# Deployment & Virtualization

Joseph Chazalon, Clément Demoulins {**firstname.lastname@lrde.epita.fr**}

October'19

EPITA Research & Development Laboratory (LRDE)

# About this course

# Agenda

This is a course about **Docker**

- What it is.
- How to use it for simple, then less simple cases.
- Practice.

Course outline: 3 sessions of 4 hours

- session #1: Docker basics – Using *containers*
- session #2: Write Dockerfiles and create *images*
- session #3: *Dockerize* some piece of software to distribute it

# Tools and Grading

Tools

- Website:
  **`https://www.lrde.epita.fr/~jchazalo/teaching/DVIRT19/`**
- Moodle :
  **`https://moodle-1920.cri.epita.fr/course/view.php?id=7`**
  - **Enrol ASAP to be able to complete the first quiz**

Graded content for *each session*, using *Moodle*.

- For sessions 1 and 2: 10 minutes quiz on Moodle at the end of each session
  - opens approx. 20 mn before session end
  - closes approx. 5 mn after session end
  - **10 mn to answer all questions once you started the quiz**
  - 10 questions about **both** lecture **and** practice
- Session 3: Mini-project
  - results for final session must be submitted through Moodle
  - **Deadline: Monday, Sept. 7th, 23:59**

# Motivation

# What is Docker and why should I use it?

Docker is like a virtual machine (VM) but lighter

- Minimal storage and memory overhead
- No CPU overhead
- Resource limitation (net. & disk IO, mem, cpu)
- Can share storage, network, etc. with host

Benefits

- Complete isolation of the software stack running inside a container
- Fast and well integrated with Linux:
  you can start commands inside the container directly from the host system

Extra advantage of Docker over VMs: GPU support

# Reasons for NOT using Docker (currently)

- **Archive your program (because it is not made for that)**
- **Your program uses OSX primitives**
- ~~**Your program runs on Windows only**~~
- You need to deploy many containers on clusters
- You cannot get root-like access on your machine
- You do not want to use Linux, and hate terminals
- You use your own custom schroot-based technique with a layered filesystem and custom SELinux rules, and manage network bridging by hand
- You like having dozens of VMs running, and/or you are a Qubes OS user

**Bold** = reasons you may actually have

# Software stack and dependency hell

Suppose you have a little program in Python which processes video files. It is a demonstrator of some fancy cat detection software you want to sell as a service for worried cat owners who love to waste kilowatts.

You use some version of OpenCV to process test files, say version 4.0.

You start thinking about turning this into a web service, and need more libraries to handle *video processing* and *web requests*.

Also, your remote cloud instance has a base distribution (“ProComplicatedLinux”) different from yours (“DumbLinux”).

And OpenCV just got updated and it promises to be much faster.

You spend 2 days recompiling the lastest version of OpenCV to make it work with the special video stream format you process.

You broke by accident your “DumbLinux” installation because it requires OpenCV 4.0.
And nothing works on the remote machine, because it lacks all the fancy libraries “ProComplicatedLinux” maintainers were too snobbish to pre-install, or maybe because they only ship super-old-stable versions of everything (Debian?).
And a prospect wants to test your solution with OpenCV 3.7.

You have to rebuild your complete stack. **Welcome in dependency hell.**

# Software stack illustrated

A real case of two incompatible software stacks we had to handle.

