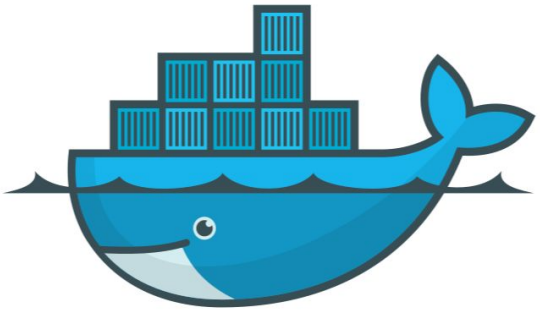
Introduction to Docker and containerization



Introduction

- Solomon Hykes is the creator of Docker (internal project at dotCloud with Andrea Luzzardi, Francois-Xavier Bourlet and others)
- First version published in march 2013
- Docker changed a lot of software development and operations paradigms all at once
- Docker forces people to think in terms of microservices
- Its success was not the technology itself, but the human-friendly interface, APIs and the ecosystem around the project

What is Docker?

- It's a Linux software (now not only on linux)
- It's not a programming language, nor a framework for building software
- It solves problems like installing, removing, upgrading, distributing, trusting and managing software
- Docker simplifies life to sysadmins and developer
- Docker accomplishes this using a UNIX technology called containers

Some history...

- The term “jail” was used in UNIX-style OSs to describe a particular runtime environment where a program was allowed to access protected resources

- In 2005 a new term appeared:
  “container”, access to protected resources + isolated process

Docker technology

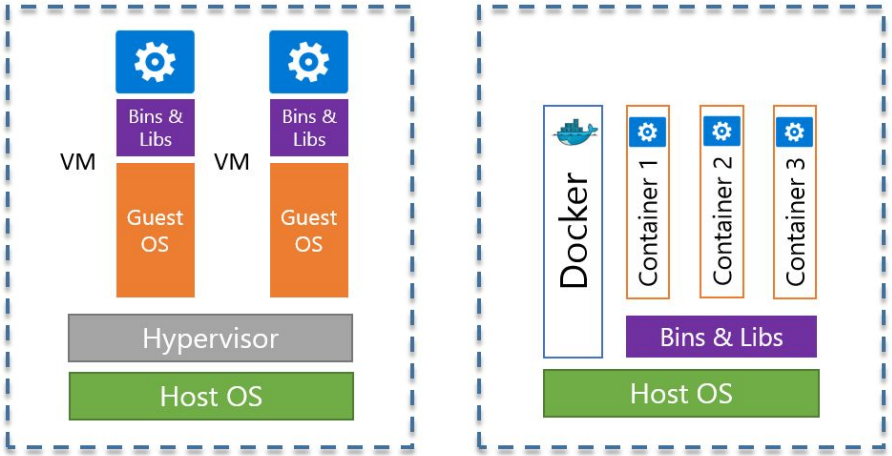
- Containers have existed for decades

- Docker uses Linux namespaces and cgroups (since 2007)
  - https://man7.org/linux/man-pages/man7/namespaces.7.html
  - https://man7.org/linux/man-pages/man7/cgroups.7.html

- Docker makes it simpler to use an existing technology

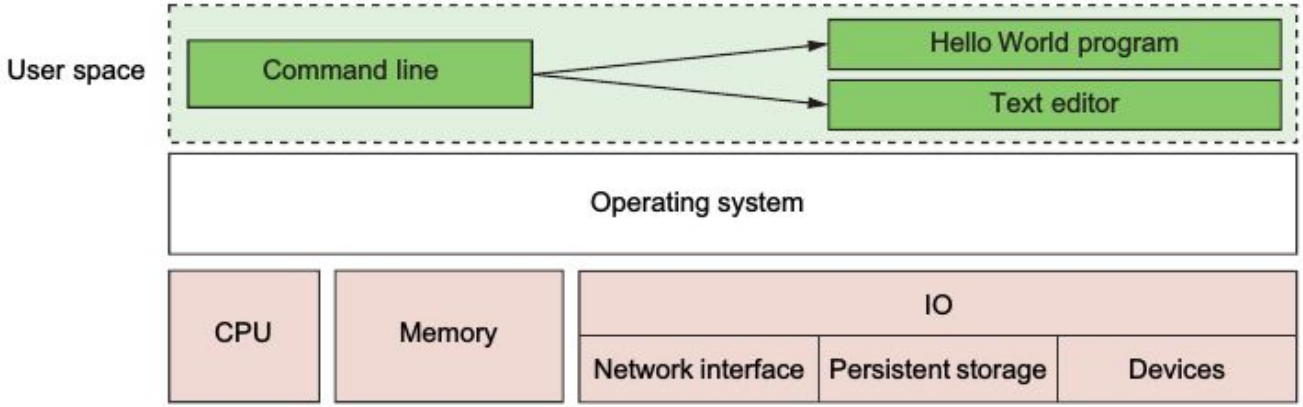
Containers vs VMs

- VMs use hardware virtualization (OS + programs)
- Docker containers don't use hardware virtualization:
  they directly interact with host's Linux kernel



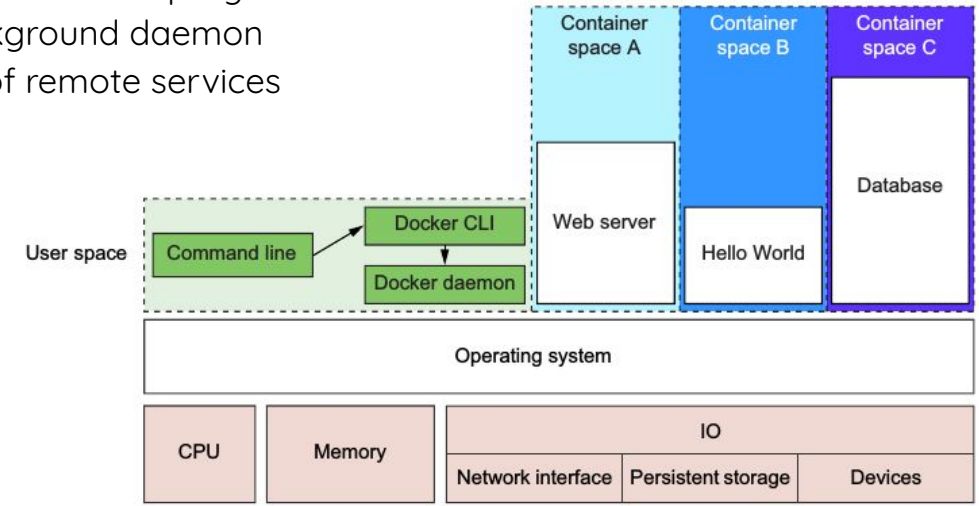
Containers vs VMs

- Programs running in user space cannot modify kernel space memory
- Kernel is the interface between programs and hardware



Containers vs VMs

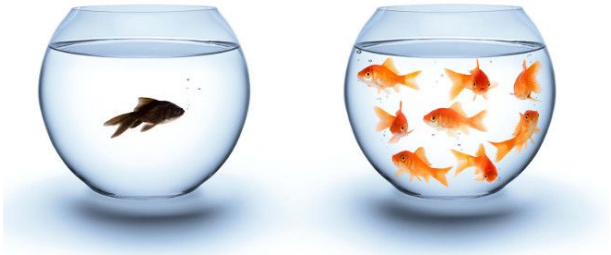
- Example with Docker and 3 containers
- Programs running in a container can access only memory and resources as scoped by the container
- Docker is:
    - A command-line program
    - A background daemon
    - A set of remote services



Some history...

➢ Containers that Docker builds are isolated with respect to 8 aspects:

1. **PID namespace** - Process identifiers and capabilities
2. **UTS namespace** - Host and domain name
3. **MNT namespace** - File system access and structure
4. **IPC namespace** - Process communication over shared memory
5. **NET namespace** - Network access and structure
6. **USR namespace** - User names and identifiers
7. **Chroot()** - Controls the location of the file system root
8. **Cgroups** - Resource protection



Docker containers

- A docker container is like a physical shipping container
- Docker can run, copy and distribute containers
- A container is filled with an image
  (snapshot of all files available to a running program inside a container)
- Images are distributed using registries and indexes
  (public or private)

Docker containers

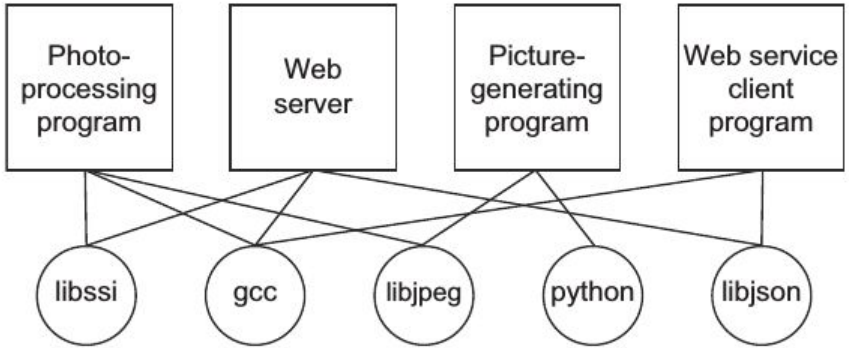
Why Docker?

- Software deployment is a complex task

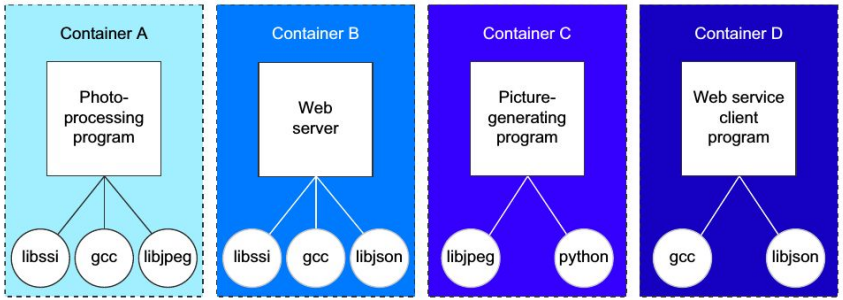
- A lot of considerations before acting:
  - How many resources?
  - Which dependencies?
  - What about other running applications?
  - Security?
  - Updates?

