Docker Training

Paolo Radaelli

What we'll cover

Morning:

- Introduction and Docker theory
- Single Docker container usage
- Creating your own Docker images with Dockerfile

Afternoon:

- Docker Compose
- Docker CI/CD
- Kubernetes Intro

Logistics

- Clone the Github Repo
- You'll need a text editor and a terminal (for Windows, PowerShell)
- You need a running version of Docker Desktop, local or Google Cloud Shell
- Topic exercises and a use case

What would you like to learn?

A use case Sunny Bikes

Sunny bikes is a bike sharing company. Currently sharing bikes in Austin (Texas), New York and San Francisco.

Their system consists of several components written in Python, but they're facing deployment issues because all components work with different versions and have different dependencies.

Throughout this course we will Dockerize and deploy the system!

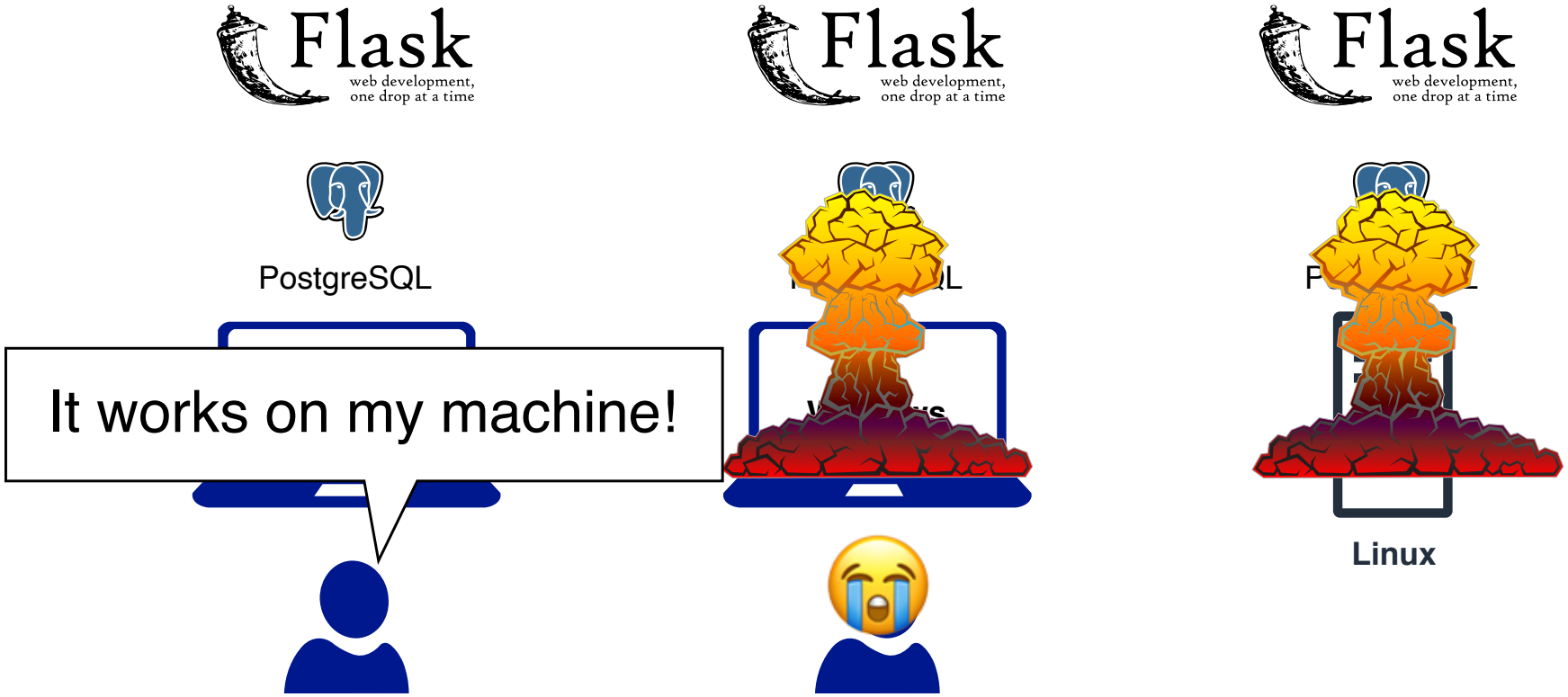
Exercises

1. First steps with Docker
2. Running a single container
3. Building a Docker image
4. Running multiple Docker containers with Docker Compose

The problem


Flask
web development,
one drop at a time
Flask
web development,
one drop at a time
Flask
web development,
one drop at a time
PostgreSQL
PostgreSQL
PostgreSQL
Mac
Windows
Linux

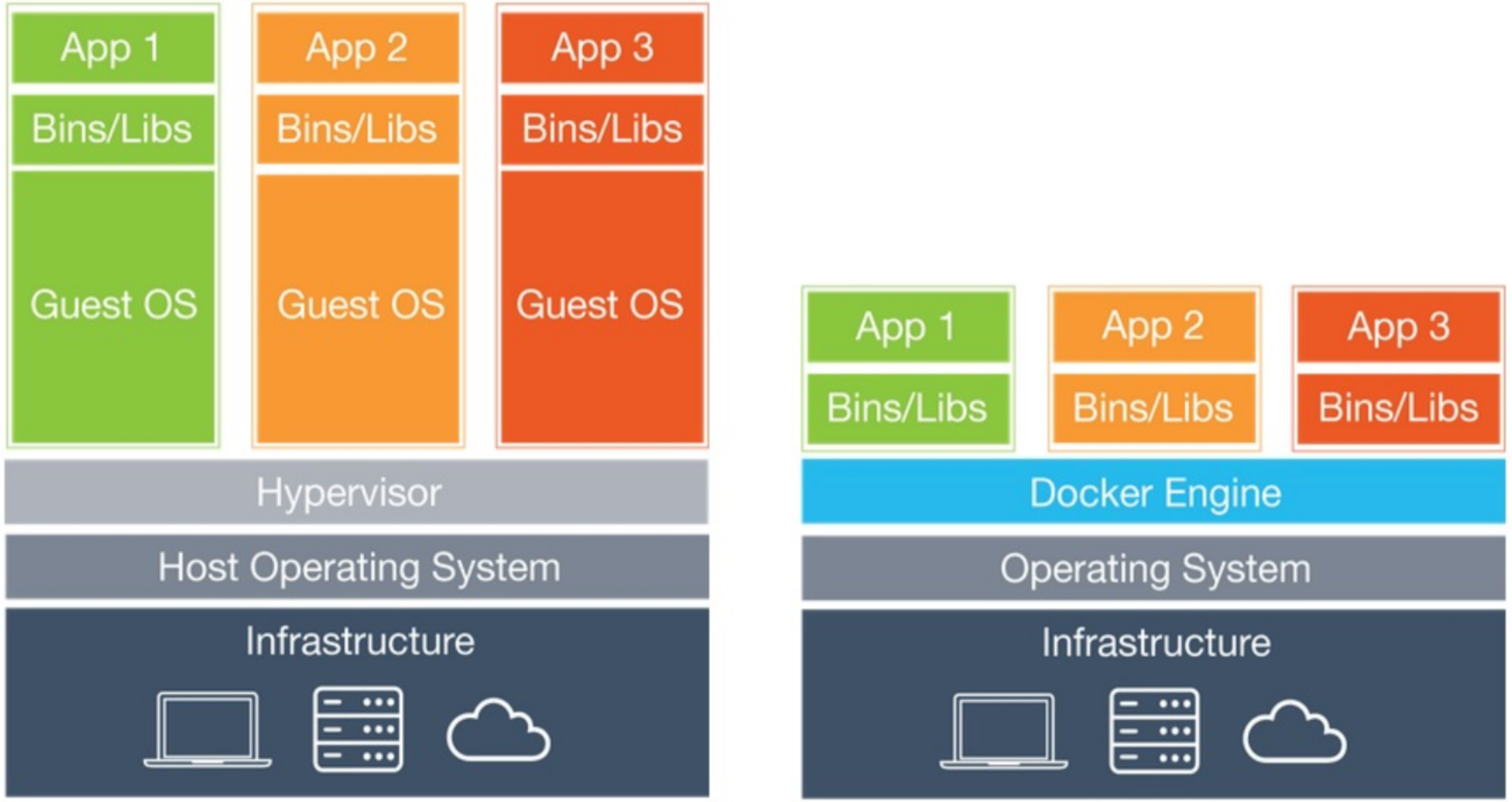
The problem


Flask
web development,
one drop at a time
Flask
web development,
one drop at a time
Flask
web development,
one drop at a time
PostgreSQL
It works on my machine!
Linux

Containers to the rescue

- Bundled dependencies
- Run it anywhere
- Lightweight
- Safety
- Easy Upgrades

Containers vs Virtual Machines

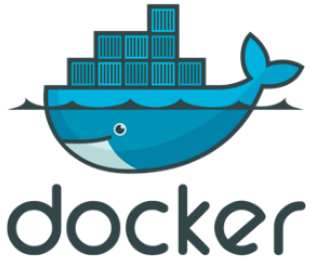


Docker introduction

- Most popular container engine, written in Go
- Provides tools to create and manage containers
- Rapid development – new release every ±3 months
- Docker is also the name of the company behind the technology
- Concept of isolation is not new. First implementation of process isolation goes back to 1979 with `chroot`.