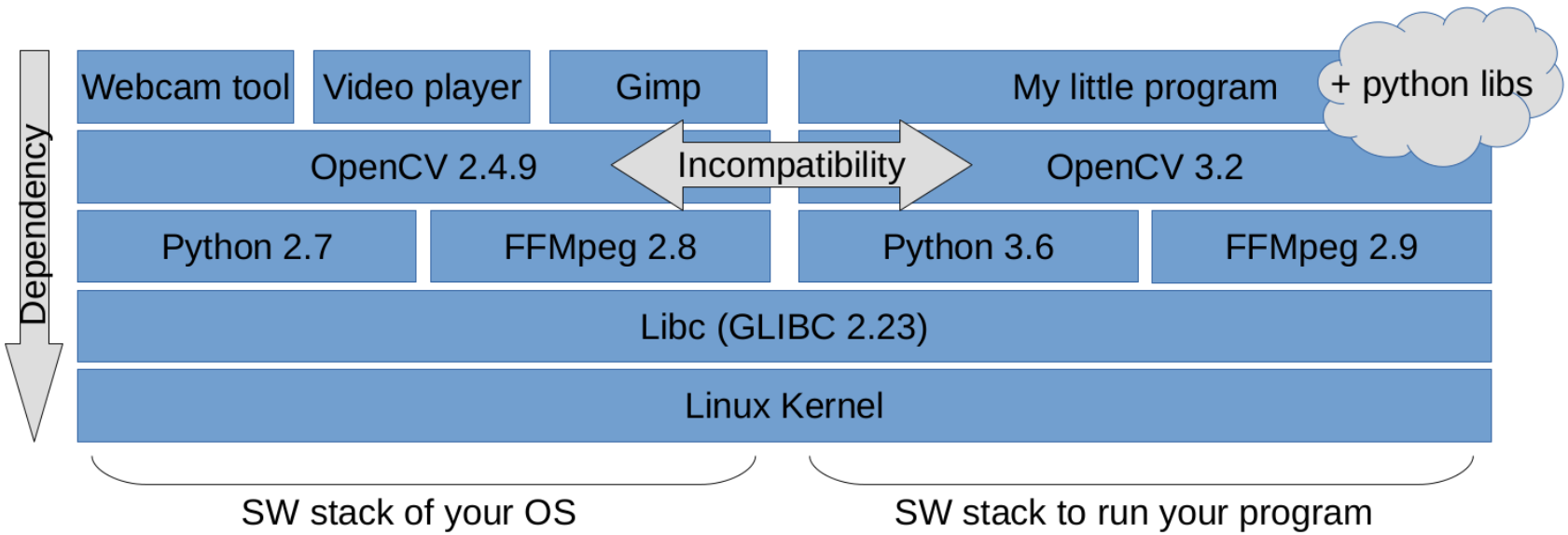



**Figure 1:** Incompatible software stacks

# ( Of course you can make it work...)

- Use libs with forward/backward compatibility (not so common)
- Fix bad dependency declarations in packages (ex: 2.2 vs 2.1+)
- Use language compatibility layer (Python six)
- Rebuild stuff manually

  opencv 4.0 > ffmpeg > h.264 > some weird assembler > libc issue
- Install various versions of libs at different places
  - Heavy use of **$LD_LIBRARY_PATH**
  - Tricky build issues with Python packages
  - Use complicated tools to manage that (env_modules)
- Use virtual environment with Python
  - Then try to use matplot lib and break display
  - Or try to install PyQt4 and start using miniConda
- Force everyone to use the same version of CUDA / CUDNN
- Become a distro package maintainer

**But what you really want is simply to separate:**

- **your development & product software stacks**
- **your OS & userland software stack**

# Docker core concepts

# The promise

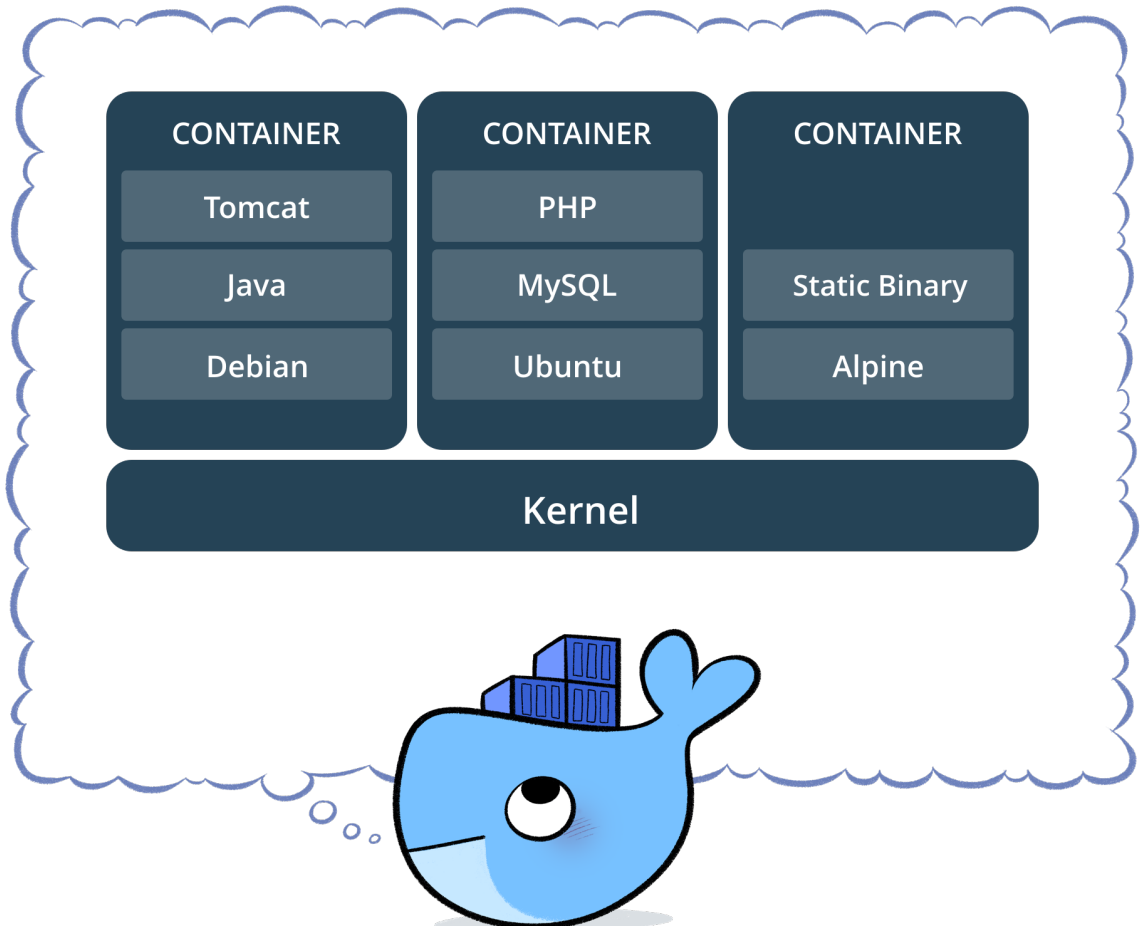
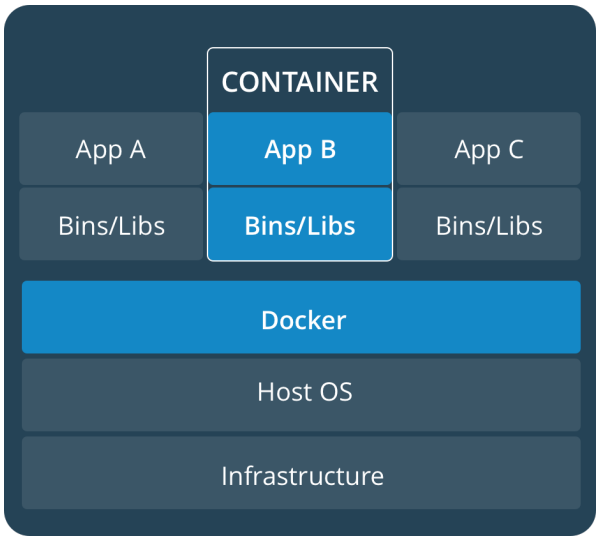



