

Deployment & Virtualization

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

February 2020

EPITA Research & Development Laboratory (LRDE)

About this course

This is a course about *containers* using **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

- Website: Where all resources are
 - <https://www.lrde.epita.fr/~jchazalo/teaching/DVIRT/>
 - Find the subdirectory for your session: **202002_IMAGE_S8**
- Moodle: Where you need to go for grading
 - <https://moodle.cri.epita.fr/course/view.php?id=59>
 - 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, March 2nd, 23:59**

Development and Deployment challenges

Software stack illustrated

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

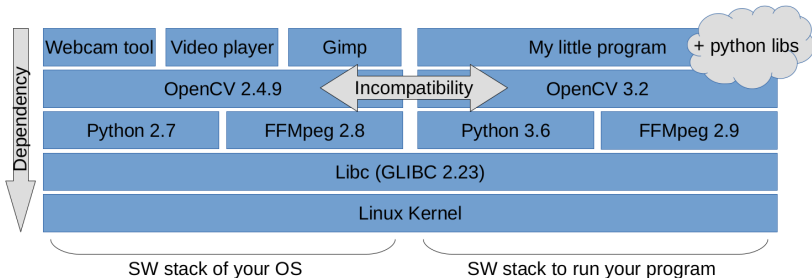


Figure 1: Incompatible software stacks

Many solutions, sometimes good, sometimes bad...

- 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 matplotlib and say *adios* to display windows
 - Or try to install PyQt4 and start using miniConda
- Force everyone to use the same version of CUDA / CUDNN
- Become a distro package maintainer
- ...

When you have to rebuild manually, step by step, all your software stack checking each dependency.

You end up doing the job of distribution maintainers, which is hard and painful.

It takes you ages and a lot of computing power to recompile everything.

What are you paid for?

But what you really want is simply to separate:

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

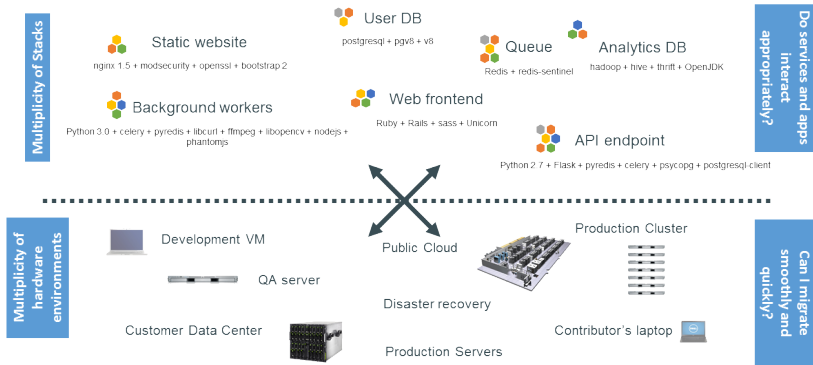
What are you paid for?

But what you really want is simply to separate:

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











And what about deployment?

Deployment challenge



Credit: J. Petazzoni

Deployment matrix of hell

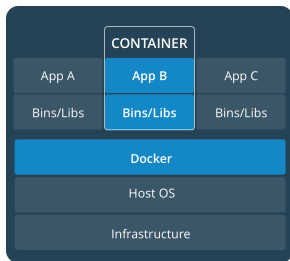
	Static website	?	?	?	?	?	?	?
	Web frontend	?	?	?	?	?	?	?
	Background workers	?	?	?	?	?	?	?
	User DB	?	?	?	?	?	?	?
	Analytics DB	?	?	?	?	?	?	?
	Queue	?	?	?	?	?	?	?
		Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
								

Credit: J. Petazzoni

Solutions

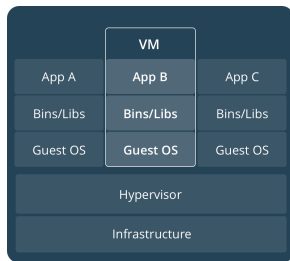
Containers and Virtual Machines (1/2)

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 and Virtual Machines (2/2)

Containers and virtual machines:

- are two good solutions to software stacks isolation
- have similar resource isolation and allocation benefits (CPU, mem, net & disk IO)
- but function differently because
 - containers virtualize the operating system (the kernel)
 - instead of hardware
- so containers are
 - lighter and faster than VMs (minimal storage and memory overhead, negligible CPU overhead)
 - more portable (arguably) and efficient (better density)
 - but less secure.

