

# Docker Tutorial

Anthony Baire

Université de Rennes 1 / UMR IRISA

March 2, 2020



This tutorial is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 France License](https://creativecommons.org/licenses/by-nc-nd/3.0/fr/)

# Summary

1. Introduction
2. Managing docker containers
3. Inputs/Outputs
4. Managing docker images
5. Building docker images
6. Security considerations
7. The ecosystem & the future

# Part 1.

## Introduction

# What is Docker (1/3)

Rong Tao



*"Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications.*

*Consisting of Docker Engine, a portable, lightweight runtime and packaging tool, and Docker Hub, a cloud service for sharing applications and automating workflows, Docker enables apps to be quickly assembled from components and eliminates the friction between development, QA, and production environments. As a result, IT can ship faster and run the same app, unchanged, on laptops, data center VMs, and any cloud."*

source: <https://www.docker.com/whatisdocker/>

## What is Docker (2/3)

- a container manager
  - lightweight virtualisation  
(*host and guest systems share the same kernel*)
  - based on linux namespaces and cgroups
- massively copy-on-write
  - immutable images
  - instant deployment
  - suitable for micro-services (one process, one container)

→ immutable architecture

## What is Docker (3/3)



- a build system
  - images may be build from sources
  - using a simple DSL (Dockerfile)
- a set of REST APIs
  - Engine API (control the docker engine)
  - Plugin API (extend the engine → network, storage, authorisation)
  - Registry API (publish/download images)
  - Swarm API (manage a clustered of docker machines)

## How Docker helps?

- **normalisation:** same environment (container image) for
  - development
  - jobs on the computing grid
  - continuous integration
  - peer review
  - demonstrations, tutorials
  - technology transfer
- **archival** (*ever tried to reuse old codes*)
  - source → Dockerfile = recipe to rebuild the env from scratch
  - binary → docker image = immutable snapshot of the software with its runtime environment
    - can be rerun it at any time later

## In practice

A docker image is an immutable snapshot of the filesystem

A docker **container** is

- **a temporary file system**
  - layered over an immutable fs (docker image)
  - fully writable (copy-on-write<sup>1</sup>)
  - dropped at container's end of life (unless a **commit** is made)
- **a network stack**
  - with its own private address (*by default in 172.17.x.x*)
- **a process group**
  - one main process launched inside the container
  - all sub-process **SIGKILLed** when the main process exits

---

<sup>1</sup>several possible methods: overlaysfs (default), btrfs, lvm, zfs, aufs



## Installation

<https://docs.docker.com/engine/installation/>

Native installation:

- requires linux kernel >3.8

## Docker Machine:

- a command for provisioning and managing docker nodes deployed:
  - in a local VM (virtualbox)
  - remotely (many cloud API supported)

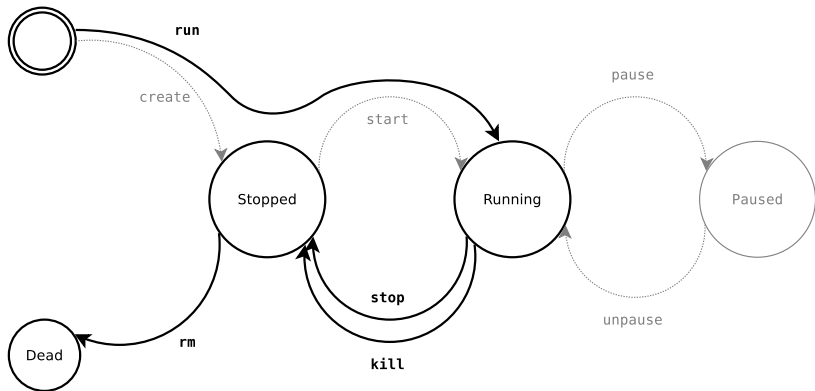
## Part 2.

# Managing containers

- create/start/stop/remove containers
- inspect containers
- interact, commit new images

# Lifecycle of a docker container

Rong Tao



## Container management commands

command	description
<code>docker create image [ command ]</code> <code>docker run image [ command ]</code>	create the container = <b>create</b> + <b>start</b>
<code>docker rename container new_name</code> <code>docker update container</code>	rename the container update the container config
<code>docker start container...</code> <code>docker stop container...</code> <code>docker kill container...</code> <code>docker restart container...</code>	start the container graceful <sup>2</sup> stop kill (SIGKILL) the container = <b>stop</b> + <b>start</b>
<code>docker pause container...</code> <code>docker unpause container...</code>	suspend the container resume the container
<code>docker rm [ -f<sup>3</sup> ] container...</code>	destroy the container

<sup>2</sup>send SIGTERM to the main process + SIGKILL 10 seconds later

<sup>3</sup>-f allows removing running containers (= **docker kill** + **docker rm**)

## Notes about the container lifecycle

- the container filesystem is created in `docker create` and dropped in `docker rm`
  - it is persistent across `stop/start`
- the container configuration is mostly static
  - config is set in `create/run`
  - `docker update` may change only a few parameters (eg: cpu/ram/blkio allocations)
  - changing other parameters require destroying and re-creating the container
- other commands are rather basic

```
Usage: docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

[Create a new container](#)

<code>--attach=[]</code>	Attach to STDIN, STDOUT or STDERR
<code>--add-host=[]</code>	Add a custom host-to-IP mapping (host:ip)
<code>--blkio-weight=0</code>	Block IO (relative weight), between 10 and 1000
<code>--cpu-shares=0</code>	CPU shares (relative weight)
<code>--cap-add=[]</code>	Add Linux capabilities
<code>--cap-drop=[]</code>	Drop Linux capabilities
<code>--cgroup-parent=</code>	Optional parent cgroup for the container
<code>--cidfile=</code>	Write the container ID to the file
<code>--cpu-period=0</code>	Limit CPU CFS (Completely Fair Scheduler) period
<code>--cpu-quota=0</code>	Limit CPU CFS (Completely Fair Scheduler) quota
<code>--cpuset-cpus=</code>	CPUs in which to allow execution (0-3, 0,1)
<code>--cpuset-mems=</code>	MEMs in which to allow execution (0-3, 0,1)
<code>--device=[]</code>	Add a host device to the container
<code>--disable-content-trust=true</code>	Skip image verification
<code>--dns=[]</code>	Set custom DNS servers
<code>--dns-opt=[]</code>	Set DNS options
<code>--dns-search=[]</code>	Set custom DNS search domains
<code>--e, --env=[]</code>	Set environment variables
<code>--entrypoint=</code>	Override the default ENTRYPOINT of the image
<code>--env-file=[]</code>	Read in a file of environment variables
<code>--expose=[]</code>	Expose a port or a range of ports
<code>--group-add=[]</code>	Add additional groups to join
<code>--h, --hostname=</code>	Container host name
<code>--help=false</code>	Print usage
<code>-i, --interactive=false</code>	Keep STDIN open even if not attached
<code>--ipc=</code>	IPC namespace to use
<code>--kernel-memory=</code>	Kernel memory limit
<code>-l, --label=[]</code>	Set meta data on a container
<code>--label-file=[]</code>	Read in a line delimited file of labels
<code>--link=[]</code>	Add link to another container
<code>--log-driver=</code>	Logging driver for container
<code>--log-opt=[]</code>	Log driver options
<code>--ipc-conf=[]</code>	Add custom ipc options
<code>--m, --memory=</code>	Memory limit
<code>--mac-address=</code>	Container MAC address (e.g. 92:dc:c6:0a:29:33)
<code>--memory-reservation=</code>	Memory soft limit
<code>--memory-swap=</code>	Total memory (memory + swap), '-1' to disable swap
<code>--memory-swappiness=i</code>	Tuning container memory swappiness (0 to 100)
<code>--name=</code>	Assign a name to the container
<code>--net=default</code>	Set the Network for the container
<code>--oom-kill-disable=false</code>	Disable OOM Killer
<code>--p, --publish-all=false</code>	Publish all exposed ports to random ports
<code>--p, --publish=[]</code>	Publish a container's port(s) to the host
<code>--pid=</code>	PID namespace to use
<code>--privileged=false</code>	Give extended privileges to this container
<code>--read-only=false</code>	Mount the container's root filesystem as read only
<code>--restart=no</code>	Restart policy to apply when a container exits
<code>--security-opt=[]</code>	Security Options
<code>--stop-signal=SIGTERM</code>	Signal to stop a container, SIGTERM by default
<code>-t, --tty=false</code>	Allocate a pseudo-TTY
<code>--u, --user=</code>	Username or UID (format: <name> uid[:<group gid>])
<code>--ulimit=[]</code>	Ulimit options
<code>--uts=</code>	UTS namespace to use
<code>--v, --volume=[]</code>	Bind mount a volume
<code>--volume-driver=</code>	Optional volumes driver for the container
<code>--volumes-from=[]</code>	Mount volumes from the specified container(s)
<code>--w, --workdir=</code>	Working directory inside the container

