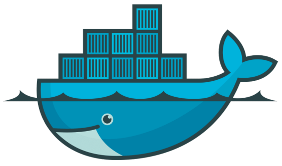
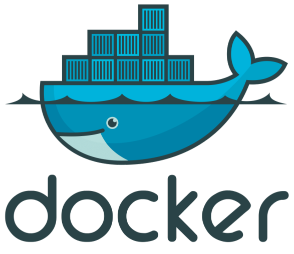

docker



Motivation

History

Docker (desktop) architecture

Hello world

Motivation

Without Docker:

- "To set up the development environment install Postgres, MongoDB, and run these 5 scripts. Oh wait, you are on Windows? Then also change these configurations!"
- "To deploy the application you need a server running Ubuntu. Run this Ansible playbook to install the dependencies and configure the system, then copy the binary and run it with these options."

With Docker:

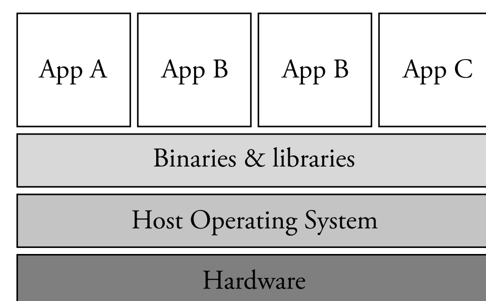
- "docker compose up"
- "docker run [options] container_image"

History

In the beginning there were **physical machines** (bare metal)...

Issues:

- *difficult maintenance* due to dependency hell
  e.g. apps require different versions of bins & libs
- *low resource utilization*
  due to limited no. of applications that can coexist
- *isolation*
  e.g. apps can interfere with each other
- *slow startup / shutdown / reboot*



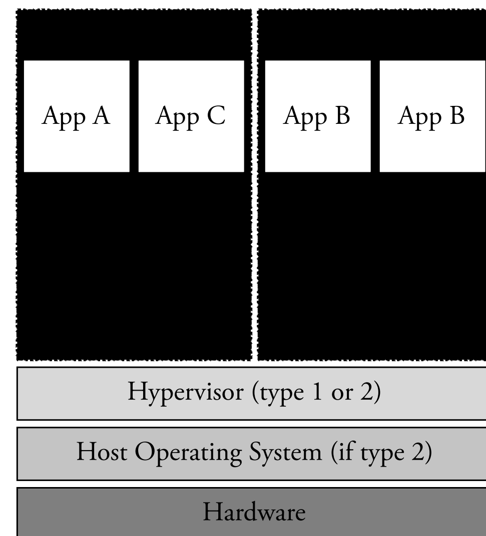
History

In the beginning there were **physical machines** (bare metal)...

...then **virtual machines** were created (type 1 and type 2)

Issues:

- ~~*difficult maintenance* due to dependency hell~~
  ~~e.g. apps require different versions of bins & libs~~
- ~~*low resource utilization*~~
  ~~due to limited no. of applications that can coexist~~
- ~~*isolation*~~
  ~~e.g. apps can interfere with each other~~
- ~~*slow startup / shutdown / reboot*~~
- *waste of resources*
  e.g. operating systems need to be replicated in each VM



History

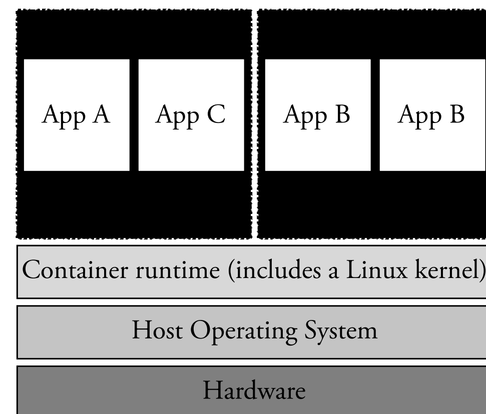
In the beginning there were **physical machines** (bare metal)...

...then **virtual machines** were created (type 1 and type 2)

Finally, **containers** were invented, by building up on key *features of Linux* kernels

Issues:

- ~~*difficult maintenance* due to dependency hell~~
  ~~e.g. apps require different versions of bins & libs~~
- ~~*low resource utilization*~~
  ~~due to limited no. of applications that can coexist~~
- ~~*isolation*~~
  ~~e.g. apps can interfere with each other~~
- ~~*slow startup / shutdown / reboot*~~
- *waste of resources*
  e.g. operating systems need to be replicated in each VM



containers

Governance structure, established in 2015, with the purpose of creating open industry standards for the concept of container.