Docker vs containers

- "Container" is a concept
- Docker is an implementation of that concept
- Alternatives:
  - LXD
  - OpenVZ by Virtuozzo
  - Podman ✨
  - Hyper-V Containers
  - Kata Containers by OpenStack
  - Buildah



Running Docker

- Built on Linux kernel features for process isolation.
- MacOS and Windows do not have these kernel features! They require a very lightweight VM to run Docker
- MacOS, Windows, Linux -> Docker Desktop (all use VM)
- Linux -> Docker Engine
- ⚠️ All require root/admin permissions to install

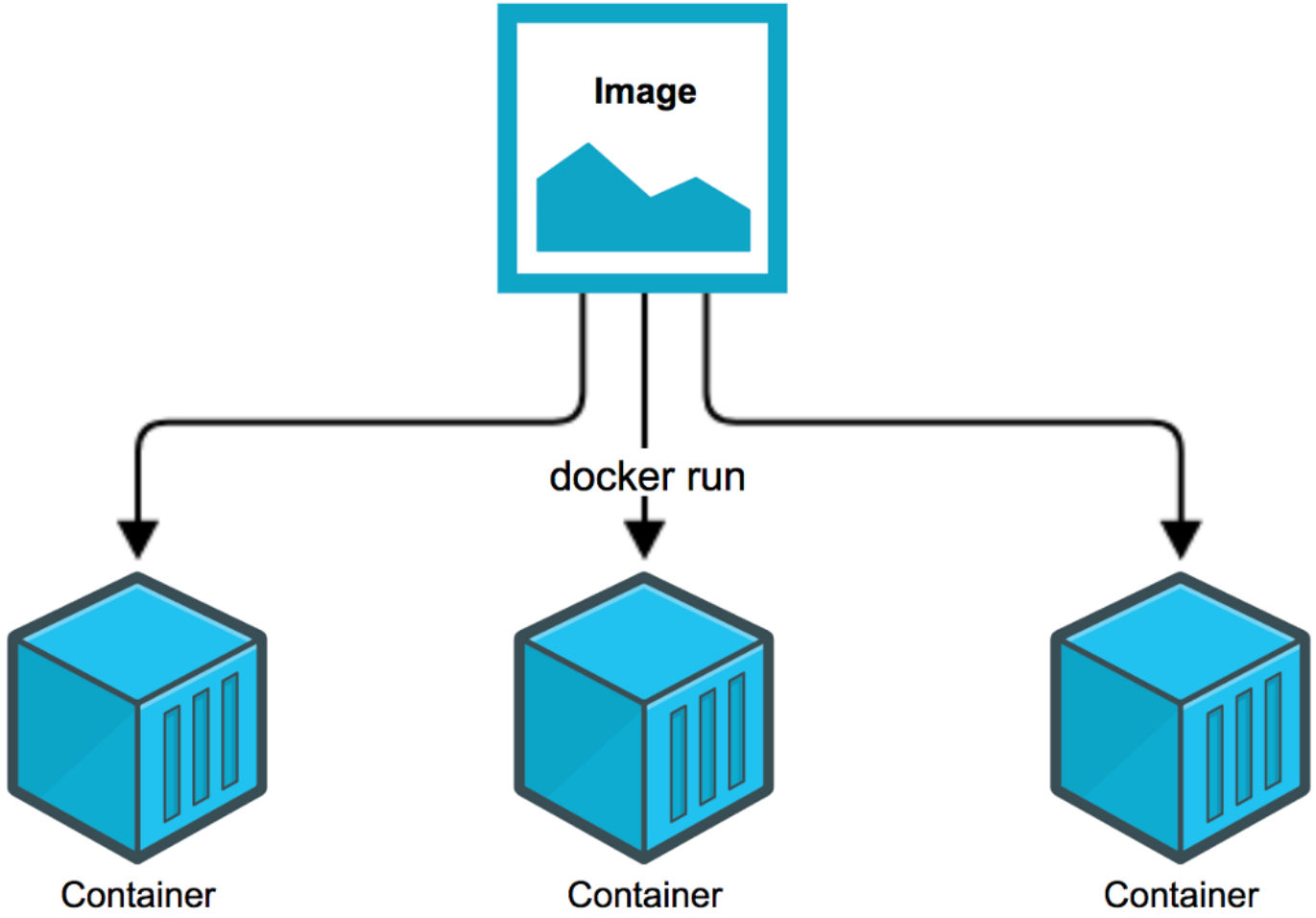
Docker popularity and licensing

- *Docker's surge in popularity created an environment where containers went from relatively obscure Linux kernel technology to standard developer tool for packaging software*. Phil Estes
- Docker was released in 2013 and its ease and simplicity quickly changed the way people develop and deploy software today.
- ⚠️ Licensing for Docker Desktop (+250 employees or +$10m revenue pa)
- Docker is no longer used as container runtime for Kubernetes