```
Usage: docker start [OPTIONS] CONTAINER [CONTAINER...]
```

Start one or more stopped containers

```
-a, --attach=false      Attach STDOUT/STDERR and forward signals
--help=false           Print usage
-i, --interactive=false Attach container's STDIN
```

```
Usage:  docker stop [OPTIONS] CONTAINER [CONTAINER...]
```

Stop a running container.

Sending SIGTERM and then SICKILL after a grace period

```
--help=false      Print usage
-t, --time=10    Seconds to wait for stop before killing it
```

Usage: `docker restart [OPTIONS] CONTAINER [CONTAINER...]`

## Restart a container

```
--help=false      Print usage
-t, --time=10    Seconds to wait for stop before killing the container
```

Usage: `docker kill [OPTIONS] CONTAINER [CONTAINER...]`

## Kill a running container

```
--help=false      Print usage
-g, --signal=KILL Signal to send to the container
```

Usage: `docker rm [OPTIONS] CONTAINER [CONTAINER...]`

Remove one or more containers

```
--f, --force=false      Force the removal of a running container (uses SIGKILL)
--help=false           Print usage
--l, --link=false       Remove the specified link
--v, --volumes=false    Remove the volumes associated with the container
```

Usage: **docker pause** [OPTIONS] CONTAINER [CONTAINER...]

## Pause all processes within a container

```
--helpfalse    Print usage
```

# docker run — Run a container

<https://docs.docker.com/reference/run/>

```
docker run [ options ] image [ arg0 arg1...]
```

→ create a container and start it

- the container filesystem is initialised from image *image*
- *arg0..argN* is the command run inside the container (as PID 1)

```
$ docker run debian /bin/hostname
f0d0720bd373
$ docker run debian date +%H:%M:%S
17:10:13
$ docker run debian true ; echo $?
0
$ docker run debian false ; echo $?
1
```

## `docker run` — Foreground mode vs. Detached mode

- Foreground mode is the default
  - *stdout* and *stderr* are redirected to the terminal
  - `docker run` propagates the exit code of the main process
- With `-d`, the container is run in detached mode:
  - displays the ID of the container
  - returns immediately

```
$ docker run debian date
Tue Jan 20 17:32:07 UTC 2015
$ docker run -d debian date
4cbdefb3d3e1331ccf7783b32b47774fefca426e03a2005d69549f3ff06b9306
$ docker logs 4cbdef
Tue Jan 20 17:32:16 UTC 2015
```



## docker run — TTY allocation

Use `-t` to allocate a pseudo-terminal for the container

→ without a tty

```
$ docker run debian ls
bin
boot
dev
...
$ docker run debian bash
$
```

→ with a tty (`-t`)

```
$ docker run -t debian ls
bin  dev  home  lib64  mnt  proc  run  selinux  sys  usr
boot  etc  lib   media  opt  root  sbin  srv      tmp  var
$ docker run -t debian bash
root@10d90c09d9ac:/#
```

## `docker run` — interactive mode

- By default containers are non-interactive
  - *stdin* is closed immediately
  - terminal signals are not forwarded<sup>4</sup>

```
$ docker run -t debian bash
root@6fecc2e8ab22:/# date
^C
$
```

- With `-i` the container runs interactively
  - *stdin* is usable
  - terminal signals are forwarded to the container

```
$ docker run -t -i debian $ bash
root@78ff08f46cdb:/# date
Tue Jan 20 17:52:01 UTC 2015
root@78ff08f46cdb:/# ^C
root@78ff08f46cdb:/#
```

---

<sup>4</sup>^C only detaches the terminal, the container keeps running in background

## docker run — override defaults (1/2)

### user (-u)

```
$ docker run debian whoami
root
$ docker run -u nobody debian whoami
nobody
```

### working directory (-w)

```
$ docker run debian pwd
/
$ docker run -w /opt debian pwd
/opt
```

## docker run — override defaults (2/2)

### environment variables (-e)

```
$ docker run debian sh -c 'echo $FOO $BAR'
```

```
$ docker run -e FOO=foo -e BAR=bar debian sh -c 'echo $FOO $BAR'
foo bar
```

### hostname (-h)

```
$ docker run debian hostname
```

```
830e47237187
```

```
$ docker run -h my-nice-container debian hostname
```

```
my-nice-hostname
```

## docker run — set the container name

`--name` assigns a name for the container  
(*by default a random name is generated*)

```
$ docker run -d -t debian
da005df0d3aca345323e373e1239216434c05d01699b048c5ff277dd691ad535
$ docker run -d -t --name blahblah debian
0bd3cb464ff68eaf9fc43f0241911eb207fe9c1341a0850e8804b7445ccd21
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          ..  NAMES
0bd3cb464ff6   debian:7.5     "/bin/bash"             6 seconds ago    ..  blahblah
da005df0d3ac   debian:7.5     "/bin/bash"             About a minute ago  drunk_darwin
$ docker stop blahblah drunk_darwin
```

**Note:** Names must be unique

```
$ docker run --name blahblah debian true
2015/01/20 19:31:21 Error response from daemon: Conflict, The name blahblah is already assigned
to 0bd3cb464ff6. You have to delete (or rename) that container to be able to assign blahblah to a
container again.
```

## docker run — autoremove

By default the container still exists after command exit

```
$ docker run --name date-ctr debian date
Tue Jan 20 18:38:21 UTC 2015
$ docker start date-ctr
date-ctr
$ docker logs date-ctr
Tue Jan 20 18:38:21 UTC 2015
Tue Jan 20 18:38:29 UTC 2015
$ docker rm date-ctr
date-ctr
$ docker start date-ctr
Error response from daemon: No such container: date-ctr
2015/01/20 19:39:27 Error: failed to start one or more containers
```

With `--rm` the container is automatically removed after exit

```
$ docker run --rm --name date-ctr debian date
Tue Jan 20 18:41:49 UTC 2015
$ docker rm date-ctr
Error response from daemon: No such container: date-ctr
2015/01/20 19:41:53 Error: failed to remove one or more containers
```

## Common **rm** idioms

Launch an throwaway container for debugging/testing purpose

```
$ docker run --rm -t -i debian
root@4b71c9a39326:/#
```

Remove all zombie containers

