# DSG-SOA-M 2024:
# - Docker -

Dr. Andreas Schönberger,
Stefan Kolb, Marcel Großmann,
Johannes Manner

Lehrstuhl für Praktische Informatik
Fakultät WIAI
Otto-Friedrich-Universität Bamberg

# Docker is...

- name of the open source project Docker
- name of the company Docker, Inc., core sponsor of the Docker project
- founded in 2013 and has attracted significant attention since then

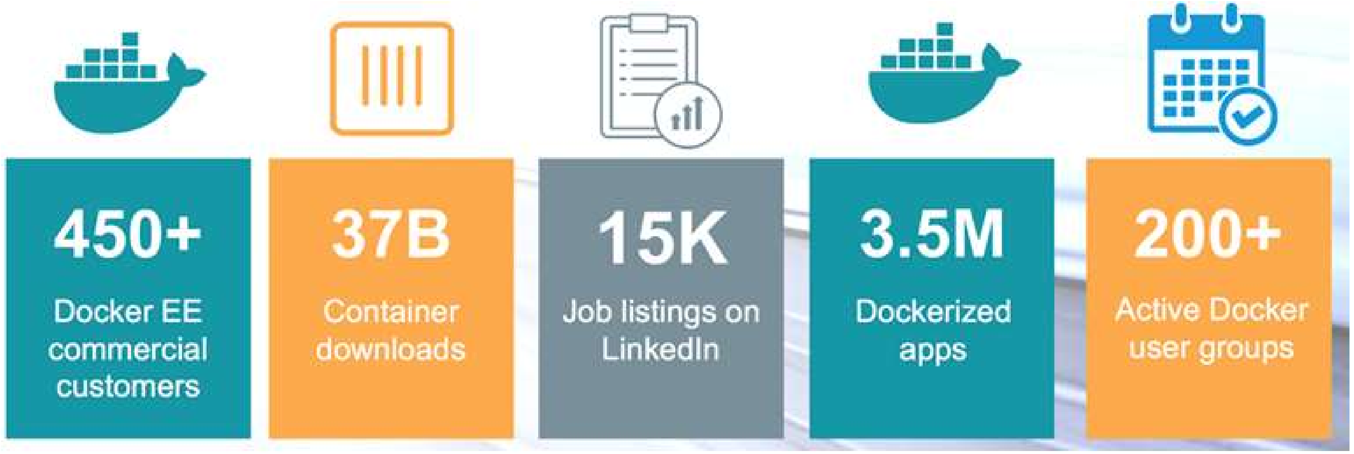


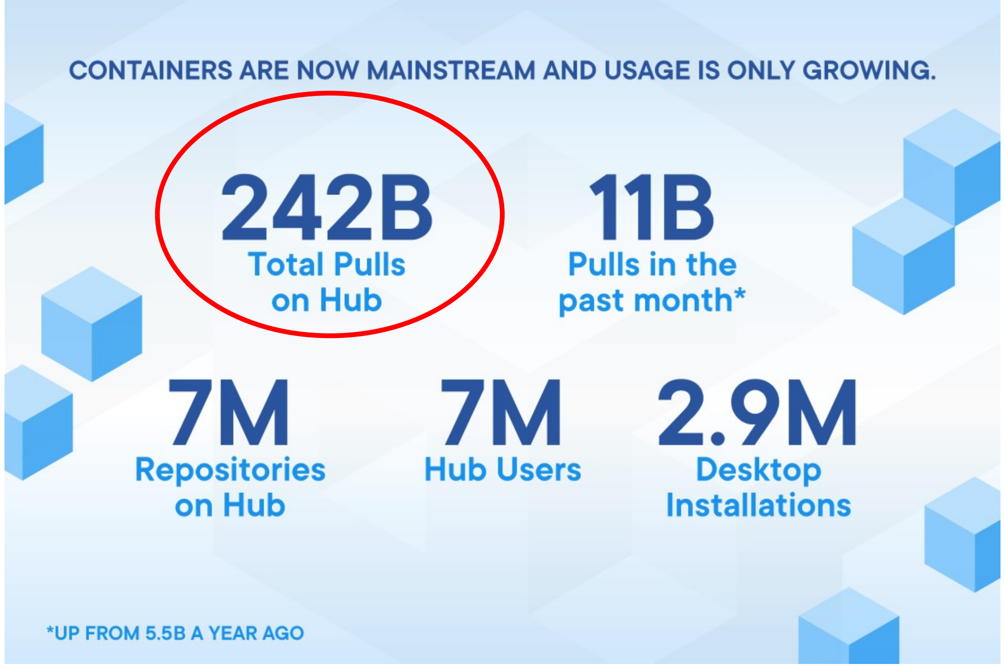
Figure from 03/2018

https://blog.docker.com/2018/03/5-years-later-docker-journey/

- a gartner report states: „By 2023, more than 70% of global organizations will be running more than two containerized applications in production, up from less than 20% in 2019."

https://www.itopstimes.com/contain/enterprise-container-strategy-its-time-to-jump-on-board/

- ❑ More statistics . . .



CONTAINERS ARE NOW MAINSTREAM AND USAGE IS ONLY GROWING.

242B Total Pulls on Hub

11B Pulls in the past month*

7M Repositories on Hub

7M Hub Users

2.9M Desktop Installations

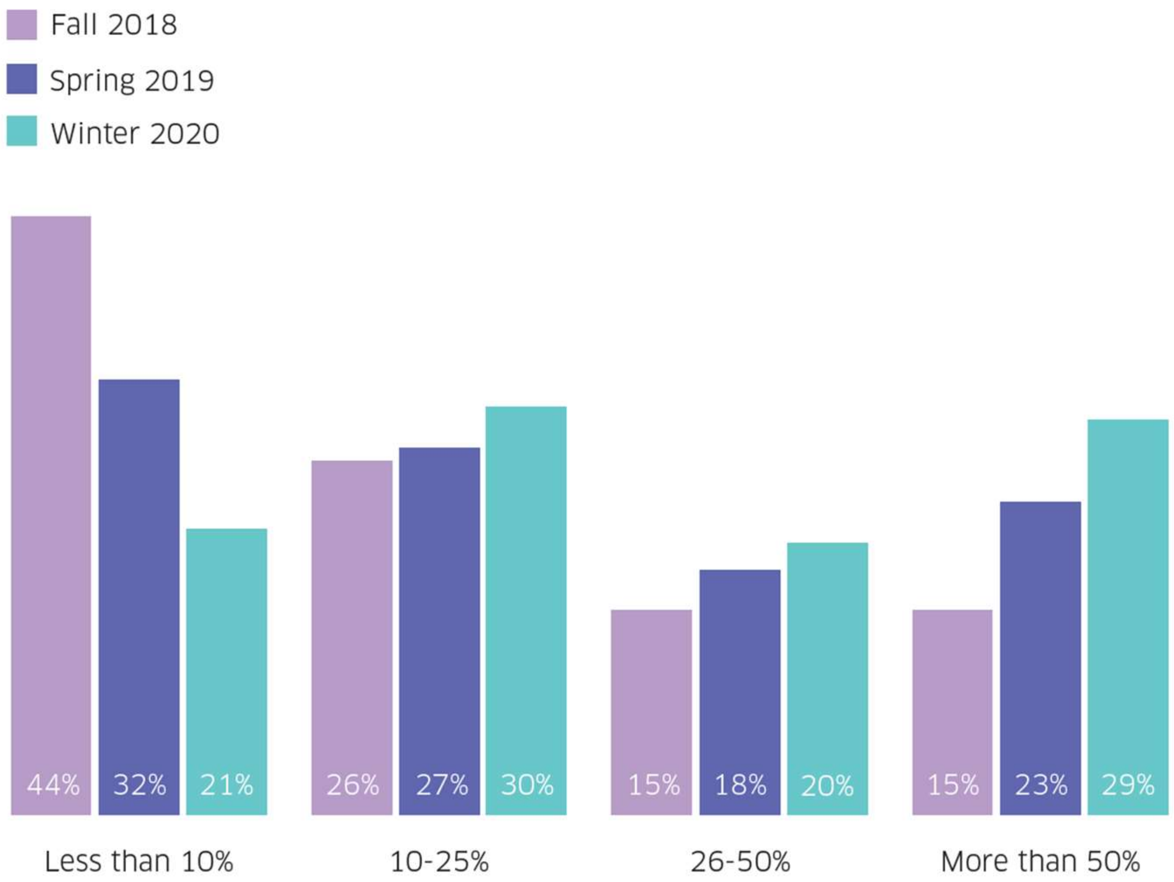
*UP FROM 5.5B A YEAR AGO

Figure from 07/2020

https://www.docker.com/blog/docker-index-dramatic-growth-in-docker-usage-affirms-the-continued-rising-power-of-developers/

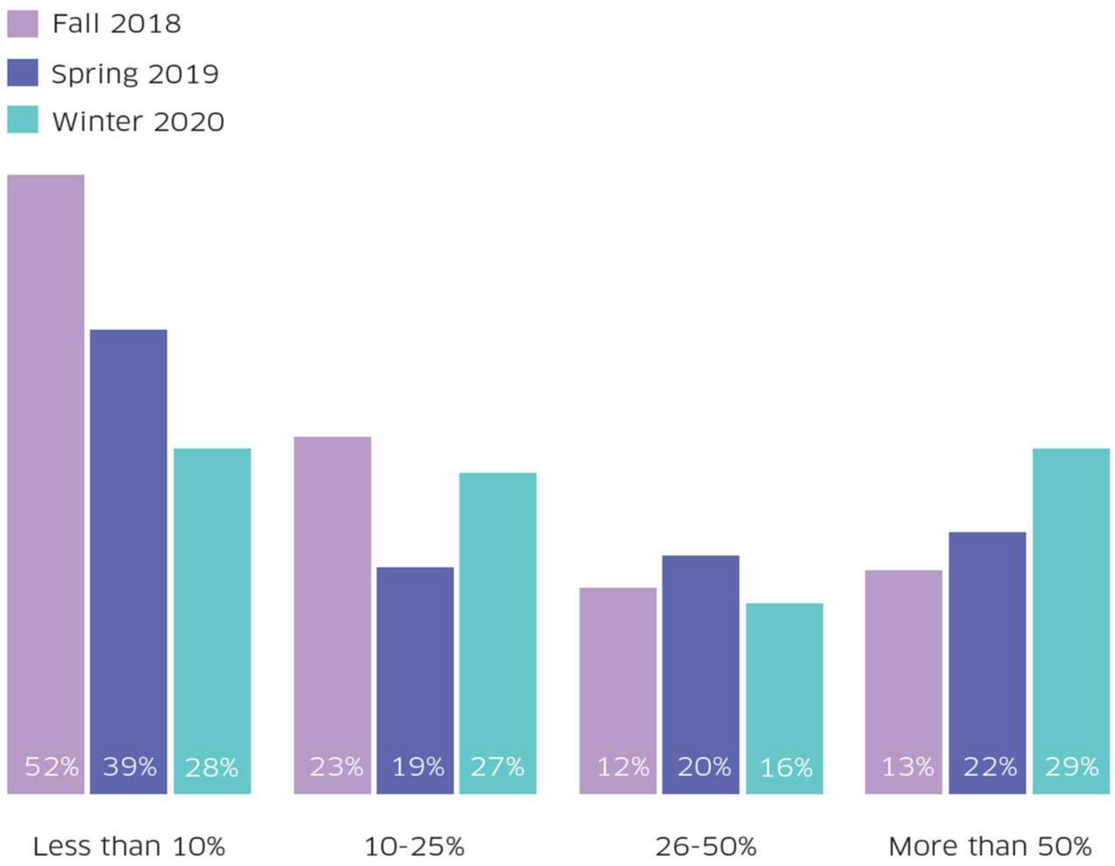
- ❑ 242B Pulls in July 2020 compared to 37B in March 2018 (654% growth and still ongoing)

