Docker Tutorial

Anthony Baire

Université de Rennes 1 / UMR IRISA

January 22, 2018

This tutorial is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 France License
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
“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 $\rightarrow$ Dockerfile = recipe to rebuild the env from scratch
  - binary $\rightarrow$ docker image = immutable snapshot of the software with its runtime environment
    $\rightarrow$ 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$^1$)
  - 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

$^1$several possible methods: overlayfs (default), btrfs, lvm, zfs, aufs
Installation

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

Native installation:

- requires linux kernel > 3.8

Docker Machine:

- a command for provisionning an 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
# Container management commands

<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker create image [ command ]</td>
<td>create the container</td>
</tr>
<tr>
<td>docker run image [ command ]</td>
<td>= create + start</td>
</tr>
<tr>
<td>docker rename container new_name</td>
<td>rename the container</td>
</tr>
<tr>
<td>docker update container</td>
<td>update the container config</td>
</tr>
<tr>
<td>docker start container...</td>
<td>start the container</td>
</tr>
<tr>
<td>docker stop container...</td>
<td>graceful(^2) stop</td>
</tr>
<tr>
<td>docker kill container...</td>
<td>kill (SIGKILL) the container</td>
</tr>
<tr>
<td>docker restart container...</td>
<td>= stop + start</td>
</tr>
<tr>
<td>docker pause container...</td>
<td>suspend the container</td>
</tr>
<tr>
<td>docker unpause container...</td>
<td>resume the container</td>
</tr>
<tr>
<td>docker rm [ -f(^3) ] container...</td>
<td>destroy the container</td>
</tr>
</tbody>
</table>

\(^2\) send SIGTERM to the main process + SIGKILL 10 seconds later

\(^3\) -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

- `--name=NAME` Assign a name to the container
- `--labels=LABEL` Set meta data on a container
- `--network=NAME` Network name
- `--publish=SERVICE:PORT` Publish a container’s port(s) to the host
- `--publish-all=false` Publish all exposed ports to random ports
- `--privileged=false` Give extended privileges to this container
**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)

```bash
$ docker run debian /bin/hostname
f0d0720bd37
$ 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

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

→ with a tty (-t)

```bash
$ 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

```
$ 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@78ff08f46cde:/# date
Tue Jan 20 17:52:01 UTC 2015
root@78ff08f46cde:/# ^C
root@78ff08f46cde:/#
```

---

4^C only detaches the terminal, the container keeps running in background
$ docker run debian whoami
root

$ docker run -u nobody debian whoami
nobody

$ docker run debian pwd
/

$ docker run -w /opt debian pwd
/opt
docker run — override defaults (2/2)

environment variables (\texttt{-e})

```bash
$ 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 (\texttt{-h})

```bash
$ 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
0bd3cb464ff68eaf9fc43f0241911eb207fefd9c1341a0850e8804b7445ccd21
$ 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

```bash
$ 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

```bash
$ 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

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

Remove all zombie containers

```bash
$ docker ps -a
CONTAINER ID    IMAGE       COMMAND                  CREATED             STATUS              RC
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

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

5 with `-f`, `docker logs` follows the output (à la `tail -f`)
6 `docker top` is the equivalent of the `ps` command in unix
7 `docker stats` is the equivalent of the `top` command in unix
## Interacting with the container

<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker attach <code>container</code></td>
<td>attach to a running container (stdin/stdout/stderr)</td>
</tr>
<tr>
<td>docker cp <code>container:path</code> hostpath</td>
<td>copy files from the container</td>
</tr>
<tr>
<td>docker cp <code>hostpath</code></td>
<td>copy files into the container</td>
</tr>
<tr>
<td>docker export <code>container</code></td>
<td>export the content of the container (tar archive)</td>
</tr>
<tr>
<td>docker exec <code>container</code> args...</td>
<td>run a command in an existing container (useful for debugging)</td>
</tr>
<tr>
<td>docker wait <code>container</code></td>
<td>wait until the container terminates and return the exit code</td>
</tr>
<tr>
<td>docker commit <code>container image</code></td>
<td>commit a new docker image (snapshot of the container)</td>
</tr>
</tbody>
</table>
$ 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)
**docker run** — mount external volumes