(Docker is a specific implementation of these standards, but other implementations exist, e.g. containerd, CoreOS rkt / Rocket, Podman)

These standards concern:

- *Runtime specification*
  how to run a container image that was downloaded and unpacked into a "runtime filesystem bundle"
- *Image specification*
  specifies what a container image can/should contain, and its format, e.g. as a "serializable filesystem"
- *Distribution specification*
  how images should be distributed and downloaded, e.g. using a "container image registry"

From www.docker.com:

A **container** is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.

features of Linux kernels

namespaces

Shared system resources are
wrapped into abstractions
that make them appear as
isolated within each container.

Examples:

- *process IDs*
  e.g. a process in a container
  may appear to have PID 1

- *user IDs*
  e.g. a user ID within a container
  can be mapped to a different
  user ID in another container, and
  this may also affect privileges

cgroups

Processes can be organized
into groups with various types
of *constraints on resource usage*.

Examples:

- App A uses at most 30% of CPU
  and up to 50MB of RAM

- App B uses at most 50% of CPU,
  up to 100MB of RAM, and
  throttle reads to 10Mb/s

overlaid filesystems

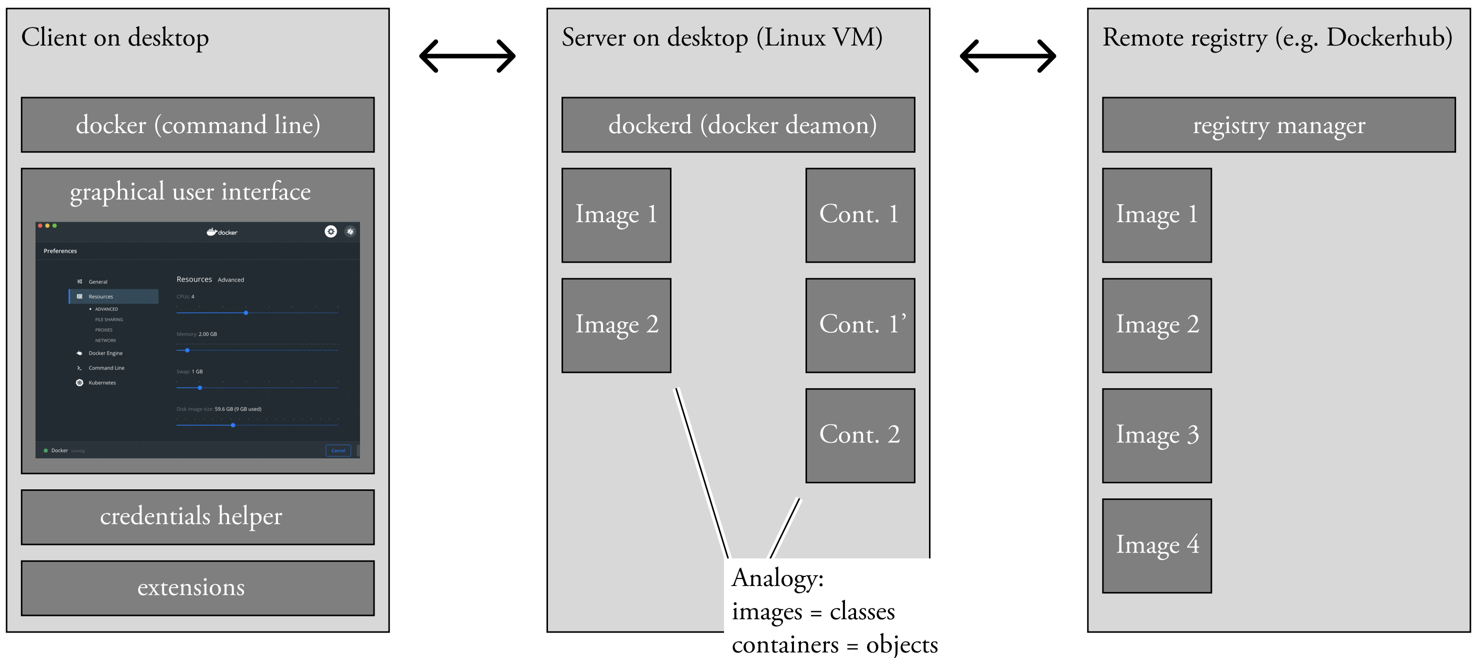
Elements of different filesystems
can be *transparently overlaid*
to form a single coherent
filesystem within each container.