- The more software, the more difficulties to manage

Before Docker...



With Docker...



Advantages with Docker

- Improving portability
- More abstraction (containers are movable, resizable)
- Consistent and specific environments
- More and better focus on software development
- Company infrastructure as a cloud
- More protection because of isolation:
    - Anything inside a container has access only to a limit set of resources
    - In case of failure or malicious software, the problem is limited to a single container
- Save time, money, energy
- Companies like Amazon, Google, Microsoft contribute, support and push Docker adoption (instead of developing their own solutions)

Docker installation

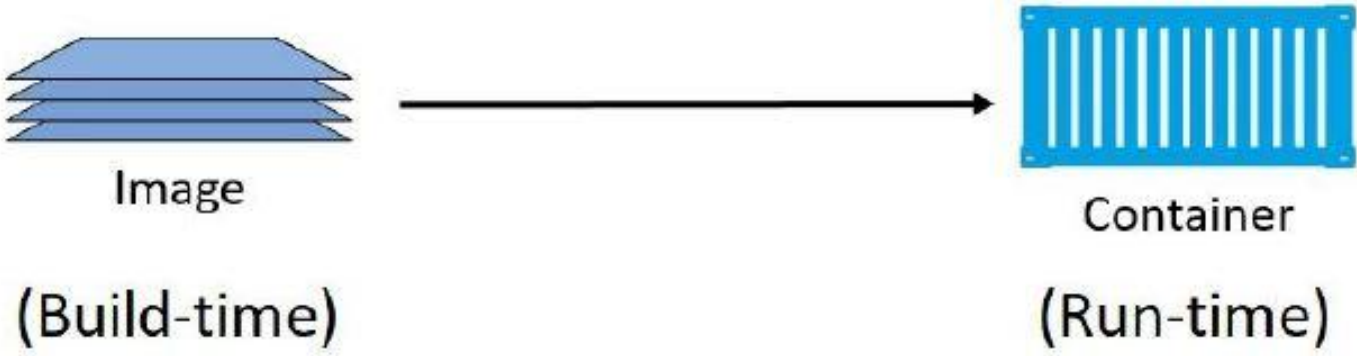
- Instructions at: https://docs.docker.com/engine/installation/

- In this course we are going to use docker on Linux

- Test your installation:

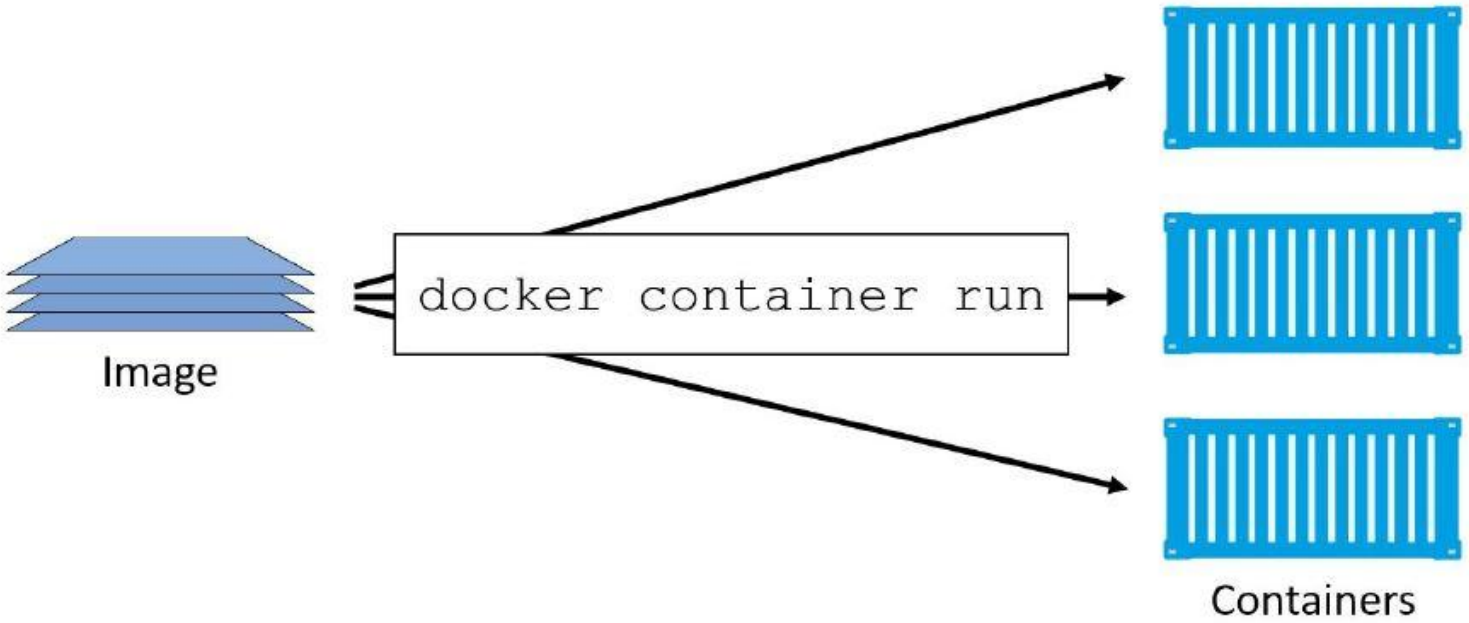
  % **docker run hello-world**

Container’s building blocks



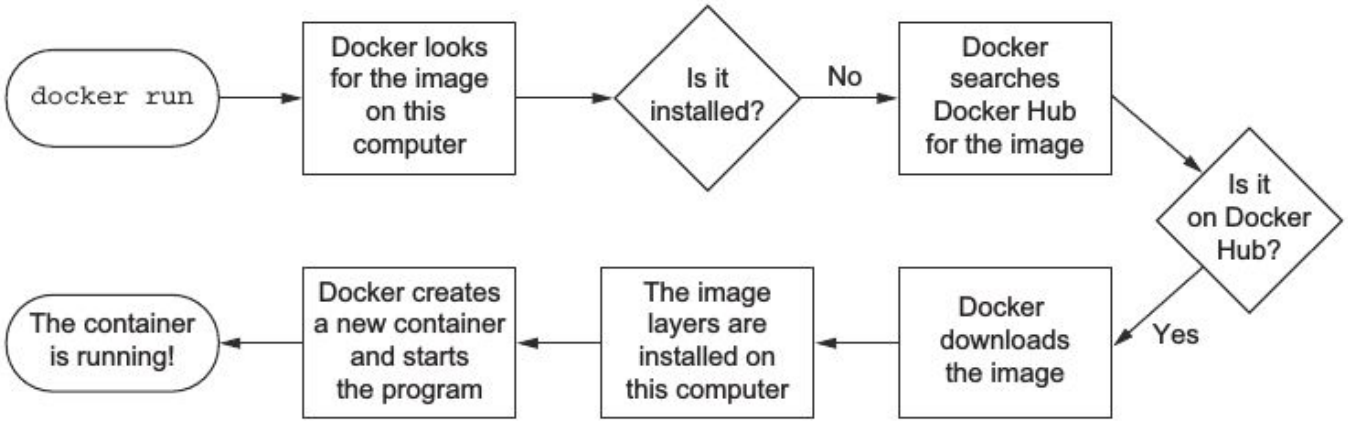
Open Container Initiative (OCI) standardized image-spec and runtime-spec (since Docker 1.11)

Docker Run



A container is the runtime instance of an image

What happens with run command?



If run a second time, the installed image now will be found in local machine

Docker command line

- We are going to use CLI

  % **docker help**

  displays all available commands

  % **docker help cp**

  displays usage pattern for “cp” command

First step with docker

- Let's install NGINX with the following command:

  % **docker run --detach --name web nginx:latest**

  Unable to find image 'nginx:latest' locally

  latest: Pulling from library/nginx

  afeb2bfd31c0: Pull complete

  7ff5d10493db: Pull complete

  d2562f1ae1d0: Pull complete

  Digest: sha256:af32e714a9cc3157157374e68c818b05ebe9e0737aac06b55a09da374209a8f9

  Status: Downloaded newer image for nginx:latest

  bd46d087282c701d274f0e21dd9548c8ab1d9906128775ef6d324cce2059419c

- New image nginx is downloaded from docker hub

  (this image is composed by 3 parts: afeb2bfd31c0, 7ff5d10493db, d2562f1ae1d0)

- A new container is created and its identifier is shown (bd46d08728...)
- The **--detach** (**-d**) option runs the program in background

What's running?

- "**docker ps**" command shows the running containers with their status and other info

```
% docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS         PORTS     NAMES
bd46d087282c   nginx:latest   "nginx -g 'daemon ..."   20 minutes ago  Up 20 minutes  80/tcp    web
```

```
% docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS                     PORTS     NAMES
e30ba27db63f   busybox:latest "/bin/sh"                2 minutes ago   Exited (0) 1 second ago              test_web
b4a783a65f77   hello-world    "/hello"                 20 minutes ago  Exited (0) 20 minutes ago            practical_ardinghelli
bd46d087282c   nginx:latest   "nginx -g 'daemon ..."   20 minutes ago  Up 20 minutes              80/tcp    web
```

- If not specified, a random name is chosen for the container. Names are more human-friendly than Ids (which usually are truncated to only 12 chars). You can use both to identify a container

Container duplication

- Let’s try to start another container:

  % **docker run --detach --name web nginx:latest**

  docker: Error response from daemon: Conflict. The container name "/web" is already in use by container "bd46d087282c701d274f0e21dd9548c8ab1d9906128775ef6d324cce2059419c". You have to remove (or rename) that container to be able to reuse that name.

