

Learning to Ride the Whale: An Introduction to Containers

Be kind to the WiFi!

*Don't stream videos or download big files during the workshop.
Thank you!*

Slides: <http://container.training/>

A brief introduction

- This was initially written to support in-person, instructor-led workshops and tutorials
- These materials are maintained by [Jérôme Petazzoni](#) and [multiple contributors](#)
- You can also follow along on your own, at your own pace
- We included as much information as possible in these slides
- We recommend having a mentor to help you ...
- ... Or be comfortable spending some time reading the Docker [documentation](#) ...
- ... And looking for answers in the [Docker forums](#), [StackOverflow](#), and other outlets

About these slides

- All the content is available in a public GitHub repository:

<https://github.com/jpetazzo/container.training>

- You can get updated "builds" of the slides there:

<http://container.training/>

About these slides

- All the content is available in a public GitHub repository:

<https://github.com/jpetazzo/container.training>

- You can get updated "builds" of the slides there:

<http://container.training/>

- Typos? Mistakes? Questions? Feel free to hover over the bottom of the slide ...



Try it! The source file will be shown and you can view it on GitHub and fork and edit it.



Extra details

- This slide has a little magnifying glass in the top left corner
- This magnifying glass indicates slides that provide extra details
- Feel free to skip them if:
 - you are in a hurry
 - you are new to this and want to avoid cognitive overload
 - you want only the most essential information
- You can review these slides another time if you want, they'll be waiting for you ☺

Chapter 1

- Docker 30,000ft overview
- History of containers ... and Docker
- Our training environment
- Our first containers
- Background containers
- Restarting and attaching to containers

Chapter 2

- Understanding Docker images
- Building Docker images with a Dockerfile
- `CMD` and `ENTRYPOINT`
- Copying files during the build

Chapter 3

- Reducing image size
- Multi-stage builds
- Publishing images to the Docker Hub

Chapter 4

- Naming and inspecting containers
- Labels
- Getting inside a container

Chapter 5

- Container networking basics
- Working with volumes
- Limiting resources

Chapter 6

- Deep dive into container internals
- Namespaces
- Control groups
- Security features
- Copy-on-write filesystems

Chapter 7

- Docker Engine and other container engines
- The container ecosystem
- Links and resources



Docker 30,000ft overview

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Docker 30,000ft overview

In this lesson, we will learn about:

- Why containers (non-technical elevator pitch)
- Why containers (technical elevator pitch)
- How Docker helps us to build, ship, and run
- The history of containers

We won't actually run Docker or containers in this chapter (yet!).

Don't worry, we will get to that fast enough!

Elevator pitch (for your manager, your boss...)

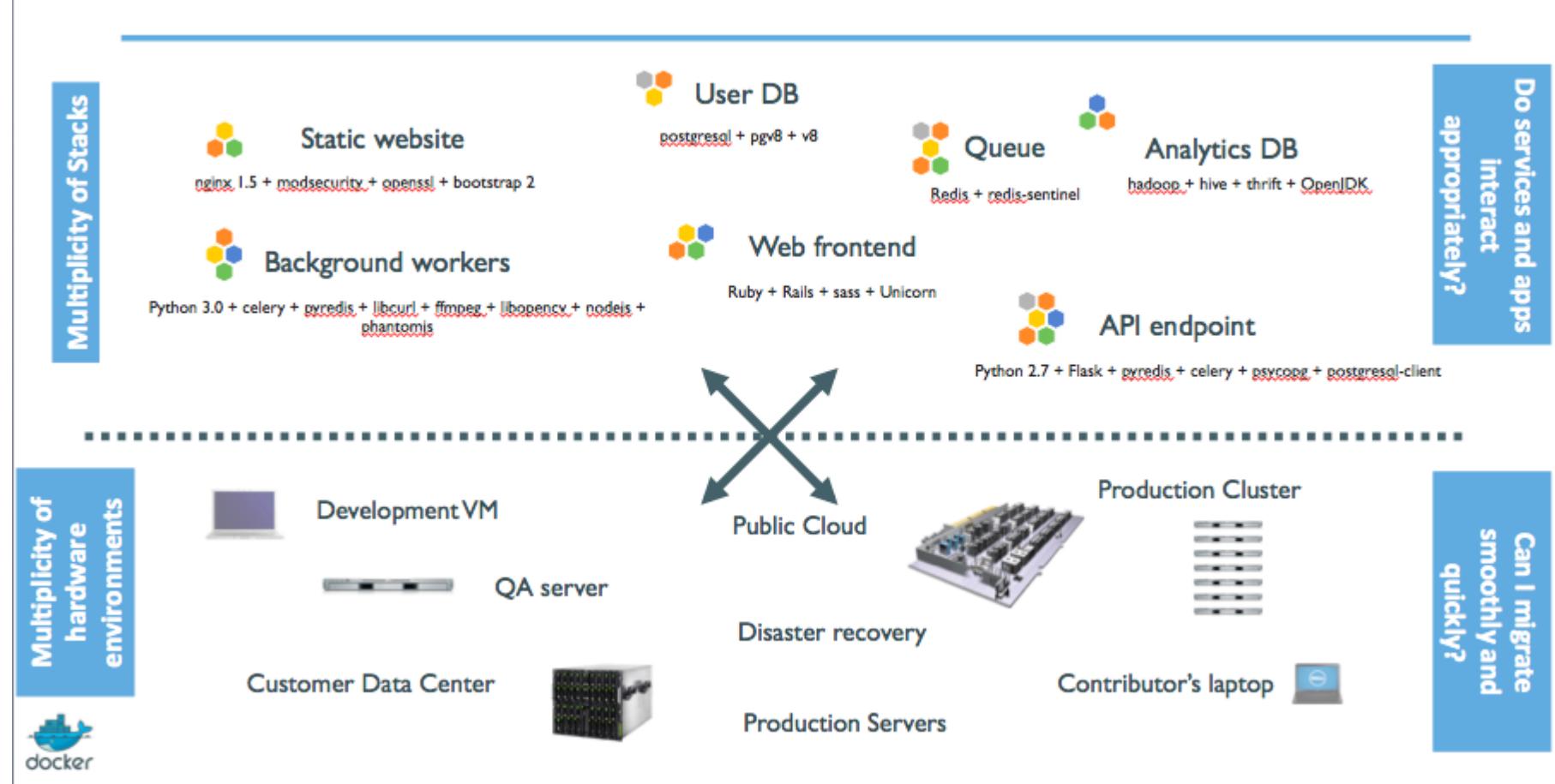
OK... Why the buzz around containers?

- The software industry has changed
- Before:
 - monolithic applications
 - long development cycles
 - single environment
 - slowly scaling up
- Now:
 - decoupled services
 - fast, iterative improvements
 - multiple environments
 - quickly scaling out

Deployment becomes very complex

- Many different stacks:
 - languages
 - frameworks
 - databases
- Many different targets:
 - individual development environments
 - pre-production, QA, staging...
 - production: on prem, cloud, hybrid