In particular, this enables
the sharing of common
parts of binaries & libraries
across different containers
(so each container image will
only store differences w.r.t.
a base image).

Apartment vs house analogy

Cargo ship

Docker (desktop) architecture



Hello world

To install docker (on Linux, MacOS, Windows) go to https://docs.docker.com/get-docker and download appropriate distribution (e.g. you can rely on Windows Subsystem for Linux)

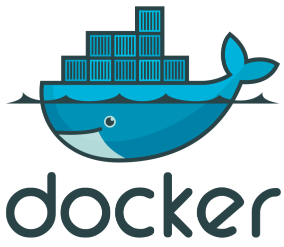
Now play with it!

create a new container from image (+ fetch and download image if not found)
Note: multiple containers can be created from the same image, each with its own ID and name

detach (run in the background)

port binding (host-to-contanier)

removes container after stop



Motivation

Without Docker:

- "To set up the development environment install Postgres, MongoDB, and run these 5 scripts. Oh wait, you are on Windows? Then also change these configurations!"
- "To deploy the application you need a server running Ubuntu. Run this Ansible playbook to install the dependencies and configure the system, then copy the binary and run it with these options."

With Docker:

- "docker compose up"
- "docker run [options] container_image"

History

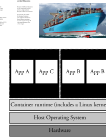
In the beginning there were **physical machines** (bare metal)...

...then **virtual machines** were created (type 1 and type 2)

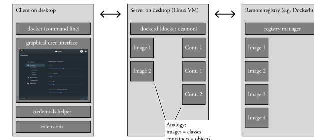
Finally, **containers** were invented, by building up on key *features of Linux* kernels

Issues:

- ~~*difficult maintenance* due to dependency hell~~
  ~~e.g. apps require different versions of bins & libs~~
- ~~*low resource utilization*~~
  ~~due to limited no. of applications that can coexist~~
- ~~*isolation*~~
  ~~e.g. apps can interfere with each other~~
- ~~*slow startup / shutdown / reboot*~~
- *waste of resources*
  e.g. operating systems need to be replicated in each VM



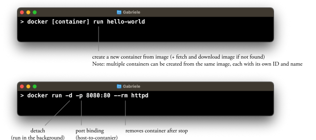
Docker (desktop) architecture



Hello world

To install docker (on Linux, MacOS, Windows) go to https://docs.docker.com/get-docker and download appropriate distribution (e.g. you can rely on Windows Subsystem for Linux)

Now play with it!



Persistence of data

Composing

Networking

Persistence of data

Data created or modified in a container is not persistent: it is lost if container is stopped or deleted!

2 mechanisms to enforce persistence

Save data to external volume or DB

Commit modifications to new image

Save data to external volume or DB

This is common for data generated by applications (e.g. user metadata, logs, etc.).
However, if another container is created from the initial image, the generated data
is not immediately available (external volume/DB needs to be “re-connected” to new container)

Gabriele

```
> docker run -it ubuntu
```

interactive — terminal

Gabriele

```
root@1efd184868d2:/# mkdir /myfolder
root@1efd184868d2:/# echo “Hello” > /myfolder/myfile.txt
root@1efd184868d2:/# exit
```

volume creation (folder on Linux VM) — binds *volume*

Gabriele

```
> docker volume create myvolume
> docker run -it -v myvolume:/myfolder ubuntu
> docker run -it -v ~/hostfolder:/myfolder ubuntu
```

binds *folder on host* (slower than volume)

```
> docker run -it ubuntu
```

interactive — terminal

```
root@1efd184868d2:/# mkdir /myfolder
root@1efd184868d2:/# echo "Hello" > /myfolder/myfile.txt
root@1efd184868d2:/# exit
```

volume creation (folder on Linux VM) — binds *volume*

```
> docker volume create myvolume
> docker run -it -v myvolume:/myfolder ubuntu
> docker run -it -v ~/hostfolder:/myfolder ubuntu
```

binds *folder on host* (slower than volume)