Image credit: Docker.com

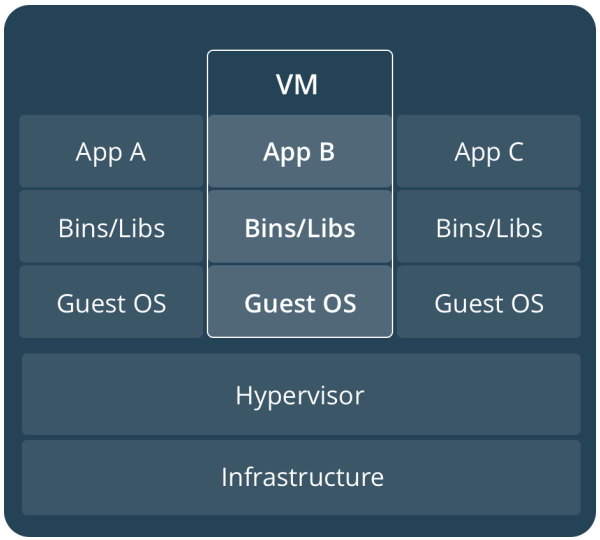
# Containers vs Virtual Machines (1/2)

## Containers




Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.

## Virtual Machines




Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

# Containers vs Virtual Machines (2/2)

Containers and virtual machines:

- have similar resource isolation and allocation benefits,
- but function differently because
  - containers virtualize the operating system
  - instead of hardware
- so containers are
  - more portable and efficient,
  - lighter and faster than VMs,
  - but less secure.

Plus it is great: you can start specific programs directly from the host.

Ok, you *can* with Vagrant, but it is ugly.