```bash
docker run -v /hostpath:/containerpath[:ro] ...
```

`-v` mounts the location `/hostpath` from the host filesystem at the location `/containerpath` inside the container.

With the“`:ro`” suffix, the mount is read-only.

**Purposes:**

- store persistent data outside the container
- provide inputs: data, config files, … (read-only mode)
- inter-process communication (unix sockets, named pipes)
mount examples (1/2)

Persistent data

```bash
$ docker run --rm -t -i -v /tmp/persistent:/persistent debian
root@0aeedfeb7bf9:/# echo "blahblah" >/persistent/foo
root@0aeedfeb7bf9:/# 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)

```bash
$ 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

```bash
$ 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

```bash
$ 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 halfloat 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

```bash
$ 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
```

---

8[https://docs.docker.com/engine/tutorials/dockervolumes/](https://docs.docker.com/engine/tutorials/dockervolumes/)
**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)**

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

```bash
$ 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
```

---

9 since v1.9.0, links are superseded by user-defined networks
Legacy links

⚠️ deprecated feature

![Diagram of Docker network with containers and links](image-url)
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

```bash
docker network create --subnet=10.0.1.0/24 br-prod
```

DOCKER HOST

- `br-prod` 10.0.1.1/24
- `bridge` 172.17.0.1/16
- `eth0` a.b.c.d (public address)

Public network
User-defined networks example

docker run --net=br-prod ...

bridge 172.17.0.1/16
br-prod 10.0.1.1/24
eth0 a.b.c.d (public address)

Public network
User-defined networks example

**DOCKER HOST**

- br-prod: 10.0.1.1/24
- bridge: 172.17.0.1/16
- eth0: a.b.c.d (public address)

Public network
User-defined networks example

docker network create --subnet=10.0.2.0/24 br-qualif
User-defined networks example

```
docker run --net=br-qualif
```

![Diagram showing user-defined networks example](image)
User-defined networks example

docker network connect bridge c0
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**

**DOCKER HOST**

**CONTAINERS**

- 172.17.0.1
- 172.17.0.14
- 172.17.6.16

**Internal bridge** 172.17.0.0/16

**TCP redirection**

- 172.17.42.1
- a.b.c.d (public address)
- tcp port 80
- tcp port 8080

**docker run --publish 80:8080**

**Public network**
publish example

bind to all host addresses

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

$ wget -nv http://localhost/
$ wget -nv http://172.17.42.1/
```

bind to 127.0.0.1

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

$ wget -nv http://localhost/
$ 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

- multiple web apps deployed in the same engine
- each app may have multiple servers (frontend, db, storage, computing...)
- apps are isolated from each other (virtual networks)
The whole picture

HTTP frontend
- reverse-proxy
- with virtual hosting
- with state-of-art security (TLS, PFS, HSTS, ...)

Public network

a.b.c.d (public address)
+ DNS aliases
app1.irisa.fr
app2.irisa.fr
app3.irisa.fr
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 ID (hashed from the image content)

- may be tagged\(^{10}\) with a human-friendly name
  eg: debian:wheezy debian:jessie debian:latest