We can also bind volumes / host folders to container folders that store databases:

environment variables (e.g. username, password, db)

```
> docker run -d --rm -e POSTGRES_PASSWORD=foo -e POSTGRES_DB=mydb
       -p 5432:5432 -v ~/hostfolder:/var/lib/postgresql/data postgres
> docker exec -it containerID bash    // then psql
```

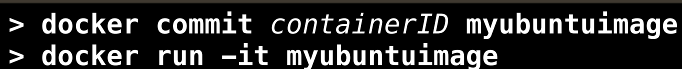
where PostgreSQL stores its DB data

then psql

```
> psql -U postgres -d mydb

    # create a table
    CREATE TABLE mytable (id INT, name VARCHAR(255));

    # populate table
    INSERT INTO mytable (id, name) VALUES (1, 'Alice');
    INSERT INTO mytable (id, name) VALUES (2, 'Bob');
    INSERT INTO mytable (id, name) VALUES (3, 'Charles');

    # query table
    SELECT * FROM mytable;
```

Commit modifications to new image

This is used when we modified a container by installing new binaries or libraries, and we want to incorporate those modifications in a new image



```
> docker run -it ubuntu
```

```
# ping google.com
# apt update && apt install nettools iputils-ping
# exit
```

```
> docker commit containerID myubuntuimage
> docker run -it myubuntuimage
```

We can also build a new image with a script file called "**dockerfile**" (without extension), just like a makefile

image name (tag)

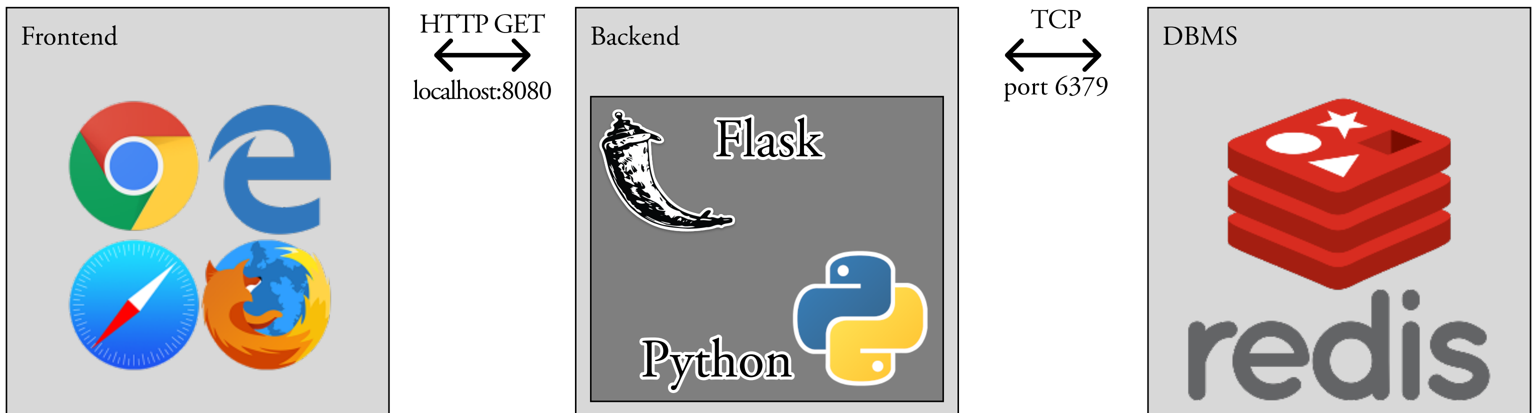
```
> docker build -t myimage
```

dockerfile

```
# Use “ubuntu” as base image
FROM ubuntu

# Install additional binaries
RUN apt-get update && \
    apt-get install -y iputils-ping && \
    apt-get clean
```