# Images and containers

**(base) Image** original content of the filesystem of a container

**Container** kernel-backed sandbox for programs with optional interfaces with the host OS

# Images and containers *illustrated*

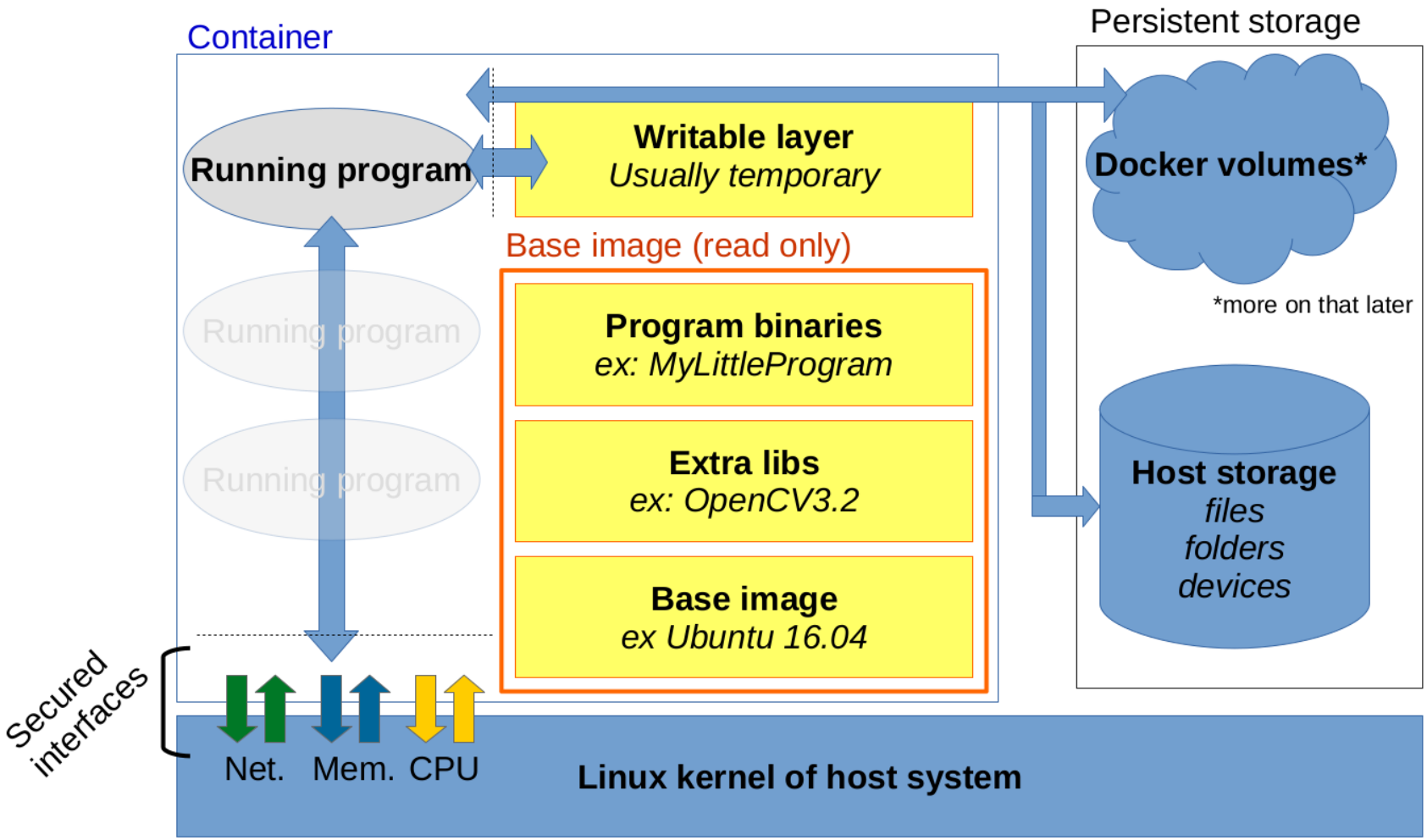



**Figure 2:** Container and base image

# Good to remember

- A “Docker container” is just:
  - a root filesystem with some bind mounts (more on that later), containing all the software stack down to (but not including) the kernel;
  - a control policy enforced by the kernel with some isolation mechanisms: PID, network, etc.;
  - some environment variable and automatically generated files: for hostname, DNS resolution, etc.
- Programs run “inside” containers
  - Such programs are “jailed” with limited capabilities (such as file writes, network, memory, etc.)
  - They see the container’s filesystem
  - and have some environment variables defined automatically
- Docker uses tricks to limit disk usage: layered filesystem in particular
- Docker containers are supposed to be transient and to encapsulate only one program (but nothing forces you to do so)

# A word about cgroups

According to Wikipedia:

> *cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.*

**Features**

**Resource limiting** groups can be set to not exceed a configured memory limit, which also includes the file system cache

**Prioritization** some groups may get a larger share of CPU utilization[10] or disk I/O throughput

**Accounting** measures a group's resource usage, which may be used, for example, for billing purposes

**Control** freezing groups of processes, their checkpointing and restarting

**To understand better**

- `tree /sys/fs/cgroup | less`
- `docker inspect` some container

# Docker is just few things

- A framwork to run programs with different software stacks and capabilities.
- A layered filesystem trick.
- A set of tools to create those stacks, manage them and run programs.
- An ecosystem: Hub, Dockerfiles, etc.

# Using Docker

# Documentation

## Installation

- under Linux

  `https://docs.docker.com/install/linux/docker-ce/ubuntu/`

- Windows / Mac

  `http://docker.com/`

More generally, the official documentation

`https://docs.docker.com/engine` and `http://docker.com/`

Stackoverflow, `docker` tag

Command line help

- `docker help COMMAND [SUBCOMMAND]`
- `docker COMMAND [SUBCOMMAND] --help`

Man pages

- `man docker`
- `man Dockerfile`

# Regular workflow

1. Obtain an image | $\emptyset$ → image on local disk

   = Build a filesystem for the programs to run within the container

   - Pull from Docker Hub or private hub
   - Import from dump
   - Build it from Dockerfile

2. Create a container from image | image → container

   = Define isolation policy: File sharing with host? Ports exposed? Transient?