```
$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
2b291251a415	debian:7.5	"hostname"	About a minute ago	Exited (0) About a mi
6d36a2f07e18	debian:7.5	"false"	2 minutes ago	Exited (1) 2 minutes
0f563f110328	debian:7.5	"true"	2 minutes ago	Exited (0) 2 minutes
4b57d0327a20	debian:7.5	"uname -a"	5 minutes ago	Exited (0) 5 minutes

```
$ docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
2b291251a415
6d36a2f07e18
0f563f110328
4b57d0327a20
```

## Inspecting the container

command	description
<code>docker ps</code>	list running containers
<code>docker ps -a</code>	list all containers
<code>docker logs [ -f<sup>5</sup> ] container</code>	show the container output ( <i>stdout+stderr</i> )
<code>docker top container [ ps options ]</code>	list the processes running inside the containers <sup>6</sup>
<code>docker stats [ container ]</code>	display live usage statistics <sup>7</sup>
<code>docker diff container</code>	show the differences with the image (modified files)
<code>docker port container</code>	list port mappings
<code>docker inspect container...</code>	show low-level infos (in json format)

<sup>5</sup>with `-f`, `docker logs` follows the output (à la `tail -f`)

<sup>6</sup>`docker top` is the equivalent of the `ps` command in unix

<sup>7</sup>`docker stats` is the equivalent of the `top` command in unix



## Interacting with the container

command	description
<code>docker attach container</code>	attach to a running container (stdin/stdout/stderr)
<code>docker cp container:path hostpath </code> <code>docker cp hostpath - container:path</code>	copy files from the container copy files into the container
<code>docker export container</code>	export the content of the container (tar archive)
<code>docker exec container args. . .</code>	run a command in an existing container ( <b>useful</b> for debugging)
<code>docker wait container</code>	wait until the container terminates and return the exit code
<code>docker commit container image</code>	commit a new docker image (snapshot of the container)

## docker commit example

```
$ docker run --name my-container -t -i debian
root@3b397d383faf:/# cat >> /etc/bash.bashrc <<EOF
> echo 'hello!'
> EOF
root@3b397d383faf:/# exit
$ docker start --attach my-container
my-container
hello!
root@3b397d383faf:/# exit
$ docker diff my-container
C /etc
C /etc/bash.bashrc
A /.bash_history
C /tmp
$ docker commit my-container hello
a57e91bc3b0f5f72641f19cab85a7f3f860a1e5e9629439007c39fd76f37c5dd
$ docker rm my-container
my-container
$ docker run --rm -t -i hello
hello!
root@386ed3934b44:/# exit
$ docker images -t
511136ea3c5a Virtual Size: 0 B
af6bdc397692 Virtual Size: 115 MB
667250f9a437 Virtual Size: 115 MB Tags: debian:wheezy, debian:latest
a57e91bc3b0f Virtual Size: 115 MB Tags: hello:latest
```

# Part 3.

## Inputs/Outputs

- Data volumes (persistent data)
  - mounted from the host filesystem
  - named volumes (internal + volume plugins)
- Devices
- Links
- Publishing ports (NAT)



## mount examples (1/2)

### Persistent data

```
$ docker run --rm -t -i -v /tmp/persistent:/persistent debian
root@0aaedfeb7bf9:/# echo "blahblah" >/persistent/foo
root@0aaedfeb7bf9:/# exit
$ cat /tmp/persistent/foo
blahblah
$ docker run --rm -t -i -v /tmp/persistent:/persistent debian
root@6c8ed008c041:/# cat /persistent/foo
blahblah
```

### Inputs (read-only volume)

```
$ mkdir /tmp/inputs
$ echo hello > /tmp/inputs/bar
$ docker run --rm -t -i -v /tmp/inputs:/inputs:ro debian
root@05168a0eb322:/# cat /inputs/bar
hello
root@05168a0eb322:/# touch /inputs/foo
touch: cannot touch `/inputs/foo': Read-only file system
```

## mount examples (2/2)

### Named pipe

```
$ mkfifo /tmp/fifo
$ docker run -d -v /tmp/fifo:/fifo debian sh -c 'echo blah blah> /fifo'
ff0e44c25e10d516ce947eae9168060ee25c2a906f62d63d9c26a154b6415939
$ cat /tmp/fifo
blah blah
```

### Unix socket

```
$ docker run --rm -t -i -v /dev/log:/dev/log debian
root@56ec518d3d4e:/# logger blah blah blah
root@56ec518d3d4e:/# exit
$ sudo tail /var/log/messages | grep logger
Jan 21 08:07:59 halfoat logger: blah blah blah
```

## docker run — named volumes

### Named volumes

- stored inside `/var/lib/docker`
- lifecycle managed with the `docker volume` command
- plugin API to provide shared storage over a cluster/cloud<sup>8</sup>

```
$ docker volume create my-volume
my-volume
$ docker volume ls
DRIVER          VOLUME NAME
local           my-volume
$ docker run --rm -t -i -v my-volume:/vol busybox
/ # echo foo > /vol/bar
/ # ^D
$ docker volume inspect my-volume|grep Mountpoint
      "Mountpoint": "/var/lib/docker/volumes/my-volume/_data",
$ docker run --rm -t -i -v my-volume:/vol busybox cat /vol/bar
foo
$ docker volume rm my-volume
my-volume
```

<sup>8</sup><https://docs.docker.com/engine/tutorials/dockervolumes/>

## initialisation: bind volumes vs named volumes

- bind volumes are created empty
- named volumes are created with a copy of the image content at the same mount point

```
$ docker run --rm -t alpine ls /etc/apk
arch                keys                protected_paths.d  repositories        world

$ docker run --rm -t -v /tmp/dummy:/etc/apk alpine ls /etc/apk
$ ls /tmp/dummy/
$

$ docker run --rm -t -v dummy:/etc/apk alpine ls /etc/apk
arch                keys                protected_paths.d  repositories        world
$ ls /var/lib/docker/volumes/dummy/_data
arch                keys                protected_paths.d  repositories        world
```



## docker run — grant access to a device

By default devices are not usable inside the container

```
$ docker run --rm debian fdisk -l /dev/sda
root@dcba37b0c0bd:/# fdisk -l /dev/sda
fdisk: cannot open /dev/sda: No such file or directory

$ docker run --rm debian sh -c 'mknod /dev/sda b 8 0 && fdisk -l /dev/sda'
fdisk: cannot open /dev/sda: Operation not permitted

$ docker run --rm -v /dev/sda:/dev/sda debian fdisk -l /dev/sda
fdisk: cannot open /dev/sda: Operation not permitted
```

They can be **whitelisted** with `--device`

```
docker run --device /hostpath[:/containerpath ] ...
```

```
$ docker run --rm --device /dev/sda debian fdisk -l /dev/sda
Disk /dev/sda: 250.1 GB, 250059350016 bytes
...
```

## docker run — inter-container links (legacy links<sup>9</sup>)

Containers cannot be assigned a static IP address (by design)

→ service discovery is a must

Docker “links” are the most basic way to discover a service

```
docker run --link ctr:alias ...
```

→ container *ctr* will be known as *alias* inside the new container

```
$ docker run --name my-server debian sh -c 'hostname -i && sleep 500' &  
172.17.0.4
```

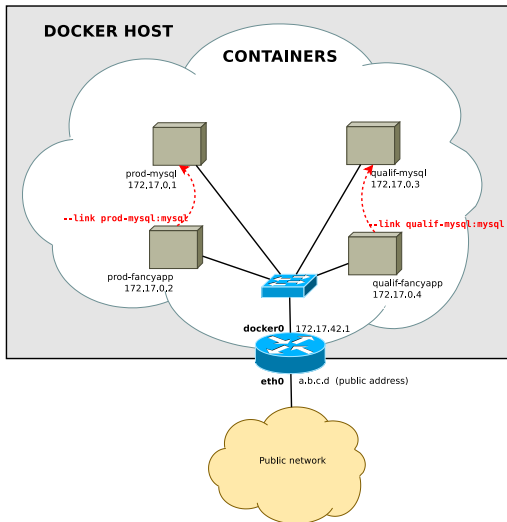
```
$ docker run --rm -t -i --link my-server:srv debian  
root@d752180421cc:/# ping srv  
PING srv (172.17.0.4): 56 data bytes  
64 bytes from 172.17.0.4: icmp_seq=0 ttl=64 time=0.195 ms
```