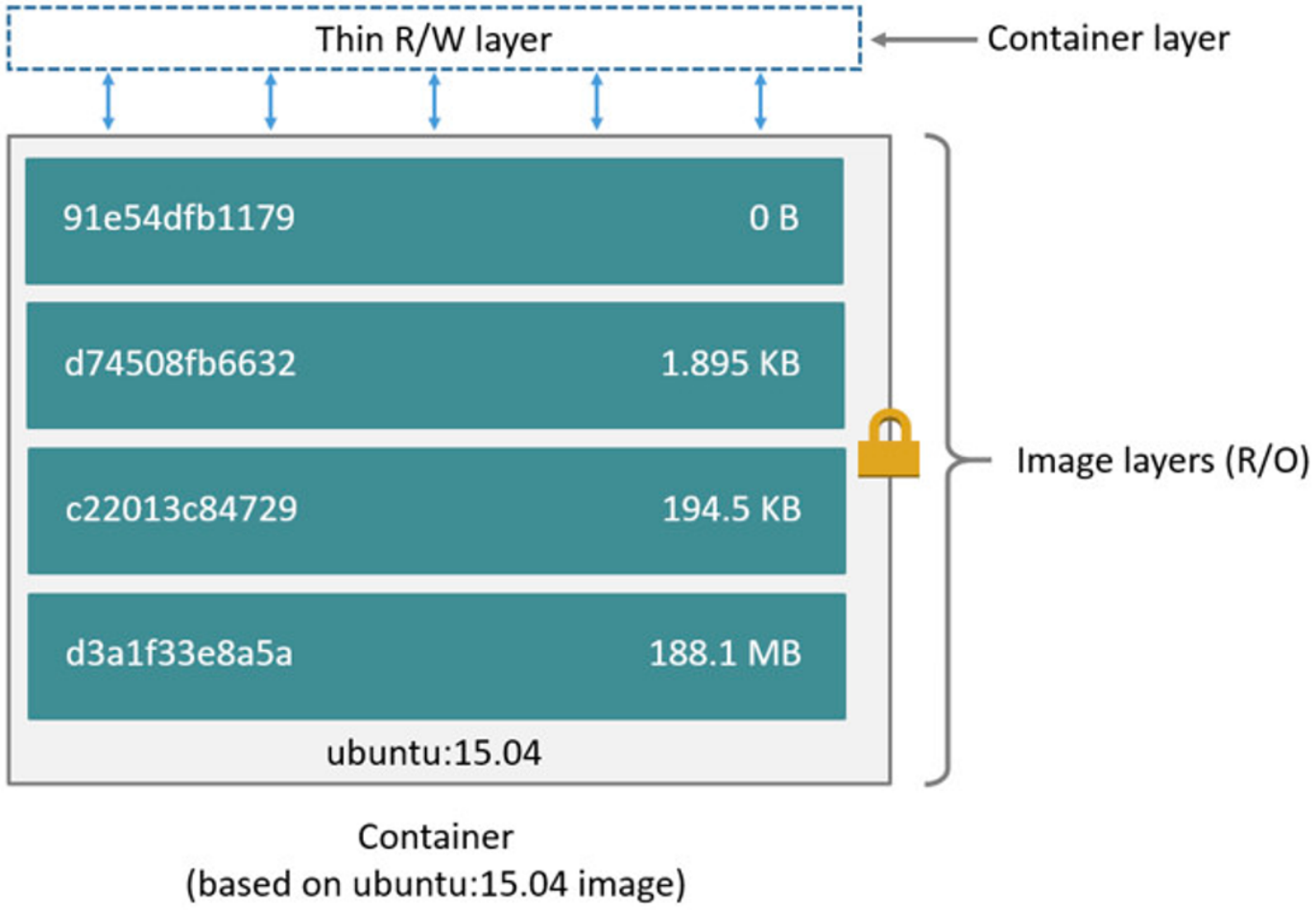
Docker terminology

1. Container
2. Image
3. Layer
4. Registry
5. Host

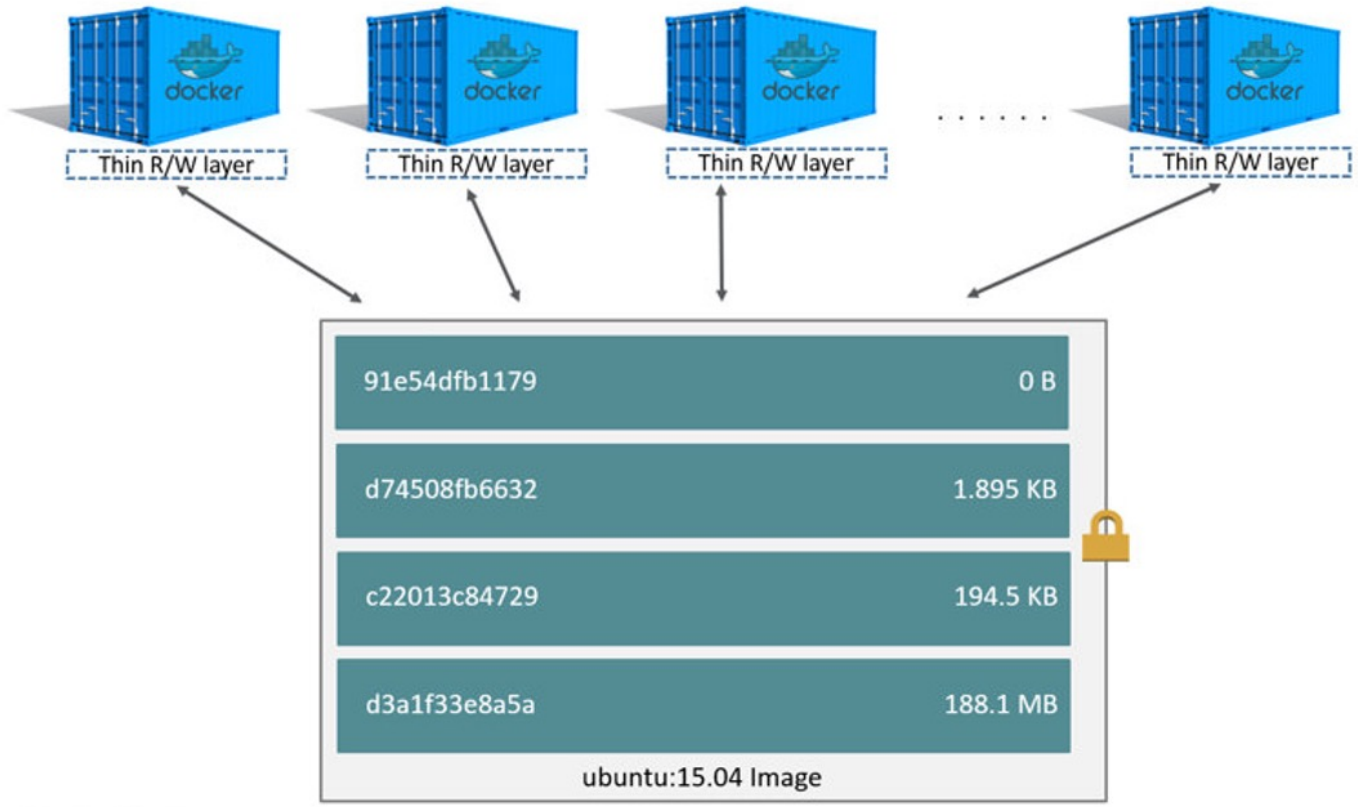
Docker terminology – Container & Image



Docker terminology - Layer



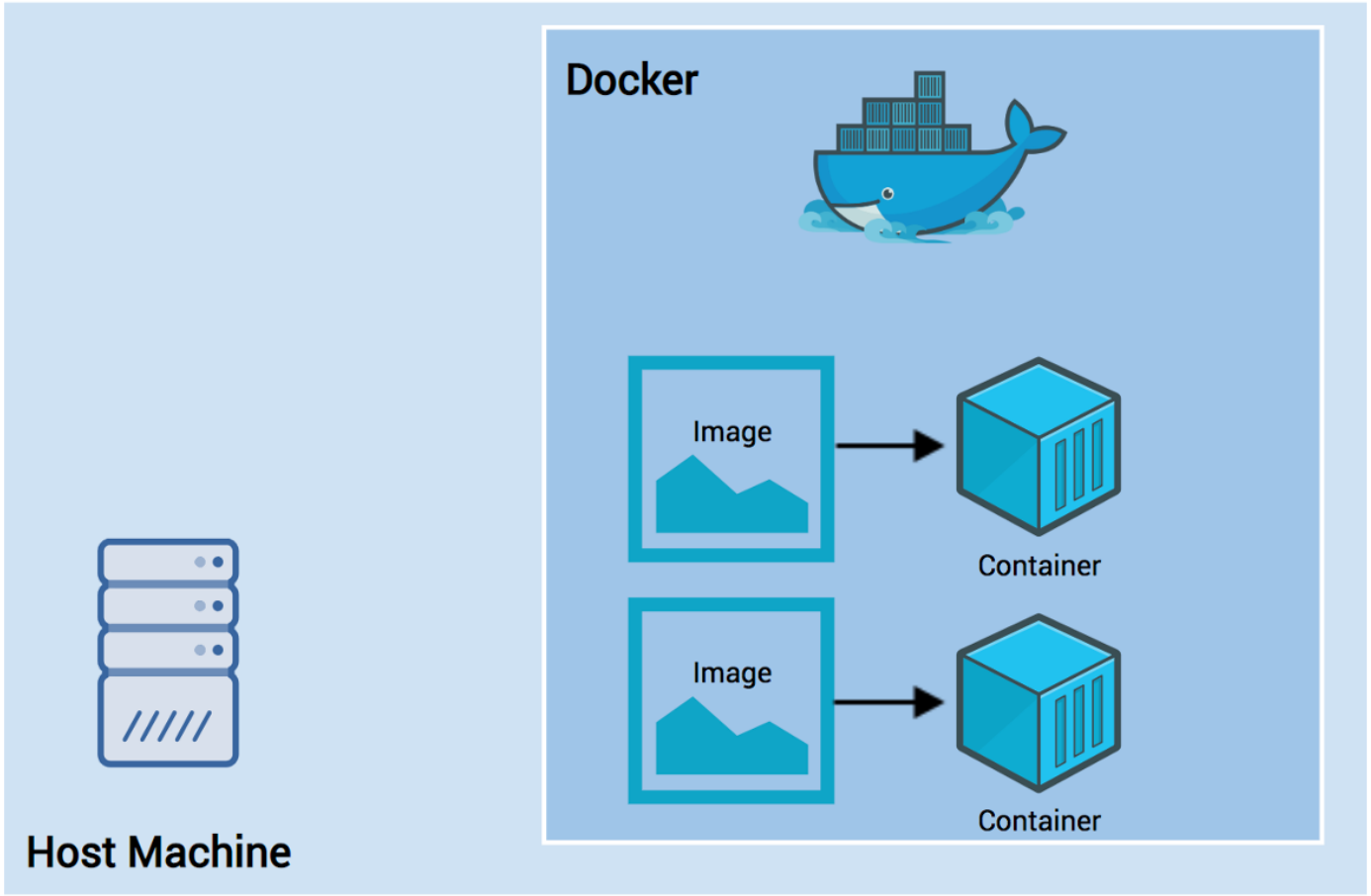
Docker terminology - Container vs Image



Docker terminology - Registry

- Collection of images and tags
- Analogy: a library of books -> books can have different versions
- Docker Hub is most used public registry
- You can deploy your own registry with Docker Distribution
- Most cloud providers offer private registries

Docker terminology - Host



Docker example

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

Docker example – what does it do?

```
docker run busybox echo hello world!
```

docker run [IMAGE] [COMMAND]

1. `docker run busybox`
2. `echo hello world!`

busybox

DOCKER OFFICIAL IMAGE · 1B+ · 2.9K

Busybox base image.

```
docker pull busybox
```

Overview Tags

Quick reference

- Maintained by:
  the Docker Community
- Where to get help:
  the Docker Community Slack, Server Fault, Unix & Linux, or Stack Overflow

Supported tags and respective `Dockerfile` links

- `1.36.0-glibc`, `1.36-glibc`, `1-glibc`, `unstable-glibc`, `glibc`
- `1.36.0-uclibc`, `1.36-uclibc`, `1-uclibc`, `unstable-uclibc`, `uclibc`
- `1.36.0-musl`, `1.36-musl`, `1-musl`, `unstable-musl`, `musl`

Recent Tags

latest uclibc musl glibc 1-uclibc 1-musl 1-glibc 1 unstable-uclibc unstable-glibc

About Official Images

Docker Official Images are a curated set of Docker open source and drop-in solution repositories.

