Docker containers.
Building & running

Hello!

I am
Roberto Innocente

I work at SISSA/ITCS and since some years I am involved with docker containers.
I designed the first *dockerization* of Quantum Espresso in Summer 2016.
New version with QE 6.0 is available at
https://hub.docker.com/r/rinnocente/qe-full-6.0.

Docker containers/1

At the end of 2013, **dotCloud, Inc.** , a cloud service provider, made public and opensource its tool for managing customer apps : a client/server application called **docker.**

In a few months it had a phenomenal attraction for many developers and users.

This convinced **dotCloud** to make its tool the new focus of its business and to change its name in **Docker, Inc.**

Docker containers/2

As you probably know, a *docker* or *longshoreman* is someone who loads and unloads goods from ships on the docks of the harbour.

Left
“On the waterfront”
Elia Kazan, 1954
Featuring Marlon Brando
working as a docker.

Docker containers/3

Today, dockers have mostly to manage standardized boxes for transferring goods called **containers.**

This had a tremendous impact on shipping costs. Almost all operations are now automated with the help of *ad hoc* machines. Today almost no goods are loaded/unloaded from a ship if not in a container

Cloud computing and containers

Cloud computing refers to the situation in which you get a computer service from the Internet on-demand in real-time, you don’t really care where the service is run, and you pay for how long and what you use.

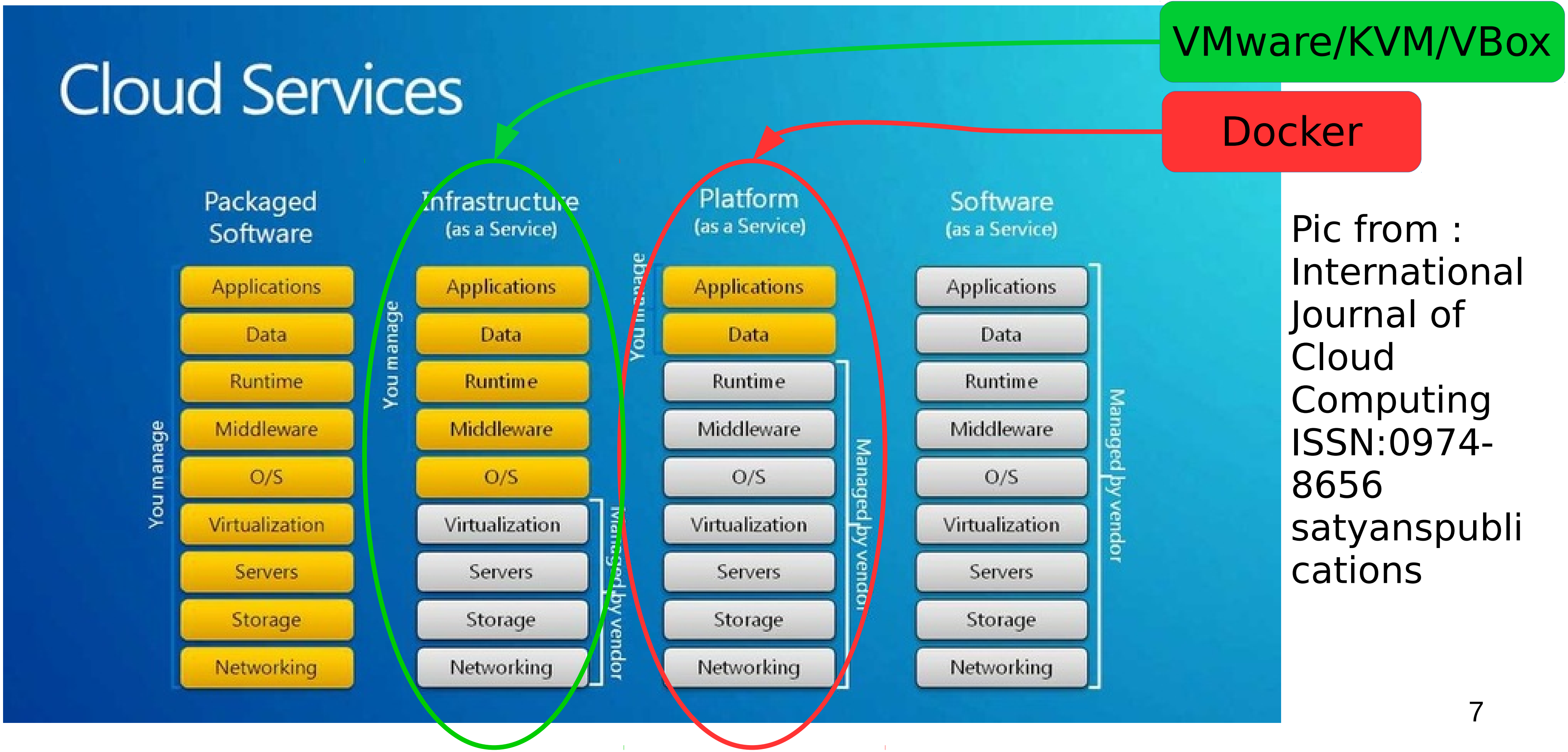
The usual cloud providers like Amazon/Rackspace/Ibm/Microsoft are usually lending virtual machines :

- **IaaS (Infrastructure as a Service**) : it is then responsibility of the customer to dress it up with an OS, middleware, libraries, data and apps

With Docker **cloud computing** can provide also the OS, middleware and libraries (like dotCloud was doing) :

- **PaaS(Platform as a Service) :** the only responsabilities that remain on the customer’s shoulders are the management of data and apps.

Cloud services (IaaS,PaaS,SaaS) : where is Docker ?



Pic from : International Journal of Cloud Computing ISSN:0974-8656 satyanspublications

1

Virtualization methods

Full Virtualization : Hypervisors, VM

OS level/
lightweight Virtualization : Containers

Full/lightweight virtualization

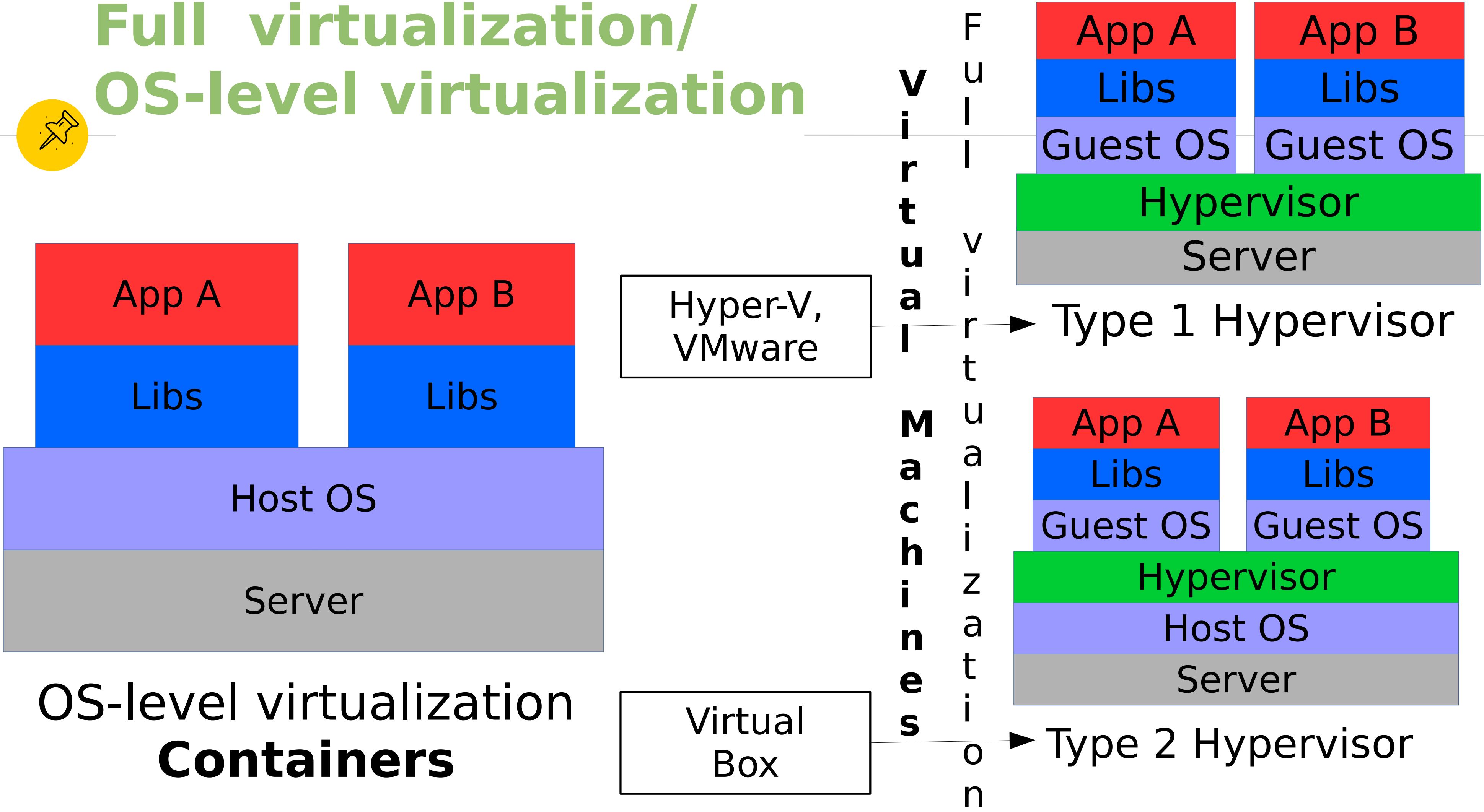
Probably you are aware of **Virtual Machines** and the way to use them.
The usual *virtual machines* depend on a program called **hypervisor** that pretends to be a bare machine to the upper software, so that you can mount an OS on it. Of course this implies a performance penalty.

There is another more **lightweight virtualization** or **os-level virtualization**, that reached maturity later on Linux, in which the OS encapsulates an environment by means of software barriers. This insulated environment is called a **container.** It is more efficient because container processes are simply host processes.

Important fact :

- a **container** starts/stops in hundredths of milliseconds
- a **virtual machine** starts/stops in tens of seconds (~ 100x)

Full virtualization/ OS-level virtualization



App A

Libs

App B

Libs

Host OS

Server

OS-level virtualization
Containers

Hyper-V, VMware

Virtual Box

Virtual Machines

Full virtualization

App A

Libs

Guest OS

App B

Libs

Guest OS

Hypervisor

Server

Type 1 Hypervisor

App A

Libs

Guest OS

App B

Libs

Guest OS

Hypervisor

Host OS

Server

Type 2 Hypervisor

2

Linux cgroups

Linux Control Groups

Containers don’t exist inside the linux kernel. They are runtime creatures that are generated using two important features added to the Linux kernel from 2006 on :

- Control groups
- Namespaces

cgroups/1

cgroups (= control groups) : is a Linux kernel feature that limits, accounts and isolates resources used by a set of processes.