---

<sup>9</sup>since v1.9.0, links are superseded by user-defined networks

# Legacy links

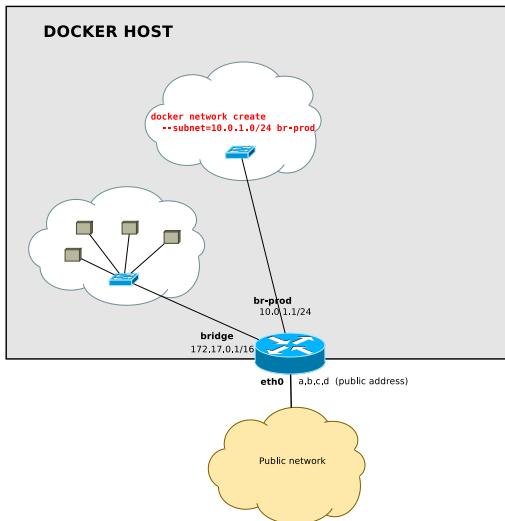
⚠ deprecated feature



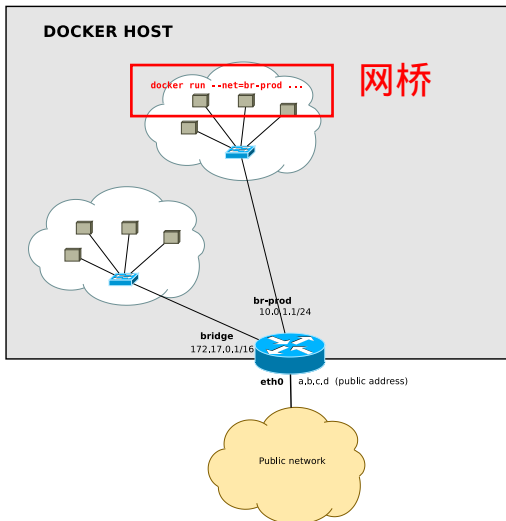
## User-defined networks (since v1.9.0)

- by default new containers are connected to the main network (named "bridge", 172.17.0.0/16)
- the user can create additional networks:  
`docker network create NETWORK`
- newly created containers are connected to one network:  
`docker run --net=NETWORK`
- container may be dynamically attached/detached to any network:  
`docker network connect NETWORK CONTAINER`  
`docker network disconnect NETWORK CONTAINER`
- networks are isolated from each other, communications is possible by attaching a container to multiple networks

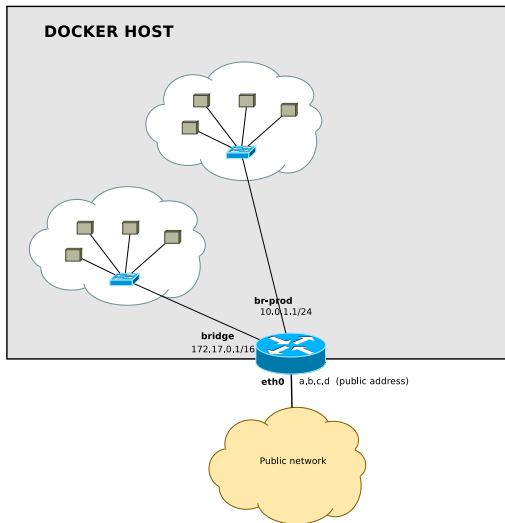
# User-defined networks example



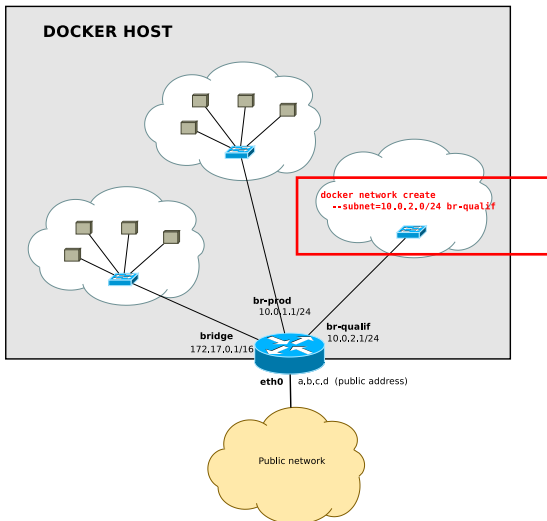
## User-defined networks example



## User-defined networks example

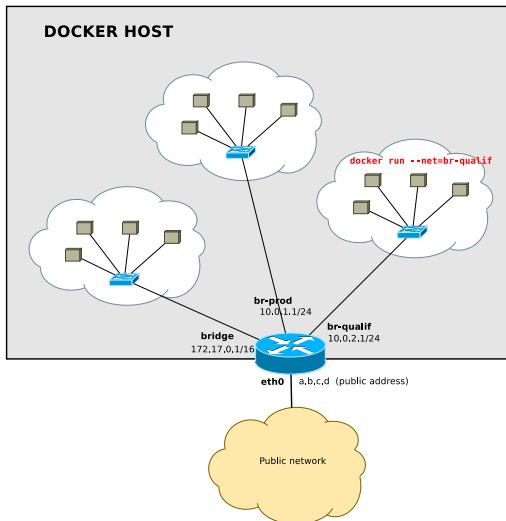


## User-defined networks example

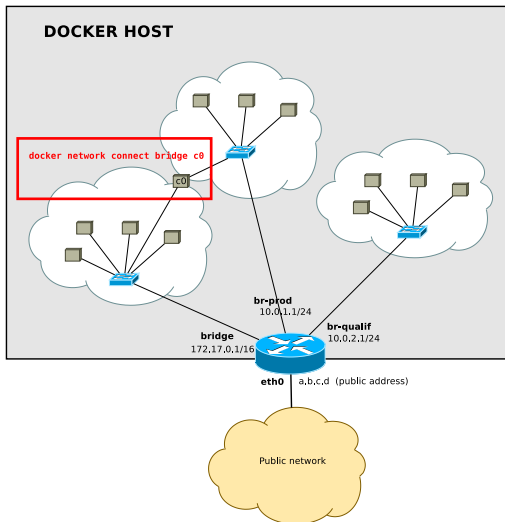




## User-defined networks example



## User-defined networks example



## `docker run` — publish a TCP port

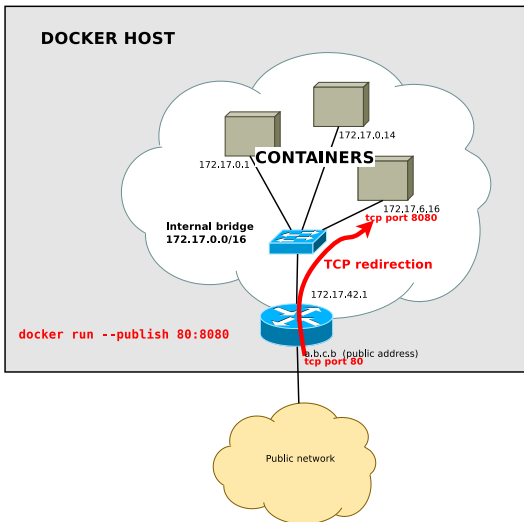
Containers are deployed in a private network, they are not reachable from the outside (unless a redirection is set up)

```
docker run -p [ipaddr:]hostport:containerport
```

