Operating System Management – 4

Virtualization & containers

Samuel Pélissier
(samuel.pelissier@centralesupelec.fr)

| 2026

CentraleSupélec | université PARIS-SACLAY

Virtualization • Containers 101 • Docker • Orchestration • Conclusion

Today's menu

1. Virtualization
2. Containers 101
3. Docker
4. Orchestration
5. Conclusion

Goals:

- Understand why we need virtualization
- Understand the difference between virtual machines & containers
- Know what **is** a container
- Know how to build, configure, and run containers

Today's menu

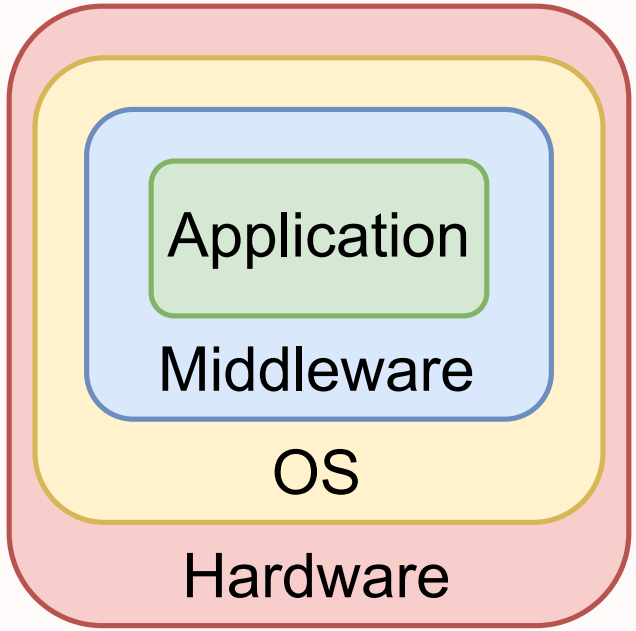
We have multiple problems:

- "It runs on my machine"
  - Different hardware architectures
  - Different softwares
  - Software install / packaging is **hard** (dependency hell)
- "I just want to use this application for 5 minutes"
  - Start/stop the application only when it's needed

Let's see how we can try to solve them...

Resource management

Reminder: a simple architecture

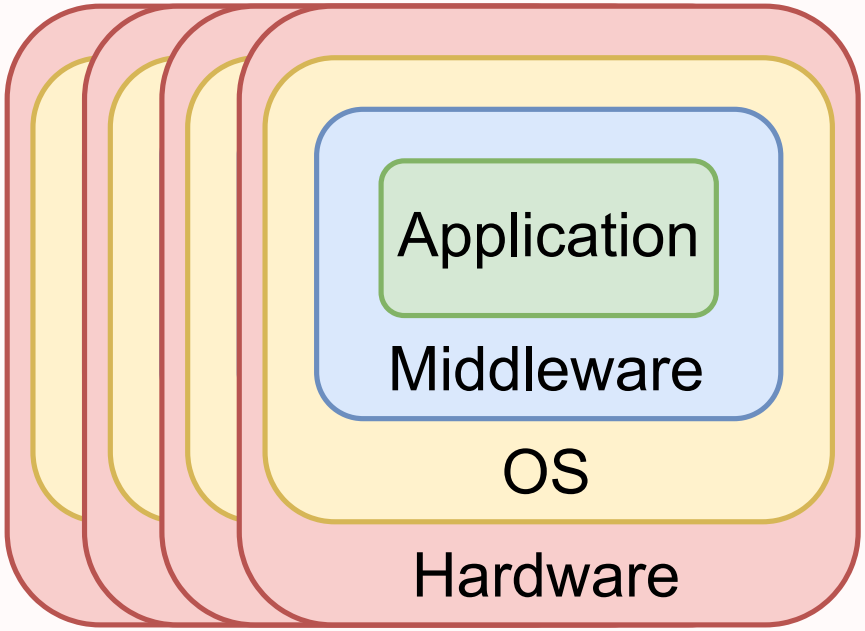


Resource management

What if we need more resources?

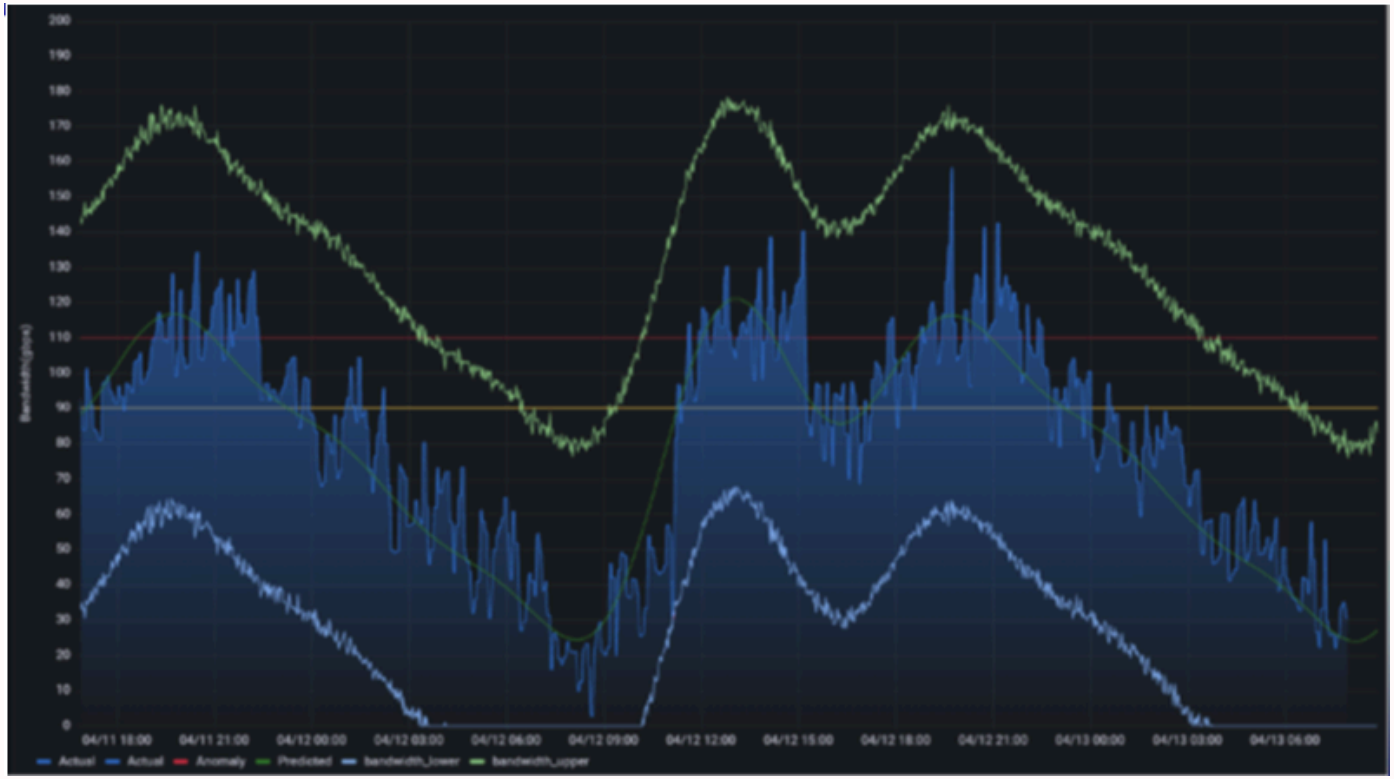
Resource management

What if we need more resources?



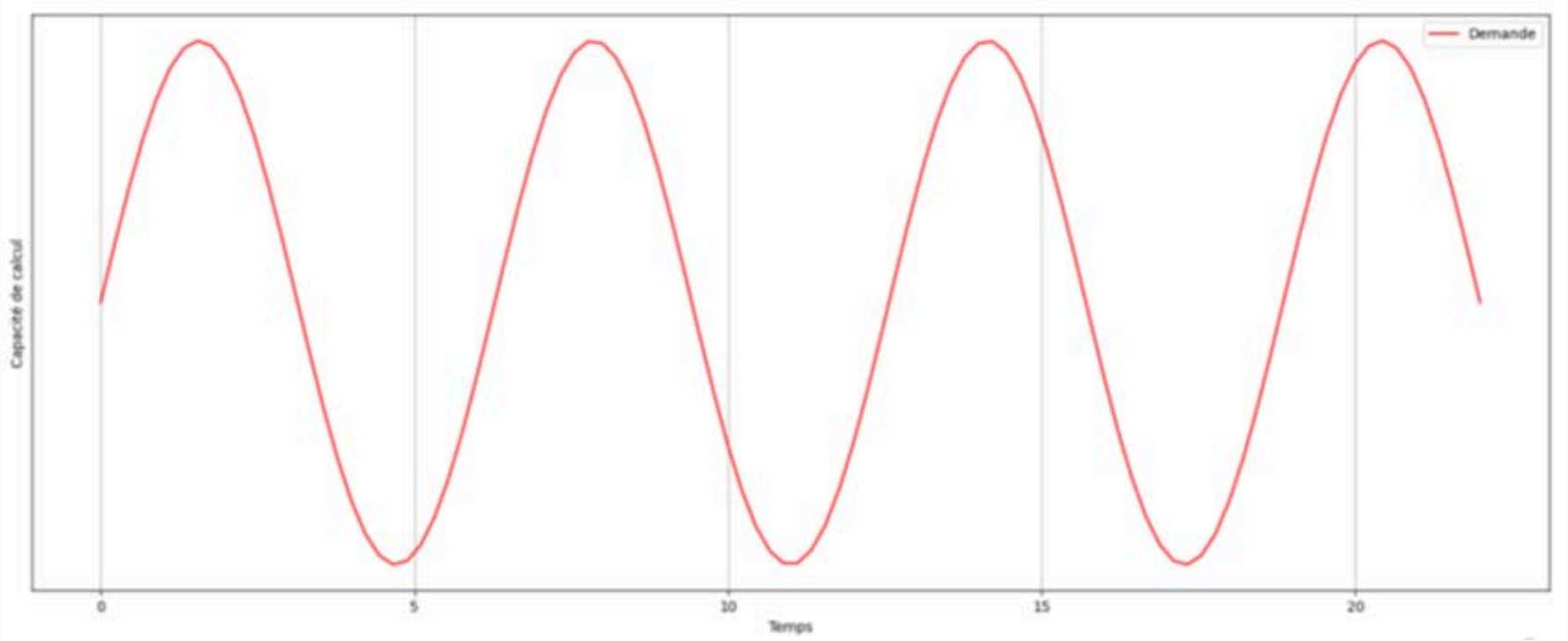
Resource management

Usage change over time:



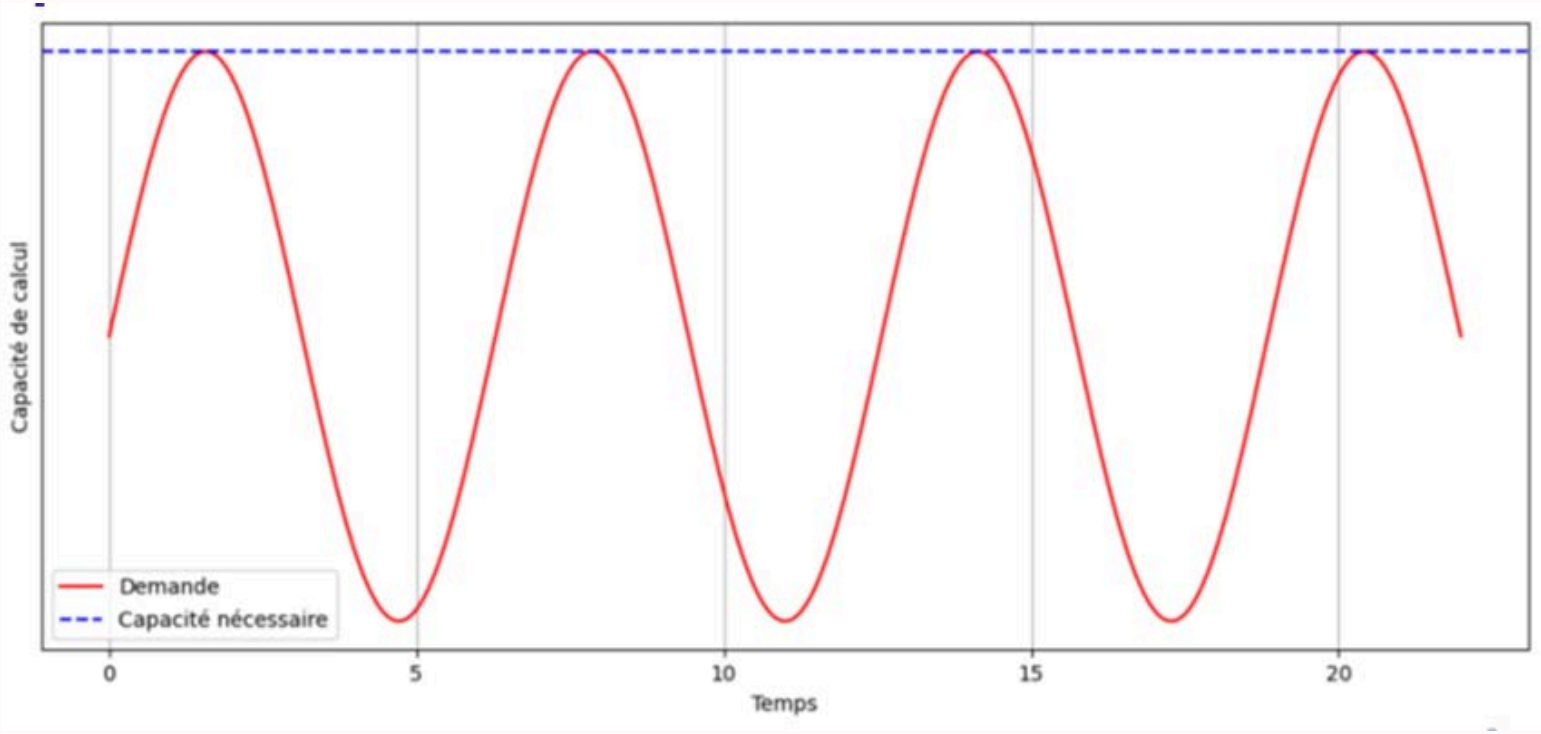
Resource management

Varying demand



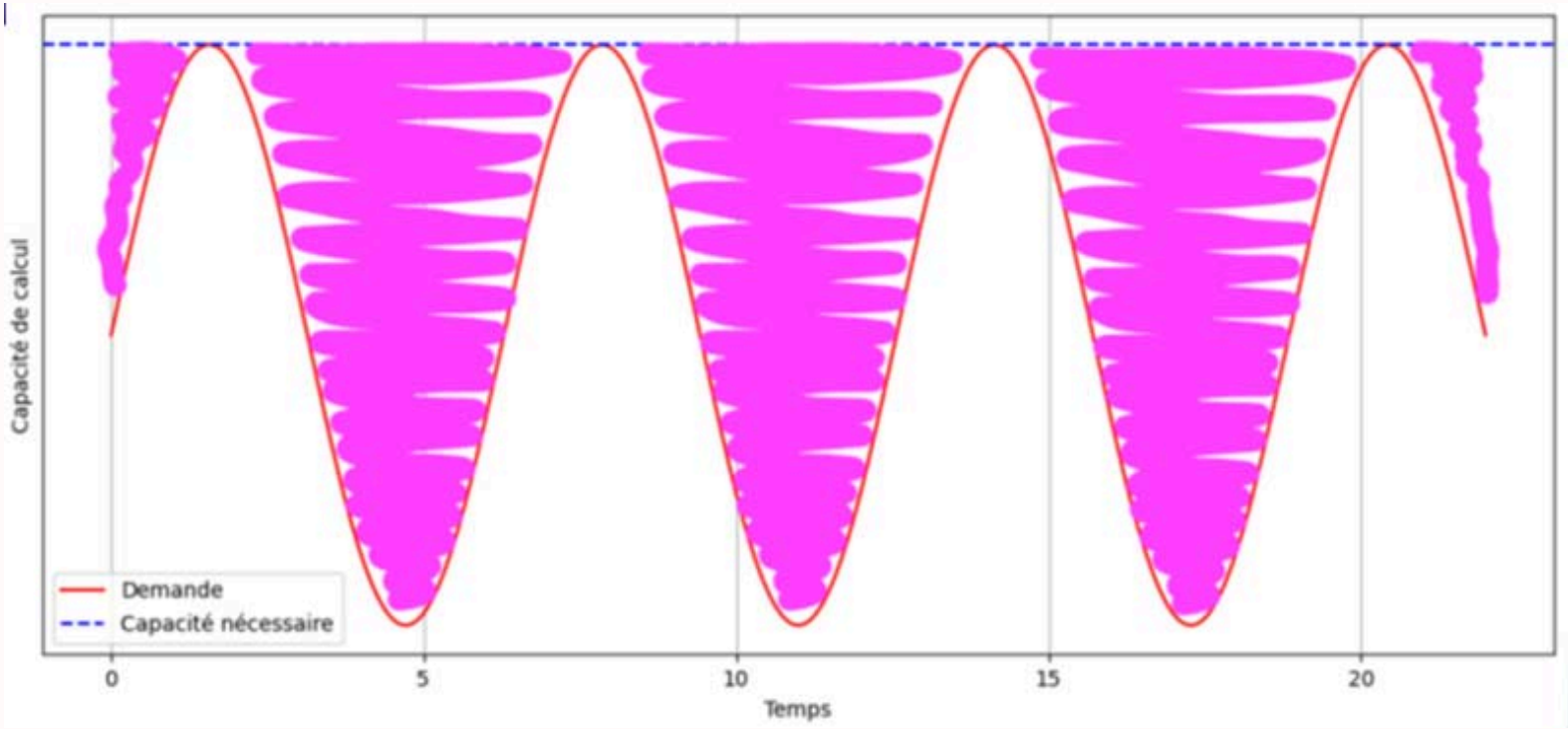
Resource management

Actual capacity



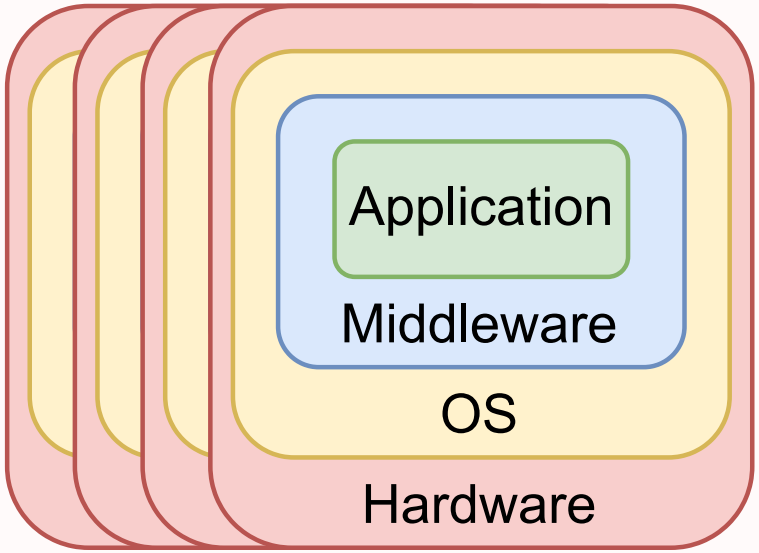
Resource management

This corresponds to lost / underused resources



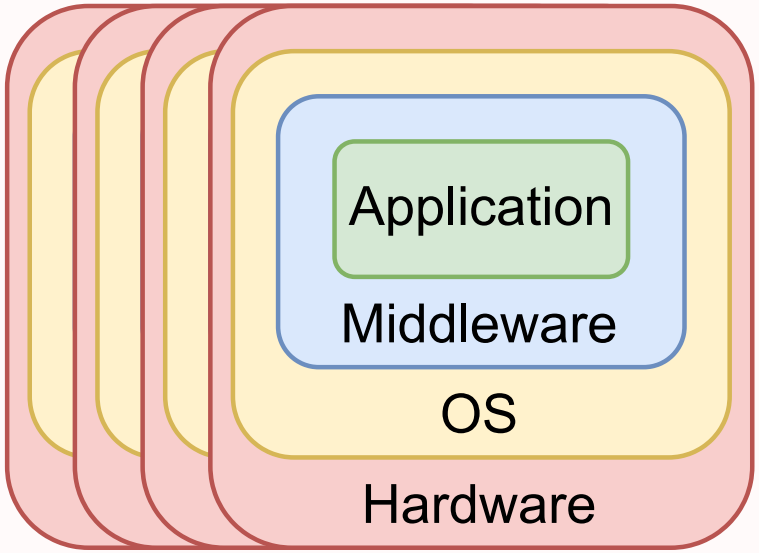
Resource management

- Hard to follow variations, capacity are fixed
- Manual resource management

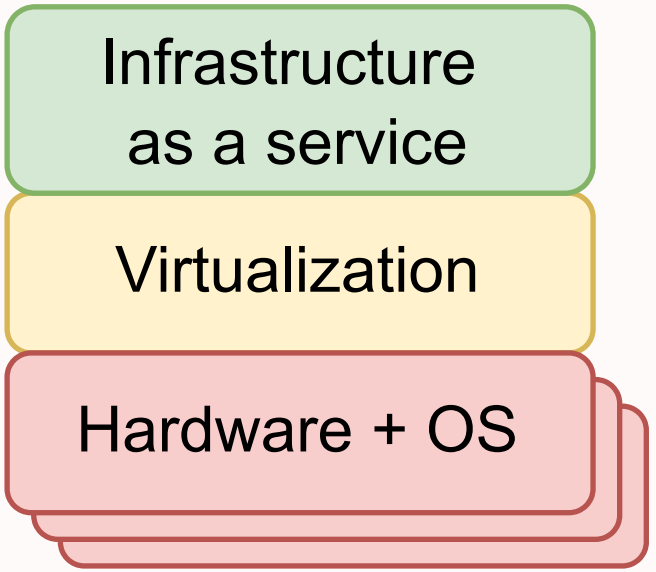


Resource management

- Hard to follow variations, capacity are fixed
- Manual resource management



- Dynamic, on demand resources
- Automatic resource management
- Pay for what you use

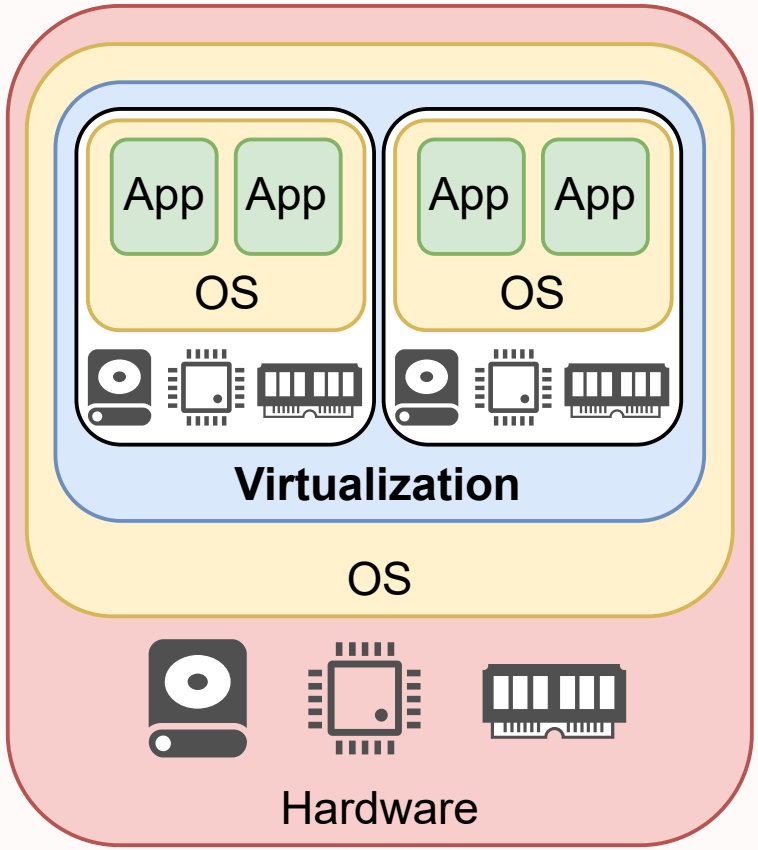


IAAS: definition

- “computing”?
  - A **service** that clients use
  - No physical access to hardware
  - Hardware geographical location is (almost) irrelevant
- The Cloud™ hides the complexity away
  - Users only use an API
- Services are available on demand
  - Anytime, anywhere
- Pay-per-use
  - Users pay only for what they use. They can easily discard a resource if not needed anymore.

Virtualization

- Virtualization = a layer above the OS
- Used to run other operating systems

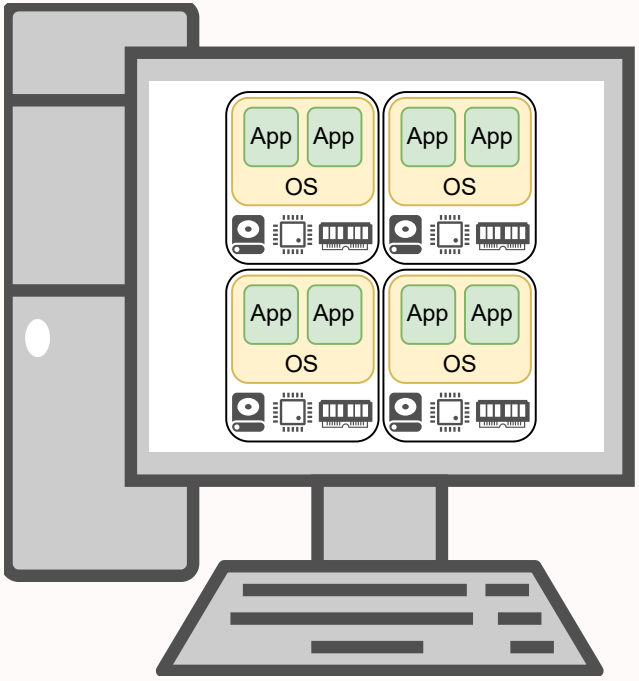


Virtualization

- Hardware-Level Abstraction
  - Virtual hardware: processors, memory, chipset, input/output buses and peripherals, etc.
  - Encapsulates all OSes and application states
- Virtualization Software
  - Additional layer of abstraction to decouple hardware and the OS
  - Hardware multiplexing among several VMs
  - Strong isolation between VMs
  - Management of physical resources to optimize usage

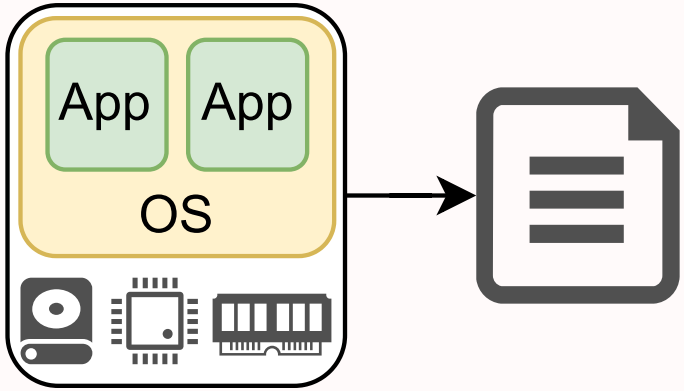
Virtual machines isolation

- Secure Multiplexing
  - Operation of multiple VMs on the same physical machine
  - The processor provides mechanisms to isolate VMs (the MMU)
- Strong Guarantees
  - Crashes, bugs, or viruses in one VM cannot affect the other VMs
- Performance Isolation
  - Partitioning of system resources
  - The virtualization layer controls reservations, limits, and sharing



Encapsulation

- A VM is a file
  - OS, applications, data
  - Memory and device state



- Snapshots and clones
  - Capture the state of a VM on the fly and restore it at the moment of capture
  - Fast system provisioning, backup, remote duplication
- Content distribution
  - Preconfigured applications, demos
  - “Virtual Appliance”: VM image containing the entire environment for running one or more services

Compatibility

- Hardware-independent
  - The hardware is hidden by the virtualization layer
  - A “standard” virtual hardware is exposed to the VM
- Created once, run everywhere
  - No configuration issues
  - Allows VM migration between hosts
- Legacy support
  - Ability to run older OSes on a new platform

Limitations

- A VM is heavy
  - The hypervisor adds overhead on the server
  - Each guest OS requires a lot of memory
  - Each OS runs many services
  - Boot times can be long (on the order of a minute)
  - On a single machine, it is not possible to run a large number of VMs

Containers!

why containers?

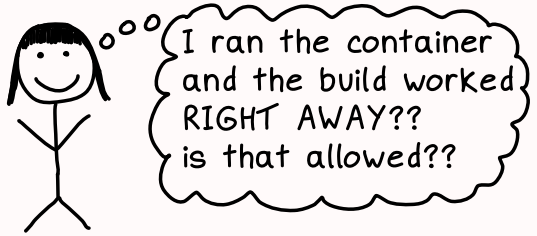
there's a lot of container hype

Here are 2 problems they solve...

problem: building software is annoying

```
$ ./configure
$ make all
ERROR: you have version
       2.1.1 and you need
       at least 2.2.4
```

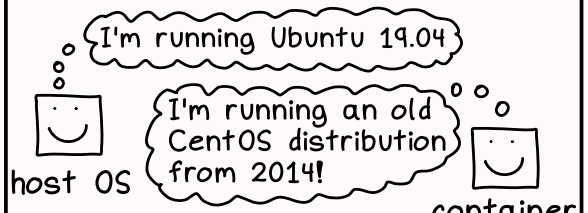
solution: package all dependencies in a ★container★



Many CI systems use containers.

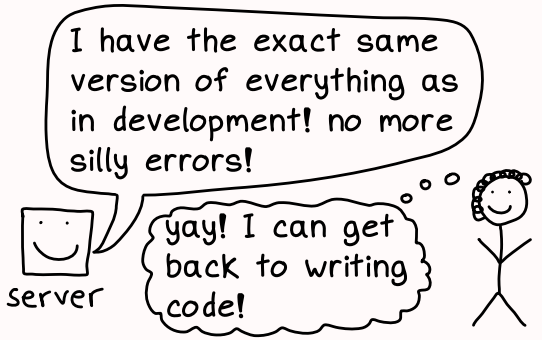
containers have their own filesystem

This is the big reason containers are great.



problem: deploying software is annoying too

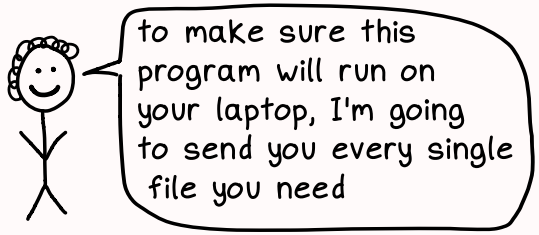
solution: deploy a container



Author: Julia Evans

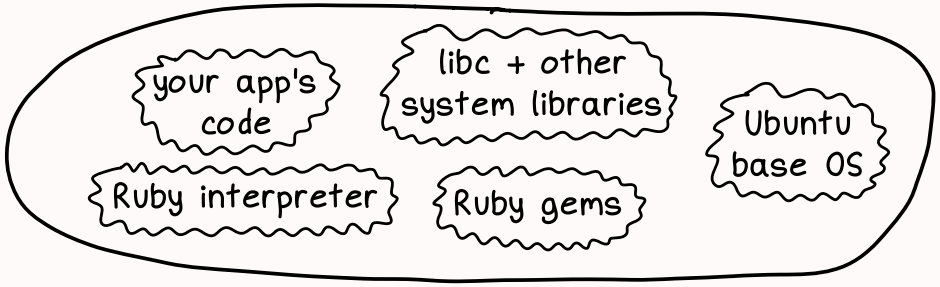
the big idea: include EVERY dependency

containers package EVERY dependency together



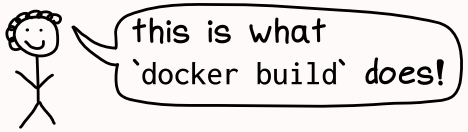
a container image is a tarball of a filesystem