Added to the Linux kernel initially by Google engineers Paul Menage and Rohit Seth in 2006 and named *process containers.*

Renamed control groups to avoid confusion with other entities, appeared in official kernel 2.6.24 in 2008

(this version is now called cgroups-v1).

Development and maintenance passed then to Tejun Heo, who rewrote and redesigned cgroups from 2013 on.

This rewrite is now called cgroups-v2 and its documentation appeared in linux 4.5 on March,14 2016.

cgroups/2

- Processes in linux are organized hierarchically : a **single tree** (all processes are born out of the initial ***init*** process and inherit resources from the parents)

- cgroups are similar, just are organized as a ***forest*** (multiple trees) where there is also the inheritance from parents

Ubuntu resource groups :

- *blkio*
- *cpu,cpuacct*
- *cpuset*
- *devices*
- *freezer*
- *hugetlb*
- *memory*
- *net_cls,net_prio*
- *perf_event*
- *pids*
- *name=systemd*

cgroups/3

They provide control of some resources over a set of processes (a cgroup)   :

- Limit : can limit memory, cpu, io, ..
- Accounting : report use of resource by cgroups
- Priority : can change sharing of resources of some cgroups vs other
- Control : freezing, checkpoint, restarting of cgroups

Software that use cgroups (= control groups) :

- **Docker**
- **Linux Containers (LXC)**
- **libvirt**
- **systemd**
- **Open Grid Scheduler**/Grid Engine
- Google's **lmctfy** (“let me contain that for you”), now merged with Docker **libcontainer** library.

cgroups/5

With docker usually you will not need to access cgroups directly.

Resource limits and accounting will be established by :

- Docker daemon cgroup/ulimit options
  - dockerd --parent-cgroup … *# will be the parent cgroup of all containers*
  - dockerd --default-ulimit=[] *# Default ulimits for all containers*
- Docker run options :
  - docker run --blkio-weight value # Block IO (relative weight), between 10 and 1000
    --cpu-shares int # CPU shares (relative weight up to 1024)
    --cpuset-cpus string # CPUs in which to allow execution (0-3, 0,1)
    --memory string # Memory limit
    --ulimit value # Ulimit options (default [])

3

Linux namespaces

Linux Namespaces

Namespaces/1

Or better Linux Namespaces. They are a linux kernel feature (for the mnt namespace *[chroot]* appeared in 2002 but most of the work appeared recently in kernel 3.8) that isolates and virtualizes resources of a collection of processes (cgroups):

- Filesystems : *mnt*
- Pid : *pid*
- Network : *net*
- Userid : *user*
- Ipc : *ipc*
- Cgroup root dir : *cgroup*
- Host/Domainname : *uts*

Every process is associated with a namespace and it can see only the resources associated with that namespace.
Namespaces can be created and joined.
After boot all processes belong to a single namespace.

Linux namespaces were inspired by the more general implementation in Bell Lab Plan9 O.S.

```
ls -l /proc/[pid]/ns/
```

Namespaces/2

```
inno@geist:~$ ls -l /proc/$PPID/ns

total 0

lrwxrwxrwx 1 inno inno 0 Mar 25 11:47 ipc -> ipc:[4026531839]

lrwxrwxrwx 1 inno inno 0 Mar 25 11:47 mnt -> mnt:[4026531840]

lrwxrwxrwx 1 inno inno 0 Mar 25 11:47 net -> net:[4026531957]

lrwxrwxrwx 1 inno inno 0 Mar 25 11:47 pid -> pid:[4026531836]

lrwxrwxrwx 1 inno inno 0 Mar 25 11:47 user -> user[4026531837]

lrwxrwxrwx 1 inno inno 0 Mar 25 11:47 uts -> uts:[4026531838]

root@geist:~# readlink   /proc/$PPID/ns/user

user:[4026531837]
```

For each namespace kind every process is

assigned a symbolic link in */proc/<pid>/ns.*

The link points to an *inode* that is the same for every process in the same namespace.

When a namespace is not referenced it is deleted automatically. References are :

- A process belonging to the ns
- An open file descriptor pointing to the ns symlink
- A bind mount of ns symlink

Namespaces

What can manage namespaces ?
3 syscalls :

- *clone(2)* : there are flags to specify to which namespace to migrate the new process
- *unshare(1)* : flags to specify when the process will be migrated out of the current namespace where to go
- *setns(2)* : specifies in which namespace to migrate

Linux manual clone(2) :
“CLONE_NEWIPC (since Linux 2.6.19)
If CLONE_NEWIPC is set, then create the process in a new IPC namespace. If this flag is not set, then (as with fork(2)), the process is created in the same IPC namespace as the calling process. This flag is intended for the implementation of containers.”
Linux manual unshare(1) :
unshare [options] program [arguments]
unshares the indicated namespaces :
unshare mount namespace ls /mnt
unshare network namespace …
...

Inside a container : namespace insulation

General security features help :

- **Apparmor**
- **Selinux**

But the most important feature for multi tenant installations is :

- **User namespace remapping** : whatever are uid and gid inside the container, interactions with host happens with controlled uid/gid
  - Applied when server/daemon is started with –**userns-remap=default** or similar

4

Linux containers

Linux Containers :

Requires disambiguation !!!!

Linux containers : disambiguation

With this term we indicate :

- A subtree of linux processes encapsulated by means of the cgroup and the namespace linux kernel features (like **lxc**, **docker** do)

- A project started in 2008, named **LXC (Linux containers),** as the tool that it produced, for the management of cgroups/namespaces to obtain these encapsulated groups of processes

At the beginning **docker** used **LXC** as a base, but after the opensourcing, made by google in 2014, of its **libcontainer** (container library), docker used and evolved this last.

Now supported by many there is a consortium called OpenContainer Initiative (**OCI**) for an open specification of the image format and runtime env (based on docker v2 image format and coreOS appC).

LXC (Linux Containers)

LXC is a userspace interface to the Linux kernel container features.

Started in 2008.

Initially used by docker as a base..

Aim is to create an environment as much isolated as possible but without the need of a new kernel.

It creates an environment somewhere in between a **chroot** and a full **virtual machine**.

It uses :

- Kernel namespaces (ipc,uts,mount,pid, …)
- Apparmor and SELinux profiles
- Seccomp policies
- Chroots
- CGroups

It is made up of :

- Liblxc library
- Language bindings :
  - Python 3,2
  - Lua
  - Go
  - Haskell

It's missing all the docker Ecosystem !!

LXC/2

Create/destroy a permanent container :

- **lxc-create** -n mycont
- **lxc-destroy** -n mycont

Running/stopping an app in a container:

- **lxc-execute** -n mycont /bin/bash
- **lxc-start** -n mycont /bin/bash
- **lxc-stop** -n mycont

Setting cgroup :

- **lxc-cgroup** -n mycont cpuset.my
- lxc-group -n mycont cpu.shares 512

Freeze/unfreeze container:

- **lxc-freeze** -n mycont
- **lxc-unfreeze** -n mycont

Connect to an available tty:

- **lxc-console** -n mycont

Getting info :

- lxc-ls
- **lxc-info** -n mycont
- **lxc-monitor** -n “mycont|yourcont”

Waiting for a container :

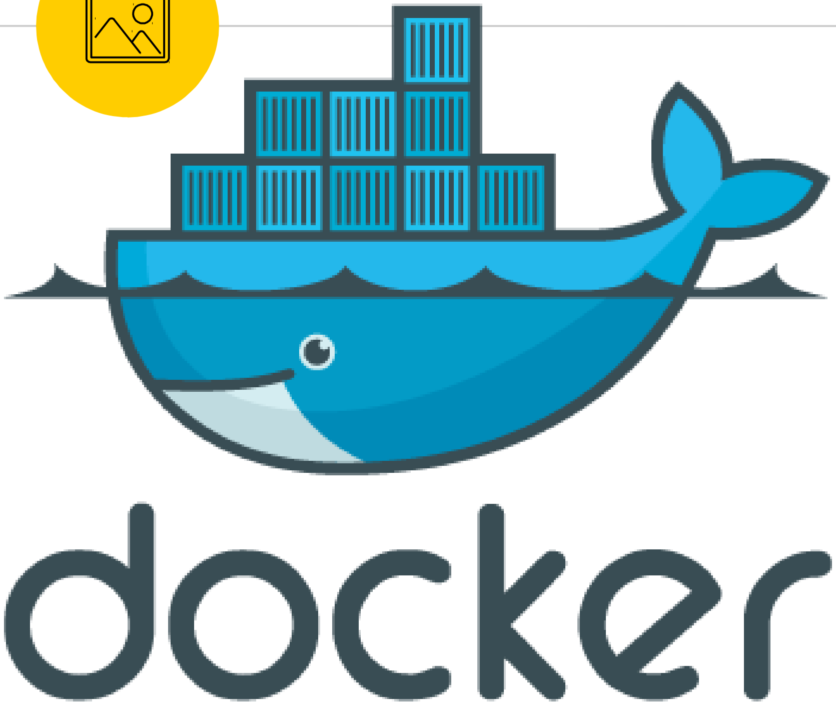
- **lxc-wait** -n mycont -s STOPPED &
- PID_TO_WAIT=$!
- **lxc-execute** -n mycont myapp
- wait $PID_TO_WAIT

It’s missing all the docker Ecosystem !!