\(^{10}\)possibly multiple times
## Image management commands

<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker images</td>
<td>list all local images</td>
</tr>
<tr>
<td>docker history <code>image</code></td>
<td>show the image history (list of ancestors)</td>
</tr>
<tr>
<td>docker inspect <code>image...</code></td>
<td>show low-level infos (in json format)</td>
</tr>
<tr>
<td>docker tag <code>image tag</code></td>
<td>tag an image</td>
</tr>
<tr>
<td>docker commit <code>container image</code></td>
<td>create an image (from a container)</td>
</tr>
<tr>
<td>docker import `url</td>
<td>- [tag]`</td>
</tr>
<tr>
<td>docker rmi <code>image...</code></td>
<td>delete images</td>
</tr>
</tbody>
</table>
Example: images & containers
Example: images & containers

docker pull img
Example: images & containers

docker run --name ctrl1 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

<table>
<thead>
<tr>
<th>img:latest</th>
<th>8f42dfb77aa9</th>
<th>ctr5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cf7a2e2d1ed6</td>
<td>30e0db62cf5c</td>
</tr>
<tr>
<td></td>
<td>3881989037c4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c7b2e63dcca8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>92d666cd711c</td>
<td></td>
</tr>
</tbody>
</table>

img:bis

scratch
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

<table>
<thead>
<tr>
<th>c7b2e63dcca8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3881989037c4</td>
</tr>
<tr>
<td>30e0db62cf5c</td>
</tr>
<tr>
<td>ctr5</td>
</tr>
<tr>
<td>92d666cd711c</td>
</tr>
<tr>
<td>scratch</td>
</tr>
</tbody>
</table>
Example: images & containers

docker rm ctr5
Example: images & containers

docker rmi 30e0
Images vs. Layers

docker &lt; v1.10
no distinction between images & layers

tagspics

foo:latest 30e0db62cf5c

3881989037c4
c7b2e63dcca8
92d666cd711c

scratch

docker &gt;= v1.10
layers are hidden to the user
(implementation detail)