Here's what's in a typical Rails app's container:



how images are built

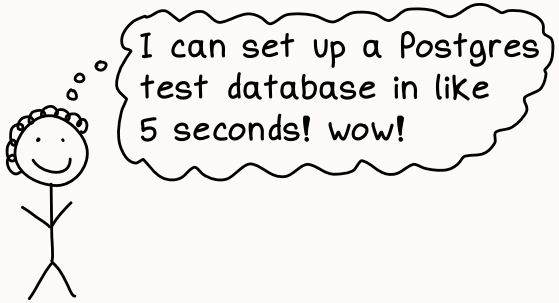
0. start with a base OS
1. install program + dependencies
2. configure it how you want
3. make a tarball of the WHOLE FILESYSTEM



running an image

1. download the tarball
2. unpack it into a directory
3. run a program and pretend that directory is its whole filesystem

images let you "install" programs really easily

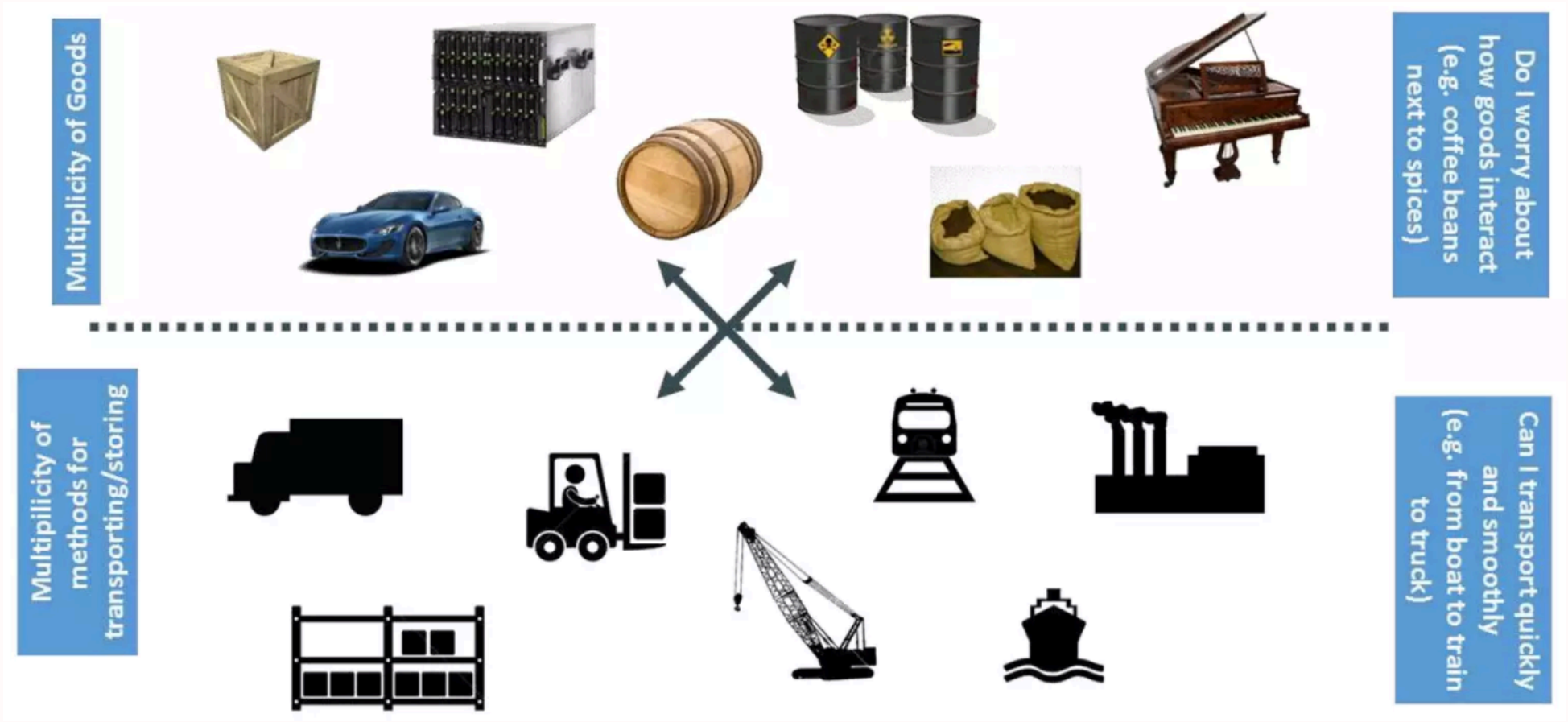


Author: Julia Evans

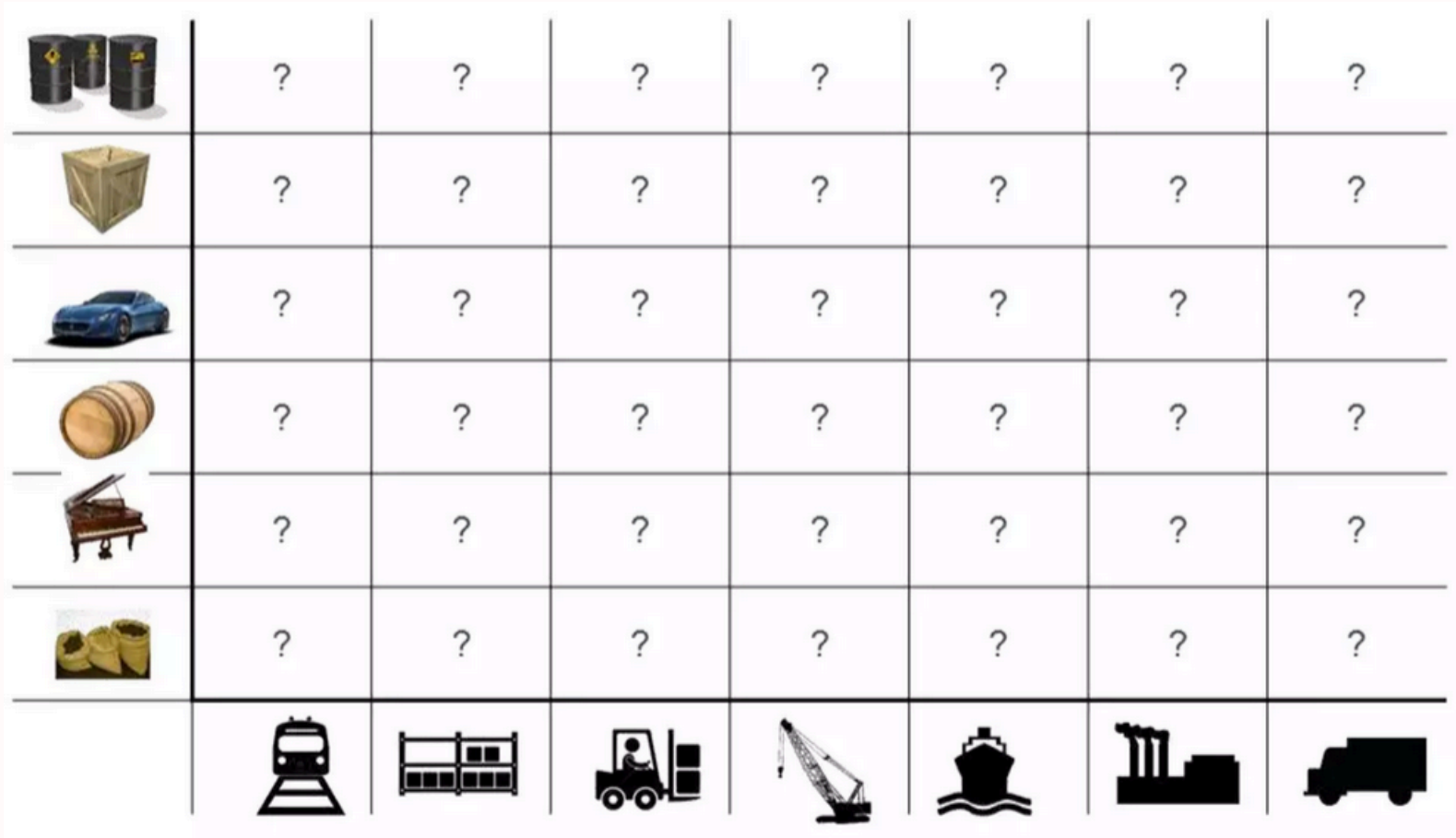
Origins: why “containers”? The Matrix From Hell

	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

Origins: Real World Cargo Transport pre-1960

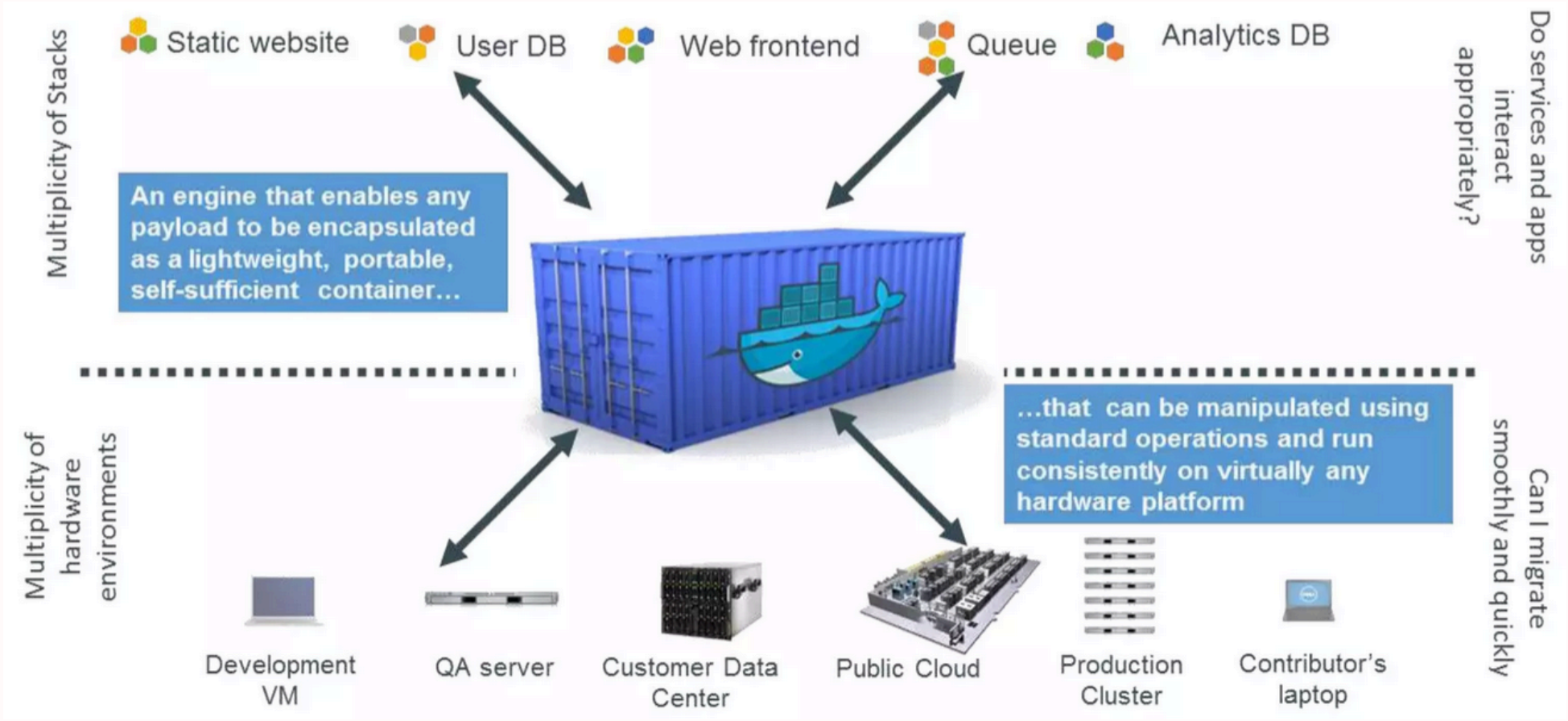


Origins: Another Matrix From Hell

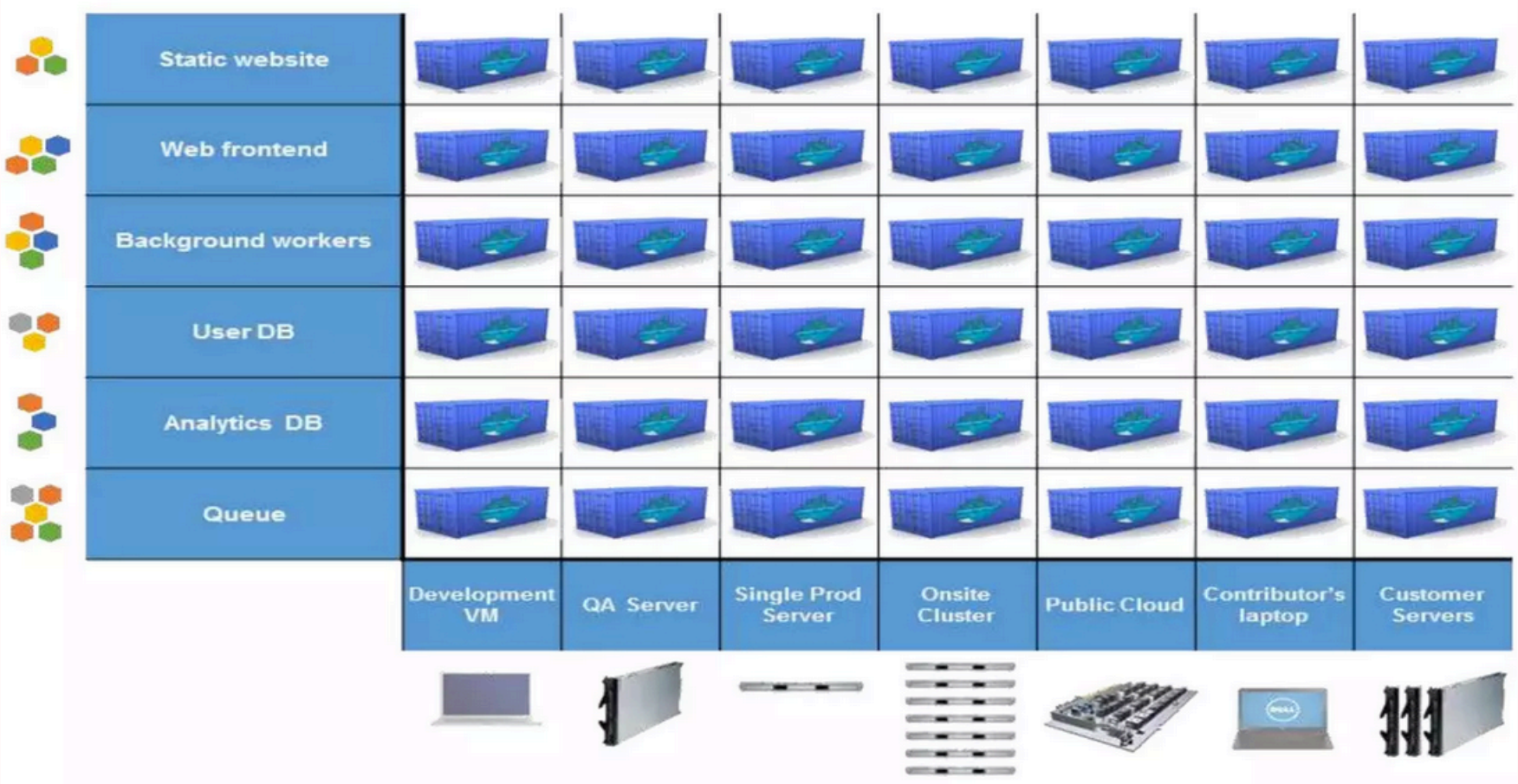


Origins: Solution: Intermodal Shipping Container

Origins: Solution: a Container System for Code



Origins: No More Matrix From Hell :)

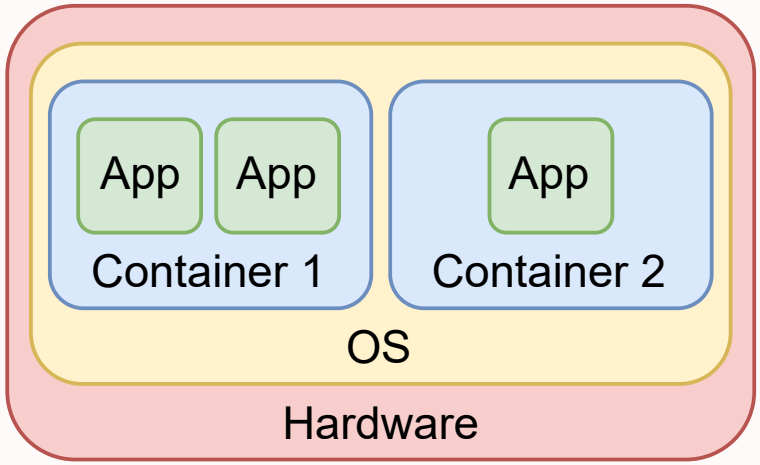


Containers are...

- A “place” to run one or multiple processes in an environment
  - Specificity: isolated and potentially limited/constrained environments
- Tools to install and configure software in this environment
  - Image build instruction files, volumes, environment variables, etc.
- A method to manage different images
  - Image management (deployment model) and distribution

Containers are not...