The deployment problem

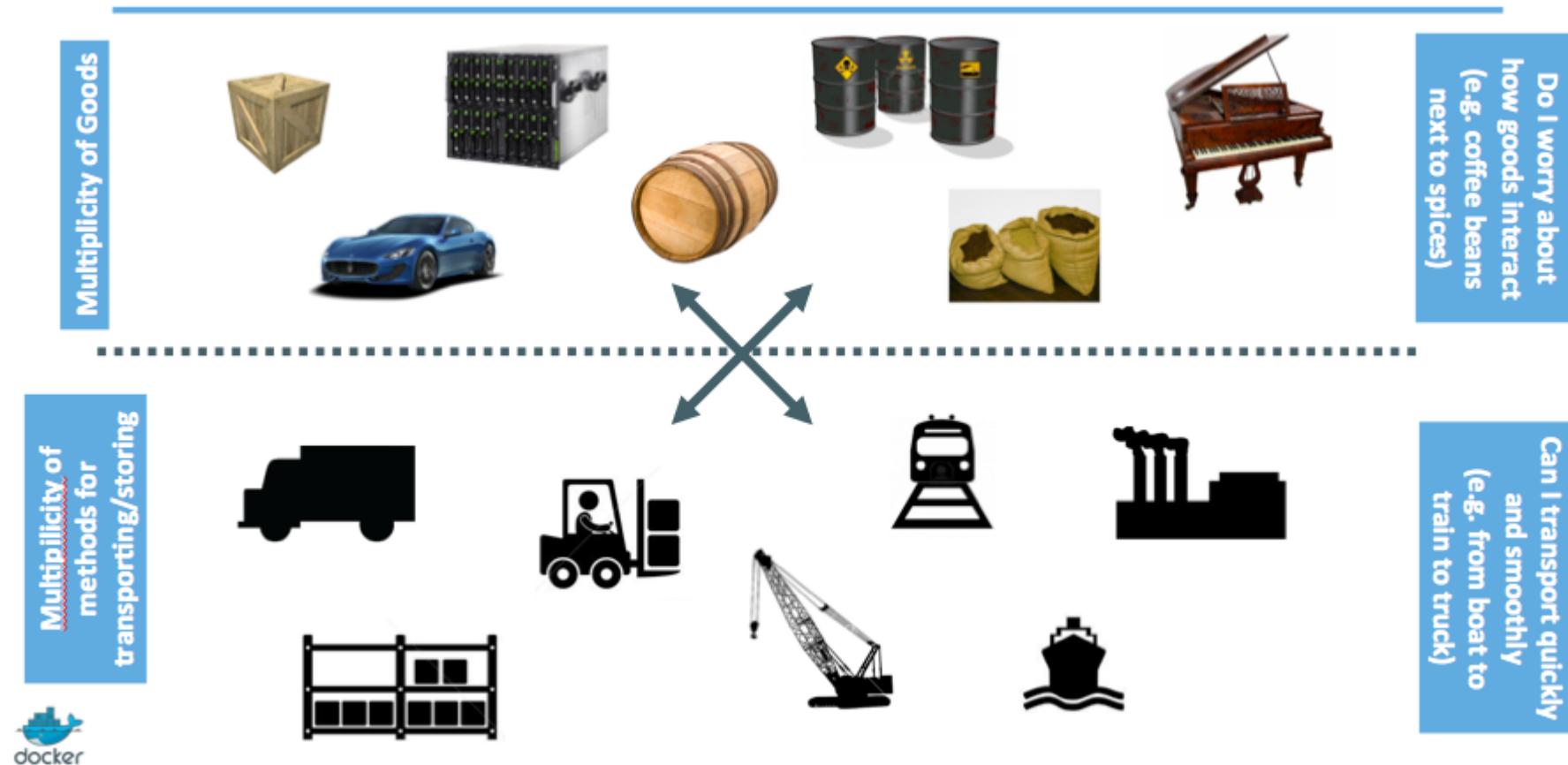


The matrix from hell

	Static website	?	?	?	?	?	?	?
	Web frontend	?	?	?	?	?	?	?
	Background workers	?	?	?	?	?	?	?
	User DB	?	?	?	?	?	?	?
	Analytics DB	?	?	?	?	?	?	?
	Queue	?	?	?	?	?	?	?
		Developmen t VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor' s laptop	Customer Servers
								



The parallel with the shipping industry



Intermodal shipping containers



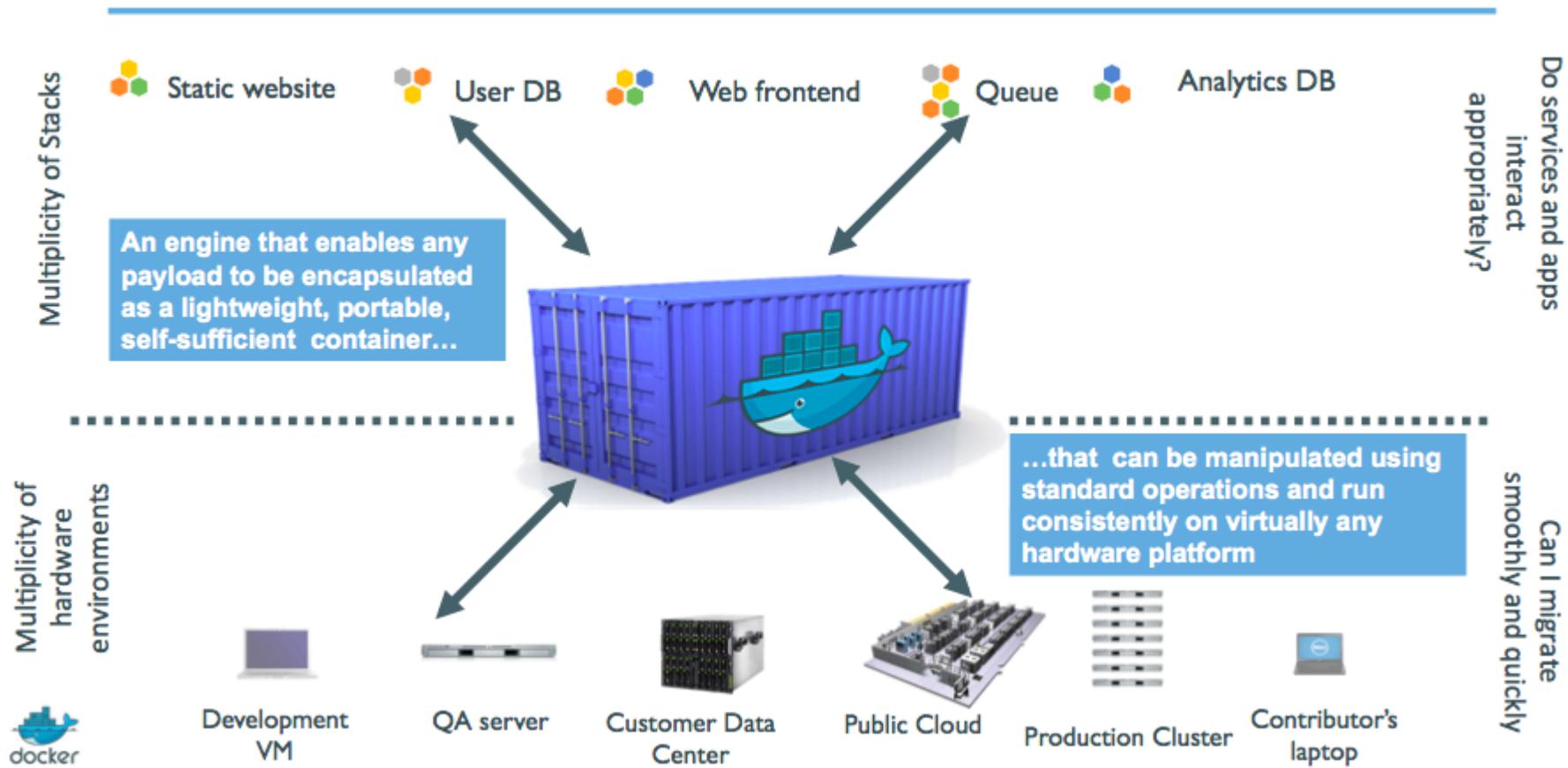
A new shipping ecosystem



- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)
→ massive globalization
- 5000 ships deliver 200M containers per year



A shipping container system for applications



Eliminate the matrix from hell

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

Results

- Dev-to-prod reduced from 9 months to 15 minutes (ING)
- Continuous integration job time reduced by more than 60% (BBC)
- Deploy 100 times a day instead of once a week (GILT)
- 70% infrastructure consolidation (MetLife)
- 60% infrastructure consolidation (Intesa Sanpaolo)
- 14x application density; 60% of legacy datacenter migrated in 4 months (GE Appliances)
- etc.

Elevator pitch (for your fellow devs and ops)

Escape dependency hell

1. Write installation instructions into an `INSTALL.txt` file
2. Using this file, write an `install.sh` script that works *for you*
3. Turn this file into a `Dockerfile`, test it on your machine
4. If the Dockerfile builds on your machine, it will build *anywhere*
5. Rejoice as you escape dependency hell and "works on my machine"

Never again "worked in dev - ops problem now!"

On-board developers and contributors rapidly

1. Write Dockerfiles for your application components
2. Use pre-made images from the Docker Hub (mysql, redis...)
3. Describe your stack with a Compose file
4. On-board somebody with two commands:

```
git clone ...
docker-compose up
```

With this, you can create development, integration, QA environments in minutes!



Implement reliable CI easily

1. Build test environment with a Dockerfile or Compose file
2. For each test run, stage up a new container or stack
3. Each run is now in a clean environment
4. No pollution from previous tests

Way faster and cheaper than creating VMs each time!



Use container images as build artefacts

1. Build your app from Dockerfiles
2. Store the resulting images in a registry
3. Keep them forever (or as long as necessary)
4. Test those images in QA, CI, integration...
5. Run the same images in production
6. Something goes wrong? Rollback to previous image
7. Investigating old regression? Old image has your back!

Images contain all the libraries, dependencies, etc. needed to run the app.



Decouple "plumbing" from application logic

1. Write your code to connect to named services ("db", "api"...)
2. Use Compose to start your stack
3. Docker will setup per-container DNS resolver for those names
4. You can now scale, add load balancers, replication ... without changing your code

Note: this is not covered in this intro level workshop!



What did Docker bring to the table?

Docker before/after



Formats and APIs, before Docker

- No standardized exchange format.
(No, a rootfs tarball is *not* a format!)
- Containers are hard to use for developers.
(Where's the equivalent of `docker run debian?`)
- As a result, they are *hidden* from the end users.
- No re-usable components, APIs, tools.
(At best: VM abstractions, e.g. libvirt.)

Analogy:

- Shipping containers are not just steel boxes.
- They are steel boxes that are a standard size, with the same hooks and holes.



Formats and APIs, after Docker

- Standardize the container format, because containers were not portable.
- Make containers easy to use for developers.
- Emphasis on re-usable components, APIs, ecosystem of standard tools.
- Improvement over ad-hoc, in-house, specific tools.



Shipping, before Docker

- Ship packages: deb, rpm, gem, jar, homebrew...
- Dependency hell.
- "Works on my machine."
- Base deployment often done from scratch (debootstrap...) and unreliable.



Shipping, after Docker

- Ship container images with all their dependencies.
- Images are bigger, but they are broken down into layers.
- Only ship layers that have changed.
- Save disk, network, memory usage.



Example

Layers:

- CentOS
- JRE
- Tomcat
- Dependencies
- Application JAR
- Configuration



Devs vs Ops, before Docker

- Drop a tarball (or a commit hash) with instructions.
- Dev environment very different from production.
- Ops don't always have a dev environment themselves ...
 - ... and when they do, it can differ from the devs'.
- Ops have to sort out differences and make it work ...
 - ... or bounce it back to devs.
- Shipping code causes frictions and delays.



Devs vs Ops, after Docker

- Drop a container image or a Compose file.
- Ops can always run that container image.
- Ops can always run that Compose file.
- Ops still have to adapt to prod environment, but at least they have a reference point.
- Ops have tools allowing to use the same image in dev and prod.
- Devs can be empowered to make releases themselves more easily.