5

Docker containers

Docker Containers

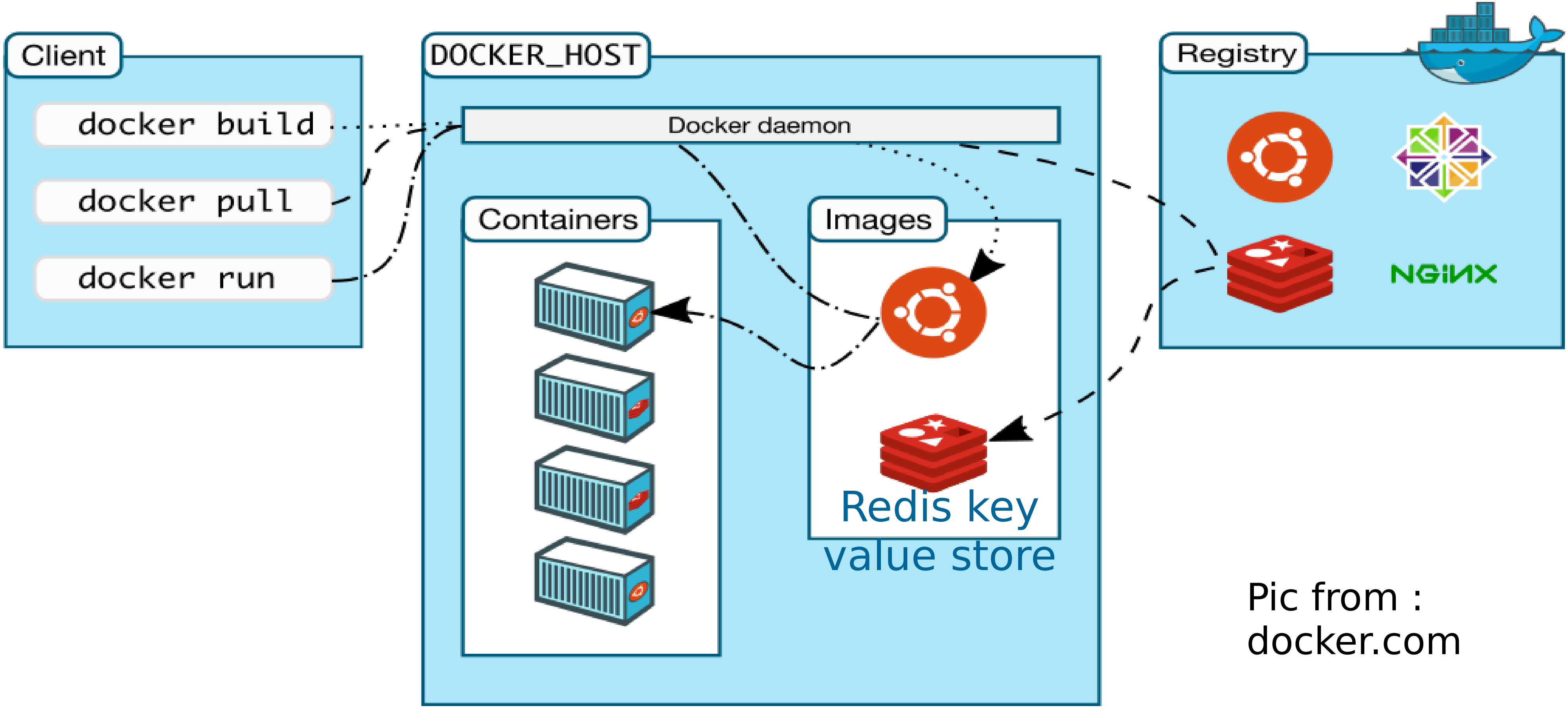


Docker

- Build once, configure once
- Deploy everything, everywhere

It’s incredible but it is really so ! The developer can transmit all his/her environment to run the apps to the test and deployment workgroups. (End of the “it works on my laptop!” developers’ assertion)

Docker Architecture



Redis key value store

Pic from : docker.com

Let's do it !

Docker check :

- **docker version**
- **docker   info**
-

```
Client:
 Version:      1.12.3
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   6b644ec
 Built:        Wed Oct 26 22:
 OS/Arch:      linux/amd64

Server:
 Version:      1.12.3
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   6b644ec
 Built:        Wed Oct 26 22:
 OS/Arch:      linux/amd64
```

```
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 62
Server Version: 1.12.3
Storage Driver: aufs
 Root Dir: /var/lib/docker/362144.362144/aufs
 Backing Filesystem: extfs
 Dirs: 75
 Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: null host bridge overlay
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Security Options: apparmor seccomp
Kernel Version: 4.8.0-29-generic
Operating System: Ubuntu 16.10
OSType: linux
Architecture: x86_64
CPUs: 4
Total Memory: 7.549 GiB
Name: geist
ID: A4Z4:I7V2:XYOP:NQYQ:HRG.........
Docker Root Dir: /var/lib/docker/362144.362144
Debug Mode (client): false
Debug Mode (server): false
Username: rinnocente
Registry: https://index.docker.io/v1/
WARNING: No swap limit support
Insecure Registries:
 127.0.0.0/8
```

Docker , first containers :

- **docker run hello-world**   ***# in every***
  # cs exercise there is 1 !
- **docker run -it busybox**

And you are in the busybox shell.
Exit with CTRL-D or CTRL-P/CTL-Q.

CTRL-D or **CTRL-P, CTRL-Q** ?

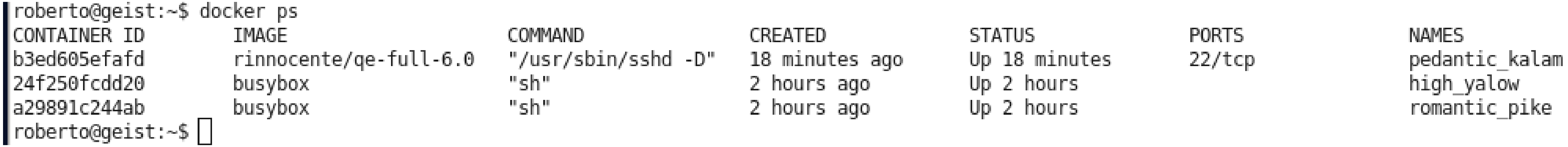
When the PID 1 of a container is a shell (the command specified on the CMD or ENTRYPOINT line of the dockerfile or in the docker run command) :

- If you exit the shell with **CTRL-D** or **exit** the shell dies and the container dies when PID 1 dies

- Exiting with **CTRL-P, CTRL-Q** will keep the shell alive and therefore the same for the containers

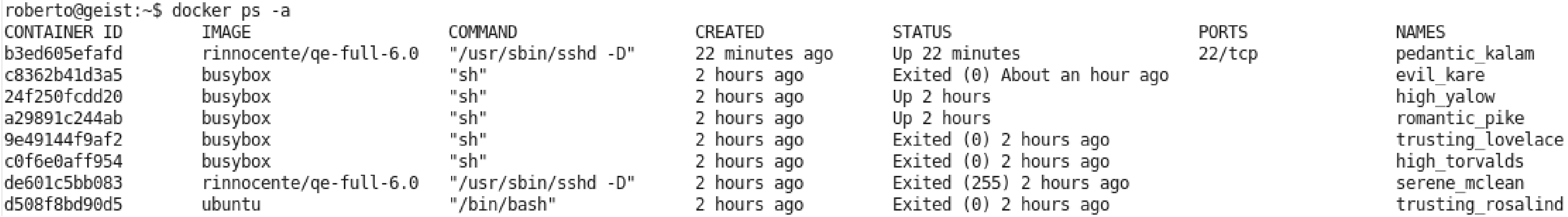
Docker: which containers exist?

$ docker ps
Running containers.



```
roberto@geist:~$ docker ps
CONTAINER ID        IMAGE                     COMMAND                CREATED             STATUS              PORTS               NAMES
b3ed605efafd        rinnocente/qe-full-6.0    "/usr/sbin/sshd -D"    18 minutes ago      Up 18 minutes       22/tcp              pedantic_kalam
24f250fcdd20        busybox                   "sh"                   2 hours ago         Up 2 hours                              high_yalow
a29891c244ab        busybox                   "sh"                   2 hours ago         Up 2 hours                              romantic_pike
roberto@geist:~$ 
```

$ docker ps -a
All containers not yet removed.



```
roberto@geist:~$ docker ps -a
CONTAINER ID        IMAGE                     COMMAND                CREATED             STATUS                          PORTS               NAMES
b3ed605efafd        rinnocente/qe-full-6.0    "/usr/sbin/sshd -D"    22 minutes ago      Up 22 minutes                   22/tcp              pedantic_kalam
c8362b41d3a5        busybox                   "sh"                   2 hours ago         Exited (0) About an hour ago                        evil_kare
24f250fcdd20        busybox                   "sh"                   2 hours ago         Up 2 hours                                          high_yalow
a29891c244ab        busybox                   "sh"                   2 hours ago         Up 2 hours                                          romantic_pike
9e49144f9af2        busybox                   "sh"                   2 hours ago         Exited (0) 2 hours ago                              trusting_lovelace
c0f6e0aff954        busybox                   "sh"                   2 hours ago         Exited (0) 2 hours ago                              high_torvalds
de601c5bb083        rinnocente/qe-full-6.0    "/usr/sbin/sshd -D"    2 hours ago         Exited (255) 2 hours ago                            serene_mclean
d508f8bd90d5        ubuntu                    "/bin/bash"            2 hours ago         Exited (0) 2 hours ago                              trusting_rosalind
```

$ docker rm [**-f**] *container-id* *# remove container/even if running*

Docker: which images exist?

$ docker images
Local images.

$ docker search busybox ***#standared repo***
Images at standard registry **index.docker.io**

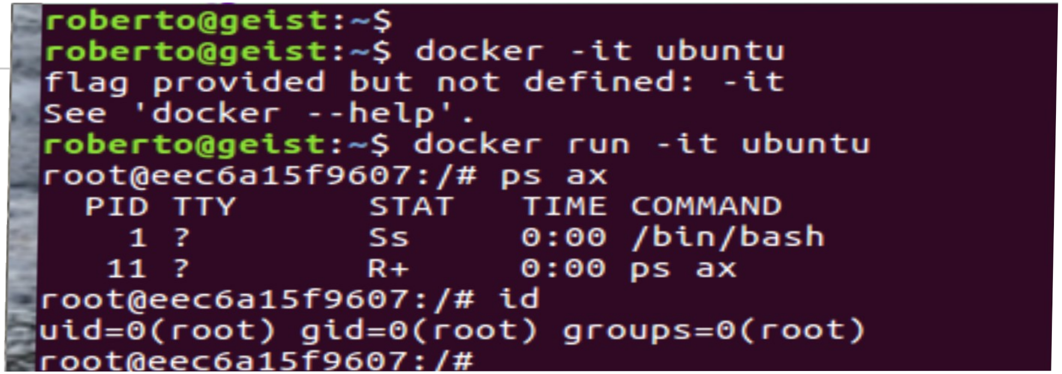
$ docker rmi [**-f**] *image-id* *# remove image/even if container*
# is using it

docker cleaning

Cleaning line scripts **:**

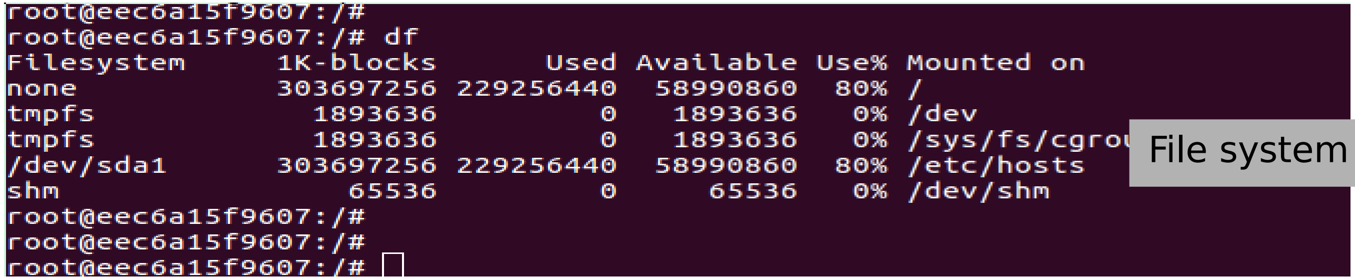
```
$ docker images -aq
$ docker ps -aq

$ docker rm -f    `docker ps -aq`
$ docker rmi -f   `docker images -aq`
```