Why Official Images?

These images have clear documentation, promote best practices, and are designed for the most common use cases.

Docker image tags

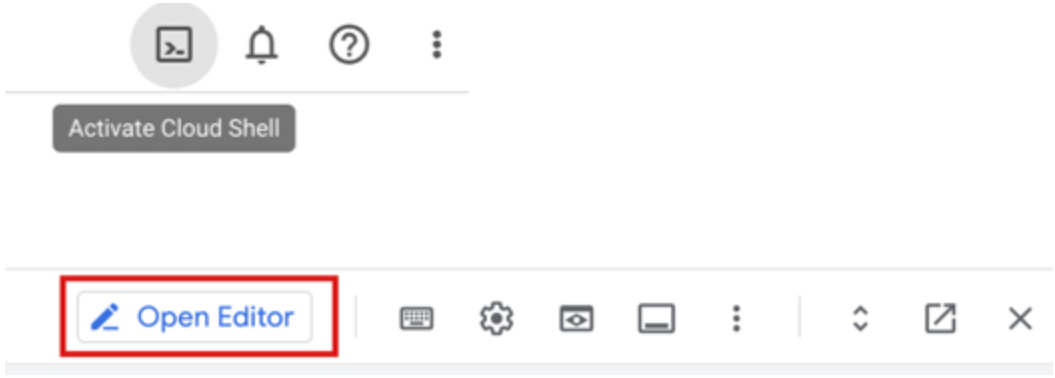
- Versioning of your images
- Notation: [IMAGE]:[TAG]
- e.g.: myimage:v1.0
- Sometimes [USER]/[IMAGE]:[TAG]
- Run specific version of an image
- Note: *tags are typically mutable!*

Docker - local install

- Linux, MacOS, Windows: Docker Desktop

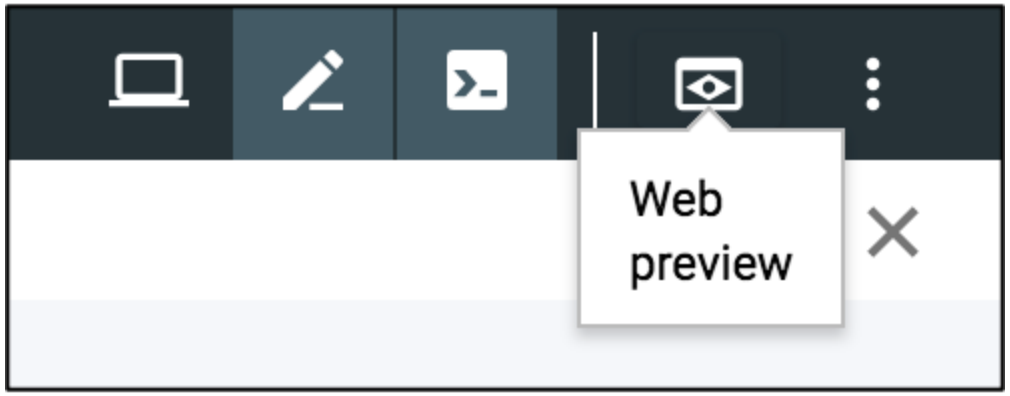
Google Cloud Shell

- Requires Google Cloud account (Gmail account is enough)
- Runs Docker version 20.10.22
- Go to https://console.cloud.google.com
- Free
- Shuts down after 20 minutes inactivity



Google Cloud Shell

- Features: https://cloud.google.com/shell/docs/how-cloud-shell-works
- 5GB of persistent disk
- Packed with tools (Docker, Minikube, NPM, Terraform, Python)
- Web preview is available on ports 2000 – 65000 (default 8080).



Exercise 1

1. `docker run -ti busybox sh`
2. `vi exercise.txt`
3. type "i" to use edit mode
4. type something
5. press "Esc" to exit edit mode and type `:wq` to save file and exit `vi`
6. `cat exercise.txt` (you should see the thing you just typed)
7. Exit the container (ctrl+D or "exit")
8. `docker run -ti busybox sh`
9. Repeat step 6, what do you notice?

Docker CLI

- Docker container management is done with the Docker CLI
- The Desktop App has a GUI
- Type docker for a list of all available commands

docker run

- Starts a container from a given image

```
docker run ––help
```

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

- Remove container when it’s done (--rm)
- Volumes (--volume/-v)
- Tty and interactive (--tty/-t and --interactive/-i)
- Mapping ports (--publish/-p)
- Environment variables (--env/-e)
- Detached mode (--detach/-d)
- Named containers (--name)

docker images

- `docker images`
  List all images.
- `docker push IMAGE`
  Push image to repository.
- `docker pull IMAGE`
  Pull image from repository (but don’t create container).
- `docker rmi IMAGE / docker image rm IMAGE`
  Remove image.

docker tag

- Create a target image that refers to a source image

  ```
  docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
  ```

- Example

  ```
  docker tag busybox:latest bob/my_image:v1
  ```

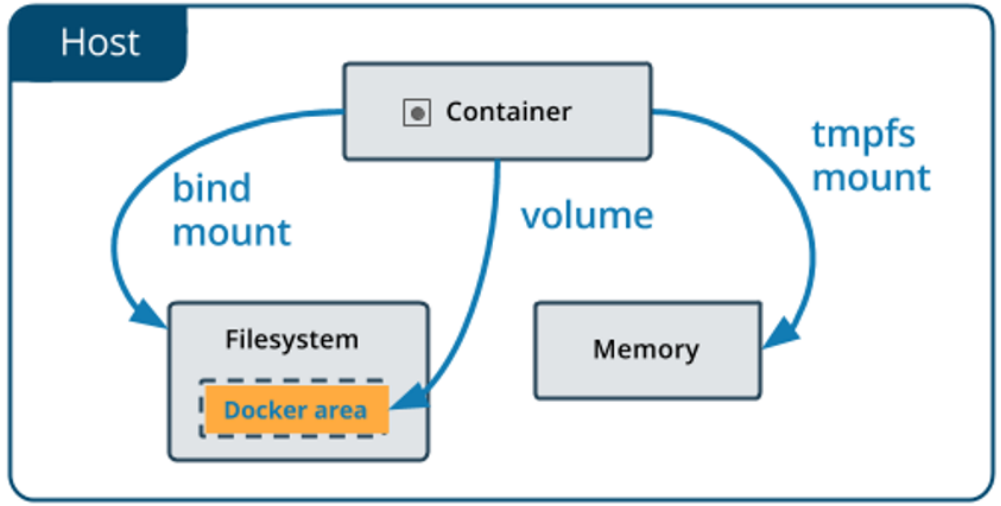
docker ps

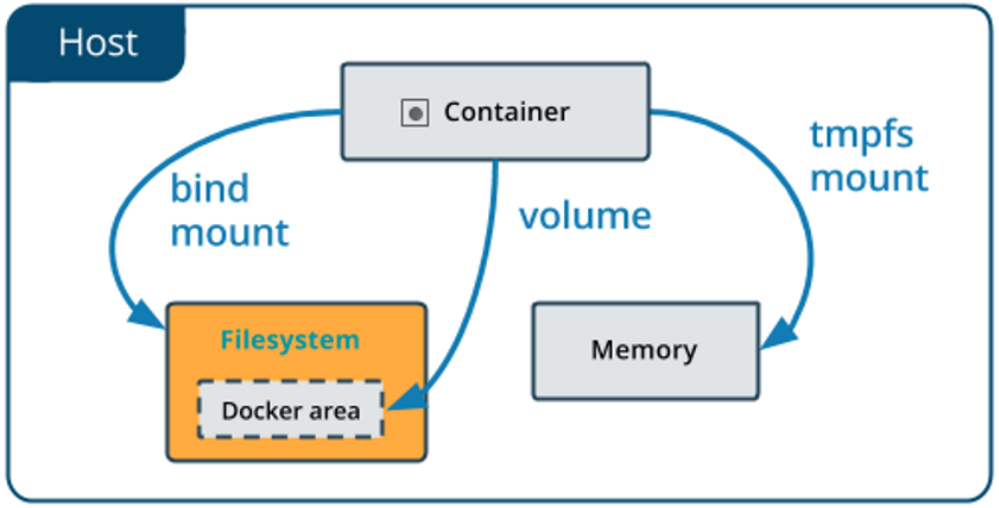
- `docker ps`
  List running containers.
- `docker ps -a`
  List all (running and stopped) containers.
- `docker rm -f $(docker ps -aq)`
  Remove all containers

docker run --rm IMAGE

- Removes an image after it closes
- Useful if you know you're not re-using the container. E.g., you want to test something in Python 3.7: `docker run --rm -ti python:3.7`
- By default, when a container stops, it remains on your system
- Check all (including stopped) containers with `docker ps -a`