# What Percentage of Apps are Containerized Today?



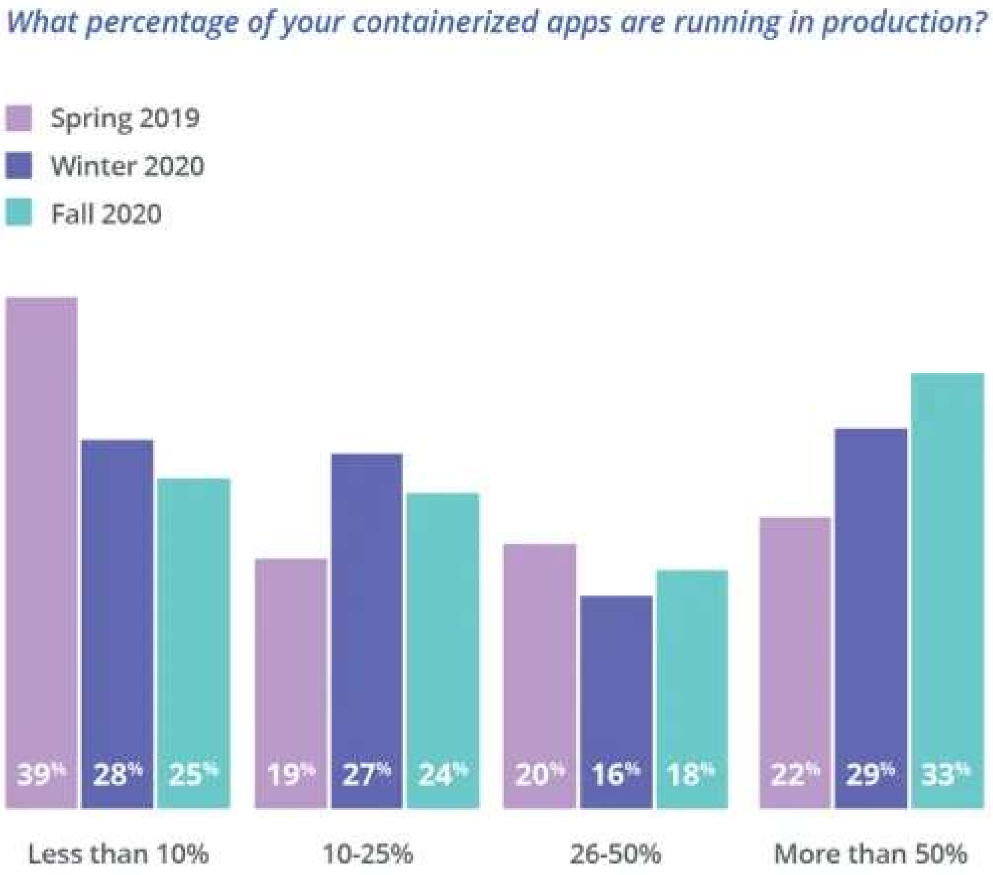
https://www.stackrox.com/post/2020/03/6-container-adoption-trends-of-2020/

# What Percentage of Containerized Apps run in Production? (2020)



https://www.stackrox.com/post/2020/03/6-container-adoption-trends-of-2020/

# What Percentage of Containerized Apps run in Production? (2021)



https://www.stackrox.com/kubernetes-adoption-security-and-market-share-for-containers/

# The Docker Hype (is it real?) – some remarks

- ❑ Larger companies are the Early Adopters
  *(datadog 2016)*
- ❑ 2/3 of companies that try Docker adopt It
  *(datadog 2016)*
- ❑ There are 460,000 Dockerized applications, a 3100% growth over two years.Over 4 billion containers have been pulled so far.
  *(Docker's CEO Ben Golub at DockerCon 2016)*
- ❑ Real Docker adoption Is Up 40% in One Year
  *(datadog 2017)*
- ❑ Larger companies are still leading adoption
  *(datadog 2017)*
- ❑ Docker now runs on 15% of the hosts we (datadog) monitor
  *(datadog 2017)*
- ❑ "Docker reaching 54 percent adoption among larger companies"
  *(RightScale 2018 State of the Cloud Report)*
- ❑ Docker now runs on more than 20% of the hosts we (datadog) monitor
  *(datadog 2018)*

**2020: Ok, let's stop asking the "Is it real?" question**

# Why all the Buzz?

- ❑ Docker uses “old” concepts, as used in, e.g., BSD Jails, Solaris Zones
- ❑ but implements them with <u>modern technologies</u> in <u>smart ways (?)</u>

## “Smart ways” of Docker

- **Make container technology very accessible**
- **Foster community for collaboration** (Open Source, Meetups, Conventions, …)
- **Offer huge ecosystem**

⇒ **Disruptive technology (?)**

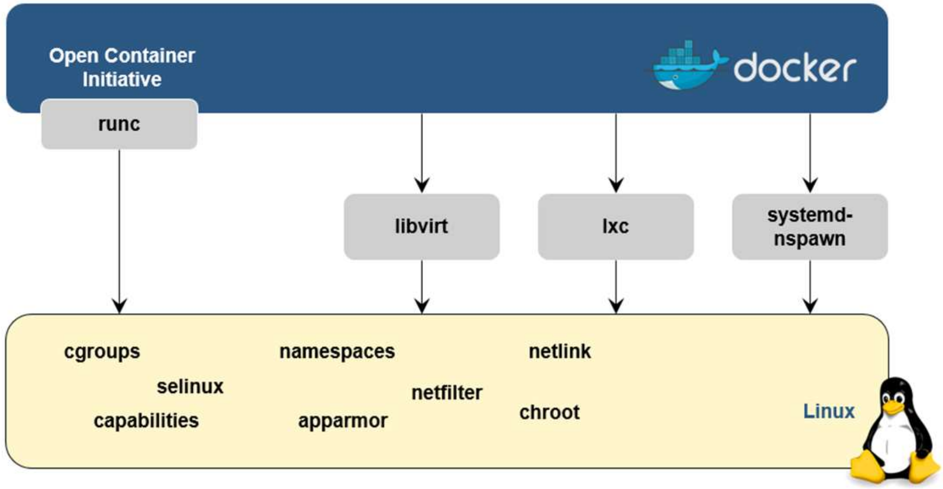
# What does Docker do?

- ❑ “Docker allows you to package an application with all of its dependencies into a **standardized** unit for software development.”
- ❑ “Docker containers wrap up a piece of software in a complete filesystem that **contains everything** it needs to run:
  code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will **always run the same**, almost regardless of the environment it is running in.”

**⇒ Build, ship and run everywhere!**

# The Underlying Technologies

- Docker makes Linux containers usable in a simple fashion
  (so Docker builds upon Linux containers, some windows integration is offered)
- As a recap: containers share the host’s kernel! (cgroups, namespaces)
- “The **Open Container Initiative** is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes.” (https://opencontainers.org/, June 2020)
- OCI’s standards
  - Image-spec
  - Runtime-spec
  - Distribution-spec
- OCI compliant runtimes
  - containerd (used by K8s)
  - lxc
  - Runc
  - …



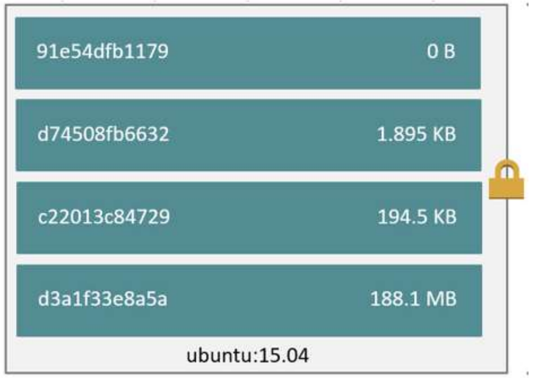
# Containers vs. Images

- **Image**: Imperative description (DOCKERFILE) of a collection of filesystem layers – read-only
- **Container**: Instance of an image – runtime adds a top writable layer
- The major difference between a container and an image is the top writable layer
- You can have many running containers of the same image
- Changes to the top writable layer like writing new files, changing existing ones are ephemeral. (When container exits, changes are gone → See Volumes to persist data)

https://docs.docker.com/storage/storagedriver/

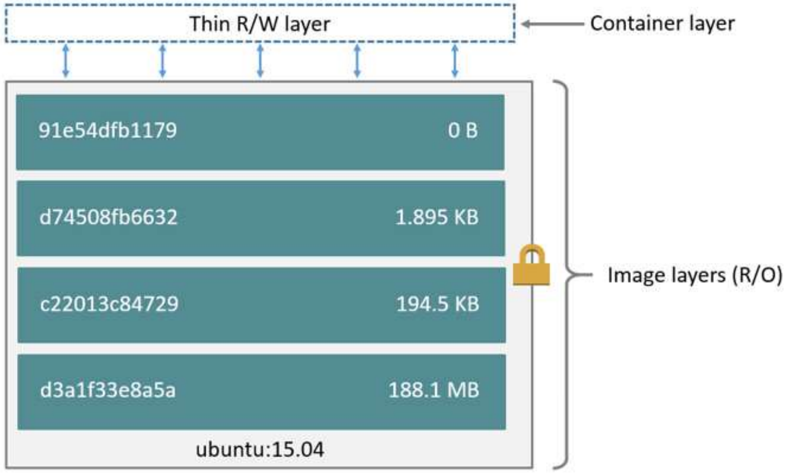
# Containers vs. Images

## Image



Container
(based on ubuntu:15.04 image)

## Container



Container
(based on ubuntu:15.04 image)

## Corresponding Dockerfile

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

- Instantiation of image
- Runtime adds a top writable layer (ephemeral)

https://docs.docker.com/storage/storagedriver/

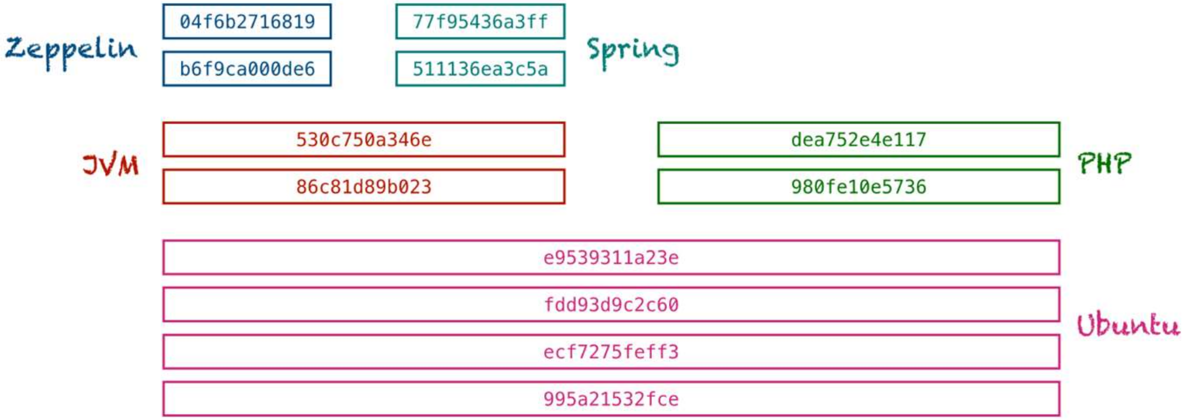
# How does Docker Achieve Isolation?