Inside a container : namespace insulation



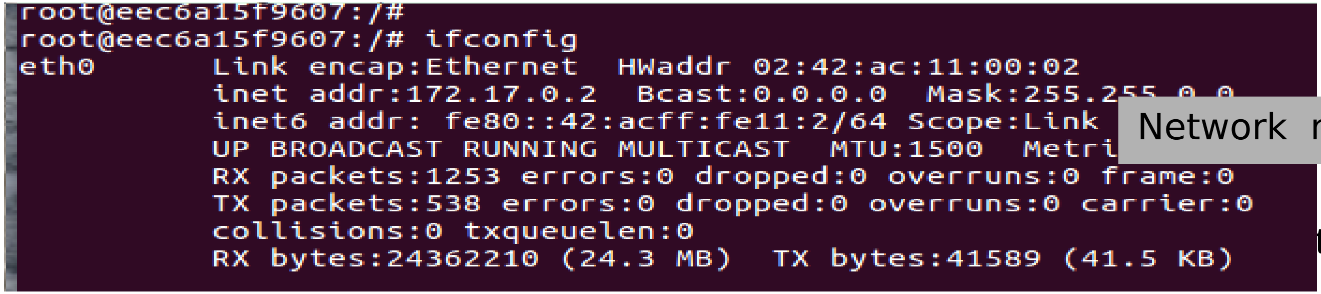
```
roberto@geist:~$
roberto@geist:~$ docker -it ubuntu
flag provided but not defined: -it
See 'docker --help'.
roberto@geist:~$ docker run -it ubuntu
root@eec6a15f9607:/# ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:00 /bin/bash
   11 ?        R+     0:00 ps ax
root@eec6a15f9607:/# id
uid=0(root) gid=0(root) groups=0(root)
root@eec6a15f9607:/#
```

PID, user, groups namespaces



```
root@eec6a15f9607:/#
root@eec6a15f9607:/# df
Filesystem     1K-blocks      Used Available Use% Mounted on
none           303697256 229256440  58990860  80% /
tmpfs            1893636         0   1893636   0% /dev
tmpfs            1893636         0   1893636   0% /sys/fs/cgrou
/dev/sda1      303697256 229256440  58990860  80% /etc/hosts
shm                65536         0     65536   0% /dev/shm
root@eec6a15f9607:/#
root@eec6a15f9607:/#
root@eec6a15f9607:/# 
```

File system namespace



```
root@eec6a15f9607:/#
root@eec6a15f9607:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metri
          RX packets:1253 errors:0 dropped:0 overruns:0 frame:0
          TX packets:538 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:24362210 (24.3 MB)  TX bytes:41589 (41.5 KB)
```

Network namespace

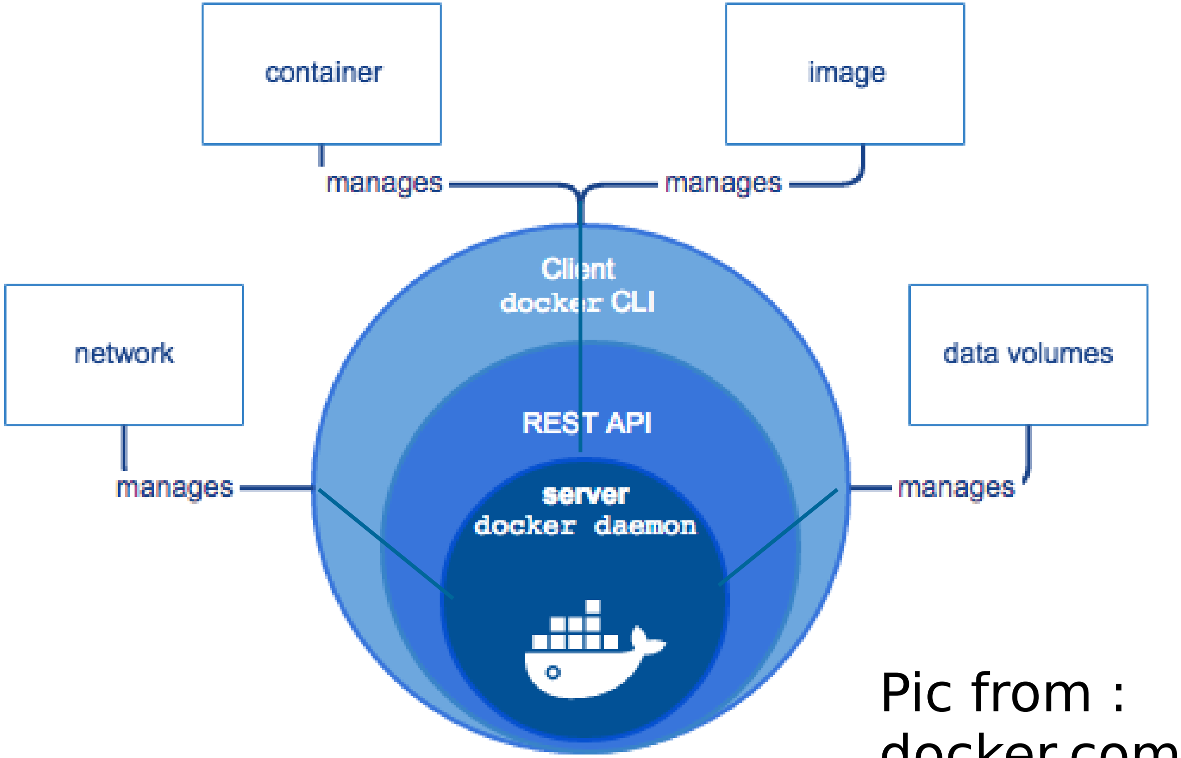
Major components : docker-engine

What is Docker Engine ?

Docker Engine is the Client/Server app, once called simply Docker, made of :

- A CLI client : the **docker client**
- A server: the **docker daemon**
- A **REST API** used by the client to communicate with the server

The objects (containers, images, data volumes, networks) are all managed by the server, according to instructions it receives through the REST API.



Pic from : docker.com

Universality of a docker app

The next slide will show you that the universality of a docker app is real.

A docker image can run everywhere !

That is, can run anywhere there is a docker daemon/server running, but you can run a docker daemon natively on :

- **Linux** (kernel at least version 3.10)

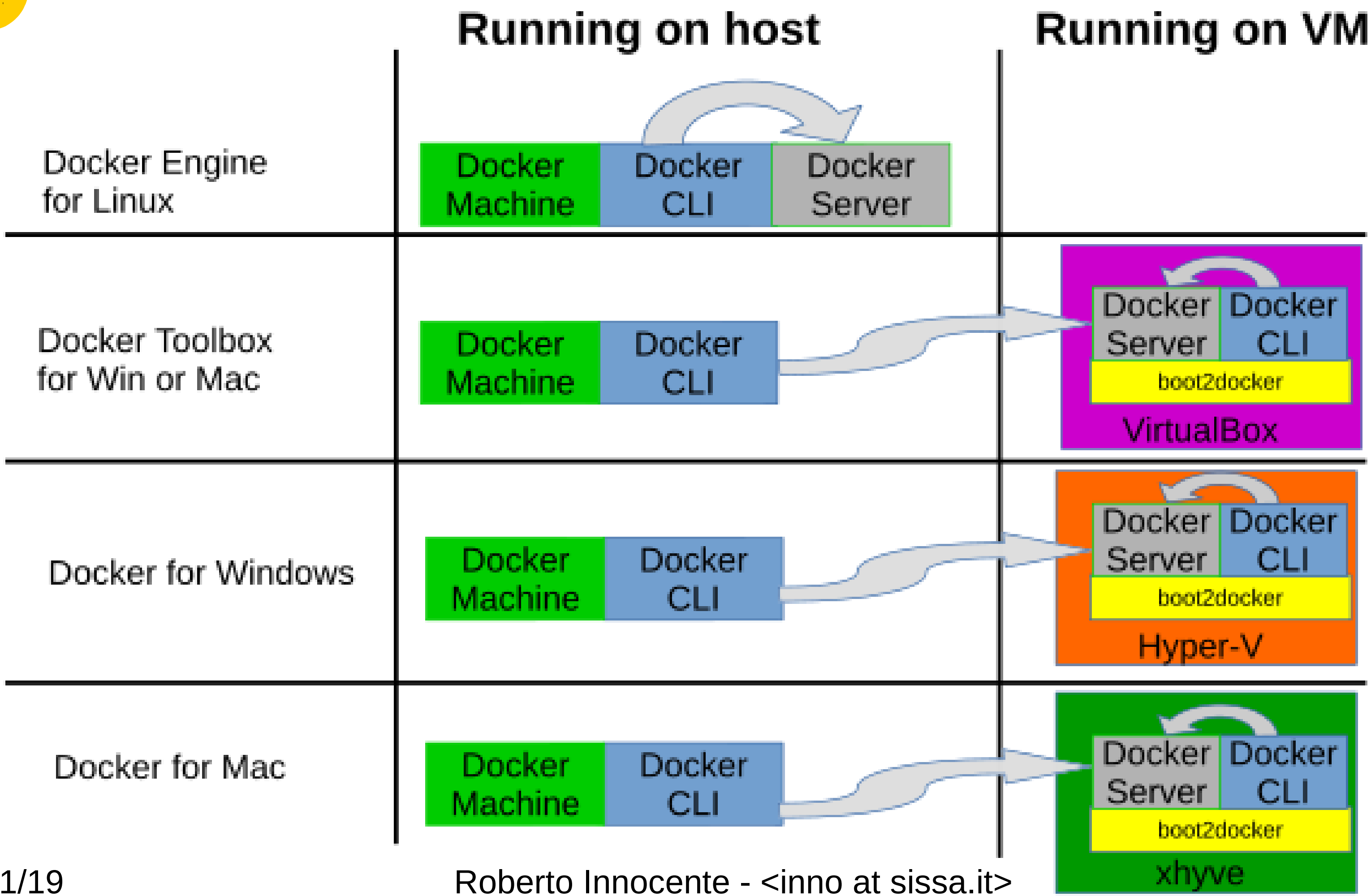
And via a *virtual machine* on :

- **Windows**
- **MacOS**

Installing **Docker Toolbox** on Win or Mac installs also VirtualBox and on it

a Linux stripped down kernel just to run containers called **boot2docker..**

Docker versions



	Running on host	Running on VM
Docker Engine for Linux	Docker Machine Docker CLI Docker Server	
Docker Toolbox for Win or Mac	Docker Machine Docker CLI	Docker Server Docker CLI boot2docker VirtualBox
Docker for Windows	Docker Machine Docker CLI	Docker Server Docker CLI boot2docker Hyper-V
Docker for Mac	Docker Machine Docker CLI	Docker Server Docker CLI boot2docker xhyve

Major components : images/layers

Docker images

Docker **images** are a kind of **root file system (rootfs)** for containers :

- They don't need kernels and modules (containers share the running host kernel)
- They don't need many intialization tools or scripts
- Usually they are minimal : include only what is needed by the apps inside (most importantly shared libraries)

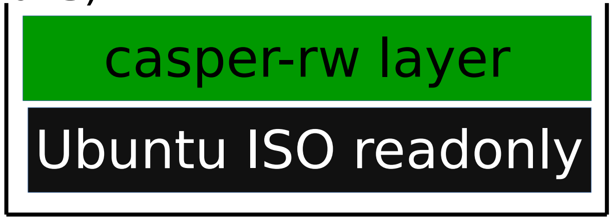
They are **layered**, that is, not monolithic, but made of different layers in such a way that they form a tree, **reusing lower layers**.