History of containers ... and Docker

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

History of containers ... and Docker

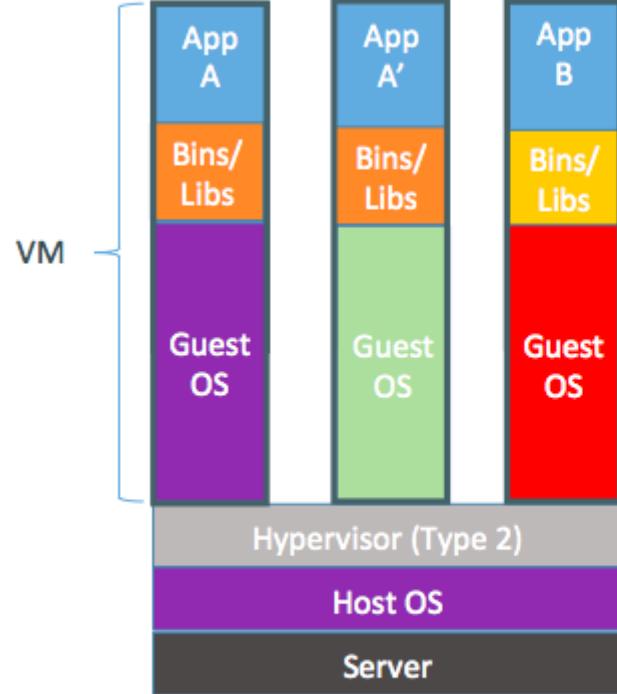
First experimentations

- IBM VM/370 (1972)
- Linux VServers (2001)
- Solaris Containers (2004)
- FreeBSD jails (1999-2000)

Containers have been around for a *very long time* indeed.

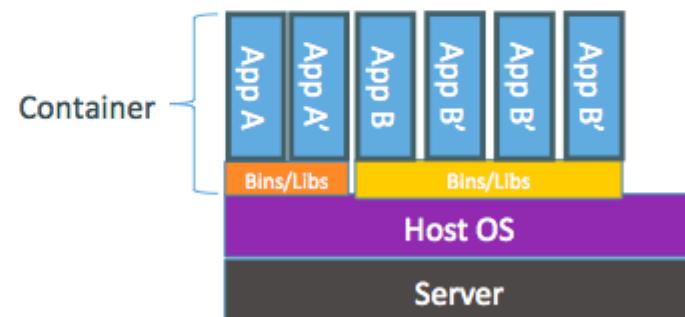
(See this excellent blog post by [Serge Hallyn](#) for more historic details.)

The VPS age (until 2007-2008)



Containers are isolated,
but share OS kernel and, where
appropriate, bins/libraries

...result is significantly faster deployment,
much less overhead, easier migration,
faster restart



Containers = cheaper than VMs

- Users: hosting providers.
- Highly specialized audience with strong ops culture.

The PAAS period (2008-2013)

[Features](#) | [Blog](#) | [About](#) | [Login](#)



Instantly Live

When you create or import an app with Heroku, it's already live on the web. No configuration or deployment necessary.

[Watch Demo ▶](#)



Create and Edit Online

Edit all of your code and data right in your browser. You can access it anywhere and there's nothing to install.

[Watch Demo ▶](#)



Share and Collaborate

Make your app public to the web or private to users you specify. Add collaborators who can edit the app with you.

[Watch Demo ▶](#)



Import & Export

Easily import and export your app's code, schema, and data at any time with just one click.



[Sign up](#) for our limited beta and we'll send you an invitation as soon as possible.

Containers = easier than VMs

- I can't speak for Heroku, but containers were (one of) dotCloud's secret weapon
- dotCloud was operating a PaaS, using a custom container engine.
- This engine was based on OpenVZ (and later, LXC) and AUFS.
- It started (circa 2008) as a single Python script.
- By 2012, the engine had multiple (~10) Python components.
(and ~100 other micro-services!)
- End of 2012, dotCloud refactors this container engine.
- The codename for this project is "Docker."

First public release of Docker

- March 2013, PyCon, Santa Clara:
"Docker" is shown to a public audience for the first time.
- It is released with an open source license.
- Very positive reactions and feedback!
- The dotCloud team progressively shifts to Docker development.
- The same year, dotCloud changes name to Docker.
- In 2014, the PaaS activity is sold.

Docker early days (2013-2014)

First users of Docker

- PAAS builders (Flynn, Dokku, Tsuru, Deis...)
- PAAS users (those big enough to justify building their own)
- CI platforms
- developers, developers, developers

Positive feedback loop

- In 2013, the technology under containers (cgroups, namespaces, copy-on-write storage...) had many blind spots.
- The growing popularity of Docker and containers exposed many bugs.
- As a result, those bugs were fixed, resulting in better stability for containers.
- Any decent hosting/cloud provider can run containers today.
- Containers become a great tool to deploy/move workloads to/from on-prem/cloud.

Maturity (2015-2016)

Docker becomes an industry standard

- Docker reaches the symbolic 1.0 milestone.
- Existing systems like Mesos and Cloud Foundry add Docker support.
- Standardization around the OCI (Open Containers Initiative).
- Other container engines are developed.
- Creation of the CNCF (Cloud Native Computing Foundation).

Docker becomes a platform

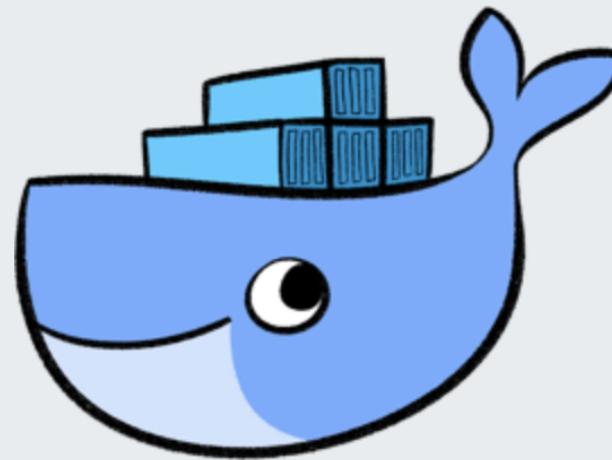
- The initial container engine is now known as "Docker Engine."
- Other tools are added:
 - Docker Compose (formerly "Fig")
 - Docker Machine
 - Docker Swarm
 - Kitematic
 - Docker Cloud (formerly "Tutum")
 - Docker Datacenter
 - etc.
- Docker Inc. launches commercial offers.



Our training environment

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Our training environment



Play with Docker

A simple, interactive and fun playground to learn Docker

Our training environment

- If you are attending at OSS Summit Vancouver:
 - we are going to use Play With Docker (PWD) to simplify usage/installation issues
 - if you already have Docker installed you can use a local installation
- If you are doing or re-doing this course on your own, you can:
 - install Docker locally (Docker for Mac or Docker for Windows on non-Linux)
 - install Docker on e.g. a cloud VM
 - use <http://www.play-with-docker.com/> to instantly get a training environment

What is Docker?

- "Installing Docker" really means "Installing the Docker Engine and CLI".
- The Docker Engine is a daemon (a service running in the background).
- This daemon manages containers, the same way that an hypervisor manages VMs.
- We interact with the Docker Engine by using the Docker CLI.
- The Docker CLI and the Docker Engine communicate through an API.
- There are many other programs, and many client libraries, to use that API.

Why don't we run Docker locally?

- We are going to download container images and distribution packages.
- This could put a bit of stress on the local WiFi and slow us down.
- Instead, we use a remote VM that has a good connectivity
- In some rare cases, installing Docker locally is challenging:
 - no administrator/root access (computer managed by strict corp IT)
 - 32-bit CPU or OS
 - old OS version (e.g. CentOS 6, OSX pre-Yosemite, Windows 7)
- It's better to spend time learning containers than fiddling with the installer!

Checking your PWD Setup

Once logged in, make sure that you can run a basic Docker command:

```
$ docker version
Client:
 Version:      18.03.0-ce
 API version:  1.37
 Go version:   go1.9.4
 Git commit:   0520e24
 Built:        Wed Mar 21 23:10:06 2018
 OS/Arch:      linux/amd64
 Experimental: false
 Orchestrator: swarm

Server:
 Engine:
  Version:      18.03.0-ce
  API version: 1.37 (minimum version 1.12)
  Go version:   go1.9.4
  Git commit:   0520e24
  Built:        Wed Mar 21 23:08:35 2018
  OS/Arch:      linux/amd64
  Experimental: false
```

If this doesn't work, raise your hand so that an instructor can assist you!



Our first containers

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Our first containers



Objectives

At the end of this lesson, you will have:

- Seen Docker in action.
- Started your first containers.

Hello World

In your Docker environment, just run the following command:

```
$ docker run busybox echo hello world  
hello world
```

(If your Docker install is brand new, you will also see a few extra lines, corresponding to the download of the `busybox` image.)

That was our first container!

- We used one of the smallest, simplest images available: `busybox`.
- `busybox` is typically used in embedded systems (phones, routers...)
- We ran a single process and echo'ed `hello world`.

A more useful container

Let's run a more exciting container:

```
$ docker run -it ubuntu  
root@04c0bb0a6c07:/#
```

- This is a brand new container.
- It runs a bare-bones, no-frills `ubuntu` system.
- `-it` is shorthand for `-i -t`.
 - `-i` tells Docker to connect us to the container's stdin.
 - `-t` tells Docker that we want a pseudo-terminal.