tagspics

foo:latest fd15304e605c

30e0db62cf5c
3881989037c4
c7b2e63dcca8
92d666cd711c

foo:bar 31c98fa90bc4

3881989037c4
c7b2e63dcca8
92d666cd711c
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”

```bash
$ docker images

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>VIRTUAL SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>debian</td>
<td>8</td>
<td>835c4d274060</td>
<td>2 weeks ago</td>
<td>122.6 MB</td>
</tr>
<tr>
<td>debian</td>
<td>8.0</td>
<td>835c4d274060</td>
<td>2 weeks ago</td>
<td>122.6 MB</td>
</tr>
<tr>
<td>debian</td>
<td>jessie</td>
<td>835c4d274060</td>
<td>2 weeks ago</td>
<td>122.6 MB</td>
</tr>
<tr>
<td>debian</td>
<td>rc-buggy</td>
<td>350a74df81b1</td>
<td>7 months ago</td>
<td>159.9 MB</td>
</tr>
<tr>
<td>debian</td>
<td>experimental</td>
<td>36d6c9c7df4c</td>
<td>7 months ago</td>
<td>159.9 MB</td>
</tr>
<tr>
<td>debian</td>
<td>6.0.9</td>
<td>3b36e4176538</td>
<td>7 months ago</td>
<td>112.4 MB</td>
</tr>
<tr>
<td>debian</td>
<td>squeeze</td>
<td>3b36e4176538</td>
<td>7 months ago</td>
<td>112.4 MB</td>
</tr>
<tr>
<td>debian</td>
<td>wheezy</td>
<td>667250f9a437</td>
<td>7 months ago</td>
<td>115 MB</td>
</tr>
<tr>
<td>debian</td>
<td>latest</td>
<td>667250f9a437</td>
<td>7 months ago</td>
<td>115 MB</td>
</tr>
<tr>
<td>debian</td>
<td>7.5</td>
<td>667250f9a437</td>
<td>7 months ago</td>
<td>115 MB</td>
</tr>
<tr>
<td>debian</td>
<td>unstable</td>
<td>24a4621560e4</td>
<td>7 months ago</td>
<td>123.6 MB</td>
</tr>
<tr>
<td>debian</td>
<td>testing</td>
<td>7f5d8ca9fddc</td>
<td>7 months ago</td>
<td>121.8 MB</td>
</tr>
<tr>
<td>debian</td>
<td>stable</td>
<td>caa04aa09d69</td>
<td>7 months ago</td>
<td>115 MB</td>
</tr>
<tr>
<td>debian</td>
<td>sid</td>
<td>f3d4759f77a7</td>
<td>7 months ago</td>
<td>123.6 MB</td>
</tr>
<tr>
<td>debian</td>
<td>7.4</td>
<td>e565fbbbc6033</td>
<td>9 months ago</td>
<td>115 MB</td>
</tr>
<tr>
<td>debian</td>
<td>7.3</td>
<td>b5fe16f2ccba</td>
<td>11 months ago</td>
<td>117.8 MB</td>
</tr>
</tbody>
</table>
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/](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/](https://docs.docker.com/reference/api/registry_api/)
Tagging conventions (2/2)

Use slashes to delimit namespaces (for subprojects):

<table>
<thead>
<tr>
<th>image name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debian</td>
<td>(semi-)official debian images</td>
</tr>
<tr>
<td>fedora</td>
<td>official fedora images</td>
</tr>
<tr>
<td>fedora/apache</td>
<td>apache images provided by the fedora project</td>
</tr>
<tr>
<td>fedora/couchdb</td>
<td>couchdb images provided by the fedora project</td>
</tr>
</tbody>
</table>
### Image transfer commands

#### Using the registry API

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>docker pull repo[:tag]...</code></td>
<td>pull an image/repo from a registry</td>
</tr>
<tr>
<td><code>docker push repo[:tag]...</code></td>
<td>push an image/repo from a registry</td>
</tr>
<tr>
<td><code>docker search text</code></td>
<td>search an image on the official registry</td>
</tr>
<tr>
<td><code>docker login ...</code></td>
<td>login to a registry</td>
</tr>
<tr>
<td><code>docker logout ...</code></td>
<td>logout from a registry</td>
</tr>
</tbody>
</table>

#### Manual transfer

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>docker save repo[:tag]...</code></td>
<td>export an image/repo as a tarbal</td>
</tr>
<tr>
<td><code>docker load</code></td>
<td>load images from a tarball</td>
</tr>
<tr>
<td><code>docker-ssh</code>⁷¹</td>
<td>proposed script to transfer images between two daemons over ssh</td>
</tr>
</tbody>
</table>

⁷¹[https://github.com/a-ba/docker-utils/](https://github.com/a-ba/docker-utils/)
Transferring images

user[/image][:tag]

host/user[/image][:tag]

official registry

3rd party registry

push

pull

push

pull

push
Part 5.
Docker builder
What is the Docker builder?

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\(^{12}\) 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*

---
\(^{12}\) unwanted files may be excluded if they match patterns listed in .dockerignore
# 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 somethong 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

<table>
<thead>
<tr>
<th>instruction</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FROM</strong> image</td>
<td>base image for the build</td>
</tr>
<tr>
<td><strong>COPY</strong> path dst</td>
<td>copy path from the context into the container at location dst</td>
</tr>
<tr>
<td><strong>ADD</strong> src dst</td>
<td>same as COPY but untar archives and accepts http urls</td>
</tr>
<tr>
<td><strong>RUN</strong> command</td>
<td>run an arbitrary command inside the container</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>instruction</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMD <em>command</em></td>
<td>command run inside the container</td>
</tr>
<tr>
<td>ENTRYPOINT <em>command</em></td>
<td>entrypoint(^{13})</td>
</tr>
<tr>
<td>USER <em>name[:group]</em></td>
<td>user running the command</td>
</tr>
<tr>
<td>WORKDIR <em>path</em></td>
<td>working directory</td>
</tr>
<tr>
<td>ENV <em>name</em>=&quot;value&quot;...</td>
<td>environment variables</td>
</tr>
<tr>
<td>STOPSIGNALL <em>signal</em></td>
<td>signal to be sent to terminate the container <em>(instead of SIGTERM)</em></td>
</tr>
<tr>
<td>HEALTHCHECK CMD <em>command</em></td>
<td>test command to check if the container works well</td>
</tr>
<tr>
<td>EXPOSE <em>port...</em></td>
<td>listened TCP/UDP ports</td>
</tr>
<tr>
<td>VOLUME <em>path...</em></td>
<td>mount-point for external volumes</td>
</tr>
<tr>
<td>LABEL <em>name</em>=&quot;value&quot;...</td>
<td>arbitrary metadata</td>
</tr>
</tbody>
</table>

\(^{13}\) the **ENTRYPOINT** is a command that wraps the **CMD** command

\(^{14}\) i.e. the default configuration of containers running this image
Builder instructions (3/3)

Extra instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG name[=value]</td>
<td>build-time variables</td>
</tr>
<tr>
<td>ON BUILD instruction</td>
<td>instruction run when building a derived image</td>
</tr>
</tbody>
</table>

- 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\textsuperscript{15} 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"`)

\textsuperscript{15}see also 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 an install the build dependencies
RUN apt-get -qqy update
RUN apt-get -qqy 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 -qqy update
RUN apt-get -qqy libacme1.0

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

CMD ["myapp"]
Part 6.
Security

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

Container/Host isolation

- run the docker engine inside a VM
- run the container with an ordinary user (`docker run -u`)
- reduce root privileges
  - capabilities (since v1.2)
  - seccomp (since v1.10)
  - AppArmor/SELinux
- use a user namespace (since v1.10)

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 attacker is the provider of the image
Reduced root capabilities

- kernel capabilities supported since v1.2
- containers use a default set limited to 14 capabilities\(^{16}\):
  - `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

\(^{16}\) 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 v1.10
- default built-in profile\(^{17}\) whitelists only harmless syscalls\(^{18}\)
- alternative configs:
  - disable seccomp (`--security-opt=seccomp:unconfined`)
  - provide a customised profile (derived from the default\(^{19}\))