→ redirect incoming connections to the TCP port *hostport* of the host to the TCP port *containerport* of the container

The listening socket binds to 0.0.0.0 (all interfaces) by default or to *ipaddr* if given

## publish example



## publish example

### bind to all host addresses

```
$ docker run -d -p 80:80 nginx
52c9105e1520980d49ed00ecf5f0ca694d177d77ac9d003b9c0b840db9a70d62

$ wget -nv http://localhost/
2016-01-12 18:32:52 URL:http://localhost/ [612/612] -> "index.html" [1]

$ wget -nv http://172.17.42.1/
2016-01-12 18:33:14 URL:http://172.17.42.1/ [612/612] -> "index.html" [1]
```

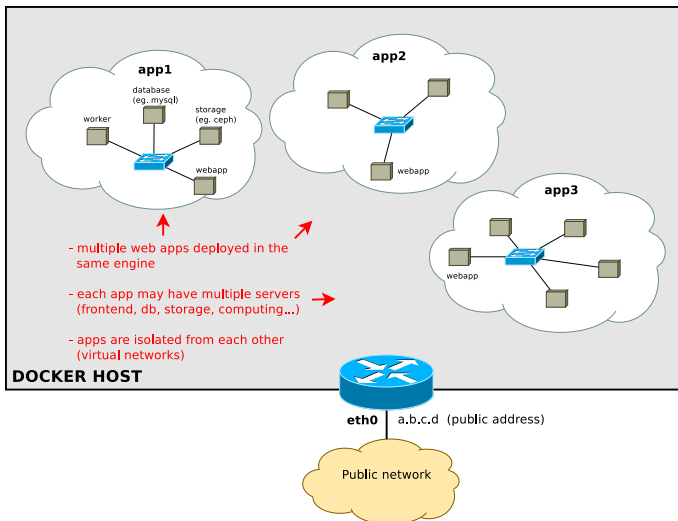
### bind to 127.0.0.1

```
$ docker run -d -p 127.0.0.1:80:80 nginx
4541b43313b51d50c4dc2722e741df6364c5ff50ab81b828456ca55c829e732c

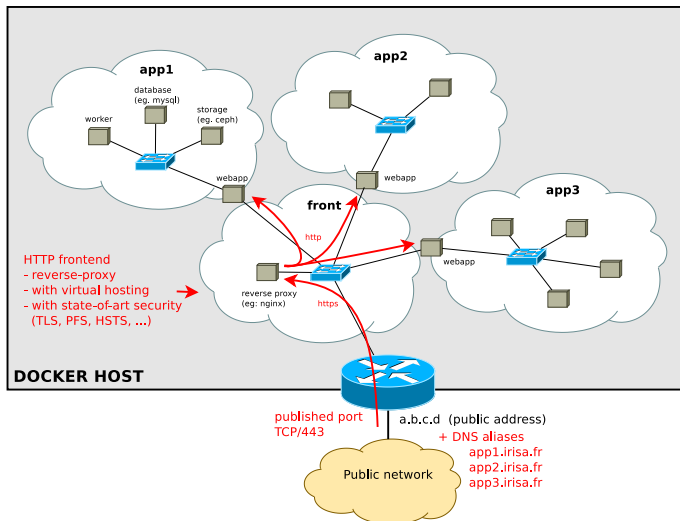
$ wget -nv http://localhost/
2016-01-12 18:37:10 URL:http://localhost/ [612/612] -> "index.html.1" [1]

$ wget http://172.17.42.1/
--2016-01-12 18:38:32-- http://172.17.42.1/
Connecting to 172.17.42.1:80... failed: Connection refused.
```

# The whole picture



# The whole picture



## Part 4.

# Managing docker images



## Docker images

A docker image is a snapshot of the filesystem + some metadata

- immutable
- copy-on-write storage
  - for instantiating containers
  - for creating new versions of the image (multiple layers)
- identified by a unique hex IDs
  - Image ID: randomly generated
  - Digest: hashed from the content
- may be tagged<sup>10</sup> with a human-friendly name  
eg: `debian:wheezy` `debian:jessie` `debian:latest`

---

<sup>10</sup>possibly multiple times

## Image management commands

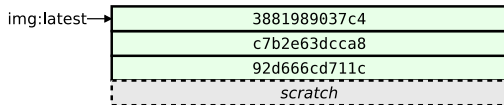
command	description
<code>docker images</code> <code>docker history image</code> <code>docker inspect image...</code>	list all local images show the image history (list of ancestors) show low-level infos (in json format)
<code>docker tag image tag</code>	tag an image
<code>docker commit container image</code> <code>docker import url - [tag]</code>	create an image (from a container) create an image (from a tarball)
<code>docker rmi image...</code>	delete images

## Example: images & containers

*scratch*

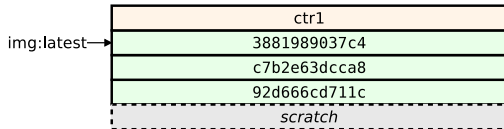
## Example: images & containers

```
docker pull img
```



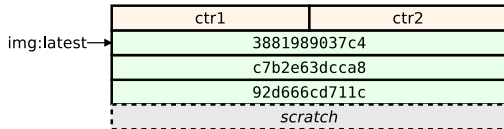
## Example: images & containers

```
docker run --name ctr1 img
```



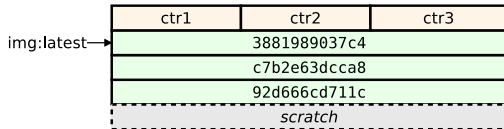
## Example: images & containers

```
docker run --name ctr2 img
```



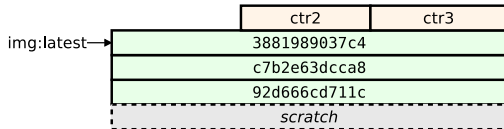
## Example: images & containers

```
docker run --name ctr3 img
```



## Example: images & containers

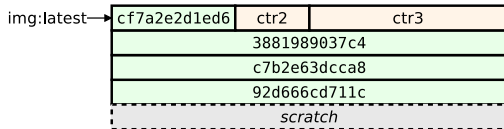
```
docker rm ctr1
```





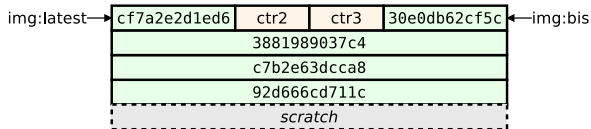
## Example: images & containers

```
docker commit ctr2 img
```



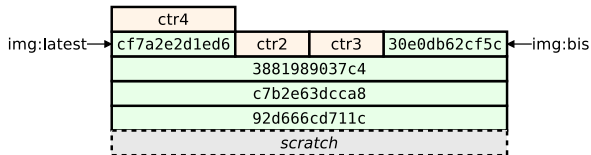
## Example: images & containers

```
docker commit ctr3 img:bis
```



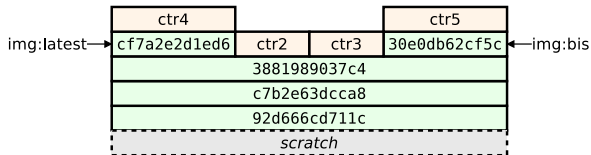
## Example: images & containers

```
docker run --name ctr4 img
```



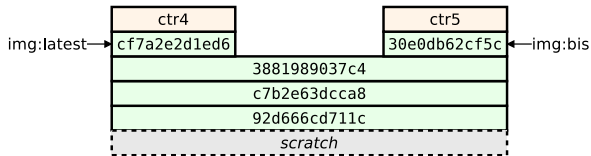
## Example: images & containers

```
docker run --name ctr5 img:bis
```