1. cgroups (known from the lecture before)
2. namespaces (known from the lecture before)
3. stackable images and copy on write
   - Docker uses a copy-on-write mechanism when deriving new images from existing ones
   - Docker only keeps track of changes between this image and our container. Docker also pulls the corresponding layer only once.
   - All changes are organized in so-called layers, where only the uppermost layer is writable.
4. virtual network bridges
   - Enables communication between hosts or even not
   - Restricts accessibility

https://blog.codecentric.de/en/2019/06/docker-demystified/

# Stackable Images and Copy-on-Write

- Layers are only stored once (Ubuntu layers are stored once and used by PHP, JVM and the other images)
- If PHP changes e.g. the top layer of Ubuntu stack (e9539..), docker copies this layer, makes changes and therefore stores it as a new layer.
- Starting a container means: putting all layers in an isolated filesystem namespace and add a top writable layer

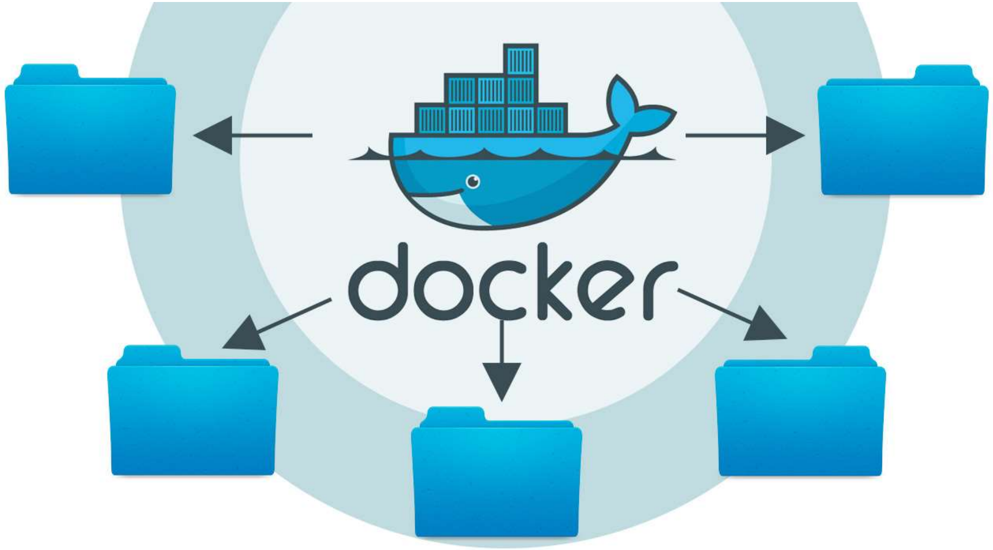


https://blog.codecentric.de/en/2019/06/docker-demystified/

# Networking – how can Isolated Containers Communicate?

- Each container gets an IP and is part of a network
- Docker offers different network drivers:
  - bridge: default driver – enables communication within the docker host between containers within the same bridge network
    - Automatically generated and called *bridge* and also used by default
    - Do not use default bridge network in production!
    - Use user-defined custom bridge networks

    !!! Default bridge does not support DNS resolution
  - host: uses the docker host’s networking directly
  - overlay: connects multiple daemons (some sort of a cluster)
  - none: disable networking for this container



# Storage in Docker

https://learning-continuous-deployment.github.io/docker/container/volumes/2015/05/22/persistent-data-with-docker/

# “External” storage – why?

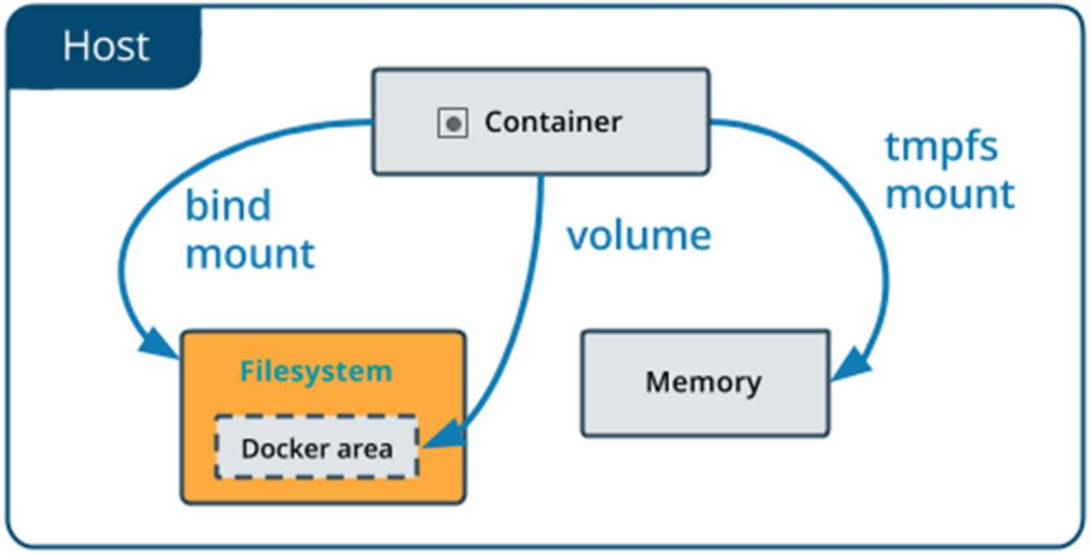
- Data written in the top writable layer only persists as long as the container is running (ephemeral data)

- Top writable layer is tightly coupled to the host, moving data is not that easy

- When writing into the top writable layer, you need a storage driver, which consumes extra resources and reduces performance

# Storage in Docker

- We discuss two types of permanent storage
  - Volumes (managed by docker host – special location within host's filesystem, preferred option)
  - Bind mounts (arbitrary folder on docker host, mounted in a container)
    Can be security critical when sharing config or sensitive data.

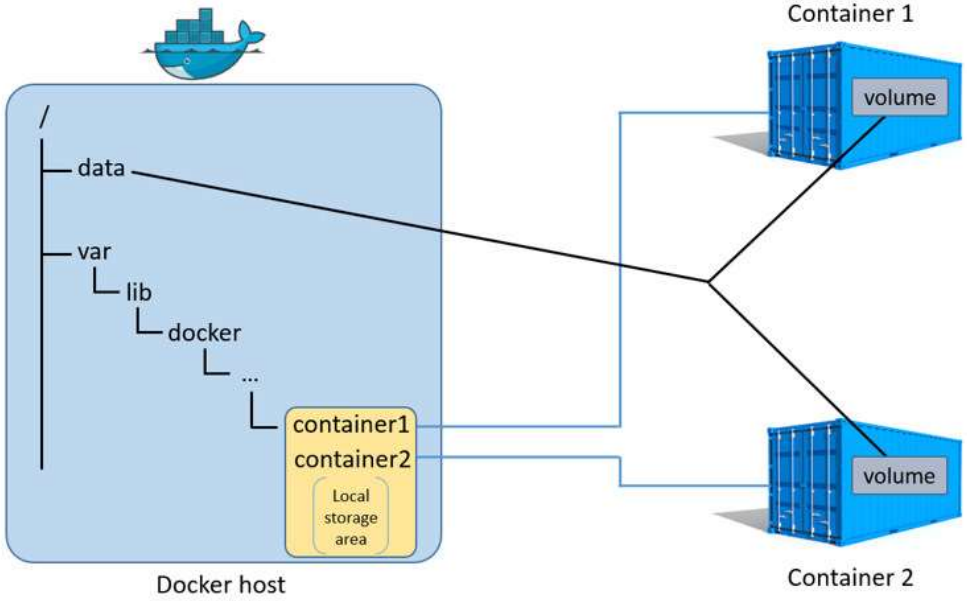


https://docs.docker.com/storage/

# Data Volumes (preferred way)

- Data volumes are managed by docker engine
- A data volume is a directory or file in the **host's filesystem** that is mounted directly into a container
- When a container is deleted, any data written to the container that is not stored in a *data volume* is deleted along with the container
- You can mount any number of data volumes into a container
- Multiple containers can also share one or more data volumes

# Data Volumes



- Each container has it's own folder on the system controlled by docker (/var/lib/docker/containers/*)
- Volumes are stored under a separate folder (/var/lib/docker/volume)