- Virtualization in the general sense
  - Not a full OS, therefore no OS boot time, no "resource waste," much smaller image size
- A "virtualization layer"
  - The host system is isolated from the container
  - Container resources are isolated



Other examples...

... that are not containers but provide some sort of isolation.

- The JVM
  - Java code can be executed on very different machines because the code is interpreted by an isolated program, the Java Virtual Machine
- Python virtual environments
  - Python code can be executed by changing the directories where the function libraries are located
  - Allows having projects that use incompatible libraries
- Sandboxes
  - Isolated environments for running untrusted programs

Building blocks of a container

Containers are just processes with some restrictions enforced by the Linux kernel.

Main features required to build a container:

- Isolation
  - OS-level virtualization
  - Namespaces
  - Chroot
  - Capabilities
  - Seccomp (restrict system calls)
- Resource restrictions
  - Cgroups

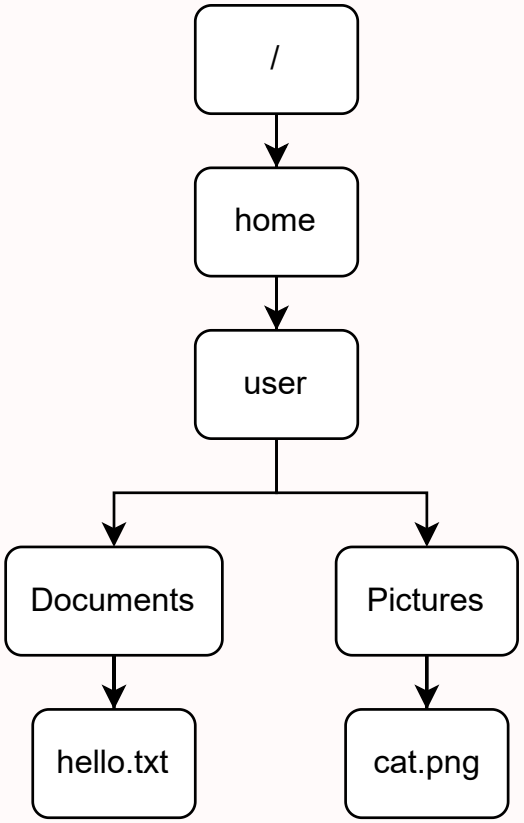
Building blocks of a container: OS-level virtualization

- Run processes in an isolated environment
  - Virtual memory: each container believes it has its own physical memory (like other processes)
  - Process isolation: processes belonging to one container are isolated from those of other containers
  - User mode / Kernel mode differentiation
  - System calls: controlled so they do not disturb the host system (seccomp-bpf)
- Use of **namespaces**
  - Linux kernel mechanism that isolates processes, filesystem, network, user IDs, etc.
  - Deny an operation: prevent access to a resource

Building blocks of a container: chroot

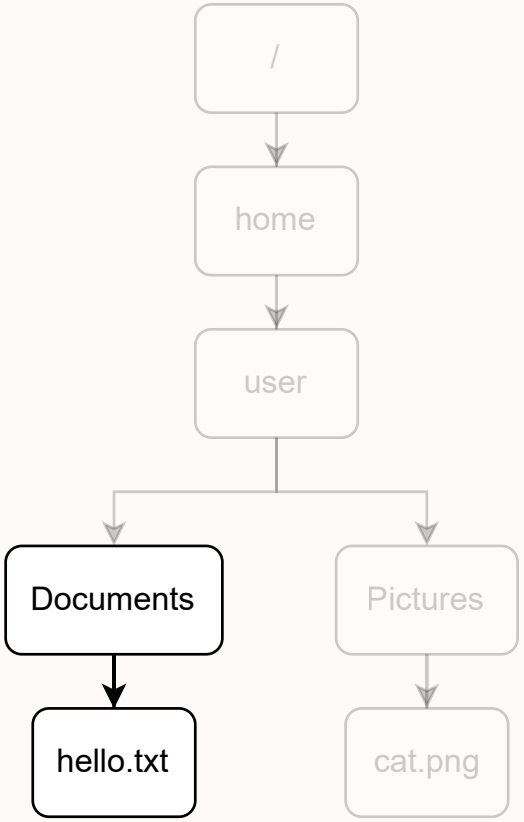
- The `chroot` command changes the root directory of the system for the current process and its children
- Isolation because it separates the filesystem of the container
- Isolation is not complete: a user with sufficient privileges can “escape” this environment

Building blocks of a container: chroot

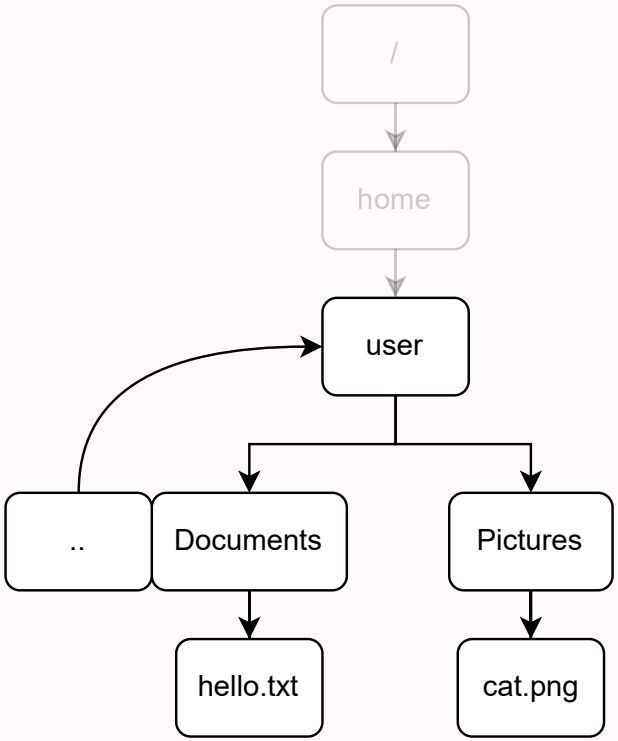


Building blocks of a container: chroot

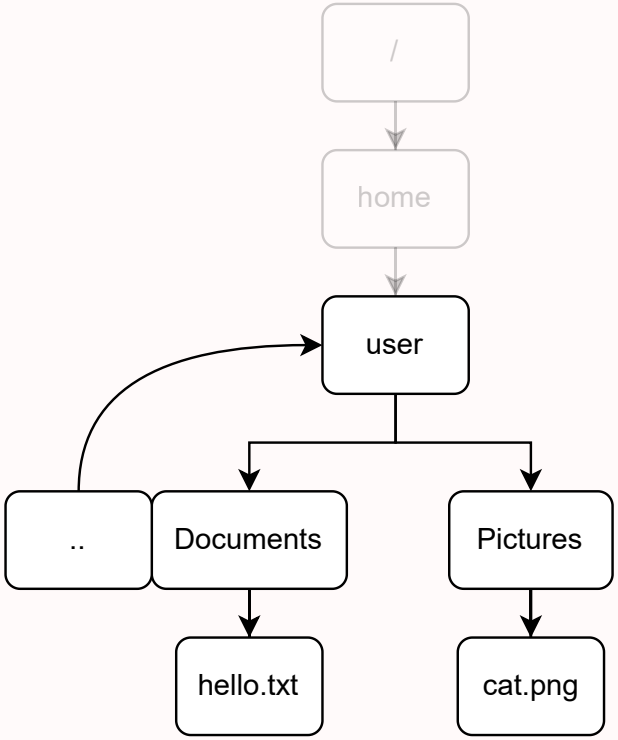
```
chroot /home/user/Documents/ /bin/bash
```



Building blocks of a container: chroot



Building blocks of a container: chroot



In practice, modern containers leverage a combination of pivot_root and namespaces, which makes it impossible to get out of /

Building blocks of a container: capabilities

Sorry, you got lied to (again)

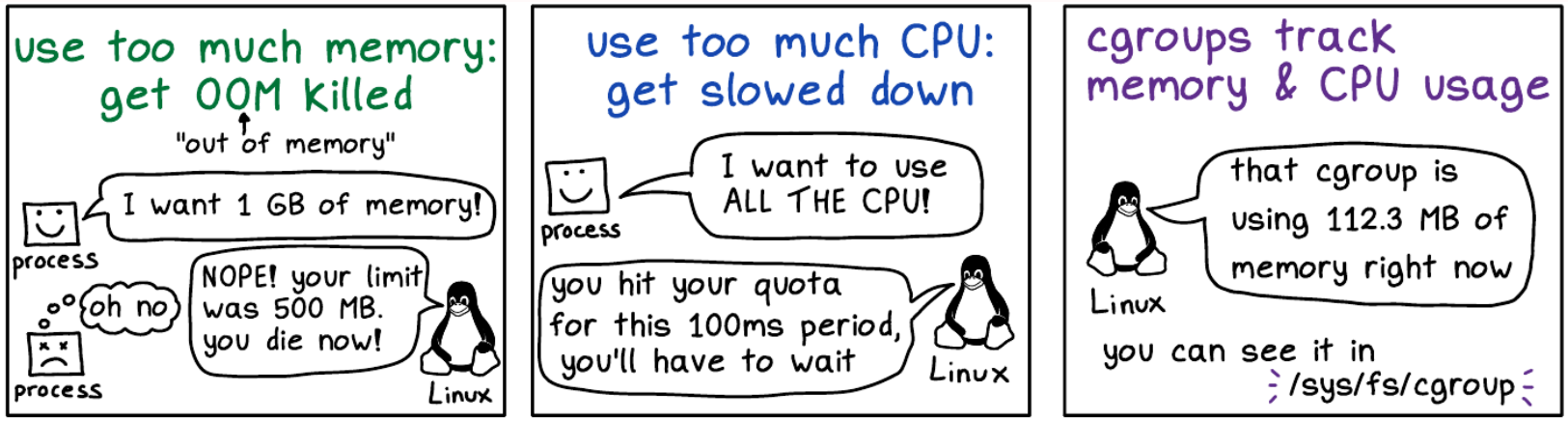
- `root` is not All Powerful
- Processes need the right **capabilities** to do stuff

Building blocks of a container: capabilities

- Linux kernel security mechanism
- Confines application execution according to the principle of least privilege
- Allows adding or removing permissions for certain functionalities
- Examples:
  - Changing identity
  - Opening listening sockets on reserved ports
  - Sending signals
  - Restricting filesystem access
  - ...
- Represented in a set:
  - Permitted
  - Inheritable
  - Effective

Building blocks of a container: cgroups

- A Linux feature that **groups** processes together
- Limit access to resources
- Account resource usage
- Apply priorities
- Create checkpoints
- Restore groups



Note: namespaces = isolation, cgroups = resource limits

containers aren't magic

These 15 lines of bash will start a container running the fish shell. Try it!

(download this script at bit.ly/containers-arent-magic)

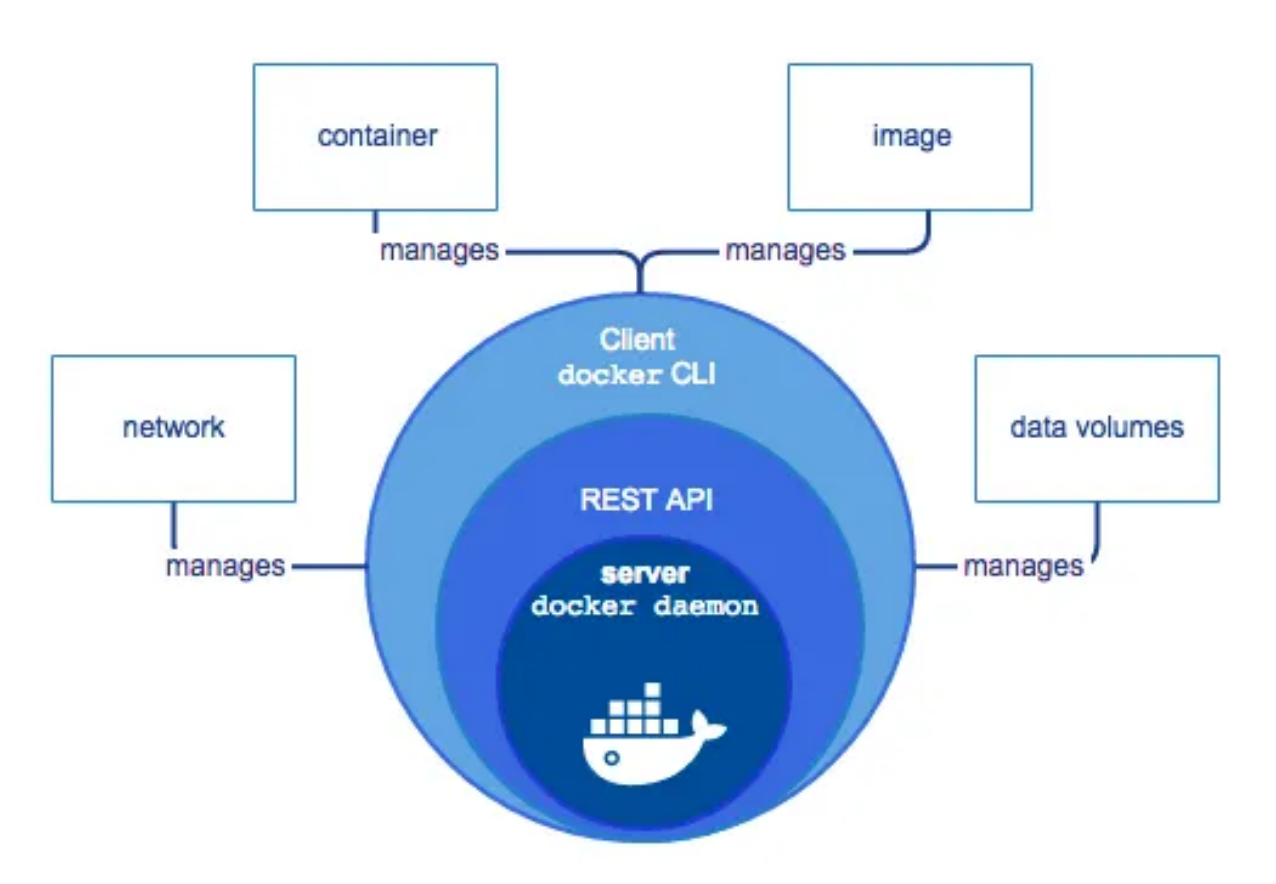
It only runs on Linux because these features are all Linux-only.

```
wget bit.ly/fish-container -O fish.tar       # 1. download the image
mkdir container-root; cd container-root
tar -xf ../fish.tar                          # 2. unpack image into a directory
cgroup_id="cgroup_$(shuf -i 1000-2000 -n 1)" # 3. generate random cgroup name
cgcreate -g "cpu,cpuacct,memory:$cgroup_id"  # 4. make a cgroup &
cgset -r cpu.shares=512 "$cgroup_id"             set CPU/memory limits
cgset -r memory.limit_in_bytes=1000000000 \  #
      "$cgroup_id"                           #
cgexec -g "cpu,cpuacct,memory:$cgroup_id" \  # 5. use the cgroup
    unshare -fmuipn --mount-proc \           # 6. make + use some namespaces
    chroot "$PWD" \                          # 7. change root directory
    /bin/sh -c "                             #
        /bin/mount -t proc proc /proc &&     # 8. use the right /proc
        hostname container-fun-times &&      # 9. change the hostname
        /usr/bin/fish"                       # 10. finally, start fish!
```