```bash
$ 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
```

\(^{17}\) [https://docs.docker.com/engine/security/seccomp/](https://docs.docker.com/engine/security/seccomp/)
\(^{18}\) harmful means everything that deals with administration (eg: set time) or debugging (eg: ptrace)
\(^{19}\) [https://github.com/moby/moby/blob/master/profiles/seccomp/default.json](https://github.com/moby/moby/blob/master/profiles/seccomp/default.json)
User namespaces

since 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)
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)
  → use `docker run --cap-drop=NET_RAW`

---

20 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)

- avoid the insecure v1 registry API (for push/pull)
  → run the daemon with `--insecure-registry=false --disable-legacy-registry`

- 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 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

$ machine create

Local VMs

Public cloud

Private cloud
Docker Swarm
manage a cluster of hosts running docker

⚠️ Docker Inc. folks are misleading: the name **swarm** is actually used for two different products:

- **docker swarm** (or **legacy swarm** or just **swarm**)
  - early solution (first released in dec 2014)
  - standalone server
  - superset of the docker engine API
  - requires an external discovery service (eg. etcd, consul)
  - network-agnostic (overlay networks to be configured separately)

- **the swarm mode**
  - embedded within the docker engine (since v1.12 in july 2016)
  - turnkey cluster (integrated discovery service, distributed, network aware, encryption by default)
  - API break: introduces the **service** abstraction
Docker Compose

configure and deploy a collection of containers

```
# 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)
Experimental features

as of v17.09

- checkpoint and restore
- distributed application bundles
- vlan network driver
- service logs
  https://github.com/docker/cli/blob/master/docs/reference/commandline/service_logs.md
- docker build --squash
  https://github.com/docker/docker/pull/22641
- digital signature of images (docker trust)
  https://github.com/docker/cli/pull/472

---

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 blocks (*runc, containerd*) are open and the competitors cooperate to keep them standard.

- docker itselfs is still free software, although the company culture 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 fundation and adopted by others that use it as their orchestration backend
  - Openshift
  - Docker EE