Composing

Let us try to combine components and build a simple RESTful web service…



Python

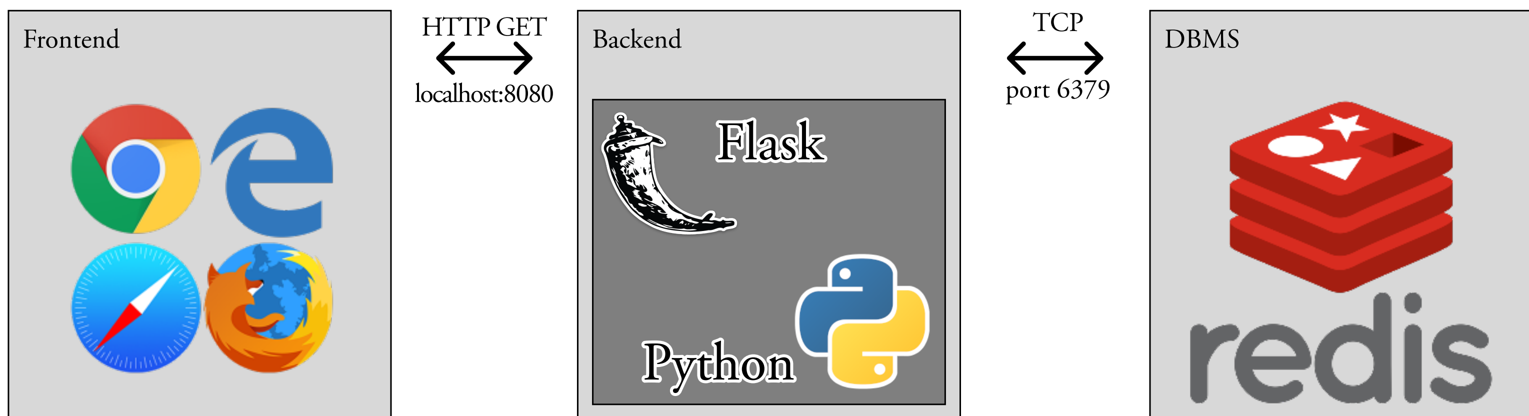
```python
import flask                    # simple framework to develop web-based apps
import flask_restful            # extension to implement RESTful services
import redis                    # in-memory storage

flask_app = flask.Flask(__name__)
flask_api = flask_restful.Api(flask_app)

# redis will run on a separate container, so the host is the name of
# that container (see compose.yaml), instead of the default "localhost",
# and the port is the default 6379
redis_db = redis.Redis(host='my_redis_db')

# the following class will handle GET requests at URL '/'
class Counter(flask_restful.Resource):
    def get(self):        # callback function for GET requests
        hits = redis_db.incr('hits')
        return 'I have been hit {} times.\n' . format(hits)

flask_api.add_resource(Counter, '/')
```

Next prepare a script **my_flask_app/dockerfile** that builds an image containing Python and Flask + Redis libraries:

```
# start with base image containing python
FROM python

# extend python with flask, flask_restful, and redis libraries
RUN pip install flask flask_restful redis

# set working folder on image to "/src" and copy app.py there
WORKDIR /src
COPY ./app.py .

# this container will be listening on port 5000 (Flask web service)
EXPOSE 5000

# tell Flask to run app.py at startup, and
# set default container command to "flask run"
ENV FLASK_APP app.py
ENV FLASK_RUN_HOST 0.0.0.0
CMD ["flask", "run"]
```

Composing

Let us try to combine components and build a simple RESTful web service…




```python
import flask                    # simple framework to develop web-based apps
import flask_restful            # extension to implement RESTful services
import redis                    # in-memory storage

flask_app = flask.Flask(__name__)
flask_api = flask_restful.Api(flask_app)

# redis will run on a separate container, so the host is the name of
# that container (see compose.yaml), instead of the default "localhost",
# and the port is the default 6379
redis_db = redis.Redis(host='my_redis_db')

# the following class will handle GET requests at URL '/'
class Counter(flask_restful.Resource):
    def get(self):       # callback function for GET requests
        hits = redis_db.incr('hits')
        return 'I have been hit {} times.\n' . format(hits)

flask_api.add_resource(Counter, '/')
```

Finally, we merge!