3. Start the container | container → container started

   = Start custom isolation enforcement by the kernel and run default/custom program

4. (opt.) Execute more programs within the container | cont. started

   = Run a binary withing the custom isolation context

5. Attach your console to the container | cont. started → cont. w/ console

   = See what is sent to STDOUT & STDERR (and write to STDIN)

6. Manage/monitor the container

   = Pause, stop, destroy it – you cannot change the isolation policy once started

# Commands to manage containers

1. Obtain an image | $\emptyset$ → image on local disk

   - `docker image pull USER/IMAGENAME:TAG`
   - `docker image import ARCHIVE`
   - `docker image build ...`

2. Create a container from image | image → container

   `docker container create --name CONTAINER_NAME IMAGE`

3. Start the container | container → container started

   `docker container start CONTAINER_NAME`

4. (opt.) Execute more programs within the container | cont. started

   `docker container exec CONTAINER_NAME command commandargs`

5. Attach your console to the container | cont. started → cont. w/ console

   `docker container attach CONTAINER_NAME`

6. Manage/monitor the container

   `docker system ... / docker container ... / docker image ...`

The **`docker container run`** command handles steps 1 to 5 directly.

# Monitor and manage containers

- List local images

  ```
  docker images ls
  ```

- Show disk space used by Docker

  ```
  docker system df
  ```

- Show container (running and stopped + space)

  ```
  docker container ls -as
  ```

- Show processes running inside a container

  ```
  docker container top CONT_NAME
  ```

- Search for some image on Docker Hub

  ```
  docker search KEYWORD
  ```

- Remove image

  ```
  docker image rm IMAGE_NAME
  ```

- Remove container (but not the persistent storage)

  ```
  docker container rm CONT_NAME   # must be stopped
  ```

- Remove stopped container + unused images

  ```
  docker system prune
  ```

# Container storage explained

# Storage overview

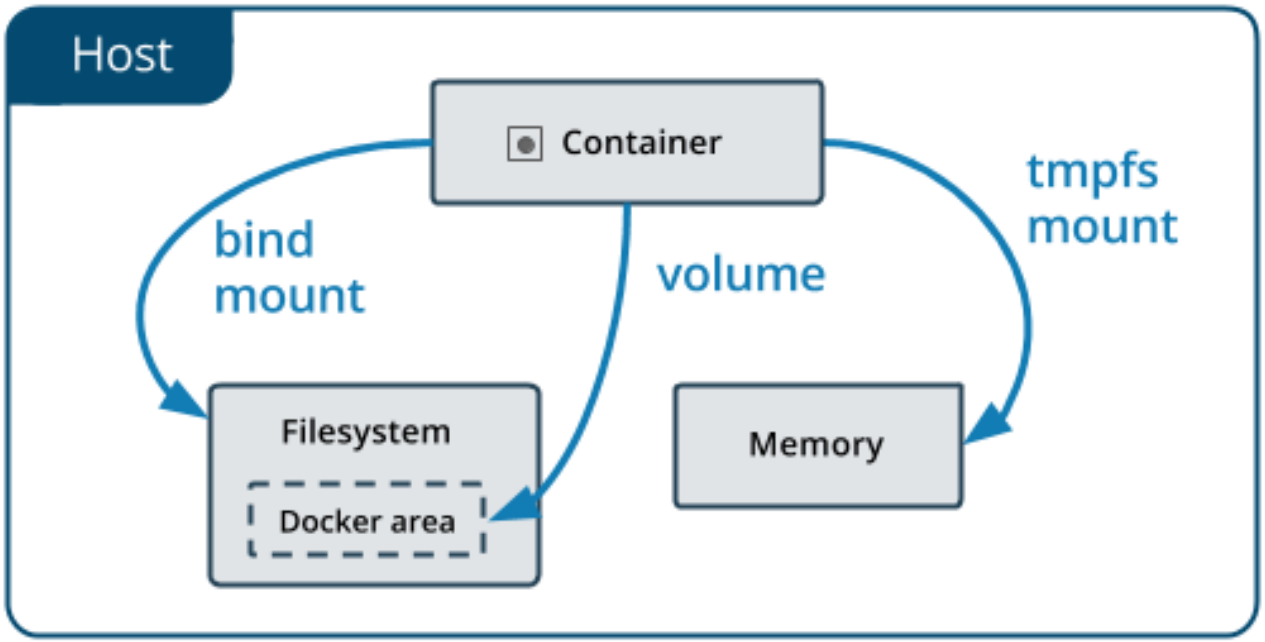



**Figure 3:** Storage spaces for a container

Image credit: Docker.com

# Where is Docker data stored?

Under /var/lib/docker which can get big

```
# ls -lA /var/lib/docker/
total 56
drwx------  2 root root  4096 sept. 23 12:30 builder
drwx--x--x  4 root root  4096 sept. 23 12:30 buildkit
drwx------  2 root root  4096 oct.   1 20:06 containers
drwx------  3 root root  4096 sept. 23 12:30 image
drwxr-x---  3 root root  4096 sept. 23 12:30 network
drwx------ 49 root root 12288 oct.   1 20:06 overlay2
drwx------  4 root root  4096 sept. 23 12:30 plugins
drwx------  2 root root  4096 sept. 27 09:31 runtimes
drwx------  2 root root  4096 sept. 23 12:30 swarm
drwx------  2 root root  4096 sept. 30 22:42 tmp
drwx------  2 root root  4096 sept. 23 12:30 trust
drwx------  2 root root  4096 sept. 30 23:26 volumes
```