Do something in our container

Try to run `figlet` in our container.

```
root@04c0bb0a6c07:/# figlet hello  
bash: figlet: command not found
```

Alright, we need to install it.

Install a package in our container

We want `figlet`, so let's install it:

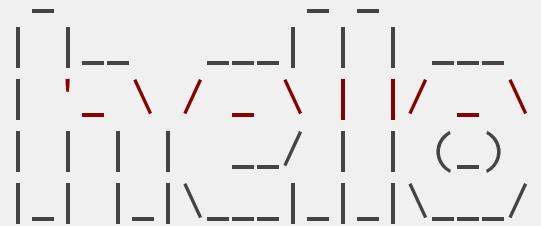
```
root@04c0bb0a6c07:/# apt-get update
...
Fetched 1514 kB in 14s (103 kB/s)
Reading package lists... Done
root@04c0bb0a6c07:/# apt-get install figlet
Reading package lists... Done
...
```

One minute later, `figlet` is installed!

Try to run our freshly installed program

The `figlet` program takes a message as parameter.

```
root@04c0bb0a6c07:/# figlet hello
```



Beautiful! 😍

Counting packages in the container

Let's check how many packages are installed there.

```
root@04c0bb0a6c07:/# dpkg -l | wc -l  
190
```

- `dpkg -l` lists the packages installed in our container
- `wc -l` counts them

How many packages do we have on our host?

Counting packages on the host

Exit the container by logging out of the shell, like you would usually do.

(E.g. with `^D` or `exit`)

```
root@04c0bb0a6c07:/# exit
```

Now, try to:

- run `dpkg -l | wc -l`. How many packages are installed?
- run `figlet`. Does that work?

Host and containers are independent things

- We ran an `ubuntu` container on an Linux/Windows/macOS host.
- They have different, independent packages.
- Installing something on the host doesn't expose it to the container.
- And vice-versa.
- Even if both the host and the container have the same Linux distro!
- We can run *any container* on *any host*.

(One exception: Windows containers cannot run on Linux machines; at least not yet.)

Where's our container?

- Our container is now in a *stopped* state.
- It still exists on disk, but all compute resources have been freed up.
- We will see later how to get back to that container.

Starting another container

What if we start a new container, and try to run `figlet` again?

```
$ docker run -it ubuntu
root@b13c164401fb:/# figlet
bash: figlet: command not found
```

- We started a *brand new container*.
- The basic Ubuntu image was used, and `figlet` is not here.
- We will see in the next chapters how to bake a custom image with `figlet`.



Background containers

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Background containers



Objectives

Our first containers were *interactive*.

We will now see how to:

- Run a non-interactive container.
- Run a container in the background.
- List running containers.
- Check the logs of a container.
- Stop a container.
- List stopped containers.

A non-interactive container

We will run a small custom container.

This container just displays the time every second.

```
$ docker run jpetazzo/clock
Fri Feb 20 00:28:53 UTC 2015
Fri Feb 20 00:28:54 UTC 2015
Fri Feb 20 00:28:55 UTC 2015
...
...
```

- This container will run forever.
- To stop it, press `^C`.
- Docker has automatically downloaded the image `jpetazzo/clock`.
- This image is a user image, created by `jpetazzo`.
- We will hear more about user images (and other types of images) later.

Run a container in the background

Containers can be started in the background, with the `-d` flag (daemon mode):

```
$ docker run -d jpetazzo/clock  
47d677dcfba4277c6cc68fcaa51f932b544cab1a187c853b7d0caf4e8debe5ad
```

- We don't see the output of the container.
- But don't worry: Docker collects that output and logs it!
- Docker gives us the ID of the container.

List running containers

How can we check that our container is still running?

With `docker ps`, just like the UNIX `ps` command, lists running processes.

```
$ docker ps
CONTAINER ID  IMAGE          ...  CREATED        STATUS        ...
47d677dcfba4  jpetazzo/clock  ...  2 minutes ago  Up 2 minutes  ...
```

Docker tells us:

- The (truncated) ID of our container.
- The image used to start the container.
- That our container has been running (`Up`) for a couple of minutes.
- Other information (COMMAND, PORTS, NAMES) that we will explain later.

Starting more containers

Let's start two more containers.

```
$ docker run -d jpetazzo/clock  
57ad9bdfc06bb4407c47220cf59ce21585dce9a1298d7a67488359aeaea8ae2a
```

```
$ docker run -d jpetazzo/clock  
068cc994ffd0190bbe025ba74e4c0771a5d8f14734af772ddee8dc1aaaf20567d
```

Check that `docker ps` correctly reports all 3 containers.

Viewing only the last container started

When many containers are already running, it can be useful to see only the last container that was started.

This can be achieved with the `-l` ("Last") flag:

```
$ docker ps -l
CONTAINER ID  IMAGE          ...  CREATED        STATUS      ...
068cc994ffd0  jpetazzo/clock  ...  2 minutes ago  Up 2 minutes  ...
```

View only the IDs of the containers

Many Docker commands will work on container IDs: docker stop, docker rm...

If we want to list only the IDs of our containers (without the other columns or the header line), we can use the -q ("Quiet", "Quick") flag:

```
$ docker ps -q  
068cc994ffd0  
57ad9bdfc06b  
47d677dcfba4
```

Combining flags

We can combine `-l` and `-q` to see only the ID of the last container started:

```
$ docker ps -lq  
068cc994ffd0
```

At a first glance, it looks like this would be particularly useful in scripts.

However, if we want to start a container and get its ID in a reliable way, it is better to use `docker run -d`, which we will cover in a bit.

View the logs of a container

We told you that Docker was logging the container output.

Let's see that now.

```
$ docker logs 068
Fri Feb 20 00:39:52 UTC 2015
Fri Feb 20 00:39:53 UTC 2015
...
```

- We specified a *prefix* of the full container ID.
- You can, of course, specify the full ID.
- The `logs` command will output the *entire* logs of the container.
(Sometimes, that will be too much. Let's see how to address that.)

View only the tail of the logs

To avoid being spammed with eleventy pages of output, we can use the `--tail` option:

```
$ docker logs --tail 3 068
Fri Feb 20 00:55:35 UTC 2015
Fri Feb 20 00:55:36 UTC 2015
Fri Feb 20 00:55:37 UTC 2015
```

- The parameter is the number of lines that we want to see.

Follow the logs in real time

Just like with the standard UNIX command `tail -f`, we can follow the logs of our container:

```
$ docker logs --tail 1 --follow 068
Fri Feb 20 00:57:12 UTC 2015
Fri Feb 20 00:57:13 UTC 2015
^C
```

- This will display the last line in the log file.
- Then, it will continue to display the logs in real time.
- Use `^C` to exit.

Stop our container

There are two ways we can terminate our detached container.

- Killing it using the `docker kill` command.
- Stopping it using the `docker stop` command.

The first one stops the container immediately, by using the `KILL` signal.

The second one is more graceful. It sends a `TERM` signal, and after 10 seconds, if the container has not stopped, it sends `KILL`.

Reminder: the `KILL` signal cannot be intercepted, and will forcibly terminate the container.

Stopping our containers

Let's stop one of those containers:

```
$ docker stop 47d6  
47d6
```

This will take 10 seconds:

- Docker sends the TERM signal;
- the container doesn't react to this signal (it's a simple Shell script with no special signal handling);
- 10 seconds later, since the container is still running, Docker sends the KILL signal;
- this terminates the container.

Killing the remaining containers

Let's be less patient with the two other containers:

```
$ docker kill 068 57ad  
068  
57ad
```

The `stop` and `kill` commands can take multiple container IDs.

Those containers will be terminated immediately (without the 10 seconds delay).

Let's check that our containers don't show up anymore:

```
$ docker ps
```

List stopped containers

We can also see stopped containers, with the `-a` (`--all`) option.

```
$ docker ps -a
CONTAINER ID  IMAGE      ...  CREATED      STATUS
068cc994ffd0  jpetazzo/clock  ...  21 min. ago  Exited (137) 3 min. ago
57ad9bdfc06b  jpetazzo/clock  ...  21 min. ago  Exited (137) 3 min. ago
47d677dcfba4  jpetazzo/clock  ...  23 min. ago  Exited (137) 3 min. ago
5c1dfd4d81f1  jpetazzo/clock  ...  40 min. ago  Exited (0) 40 min. ago
b13c164401fb  ubuntu      ...  55 min. ago  Exited (130) 53 min. ago
```



Restarting and attaching to containers

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Restarting and attaching to containers

We have started containers in the foreground, and in the background.

In this chapter, we will see how to:

- Put a container in the background.
- Attach to a background container to bring it to the foreground.
- Restart a stopped container.

Background and foreground

The distinction between foreground and background containers is arbitrary.

From Docker's point of view, all containers are the same.

All containers run the same way, whether there is a client attached to them or not.

It is always possible to detach from a container, and to reattach to a container.

Analogy: attaching to a container is like plugging a keyboard and screen to a physical server.

Detaching from a container

- If you have started an *interactive* container (with option `-it`), you can detach from it.
- The "detach" sequence is `^P^Q`.
- Otherwise you can detach by killing the Docker client.

(But not by hitting `^C`, as this would deliver `SIGINT` to the container.)

What does `-it` stand for?

- `-t` means "allocate a terminal."
- `-i` means "connect stdin to the terminal."



Specifying a custom detach sequence

- You don't like `^P^Q`? No problem!
- You can change the sequence with `docker run --detach-keys`.
- This can also be passed as a global option to the engine.

Start a container with a custom detach command:

```
$ docker run -ti --detach-keys ctrl-x,x jpetazzo/clock
```

Detach by hitting `^X x`. (This is `ctrl-x` then `x`, not `ctrl-x` twice!)