# Bind Mounts

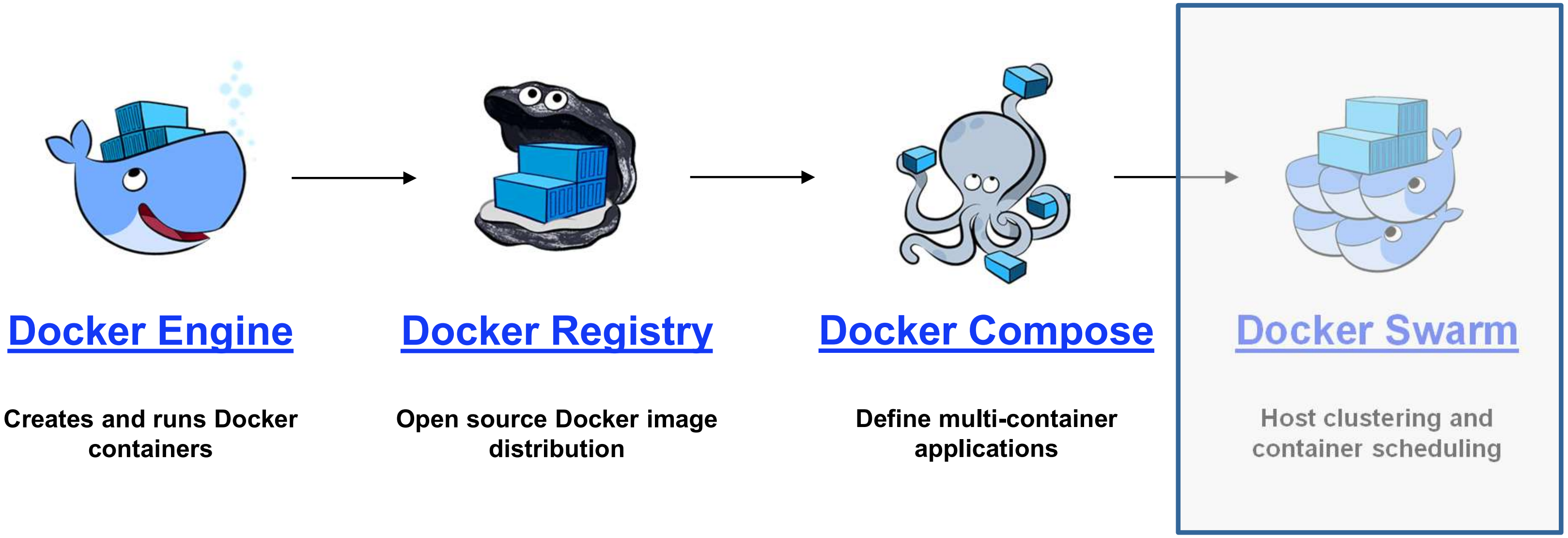
- Managed by the user (!!)
- Read-only access might be a good choice in many cases
- Sharing configuration files might be beneficial (they normally do not reside within docker's filesystem)

- **BUT**: Whenever possible –

  Use Volumes instead

https://blogs.sap.com/2020/03/10/understanding-containers-part-05-shared-files-between-the-host-and-containers/

# (Part of) The Docker Stack



**Docker Engine**

Creates and runs Docker containers

**Docker Registry**

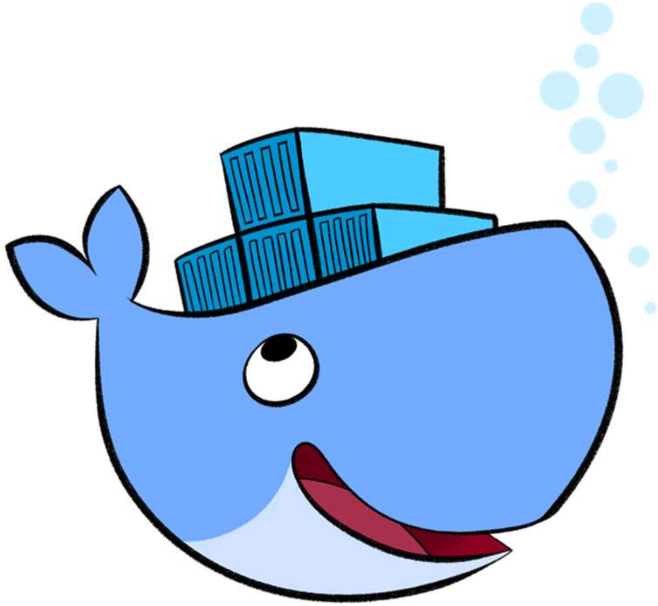
Open source Docker image distribution

**Docker Compose**

Define multi-container applications

**Docker Swarm**

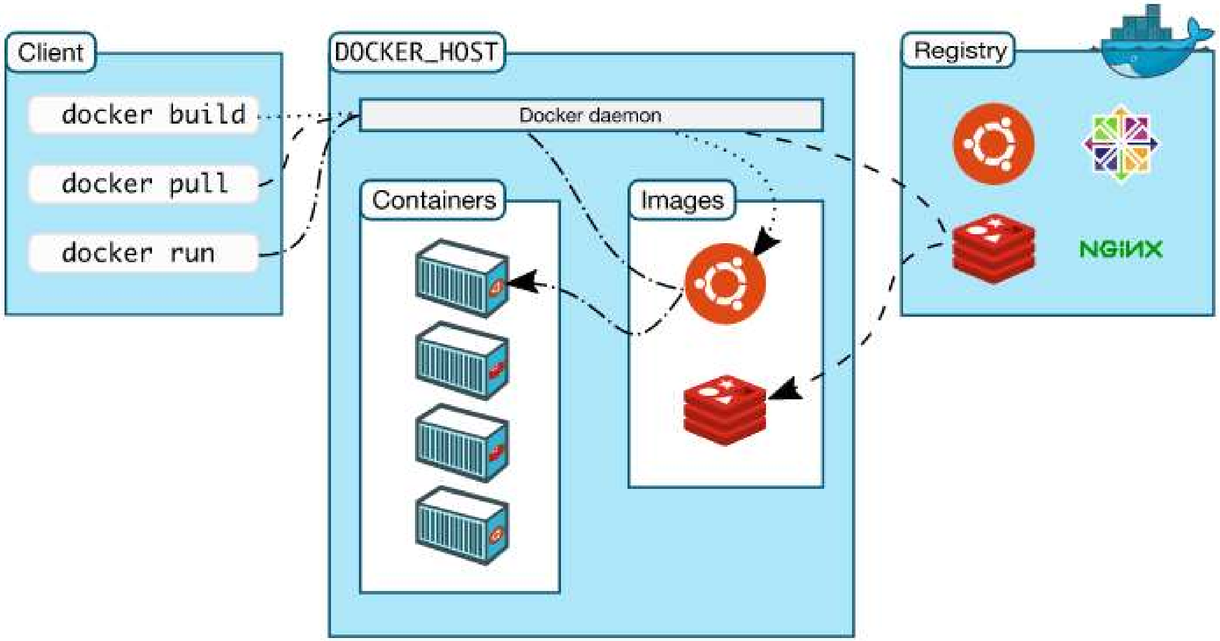
Host clustering and container scheduling



# Docker Engine

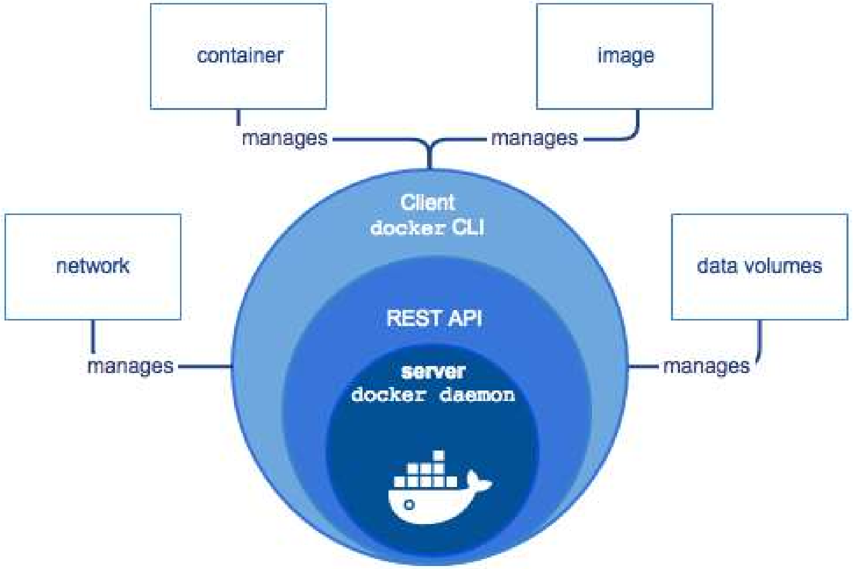
# Overall Docker Architecture

- Daemon: daemon of the server process to manage containers
- Client: user client to (remotely) control the daemon
- Registry: platform for sharing and managing images



https://docs.docker.com/engine/understanding-docker/

# Engine – Overview

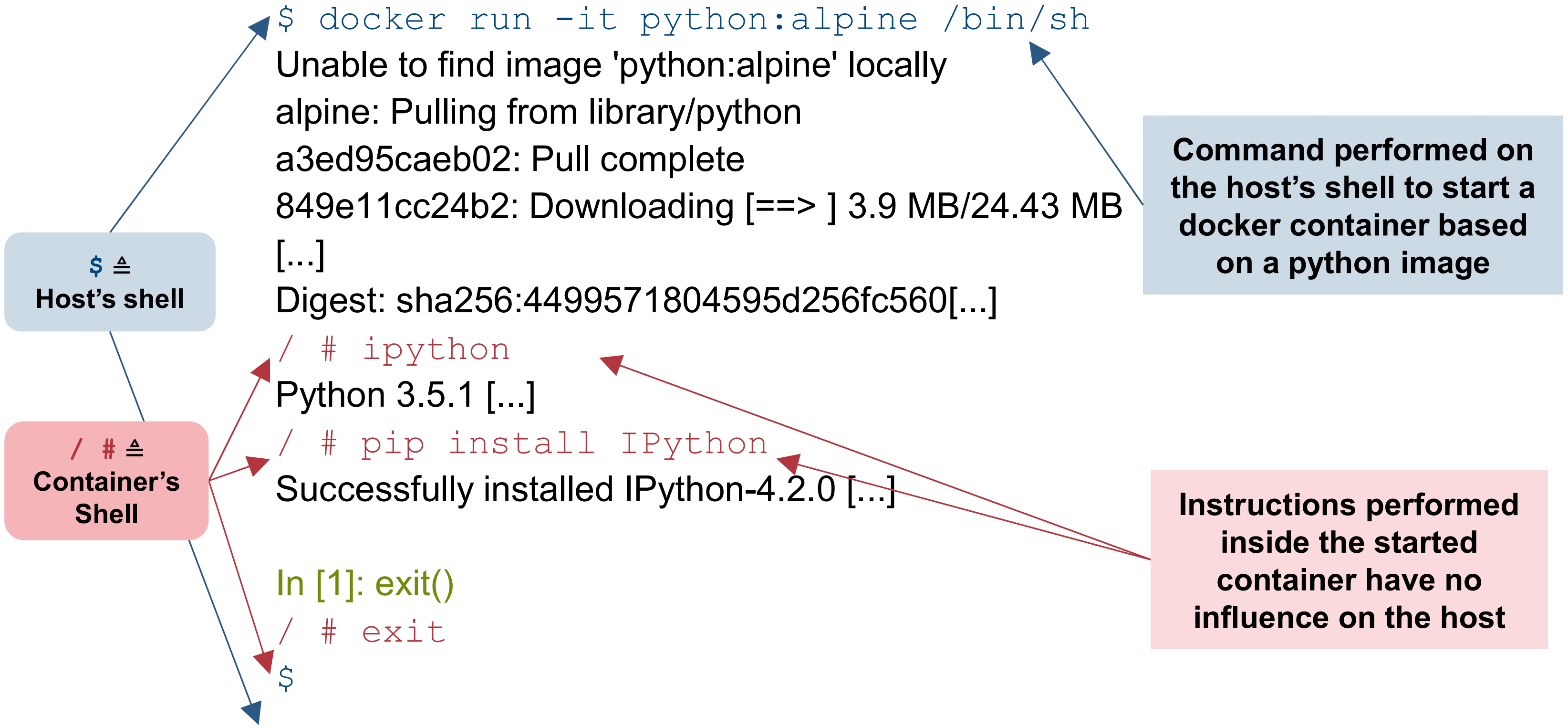


Consists of three parts:

- Server: Daemon of the server process to manage the containers
- Client: Client to (remotely) control the daemon
- REST API

https://docs.docker.com/get-started/overview/

# Engine – Example



```
$ docker run -it python:alpine /bin/sh
Unable to find image 'python:alpine' locally
alpine: Pulling from library/python
a3ed95caeb02: Pull complete
849e11cc24b2: Downloading [==> ] 3.9 MB/24.43 MB
[...]
Digest: sha256:44995718045 95d256fc560[...]
/ # ipython
Python 3.5.1 [...]
/ # pip install IPython
Successfully installed IPython-4.2.0 [...]

In [1]: exit()
/ # exit
$
```

$ ≙ Host's shell

/ # ≙ Container's Shell

Command performed on the host's shell to start a docker container based on a python image

Instructions performed inside the started container have no influence on the host

# Engine – Example

- ❑ **What happened here?**
  - We created a *container* with its own:
    - filesystem (based initially on a `python` image)
    - network stack
    - process space
  - We started a shell process (no `init`, no `systemd`, no problem)
  - We installed IPython with `pip`

- ❑ **What did not happen here?**
  - We did not make a full copy of the `python` image
  - We did not modify the `python` image itself
  - We did not affect any other container (currently using this image or any other image)
- ❑ **List of comands and their explanation: Docker CLI**

# Engine – Dockerizing applications

❑ Running an application based on an image needs a single command:

```
$  docker run <IMAGE>
```

❑ Let‘s echo hello world inside a container

```
$  docker run ubuntu /bin/echo 'Hello world‘
   Hello world
```

❑ Run an interactive container

```
$  docker run -t -i ubuntu /bin/bash
   root@af8bae53bdd3:/#
   root@af8bae53bdd3:/# ls
   bin boot dev etc home lib lib64 media mnt opt proc
   root run sbin srv sys tmp usr var
   root@af8bae53bdd3:/# exit
```

# Engine – Working with images

- ❑ Managing local images

```
$  images     List images
   rmi        Remove one or more images
   tag        Tag an image into a repository
   inspect    Return low-level information on an image
```

- ❑ Working with an image registry

```
$  pull       Pull an image from a registry
   push       Push an image to a registry
   search     Search the Docker Hub for images
```

# Engine – Working with containers

- ❑ Managing containers

```
$  ps         List containers
   run        Run a command in a new container
   start      Start one or more stopped containers
   stop       Stop a running container
   commit     Create a new image from a container
   rm         Remove one or more containers
```

- ❑ Inspecting/debugging containers

```
$  diff       Inspect changes on a container's filesystem
   inspect    Return low-level information on a container
   logs       Fetch the logs of a container
   stats      Display a resource usage statistics
   top        Display running processes of a container
```

# Engine – Creating a new image

- ❑ Add something to an existing image **`training/sinatra`**

```
$  docker run -t -i training/sinatra /bin/bash
   root@0b2616b0e5a8:/#
   root@0b2616b0e5a8:/# gem install json
   root@0b2616b0e5a8:/#exit
```

- ❑ Save the container to a new image

```
$  docker commit -m "Added json gem" -a "Kate Smith" \
   0b2616b0e5a8 ouruser/sinatra:v2
   4f177bd27a9ff0f6dc2a830403925b5360bfe0b93d476f7fc3231
   110e7f71b1
```

- ❑ Use your new image

```
$  docker run -t -i ouruser/sinatra:v2 /bin/bash
   root@78e82f680994:/#
```

# Engine – Creating a new image

- ❑ Using the **`docker commit`** command is a pretty simple way of extending an image
- ❑ but it’s a bit cumbersome and it’s not easy to share a development process for images amongst a team
- ❑ Repeatable way to create images?!

→ **Dockerfiles**

# Dockerfile

- ❑ Skeleton for an Image
- ❑ Contains all neccessary instructions to generate an image
- ❑ Makes image creation reproducable
- ❑ Instructions, e.g., **`RUN`** are:
  - single line statements and contain a key word
  - not case sensitive
  - always create a new layer

```
# This is a comment
FROM ubuntu:14.04
MAINTAINER Kate Smith <ksmith@example.com>
RUN apt-get update && apt-get install -y
ruby ruby-dev
RUN gem install sinatra
```

# Dockerfile – Example

Extend base image

Run a command

*Here: install dependencies from the package repository*

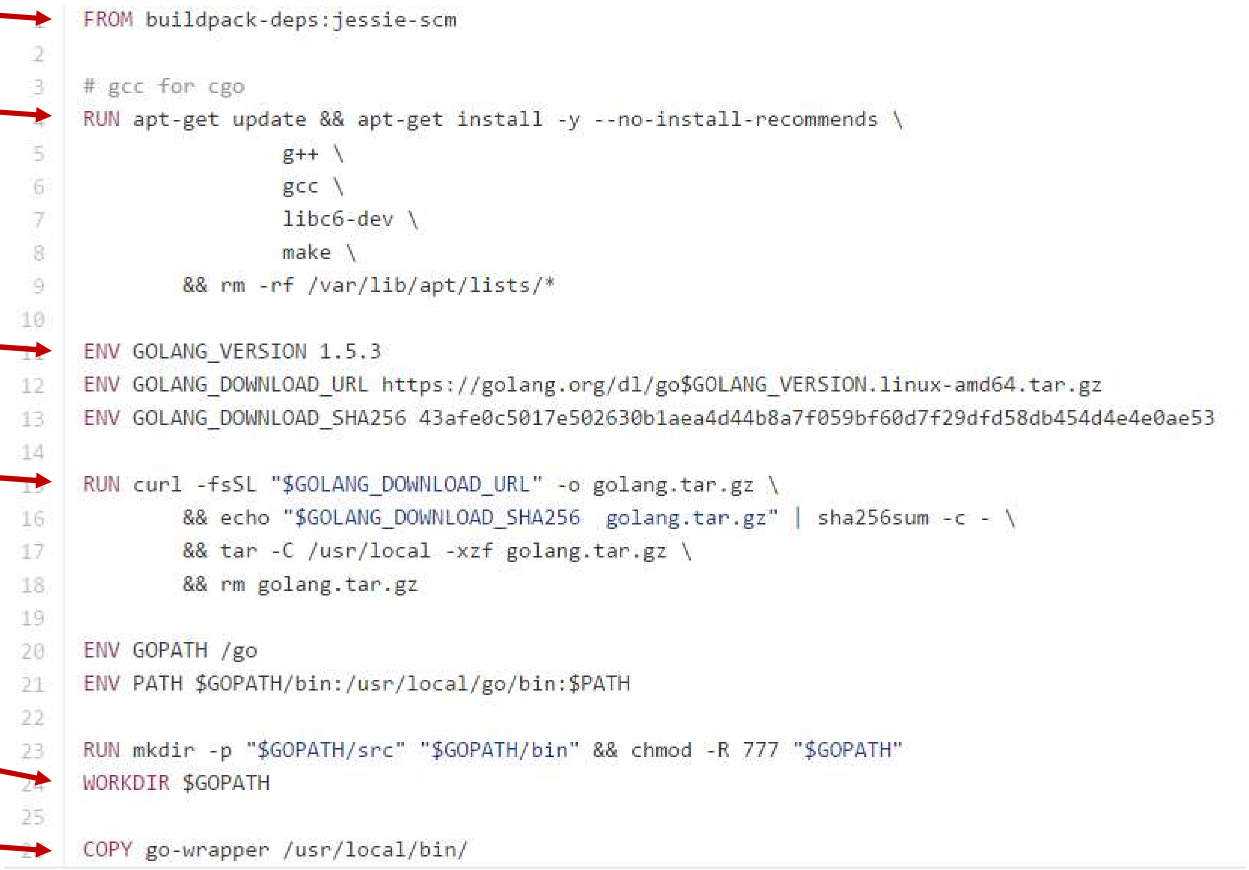
Set env variables

Run a command

*Here: download Go runtime*

Set default workdir

Copy from host to image

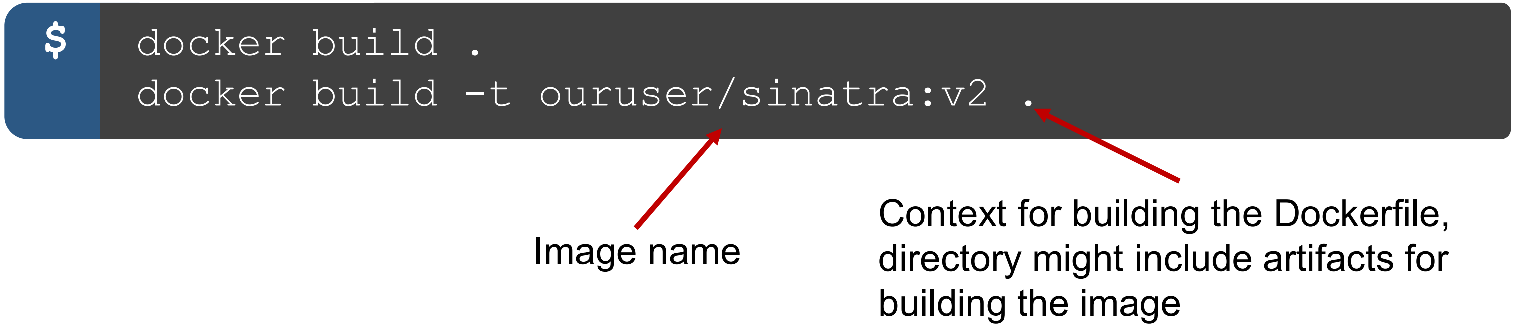


```
FROM buildpack-deps:jessie-scm

# gcc for cgo
RUN apt-get update && apt-get install -y --no-install-recommends \
                g++ \
                gcc \
                libc6-dev \
                make \
        && rm -rf /var/lib/apt/lists/*

ENV GOLANG_VERSION 1.5.3
ENV GOLANG_DOWNLOAD_URL https://golang.org/dl/go$GOLANG_VERSION.linux-amd64.tar.gz
ENV GOLANG_DOWNLOAD_SHA256 43afe0c5017e502630b1aea4d44b8a7f059bf60d7f29dfd58db454d4e4e0ae53

RUN curl -fsSL "$GOLANG_DOWNLOAD_URL" -o golang.tar.gz \
        && echo "$GOLANG_DOWNLOAD_SHA256  golang.tar.gz" | sha256sum -c - \
        && tar -C /usr/local -xzf golang.tar.gz \
        && rm golang.tar.gz

ENV GOPATH /go
ENV PATH $GOPATH/bin:/usr/local/go/bin:$PATH

RUN mkdir -p "$GOPATH/src" "$GOPATH/bin" && chmod -R 777 "$GOPATH"
WORKDIR $GOPATH

COPY go-wrapper /usr/local/bin/
```

# Dockerfile – Workflow

1. Define image in a **`Dockerfile`**

2. Use **`docker build`** to create a new image based on the Dockerfile



```
$  docker build .
   docker build -t ouruser/sinatra:v2 .
```

Image name

Context for building the Dockerfile, directory might include artifacts for building the image

3. Create a new container from your image via **`docker run`**

# (Important) Dockerfile instructions

$

**FROM** -- sets the Base Image for subsequent instructions (mandatory).

**RUN** -- will execute any commands in a new layer on top of the current image and commit the results.

**CMD** -- The main purpose of a CMD is to provide defaults for an executing container (only one allowed).

**EXPOSE** -- informs Docker that the container listens on the specified network ports at runtime.

**ENV** -- sets an environment variable

**ADD or COPY** -- copies files, directories or remote file URLs to the filesystem of the container

# (Important) Dockerfile instructions

$

**ENTRYPOINT** -- allows you to configure a container that will run as an executable – first command which is executed. Also see CMD.