*In what follows, we assume we use the overlay2 storage driver.*

# Base image content

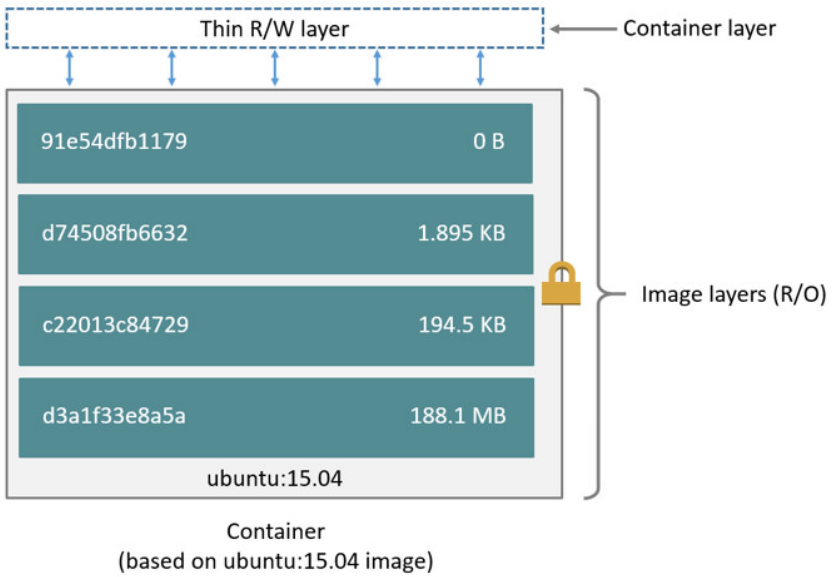



**Figure 4:** Container layers vs base image

Image credit: Docker.com

What

- read only image
- changes go to external mount points or container storage (“thin layer”)

Where

- Under **`/var/lib/docker/overlay2/`**
- As stack of *layers*

Use **`docker inspect`** to locate the files.

# Container thin layer storage

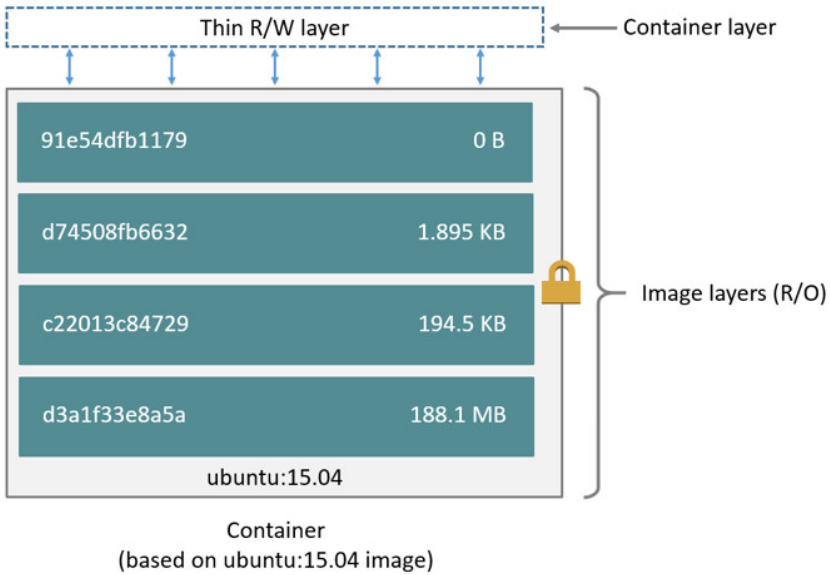



**Figure 5:** Container layers vs base image

Image credit: Docker.com

## What

- Top layer above the stack of layers forming the image
- Writable, eventually transient if container started with **`--rm`** flag

## Where

- Under **`/var/lib/docker/overlay2/`**
- As a single layer

Use **`docker inspect`** to locate the files.