Union File Systems/1

Maybe you have used an ubuntu USB live distro with persistent storage. This setup uses a **union file sytem** (in particular Ubuntu uses preferably **aufs**).

A union file system merges at the user level the contents of multiple file systems.

In this simple setup the base fs (a distro ISO) is mounted read only, the upper one is made from a file named **casper-rw** and mounted read/write through a **loop** device driver.



How it works with the file or dir *name-of* :

- **reads** : the file or dir *name-of* is searched in the **casper-rw layer.** If it is there that one is returned. If not the rw layer is searched for *.wh.name-of* (**whiteout file**: the file/dir was deleted in the r/w layer) and in case it exists returns file does not exist. The search is eventually then continued on the ISO fs.
- **writes** : the file or dir is written after eventually being completely copied (**COW : copy-on-write**) on the **casper-rw layer**

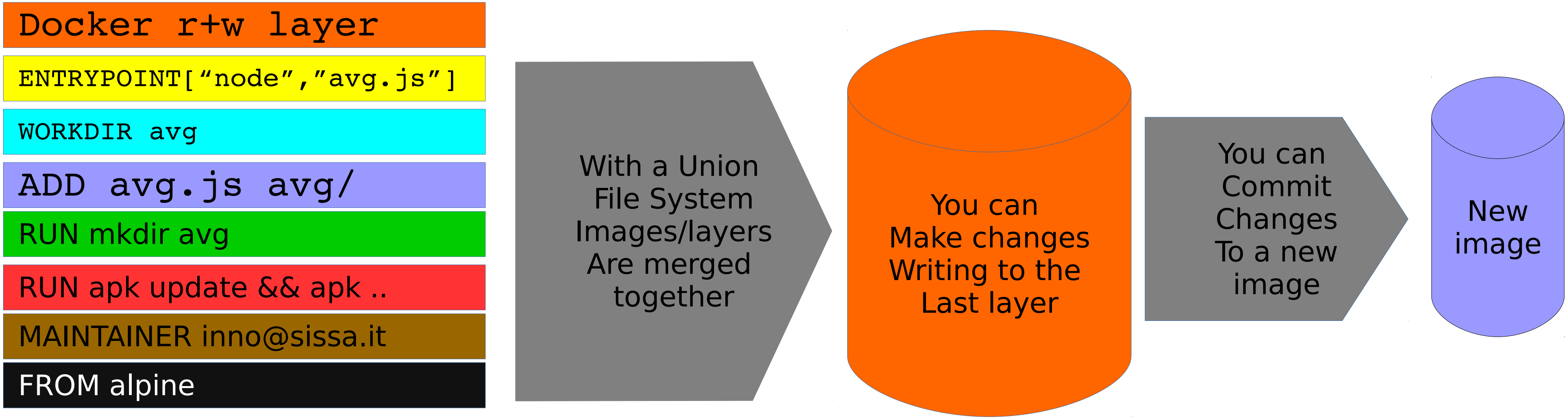
Docker Union File Systems/2

Docker can use many different union file systems : aufs, devicemapper, btrfs, overlayfs.
For a long time no UFS was accepted in the Linux kernel.
Docker can use what it finds :

- **aufs** is a stable and proven version, it is used on ubuntu
- **devicemapper** is used usually on RedHat
- **overlayfs** is a newcomer, but it was accepted in the official linux kernel, so expect its use will raise

Union file system again

- Used by Docker for building containers rootfs from images :
  - For every line in the docker file a new image layer is created, they are all read-only



Docker r+w layer

ENTRYPOINT[“node”,”avg.js”]

WORKDIR avg

ADD avg.js avg/

RUN mkdir avg

RUN apk update && apk ..

MAINTAINER inno@sissa.it

FROM alpine

With a Union File System Images/layers Are merged together

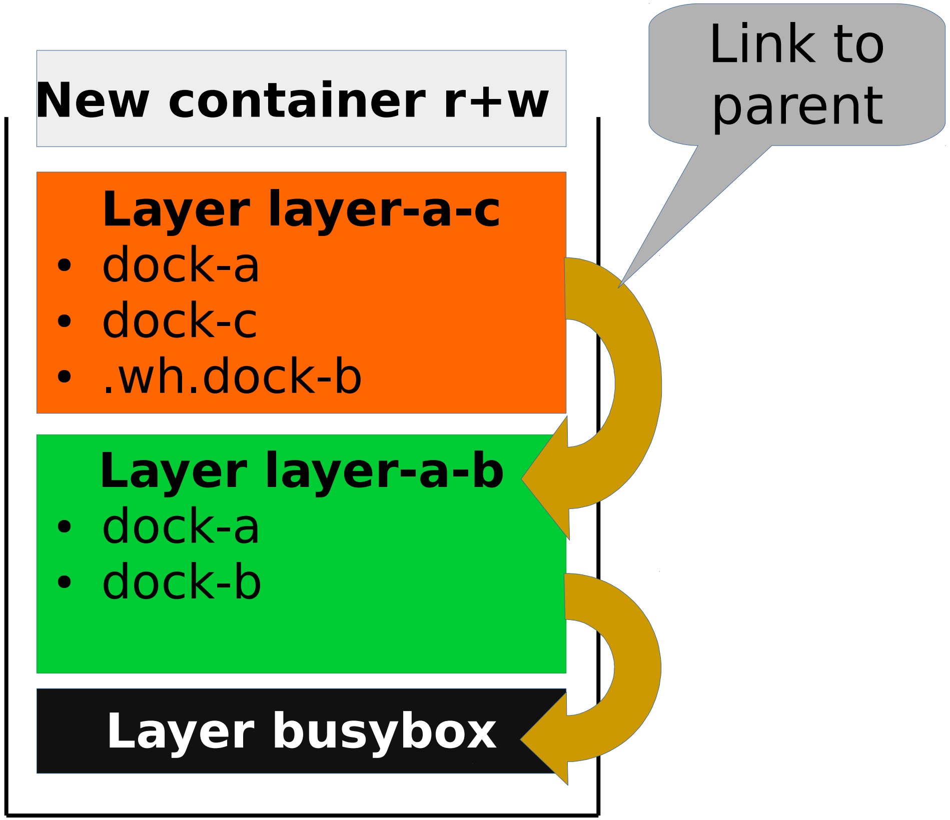
You can Make changes Writing to the Last layer

You can Commit Changes To a new image

New image

Docker layered images

```
$ docker run -it busybox
/ # echo dock-a >dock-a
/ # echo dock-b >dock-b
/ # CTRL-P CTRL-Q

$ docker ps
$ docker diff   0c06
$ docker commit 0c06   layer-a-b
$ docker rm -f   0c06
$ docker run -it layer-a-b
/ # echo new-dock-a >dock-a
/ # echo dock-c >dock-c
/ # rm dock-b
/# CTRL-P CTRL-Q
$ docker ps
$ docker diff   de1
$ docker commit   de1   layer-a-c
```



New container r+w

Layer layer-a-c
- dock-a
- dock-c
- .wh.dock-b

Layer layer-a-b
- dock-a
- dock-b

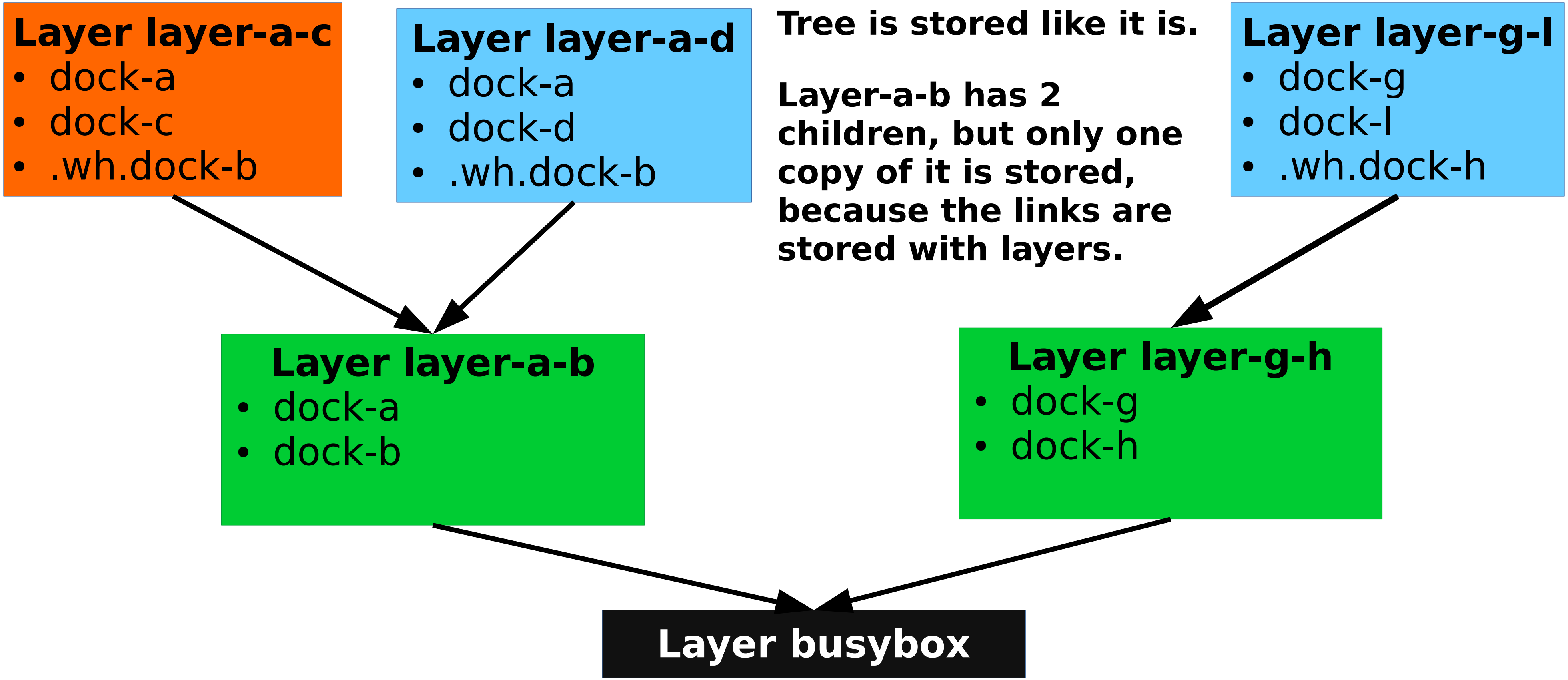
Layer busybox

Link to parent

Tree of images/layers : re-use of layers through links

Tree is stored like it is.