**VOLUME** -- creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

**WORKDIR** -- sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions.

**ONBUILD** -- adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build, e.g., an application runtime container.

# Best practices for images (1/2)

- **Containers should be ephemeral**
  - can be stopped and destroyed and a new one built and put in place with an absolute minimum of set-up and configuration
- **Avoid installing unnecessary packages**
  - to reduce complexity, dependencies, file sizes, and build times, you should avoid installing extra or unnecessary packages
- **Domain-driven assignment of processes to containers**
  - Decoupling applications into multiple containers makes it much easier to scale horizontally and reuse containers. If that service depends on another service, make use of container linking
- **Minimize the number of layers**
  - find the balance between readability of the Dockerfile and minimizing the number of layers it uses.

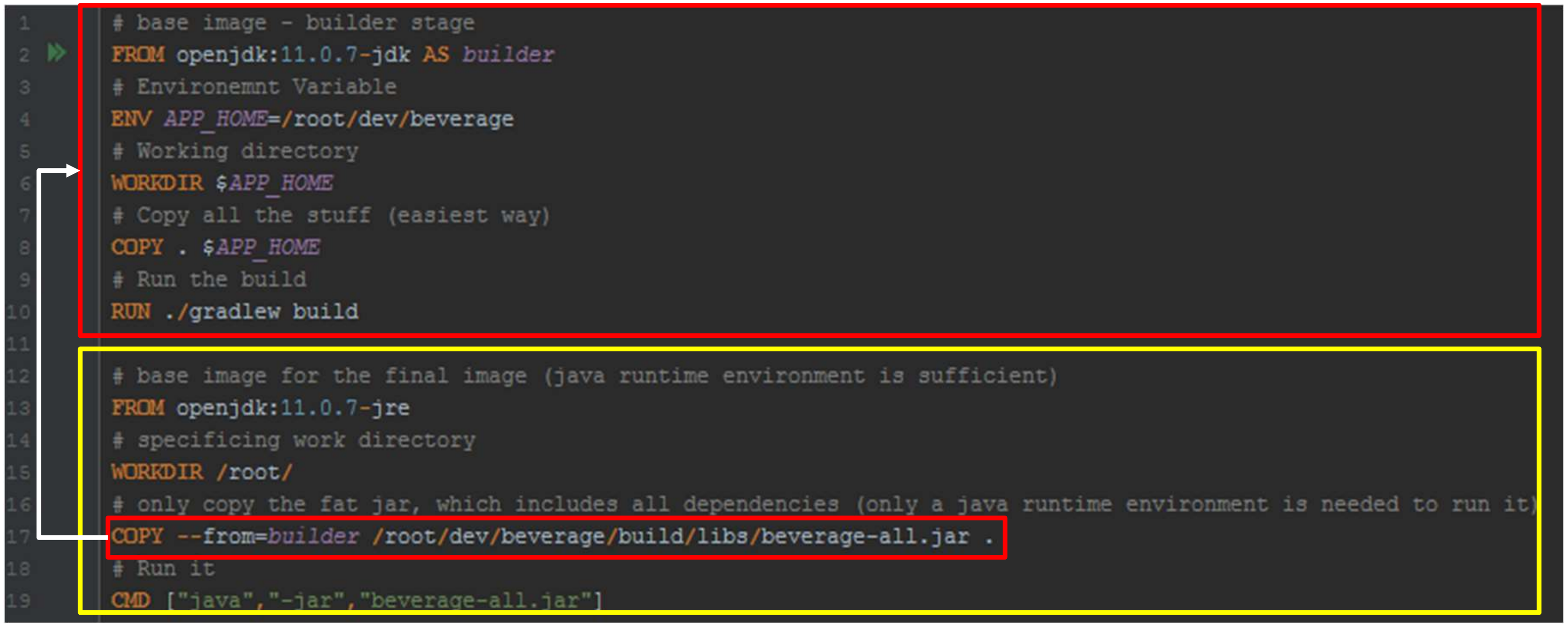
# Best practices for images (2/2)

- **Keep your images as small as possible** (faster pull times, loading into memory, starting containers)
  - Start with the right base image (only the stuff you need)
  - Use multi stage builds (next slide)
- **Share layers as much as possible**
  - If you have a common basis, define your own base images
  - docker only pulls the layers once – they are cached afterwards
- **Tag your images with meaningful tags**
- **Do not store application data in the top writable layer of your container** (you know this data is ephemeral)

# Best practices for Dockerfiles

- Each instruction creates one layer
  - Define the minimum of needed layers
- Use a .dockerignore file (works as .gitignore)
  - Ignores all specified files within the working directory
- Multi Stage Builds
  - Multiple bases
  - Select, copy, alter files from one stage to others
  - Name your build stages
  - You can use external images as stages
  - Only the layers of the last stage are included in the image

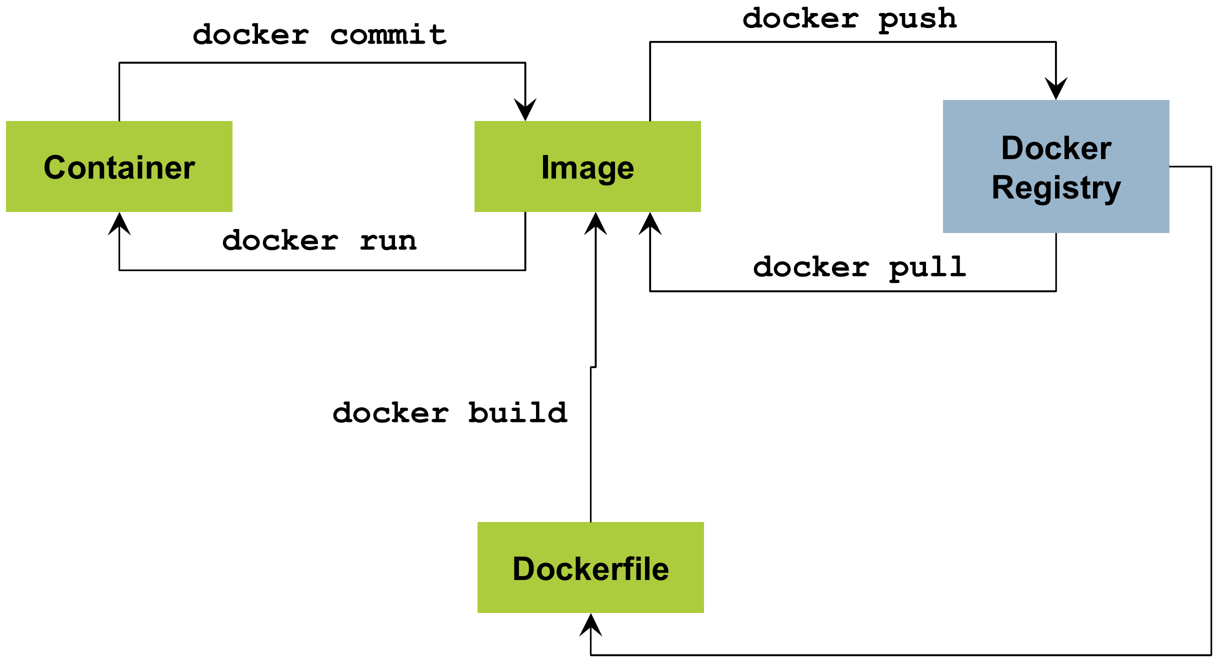
# Multi-Stage Builds



```
# base image - builder stage
FROM openjdk:11.0.7-jdk AS builder
# Environemnt Variable
ENV APP_HOME=/root/dev/beverage
# Working directory
WORKDIR $APP_HOME
# Copy all the stuff (easiest way)
COPY . $APP_HOME
# Run the build
RUN ./gradlew build

# base image for the final image (java runtime environment is sufficient)
FROM openjdk:11.0.7-jre
# specificing work directory
WORKDIR /root/
# only copy the fat jar, which includes all dependencies (only a java runtime environment is needed to run it)
COPY --from=builder /root/dev/beverage/build/libs/beverage-all.jar .
# Run it
CMD ["java","-jar","beverage-all.jar"]
```

- ❑ Builder stage (not included in the image – only the last stage is included – beginning at last FROM statement)
- ❑ “Image stage” – All commands here result in a single layer
- ❑ Access to the builder stage and copying of the relevant file

# Docker Process (so far)



Imported by Dockerfile's FROM statement

# Docker Registry

# Registry

- ❑ Often, containers are based on or reuse existing images
- ❑ The Registry is a stateless, highly scalable server side application that stores and distributes Docker images
- ❑ Stores the layers and the description of how they make up an image
- ❑ Can be hosted locally to own the images pipeline
- ❑ Most users will be satisfied with Docker‘s public instance

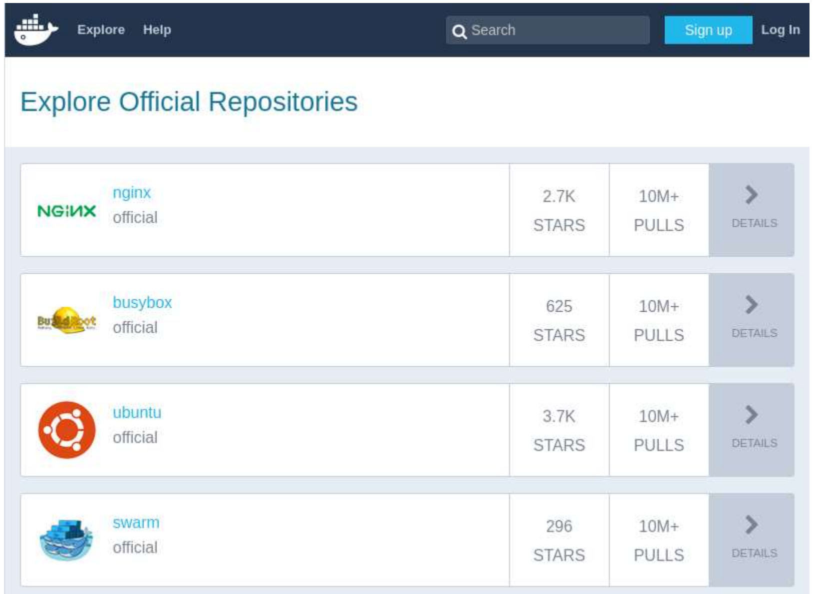
→ **Public central registry**: **Docker Hub**