Next prepare a script **my_flask_app/dockerfile** that builds
an image containing Python and Flask + Redis libraries:

```
# start with base image containing python
FROM python

# extend python with flask, flask_restful, and redis libraries
RUN pip install flask flask_restful redis

# set working folder on image to "/src" and copy app.py there
WORKDIR /src
COPY ./app.py .

# this container will be listening on port 5000 (Flask web service)
EXPOSE 5000

# tell Flask to run app.py at startup, and
# set default container command to "flask run"
ENV FLASK_APP app.py
ENV FLASK_RUN_HOST 0.0.0.0
CMD ["flask", "run"]
```

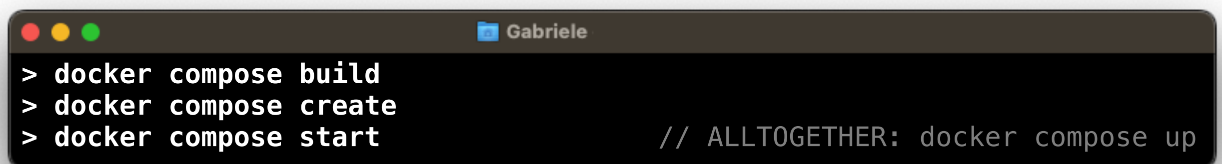
Finally, we merge!

```yaml
services:

  my_flask_app:
    build: ./my_flask_app/
    ports:
      - 8080:5000

  my_redis_db:
    image: redis
```

This is a **compose.yaml** file, which represents data just like .json but with indentation instead of { }, and - instead of [...] (less clutter)

It tells docker-compose program to do the following:

- build image my_flask_app using the previous dockerfile script
- download Redis image from Docker hub (no build needed)
- combine the two images into a multi-container service
  (the resulting package will have the name of the working folder)



Networking

Containers exist inside specific **networks**, which help isolating services or connecting them together.

Docker provides built-in networks...

Docker provides built-in networks...

Gabriele

```
> docker network ls

NETWORK ID     NAME     DRIVER   SCOPE
68c87f0309cf   bridge   bridge   local
8596a9fa7c73   host     host     local
028d1cdc4d1a   none     null     local
```

containers are connected to this network by default and they can communicate inside the same Docker VM

any container connected to this network shares the network namespace with the host machine (e.g. no need to map ports)

a container connected to this network is isolated from the rest of the world

Let us play with two containers within the same network…

```
> docker run -dit --name ubuntu1 ubuntu
> docker exec -it ubuntu1 bash
  # apt update && apt install nettools iputils-ping
  # exit

> docker commit ubuntu1 myubuntuimage

> docker run -dit --name ubuntu2 myubuntuimage

> docker network inspect bridge

> docker exec -it ubuntu1 bash
  # ping 172.17.0.3
```

we start a container from ubuntu image and we install binaries for networking

we commit the modifications to a new image

we run another contained from the new image

both containers run within the same *bridge* network (take note of their IPs)

we can ping one container from another but none is pingable from the host machine

We can also isolate containers, or connect to other networks…

```
> docker network disconnect bridge ubuntu1
> docker network connect none ubuntu1
```

Networking

Containers exist inside specific **networks**, which help isolating services or connecting them together.

Docker provides built-in networks...

...and custom networks can also be created

```
> docker network ls

NETWORK ID     NAME     DRIVER   SCOPE
68c87f0309cf   bridge   bridge   local
8596a9fa7c73   host     host     local
028d1cdc4d1a   none     null     local
```

containers are connected to this network by default
and they can communicate inside the same Docker VM

any container connected to this network
shares the network namespace with the
host machine (e.g. no need to map ports)

a container connected to this network
is isolated from the rest of the world

Let us play with two containers within the same network...

```
> docker run -dit --name ubuntu1 ubuntu
> docker exec -it ubuntu1 bash
  # apt update && apt install nettools iputils-ping
  # exit

> docker commit ubuntu1 myubuntuimage

> docker run -dit --name ubuntu2 myubuntuimage

> docker network inspect bridge

> docker exec -it ubuntu1 bash
  # ping 172.17.0.3
```

we start a container from ubuntu image
and we install binaries for networking

we commit the modifications to a new image

we run another contained from the new image

both containers run within the same *bridge* network
(take note of their IPs)