Volumes and bind mounts





Volumes

- Volumes persist data generated and used by containers
- Completely managed by Docker

  `docker run -v VOLUME_NAME:CONTAINER_PATH IMAGE`

  `docker run --mount source=VOLUME_NAME,target=CONTAINER_PATH`
- Example: `docker run -v myvol2:/app ubuntu`

Bind mounts

- If you want to access data from your host system

```
docker run --volume/-v HOST_PATH:CONTAINER_PATH[:PERMISSIONS] IMAGE
```

- Note that both paths need to be absolute paths
- Examples:

```
# Linux
docker run -v "$(pwd)":/app:ro ubuntu

# Windows Powershell
docker run -v "${PWD}":/app:ro ubuntu
```

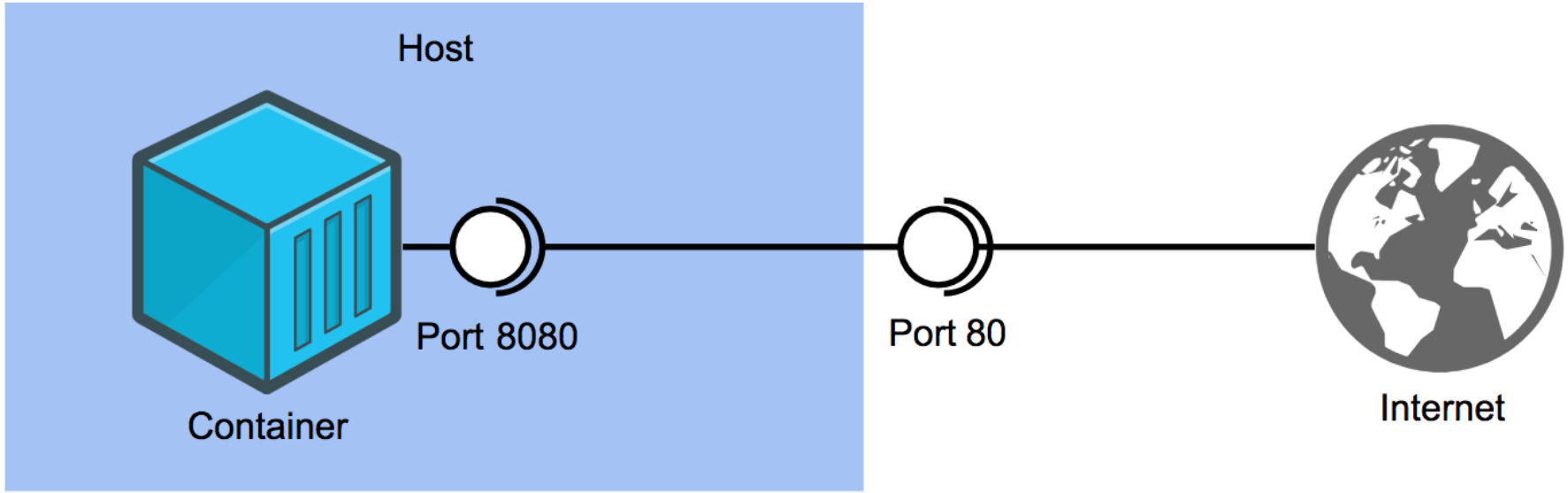
Volume management

- View volumes: `docker volume ls`
- Inspect volumes: `docker volume inspect my-vol`
- Remove a volume: `docker volume rm my-vol`

Port mapping

```
docker run -p HOST_PORT:CONTAINER_PORT IMAGE
```

```
docker run -p 80:8080 IMAGE
```



Docker run environment variables

- `docker run --env/-e VARIABLE=VALUE IMAGE`
- Used for secrets, configuration, etc...
- For example:
  `docker run -e PASSWORD=supersecret123! mywebserver`

Docker run detached mode

- `docker run --detach/-d IMAGE`
- Runs a container in the background
- Use if you want your container to run but not view the output.
- E.g. `docker run --detach postgres`
- Check stdout of detached container with `docker logs [-f] CONTAINER`

docker logs

- Tail stdout with `docker logs -f CONTAINER`

Docker named containers

- `docker run IMAGE` will give the container a randomly generated name
- Though easier to read than a hash, it's still random:

```
$ docker ps -a
CONTAINER ID IMAGE          COMMAND           CREATED       STATUS                     PORTS   NAMES
52e1304fd006 python:3.6.6   python3           10 hours ago  Exited (0) 10 hours ago          wonderful_shannon
8fd34d6cfb9b busybox        sh                14 hours ago  Exited (0) 14 hours ago          flamboyant_volhard
56fe2e718618 busybox        sh                14 hours ago  Exited (0) 14 hours ago          jolly_galois
b7bddba6a94c busybox        echo hello world! 17 hours ago  Exited (0) 17 hours ago          dreamy_kalam
```

Docker named containers

- You might like to give your container a meaningful name
- E.g. `docker run --name my_postgres postgres`
- Makes exec-ing into a container easier: `docker exec -ti my_postgres sh`

Other Docker container management

- `docker stop CONTAINER`
  Graceful shutdown of default 10 seconds, after which the container receives a SIGKILL if it hasn't shutdown by then. The time can be configured with --time/-t.
- `docker kill CONTAINER` Force kill container. ☠️
- `docker rm CONTAINER`
  Both docker stop & docker kill result in a stopped container. The container is not removed!
  Check stopped containers with `docker ps --filter "status=exited".`

Executing commands in a container

- `docker exec --tty --interactive CONTAINER sh`
- Short: `docker exec -ti CONTAINER sh`
- `--tty/-t` attaches the terminal to the docker container
- `--interactive/-i` accept stdin keystrokes
- *If you start a container with `docker run -ti IMAGE sh`, then Shell is the main process, and it will close once you quit Shell*

Other useful Docker CLI commands

```
docker system prune
```

Removes:

- all stopped containers
- all networks not used by at least one container
- all dangling images *(layers without relation to tagged images, consume disk space)*
- all dangling build cache

Other useful Docker CLI commands

```
docker inspect <image or tag>
```

To see more about an image

```
docker history --no-trunc <image or tag>
```

See full commands in history

Exercise 2

Sunny Bikes want to run their website in a Docker container.

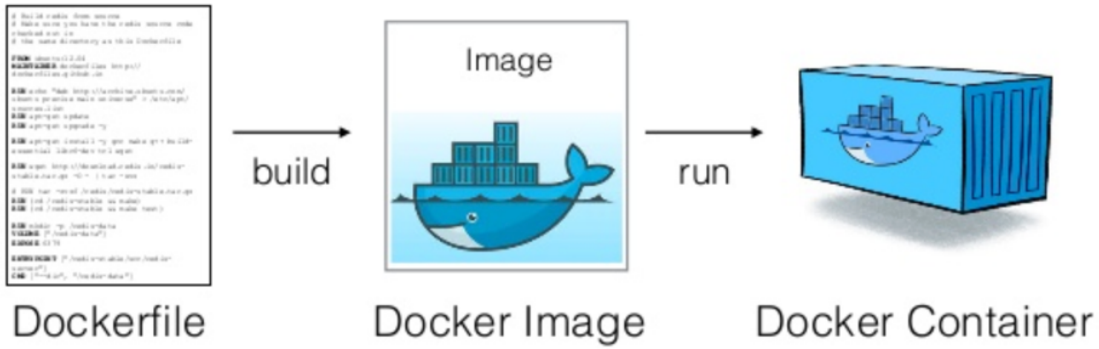
- They've already built the image, now it's up to you to deploy it!
- The image is available on Docker Hub at `pugillum/sunnybikesflask`.
- The image requires:
    i. To mount the file `data.txt` (available from Github files) to be mounted in `/root/data`
    ii. Environment variable `DATA_PATH` set to the file mounted in `/root/data`
    iii. Environment variable `FLASK_PORT` set to `8080`
    iv. Access to be available on `localhost:8080`
    v. Remove container when done