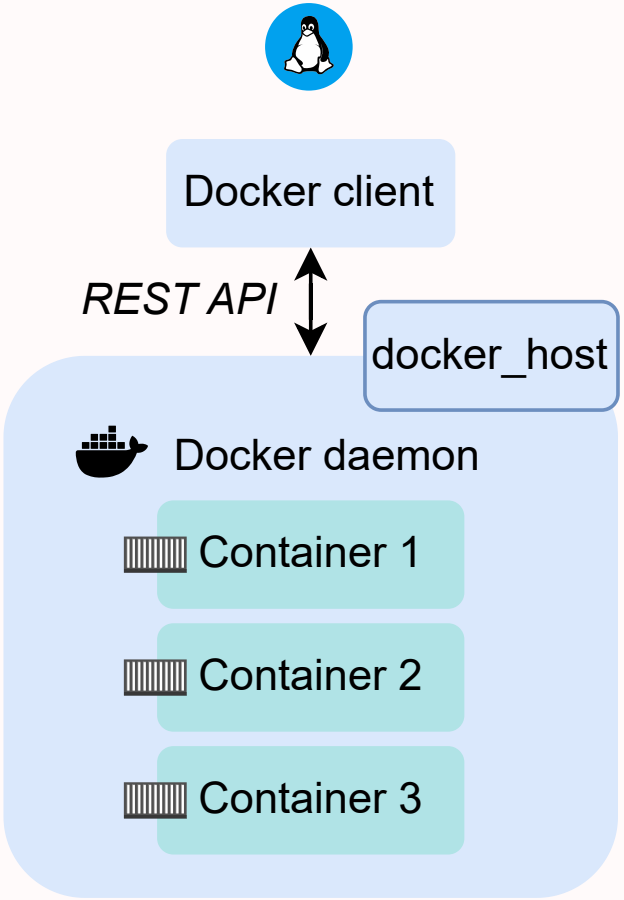
Author: Julia Evans

Docker

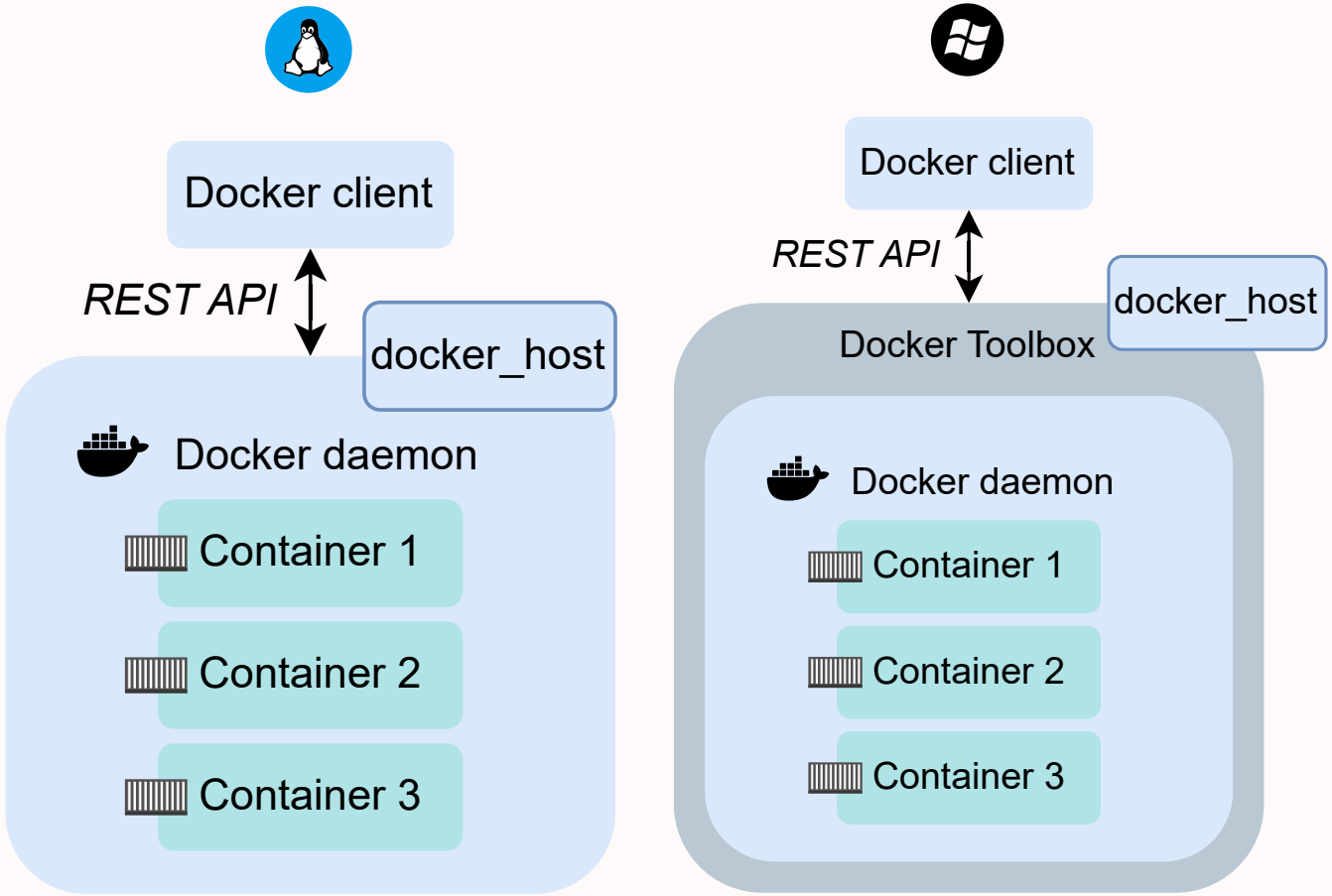
A software used to manipulate containers



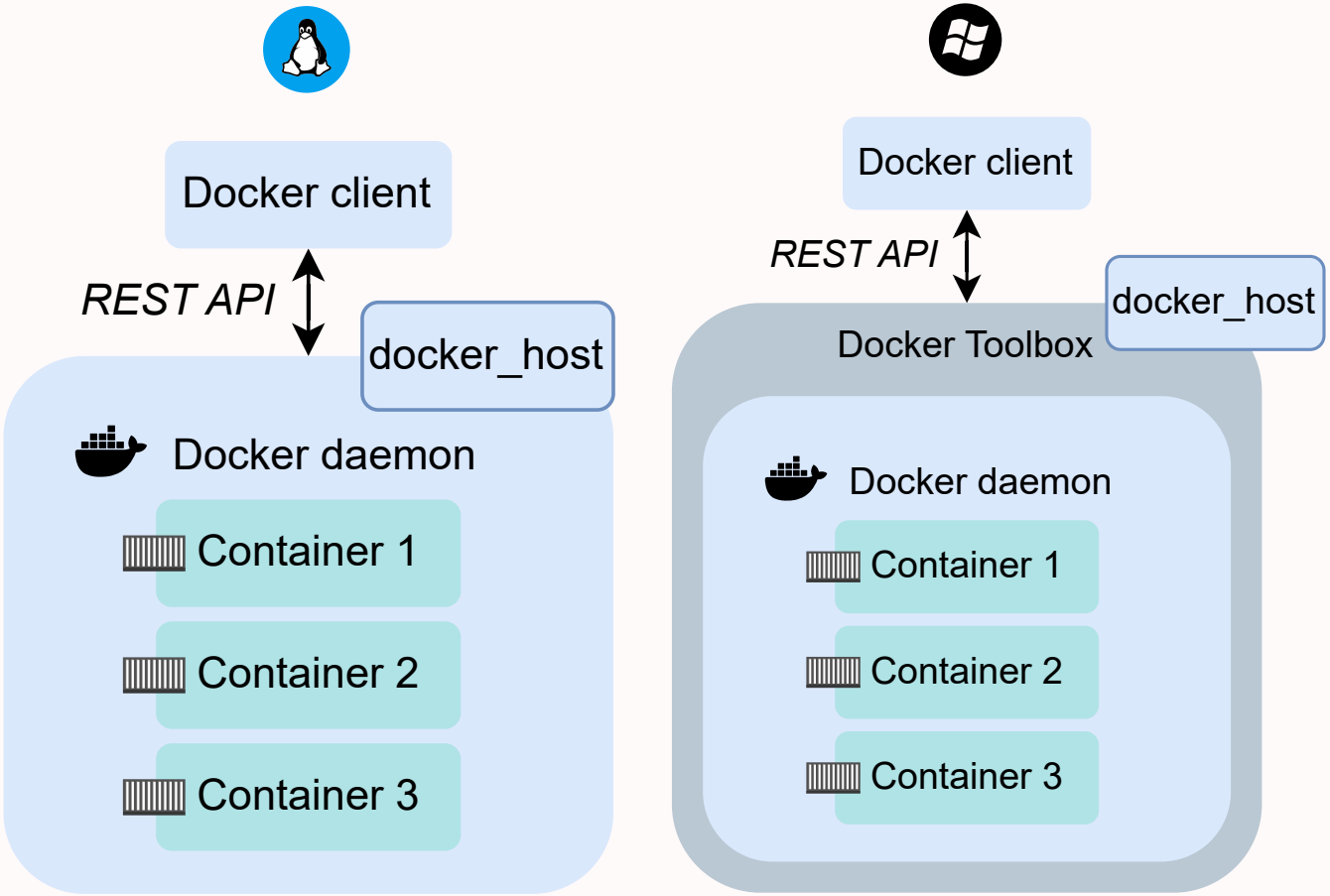
Docker


Docker client
REST API
docker_host
Docker daemon
Container 1
Container 2
Container 3

Docker

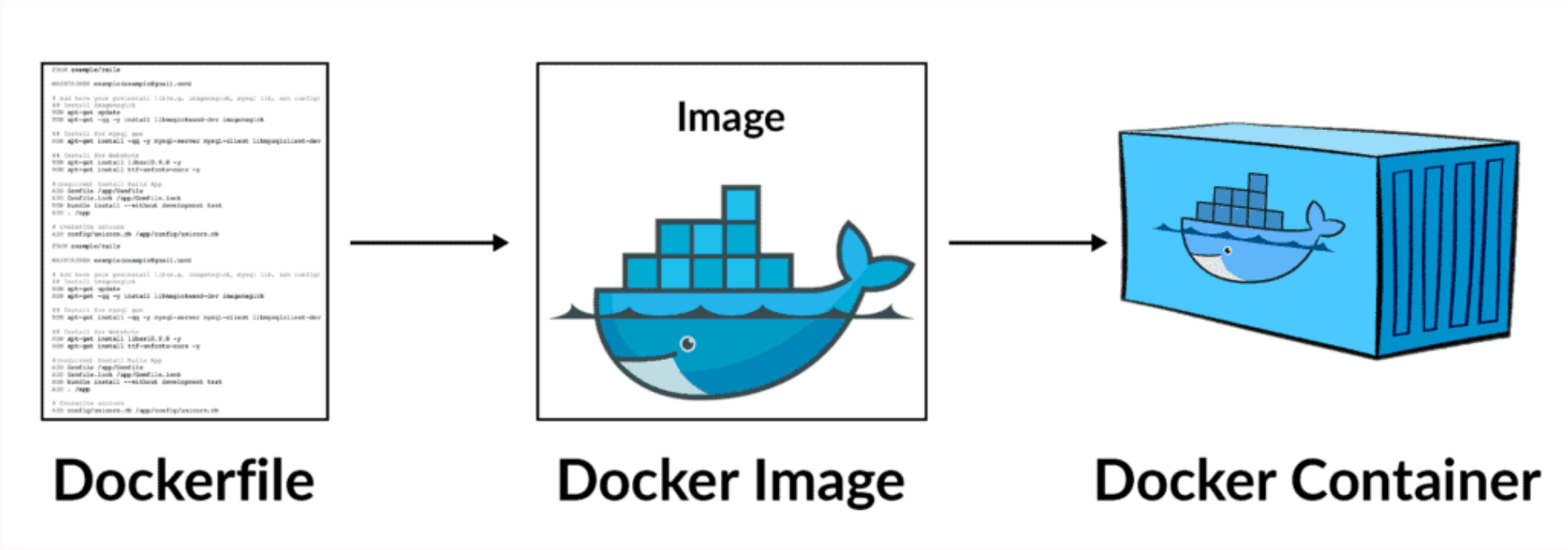

Docker client
REST API
docker_host
Docker daemon
Container 1
Container 2
Container 3
Docker client
REST API
docker_host
Docker Toolbox
Docker daemon
Container 1
Container 2
Container 3

Docker



Docker Toolbox == VirtualBox + VM + some secret third things

Basic docker usage workflow



A Docker container

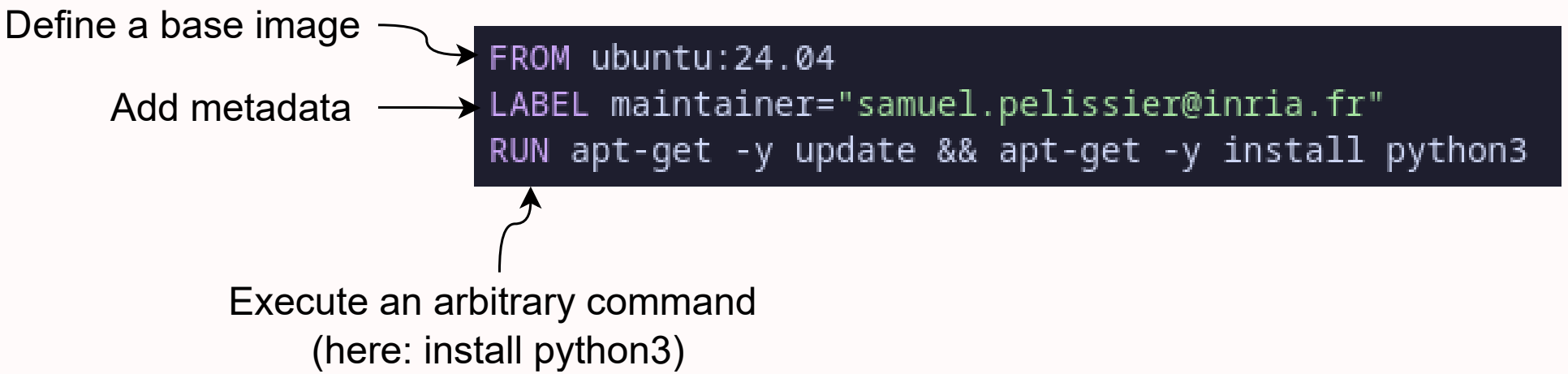
- Images
  - A package containing the application and its dependencies
  - Can be executed in different environments
  - Described by a text file
- An instance of a Docker image
- Analogy with object-oriented programming
  - Image → class
  - Container → instance

DEMO: running a simple Ubuntu with Python

Create a place to work:

```
mkdir -p docker_ubuntu_project
cd docker_ubuntu_project
```

Prepare the Dockerfile:



DEMO: running a simple Ubuntu with Python

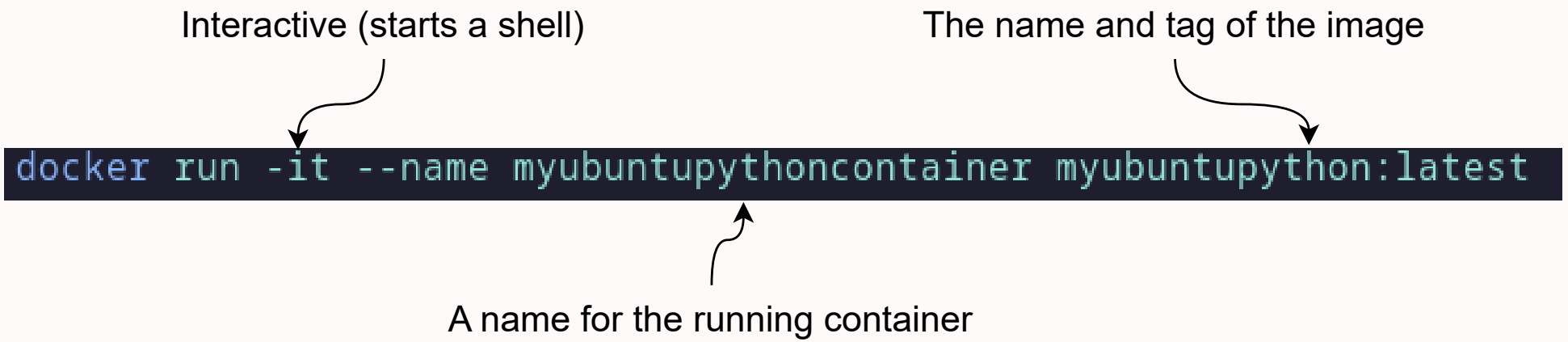
Build the image:

```
docker build -t myubuntupython .
```

```
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM ubuntu:24.04
 ---> c3a134f2ace4
Step 2/3 : LABEL maintainer="samuel.pelissier@inria.fr"
 ---> Using cache
 ---> ceb0edfba5cc
Step 3/3 : RUN apt-get -y update && apt-get -y install python3
 ---> Using cache
 ---> a3e721014e06
Successfully built a3e721014e06
Successfully tagged myubuntupython:latest
```