Plus it is great; with containers:

- You can start specific programs directly from the host.
Ok, you *can* with Vagrant, but it is ugly.
- It is easier to use and share GPUs with the host.

CPU pass through and virtualization is possible but more complex with VMs

Docker promise #1: easy SW stack

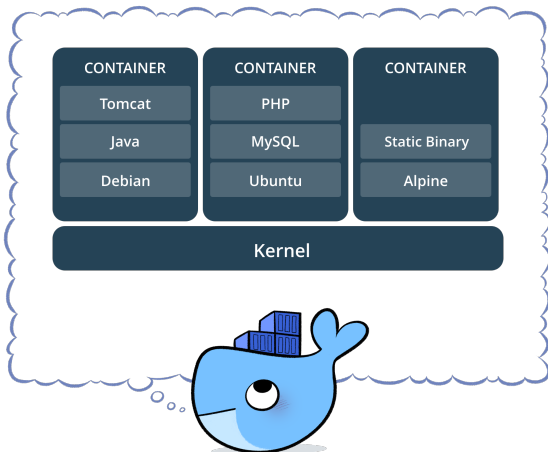
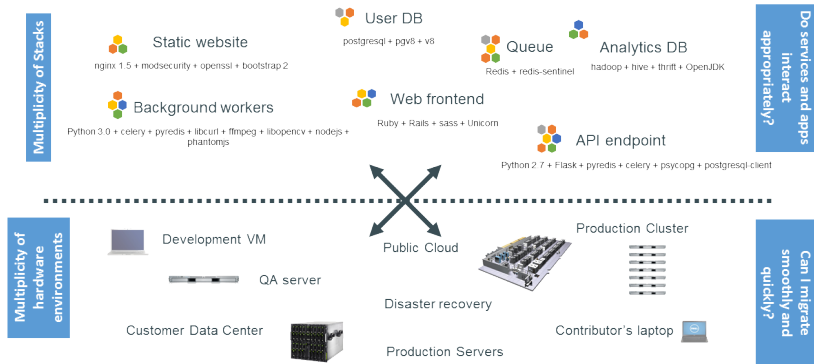


Image credit: Docker.com



Docker promise #2: easy deployment

Remind the challenge?



Credit: J. Petazzoni

And the matrix from hell?

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

Credit: J. Petazzoni

Before 1960, cargo shipping had this issue

Multiplicity of Goods



Do I worry about
how goods interact
(e.g. coffee beans
next to spices)

Multiplicity of
methods for
transporting/storing



Can I transport quickly
and smoothly
(e.g. from boat to train
to truck)