# Docker Hub

- Centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline
  - **Image Repositories**: Find, manage, and push and pull images from community, official, and private image libraries
  - **Automated Builds**: Automatically create new images when you make changes to a source repository
  - **Organizations**: Create work groups to manage user access to image repositories

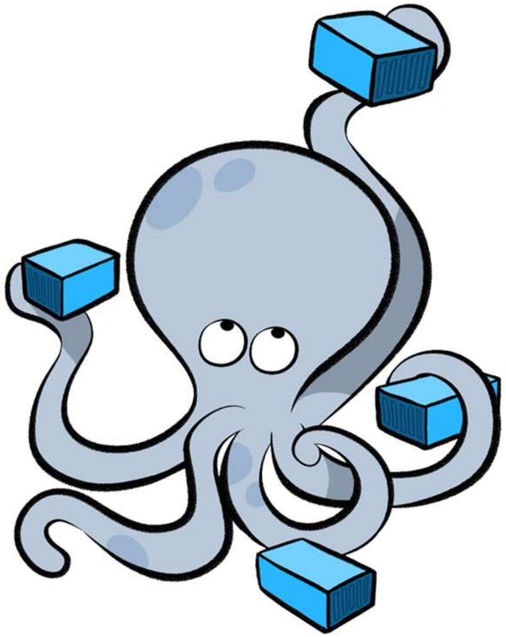
# Docker Hub

- Docker Hub hosts public and private Docker images
- Docker Hub will be used for looking up missing local images or when using **`docker pull`**
- Includes official and community-maintained images



Explore Official Repositories

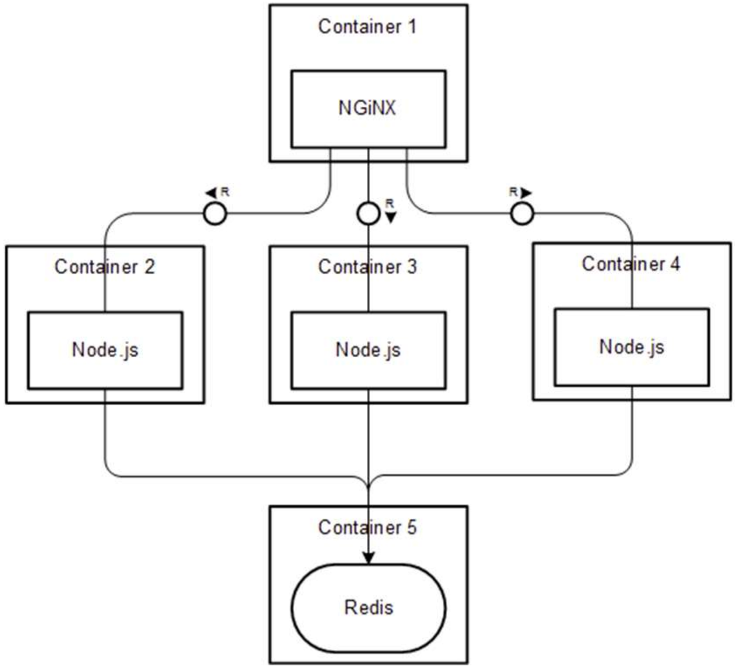
| | | | |
|---|---|---|---|
| nginx<br>official | 2.7K<br>STARS | 10M+<br>PULLS | DETAILS |
| busybox<br>official | 625<br>STARS | 10M+<br>PULLS | DETAILS |
| ubuntu<br>official | 3.7K<br>STARS | 10M+<br>PULLS | DETAILS |
| swarm<br>official | 296<br>STARS | 10M+<br>PULLS | DETAILS |



# Docker Compose

# Compose – The whys and wherefores

- ❑ Typically, an application is not one service but an orchestration of multiple smaller, isolated service units



http://anandmanisankar.com/posts/docker-container-nginx-node-redis-example/

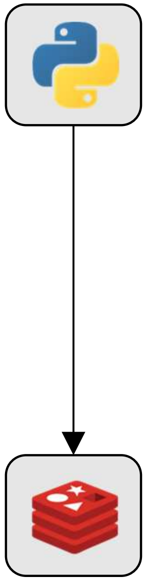
# Compose – The whys and wherefores

***Multi-container apps are a hassle!***

1. Build images from Dockerfiles
2. Pull existing images from the Docker Hub
3. Create Containers
4. Start/Stop Containers
5. Stream container logs

# Compose – The whys and wherefores

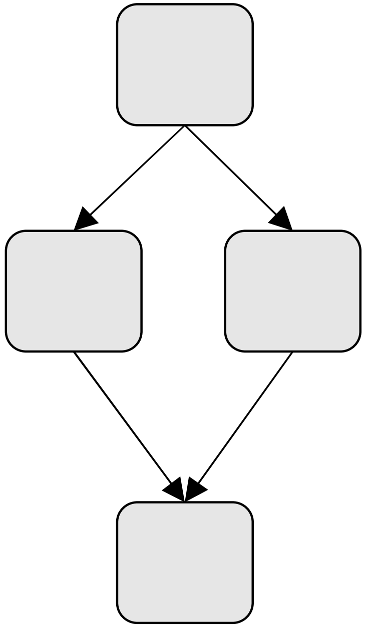
***Multi-container apps are a hassle!***

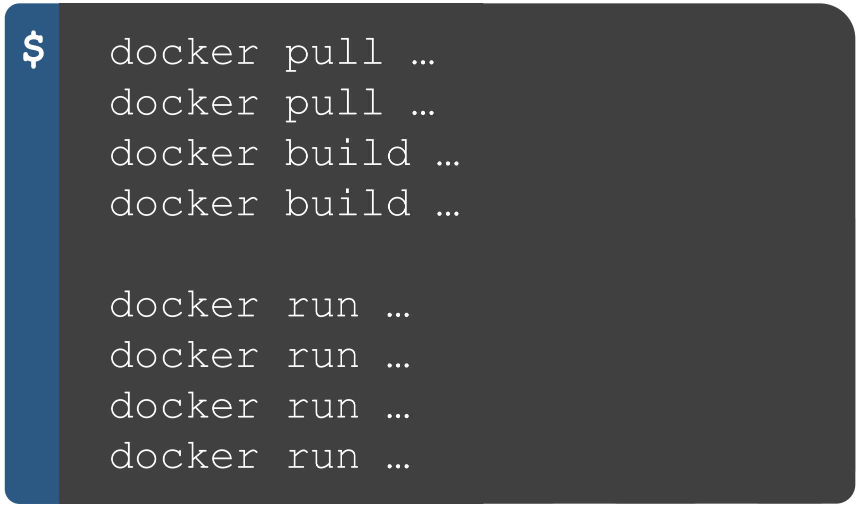


```
$  docker pull redis:latest

   docker build -t web .

   docker run -d --name=db redis:latest
   redis–server --appendonly yes

   docker run -d --name=web -p 5000:5000 -v
   `pwd`:/code web python app.py
```

# Compose – The whys and wherefores

***Multi-container apps are a hassle!***

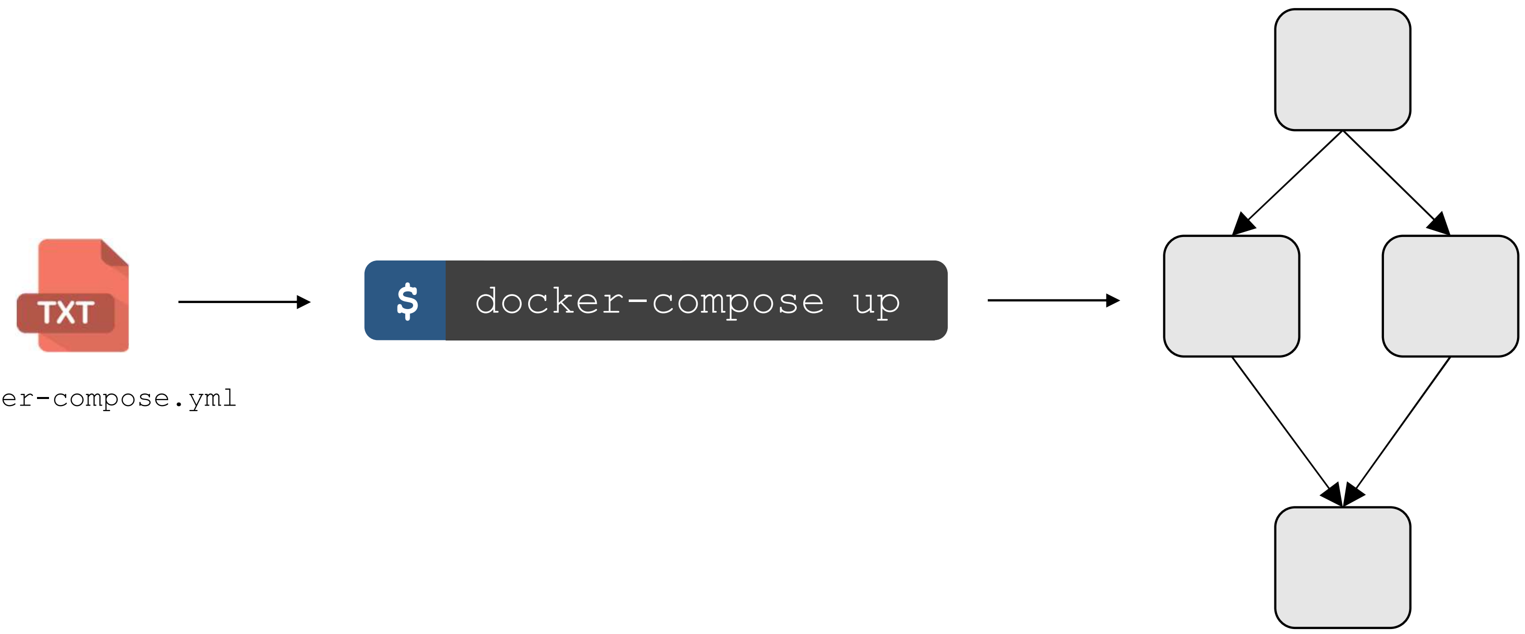




```
$   docker pull …
    docker pull …
    docker build …
    docker build …

    docker run …
    docker run …
    docker run …
    docker run …
```