# Bind mounts

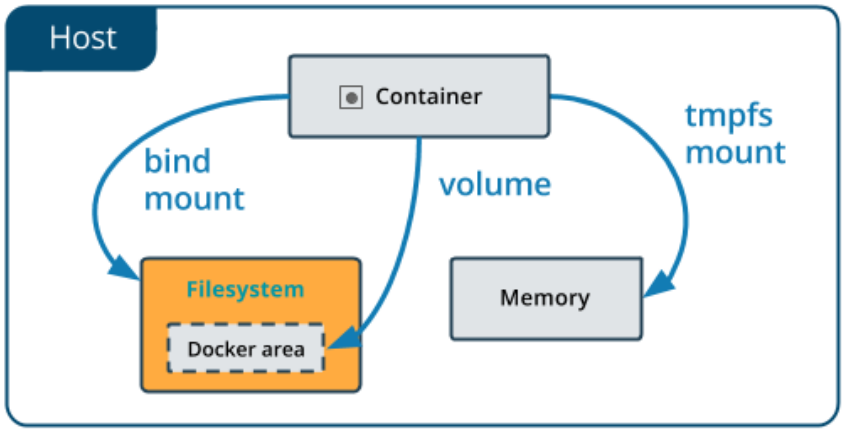



**Figure 6:** Bind mounts

Image credit: Docker.com

## What

- Share folder *or files* with host
- Use **`--mount type=bind,...`** on start/run to activate, can be read only or writable

## Where

- Host path and container mount path

# Volumes (1/3)

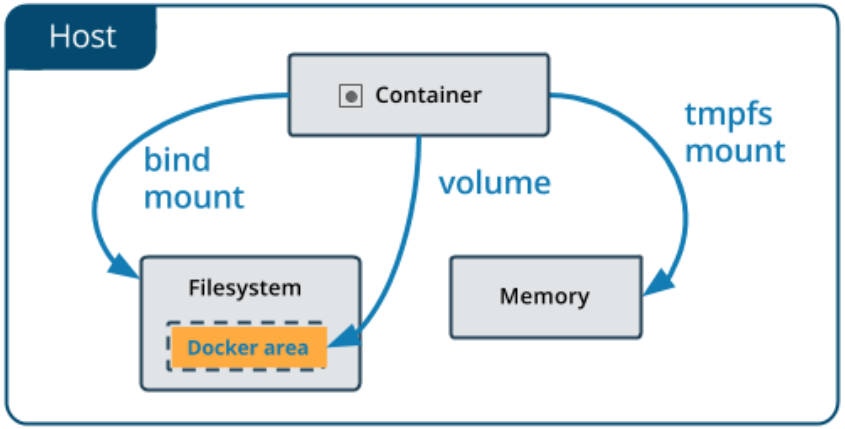



**Figure 7:** Volumes

Image credit: Docker.com

## What

- Shareable space managed by Docker.
- Can be used to share data between container (instead of manually managed bind mounts)
- Create using docker **`volume create VOLNAME`** or **`--volume`** or **`--mount type=volume`** on start/run.
- **Survive container removal**: must be removed manually

## Where

- Stored under **`/var/lib/docker/volumes/`** + name or unique id

# Volumes (2/3)

## Named volumes

To name a volume:

- create it before using it
- or specify a name in the **`--volume`** or **`--mount`** command

Example:

```
$ docker run --rm -it --mount type=volume,src=vol1,dst=/store busybox
...
$ docker volume ls
DRIVER              VOLUME NAME
local               vol1
```

# Volumes (3/3)

## Anonymous volumes

- Like named, but created automatically when requested.
- Do not provide a name.

Example:

```
$ docker run --rm -it --mount type=volume,dst=/store busybox
...
$ docker volume ls
DRIVER              VOLUME NAME
local               c56a1620b60ea3c549cebfb2...
```

# Temporary RAM filesystem

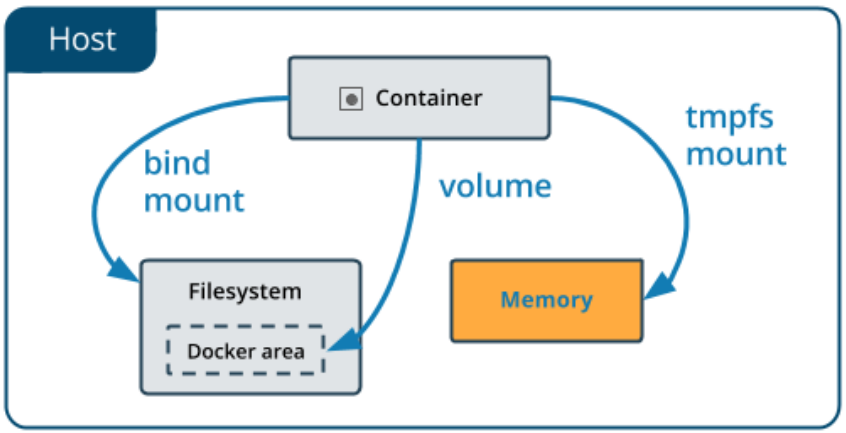



**Figure 8:** Tmpfs

Image credit: Docker.com

## What

- simple temporary RAM storage
- Use **`--tmpfs`** or **`--mount type=tmpfs`** (more options) on start/run
- Size can be limited

## Where

- Mountpoint of the container
- RAM, and not under **`/var/lib/docker`**

# Reusing volumes from another container

It is possible to mount volumes from another container.