Credit: J. Petazzoni

With their own matrix of hell

	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
							

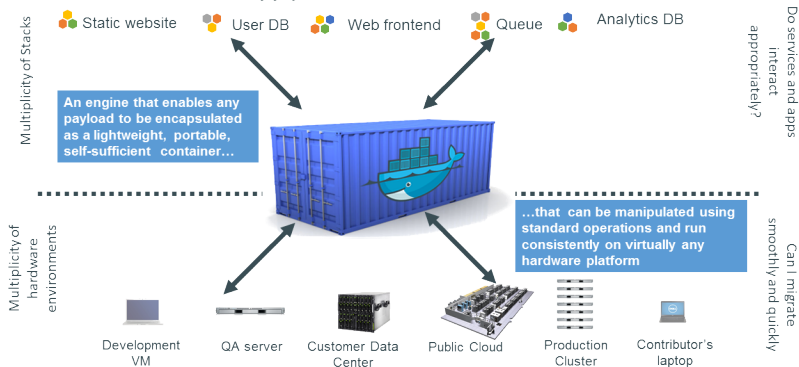
Credit: J. Petazzoni

They found a solution



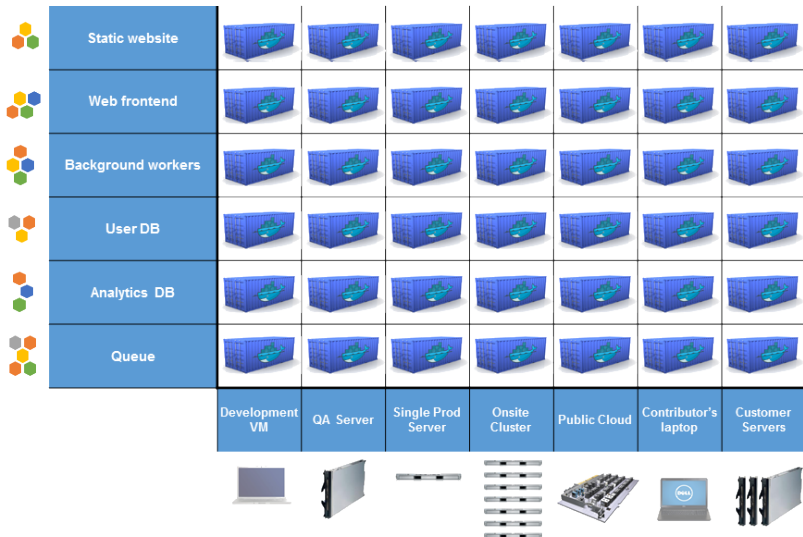
Credit: J. Petazzoni

And software containers apply the same idea



Credit: J. Petazzoni

To solve the same problem



Credit: J. Petazzoni

Benefits for developers

Build once... run anywhere¹

- Portable runtime environment for your app.
- No worries about missing dependencies, packages and other pain points during subsequent deployments.
- Run each app in its own isolated container, so you can run various versions of libraries and other dependencies for each app without worrying.
- Automate testing, integration, packaging...anything you can script.
- Reduce/eliminate concerns about compatibility on different platforms, either your own or your customers.
- Cheap, zero-penalty containers to deploy services. A VM without the overhead of a VM. Instant replay and reset of image snapshots.

Developers focus in the *inside* of the container: code, libs, data...

All Linux servers look the same!

Credit: J. Petazzoni

¹Where “anywhere” *usually* means a x86 server running a modern Linux kernel

Configure once... run anything

- Make the entire lifecycle more efficient, consistent, and repeatable
- Increase the quality of code produced by developers.
- Eliminate inconsistencies between development, test, production, and customer environments.
- Support segregation of duties.
- Significantly improves the speed and reliability of continuous deployment and continuous integration systems.
- Because the containers are so lightweight, address significant performance, costs, deployment, and portability issues normally associated with VMs.

Administrators focus on the *outside* of the container: logging, networking...

All containers can be started, stopped, migrated... the same way!

Docker was launched in 2013 (7 years ago) and became a massive trend.

Github project search “docker” → > 450,000 projects

Moby project on Github (Docker container management system) → > 56k ★

Docker Hub (Image sharing) → > 3M images

According to Stackoverflow's 2019 survey:

- Docker was the third platform developers deploy on:
Linux 53%, Windows 51%, Docker 32%...
- It was the second most loved platform, after Linux.
- More than half of developers use containers.

Reasons for NOT using (Docker) containers (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

Docker internals

Implementation of Virtual Machines (for reference)

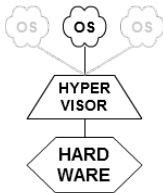
Virtualization is performed by a special software: a **hypervisor**.

Virtualization requires hardware support like *Intel-VT*, *AMD-V*, etc.

Type-1, native or bare-metal hypervisors

These hypervisors run directly on the host's hardware to control the hardware and to manage guest operating systems.

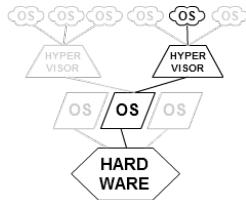
Examples: Nutanix AHV, AntsleOs, Xen, XCP-ng, Oracle VM Server, Microsoft Hyper-V, VMware ESXi



Type-2 or hosted hypervisors

These hypervisors run on a conventional operating system (OS) just as other computer programs do. A guest operating system runs as a process on the host.