Layer-a-b has 2 children, but only one copy of it is stored, because the links are stored with layers.



Layer layer-a-c

- dock-a
- dock-c
- .wh.dock-b

Layer layer-a-d

- dock-a
- dock-d
- .wh.dock-b

Layer layer-g-l

- dock-g
- dock-l
- .wh.dock-h

Layer layer-a-b

- dock-a
- dock-b

Layer layer-g-h

- dock-g
- dock-h

Layer busybox

Memory and Disk Space used by 100 containers/100 virtual machines running a web server

Virtual Machines :

Disk at least 10GB virtual disk per machine = 1,000 GB

Memory at least 2 GB per virt.machine = 200 GB

Management burden
100 OS

Containers:

Disk 5 GB disk per container = 500 GB

Memory 500 MB per container = 50 GB

Management burden
1 OS

Docker layered images

```
$ docker history image

$ docker save
$ docker load

$ docker export
$ docker import
```

How to move around docker images (!!not containers !!)

Produces a tarred repository of image layers :

- **$ docker save** *IMAGE [IMAGE ..] >image.tar*
- **$ docker save -o** *image.tar IMAGE [IMAGE ..]*

All layers and parent layers are saved with their tags (that is all layers + metadata singularly)..

Loads a tarred repository of images :

- **$ docker load** *# from stdin*
- **$ docker load** *-i image.tar*

Major components : containers

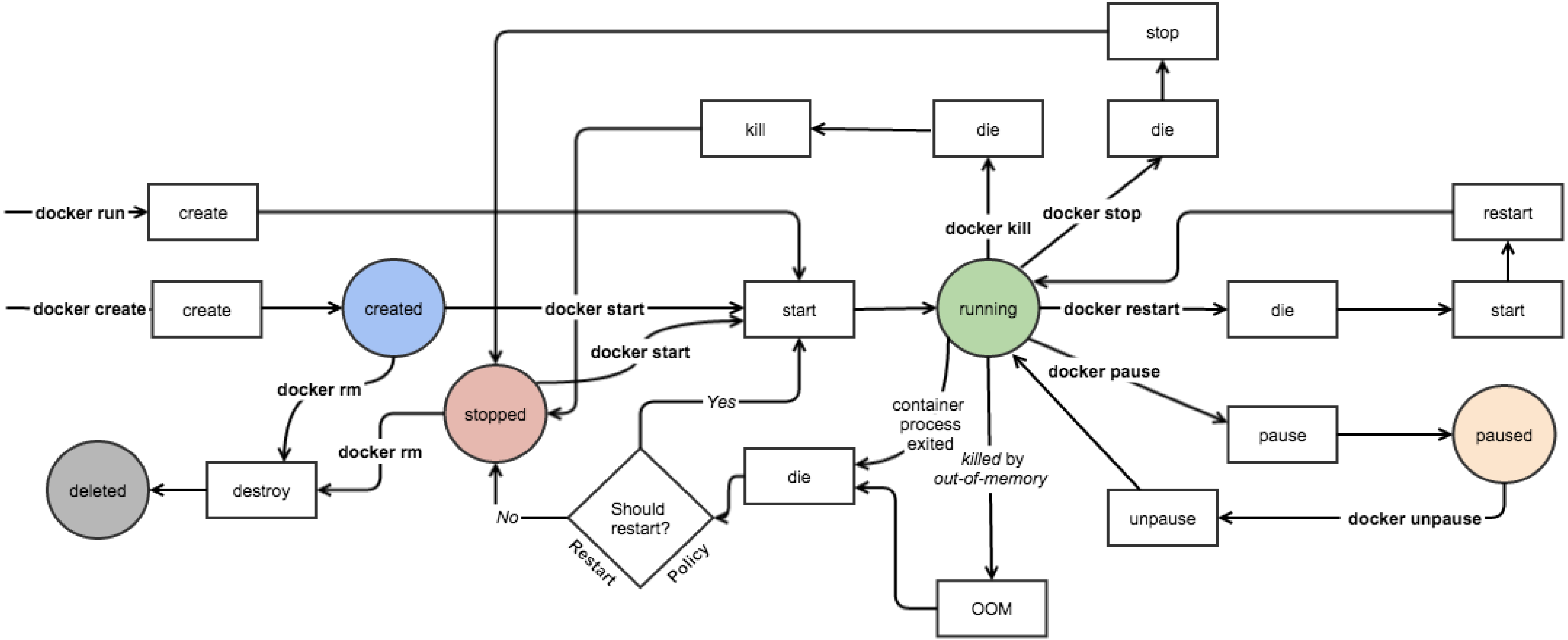
What goes on
when you run a container ?

$ docker run -it ubuntu /bin/bash

1. Trough the REST API the instruction is sent to the server
2. The **image ubuntu** is pulled : if it is found locally than that is used otherwise it is pulled from the registry
3. Using the image the server creates a **new container**
4. A **new file system** is allocated and mounted r+w over the layers of the image
5. A network interface / bridge is created to allow the container to talk with the local host
6. Sets up an IP address and other parms using DHCP (usually a private one : 168.254.x, 172.17.x.x)
7. Executes the process specified (in this case /bin/bash)
8. Captures and provides application input/output

Docker containers commands/states

Pic from http://docker-saigon.github.io/



How to backup/restore docker containers

Exports in a tar the rootfs of the container :

- **$ docker export** *CONTAINER* *>container.tar*
- **$ docker export -o** *container.tar* *CONTAINER*

A single image is saved for the rootfs of the container (unlike **docker save).**

Loads a tarred container rootfs :

- **$ docker import** *FILE|URL|-* *REPOSITORY[:TAG]*

Can load a rootfs from a tar file, from an URL or from stdin, will store it like an image with given name and tag.

EG: **$ docker import** *busybox.tar* *busybox-2:latest*

docker commit

$ docker commit *CONTAINER* *REPOSITORY[:TAG]*
To commit changes made inside the r/w layer of a container into a new image.
It builds a new image from a container.
By default it pauses the container till the image is committed (like a db snapshot).

You can change some metadata like :
$ docker commit -a *“ Author author””* …
$ docker commit -m *“commit message”* …

And some dockerfile entries like *ENV, CMD, ENTRYPOINT, EXPOSE, …*
EG: **$ docker commit -change** *”ENV DEBUG TRUE”* …..

docker copy from/to container

Copy from host to container :

- **$ docker cp** *SOURCE_PATH|-* *CONTAINER:DEST_PATH*

Equivalent to *cp -a* (or *cp -dT –preserve-all* **).**
Copies a single file or recusively a directory to the *DEST_PATH* or gets a tar from stdin (if the first option is -) and untars it in the *DEST_PATH*.

Copy from container to host :

- **$ docker cp** *CONTAINER:SOURCE_PATH* *DEST_PATH|-*

Opposite of above.

EG: **$ tar cf - ./html | docker cp -** *CONTAINER:/var/www/*

Major components :
volumes
(sharing host directories)

Volumes/
Sharing host directories

Union filesystem are usually inefficient.

That’s why I recommend you to use a volume to read/write large files.

This volume can be a directory on your host.

It can be shared in a very simple way when you type the

- **docker run -v**

command (-v for volume)

Sharing the host **~/qe** subdir of your home with the **/shared-qe** dir of the container :

- **$ mkdir ~/qe**
- **$ cd ~/qe**
- **$ touch qe-file**
- **$ docker run -v /home/*USER*/qe:/shared-qe -it busybox**
  - **$ ls -l   /shared-qe**

Major components : linking containers (docker-compose)

linking containers

```
$ docker run -itd –-name cont-a busybox
```

```
$ docker run -itd –-name cont-b –link=cont-a:origin  busybox
```

Will set variable ORIGIN_NAME=/dock-b/dock-a in the dock-b container and will add an entry for it in the /etc/hosts file : dock-a 172.17.0.2

```
$ docker attach cont-b
$ set
$ tail /etc/hosts
```

In this way the destination container can easily reach the origin over the bridged network.

docker-compose

Running multi-container apps manually can be done, but in complicate situations is a pain.
Luckily a tool that does this automatically was devised : **docker-compose**.
docker-compose reads a .yml file and start containers in order and with the proper environment variables.

$ docker-compose *wikipedia.yml*

Major components : registries/repos

Docker Registries

Web Interface to the General public repository
Https://Hub.docker.com

Web Interface to the New Trusted and enterprise ready containers :
https://store.docker.com

General registry used by pull/push :
https://index.docker.io
How to use a private registry ?

Be careful about
v1 and v2 repositories :
index.docker.io/v1/
index.docker.io/v2/_catalog

```
$ docker pull ubuntu
$ docker tag 0345829347592435 mylocalregistry:myport/ubuntu
$ docker push mylocalregistry:myport/ubuntu
$ curl      http://mylocalregistry:myport/v2/_catalog
```

Docker Local Registry

We can run a private Docker Registry via a docker container.

```
$ docker run -d -p 5000:5000 –restart always -name registry registry:2
```

This will run a container from the image *registry* version 2 and will map port 5000 on the container to port 5000 on all host interfaces. It can only be used from localhost because it misses tls certificates and this is outside the scope of this introduction.