- See '**docker run --help**'
  - Solution 1**:** rename

    % **docker rename web web_old**
  - Solution 2**:** stop and remove

    % **docker stop web**

    % **docker rm web**

Restart and Log

- Container's restart:

  % **docker restart web**

  web

- Show container's logs:

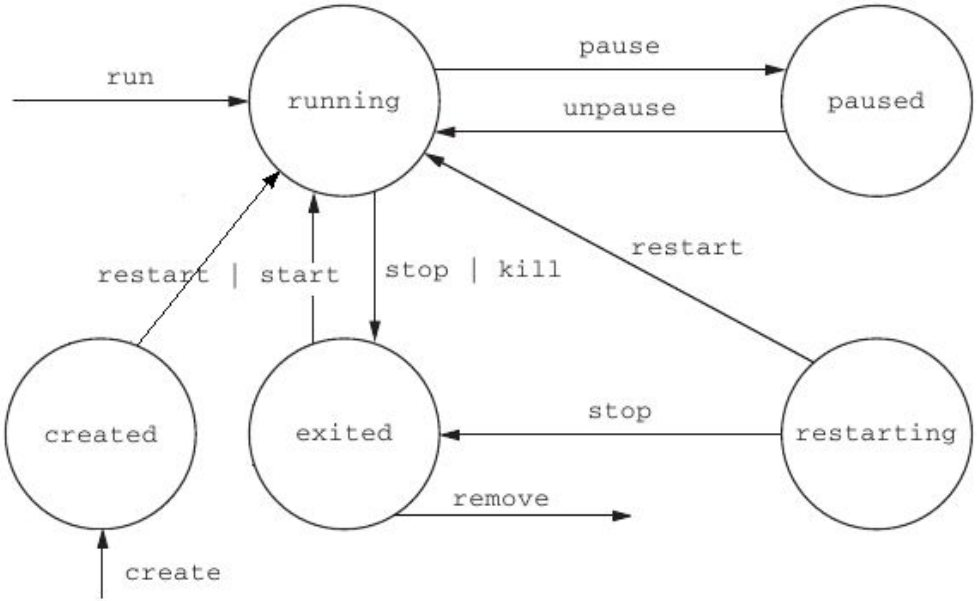
  % **docker logs web**

  172.17.0.3 - - [02/Oct/2017:14:10:17 +0000] "GET /
  HTTP/1.1" 200 612 "-" "Wget" "-"

Containers transition state

- The Docker Remote API (version 1.24) defines the following states:
  - **created:** a container that has been created (e.g. with docker create) but not started
  - **restarting:** a container that is in the process of being restarted
  - **running:** a currently running container
  - **paused:** a container whose processes have been paused
  - **exited:** a container that ran and completed ("stopped" in other contexts, although a created container is technically also "stopped")
  - **dead:** a container that the daemon tried and failed to stop (usually due to a busy device or resource used by the container)

Commands to manage states



What about container failures?

- It's important to restore the service as quickly as possible
- Basic strategy: restart the process (**--restart** option)
- Use of a supervisor process like: init, systemd, runit, upstart, supervisord
  – *Supervisor will restart the failed process*
- Connect to the container for troubleshooting or debug: **docker exec -ti <container> sh**
- Use of entrypoint (**--entrypoint**) to pass a script with preconditions for starting the contained software

Clean up

% **docker stop <container>**

% **docker rm <container>**

- Container must be in "exited" state otherwise...

  Error response from daemon: Conflict, You cannot remove a running container.

  Stop the container before attempting removal or use -f

  FATA[0000] Error: failed to remove one or more containers

- Brutal ways to stop a container:

  % **docker kill <container>**

  % **docker rm -f <container>**

- **--rm** option will remove the container when exited:

  % **docker run --rm <container>**

  % **docker rmi <image>**

Software installation

- Docker gives you 3 choices:
  - Install software from a registry (ex. Docker hub)
  - Install software from a image file (docker save, docker load)
  - Building an image with a dockerfile

- Identify your software:
  - <registry_host>/<username>/<name>:<tag>
  - example: hub.docker.com/repoName/gitea:v1

  % **docker search postgres**

  % **docker pull <image>**

- Be careful with downloaded third-party images!

Other ways to install

- Use private or alternative registries:

  % **docker pull quay.io/johndoe/pingpong:v3.2**

  % **docker save -o myimage.tar busybox:latest**

  % **docker load -i myimage.tar**

- Docker can build images automatically by reading the instructions from a Dockerfile, a text file containing all commands needed to build an image

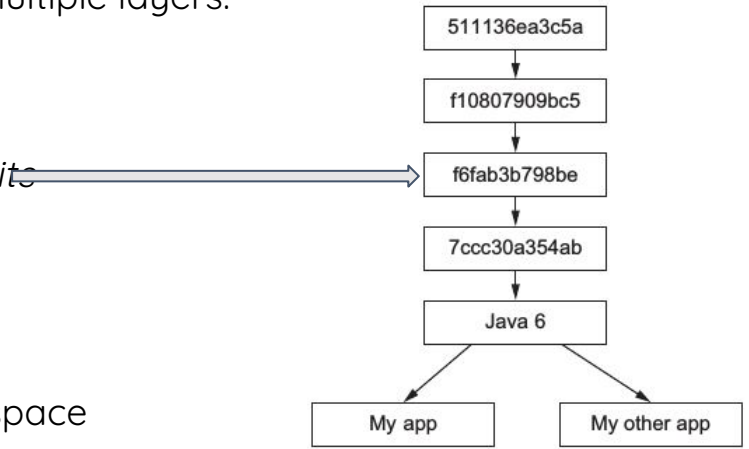
```
FROM busybox:latest
MAINTAINER demo@demo.it
ADD demo.sh /demo/
WORKDIR /demo/
CMD ./demo.sh
```

.

More about Docker images...

- Images are not monolithic files but they are organized in reusable layers
- Images are read-only
- Example of Java6 docker image composed by multiple layers:

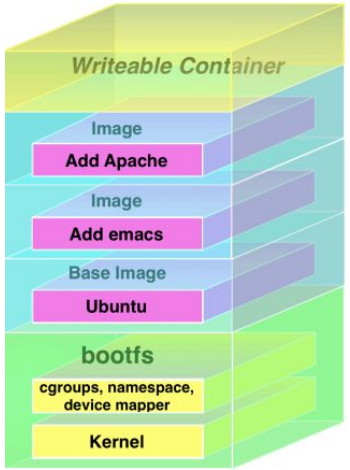
If an image is not tagged, it is identified by 12 digits
(shortened version of a 65 digit UID)



- Layers enable reuse and saving bandwidth and space

More about Docker images...

- Union file systems, or UnionFS, are file systems that operate by creating layers, making them very lightweight and fast
- Docker Engine uses UnionFS to provide the building blocks for containers
- Docker Engine can use multiple UnionFS variants:
  - AUFS
  - btrfs
  - vfs
  - DeviceMapper
  - overlay2



Environment variable injection

- Envars are used to communicate information between containers
- Injection at execution:

```
% docker run --env MY_VAR="hello" alpine:latest env
```

- Env vars depend on the image you use:

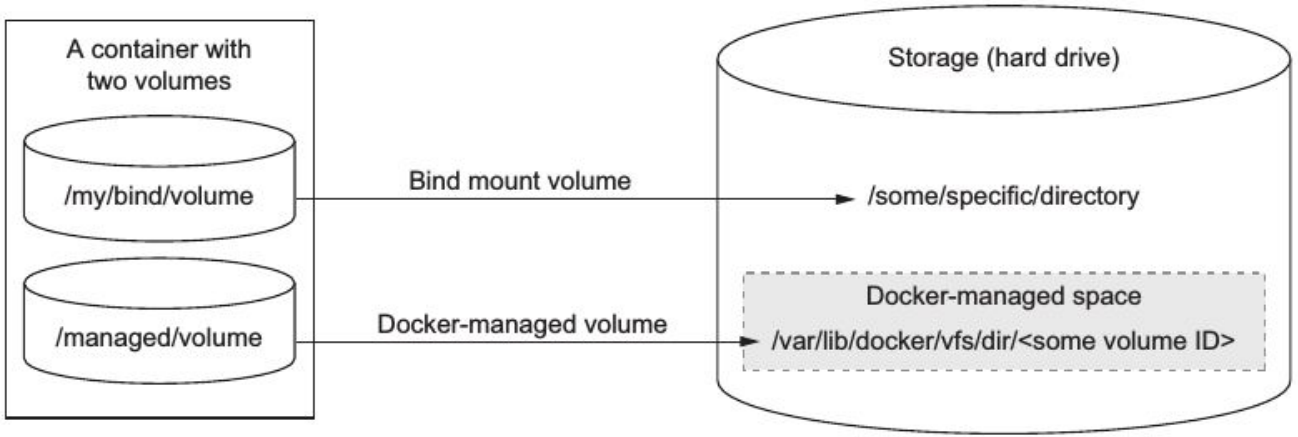
```
% docker run -d --name db postgres:alpine
% docker ps
% docker logs db
% docker run -d --name db -e POSTGRES_PASSWORD=password
postgres:alpine
```

Volumes

- So far we've seen ephemeral containers: data are not stored on a physical support and are lost when the container is turned off
- In Docker volumes are used to persist data
- A volume is a mount point on the container's directory tree

Volumes types

- There are 2 types of volumes:
    1. Bind mount: directory or file on the host OS
    2. Docker-managed: space controlled by Docker daemon



Bind mount volume

- Useful when you want to shared data with some processes running outside the container
- Example: documents for apache server

% **docker run -d --name myweb -v ~/apache-docs:/usr/local/apache2/htdocs:ro -p 8001:80 httpd:latest**

- The **-v <host_location>:<container_location>**: options is the parameter to create a bind mount. A folder called apache-docs is going to be created in current folder on host computer (but it's better create the folder manually). Its content is read-only ('ro' option) by the container processes
- The original content in **/usr/local/apache2/htdocs** is masked by the content in **apache-docs**
- Bind mount works for individual files too, not only for folders
- Be careful with concurrency on the same volume (ex. with databases)

Docker-managed volume

- Volumes are created in a portion of the host's filesystem, owned by Docker:

  % **docker volume create myvol**

  The **-v <container_location>** options is the parameter to create a docker-manage volume

  % **docker run -d --name myweb -v myvol:/usr/local/apache2/htdocs -p 8001:80 httpd:latest**

  % **docker inspect -f "{{json .Volumes}}" myweb**

  to find the location of the mount point, eg::

  {"/usr/local/apache2/htdocs":"/mnt/sda1/var/lib/docker/vfs/dir/612fd64c..."}
- When specific locality of data is not important, Docker-managed volumes are a much more effective way to organize data
- When you're finished with volumes, you may ask Docker to erase them for you
- What are my defined volumes?