DEMO: running a simple Ubuntu with Python

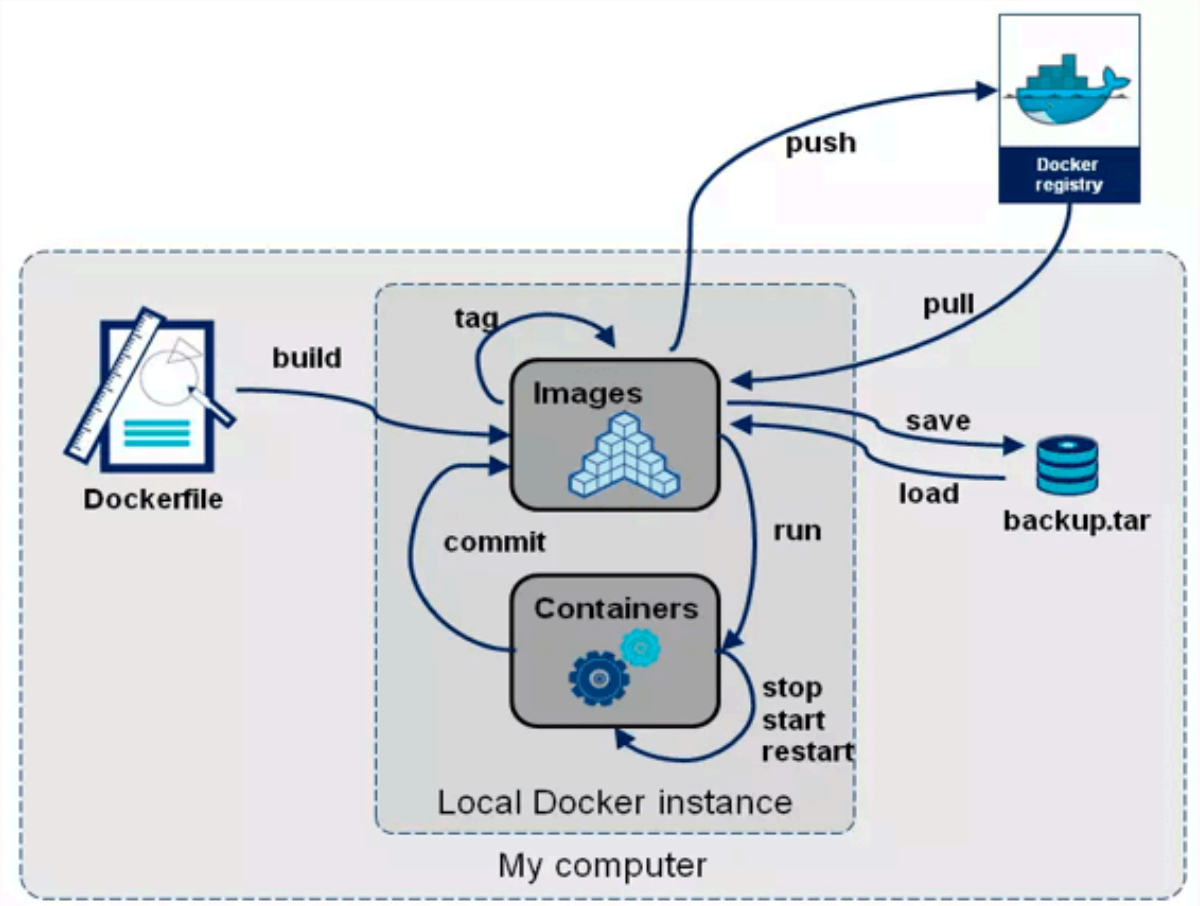
Run the container:



First small victory of the day:

```
root@fc50fbcbadc8:/# pwd
/
root@fc50fbcbadc8:/#
```

A more complete docker usage workflow



For distribution

- An application is built from a Dockerfile
- Images are stored in a registry
  - Durable storage
  - Accessible to everyone
- Execution in production
  - All dependencies are included in the images for the applications
- Versioning
  - Possibility of adding tags to images (e.g., 1.0.2, latest)

In continuous integration (CI)

- Test environments are created from Docker images
  - Tests can be run on any platform
- A new container is created for each new test step
  - No interference between the steps of a test
  - No interference between multiple test executions
- Tests can be performed very frequently

Separating application logic and infrastructure

- Service names can be used in the code (“api”, “db”, etc.)
- A compose file is used to start the application
  - If multiple services are involved, an orchestrator is used
- Service name resolution is done automatically
- Without changing the code, it becomes possible to scale, load-balance, or replicate

DevOps

- To move from development to production
  - A container image and a composition are created
  - The image can be deployed directly to production
  - The dev / test / prod environments are the same
  - Deployment is simplified
  - Developers can handle deployment
- Moving to production is easier

Infrastructure-as-code

- Infrastructure is described in text files
  - Dockerfile
  - This also serves as documentation of the infrastructure
- This description can naturally be versioned
  - Track changes to the description
  - Roll back
- Infrastructure setup is automatic
  - It is deterministic
  - It is reproducible

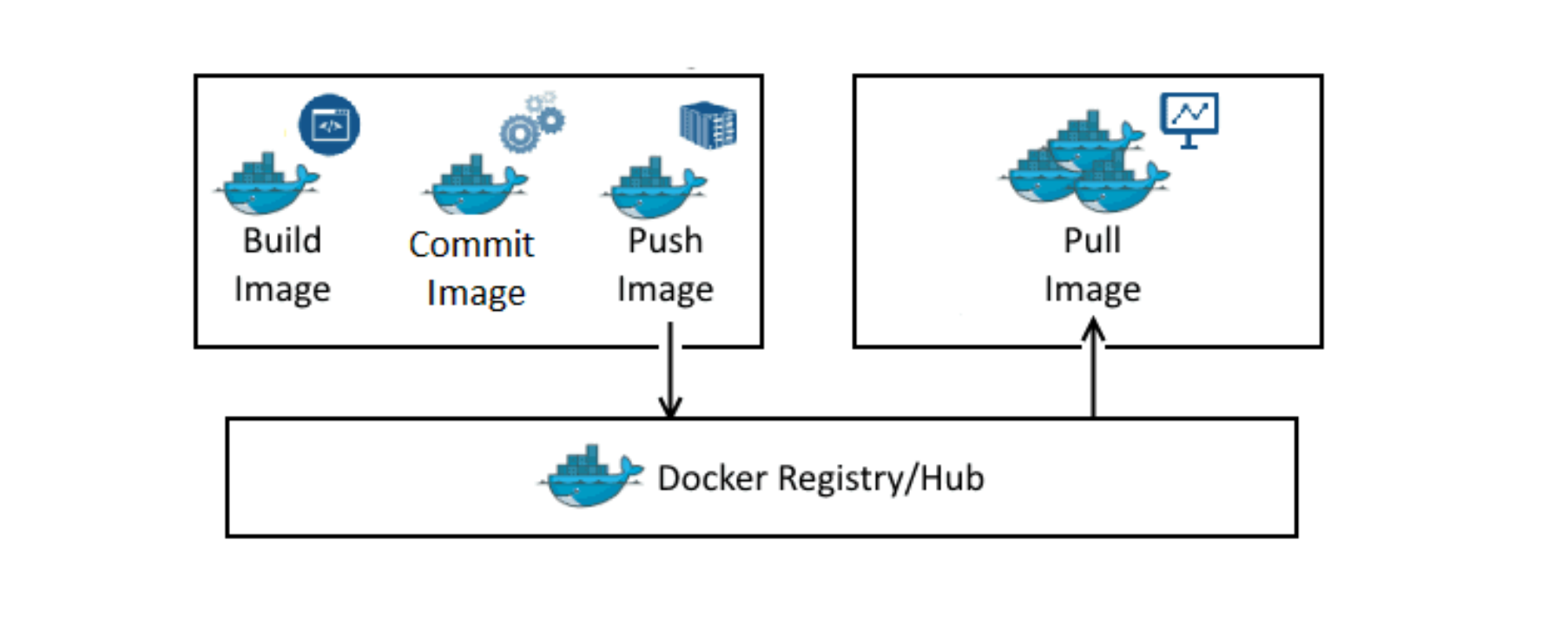
What does Docker provide? Services

- Image building
- Image management
  - Locally
  - Globally (Docker Hub)
- Container execution and management

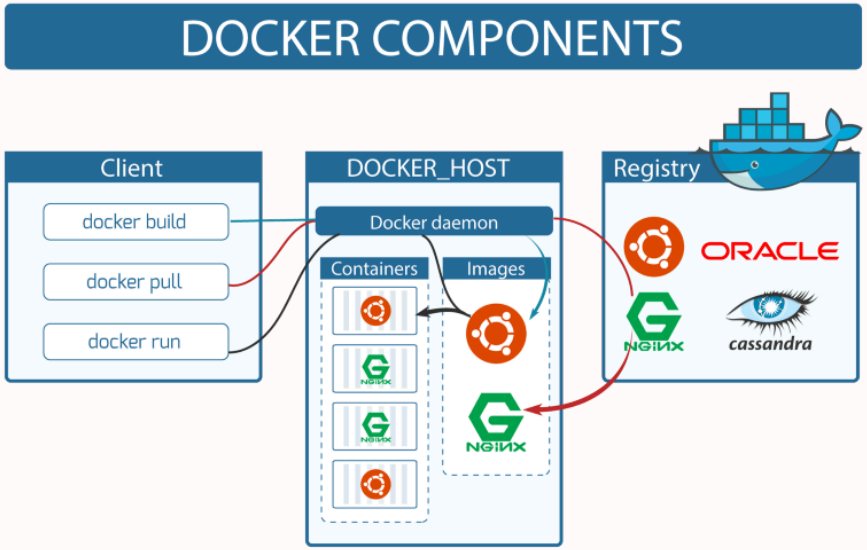
What does Docker provide? Running things

- Docker Engine
  - Runtime environment with a set of services to manage docker containers on a machine
  - Client / Server application
  - Server (Daemon): manages containers on a machine
  - Client: command-line interface
- A Docker image registry
  - Library of available images
  - “Docker Hub”
  - Multiple registries possible
  - Push / pull of images

Docker registry



Other 3rd party registry services: RedHat Quay, Google Container Registry, Azure Container Registry, Amazon ECR...

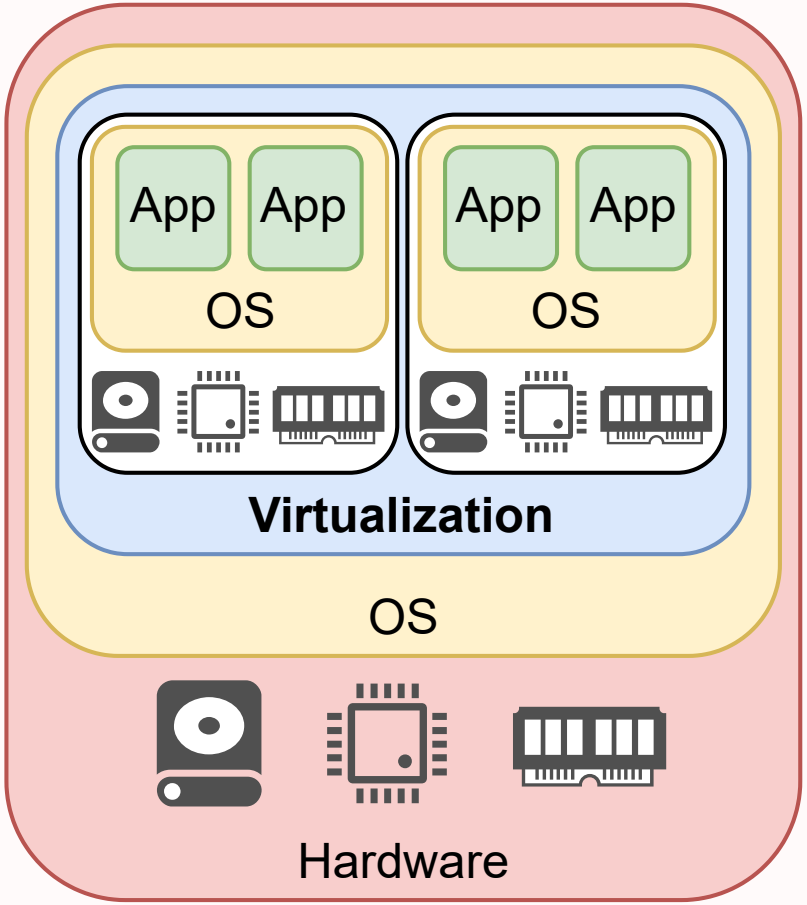


- Client: command-line interface
- Docker host: Daemon managing images and containers
- Registry: a shared registry where images are stored and shared

Bonus question

What are the risks of pulling and running an image from a random person?

Images



VM images

- Snapshot: the VM state is saved (memory, disks, etc.) at a given moment
- The VM restarts in the saved state

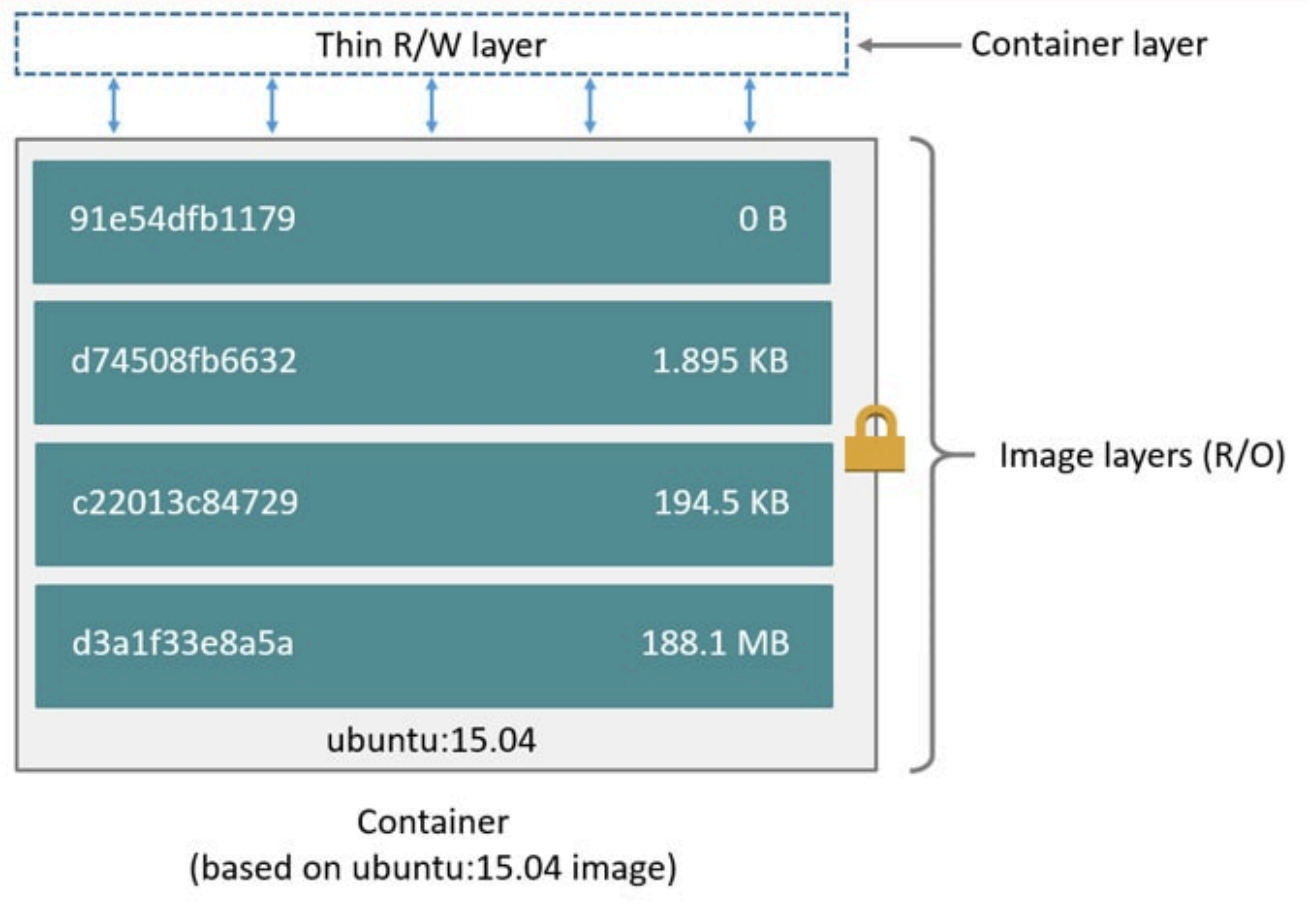
Docker images

- A copy of part of a filesystem
- Immutable filesystem snapshots (the state does not change)

Docker images

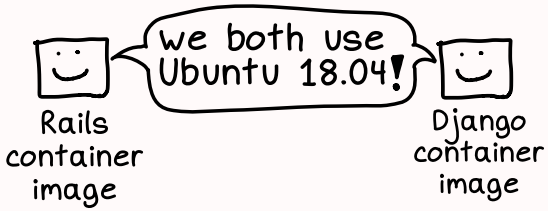
- Docker images rely on the use of the Union File System
  - It allows creating a consistent view of a filesystem from directories and files belonging to different filesystems
- Images consist of a set of layers
  - To combine layers: Union File System
  - By default, the filesystem used is called `overlay2`
  - Each layer corresponds to an instruction in the Dockerfile describing the image

Images built from layers

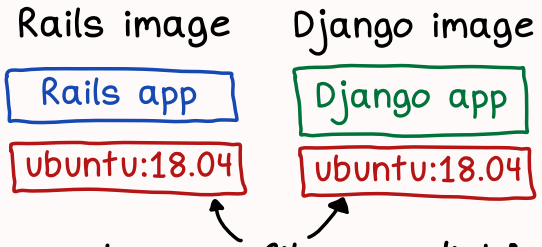


layers

different images have similar files



reusing layers saves disk space



a layer is a directory

```
$ ls 8891378eb*
bin/ home/ mnt/ run/ tmp/
boot/ lib/ opt sbin/ usr/
dev/ lib64/ proc/ srv/ var/
etc/ media/ root/ sys/
```

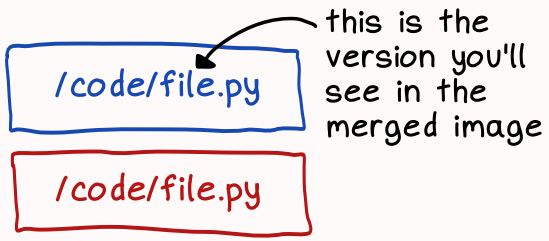
files in an ubuntu:18.04 layer

every layer has an ID

usually the ID is a sha256 hash of the layer's contents

example: `8e99fae2..`

if a file is in 2 layers, you'll see the version from the top layer



by default, writes go to a temporary layer

these files might be deleted after the container exits

To keep your changes, write to a directory that's mounted from outside the container

Author: Julia Evans

Why use layers?