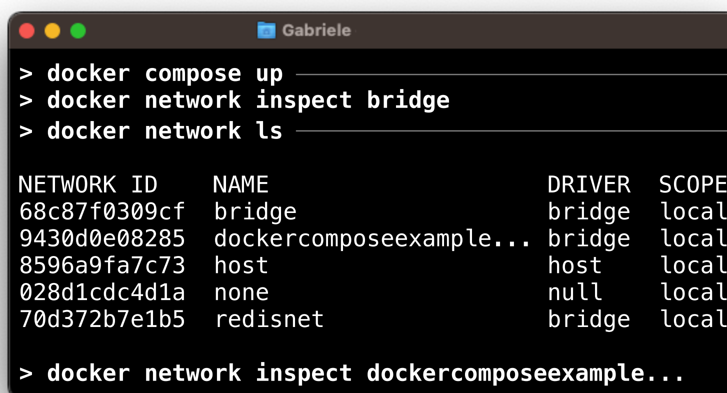
we can ping one container from another
but none is pingable from the host machine

We can also isolate containers, or connect to other networks...

```
> docker network disconnect bridge ubuntu1
> docker network connect none ubuntu1
```

...and custom networks can also be created

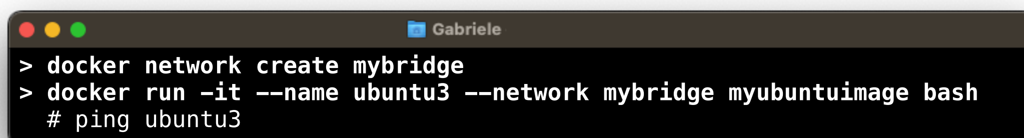
For example, when using *docker compose*, the service is automatically created with a custom network

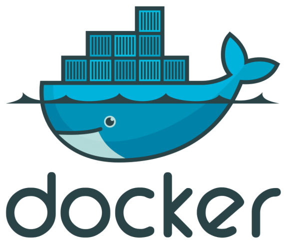


start service from our previous example, service will not be in the *bridge* network

however a new network was created, which contains the service components

Custom networks can also be created and configured manually (this will enable, for instance, DNS lookup)





Motivation

Without Docker:

- "To set up the development environment install Postgres, MongoDB, and run these 5 scripts. Oh wait, you are on Windows? Then also change these configurations!"
- "To deploy the application you need a server running Ubuntu. Run this Ansible playbook to install the dependencies and configure the system, then copy the binary and run it with these options."

With Docker:

- "docker compose up"
- "docker run [options] container_image"

History

In the beginning there were **physical machines** (bare metal)...

...then **virtual machines** were created (type 1 and type 2)

Finally, **containers** were invented, by building up on key *features of Linux* kernels

Issues:

- ~~*difficult maintenance* due to dependency hell~~
  ~~e.g. apps require different versions of bins & libs~~
- ~~*low resource utilization*~~
  ~~due to limited no. of applications that can coexist~~
- ~~*isolation*~~
  ~~e.g. apps can interfere with each other~~
- ~~*slow startup / shutdown / reboot*~~
- *waste of resources*
  e.g. operating systems need to be replicated in each VM

Cargo ship

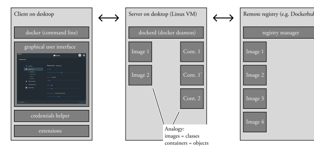
App A | App C | App B | App B

Container runtime (includes a Linux kernel)

Host Operating System

Hardware

Docker (desktop) architecture



Hello world

To install docker (on Linux, MacOS, Windows) go to https://docs.docker.com/get-docker
and download appropriate distribution (e.g. you can rely on Windows Subsystem for Linux)

Now play with it!

```
> docker [container] run hello-world
```

create a new container from image (+ fetch and download image if not found)
Note: multiple containers can be created from the same image, each with its own ID and name

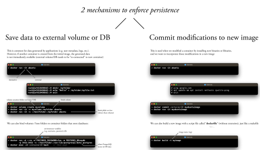
```
> docker run -d -p 8080:80 --rm httpd
```

detach (run in the background) — port binding (host-to-contanier) — removes container after stop

Persistence of data

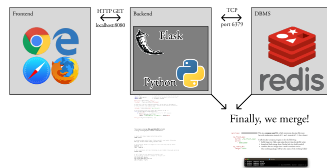
Data created or modified in a container is not persistent: it is lost if container is stopped or deleted!