Exercise 2 solution

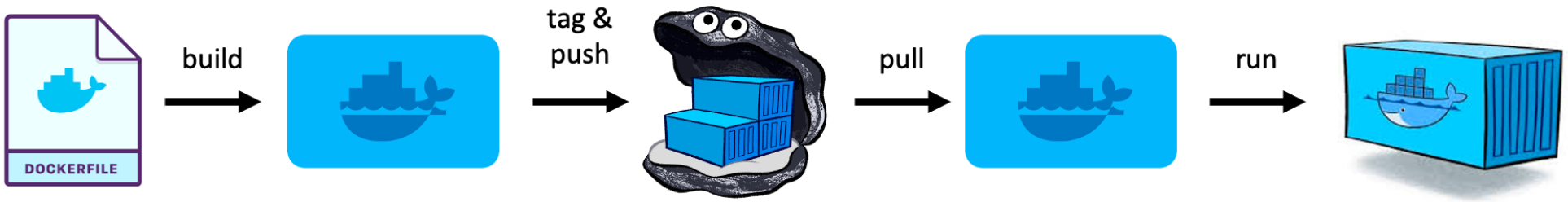
```
docker run \
-p 8080:8080 \
-e FLASK_PORT=8080 \
-v $(pwd)/data.txt:/root/data/data.txt:ro \
-e DATA_PATH=/root/data/data.txt \
--rm \
pugillum/sunnybikesflask
```

Dockerfile

- The recipe for a Docker image
- Define the recipe in a file called `Dockerfile`
- Run `docker build .` in directory containing the Dockerfile
- If for some reason you use a different filename:
  `docker build --file/-f [filename] .`



Docker lifecycle



Dockerfile structure

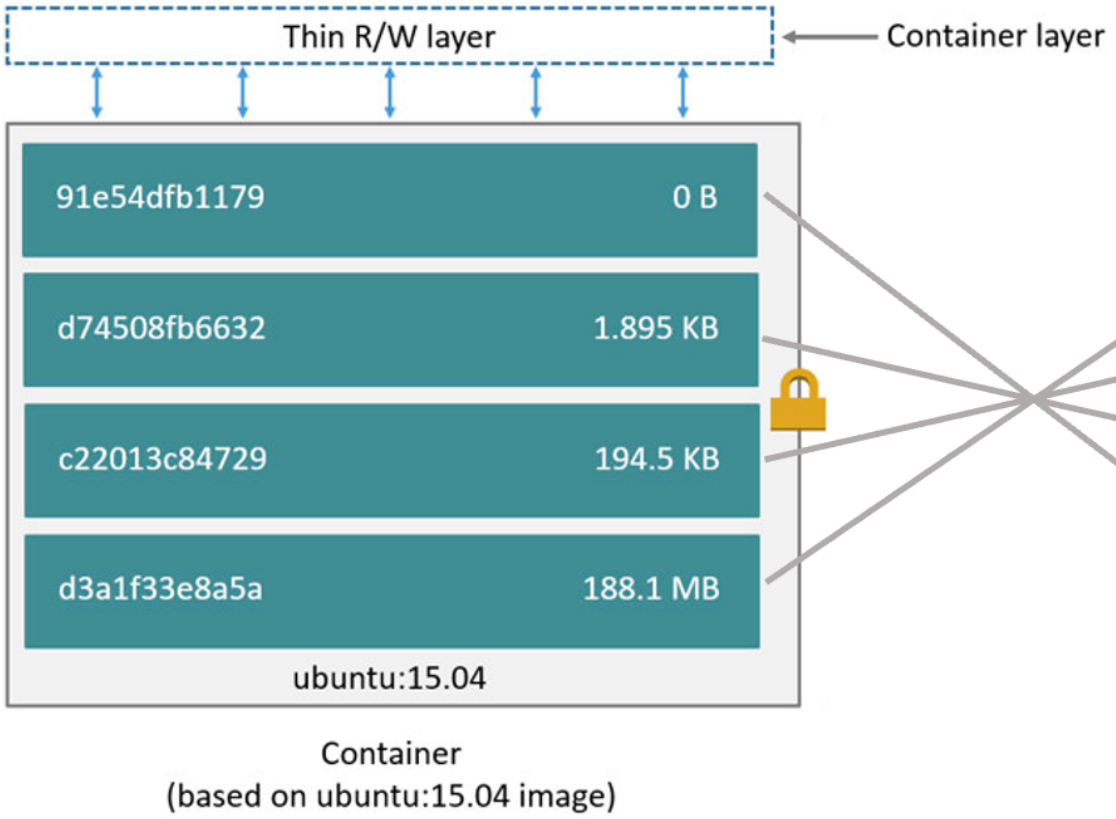
- FROM
- RUN
- ADD
- COPY
- ENV
- EXPOSE
- VOLUME
- WORKDIR
- ENTRYPOINT
- CMD
- See all at https://docs.docker.com/engine/reference/builder

Dockerfile example

```
FROM python:3.7
LABEL "project"="ACME"
LABEL "maintainer"="foo"
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
EXPOSE 8080
ENTRYPOINT ["python"]
CMD ["app.py"]
```

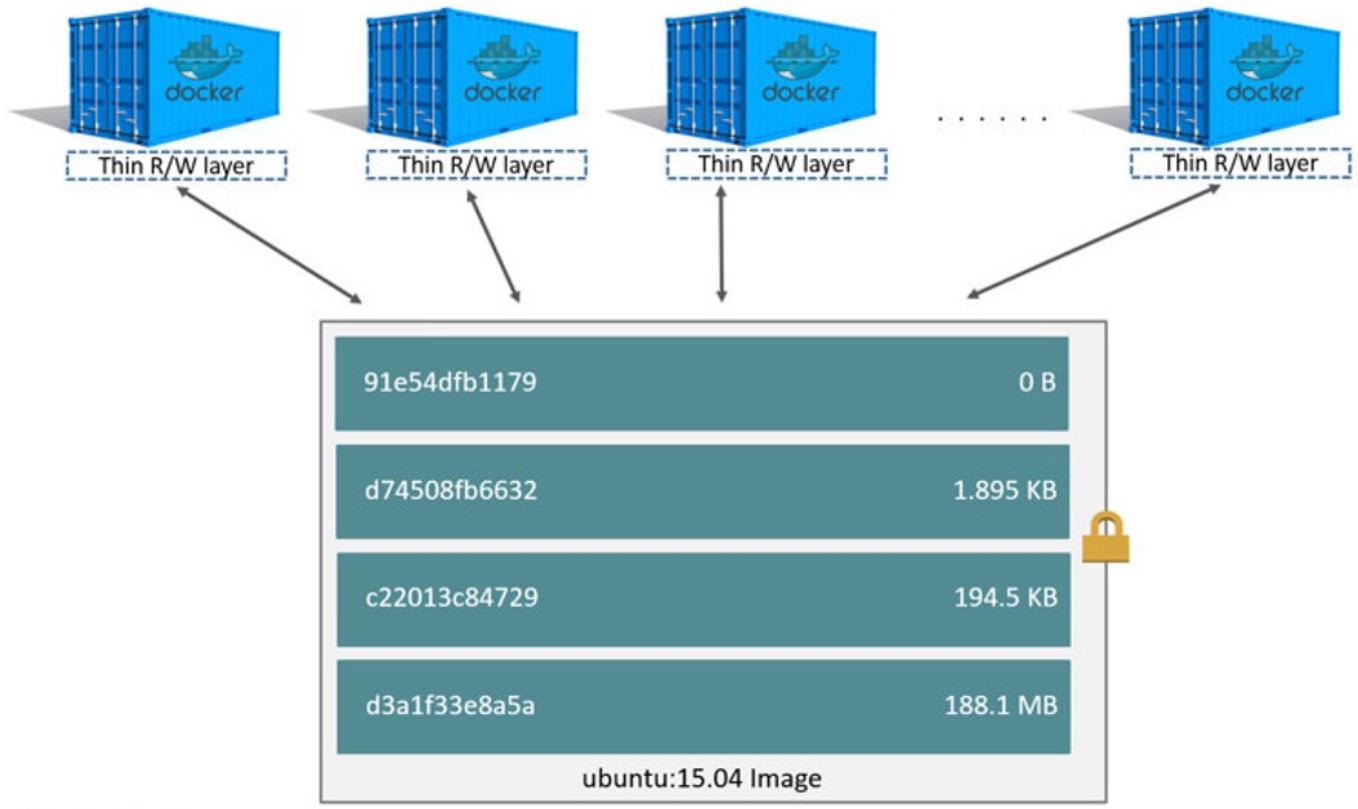
Build with `docker build [-t NAME] .`

Layered filesystem




```
FROM python:3.7
COPY . /app
RUN pip install -r requirements.txt
CMD ["app.py"]
```

Many containers



Dockerfile – FROM

- FROM sets the base image of the image to build
- These give the same result (at time of writing):

```
FROM python
FROM python:3.7.0
FROM python@sha256:10608fb357a18383f792efbf7472ec6d2e166dad62efc0d7c409ef2777aaafd0
```

- Check an image's digest with `docker images --digests`
- Default tag: `latest`
- Best practice: pin the base image version: `FROM python:3.7.0`

Dockerfile – RUN

- Execute shell statements

```
FROM python:3.7

RUN apk update && \
    apk add nginx \
    bash \
    make && \
    rm -r /var/cache

ENTRYPOINT ["python"]
```

- Put multiple RUN statements in a single layer!