Examples: VMware Workstation, VMware Player, VirtualBox, Parallels Desktop for Mac, QEMU



Implementation of Docker containers

Under the hood, Docker is built on the following components:

- The Go programming language
- The following features of the Linux kernel:
 - namespaces,
 - cgroups
 - capabilities
 - (Seccomp, SELinux, AppArmor)...
- The following Open Container Initiative specifications:
 - runtime (ie *container*)
 - image
 - distribution

Let us have a brief look at them to better understand what containers are and how to use them.

You can also check this good presentation by Jérôme Petazzoni ([link](#)).

namespaces

According to **man namespaces**:

A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes.

namespaces supported by Docker

pid processes inside the container will only be able to see other processes inside the same container / pid namespace.

network the container will have its own network stack.

mount the container will have an isolated mount table.

ipc processes inside the container will only be able to communicate to other processes inside the same container via system level IPC.

uts the container will have its own hostname and domain name.

user the container will be able to remap user and group IDs from the host to local users and groups within the container.

cgroup the container will have an isolated view of the cgroup hierarchy.

To better understand: `ls -la /proc/$PID/ns/` for a given process

cgroups (1/2)

According to **man cgroups**:

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 or disk I/O throughput

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

Control freezing, checkpointing and restarting groups of processes

To better understand

- Explore **/sys/fs/cgroup**, cgroups virtual file system
- **docker inspect** some container
- for each process: **/proc/\$PID/cgroup**

Available controllers

- memory** Report and limit of process memory, kernel memory, and swap used.
- devices** Control which processes may create (mknod) devices as well as open them for reading or writing.
- cpu, cpuacct** Account for CPU usage by groups of processes.
 - cpuset** Bind the processes in a cgroup to a specified set of CPUs and NUMA nodes.
- freezer** Suspend and restore (resume) all processes in a cgroup.
- net_cls** Place a classid on network packets created by a cgroup. Can then be used in firewall rules.
- blkio, io** Control and limit access to specified block devices by applying IO control in the form of throttling and upper limits.
- perf_event** Allow **perf** monitoring of the set of processes grouped in a cgroup.
- net_prio** Allow priorities to be specified, per network interface, for cgroups.
- hugetlb** Limit the use of huge pages by cgroups.
 - pids** Limit the number of process that may be created in a cgroup (and its descendants).
 - rdma** Limit the use of RDMA/IB-specific resources per cgroup.

Capabilities

According to **man capabilities**:

Traditional UNIX implementations distinguish two categories of processes: privileged (PID = 0) and unprivileged processes (PID \neq 0).

Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials (usually: effective UID, effective GID, and supplementary group list).

Starting with kernel 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled. Capabilities are a per-thread attribute.

Capabilities list

- | | | | |
|-----------------------|------------------------|------------------|----------------------|
| • CAP_AUDIT_CONTROL | • CAP_IPC_OWNER | • CAP_NET_RAW | • CAP_SYS_PACCT |
| • CAP_AUDIT_READ | • CAP_KILL | • CAP_SETFCAP | • CAP_SYS_PTRACE |
| • CAP_AUDIT_WRITE | • CAP_LEASE | • CAP_SETGID | • CAP_SYS_RAWIO |
| • CAP_BLOCK_SUSPEND | • CAP_LINUX_IMMUTABLE | • CAP_SETPCAP | • CAP_SYS_RESOURCE |
| • CAP_CHOWN | • CAP_MAC_ADMIN | • CAP_SETUID | • CAP_SYS_TIME |
| • CAP_DAC_OVERRIDE | • CAP_MAC_OVERRIDE | • CAP_SYS_ADMIN | • CAP_SYS_TTY_CONFIG |
| • CAP_DAC_READ_SEARCH | • CAP_MKNOD | • CAP_SYS_BOOT | • CAP_SYSLOG |
| • CAP_FOWNER | • CAP_NET_ADMIN | • CAP_SYS_CHROOT | • CAP_WAKE_ALARM |
| • CAP_FSETID | • CAP_NET_BIND_SERVICE | • CAP_SYS_MODULE | |
| • CAP_IPC_LOCK | • CAP_NET_BROADCAST | • CAP_SYS_NICE | |

Open Container Initiative runtime (container) specifications

Container configuration (namespace, cgroups, capabilities, etc.), lifecycle, and how to represent them using JSON files.