2 mechanisms to enforce persistence



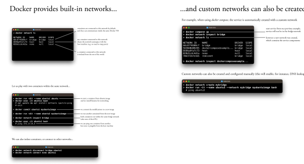
Composing

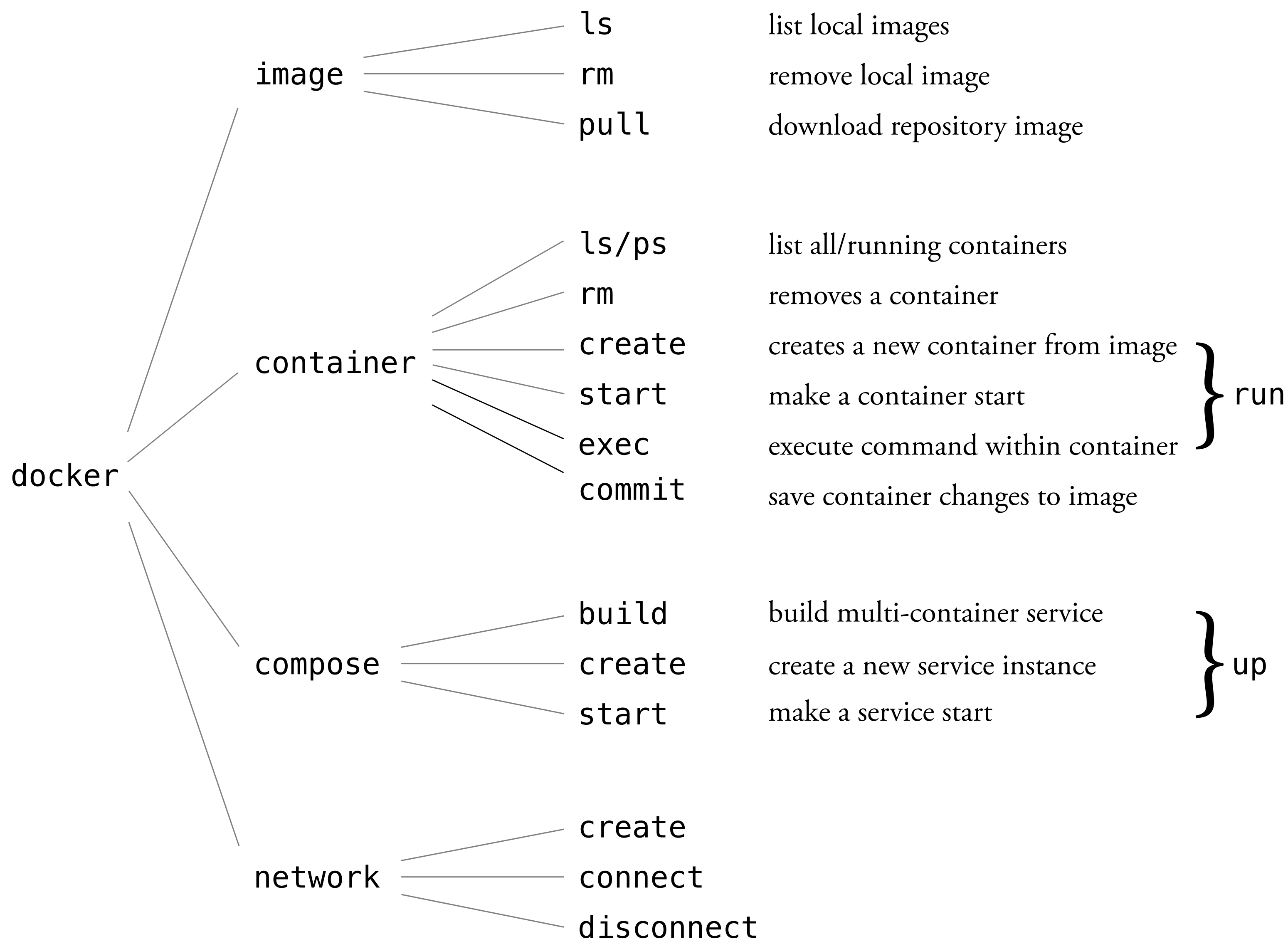
Let us try to combine components and build a simple RESTful web service...



Networking

Containers exist inside specific **networks**, which help isolating services or connecting them together.




docker
image
ls
list local images
rm
remove local image
pull
download repository image
container
ls/ps
list all/running containers
rm
removes a container
create
creates a new container from image
start
make a container start
exec
execute command within container
commit
save container changes to image
run
compose
build
build multi-container service
create
create a new service instance
start
make a service start
up
network
create
connect
disconnect