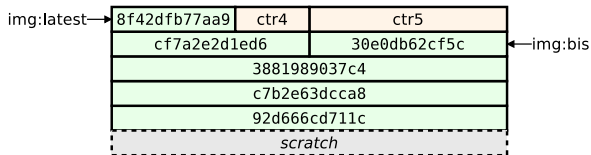
## Example: images & containers

```
docker rm ctr2 ctr3
```



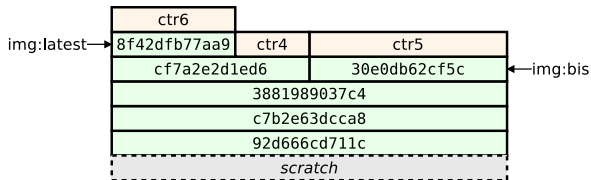
## Example: images & containers

```
docker commit ctr4 img
```



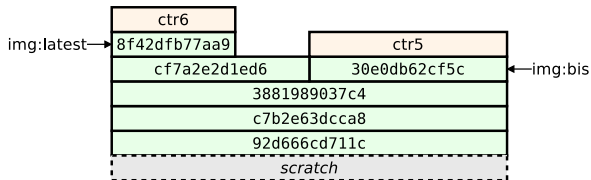
## Example: images & containers

```
docker run --name ctr6 img
```



## Example: images & containers

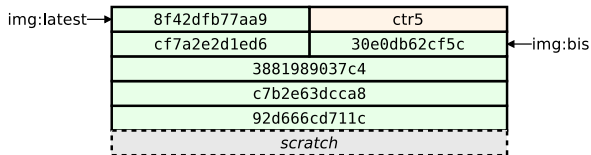
```
docker rm ctr4
```





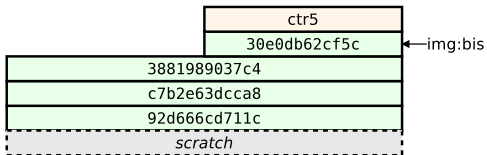
## Example: images & containers

```
docker rm ctr6
```



## Example: images & containers

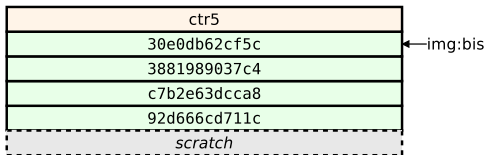
```
docker rmi img
```



## Example: images & containers

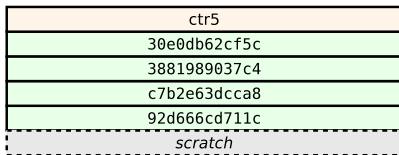
```
docker rmi img:bis
```

Error: image img:bis is reference by ctr5



## Example: images & containers

```
docker rmi -f img:bis
```



## Example: images & containers

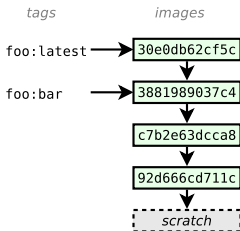
```
docker rm ctr5
```

30e0db62cf5c
3881989037c4
c7b2e63dcca8
92d666cd711c
<i>scratch</i>

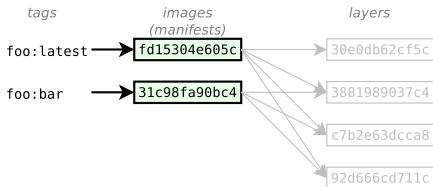


# Images vs. Layers

**docker < v1.10**  
no distinction between images & layers



**docker >= v1.10**  
layers are hidden to the user  
(implementation detail)



## Image tags

A docker tag is made of two parts: “*REPOSITORY:TAG*”

The *TAG* part identifies the version of the image. If not provided, the default is “:latest”

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
debian	8	835c4d274060	2 weeks ago	122.6 MB
debian	8.0	835c4d274060	2 weeks ago	122.6 MB
debian	jessie	835c4d274060	2 weeks ago	122.6 MB
debian	rc-buggy	350a74df81b1	7 months ago	159.9 MB
debian	experimental	36d6c9c7df4c	7 months ago	159.9 MB
debian	6.0.9	3b36e4176538	7 months ago	112.4 MB
debian	squeeze	3b36e4176538	7 months ago	112.4 MB
debian	wheezy	667250f9a437	7 months ago	115 MB
debian	latest	667250f9a437	7 months ago	115 MB
debian	7.5	667250f9a437	7 months ago	115 MB
debian	unstable	24a4621560e4	7 months ago	123.6 MB
debian	testing	7f5d8ca9fdcf	7 months ago	121.8 MB
debian	stable	caa04aa09d69	7 months ago	115 MB
debian	sid	f3d4759f77a7	7 months ago	123.6 MB
debian	7.4	e565fbbc6033	9 months ago	115 MB
debian	7.3	b5fe16f2ccba	11 months ago	117.8 MB



## Tagging conventions (1/2)

Local tags may have arbitrary names, however the `docker push` and `docker pull` commands expect some conventions

The *REPOSITORY* identifies the origin of the image, it may be:

- a name (eg: `debian`)
  - refers to a repository on the official registry
  - `https://store.docker.com/`
- a hostname+name (eg: `some.server.com/repo`)
  - refers to an arbitrary server supporting the registry API
  - `https://docs.docker.com/reference/api/registry_api/`

## Tagging conventions (2/2)

Use slashes to delimit namespaces (for subprojects):

image name	description
debian	(semi-)official debian images
fedora	official fedora images
fedora/apache	apache images provided by the fedora project
fedora/couchdb	couchdb images provided by the fedora project

## Image transfer commands

### Using the registry API

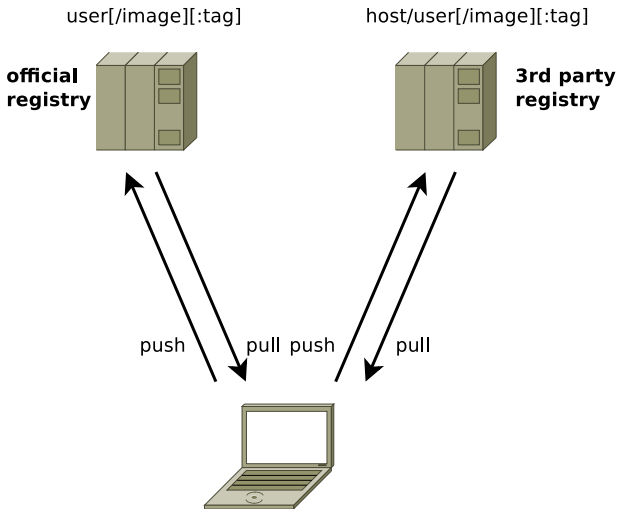
<code>docker pull repo[:tag]...</code>	pull an image/repo from a registry
<code>docker push repo[:tag]...</code>	push an image/repo from a registry
<code>docker search text</code>	search an image on the official registry
<code>docker login ...</code>	login to a registry
<code>docker logout ...</code>	logout from a registry

### Manual transfer

<code>docker save repo[:tag]...</code>	export an image/repo as a tarball
<code>docker load</code>	load images from a tarball
<code>docker-ssh<sup>11</sup> ...</code>	proposed script to transfer images between two daemons over ssh

<sup>11</sup><https://github.com/a-ba/docker-utils/>

## Transferring images



# Part 5.

## Docker builder

# What is the Docker builder ?

Rong Tao

Docker's builder relies on

- a **DSL** describing how to build an image
- a cache for storing previous builds and have quick iterations

The builder input is a **context**, i.e. a directory containing:

- a file named `Dockerfile` which describe how to build the container
- possibly other files to be used during the build

## Build an image

```
docker build [ -t tag ] path
```

→ build an image from the context located at *path* and optionally tag it as *tag*

The command:

1. makes a tarball from the content<sup>12</sup> of *path*
2. uploads the tarball to the docker daemon which will:
  - 2.1 execute the content of Dockerfile, committing an intermediate image **after each** command
  - 2.2 (if requested) tag the final image as *tag*

---

<sup>12</sup>unwanted files may be excluded if they match patterns listed in `.dockerignore`

# Dockerfile example

```
# base image: last debian release
```

```
FROM debian:wheezy
```

```
# install the latest upgrades
```

```
RUN apt-get update && apt-get -y dist-upgrade
```

```
# install nginx
```

```
RUN apt-get -y install nginx
```

```
# set the default container command
```

```
# -> run nginx in the foreground
```

```
CMD ["nginx", "-g", "daemon off;"]
```

```
# Tell the docker engine that there will be something listening on the tcp port 80
```

```
EXPOSE 80
```



# Dockerfile format

<https://docs.docker.com/reference/builder/>

- comments start with “#”
- commands fit on a single line  
(*possibly continued with \*)
- first command must be a FROM  
(indicates the parent image or scratch to start from scratch)

## Builder instructions (1/3)

### Instructions affecting the image filesystem

instruction	description
<b>FROM</b> <i>image scratch</i>	base image for the build
<b>COPY</b> <i>path dst</i>	copy <i>path</i> from the context into the container at location <i>dst</i>
<b>ADD</b> <i>src dst</i>	same as <b>COPY</b> but untar archives and accepts http urls
<b>RUN</b> <i>command</i>	run an arbitrary command inside the container

**Note:** commands may be expressed as a list (exec) or a string (shell)

*# exec form*

```
RUN ["apt-get", "update"]
```

*# shell form*

```
RUN apt-get update
```

*# equivalent to: RUN ["/bin/sh", "-c", "apt-get update"]*

## Builder instructions (2/3)

Instructions setting the default container config<sup>14</sup>

instruction	description
<b>CMD</b> <i>command</i>	command run inside the container
<b>ENTRYPOINT</b> <i>command</i>	entrypoint <sup>13</sup>
<b>USER</b> <i>name[:group]</i>	user running the command
<b>WORKDIR</b> <i>path</i>	working directory
<b>ENV</b> <i>name="value"...</i>	environment variables
<b>STOPSIGNAL</b> <i>signal</i>	signal to be sent to terminate the container ( <i>instead of SIGTERM</i> )
<b>HEALTHCHECK</b> <b>CMD</b> <i>command</i>	test command to check if the container works well
<b>EXPOSE</b> <i>port...</i>	listened TCP/UDP ports
<b>VOLUME</b> <i>path...</i>	mount-point for external volumes
<b>LABEL</b> <i>name="value"...</i>	arbitrary metadata

<sup>13</sup>the **ENTRYPOINT** is a command that wraps the **CMD** command

<sup>14</sup>i.e. the default configuration of containers running this image

## Builder instructions (3/3)

### Extra instructions


instruction	description
<b>ARG</b> <i>name[=value]</i>	build-time variables
<b>ON BUILD</b> <i>instruction</i>	instruction run when building a derived image

- build-time variables are usable anywhere in the Dockerfile (*by variable expansion*: \$VARNAME) and are tunable at build time: “**docker build --build-arg** *name=value ...*”
- instructions prefixed with **ONBUILD** are not run in this build, their execution is triggered when building a derived image

## Builder cache

Each layer created by the builder is fingerprinted according to:

- the ID of the previous image
- the command and its arguments
- the content of the imported files (for **ADD** and **COPY**)

 **RUN**'s side-effects are not fingerprinted

When rebuilding an image docker will reuse a previous image if its fingerprint is the same

## Good practices<sup>15</sup> for docker files

- use stable base images (eg. `debian:jessie`)
- run the app as PID 1 inside the container (to be killable)  
→ write `CMD ["app", "arg"]` instead of `CMD app arg`
- standardise the config, but allow the admin to override it with env variables or additional config files  
(eg. `ENV MYSQL_HOST="mysql"`)

---

<sup>15</sup>see also [https://docs.docker.com/engine/userguide/eng-image/dockerfile\\_best-practices/](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/)

## Multi-stage build (since v17.05)

```
#===== Stage 1: build the app from sources =====#
FROM debian:stretch AS builder
# update the package lists and install the build dependencies
RUN apt-get -qq update
RUN apt-get -qq install gcc make libacme-dev

# install the sources in /opt/src and build them
COPY . /opt/src
RUN cd /opt/src && ./configure && make

# install the files in a tmp dir and make an archive that we can deploy elsewhere
RUN cd /opt/src && make install DESTDIR=/tmp/dst \
    && cd /tmp/dst && tar czvf /tmp/myapp.tgz .

#===== Stage 2: final image =====#
FROM debian:stretch
# update the package lists and install the runtime dependencies
RUN apt-get -qq update
RUN apt-get -qq install libacme1.0

# install the app built in stage 1
COPY --from=builder /tmp/myapp.tgz /tmp/
RUN cd / && tar xzf /tmp/myapp.tgz

CMD ["myapp"]
```

# Part 6.

## Security

- host/container isolation
- container/container isolation
- other security considerations



## Security strategies

Docker containers are not really sandboxed from the host machine. They talk with the **same kernel**. You may want to consider strategies to reduce the risks of privilege escalation.

### Container/Host isolation

- run the container with an ordinary user (`docker run -u`)
- reduce root privileges (*capabilities, seccomp, apparmor*)
- configure a user namespace
- run the docker engine inside a VM

### Container/Container isolation

- disable intercontainer communications (`--icc=false`)
- isolate containers in different networks

## Running containers as normal user

```
docker run -u USER ...
```

should be safe, but...

- `setuid` executables in the docker image  
→ *should mount `/var/lib/docker` with `'-o nosuid'`*
- `setuid` executables in external volumes  
→ *should mount all data volumes with `'-o nosuid'`*
- `/etc/passwd` in the docker image  
→ *should use numeric ids: (`docker run -u UID:GID`)*

→ not easily enforcable if the image provider is malicious

## Reduced root capabilities

- kernel capabilities supported since docker v1.2
- containers use a default set limited to 14 capabilities<sup>16</sup>:

AUDIT_WRITE	CHOWN	NET_RAW	SETPCAP
DAC_OVERRIDE	FSETID	SETGID	KILL
NET_BIND_SERVICE	FOwner	SETUID	
SYS_CHROOT	MKNOD	SETFCAP	
- add additional capabilities: `docker run --cap-add=XXXXX ...`
- drop unnecessary capabilities: `docker run --cap-drop=XXXXX ...`  
→ should use `--cap-drop=all` for most containers

```
$ docker run --rm -t -i debian
root@04223cbb1334:/# ip addr replace 172.17.0.42/16 dev eth0
RTNETLINK answers: Operation not permitted
root@04223cbb1334:/# exit

$ docker run --rm -t -i --cap-add NET_ADMIN debian
root@9bf2a570a6a6:/# ip addr replace 172.17.0.42/16 dev eth0
root@9bf2a570a6a6:/#
```

<sup>16</sup>over the 38 capabilities defined in the kernel (man 7 capabilities)