A container, when existing, can be in the following states:

- creating** the container is being created: namespace, cgroups, mounts, capabilities, etc. are initialized based on configuration
- created** the runtime has finished the create operation, and the container process has neither exited nor executed the user-specified program
- running** the container process has executed the user-specified program but has not exited
- stopped** the container process has exited (killed or graceful exit)

Abstract operations: create, start, kill, delete

A container has a base image (root FS).

Open Container Initiative image specifications

An image stores the files for the root FS of a *container*, ie the files our containerized program will see.

Problem(s):

- Many containers share the same basis (like Ubuntu, Alpine, Debian, etc.)
- because we do not want to rebuild a complete software stack by hand down to the kernel²

Solution :

- Split images into meaningful **layers**
Ubuntu base, Python dependencies, App...
- **Share common layers** between containers in read-only
- Add a **thin writable layer** on top of this stack of layers
- View this stack as a **single, consistent and writable filesystem**

²Go and Rust are pretty good tools for this, but this is not a very common case.

Image Layers

Efficiently implemented using *Copy-on-Write* (CoW) storage.

Layers have *lower* (base, RO), *upper* (prev. changes, RO) and *diff* (current, RW — if applicable) contents.

Existing implementations

- **Unioning filesystems**

Default solution with Docker

Ex: AUFS, overlayFS

- **Snapshotting (CoW) filesystems**

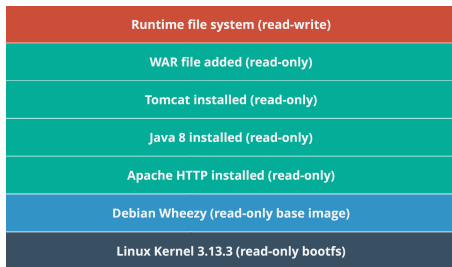
Good idea to test in production

Ex: btrfs, ZFS

- **copy-on-write block devices**

Not appropriate for containers?

Ex: thin snapshots with LVM or device-mapper



API protocol to facilitate distribution of images:

- What is a repository
- How to list, pull, push images
- HTTP API

When using Docker, you think about *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*

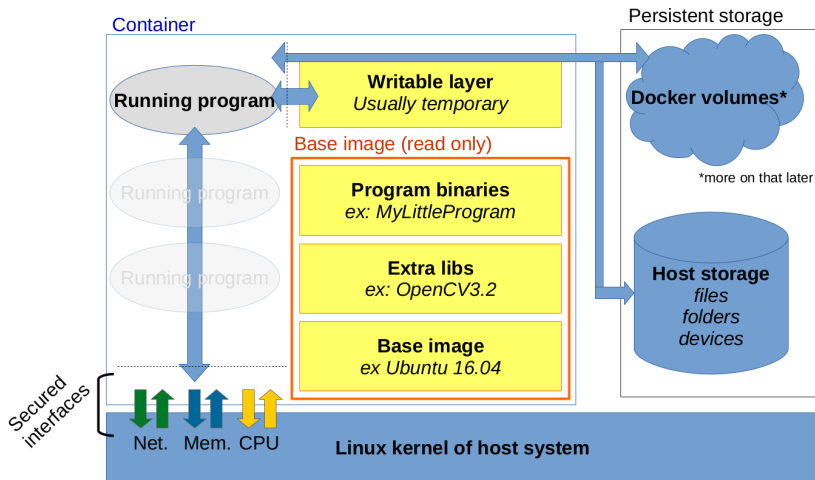


Figure 2: Container and base image

- 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 variables, kernel configuration and automatically generated files: for hostname, DNS resolution, etc.
 - *an abstract view of a group of processes, not even a single kernel object!*
- Programs run “inside” *containers*
 - Such programs are “jailed” with limited capabilities (such as file writes, network, memory, etc.)
 - They see the container’s filesystem, processes, networks, users...
 - 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 running program (but nothing forces you to do so)

Docker (as a product) is just a few things

- A framework to run programs with different software stacks and capabilities.
 - Using existing Linux Kernel features.
- A layered filesystem trick.
- A set of tools to create those stacks, manage them and run programs.
 - Everything can be done by hand with standard tools, but Docker it much much easier and quicker to use.
- An ecosystem: Hub, Dockerfiles, specifications, community, etc.