Dockerfile – COPY & ADD

- Basic functionality is the same – copy files from host to image
- Not interested in magic details? Use `COPY`
- `ADD` can do more magic 🪄
- E.g.: `ADD foo.tar.gz` -> auto-extracts tar files

.dockerignore for COPY & ADD

- Exclude files and directories that match patterns in it

```
# This is a comment and is ignored
README.md
*/temp*
*/*/temp*
temp?
```

Dockerfile – WORKDIR

- Sets working directory for RUN, CMD, ENTRYPOINT, COPY, ADD and consecutive WORKDIR statements

```
FROM busybox WORKDIR /a
WORKDIR b
WORKDIR c
ENTRYPOINT ["pwd"]
```

```
$ docker build -t test .
$ docker run test /a/b/c
```

Dockerfile – LABEL

- Add metadata to an image
- Multi labels doesn’t increase image size

```
LABEL "creator"="Paolo Radaelli"
LABEL "project"="ACME"
```

Dockerfile – EXPOSE

- Only informs Docker that the container listens on the given port
- ⚠️EXPOSE does not publish the port! So you still need to do e.g.:
  `docker run -p 8080:8080 python`
  `docker run -P python` will map all exposed ports to a random host port
- Expose multiple ports with:
  `EXPOSE 80 443 8080`

Dockerfile – ENTRYPOINT & CMD

- The main process in the container
- Must have at least one of ENTRYPOINT or CMD

- Convention is to point ENTRYPOINT to the executable
- And pass the default arguments with CMD

For example:

```
FROM alpine
ENTRYPOINT ["ping"]
CMD ["www.google.com"]
```

Dockerfile – ENTRYPOINT & CMD

- Difference between ENTRYPOINT and CMD allows overriding arguments with docker run:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

```
FROM alpine
ENTRYPOINT ["ping"]
CMD ["www.google.com"]
```

```
docker build -t . myping
```

```
docker run myping www.facebook.com
```

Dockerfile – ENTRYPOINT & CMD

These all have the same effect:

```
FROM alpine
ENTRYPOINT ["ping", "www.google.com"]
```

```
FROM alpine
CMD ["ping", "www.google.com"]
```

```
FROM alpine
ENTRYPOINT ping www.google.com
```

```
FROM alpine
CMD ping www.google.com
```

Dockerfile – ENTRYPOINT & CMD

Exec form

```
FROM alpine
ENTRYPOINT ["ping", "www.google.com"]
```

```
FROM alpine
CMD ["ping", "www.google.com"]
```

Shell form

```
FROM alpine
ENTRYPOINT ping www.google.com
```

```
FROM alpine
CMD ping www.google.com
```

⚠️ In Shell form, signals are not sent to the subshell

Dockerfile – ENTRYPOINT & CMD

- Point to executable in `ENTRYPOINT`
- Pass default arguments in `CMD`
- Use exec form for both `ENTRYPOINT` and `CMD`

```
FROM alpine
ENTRYPOINT ["ping"]
CMD ["www.google.com"]
```

- `ENTRYPOINT` can be overridden with docker run with `--entrypoint`:
  `docker run --entrypoint ls IMAGE /var`

Dockerfile – ARG

- ARG instruction defines a variable that can be passed in at build time
- Can have one or more
- Can have defaults

```
FROM busybox
ARG user1=someuser
ARG buildno=1
```

- Call during build with `--build-arg <varname>=<value>`
- Not recommended for secrets, visible in history

Dockerfile – ENV

- Set environment variables in the image with `ENV`
- Only set non-sensitive variables
- If using the variable only once, you can also do: `RUN KEY=VALUE` command

Define single variable:

```
ENV key value
ENV key2 value2
ENV key3 value3
```

Define multipe variables:

```
ENV key=value \
key2=value2 \
key3=value3
```

Dockerfile statement ordering

- Docker layers are cached and re-used when rebuilding
- Which is better and why?

```
FROM python:3.7
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
EXPOSE 8080
ENTRYPOINT ["python"]
CMD ["app.py"]
```

```
FROM python:3.7
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY ./app.py .
EXPOSE 8080
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Linting

- Check the quality of your Dockerfile
- hadolint `hadolint Dockerfile`
- https://www.fromlatest.io - an online Dockerfile scanner

Multi-stage builds

```
FROM python:3.7-alpine as base
WORKDIR /code
COPY requirements.txt .
RUN pip install -r requirements.txt
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
EXPOSE 5000

FROM base as dev
ENV FLASK_DEBUG=1
CMD ["flask", "run"]

FROM base as prd
COPY . .
CMD ["flask", "run"]
```

Build just dev stage with:

```
docker build --target dev .
```

Multi-stage builds - advanced optimization

```
FROM python:3.7-slim AS compile-image
RUN apt-get update
RUN apt-get install -y --no-install-recommends build-essential gcc

RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY setup.py .
COPY myapp/ .
RUN pip install .

FROM python:3.7-slim AS build-image
COPY --from=compile-image /opt/venv /opt/venv

# Make sure we use the virtualenv:
ENV PATH="/opt/venv/bin:$PATH"
CMD ["myapp"]
```

Exercise 3

- Deploy a Flask API in a Docker container
- The Flask API code will be made available
- Write the Dockerfile to create the Docker image
- Success once you see this:

No name supplied. Could not find any bike rides for this user.

- Or this: (browse to `localhost:8080?name=bas`)

Hello bas!

You bike rides are:

- San Fransisco - 5 km - 2018/01/01
- New York - 15 km - 2018/02/01
- New York - 8 km - 2018/03/01
- Austin - 18 km - 2018/04/02

Exercise 3 solution

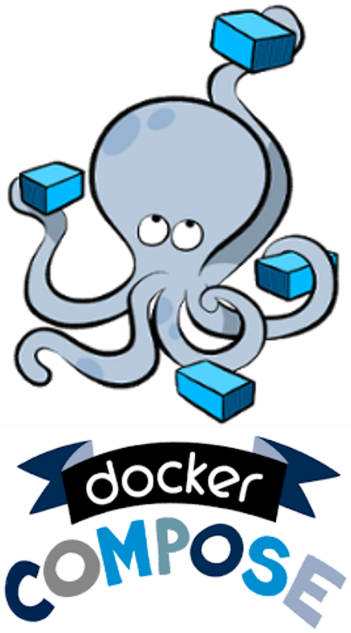
```
FROM python:3.7.1-slim
WORKDIR /app

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .
EXPOSE 5000
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Dockerfile tips

1. Use official images if possible
2. The order matters (caching)!
3. Minimize the number of layers
4. Consider multi-stage builds for advanced optimisation
5. Use a linter
6. Use .dockerignore just like .gitgnore
7. Don’t build secrets into an image

Docker Compose

- `docker compose` extension comes installed with Docker Desktop
- it can be installed separately on Linux
- Easy to get multiple containers running on your own machine
- Only recreate containers that have changed
- Docker network links between services by hostname
- Defined in file called docker-compose.yaml