Check that our container is still running:

```
$ docker ps -l
```



Attaching to a container

You can attach to a container:

```
$ docker attach <containerID>
```

- The container must be running.
- There *can* be multiple clients attached to the same container.
- If you don't specify `--detach-keys` when attaching, it defaults back to `^P^Q`.

Try it on our previous container:

```
$ docker attach $(docker ps -lq)
```

Check that `^X x` doesn't work, but `^P ^Q` does.

Detaching from non-interactive containers

- **Warning:** if the container was started without `-it`...
 - You won't be able to detach with `^P^Q`.
 - If you hit `^C`, the signal will be proxied to the container.
- Remember: you can always detach by killing the Docker client.

Checking container output

- Use `docker attach` if you intend to send input to the container.
- If you just want to see the output of a container, use `docker logs`.

```
$ docker logs --tail 1 --follow <containerID>
```

Restarting a container

When a container has exited, it is in stopped state.

It can then be restarted with the `start` command.

```
$ docker start <yourContainerID>
```

The container will be restarted using the same options you launched it with.

You can re-attach to it if you want to interact with it:

```
$ docker attach <yourContainerID>
```

Use `docker ps -a` to identify the container ID of a previous `jpetazzo/clock` container, and try those commands.

Attaching to a REPL

- REPL = Read Eval Print Loop
- Shells, interpreters, TUI ...
- Symptom: you `docker attach`, and see nothing
- The REPL doesn't know that you just attached, and doesn't print anything
- Try hitting `^L` or `Enter`



SIGWINCH

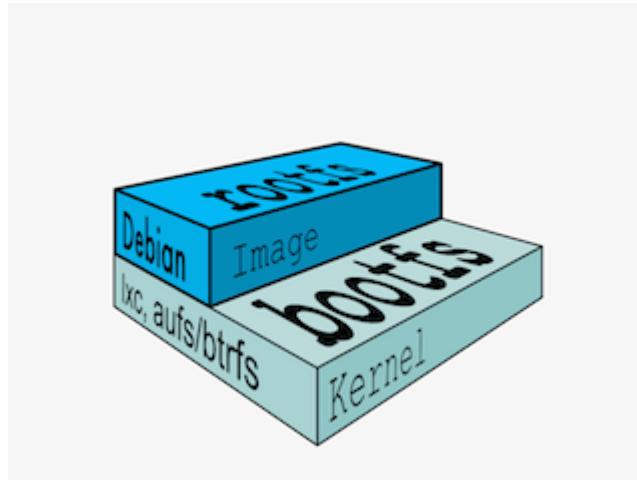
- When you `docker attach`, the Docker Engine sends SIGWINCH signals to the container.
- SIGWINCH = WINdow CHange; indicates a change in window size.
- This will cause some CLI and TUI programs to redraw the screen.
- But not all of them.



Understanding Docker images

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Understanding Docker images



Objectives

In this section, we will explain:

- What is an image.
- What is a layer.
- The various image namespaces.
- How to search and download images.
- Image tags and when to use them.

What is an image?

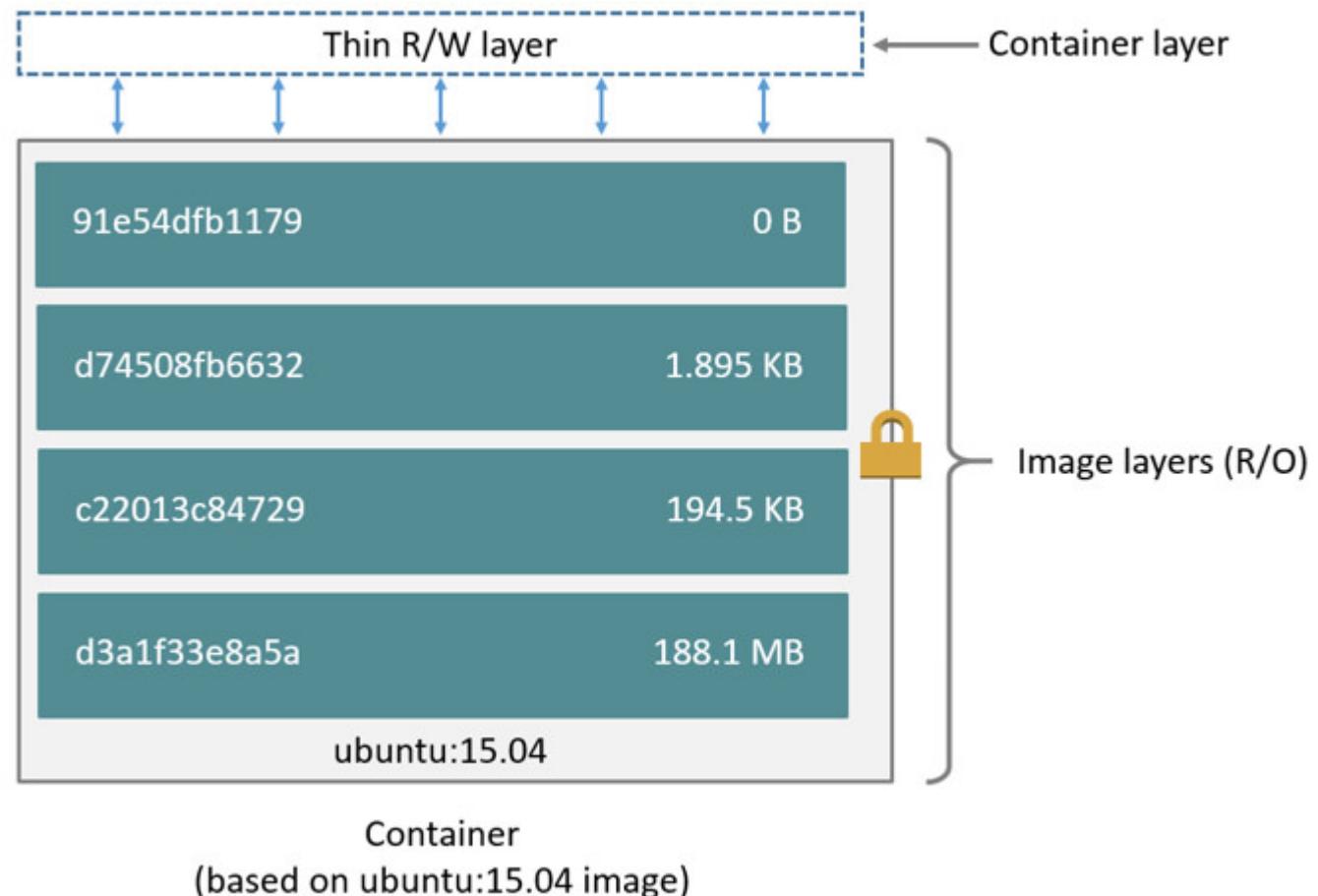
- Image = files + metadata
- These files form the root filesystem of our container.
- The metadata can indicate a number of things, e.g.:
 - the author of the image
 - the command to execute in the container when starting it
 - environment variables to be set
 - etc.
- Images are made of *layers*, conceptually stacked on top of each other.
- Each layer can add, change, and remove files and/or metadata.
- Images can share layers to optimize disk usage, transfer times, and memory use.

Example for a Java webapp

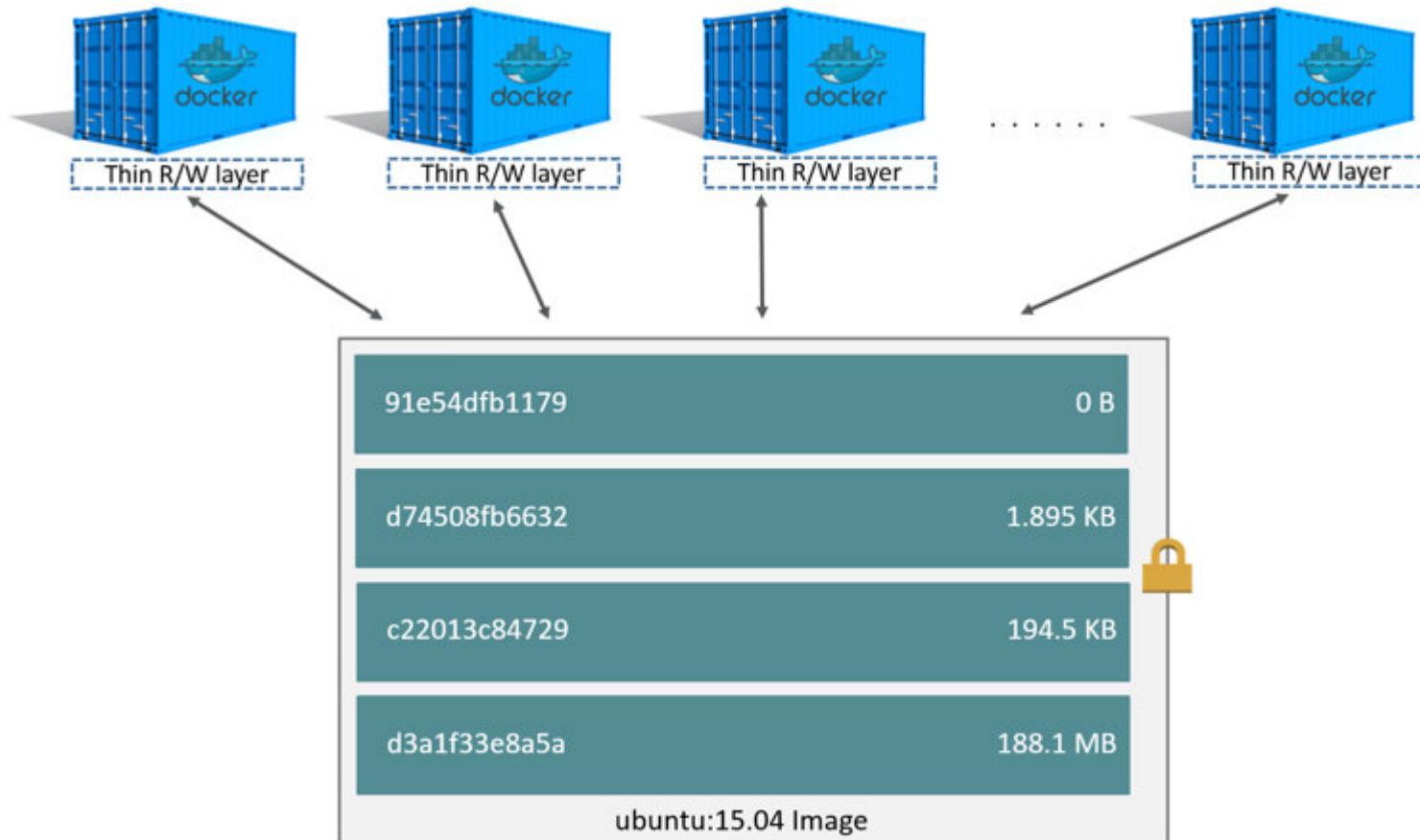
Each of the following items will correspond to one layer:

- CentOS base layer
- Packages and configuration files added by our local IT
- JRE
- Tomcat
- Our application's dependencies
- Our application code and assets
- Our application configuration

The read-write layer



Multiple containers sharing the same image



Differences between containers and images

- An image is a read-only filesystem.
- A container is an encapsulated set of processes running in a read-write copy of that filesystem.
- To optimize container boot time, *copy-on-write* is used instead of regular copy.
- `docker run` starts a container from a given image.

Comparison with object-oriented programming

- Images are conceptually similar to *classes*.
- Layers are conceptually similar to *inheritance*.
- Containers are conceptually similar to *instances*.

Wait a minute...

If an image is read-only, how do we change it?

- We don't.
- We create a new container from that image.
- Then we make changes to that container.
- When we are satisfied with those changes, we transform them into a new layer.
- A new image is created by stacking the new layer on top of the old image.

A chicken-and-egg problem

- The only way to create an image is by "freezing" a container.
- The only way to create a container is by instantiating an image.
- Help!

Creating the first images

There is a special empty image called `scratch`.

- It allows to *build from scratch*.

The `docker import` command loads a tarball into Docker.

- The imported tarball becomes a standalone image.
- That new image has a single layer.

Note: you will probably never have to do this yourself.

Creating other images

`docker commit`

- Saves all the changes made to a container into a new layer.
- Creates a new image (effectively a copy of the container).

`docker build` (**used 99% of the time**)

- Performs a repeatable build sequence.
- This is the preferred method!

We will explain both methods in a moment.

Images namespaces

There are three namespaces:

- Official images

e.g. `ubuntu`, `busybox` ...

- User (and organizations) images

e.g. `jpetazzo/clock`

- Self-hosted images

e.g. `registry.example.com:5000/my-private/image`

Let's explain each of them.

Root namespace

The root namespace is for official images. They are put there by Docker Inc., but they are generally authored and maintained by third parties.

Those images include:

- Small, "swiss-army-knife" images like busybox.
- Distro images to be used as bases for your builds, like ubuntu, fedora...
- Ready-to-use components and services, like redis, postgresql...
- Over 130 at this point!

User namespace

The user namespace holds images for Docker Hub users and organizations.

For example:

```
jpetazzo/clock
```

The Docker Hub user is:

```
jpetazzo
```

The image name is:

```
clock
```

Self-Hosted namespace

This namespace holds images which are not hosted on Docker Hub, but on third party registries.

They contain the hostname (or IP address), and optionally the port, of the registry server.

For example:

```
localhost:5000/wordpress
```

- `localhost:5000` is the host and port of the registry
- `wordpress` is the name of the image

How do you store and manage images?

Images can be stored:

- On your Docker host.
- In a Docker registry.

You can use the Docker client to download (pull) or upload (push) images.

To be more accurate: you can use the Docker client to tell a Docker Engine to push and pull images to and from a registry.

Showing current images

Let's look at what images are on our host now.

\$ docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
fedora	latest	ddd5c9c1d0f2	3 days ago	204.7 MB
centos	latest	d0e7f81ca65c	3 days ago	196.6 MB
ubuntu	latest	07c86167cdc4	4 days ago	188 MB
redis	latest	4f5f397d4b7c	5 days ago	177.6 MB
postgres	latest	afe2b5e1859b	5 days ago	264.5 MB
alpine	latest	70c557e50ed6	5 days ago	4.798 MB
debian	latest	f50f9524513f	6 days ago	125.1 MB
busybox	latest	3240943c9ea3	2 weeks ago	1.114 MB
training/namer	latest	902673acc741	9 months ago	289.3 MB
jpetazzo/clock	latest	12068b93616f	12 months ago	2.433 MB

Searching for images

We cannot list *all* images on a remote registry, but we can search for a specific keyword:

\$ docker search marathon				
NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
mesosphere/marathon	A cluster-wide init and co...	105	[OK]	
mesoscloud/marathon	Marathon	31	[OK]	
mesosphere/marathon-lb	Script to update haproxy b...	22	[OK]	
tobilg/mongodb-marathon	A Docker image to start a ...	4	[OK]	

- "Stars" indicate the popularity of the image.
- "Official" images are those in the root namespace.
- "Automated" images are built automatically by the Docker Hub.
(This means that their build recipe is always available.)

Downloading images

There are two ways to download images.

- Explicitly, with `docker pull`.
- Implicitly, when executing `docker run` and the image is not found locally.

Pulling an image

```
$ docker pull debian:jessie
Pulling repository debian
b164861940b8: Download complete
b164861940b8: Pulling image (jessie) from debian
d1881793a057: Download complete
```

- As seen previously, images are made up of layers.
- Docker has downloaded all the necessary layers.
- In this example, `:jessie` indicates which exact version of Debian we would like.

It is a *version tag*.

Image and tags

- Images can have tags.
- Tags define image versions or variants.
- `docker pull ubuntu` will refer to `ubuntu:latest`.
- The `:latest` tag is generally updated often.

When to (not) use tags

Don't specify tags:

- When doing rapid testing and prototyping.
- When experimenting.
- When you want the latest version.

Do specify tags:

- When recording a procedure into a script.
- When going to production.
- To ensure that the same version will be used everywhere.
- To ensure repeatability later.

Section summary

We've learned how to:

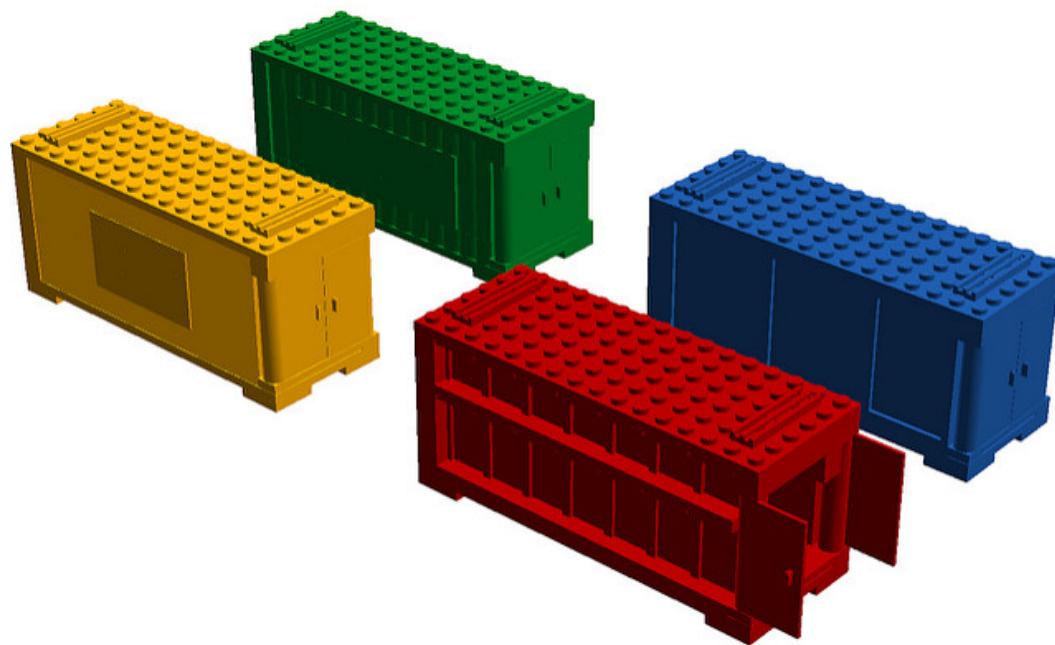
- Understand images and layers.
- Understand Docker image namespacing.
- Search and download images.



Building Docker images with a Dockerfile

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Building Docker images with a Dockerfile



Objectives

We will build a container image automatically, with a `Dockerfile`.

At the end of this lesson, you will be able to:

- Write a `Dockerfile`.
- Build an image from a `Dockerfile`.

Dockerfile overview

- A Dockerfile is a build recipe for a Docker image.
- It contains a series of instructions telling Docker how an image is constructed.
- The docker build command builds an image from a Dockerfile.