# Compose – The whys and wherefores

***Ideally, we want to get an app running in one command!***



```
$ docker-compose up
```

# Compose – Overview

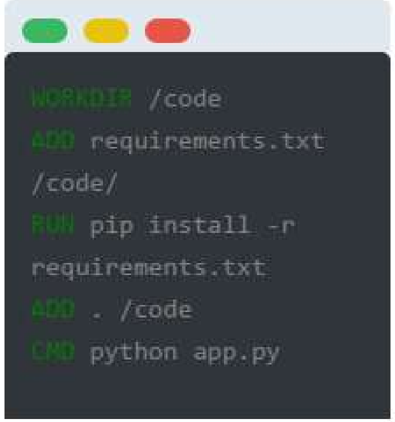
- ❑ Compose is a tool for defining and running multi-container Docker applications
- ❑ Recreate a microservices architecture on development and production machines
- ❑ All of that can be done by Compose in the scope of a single host

→ For multi-host deployment, you should use more advanced solutions, like Apache Mesos or a complete Google Kubernetes architecture

# Compose – Workflow

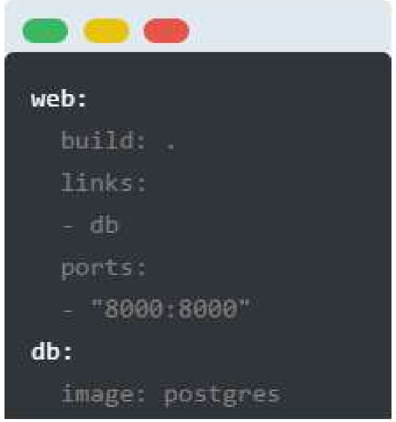
1. Define each service in a **`Dockerfile`**
2. Define the services and their relation to each other in the **`docker-compose.yml`** file
3. Use **`docker-compose up`** to start the system

**Write your dockerfile(s)**



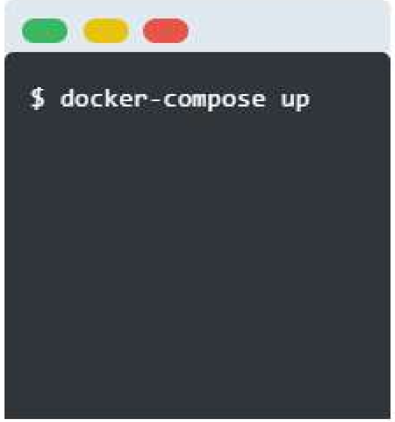
```
WORKDIR /code
ADD requirements.txt
/code/
RUN pip install -r
requirements.txt
ADD . /code
CMD python app.py
```

**Write your compose.yml file**



```
web:
  build: .
  links:
  - db
  ports:
  - "8000:8000"
db:
  image: postgres
```

**Run your app**



```
$ docker-compose up
```

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

# Compose – Orchestration

- The Compose file is a YAML file defining services, networks and volumes
- Configuration will be applied to each container started for that service, much like passing command-line parameters to **`docker run`**
- Likewise, network and volume definitions are analogous to **`docker network`** create and **`docker volume create`**
- Options specified in the Dockerfile (e.g., CMD, EXPOSE) are respected by default – no need to specify them again

**Compose File Reference – A great docu!!**

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

# Compose – Example

- Two services: `web` and `redis`
- `web` is built from the Dockerfile in the current directory
- Forwards the exposed port 5000 on the container

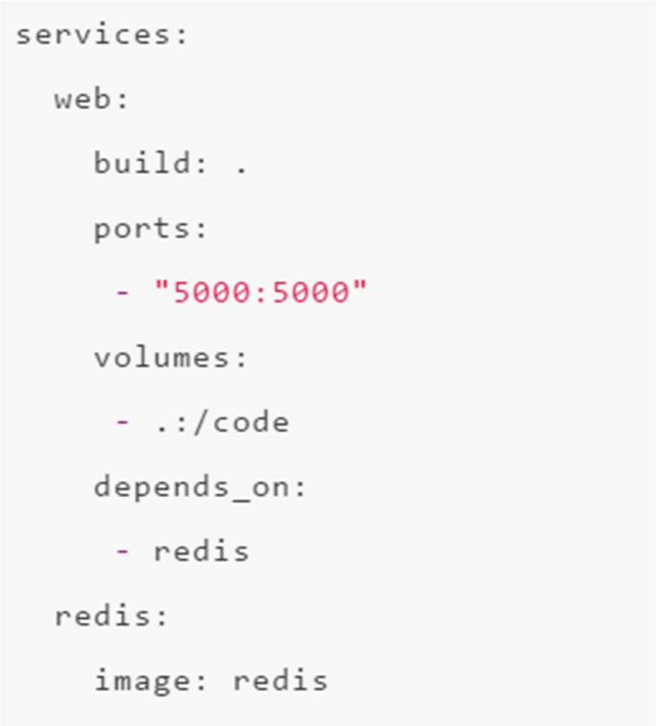
  to port 5000 on the host machine
- Mounts the current directory on the docker host (!!)

  to `/code` inside the container
- `depends_on` means, that `web` is started after

  `redis` is ready (starting order)
- Execute **`docker-compose up [-d]`**

  → After that the **`web`** container should

  be accessible at http://localhost:5000



```
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    depends_on:
     - redis
  redis:
    image: redis
```

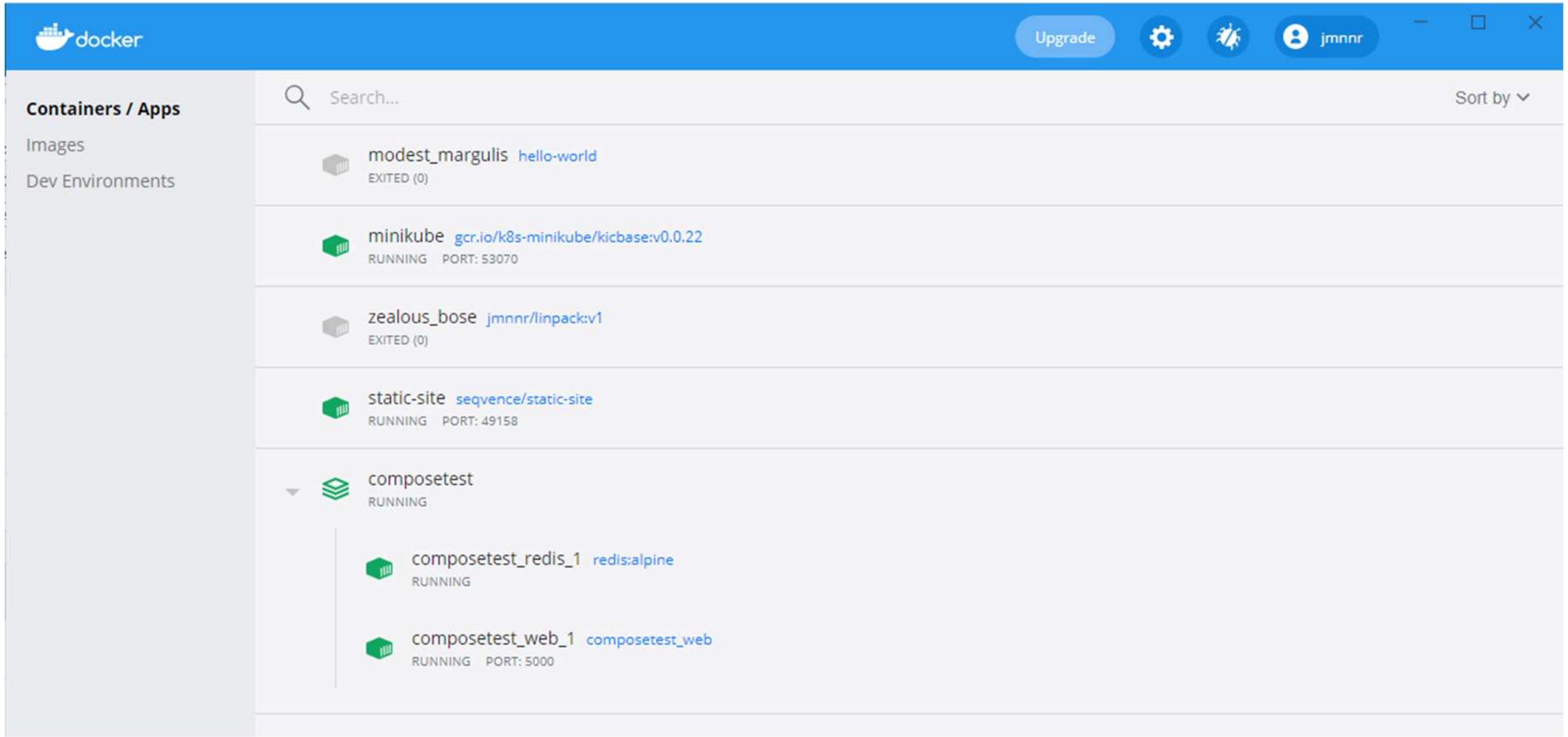
`docker-compose.yml`

# Compose – Commands

- **`docker-compose`** commands are a subset of docker counterparts but affect the whole multi-container architecture defined in docker-compose.yml

```
$ build -- Build or rebuild services
  logs -- View output from containers
  port -- Print the public port for a port binding
  ps -- List containers
  rm -- Remove stopped containers
  run -- Run a one-off command
  scale -- Set number of containers for a service
  start -- Start services
  stop -- Stop services
  up -- Create and start containers
```

# Docker Dashboard



modest_margulis hello-world
EXITED (0)

minikube gcr.io/k8s-minikube/kicbase:v0.0.22
RUNNING PORT: 53070

zealous_bose jmnnr/linpack:v1
EXITED (0)

static-site seqvence/static-site
RUNNING PORT: 49158

composetest
RUNNING

composetest_redis_1 redis:alpine
RUNNING

composetest_web_1 composetest_web
RUNNING PORT: 5000

- Benefits for development:
  - Running and terminated containers are shown in a nice view
  - Composed applications are grouped

# Docker – conclusive remarks

- State of the art development and deployment of applications via containers

- Easy to use Dashboard for Windows and Mac to interact with images and container (somehow an UI for the docker CLI):
  https://docs.docker.com/desktop/dashboard/

- Docker documentation is really great, worth reading and included in 2020/2021 a lot of information about CI/CD, building cloud native apps and deploying your containers to hosted K8s cluster or managed container services (e.g. AWS ECS)
  https://docs.docker.com/language/java/

# Related Technologies

Kubernetes is an open-source system for automating deployment, operations, and scaling of containerized applications. (Open sourced by Google)

Apache Mesos abstracts CPU, memory, storage, and other compute resources, enabling fault-tolerant and elastic distributed systems to easily be built and run effectively. (University of Berkeley)

Nomad is a tool for managing a cluster of machines and running applications on them. Nomad abstracts away machines and the location of applications, and instead enables users to declare abstract workloads. (HashiCorp)