  % **docker volume ls**

  % **docker volume inspect <volume_name>**

Sharing volumes

- You can share a volume between containers
- There are 2 types of sharing volumes:

  Host-dependent: two or more containers mount on the same host's folder

  (-v option, as seen before)

  Generalized sharing: using --volumes-from <container>
- Generalized sharing examples:

  % **docker run -v /dbdata --name dbstore alpine touch /dbdata/mydata**

  % **docker run --rm --volumes-from dbstore -v $(pwd):/backup alpine tar cvf \ /backup/backup.tar /dbdata**

  % **tar tvf backup.tar**

- Data-only containers are used by other containers to mount data volume

  (even if DO containers are in stopped state)

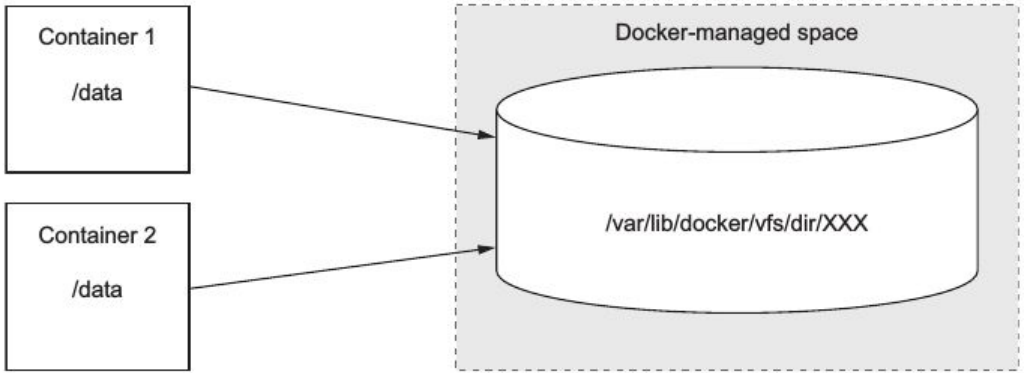
Shared volume example

- Multiple containers can share the same volume:

```
% docker volume create my-shared-vol

% docker run -d --name web1 -v my-shared-vol:/usr/local/apache2/htdocs -p 8001:80 \
httpd:alpine
% docker run -d --name web2 -v my-shared-vol:/usr/local/apache2/htdocs -p 8002:80 \
httpd:alpine

% docker cp index.html web1:/usr/local/apache2/htdocs

% curl localhost:8002
```

Managed volumes Lifecycle

- Managed volumes are only identifiable by the container that use them
- Docker tracks all the references between volumes and containers and ensures that no currently referenced volume is deleted



Cleaning up volumes

- It's a manual task (for safety reasons!)

- Docker doesn't delete bind mount, because they are outside its scope

  % **docker rm -v <container>**

  removes a container and its managed volumes

- If you fail to use -v option, you can leave some orphan volumes (unusable space). They must be removed manually:

  % **docker volume ls -qf dangling=true**

  % **docker volume rm $(docker volume ls -qf dangling=true)**

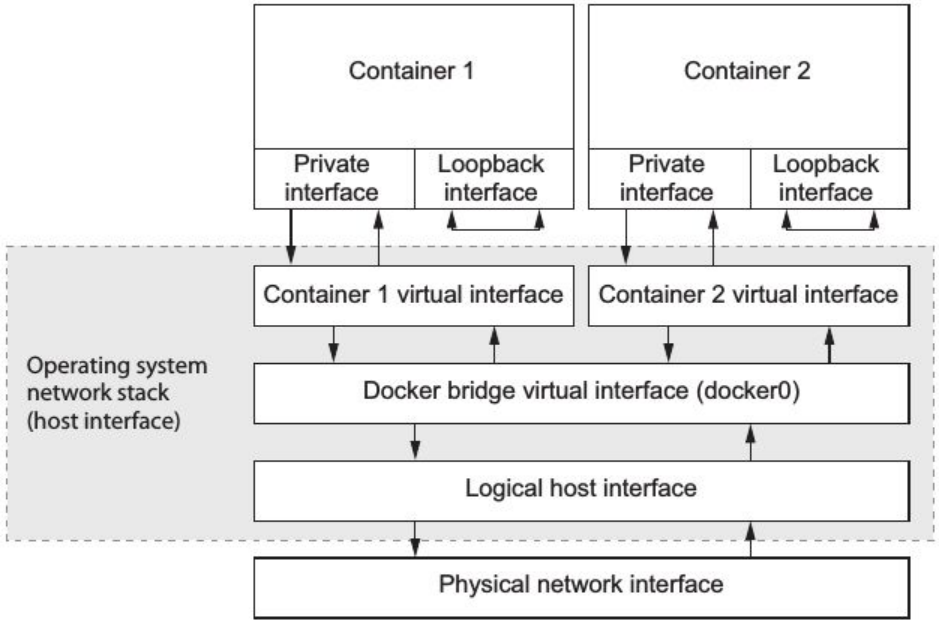
- or use directly:

  % **docker volume prune**

- A third-party image may contain a mounted volume (inspect image to know about it)

Networking

- Docker uses underlying OS to build a virtual network



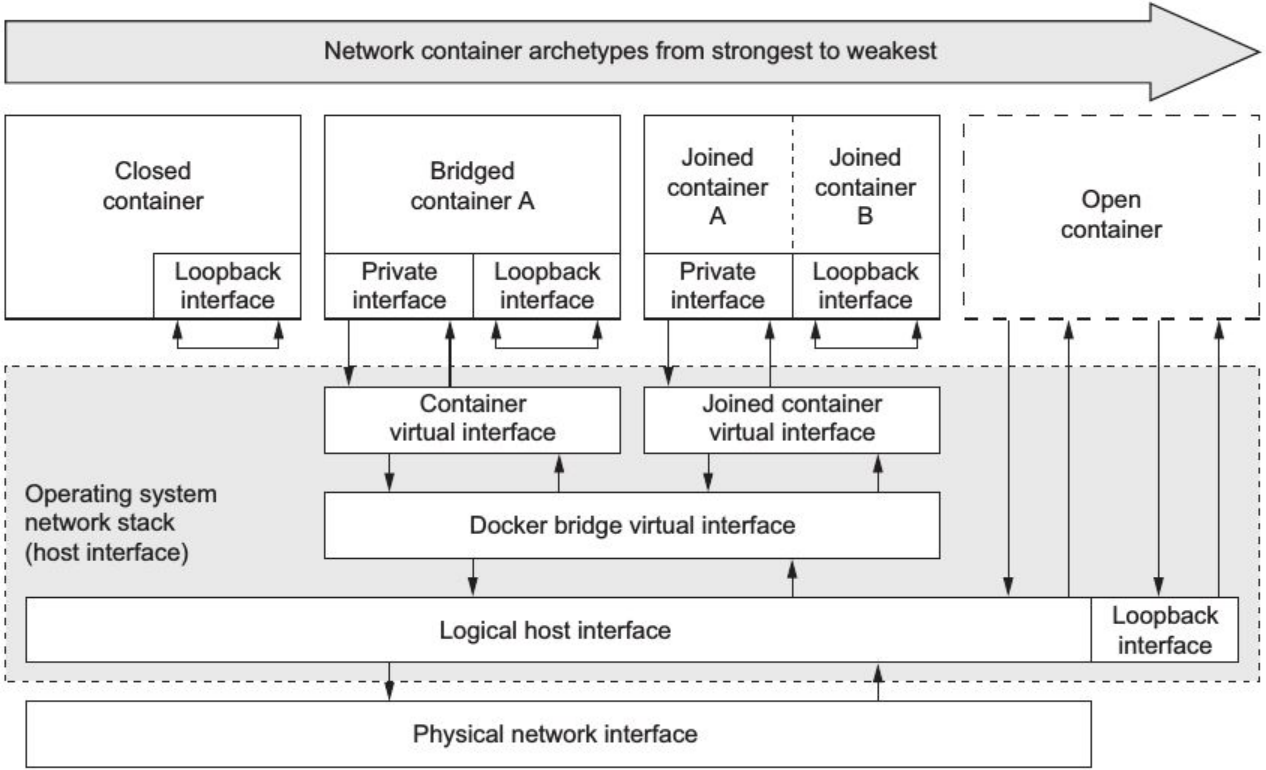
Network archetypes

- Archetypes defines how a container interacts with local and host's network:

1. Closed container
2. Bridged container
3. Open container

- Archetypes define a different level of isolation

Network archetypes



Closed container

- In a closed container only loopback interface is available

- How to create a closed container and show interfaces:

  % **docker run --rm --net none alpine:latest ip addr**

  1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000

  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

  inet 127.0.0.1/8 scope host lo

  valid_lft forever preferred_lft forever

- This kind of archetype provides the most isolated container

Bridged container

- Bridged container has a private loopback interface and another private interface
- Bridged container is the most common archetype
- How to create a bridged container and show interfaces:

    % **docker run --rm --net bridge alpine:latest ip addr**

    1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000

        inet 127.0.0.1/8 scope host lo

        valid_lft forever preferred_lft forever

    1502: eth0@if1503: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP

        link/ether 02:42:ac:11:00:04 brd ff:ff:ff:ff:ff:ff

        inet 172.17.0.4/16 brd 172.17.255.255 scope global eth0

Open container

- Open containers gives full access to host's network (there is no isolation)