Using Docker

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`

1. Obtain an image
2. Create a container to run a program (a shell) with this image
3. Launch this program (a shell)
4. Inspect and alter the content of the image
5. Quit the program and check what is left

More about all of this during the practice session.

1. Obtain an image | $\emptyset \rightarrow$ **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** \rightarrow **container**
 - = Define isolation policy: File sharing with host? Ports exposed? Transient?
3. Start the container | **container** \rightarrow **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** \rightarrow **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 | $\emptyset \rightarrow$ **image** on local disk
 - `docker image pull USER/IMAGENAME:TAG`
 - `docker image import ARCHIVE`
 - `docker image build ...`
2. Create a container from image | **image** \rightarrow **container**
`docker container create --name CONTAINER_NAME IMAGE`
3. Start the container | **container** \rightarrow **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** \rightarrow **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

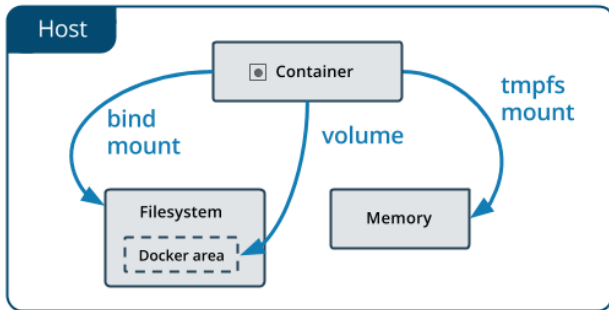


Figure 3: Storage spaces for a container

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

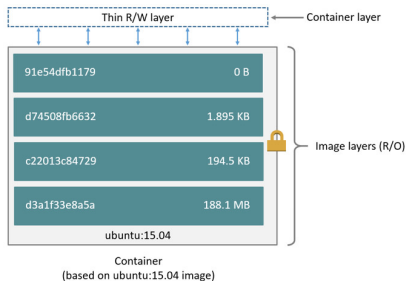


Figure 4: Container layers vs base image

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

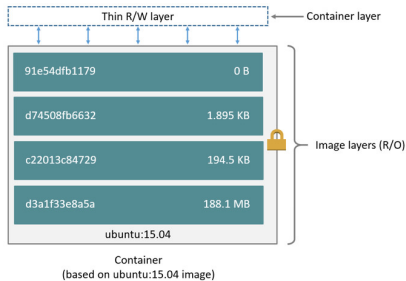


Figure 5: Container layers vs base image

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

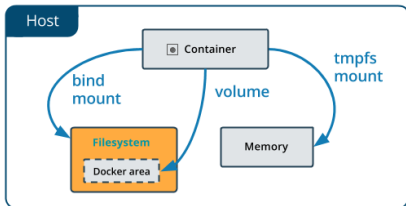


Figure 6: Bind mounts

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)

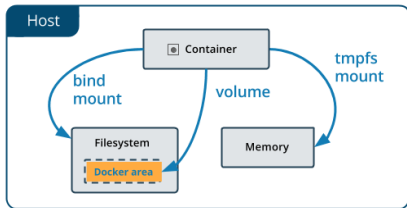


Figure 7: Volumes

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

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

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

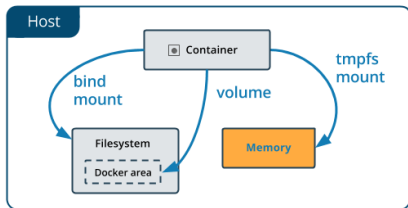


Figure 8: Tmpfs

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
```

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.

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.

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

- 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

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