## Reduced syscall whitelist

seccomp-bpf == fine-grained access control to kernel syscalls

- enabled by default since docker v1.10
- default built-in profile<sup>17</sup> whitelists only harmless syscalls<sup>18</sup>
- alternative configs:
  - disable seccomp (`--security-opt=seccomp:unconfined`)
  - provide a customised profile (derived from the default<sup>19</sup>)

```
$ docker run --rm debian date -s 2016-01-01
date: cannot set date: Operation not permitted
$ docker run --rm --cap-add sys_time debian date -s 2016-01-01
date: cannot set date: Operation not permitted
$ docker run --rm --security-opt seccomp:unconfined debian date -s 2016-01-01
date: cannot set date: Operation not permitted
$ docker run --rm --cap-add sys_time --security-opt seccomp:unconfined debian date -s 2016-01-01
Fri Jan 1 00:00:00 UTC 2016
```

<sup>17</sup> <https://docs.docker.com/engine/security/seccomp/>

<sup>18</sup> harmful means everything that deals with administration (eg: set time) or debugging (eg: ptrace)

<sup>19</sup> <https://github.com/moby/moby/blob/master/profiles/seccomp/default.json>

## User namespaces

since docker v1.10 but not enabled by default

- UIDs/GIDs inside the containers mapped to another range outside the container
- useful for:
  - preventing fs-based attacks (*eg: root user inside the container creates a setuid executable in an external volume*)
  - isolating docker users from each other (*one docker daemon for each user, with uids remapped to different ranges*)
- limits (as of v1.10)
  - global config only (daemon scope)
  - coarse mapping only (hardcoded range: 0..65535)

## Docker is not a sandbox !

Even with *capabilities+seccomp+user\_namespaces* enabled, you may still be vulnerable, because the kernel's attack surface is **big**

### CVE-2019-5736

*runc through 1.0-rc6, as used in Docker before 18.09.2 and other products, **allows attackers to overwrite the host runc binary (and consequently obtain host root access)***

### CVE-2018-15664

*In Docker through 18.06.1-ce-rc2, the API endpoints behind the 'docker cp' command are vulnerable to a symlink-exchange attack with Directory Traversal, **giving attackers arbitrary read-write access to the host filesystem with root privileges***

## Run the docker engine inside a VM

Hypervisors have a smaller attack surface and are much more mature than containers. **Use a VM if you need good isolation!**

- either manually-administrated VMs
- either transparently-launched VMs
  - on a per-engine basis (docker daemon inside a VM)  
**docker machine:** <https://docs.docker.com/machine/overview/>
  - on a per-container basis (each container in a separate VM)  
**kata containers:** <https://katacontainers.io/>  
**runv:** <https://github.com/hyperhq/runv>  
**gvisor:** <https://github.com/google/gvisor>

## Container/Container isolation

- by default all containers can connect to any other container (located in the same bridge)
  - run the daemon with `--icc=false`
    - all communications filtered by default
    - whitelist-based access with `--link`  
(*only EXPOSEd ports will be whitelisted*)
  - attach containers to different networks
- by default RAW sockets are enabled (allows ARP spoofing)<sup>20</sup>  
→ use `docker run --cap-drop=NET_RAW`

---

<sup>20</sup><http://lwn.net/Articles/689453>



## Other security considerations

- images are immutable
  - need a process to apply automatic security upgrades, e.g:
    - apply upgrades & commit a new image
    - regenerate the image from the Dockerfile
- docker engine control == root on the host machine
  - give access to the docker socket only to trusted users
- avoid `docker run --privileged` (gives full root access)
- beware of symlinks in external volumes

eg. ctr1 binds /data, ctr2 binds /data/subdir, if both are malicious and cooperate, ctr1 replaces /data/subdir with a symlink to /, then on restart ctr2 has access to the whole host filesystem

  - avoid binding subdirectories, prefer using named volumes

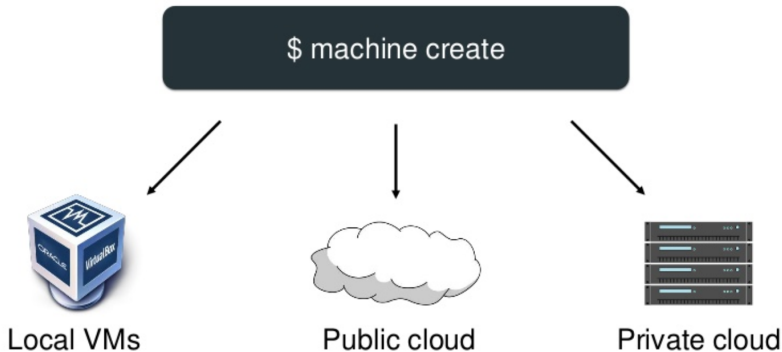
# Part 7.

## Docker Ecosystem

- infrastructure
  - docker machine (provisioning)
  - docker swarm (clustering)
  - swarm mode (clustering)
  - underlying projects (moby, containerd, infrakit, ...)
- container deployment & configuration
  - docker compose
- image distribution
  - docker distribution (registry)
  - docker notary (content trust, image signing)

# Docker Machine

abstraction for provisioning and using docker hosts





# Docker Compose

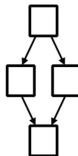
configure and deploy a collection of containers



Text file



docker-compose up



## group.yml

```
name: counter

containers:
  web:
    build: .
    command: python app.py
    ports:
      - "5000:5000"
    volumes:
      - ./code
    links:
      - redis
  redis:
    image: redis:latest
```

# Part 8.

## The Future is Now

- swarm mode (since v1.12)
- plugins (since v1.13)
- experimental features
- Docker EE & time-based releases
- The Orchestration Wars

# The Future is Now

- Swarm mode (since v1.12)
  - service abstraction
    - scaling
    - service discovery & load balancing
    - rolling updates
  - stack deployment (docker-compose) (since v1.13)
  - secrets management (since v1.13) + config objects (since v17.06)
- plugins API for datacenter integration (since v1.13)
  - volume plugins (eg: flocker)
  - network plugins (eg: contiv)
  - authorization plugins
  - swarm secrets (since v17.07)

# Docker CE & Docker EE

since march 2017

Docker inc's business strategy:

1. be flexible and interoperable with everybody (especially cloud providers) so that no competing tool emerges

→ open source engine, plugin API for network, storage, authorization integrations

2. sell Docker EE

docker EE = docker CE + support + off-the-shelves datacenter management  
(ldap integration, role-based access-control, security scanning, vulnerability monitoring)



# Time-based release

since march 2017 (docker v17.03.0-ce)

- Docker CE
  - open source
  - edge version released every month
  - stable version released every 3 months
  - security upgrades during 4 months
- Docker EE
  - proprietary
  - stable version released every 3 months
  - security upgrades during 1 year

# The Orchestration Wars

The *Container Wars* will actually be the *Orchestration Wars*

- under the hood the base building blocs (runc, containerd) are open and the competitors cooperate to keep them standard.
- docker itselfs is still free software, although the company ulture is shifting towards something more “corporate”
- the real fight will be on orchestration solutions
  - managing clouds, service hosting
  - swarm has opponents (Mesos, Kubernetes, Openshift, ...) and is lagging.

# Apache Mesos

- predates Docker
- designed for very large clusters
- agnostic to the virtualisation technology
  - multiple virtualisation tool may coexist in the same cluster
  - two-level management
- hard to configure

# Kubernetes (k8s)

- project started in 2014 by a group of google developers
- inspired from Google's internal orchestration framework
- large scale, very sophisticated, not easy to learn
- now hosted by a foundation and adopted by others that use it as their orchestration backend
  - Openshift
  - Docker EE

# The Open Container Initiative (OCI)

<https://github.com/opencontainers/>

A Linux Foundation standard for linux containers:

- v1.0.0 released in July 2017
  - runtime-spec (launching containers)
  - image-spec (image interoperability)

orchestration

docker

kubernetes

mesos

openshift

podman

...

**OCI Standard**

runtime

runc

kata

gvisor

runv

clear  
containers

...