- You should use open container only when you have no other option

% docker run --net host alpine:latest ip addr

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
2: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 14:cc:20:01:3f:93 brd ff:ff:ff:ff:ff:ff
    inet 172.16.1.41/24 brd 172.16.1.255 scope global eth2
        valid_lft forever preferred_lft forever
    inet6 fe80::16cc:20ff:fe01:3f93/64 scope link
        valid_lft forever preferred_lft forever
3: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN qlen 1000
    link/ether 94:de:80:24:49:16 brd ff:ff:ff:ff:ff:ff
    wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether e8:94:f6:06:51:81 brd ff:ff:ff:ff:ff:ff
5: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:b8:c9:7f:a4 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
        valid_lft forever preferred_lft forever
```

DNS

- Docker provides names resolution for containers inside the same network

$ docker network create mynet

$ docker run -d --name myweb --net mynet httpd:alpine

$ docker run --rm -ti --net mynet alpine ping -c2 myweb

PING myweb (172.29.0.2): 56 data bytes

64 bytes from 172.29.0.2: seq=0 ttl=64 time=0.096 ms

64 bytes from 172.29.0.2: seq=1 ttl=64 time=0.111 ms

DNS configuration

- Docker provides different options to customize DNS configuration

--hostname option is used to give a name to the container:

% **docker run --rm --hostname demo alpine:latest ping -c2 demo**

PING demo (172.17.0.2): 56 data bytes

64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.049 ms

64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.160 ms

DNS configuration

- ...more options to customize DNS configuration...

  you can specify a DNS to be used:

  % **docker run --dns 8.8.8.8 alpine:latest nslookup world.it**

  --dns=[ ] option is used to specify more than one DNS

  --dns-search = [ ] option is used to provide a search domain

  --add-host = [ ] option is used to write a /etc/hosts file

DNS configuration

➢ Example:

% **docker run --rm --hostname mycont --add-host batman:127.0.0.1 --add-host wolverine:10.10.9.7 alpine:latest**

% **cat /etc/hosts**

```
127.0.0.1       localhost
::1             localhost ip6-localhost ip6-loopback
fe00::0         ip6-localnet
ff00::0         ip6-mcastprefix
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters
127.0.0.1       batman
10.10.9.7       wolverine
172.17.0.2      mycont
```

Inbound communications

- By default a bridged container is not accessible from the network

- Option **--publish = [ ]** (or **-p=[ ]**) is used in order to create mapping between a host network port and a container network interface
- mapping examples:

<imagePort>

% **docker run -P**

<containerPort>

% **docker run -p 6789**

<hostPort>:<containerPort>

% **docker run -p 4444:5555**

<ip>::<containerPort>

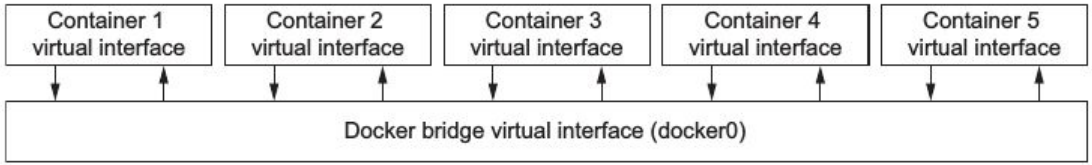
% **docker run -p 10.10.1.2::4444**

<ip>:<hostPort>:<containerPort>

% **docker run -p 10.10.1.2:3333:4444**

Inbound communications

- All local bridged containers are on the same network bridge and they can communicate with each other by default



- Disabling inter-container communication is very important for security reasons

Resources caps

- You can set restrictions on using resources when using docker run and docker create commands

- Option **-m** or **--memory** sets the memory upper limit, it is not a reservation of resource

$ docker stats --format "table {{.Name}}\t{{.CPUPerc}}\t{{.MemUsage}}\t{{.MemPerc}}"

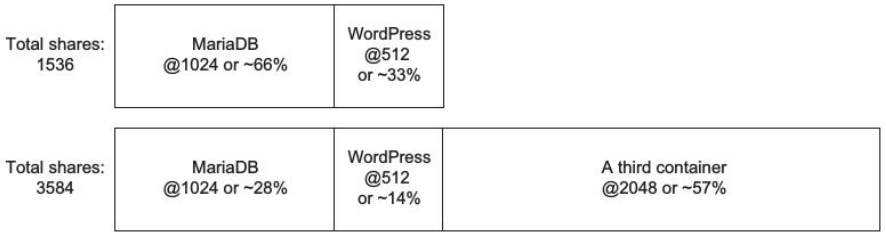
$ docker run --memory 50m --memory-swap 0m --rm alpine free -m

$ docker run --memory 100m --memory-swap 0m --rm -it progrium/stress --vm 1 --vm-bytes 62914560 --timeout 1s

$ docker run --memory 50m --memory-swap 0m --rm -it progrium/stress --vm 1 --vm-bytes 62914560 --timeout 1s

Resources caps

➢ Option **--cpu-shares <integer>** specifies the relative container weight



Resources caps

- Option **--cpuset-cpus <integers>** specifies the cores to assign to containers

  % **docker run ... --cpuset-cpus 0,1,2**
  - comma separated 0,1,2,...
  - range 0-2

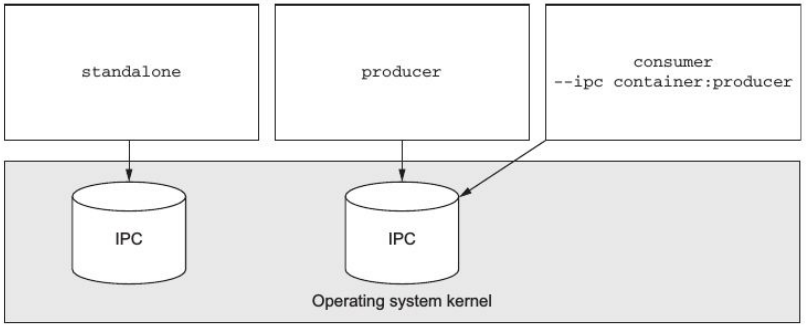
- Option **--device <host_device>:<container_device>** grants access to a device

  % **docker run ... --device /dev/video0:/dev/video0...**

Shared memory

- Linux provides methods for sharing memory between processes (IPC: inter-process communication)
- Option **--ipc container:<container>** enables this feature:

  % **docker run … --ipc container:my_db …**



- Sharing memory between containers is safer than sharing memory with the host (option **-- ipc host**)

Users

- A container is started as root user (by default)

- What user is going to be used when a container is created from an third-party image?

  % **docker inspect --format "{{.Config.User}}" <container>**

  - If the command returns no value then → root user else it returns the default run-as user

Users

- At container creation time, you can change the run-as user and group

  % **docker run --user nobody alpine:latest id**

  % **docker run -u nobody:nogroup alpine:latest id**

- You may set UID and GID that do not exist in the container

  % **docker run -u 10000:20000 alpine:latest id**

- Option **--privileged** grants the container full privileges to the host

  % **docker run --privileged alpine:latest ls /dev | wc -l**

  % **docker run alpine:latest ls /dev | wc -l**

Building Docker images

- Building Docker images is a 3 steps process:
  1. Create a container from an existing image
  2. Modify the filesystem
  3. Commit all the changes

- Step 2 will add new layers on top of the existing UFS (Union File System)

- What are my changes?

  % **docker diff <container>**

  the output will show a (long) list of files:

  Added (A)

  Changed (C)

  Deleted (D)

Building Docker images

- When the process is finished, you have to commit:

  % **docker commit -a “my@me” -m “Comment” <container> <new_image>**

- When a layer is added, metadata context is also included to the image, which includes:
  - Environment variables
  - Working directory
  - All exposed ports
  - Entrypoint
  - Command and arguments

- Every layer inherits the context from the previous layer

Images: useful commands

% **docker history <image>**

- shows all layers in an image and some historical info

% **docker export --output myimage.tar <container>**

- flattens all layers in a single container-perspective layer (lost of history)

% **docker import myimage.tar**

- import has the option '**-c <dockerfile_command>**' to specify a command (eg. entrypoint)

% **docker tag <source_image[:tag]> <targer_image[:tag]>**

- tags an existing image with a different id:tag
- Key concept: multiple tags can reference the same image
- Use pragmatic tag scheme to ease user adoption and migration control
- Be careful using latest tag (it's the default tag)

Entrypoint

- **ENTRYPOINT** has two forms:

  The exec form, which is the preferred form:
  ENTRYPOINT ["executable", "param1", "param2"]
  The shell form:
  ENTRYPOINT command param1 param2

Command line arguments to docker run <image> will be appended after all elements in an exec form ENTRYPOINT, and will override all elements specified using CMD. This allows arguments to be passed to the entry point, i.e., **docker run <image> -d** will pass the **-d** argument to the **entrypoint**. You can override the ENTRYPOINT instruction using the docker run --entrypoint flag.

CMD

➢ The **CMD** instruction has three forms:

CMD ["executable","param1","param2"] (exec form, this is the preferred form)
CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
CMD command param1 param2 (shell form)

There can only be one CMD instruction. The main purpose of a CMD is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an ENTRYPOINT instruction as well.
If CMD is used to provide default arguments for the ENTRYPOINT instruction, both the CMD and ENTRYPOINT instructions should be specified.

ENV

- **ENV** sets environment variables which are present during container build and remain existent in the image.

  ENV <key> <value>

  ENV <key>=<value> ...

- On container startup they can be overwritten with the **-e** or **--env** option:

  docker run -e key=value my_image

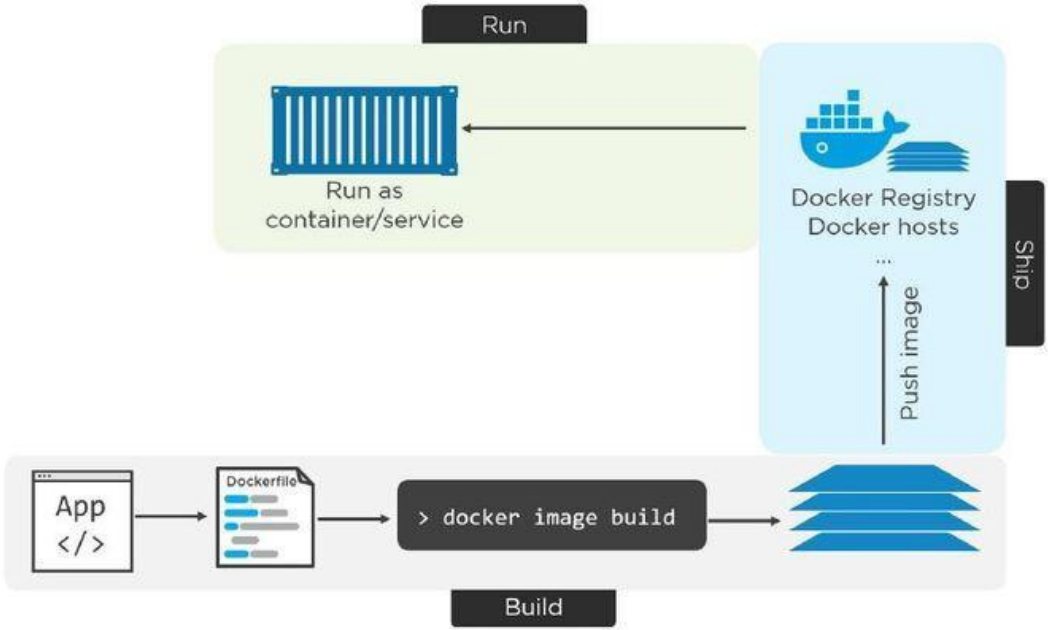
Example:

docker run -e message="The answer is" -e answer="42" cavatortaluca/exam:centos8 bash -c 'echo $message $answer'

The answer is 42

Build - Ship - Run

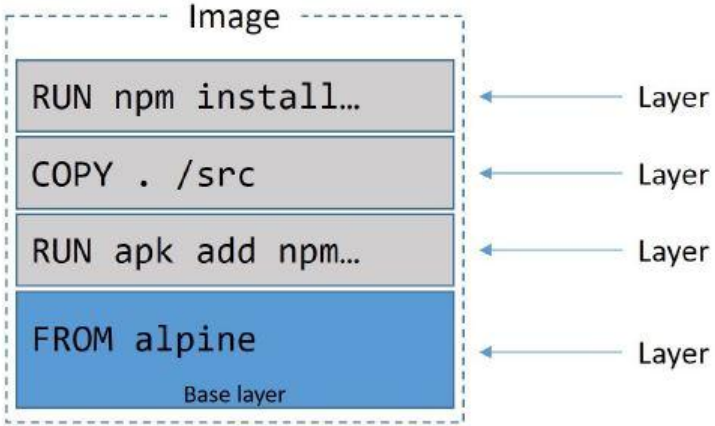
- Containers are all about apps. They're about making apps simple to build, ship, and run



Build automation

- Dockerfile is a file containing instructions for building an image