This can be convenient in several cases:

- get a shell in a super minimal container (without shell)
- migrate a database (mount storage volume with migration container)
- upgrade a container and keep the volumes
- ...

To do so, run a container with the **`--volumes-from OTHER_CONTAINER`** parameter.

# Networking

# Access exposed ports in the container

By default, when an application listen to a particular port in the container, it is not possible to access it from outside.

We need to explicitly add a port-forwarding rule when creating the container using the **`--publish`** (or **`-p`**) flag.

Examples:

```
# host-all-interfaces:80 -> container:80
docker run -p 80:80 nginx

# loopback-if:8080 -> container:80
docker run -p 127.0.0.1:8080:80 nginx
```

# DNS service, hostname, etc.

Likewise, it is possible to specify DNS servers, hostname, etc. upon container creation.

Docker automatically creates and configures the appropriate files in the container file system.

# Networks (1/2)

Like VM hypervisors, Docker supports several network modes (called “drivers”)

**No network** **`none`**

Use by specifying **`--network none`** at container creation.

Disables networking for the container: no incoming nor outgoing connexions.

**Host networks** **`host`**

Use by specifying **`--network host`** at container creation.

Disables network isolation with host: no need to **`--publish`** ports. The container shares the network stack (therefore the IP addresses) of the host.

# Networks (2/2)

User-defined bridge networks **`bridge`**

Use by creating a network (**`docker network create my_net`**) and select it **`--network my_net`** at container creation.
Docker-managed bridge networks (like a private LAN between containers) with DNS resolution based on container names.
Container can be added and removed on the fly.
Usually messes up you *iptables* configuration.

Other network types:

- **`overlay`**: like **`bridge`** but among several machines;
- **`macvlan`**: creates a virtual physical network device.

# Networks Default configuration (1/2)

By default, Docker configures 3 networks **bridge**, **host** and **none**:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
55a7d0e08c57        bridge              bridge              local
10e259ce5c67        host                host                local
ff25cfaea7dd        none                null                local
```

Their names are a bit misleading:

- **host** and **none**: only 1 network instance possible (meaningful) for each driver
- but the **bridge** is just one possible bridge network!

# Networks Default configuration (2/2)

```
$ ifconfig -a
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
      inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
      inet6 fe80::42:60ff:fe72:a2b6  prefixlen 64  scopeid 0x20<link>
         ether 02:42:60:72:a2:b6  txqueuelen 0  (Ethernet)
         RX packets 102076  bytes 30590113 (30.5 MB)
         RX errors 0  dropped 0  overruns 0  frame 0
         TX packets 115278  bytes 748404709 (748.4 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
...
```

Also check **`docker network inspect bridge`**

# Docker security

# Fragile isolation with host

- Relies on kernel security
  - *Do you have this buggy strange driver for this old serial card loaded?*
  - Less secure than virtual machines (esp. fully virtualized ones)
- You can share a lot of things with host
  - Read-write bind mounts
  - **`--net=host`**
  - **`--privileged`**
- Many public images run services as **`root`** within the container
  - Any breach compromises the whole container

# Not for every user

**`docker`** group == **`root`** group

1 liner to be root on host machine:

```
docker run --rm -it -v /:/host busybox chroot /host
```

# Extra tricks

# How to display windows?

You can bind the X11 socket to display windows!

- You also have to export a couple of environment variables
- The procedure is a bit different with OSX hosts, and I do not know if it can work under Windows hosts

```
docker run MYIMAGE xeyes --interactive --tty \
    --volume "/tmp/.X11-unix:/tmp/.X11-unix:ro" \
    --env "DISPLAY=\$DISPLAY" \
    --env "QT_X11_NO_MITSHM=1"  # opt, for QT
```

**Bold** = old syntax, you should use --mount now

# How to avoid running programs as root inside the container?

By default the programs are run as root inside the container

This can be annoying for various reasons

- Some programs refuse to be run as root
- If the container writes to a directory shared with the host, the files will be owned by root

Some possible solutions:

- Create and use another user in the container;
- Run programs as **nobody** within the container;
- Use some particular UID/GID when running a command in the container by using:

```
docker [run|exec] -it --user UID --group GID IMAGE COMMAND
```

# Can I use the webcam(s) inside my container?

Sure, you just need to share them when creating the container, using **`--device`**

Careful though: like mounts and volumes, device bindings cannot be changed after container creation.

USB webcams can cause issues when they are absent upon container restart

- Bind failure or dummy file creation on host OS
- Manual fix is simple but annoying
- Maybe it is better in recent versions

# Can I use a Nvidia GPU with CUDA inside a container?

- Yes, you need to use **nvidia-docker** or the new **`--gpus`**, **`--runtime`** and other run parameters
- This sets up appropriate permissions (if needed) and bind mounts the GPU device(s)
- The host machine needs to have Nvidia drivers (and GPU!) installed