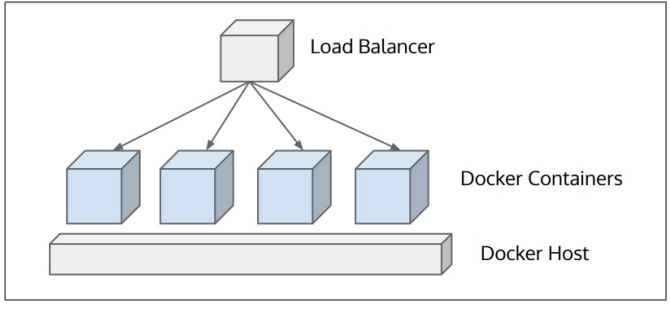
- beware of symlinks in external volumes



THE ECOSYSTEM & THE FUTURE

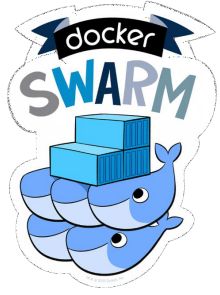
PROBLEMS WITH STANDALONE DOCKER

Running a server cluster on a set of Docker containers, on a single Docker host is vulnerable to single point of failure!



DOCKER ORCHESTRATORS

- Scalability
- Service Discovery
- Networking
- Volume Management
- Monitoring and Logging
- High Availability
- Load Balancing
- Health Checks
- Rolling Upgrades



kubernetes

Apache
MESOS™

KUBERNETES

Abstract away the underlying hardware

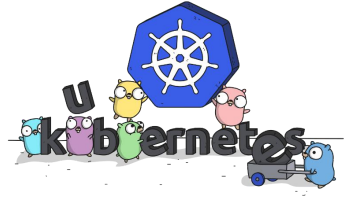
Manage your applications like cattle instead of like pets

Is responsible for maintaining the desired state

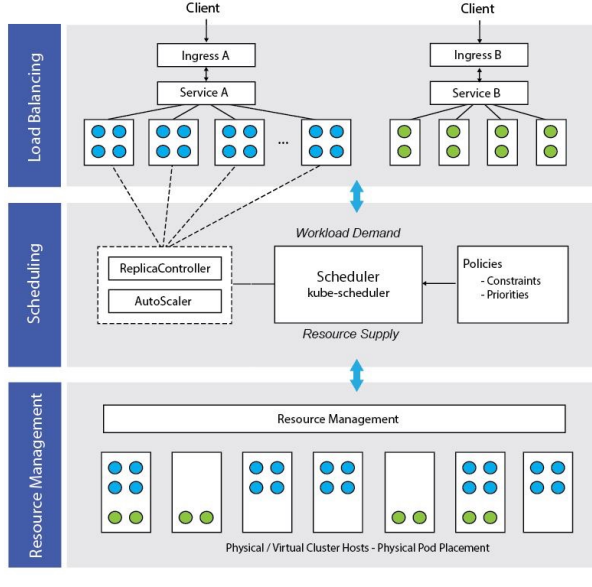
Improves reliability (scale, failover, self-healing)

Better use of infrastructure resources

Manage networking and volumes



KUBERNETES OVERVIEW




THANK
YOU
GRACIAS
ARIGATO
SHUKURIA
GOZAIMASHITA
EFCHARISTO
MERCI
BOLZÏN
BÏYAN
SHUKRIA
TINGKI
DANKSCHEEN
SPASSIBO
NUHUN
SNACHALHUYA
CHALTU
TASHAKKUR ATU
YAQHANYELAY
YUSPAGARÀTAM
HUI
WABEEJA
MAITEKA
SUKSAMA
EKHMET
SPASIBO
DENKAUJA
NENACHALHYA
UNALCHEESH
GUI
HATUR
EKOJU
SIKOMO
MANETJI
MINMONCHAR
DHANYABAAD
ANNA
ATTO
MERSI
MAAKE
GRAZIE
MEHRBANI
PALDIES
LAH
KOMAPSUMNIDA
GANCO
MERASTAWHY
GAEJTHO
AGUYJE
FAKAAUE
TAVTAPUCH
MEDAWAGSE
BAINKA
JUSPAXAR