```
FROM alpine
MAINTAINER “developer@demo.it“
RUN apk update && apk add npm…
COPY . /src
RUN npm install
ENTRYPOINT [“npm run”]
% docker build --tag myapp:1.0 .
```




Dockerfile

- Dockerfile keeps building process traceable and reproducible
- .dockerignore contains a list of files to excluded from copying
- Each instruction adds a new layer. Better merging instructions to minimize layer count (usually max 42)

Dockerfile example

```
FROM fedora:latest
RUN groupadd -r -g 2000 server && useradd -g server -u 2300 user
ENV APP_ROOT="/app" VERSION="1.1"
LABEL base.name="Server"  base.version="${VERSION}"
WORKDIR $APP_ROOT
ADD . $APP_ROOT

EXPOSE 8081

ENTRYPOINT ["sh", "server.sh"]
CMD [“arg1”, “arg2”]
```

tip: *execute multiple commands (&&) in a RUN statement to avoid the creation of several layers*

Dockerfile further commands

- **LABEL** add info to image metadata

  % **docker inspect <image>**
- **RUN** executes commands
- **ENV** adds environment variables
- **WORKDIR** sets the working directory
- **EXPOSE** opens a TCP port 54321
- **COPY [“source1”, ..., “destination”]** copies N source to destination
- **ENTRYPOINT ["param1","param2", ...]** provides defaults for an executing container
- **CMD ["param1","param2", ...]** provides defaults for an executing container, if ENTRYPOINT exists CMD params will be appended to it. It can be overwritten by runtime parameters
- Dockerfile references: https://docs.docker.com/engine/reference/builder

Publish to a repository

- Register a new user on hub.docker.com (eg: mydocker)

- Build your image:

  % **docker build -t mydocker/image:1.2 . (default is Dockerfile)**

- Login to the hub:

  % **docker login**

- Upload your image:

  % **docker push mydocker/image:1.2**

- Search your image:

  % **docker search mydocker/image**

Image design

- Most of the images needs an initial script as entrypoint
- Scripts are usually written in bash or sh but every language is possible
- Preconditions and failing fast are best practices for building images
- A script should check if everything is OK before starting container's program
  - Environment variables
  - Network
  - Volumes
  - Current user
  - ...
- Drop privileges as soon as possible:
  - USER <uid>:<gid>

Best practices

- Containers should be as ephemeral as possible
- Use .dockerignore file
- Use multi-stage builds (&&)
- Sort multi-line arguments
  - RUN apt-get update && apt-get install -y \
    bzr \
    cvs \
    git \
    mercurial \
    subversion
- Avoid installing unnecessary packages
- One container – one concern
- Build cache
- https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Image distribution

- Docker provides image distribution features
- Different solutions are available
- Flexibility/complexity vs Simplicity/restriction:

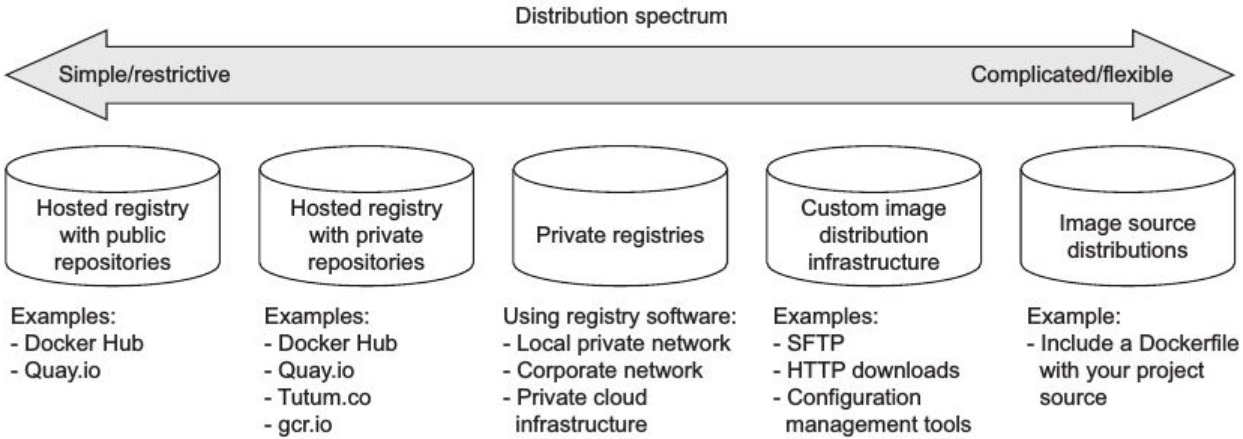