For developers:

- No need to build everything from scratch
- Get an existing image, add a new layer: you get a new, readily usable image
- Update layers independently

For system administrators:

- Lots of images have lots in common
- The same layer can be re-purposed for different images: save space

More about layers

- Docker images are always based on some other base image
  - Official images are available: ubuntu:latest, alpine:latest
  - Note: alpine is minimalist Linux image (5mb), nice for performance, but may not ship everything required to run
- Every layer is identified with a hash of the layer's content
- An image is defined based on its metadata: layer identifiers in a list
- The image identifier is hash of this metadata
- To inspect images: `docker image inspect <id>`

Images

- 3 categories/namespaces
  - Official images: ubuntu, busybox, alpine
  - Small images, distribution images
  - Ready-to-use software: redis, mysql
- User images
  - myuser/myapp
- Hosted images (outside of Docker Hub)
  - registry.example.com:5000/private/img

A note on The Network

Basics:

A note on The Network

Basics:

- We connect to **interfaces** on specific **ports**, using a specific set of **protocols**
- Examples:
  - To connect to a Web Server to get a web page, we connect to port 80 using the IP/TCP/HTTP protocols
  - To connect to a DNS server to resolve a domain name, we connect to port 53 using the IP/UDP/DNS protocols

Some restrictions:

- Only one process can bind to the same IP + port + protocol combination at the same time_

A note on The Network

When using containers and Docker:

- We have multiple containers running
- We want to:
  - Connect them on the same host (e.g. database and web server)
  - Connect to them from the outside (through the host, where other software is running / using ports)

DEMO: deploying a web server

Create a place to work

```
mkdir -p mynginxproject/files
cd mynginxproject
```

Create a base HTML file:

```
<!-- files/index.html -->
<html>
    <head><title>My web page</title></head>
    <body><p>Hello world</p></body>
</html>
```

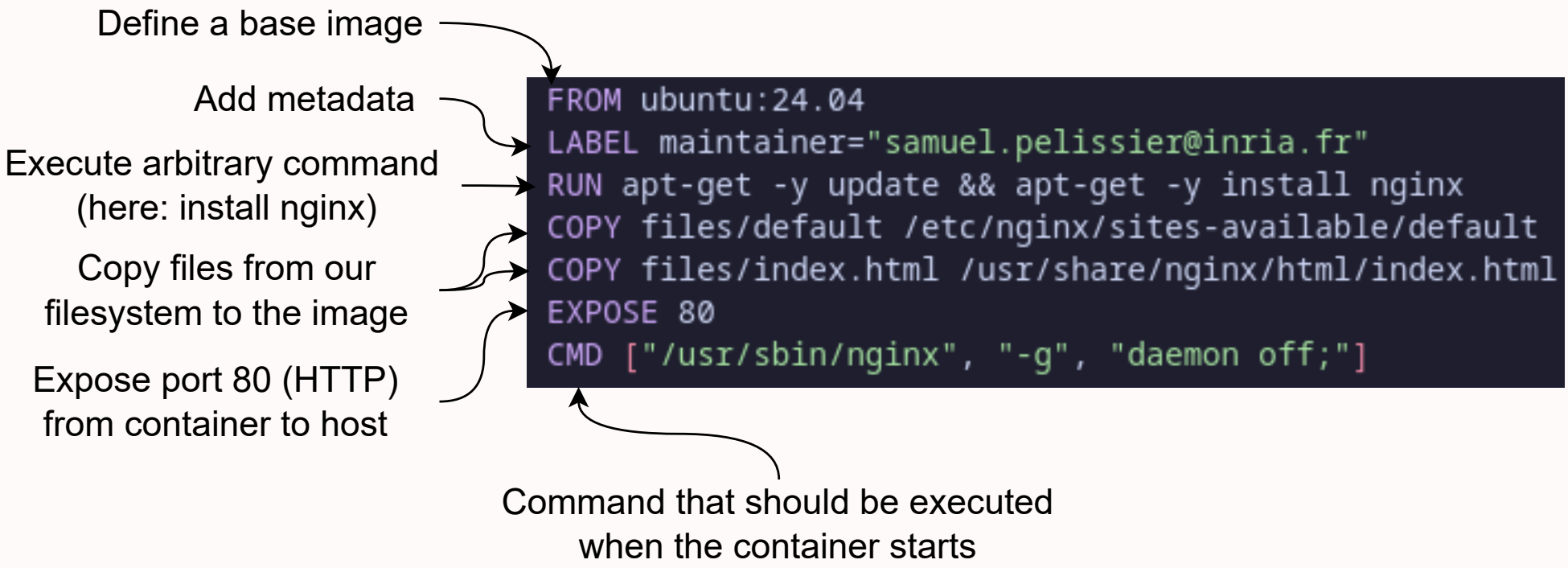
DEMO: deploying a web server

Create the nginx configuration file:

```
# files/default
server {
    listen 80 default_server;
    root /usr/share/nginx/html;
    index index.html index.htm;
    server_name _;
    location / {
        try_files $uri $uri/ =404;
    }
}
```

DEMO: deploying a web server

Prepare the `Dockerfile`:



DEMO: deploying a web server

Build the image:

```
docker build -t mynginxserver .
```

```
Sending build context to Docker daemon  4.608kB
Step 1/7 : FROM ubuntu:24.04
 ---> c3a134f2ace4
Step 2/7 : LABEL maintainer="samuel.pelissier@inria.fr"
 ---> Using cache
 ---> ceb0edfba5cc
Step 3/7 : RUN apt-get -y update && apt-get -y install nginx
 ---> Using cache
 ---> b8cf08ce538b
Step 4/7 : COPY files/default /etc/nginx/sites-available/default
 ---> Using cache
 ---> e871aa115d11
Step 5/7 : COPY files/index.html /usr/share/nginx/html/index.html
 ---> Using cache
 ---> d24a21a64138
Step 6/7 : EXPOSE 80
 ---> Using cache
 ---> 0a803fab3b24
Step 7/7 : CMD ["/usr/sbin/nginx", "-g", "daemon off;"]
 ---> Using cache
 ---> a48a1346b89c
Successfully built a48a1346b89c
Successfully tagged mynginxserver:latest
```

DEMO: deploying a web server

Run the container:

DEMO: deploying a web server

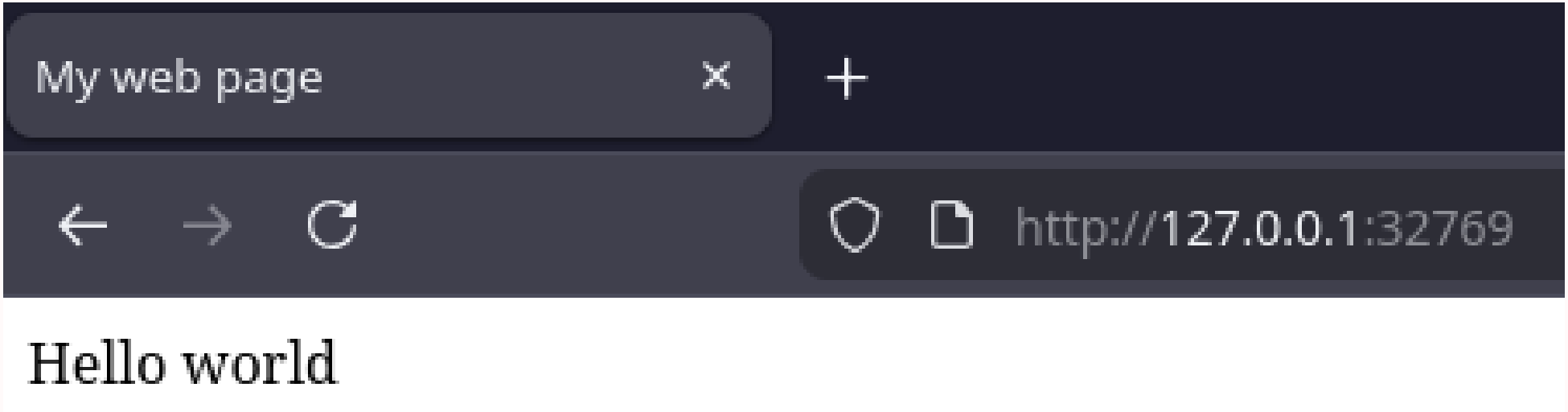
Run the container:


```
CONTAINER ID   IMAGE                  COMMAND                  CREATED
a909bb6eedab   mynginxserver:latest   "/usr/sbin/nginx -g …"   4 seconds ago
```

```
STATUS         PORTS                                          NAMES
Up 3 seconds   0.0.0.0:32769->80/tcp, [::]:32769->80/tcp      mywebservercontainer
```

DEMO: deploying a web server

Small victory of the day:



DEMO: exposing ports

We want to **expose** ports to the outside

```
STATUS         PORTS                                          NAMES
Up 3 seconds   0.0.0.0:32769->80/tcp, [::]:32769->80/tcp      mywebservercontainer
```

- The nginx web server uses port 80 of the container
- This port is linked with the port 32769 of the host
- Send a request to the host on port 32769 == reach the web server inside the container

```
docker port mywebservercontainer 80
# 0.0.0.0:32771
# [::]:32771
```

DEMO: exposing ports

What if we don't want to use a random port?

To expose a specific port of the host:

```
docker run -p 8080:80 # [...]
```

DEMO: exposing ports

What if we don't want to use a random port?

To expose a specific port of the host:

```
docker run -p 8080:80 # [...]
```

Do we **need** to redirect ports? Why not use 80:80?

DEMO: exposing ports

What if we don't want to use a random port?

To expose a specific port of the host:

```
docker run -p 8080:80 # [...]
```

- If nothing on the host runs on the port 80, yes
- May not be a good idea if we want to run multiple web server containers on the same host (e.g., multiple services, each packaged in its own container)

Other notes on Docker networks

IP addresses:

- Each container has its own IP address (bridge, not host mode)
- We can ping this address from the host

```
docker inspect \
--format "{{.NetworkSettings.IPAddress}}" mywebservercontainer
```

A Docker network is a virtual switch:

- Uses a private IP address range (172.17.0.0/16)
- Each container has its own private name
- A DNS service resolves containers' names to IP addresses
- NAT mechanisms allow routing traffic arriving on the host to the correct container

It's possible to create multiple docker networks:

- Containers can be connected to multiple networks
- Increases isolation

Docker networks use drivers

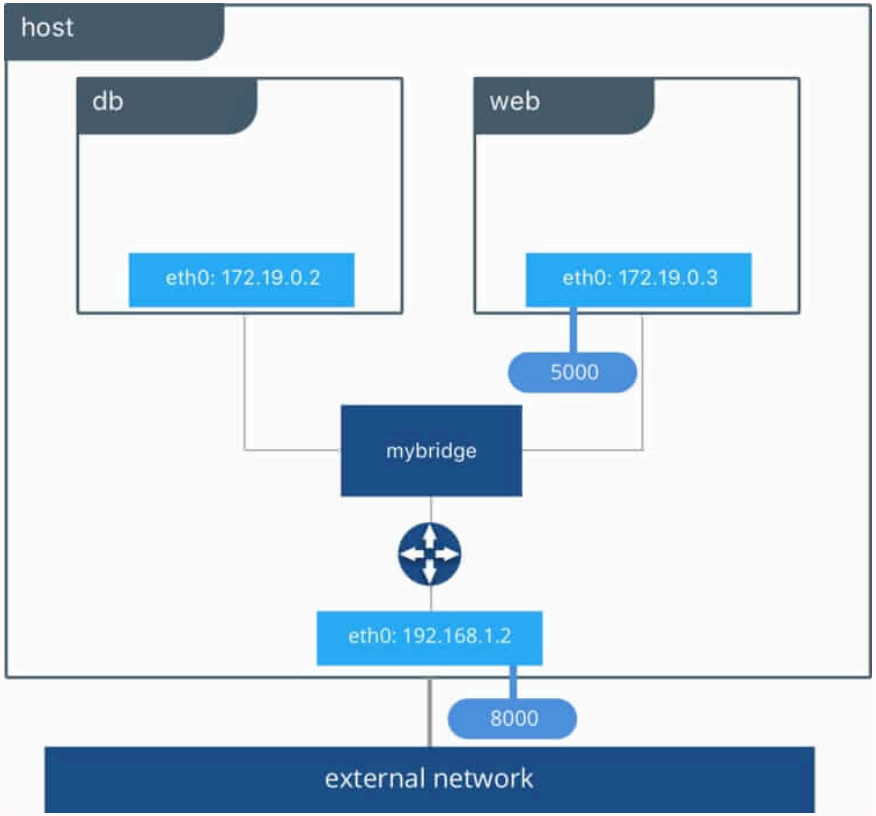
- **Bridge**
  - Created by default when running the container
  - Connected to docker0 virtual interface
  - Containers using this driver can communicate between themselves and reach the outside
  - Need port mapping to be available to the outside
- Host
  - Use the same interface as the host
  - Same IP as host
  - No network isolation

Docker networks use drivers

- Macvlan
  - Assign a MAC address to the container
  - Containers appear as another host on the outside network
- None
  - No connection possible

Docker networks: bridge

Allow containers to communicate between themselves



Docker networks: bridge

Create a bridge:

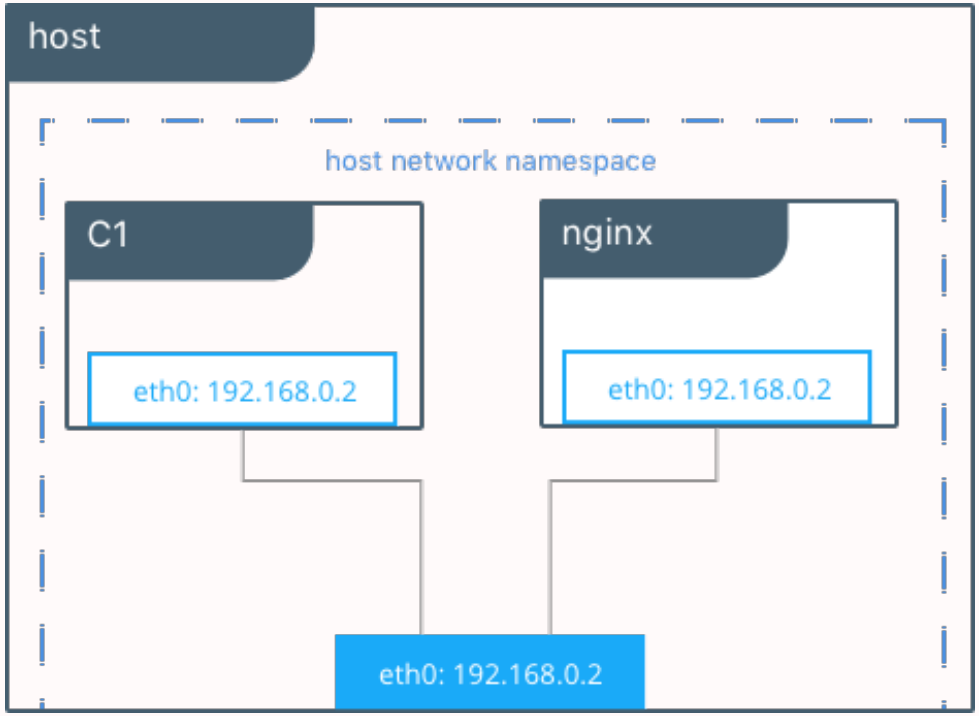
```
docker network create -d bridge my_bridge
```