Download some images :

```
$ docker pull hello-world
$ docker pull busybox
$ docker pull ubuntu
```

Tag them for the push :

```
$ docker tag hello-world localhost:5000/hello-world
$ docker tag busybox  localhost:5000/busybox
$ docker tag ubuntu localhost:5000/ubuntu
```

Push them on the localhost registry :

```
$ docker push localhost:5000/hello-world
$ docker push localhost:5000/busybox
$ docker push localhost:5000/ubuntu
```

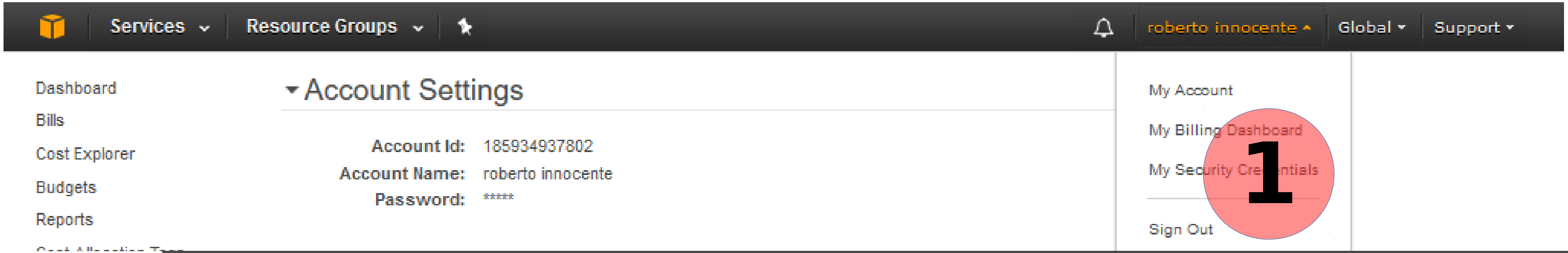
Search local registry :

```
$ curl http://localhost:5000/v2/_catalog      # still under development v2 registry interface
```

Docker on the cloud

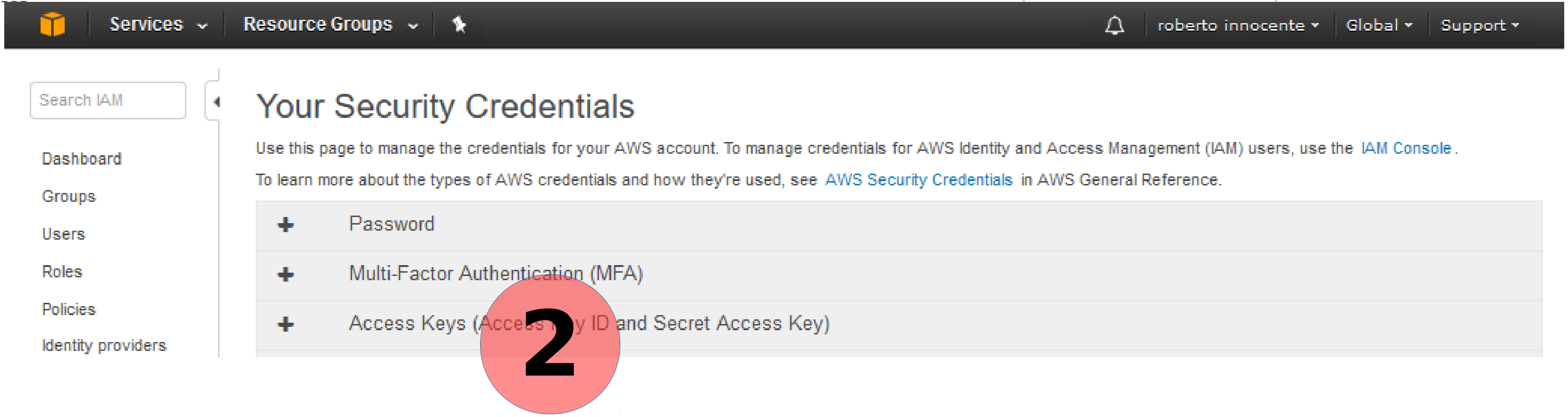
Using the VMs provided by the clouds : Amazon AWS, Microsoft Azure, generic OpenStack

Amazon AWS credentials/1



Account Settings

Account Id: 185934937802
Account Name: roberto innocente
Password: *****



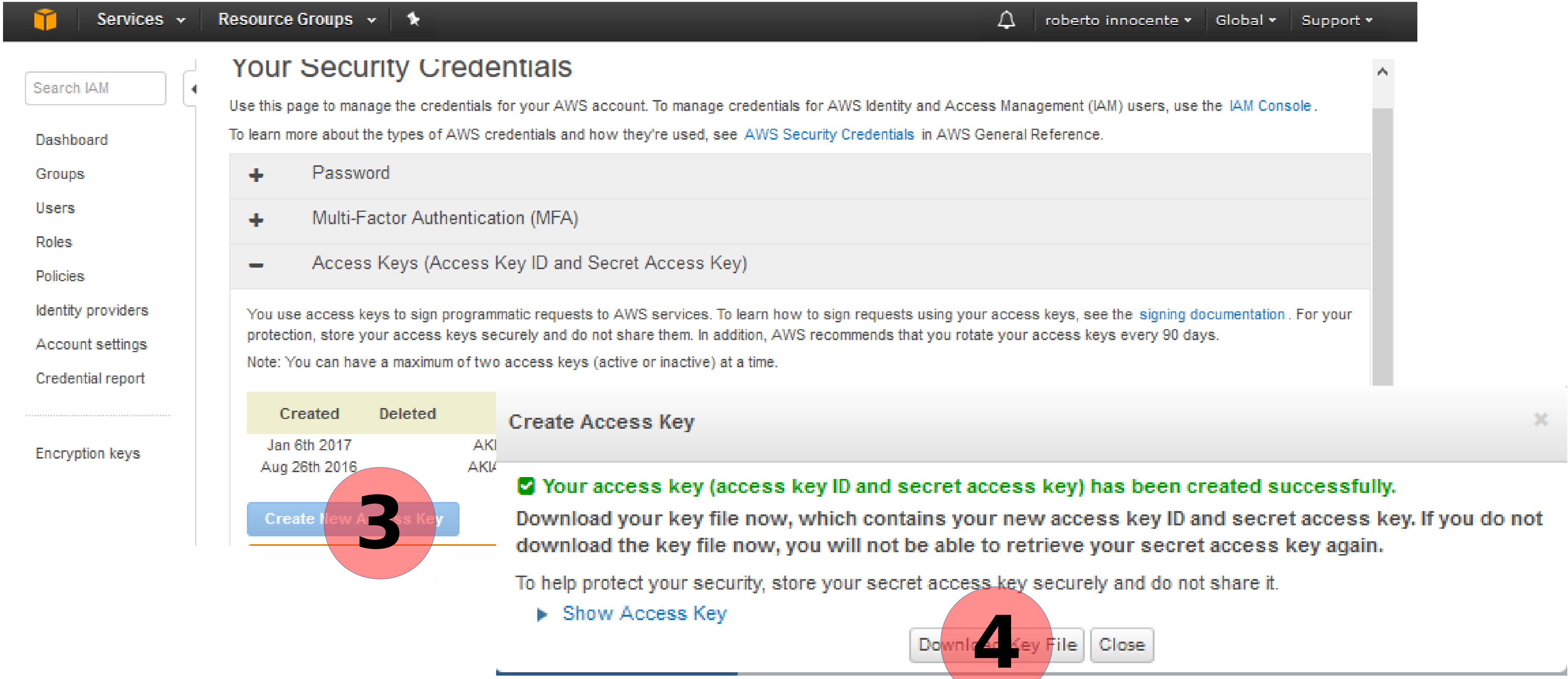
Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access Management (IAM) users, use the IAM Console.

To learn more about the types of AWS credentials and how they're used, see AWS Security Credentials in AWS General Reference.

- Password
- Multi-Factor Authentication (MFA)
- Access Keys (Access Key ID and Secret Access Key)

Amazon AWS credentials/2



Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access Management (IAM) users, use the IAM Console .

To learn more about the types of AWS credentials and how they're used, see AWS Security Credentials in AWS General Reference.

- ✚ Password
- ✚ Multi-Factor Authentication (MFA)
- ━ Access Keys (Access Key ID and Secret Access Key)

You use access keys to sign programmatic requests to AWS services. To learn how to sign requests using your access keys, see the signing documentation . For your protection, store your access keys securely and do not share them. In addition, AWS recommends that you rotate your access keys every 90 days.

Note: You can have a maximum of two access keys (active or inactive) at a time.

Created	Deleted	
Jan 6th 2017		AKI
Aug 26th 2016		AKIA

Create Access Key

☑ Your access key (access key ID and secret access key) has been created successfully.

Download your key file now, which contains your new access key ID and secret access key. If you do not download the key file now, you will not be able to retrieve your secret access key again.

To help protect your security, store your secret access key securely and do not share it.

docker-machine over Amazon AWS

This example uses the AWS credentials (access-key/secret-key) to provide a VM on which it installs docker engine and the ssh keys it generates for the machine. At this point it provides the env variables needed to point the **docker CLI** at the remote host.

The example is run on Windows.

```
PS C:\> docker-machine create      --driver amazonec2
--amazonec2-access-key AKI***
--amazonec2-secret-key w3J***
--amazonec2-region eu-central-1 aws51
```

```
Running pre-create checks...
Creating machine...
(aws51) Launching instance...
...
Waiting for SSH to be available...
...
Provisioning with ubuntu(systemd)...
Installing Docker...
...
PS C:\> & docker-machine env aws51
...
# Run this command to configure your shell:
# & docker-machine env aws51 | Invoke-Expression
PS C:\>  & docker-machine env aws51 | Invoke-Expression
PS C:\> docker-machine ssh aws51
Welcome to Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-43-generic x86_64
...
ubuntu@aws51:~$ logout
PS C:\> docker ps
PS C:\> docker run -it hello-world
```

docker-machine over generic OpenStack cloud

```
$ docker-machine create -d openstack  \
   --openstack-tenant-name ...               \
   --openstack-username ...                  \
   --openstack-password ..                   \
   --openstack-auth-url  .....               \    :keystone service base
URL.
   --openstack-flavor-name  ....             \   :identify the flavor that
will be used for the machine.
   --openstack-image-name  ...               \  :identify the image that
will be used for the machine.
            vm01                                 :machine name
```

docker-machine over a local VM

```
$ docker-machine create -d virtualbox
    --virtualbox-memory=512    vb01
```

Major components : networking

Docker containers networking/1

```
$ docker network ls
```

Available modes are **bridge**, **host**, **none**.
Default network configuration is **bridge** (when you don't specify anything).

```
$ docker run –net=bridge -it     busybox
```

This is the default networking about which I will speak more in next slide.

```
$ docker run –net=host -it         busybox
```

In this case the container simply uses the host network stack.
Container has therefore same IP addr of host. (eg nginx as a reverse proxy for the host web)
ifconfig run in the container will give the host address.
Does'nt work if usernamespaces are enabled.

```
$ docker run -net=container:CONTAINER_ID   busybox
```

Runs container using the network stack of another container.

```
$ docker run –net=none -it         busybox
```

No network is configured. Container can't be reached over the network.
Ifconfig run in the container will show only the *lo* interface.