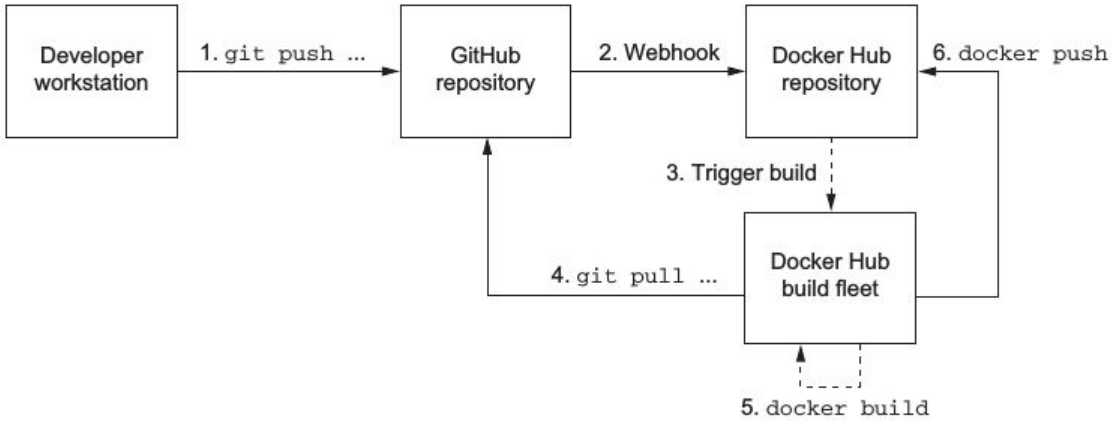


Image distribution

- Some points to evaluate as criteria for choosing the best-fit solution:
  - Cost
  - Visibility
  - Transport speed/bandwidth
  - Longevity control
  - Access control
  - Integrity
  - Confidentiality
  - Expertise

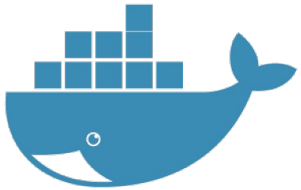
Automated builds

- A webhook is a way for a Git repository to notify an image repository that a change has been committed



Docker registries

- They are services that make repositories accessible
- A registry hosts repositories
- Some registries: Docker Hub, Quay.io, and Google Cloud Platform
- By default Docker publishes to Docker Hub



Google Cloud Platform

Registries

- Images are divided into repository
- Examples:

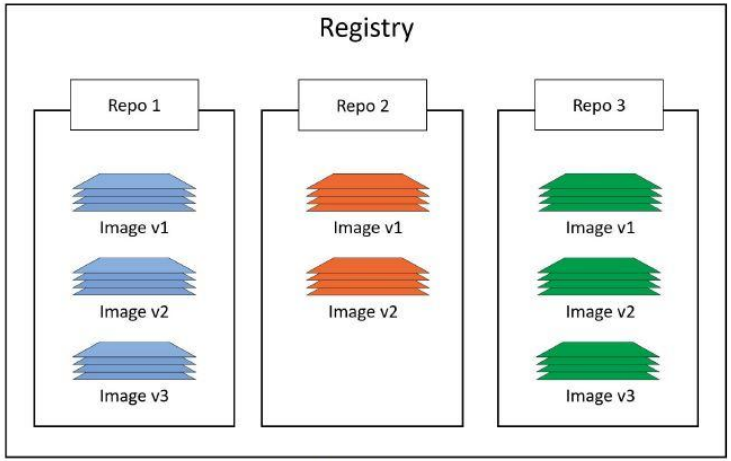
  nginx

  alpine:latest

  gcr.io/google-samples/hello-app:1.0

  camandel/whoami:green

  myregistry:5000/demo/myapp:v1.0



Manual image

- You can export/import an image as a file
- For images:

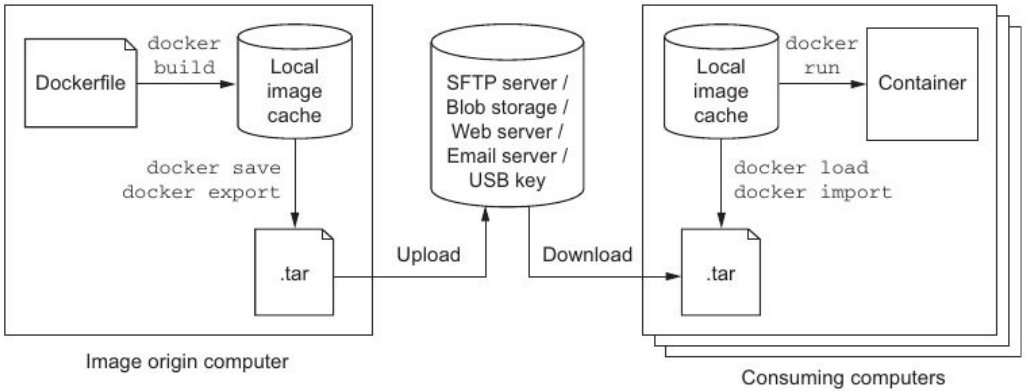
  % **docker save -o myimage.tar <image>**

  % **docker load -i myimage.tar**

- For containers:

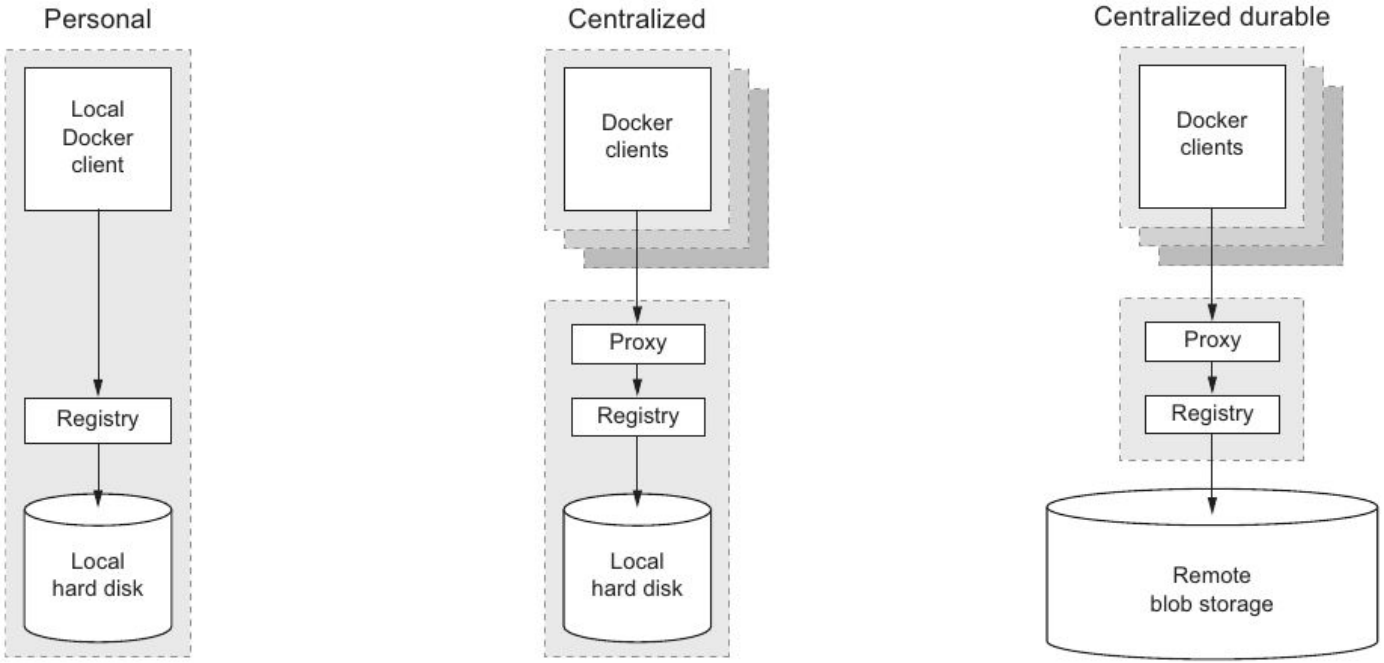
  % **docker export -o myimage.tar <container>**

  % **docker import myimage.tar**



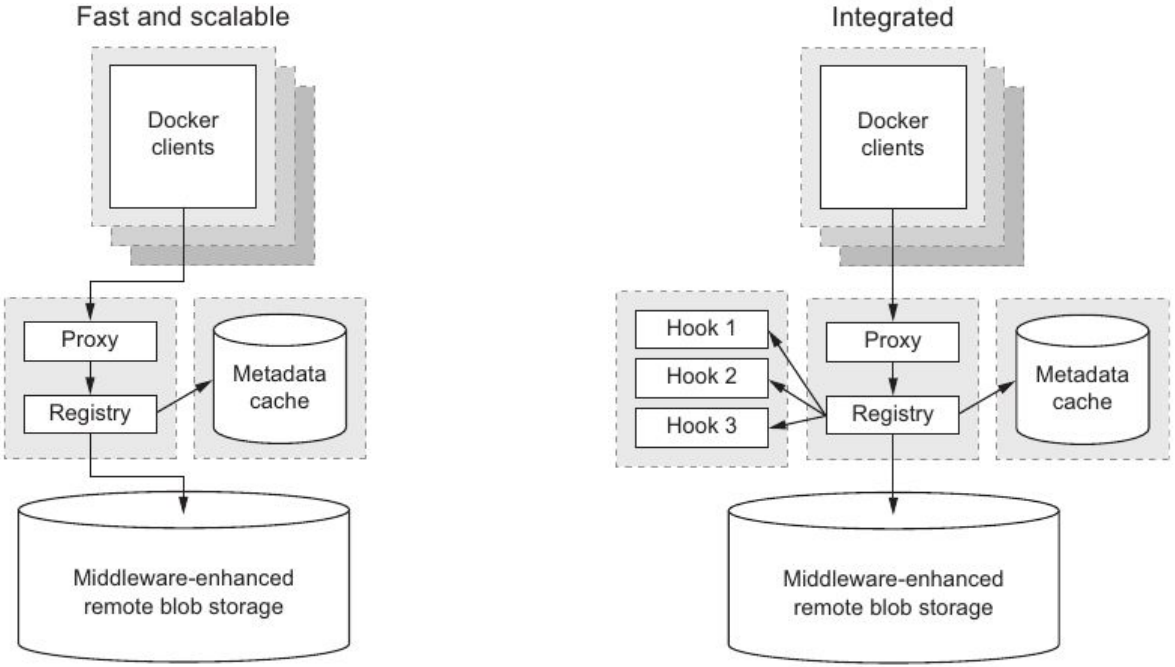
Customized registries

- Different configurations are possible



Customized registries

- Different configurations are possible



Local registry

- Just one command to start a registry container:

  % **docker run -d -p 5000:5000 -v $PWD/data:/var/lib/registry **

  --restart=always --name local-reg registry:2

- Registry is listening on localhost:5000

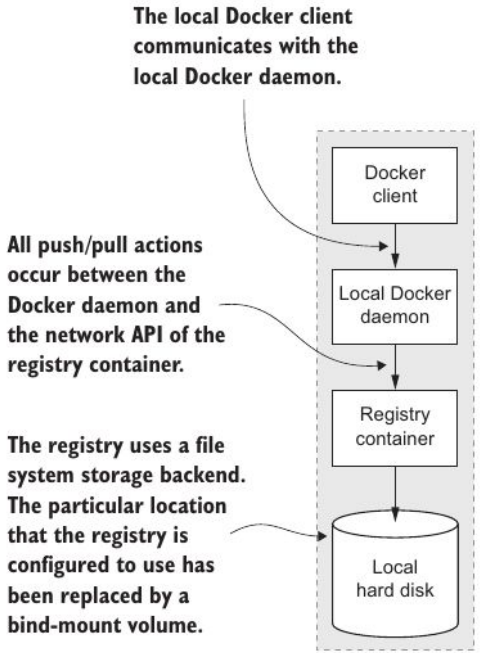
  % **docker tag alpine localhost:5000/foobar:v1**

  % **docker push localhost:5000/foobar:v1**

  % **docker rmi localhost:5000/foobar:v1**

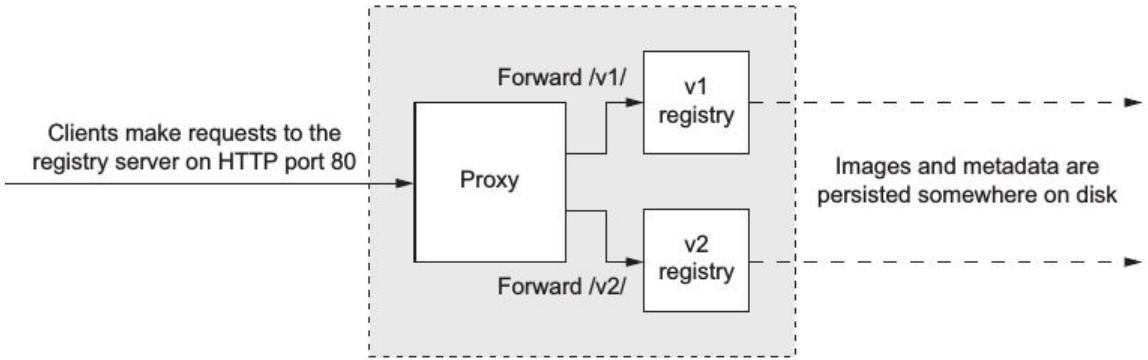
  % **docker run --rm localhost:5000/foobar:v1 id**

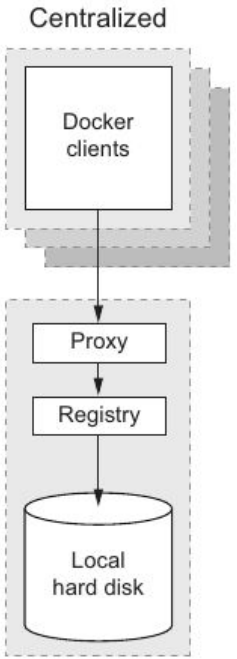
- This local registry has no security



Centralized registry

- Centralized registry is available on a network
- Multiple clients (different client versions!)
- A proxy (es. NGINX) gives flexibility
- Authentication method needed





Remote blob storage

- Blob stands for binary large object
- You can use different drivers to use a remote storage:
  - In memory
  - Filesystem
  - S3 (amazon)
  - Azure (microsoft)
  - Swift (opentalk)
  - Oss (aliyun)
  - Gcs (google)

- References:
  - https://docs.docker.com/registry/storage-drivers/

Metadata cache

- For low-latency retrieval, a cache system is needed
- Redis (redis.io) is an opensource key-value cache and data structure store
- Redis supports:
  - data structures such as strings
  - hashes
  - lists
  - sets
  - sorted sets with range queries
  - bitmaps
  - hyperloglogs
  - geospatial indexes with radius queries