Attach a bridge to a container

```
docker run -d --network my_bridge --name cont1 ubuntu
docker run -d --network my_bridge --name cont2 ubuntu
```

At this point, both cont1 and cont2 can communicate

Docker networks: host



Docker networks: host

The container uses the same interface as the host:

```
docker run -it --rm --network host \
--name net alpine ip add show myhostnetworkinterface42
```

Data persistence & docker volumes

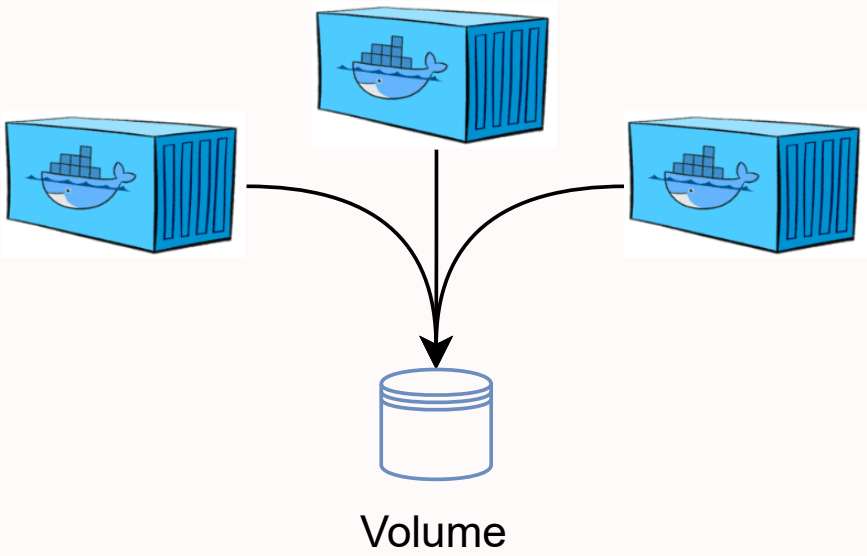
What happens if we remove and run the container again?

Data persistence & docker volumes

What happens if we remove and run the container again?

Docker volumes are the standard way of saving data

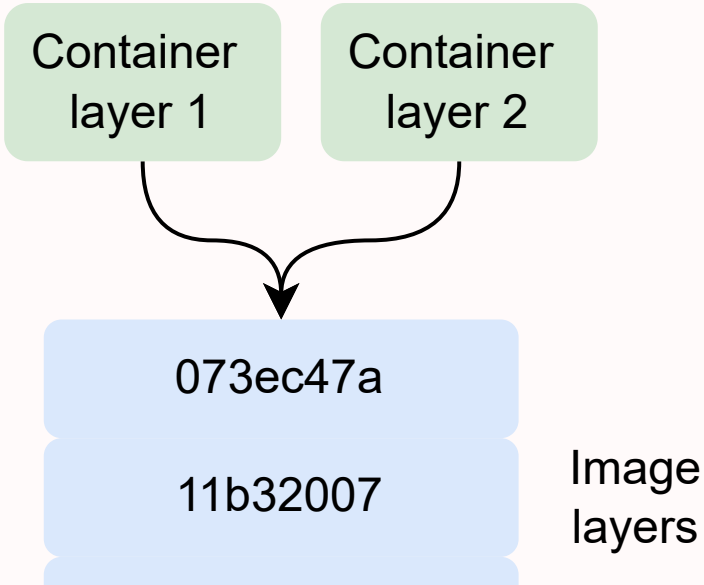
- Multiple docker containers can use the same volume
- **Volume lifecycle ≠ container lifecycle**



Data persistence & docker volumes

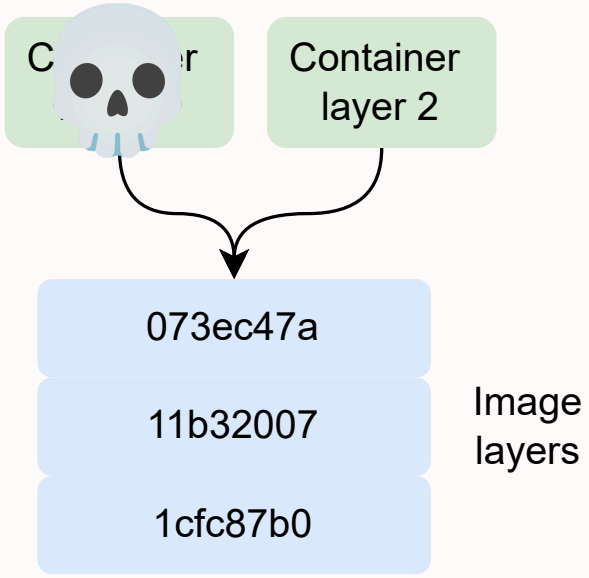
When a container starts...

- A new empty layer is created (“container layer”)
- Container’s processes can create/write/delete files in this layer
- If multiple containers use the same image, each has its own “container layer”



When a container stops...

- All data modified by the container are lost



DEMO: using volumes

To update a file in our demo server (e.g., the index.html), we could:

- Connect interactively to the container, or
- Re-build the image every time

Instead, we will now use a volume

```
mkdir -p files/html
```

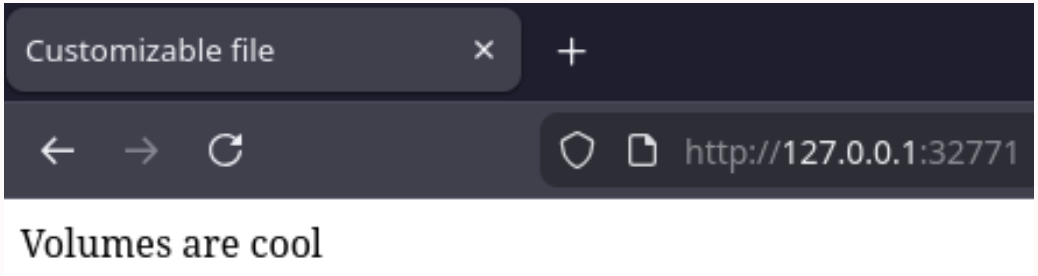
Create a new HTML file:

```
<!-- files/html/index.html -->
<html>
    <head><title>Customizable file</title></head>
    <body><p>Volumes are cool</p></body>
</html>
```

DEMO: using volumes

When starting the container, add a volume pointing to where the HTML file is on the host, and where it should be in the container:

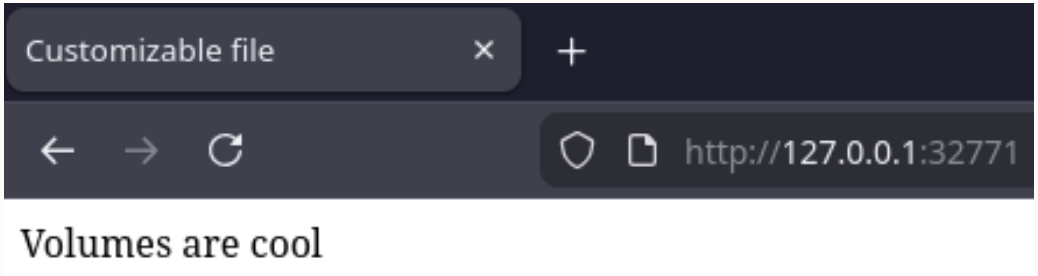
```
docker run -d -P --name mywebservercontainer \
--volume ./files/html/:/usr/share/nginx/html/ \
mynginxserver:latest
```



DEMO: using volumes

When starting the container, add a volume pointing to where the HTML file is on the host, and where it should be in the container:

```
docker run -d -P --name mywebservercontainer \
--volume ./files/html/:/usr/share/nginx/html/ \
mynginxserver:latest
```



What if we update the file live?

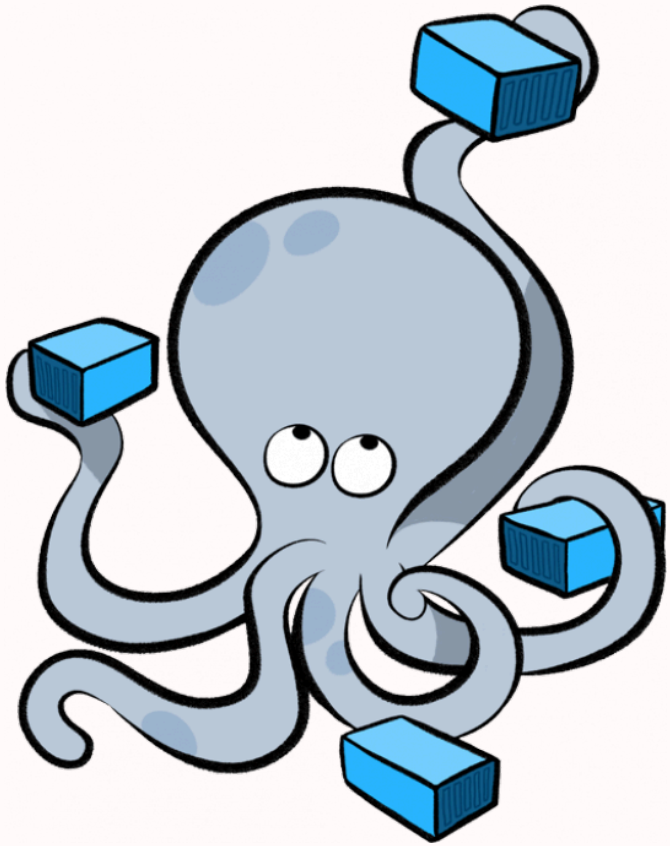
Orchestration

- Configuring everything by hand is tiring
- Commands start to use half the screen
- Managing several containers at once becomes error-prone

Answer: ~~add another complexity layer~~ orchestrators

- Docker Compose (single node, local development)
- Kubernetes (large clusters, production)

Docker Compose



What is Docker Compose?

- A tool for defining and running multi-container Docker applications
- Uses a YAML file (`docker-compose.yml`) to configure services

Benefits

- Reproducibility across environments.
- One-command startup: `docker compose up`.
- Easy dependency management.

DEMO: converting our web server to Docker compose

Create a `docker-compose.yml` file

```yaml
services:
  web:
    build: .
    image: mynginxserver:latest
    container_name: mywebservercontainer
    ports:
      - "8080:80"
    volumes:
      - ./files/html/:/usr/share/nginx/html/
    restart: always
```

- `services`: The containers that make up the app.
- `volumes`: Named volumes shared between services.
- `networks`: Custom networks for service isolation.

DEMO: converting our web server to Docker compose

```
docker compose up # -d (start in detached mode)
```

DEMO: converting our web server to Docker compose

Add a database service? Easy:

```
services:
  web:
    build: .
    image: mynginxserver:latest
    container_name: mywebservercontainer
    ports:
      - "8080:80"
    volumes:
      - ./files/html/:/usr/share/nginx/html/
    restart: always
    depends_on: ["db"]
  db:
    image: postgres:15
    volumes: ["dbdata:/var/lib/postgresql/data"]
```

DEMO: converting our web server to Docker compose

```
validating /tmp/mynginxproject/docker-compose.yml:
services.volumes additional properties 'dbdata'
not allowed
```

Fix: create a named volume

```
volumes:
  dbdata:
```

Another small victory of the day:

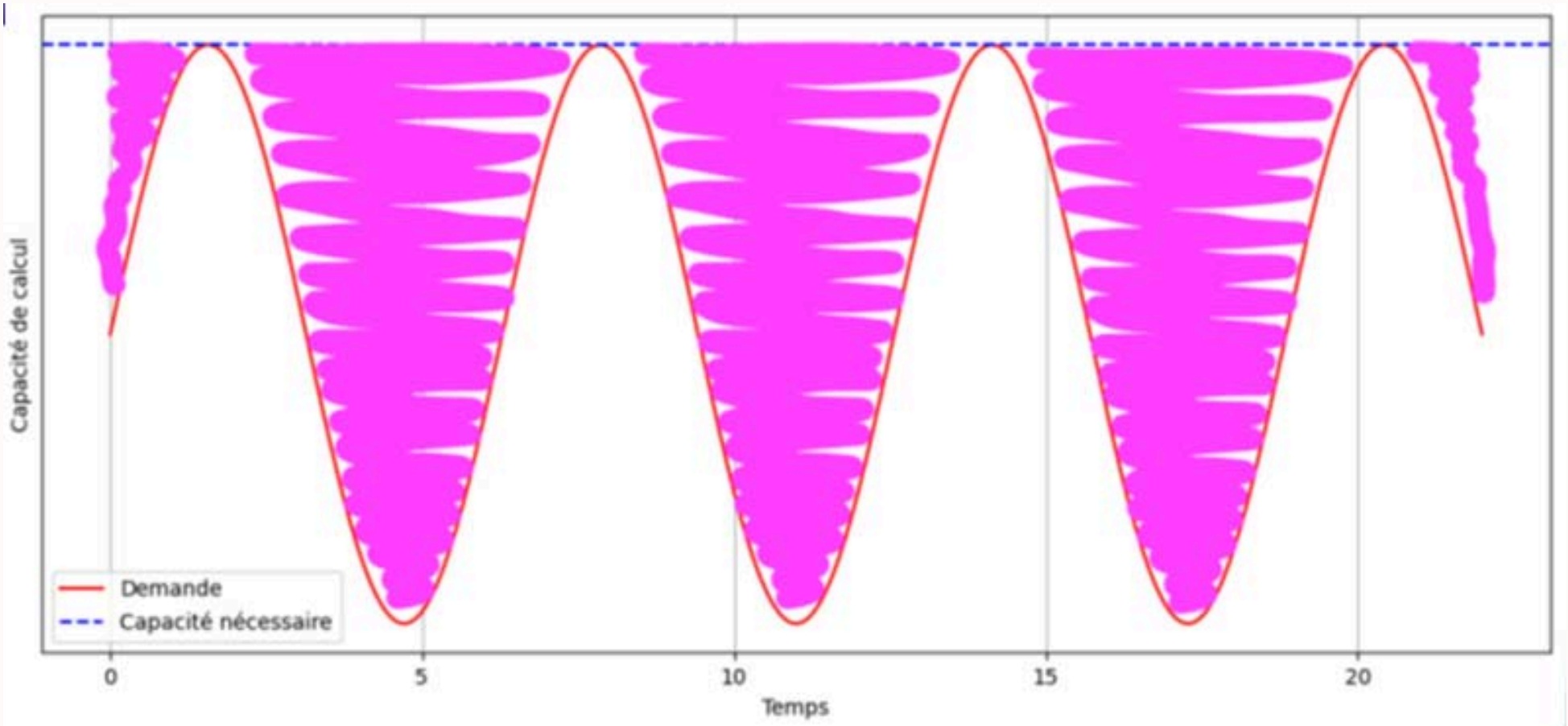
Best Practices for Docker (Compose)

- **Pin image versions** to avoid unexpected updates
- **Keep containers small**: use lightweight base images
- **Leverage healthchecks** to monitor dependent services
- **Use env files**: avoid hardcoding credentials (or ports)
- **Prefer named volumes** over bind mounts in production
- **Split prod vs dev configs**: use override files (`docker-compose.override.yml`).

Summary

- Docker is much lighter than a full-fledged virtual machine
- It uses layers
  - Modifying an image creates a layer
- Isolation between containers and host
  - **Security is not** (always) **perfect** (by default)
  - One wrong configuration == access host's data
- Good guarantee of reproducibility
  - Useful for DevOps

Remember this?



Summary

Docker is not a cloud service

- Docker is software designed to run on a single host
  - Akin to VirtualBox (but for containers)
- What is missing to actually use it as an on-demand cloud service?
  - Deploy containers on large infrastructures
  - Expose resources as containers-as-a-service
  - Resiliency, monitoring, etc.