Writing our first Dockerfile

Our Dockerfile must be in a **new, empty directory**.

1. Create a directory to hold our **Dockerfile**.

```
$ mkdir myimage
```

1. Create a **Dockerfile** inside this directory.

```
$ cd myimage  
$ vim Dockerfile
```

Of course, you can use any other editor of your choice.

Type this into our Dockerfile...

```
FROM ubuntu
RUN apt-get update
RUN apt-get install figlet
```

- `FROM` indicates the base image for our build.
- Each `RUN` line will be executed by Docker during the build.
- Our `RUN` commands **must be non-interactive**.
(No input can be provided to Docker during the build.)
- In many cases, we will add the `-y` flag to `apt-get`.

Build it!

Save our file, then execute:

```
$ docker build -t figlet .
```

- `-t` indicates the tag to apply to the image.
- `.` indicates the location of the *build context*.

We will talk more about the build context later.

To keep things simple for now: this is the directory where our Dockerfile is located.

What happens when we build the image?

The output of `docker build` looks like this:

```
docker build -t figlet .
Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM ubuntu
--> f975c5035748
Step 2/3 : RUN apt-get update
--> Running in e01b294dbffd
(...output of the RUN command...)
Removing intermediate container e01b294dbffd
--> eb8d9b561b37
Step 3/3 : RUN apt-get install figlet
--> Running in c29230d70f9b
(...output of the RUN command...)
Removing intermediate container c29230d70f9b
--> 0dfd7a253f21
Successfully built 0dfd7a253f21
Successfully tagged figlet:latest
```

- The output of the `RUN` commands has been omitted.
- Let's explain what this output means.

Sending the build context to Docker

```
Sending build context to Docker daemon 2.048 kB
```

- The build context is the `.` directory given to `docker build`.
- It is sent (as an archive) by the Docker client to the Docker daemon.
- This allows to use a remote machine to build using local files.
- Be careful (or patient) if that directory is big and your link is slow.

Executing each step

```
Step 2/3 : RUN apt-get update
---> Running in e01b294dbffd
(...output of the RUN command...)
Removing intermediate container e01b294dbffd
---> eb8d9b561b37
```

- A container (`e01b294dbffd`) is created from the base image.
- The `RUN` command is executed in this container.
- The container is committed into an image (`eb8d9b561b37`).
- The build container (`e01b294dbffd`) is removed.
- The output of this step will be the base image for the next one.

The caching system

If you run the same build again, it will be instantaneous. Why?

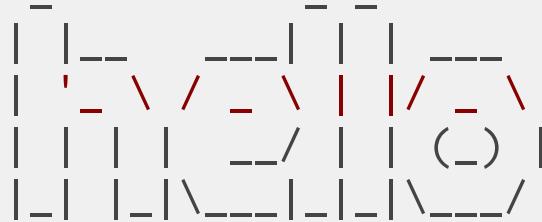
- After each build step, Docker takes a snapshot of the resulting image.
- Before executing a step, Docker checks if it has already built the same sequence.
- Docker uses the exact strings defined in your Dockerfile, so:
 - `RUN apt-get install figlet cowsay`
is different from
`RUN apt-get install cowsay figlet`
 - `RUN apt-get update` is not re-executed when the mirrors are updated

You can force a rebuild with `docker build --no-cache . . .`

Running the image

The resulting image is not different from the one produced manually.

```
$ docker run -ti figlet
root@91f3c974c9a1:/# figlet hello
```



Yay! 🎉

Using image and viewing history

The `history` command lists all the layers composing an image.

For each layer, it shows its creation time, size, and creation command.

When an image was built with a Dockerfile, each layer corresponds to a line of the Dockerfile.

```
$ docker history figlet
IMAGE          CREATED      CREATED BY               SIZE
f9e8f1642759  About an hour ago /bin/sh -c apt-get install fi  1.627 MB
7257c37726a1  About an hour ago /bin/sh -c apt-get update   21.58 MB
07c86167cdc4  4 days ago    /bin/sh -c #(nop) CMD ["/bin   0 B
<missing>      4 days ago    /bin/sh -c sed -i 's/^#\s*\(|  1.895 kB
<missing>      4 days ago    /bin/sh -c echo '#!/bin/sh'  194.5 kB
<missing>      4 days ago    /bin/sh -c #(nop) ADD file:b  187.8 MB
```

Introducing JSON syntax

Most Dockerfile arguments can be passed in two forms:

- plain string:

```
RUN apt-get install figlet
```

- JSON list:

```
RUN ["apt-get", "install", "figlet"]
```

We are going to change our Dockerfile to see how it affects the resulting image.

Using JSON syntax in our Dockerfile

Let's change our Dockerfile as follows!

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
```

Then build the new Dockerfile.

```
$ docker build -t figlet .
```

JSON syntax vs string syntax

Compare the new history:

\$ docker history figlet			
IMAGE	CREATED	CREATED BY	SIZE
27954bb5faaf	10 seconds ago	apt-get install figlet	1.627 MB
7257c37726a1	About an hour ago	/bin/sh -c apt-get update	21.58 MB
07c86167cdc4	4 days ago	/bin/sh -c #(nop) CMD ["/bin	0 B
<missing>	4 days ago	/bin/sh -c sed -i 's/^#\s*\(1.895 kB
<missing>	4 days ago	/bin/sh -c echo '#!/bin/sh'	194.5 kB
<missing>	4 days ago	/bin/sh -c #(nop) ADD file:b	187.8 MB

- JSON syntax specifies an *exact* command to execute.
- String syntax specifies a command to be wrapped within `/bin/sh -c "..."`.

When to use JSON syntax and string syntax

- String syntax:
 - is easier to write
 - interpolates environment variables and other shell expressions
 - creates an extra process (`/bin/sh -c ...`) to parse the string
 - requires `/bin/sh` to exist in the container
- JSON syntax:
 - is harder to write (and read!)
 - passes all arguments without extra processing
 - doesn't create an extra process
 - doesn't require `/bin/sh` to exist in the container



CMD and ENTRYPOINT

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

CMD and ENTRYPPOINT



Objectives

In this lesson, we will learn about two important Dockerfile commands:

`CMD` and `ENTRYPOINT`.

These commands allow us to set the default command to run in a container.

Defining a default command

When people run our container, we want to greet them with a nice hello message, and using a custom font.

For that, we will execute:

```
figlet -f script hello
```

- `-f script` tells figlet to use a fancy font.
- `hello` is the message that we want it to display.

Adding `CMD` to our Dockerfile

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
CMD figlet -f script hello
```

- `CMD` defines a default command to run when none is given.
- It can appear at any point in the file.
- Each `CMD` will replace and override the previous one.
- As a result, while you can have multiple `CMD` lines, it is useless.

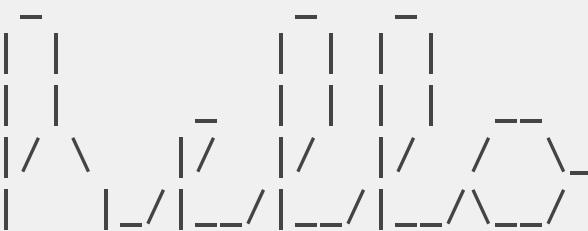
Build and test our image

Let's build it:

```
$ docker build -t figlet .
...
Successfully built 042dff3b4a8d
Successfully tagged figlet:latest
```

And run it:

```
$ docker run figlet
```



Overriding CMD

If we want to get a shell into our container (instead of running `figlet`), we just have to specify a different program to run:

```
$ docker run -it figlet bash  
root@7ac86a641116:/#
```

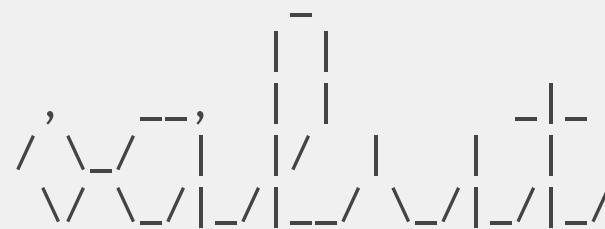
- We specified `bash`.
- It replaced the value of `CMD`.

Using ENTRYPPOINT

We want to be able to specify a different message on the command line, while retaining `figlet` and some default parameters.

In other words, we would like to be able to do this:

```
$ docker run figlet salut
```

A large, stylized letter 'S' character is displayed. It is constructed from a variety of line segments, including verticals, horizontals, and diagonal lines, some of which are enclosed in brackets. The character is oriented vertically and has a complex, blocky appearance.

We will use the `ENTRYPOINT` verb in Dockerfile.

Adding `ENTRYPOINT` to our Dockerfile

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
ENTRYPOINT ["figlet", "-f", "script"]
```

- `ENTRYPOINT` defines a base command (and its parameters) for the container.
- The command line arguments are appended to those parameters.
- Like `CMD`, `ENTRYPOINT` can appear anywhere, and replaces the previous value.

Why did we use JSON syntax for our `ENTRYPOINT`?

Implications of JSON vs string syntax

- When CMD or ENTRYPOINT use string syntax, they get wrapped in `sh -c`.
- To avoid this wrapping, we can use JSON syntax.

What if we used ENTRYPOINT with string syntax?