```
% docker -d --name redis redis
```

Docker Compose

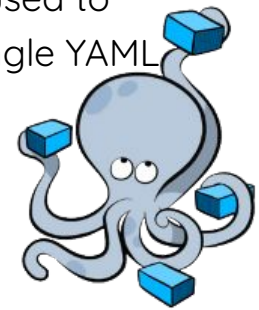
- Docker compose is a tool for defining and running multi-container Docker applications
- Services are defined in YAML files
- **docker-compose** let you:
  - Build docker images
  - Launch containerized applications
  - Launch systems of services
  - Manage the state of a service
  - Scale up/down
  - View logs
- Project link: https://github.com/docker/compose
- Complete reference at https://docs.docker.com/compose/compose-file

YAML

- **YAML** stands for "YAML Ain't Markup Language" and it is used in docker-compose.

- It's basically a human-readable structured data format. It is less complex and ungainly than XML or JSON, but provides similar capabilities. It essentially allows you to provide powerful configuration settings, without having to learn a more complex code type like CSS, JavaScript, and PHP.

- YAML is built from the ground up to be simple to use. At its core, a YAML file is used to describe data. One of the benefits of using YAML is that the information in a single YAML file can be easily translated to multiple language types.



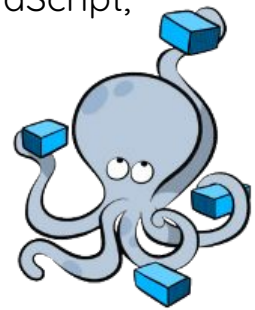
Yaml Sytntax

- YAML does not allow the use of tabs.
- YAML files should end in .yaml or .yml
- YAML is case sensitive.
- YAML does not allow the use of tabs.

Yaml Data Type

- YAML excels at working with:
  - **mappings** (hashes / dictionaries),
  - **sequences** (arrays / lists)
  - **scalars** (strings / numbers).

- While it can be used with most programming languages, it works best with languages that are built around these data structure types. This includes: PHP, Python, Perl, JavaScript, and Ruby.



Yaml Data Type

- **Scalars** are a pretty basic concept. They are the strings and numbers that make up the data on the page. A scalar could be:
  - **boolean** property, like true, integer (number) such as 5
  - **string** of text, like a sentence or the title of your website.

```
integer: 25
string: "25"
float: 25.0
boolean: true
```

Yaml Data Type

- **Sequences** is a basic list with each item in the list placed in its own line with an opening dash.
- Here is a simple sequence you might find.

```
- Cat
- Dog
- Goldfish
```

Yaml Data Type

- **Mappings** gives you the ability to list keys with values. This is useful in cases where you are assigning a name or a property to a specific element.

```
animal: pets
```

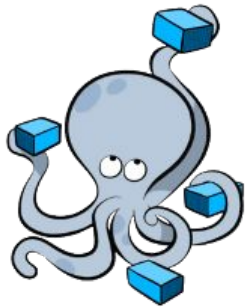
- When used in conjunction with a sequence

```
pets:
  - Cat
  - Dog
  - Goldfish
```

Docker Compose

- A docker-compose.yml looks like this:

```
version: "3.8"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
  redis:
    image: redis
volumes:
  logvolume01: {}
```



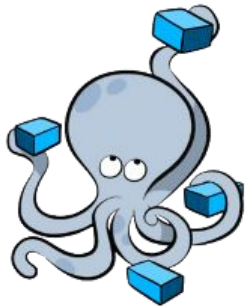
Docker Compose

- Service configuration reference
  - The Compose file is a YAML file defining
    - services, networks and volumes.
  - The default path for a Compose file is ./docker-compose.yml.
- A **service** definition contains configuration that is applied to each container started for that service
- **Network** and **volume** definitions are analogous to docker network create and docker volume create.

Docker Compose - build

- **build** can be specified either as a string containing a path to the build context:

```
version: "3.8"
services:
 webapp:
  build: ./dir
```



Docker Compose - COMMAND

- Override the default command.

```
command: bundle exec thin -p 3000
```

- The command can also be a list, in a manner similar to dockerfile:

```
command: ["bundle", "exec", "thin", "-p", "3000"]
```

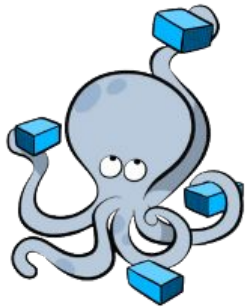
- Example

```
services:
 sshd:
   image: centos:7-sshd
   container_name: sshd
   command: ["-u", "student", "-i", "1000"]
```

Docker Compose - CONTAINER_NAME

- **container_name** Specify a custom container name, rather than a generated default name.

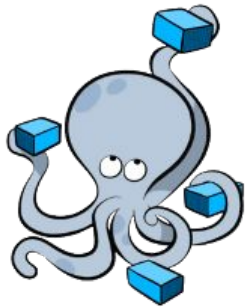
```
container_name: my-web-container
```



Docker Compose - DEPENDS_ON

- **depends_on** Express dependency between services.

```
version: "3.8"
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```



Docker Compose - ENVIRONMENT

- **environment** add environment variables into containers.
- you can use either an array or a dictionary
- any boolean values (true, false, yes, no) need to be enclosed in quotes to ensure they are not converted to True or False by the YML parser

```
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SHOW=true
  - SESSION_SECRET
```

Docker Compose - IMAGE

- **image** specify the image to start the container from. Can either be a repository/tag or a partial image ID.

```
image: redis
image: ubuntu:18.04
image: tutum/influxdb
image: example-registry.com:4000/postgresql
image: a4bc65fd
```

Docker Compose - network

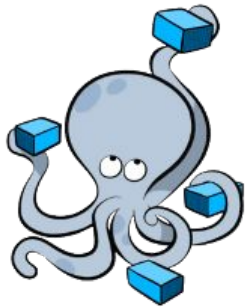
- **networks** define the setworks to join, referencing entries under the top-level networks key.

```
services:
  some-service:
    networks:
     - some-network
     - other-network
```

Docker Compose - network

- In the example below, two services are provided (web and worker), along with two networks (new and legacy).

```
version: "3.8"
services:
  web:
    image: "nginx:alpine"
    networks:
      - new
  worker:
    image: "my-worker-image:latest"
    networks:
      - legacy
      - new
networks:
  new:
  legacy:
```



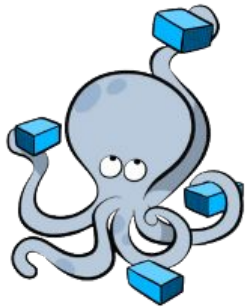
Docker Compose - ports

- ports it’s used to expose ports.

```
ports:
  - "3000"
  - "3000-3005"
  - "8000:8000"
  - "9090-9091:8080-8081"
  - "49100:22"
  - "127.0.0.1:8001:8001"
  - "127.0.0.1:5000-5010:5000-5010"
  - "6060:6060/udp"
  - "12400-12500:1240”
```

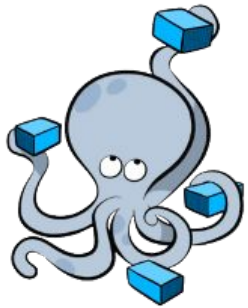
Docker Compose - volumes

- **volumes** mount host paths or named volumes, specified as sub-options to a service.
- You can mount a host path as part of a definition for a single service, and there is no need to define it in the top level volumes key.
- But, if you want to reuse a volume across multiple services, then define a named volume in the top-level volumes key.



Docker Compose - volumes

```
version: "3.8"
services:
  web:
    image: nginx:alpine
    volumes:
      - type: volume
        source: mydata
        target: /data
      - type: bind
        source: ./static
        target: /opt/app/static
  db:
    image: postgres:latest
    volumes:
      - "/var/run/postgres/postgres.sock:/var/run/postgres/postgres.sock"
      - "dbdata:/var/lib/postgresql/data"
volumes:
  mydata:
  dbdata:
```



Docker compose command line

- We are going to use CLI

% **docker compose –help**

displays all available commands

% **docker compose COMMAND –help**

displays information on a command

Docker compose command line

- Start and containers log

% **docker compose up -d**

Create and start containers

-d (--detach) Detached mode: Run containers in the background

% **docker compose -f docker-compose-file.yml up -d**

if different from docker-compose.yml/docker-compose.yaml

% **docker compose logs**

displays output from containers

Docker compose command line

- stop containers

% **docker compose stop**

Stop services

% **docker compose down**

Stop and remove containers, networks

Orchestration

- Some issue to solve...
  - Machine selection→ scheduling
  - Advertise the availability of a service → registration
  - Resolve the location of a service → service discovery

- ...developed tools
  - Docker Swarm
  - Kubernetes
  - Apache Mesos

Apache MESOS™

kubernetes

....have a good journey