Docker containers networking/2

When the docker daemon starts it configures a virtual interface **docker0** with a private network address e.g 172.17.0.1. Try on the host :   **$ ifconfig docker0**

Let's start 3 backgrounded containers with *busybox* in *bridge* mode *:*

```
$ docker run –network=bridge -itd    busybox
$ docker run –network=bridge -itd    busybox
$ docker run –network=bridge -itd    busybox
```

The host dhcp server will give them 3 different addresses from the network set up for **docker0** and will configure their gateway as 172.17.0.1.

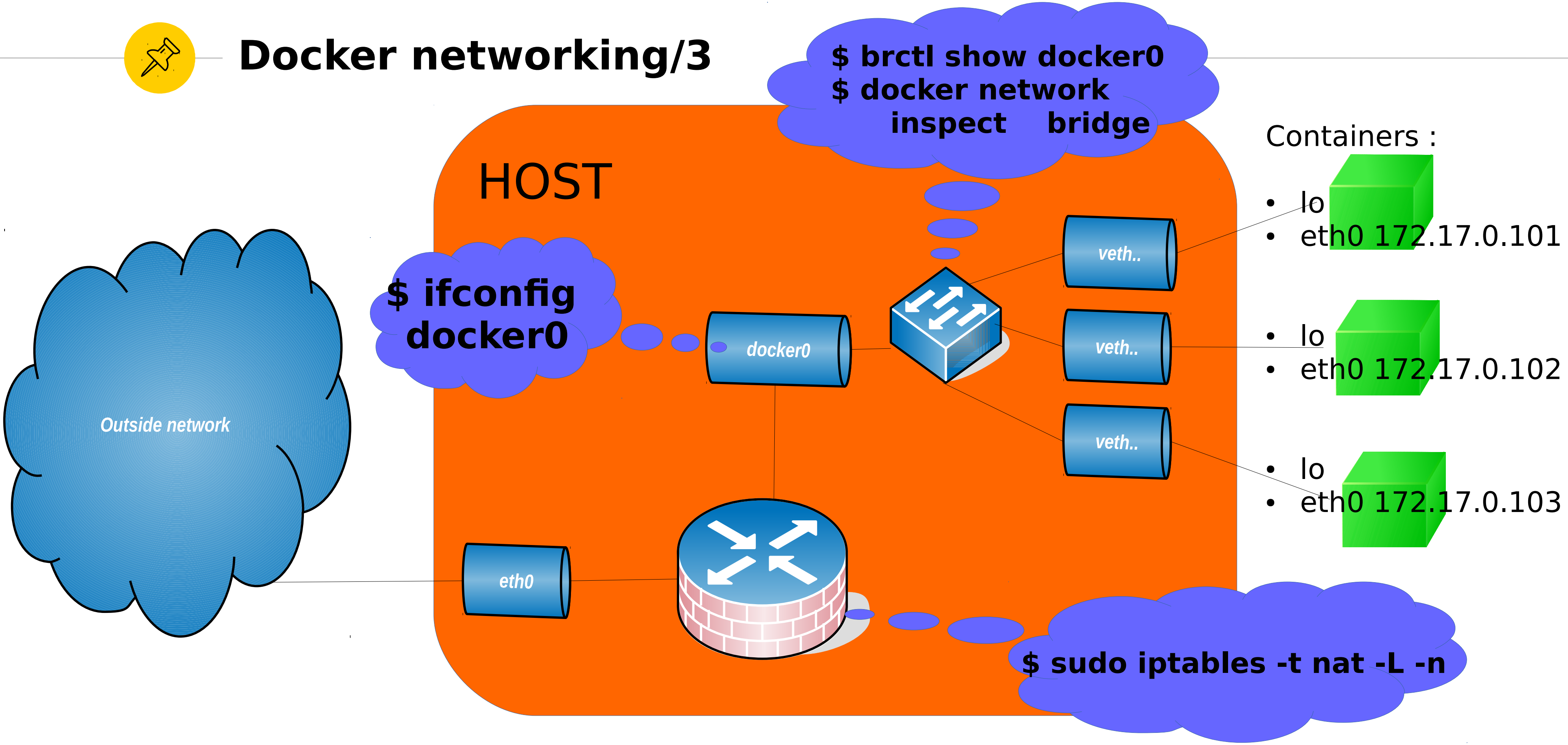
Access them and check it :

```
$ docker attach container
$ ifconfig eth0
$ ip route
```

For every virtual **eth0** in the containers the host will create a virtual **veth..** inside itself (the other end of the pipe). Try inside the host :

```
$ docker network inspect bridge
```

Docker networking/3



```
$ brctl show docker0
$ docker network
    inspect    bridge
```

```
$ ifconfig
  docker0
```

```
$ sudo iptables -t nat -L -n
```

Containers :

- lo
- eth0 172.17.0.101

- lo
- eth0 172.17.0.102

- lo
- eth0 172.17.0.103

Accessing containers from outside

You need to map and open ports from the host to the container.
Containers by default get a private address that is not routable over the Internet.
There are two ways :
- Map all exposed ports of the container to free and unprivileged ports of
 the host :
  - **docker run -P ...**

- Map some free host ports to some of the container ports :
  - **docker run -p 8080:80 -p 4430:443 ...**

If you don’t have any privilege on the host you can’t map privileged ports of the host (<1024 like the ssh=22 or web=80)

Docker clustering

Docker can be clustered in different ways:

- Native **Docker Swarm** : a clustering method that in the last versions of Docker is natively implemented. Very easy to set up for small/medium clusters. Implements load balancing.
- **Kubernetes** : the Google clustering tool, derived from the internal Korg tool. For large clusters, has some complexity.
- **Apache/Mesos**

Mayor components : Dockerfiles

Docker image creation

From another image :

```
• $ docker commit     container-id        image-name
```

From a Dockerfile :

```
• $ mkdir  new-image-dir
• $ cd       new-image-dir
• $ vi       Dockerfile

• $ docker build -t image-name    .
```

Notice
The dot !!

Dockerfile

```
# base image debian
FROM debian
MAINTAINER inno@sissa.it
# apt-get some tools
RUN apt update && apt install  curl
# copy URL, very useful cmd
RUN curl -O http://people.sissa.it/~inno/hello
RUN chmod a+x hello
CMD ./hello
```

Dockerfile for a web server

```
# Dockerfile
FROM ubuntu:16.10
MAINTAINER Roberto Innocente "inno@sissa.it"
RUN apt -yq update
RUN apt -yq install nginx
RUN echo '<h1>Web server in user container</h1>' \
 >/var/www/html/index.html
RUN echo 'Nice to meet you !' \
 >>/var/www/html/index.html
EXPOSE 80
CMD [“/usr/sbin/nginx”, "-g",”daemon off;”]
```

Cloud offerings to run directly Docker containers

Google Cloud Platform Container Engine https://cloud.google.com/container-engine/

“Container Engine Features

Run Docker containers on Google Cloud Platform, powered by Kubernetes.

Docker support

Container Engine supports the common Docker container format.

Private container registry

Google Container Registry makes it easy to store and access your private Docker images.”

Amazon EC2 Container Services https://aws.amazon.com/ecs/

“*Amazon EC2 Container Service (ECS)* is a highly scalable, high performance container management service that supports Docker containers and allows you to easily run applications on a managed cluster of Amazon EC2 instances. Amazon ECS eliminates the need for you to install, operate, and scale your own cluster management infrastructure.”

Docker and Microservices

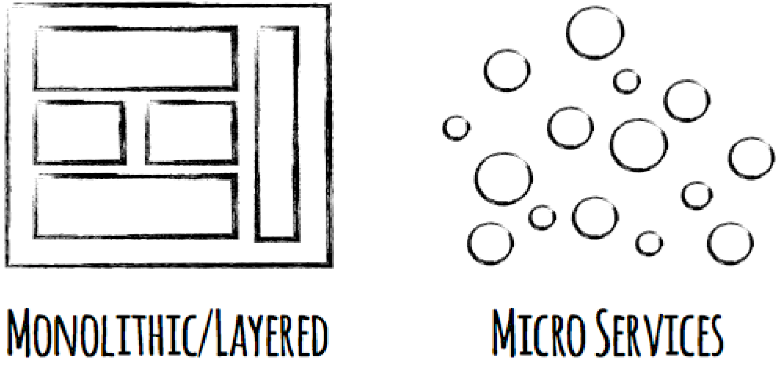
We have already mentioned that a big difference between containers and virtual machines is the short time in which containers start/stop (~ 1/100 of a vm = ~ 100/200 ms).

This enforces their role in the expansion of the **microservice** pattern.

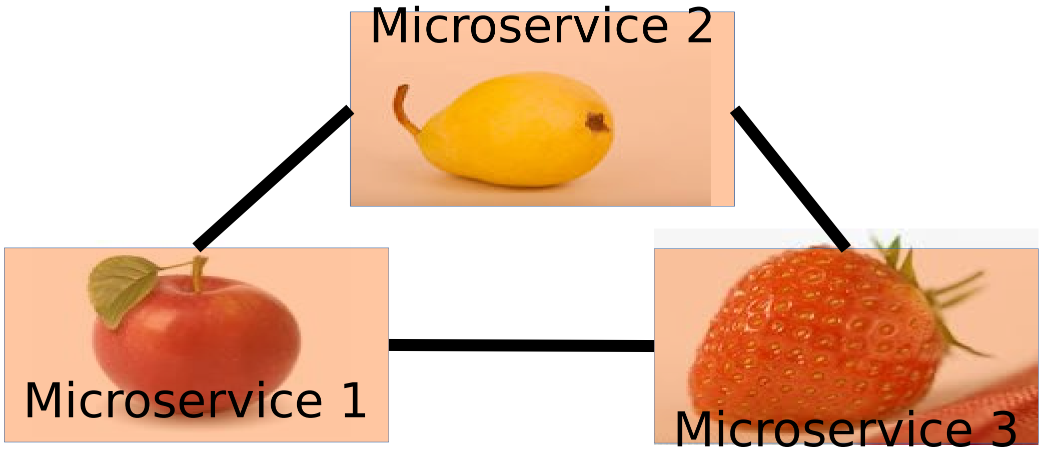
Applications are reduced to many small services performing just one task and communicating between them through a **REST API** (using **http** with **json**) like the **docker** app does.

Microservices

Pic from eugenedvorkin.com



- Opposite to monolithic app. Develop a single application as a set of small independent services (processes) communicating each other only trough a lightweight mechanism (like an http API)
- Microservices are language and tool independent



Info on docker installation for running QE

- More info on docker installation on various platforms for running QE is available at

http://people.sissa.it/%7Einno/pubs/easiest_way_to_run_qe.html

Thanks!

Any questions ?

You can find me at

- *<inno at sissa.it>*