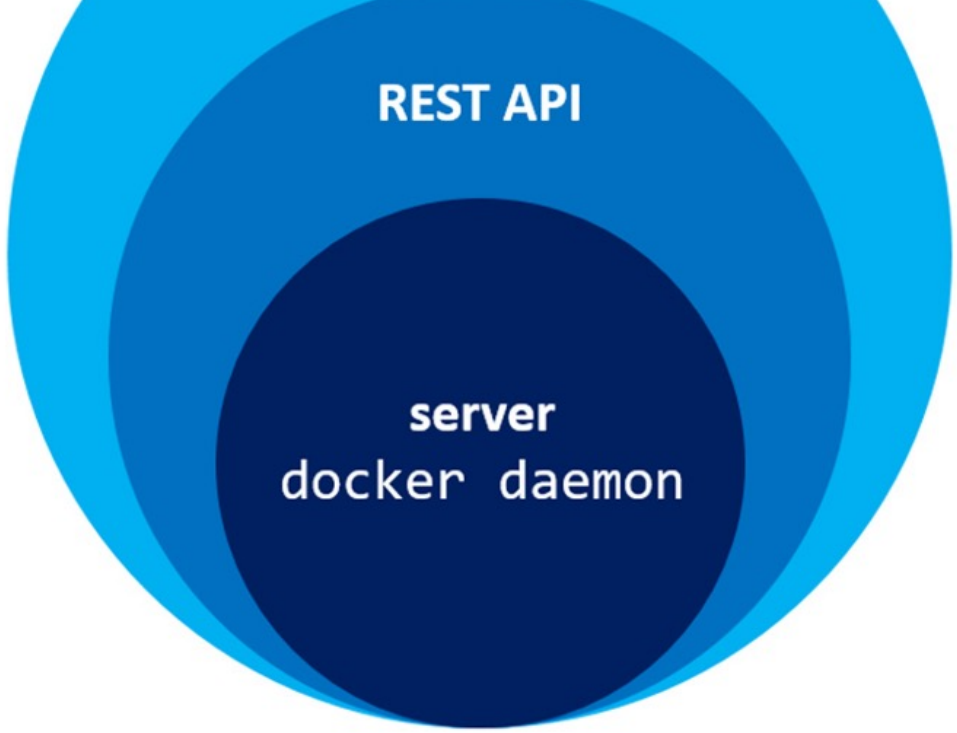


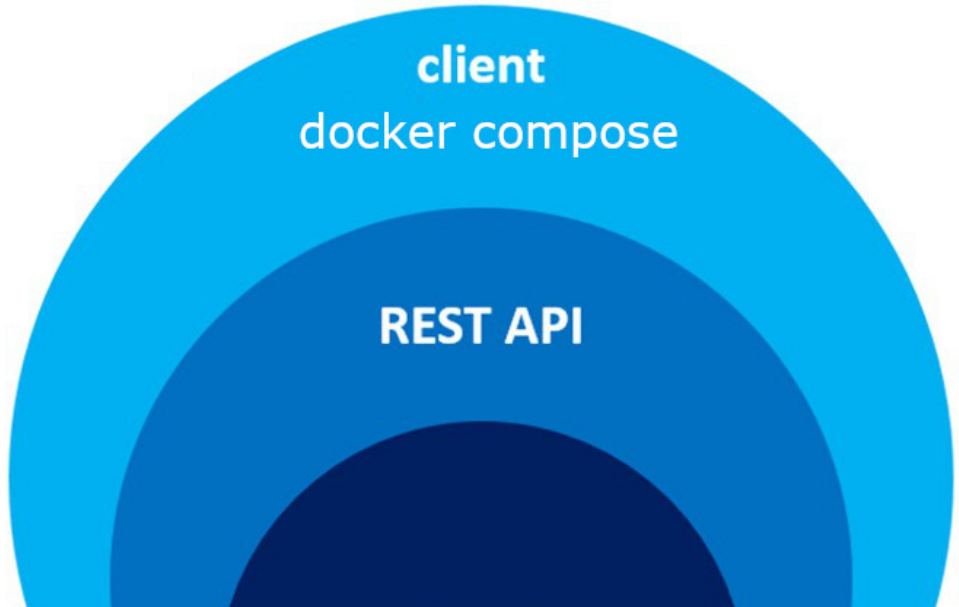
Google Cloud Shell setup

- Google Cloud Shell unfortunately does not have compose installed
- To install it run the following: `sudo apt-get install docker-compose-plugin` in the terminal

Running multiple containers

- Separate concerns over containers
- Your Flask API can be split into a backend, frontend and a Postgres database
- Multiple ways of connecting containers:
    - Docker Compose
    - Kubernetes Pods
    - Cloud solutions:
        - AWS EKS,ECS,Fargate & Beanstalk
        - GCP GKE,CloudFunctions & AppEngine
        - Azure AKS,ACI,Functions & WebAppforContainers


REST API
server
docker daemon


client
docker compose
REST API

From CLI -> Compose File

- Terminal

```
> docker build -t my-app .
> docker run --rm -p 8000:5000 -n my-app my-app
```

- docker-compose.yaml

```yaml
services:
  app:
    build: .
    container_name: my-app
    ports:
        - 8000:5000
```

To build and run

```
> docker compose up --build

# Make some changes

> docker compose up --build
```

- The `--build` is needed otherwise the image won't be rebuilt
- To remove stopped containers you can use `docker compose down`

Multiple services

```
services:
  app:
    build: .
    ports:
        - 8000:5000
    volumes:
      - .:/app
  redis:
    image: "redis:alpine"
```

- Docker-compose automatically creates a network
- Both services join this network
- Both services are reachable in this network given their hosts name
- For example: **app** can connect to a database with **postgres://db:5432**

```
services:
  db:
    image: postgres:10
    environment:
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_DB=${POSTGRES_DB}
    volumes:
      - ./postgres-data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
  app:
    build: .
    depends_on:
      - db
    ports:
```

Develop in your container

- This configures a build mount

```
services:
  web:
    build: .
    ports:
      - 8000:5000
    volumes:
      - .:/app
```

- Reference with target in `docker-compose.yaml`

```
FROM python:3.7-alpine as base
WORKDIR /code
COPY requirements.txt .
RUN pip install -r requirements.txt
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
EXPOSE 5000

FROM base as dev
ENV FLASK_DEBUG=1
CMD ["flask", "run"]

FROM base as prd
COPY . .
CMD ["flask", "run"]
```

```
services:
  web:
    build:
      context: .
      target: dev
    ports:
```

Exercise 4

now wants to productionize their API.
This includes persistent storage and the possibility to scale out.

In `exercise-04` Update the `docker-compose.yaml` to run the following images:

- `postgres:11-alpine`
- Image built from `Dockerfile` (from previous exercise)

The database should be secured with password `long-distance-ice-skating` and initialised with the `init-schema.sql` script

Expected result:

- Insert data on `127.0.0.1:8080/rent?name=bob&location=texas`
- Browse to url:8080 and see the inserted data

Exercise 4

```
services:
  postgres:
    image: postgres:11-alpine
    environment:
      POSTGRES_PASSWORD: long-distance-ice-skating
    ports:
      - "5432:5432"
    volumes:
      - './init-schema.sql:/docker-entrypoint-initdb.d/init-schema.sql'
  flask:
    build: .
    ports:
      - "8080:5000"
    environment:
      PG_PASSWORD: long-distance-ice-skating
    depends_on:
      - postgres
```

Docker in Github Workflows

```yaml
- name: Log in to Docker Hub
  uses: docker/login-action@f054a8b539a109f9f41c372932f1ae047eff08c9
  with:
    username: ${{ secrets.DOCKER_USERNAME }}
    password: ${{ secrets.DOCKER_PASSWORD }}

- name: Extract metadata (tags, labels) for Docker
  id: meta
  uses: docker/metadata-action@98669ae865ea3cffbcbaa878cf57c20bbf1c6c38
  with:
    images: my-docker-hub-namespace/my-docker-hub-repository

- name: Build and push Docker image
  uses: docker/build-push-action@ad44023a93711e3deb337508980b4b5e9bcdc5dc
  with:
    context: .
    push: true
    tags: ${{ steps.meta.outputs.tags }}
    labels: ${{ steps.meta.outputs.labels }}
```

Docker in Github Workflows

```
- name: Authenticate with Azure
  run: |
    az login --service-principal -u "${{ inputs.client_id }}" \
    -p='${{ secrets.client_secret }}' --tenant "${{ inputs.tenant_id }}"
    az account set --subscription ${{ inputs.subscription_id }}

- name: Authenticate with Azure Container Registry
  run: az acr login -n ${{ inputs.container_registry }}

- name: Build and deploy image
  run: |
    docker build . --file Dockerfile \
    --tag "${{ inputs.container_registry }}/my-app:${GITHUB_SHA::7}"
    docker push "${{ inputs.container_registry }}/my-app:${GITHUB_SHA::7}"
```

Podman

- Popular alternative to Docker
- Embeds a guest Linux system
- On Mac QEMU VM
- On Windows, uses WSL
- Less user-friendly

podman

Podman commands

Docker

```
docker build ...
docker run ...
docker ps
docker images ...
docker logs <containerid>
docker top <containerid>
```

Podman

```
podman build ...
podman run ...
podman ps
podman images ...
podman logs <containerid>
podman top <containerid>
```

Conclusion: Not much difference!

The end of the Docker Slides

- Thank you for listening!