```
$ docker run figlet salut
```

This would run the following command in the `figlet` image:

```
sh -c "figlet -f script" salut
```

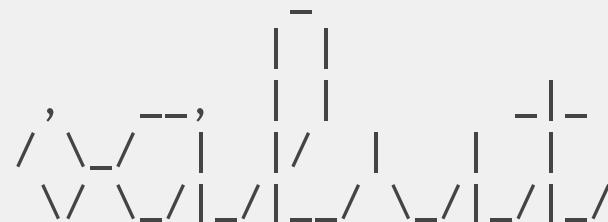
Build and test our image

Let's build it:

```
$ docker build -t figlet .
...
Successfully built 36f588918d73
Successfully tagged figlet:latest
```

And run it:

```
$ docker run figlet salut
```



Using `CMD` and `ENTRYPOINT` together

What if we want to define a default message for our container?

Then we will use `ENTRYPOINT` and `CMD` together.

- `ENTRYPOINT` will define the base command for our container.
- `CMD` will define the default parameter(s) for this command.
- They *both* have to use JSON syntax.

CMD and ENTRYPOINT together

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
ENTRYPOINT ["figlet", "-f", "script"]
CMD ["hello world"]
```

- ENTRYPOINT defines a base command (and its parameters) for the container.
- If we don't specify extra command-line arguments when starting the container, the value of CMD is appended.
- Otherwise, our extra command-line arguments are used instead of CMD.

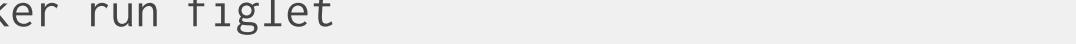
Build and test our image

Let's build it:

```
$ docker build -t figlet .  
...  
Successfully built 6e0b6a048a07  
Successfully tagged figlet:latest
```

Run it without parameters:

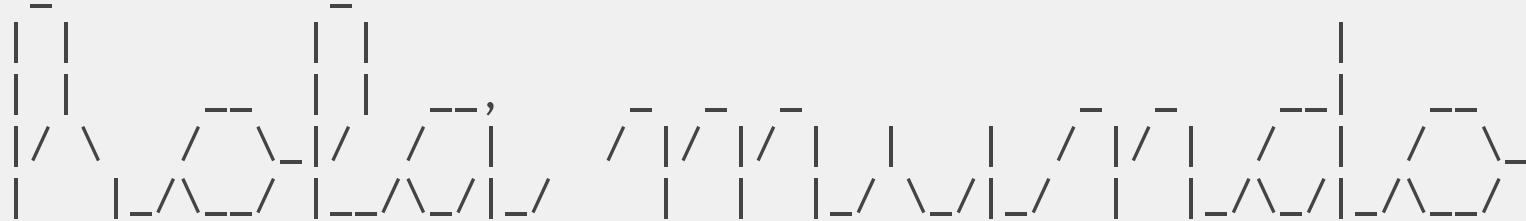
```
$ docker run figlet
```



Overriding the image default parameters

Now let's pass extra arguments to the image.

```
$ docker run figlet hola mundo
```



We overrode `CMD` but still used `ENTRYPOINT`.

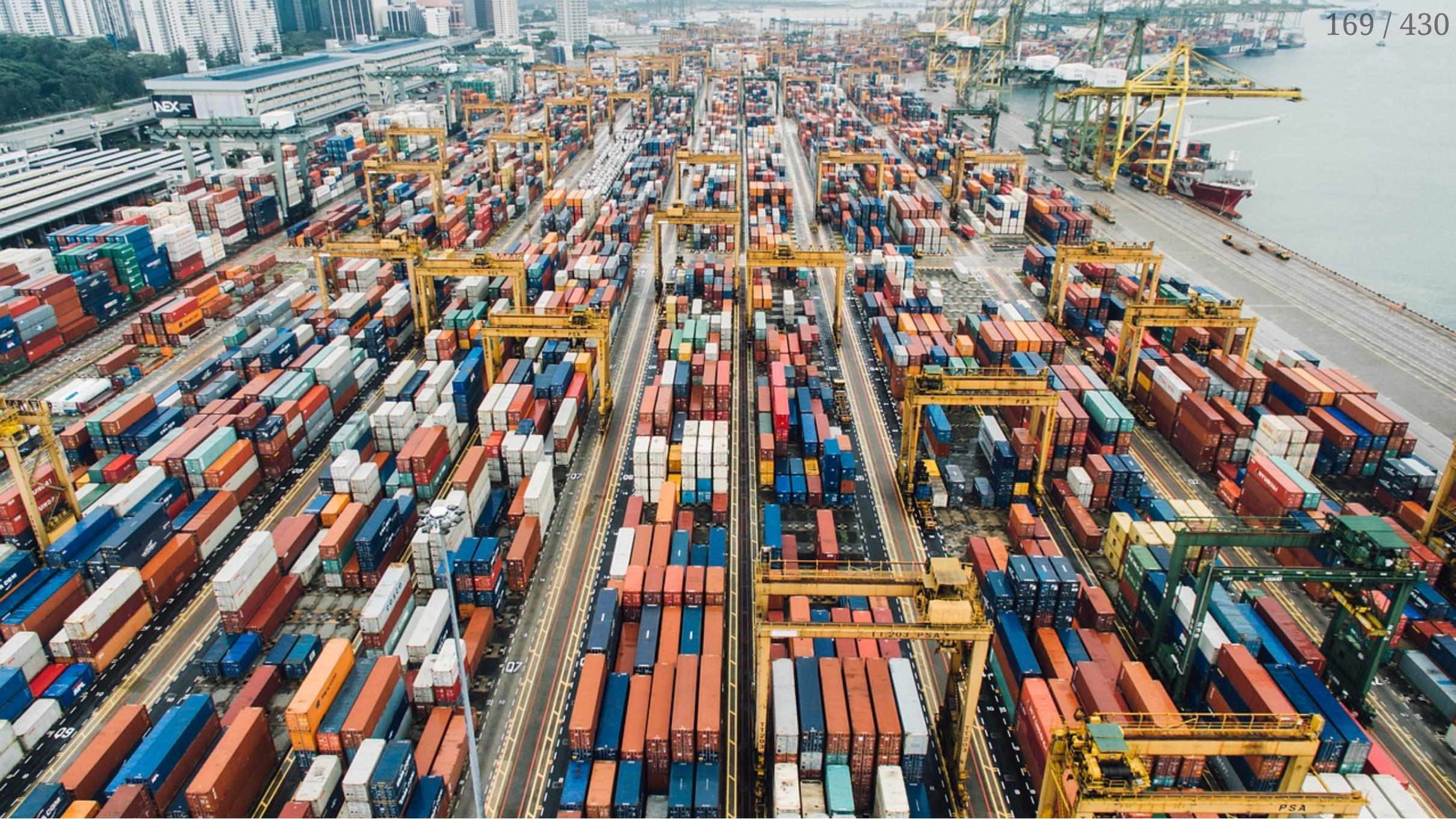
Overriding ENTRYPPOINT

What if we want to run a shell in our container?

We cannot just do `docker run figlet bash` because that would just tell figlet to display the word "bash."

We use the `--entrypoint` parameter:

```
$ docker run -it --entrypoint bash figlet
root@6027e44e2955:/#
```



Copying files during the build

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Copying files during the build



Objectives

So far, we have installed things in our container images by downloading packages.

We can also copy files from the *build context* to the container that we are building.

Remember: the *build context* is the directory containing the Dockerfile.

In this chapter, we will learn a new Dockerfile keyword: `COPY`.

Build some C code

We want to build a container that compiles a basic "Hello world" program in C.

Here is the program, `hello.c`:

```
int main () {
    puts("Hello, world!");
    return 0;
}
```

Let's create a new directory, and put this file in there.

Then we will write the Dockerfile.

The Dockerfile

On Debian and Ubuntu, the package `build-essential` will get us a compiler.

When installing it, don't forget to specify the `-y` flag, otherwise the build will fail (since the build cannot be interactive).

Then we will use `COPY` to place the source file into the container.

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y build-essential
COPY hello.c /
RUN make hello
CMD /hello
```

Create this Dockerfile.

Testing our C program

- Create `hello.c` and `Dockerfile` in the same directory.
- Run `docker build -t hello .` in this directory.
- Run `docker run hello`, you should see `Hello, world!`.

Success!

COPY and the build cache

- Run the build again.
- Now, modify `hello.c` and run the build again.
- Docker can cache steps involving `COPY`.
- Those steps will not be executed again if the files haven't been changed.

Details

- You can `COPY` whole directories recursively.
- Older Dockerfiles also have the `ADD` instruction.
It is similar but can automatically extract archives.
- If we really wanted to compile C code in a container, we would:
 - Place it in a different directory, with the `WORKDIR` instruction.
 - Even better, use the `gcc` official image.



Reducing image size

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Reducing image size

- In the previous example, our final image contained:
 - our `hello` program
 - its source code
 - the compiler
- Only the first one is strictly necessary.
- We are going to see how to obtain an image without the superfluous components.

Can't we remove superfluous files with RUN?

What happens if we do one of the following commands?

- RUN rm -rf ...
- RUN apt-get remove ...
- RUN make clean ...

Can't we remove superfluous files with RUN?

What happens if we do one of the following commands?

- RUN rm -rf ...
- RUN apt-get remove ...
- RUN make clean ...

This adds a layer which removes a bunch of files.

But the previous layers (which added the files) still exist.

Removing files with an extra layer

When downloading an image, all the layers must be downloaded.

Dockerfile instruction	Layer size	Image size
FROM ubuntu	Size of base image	Size of base image
...	...	Sum of this layer + all previous ones
RUN apt-get install somepackage	Size of files added (e.g. a few MB)	Sum of this layer + all previous ones
...	...	Sum of this layer + all previous ones
RUN apt-get remove somepackage	Almost zero (just metadata)	Same as previous one

Therefore, `RUN rm` does not reduce the size of the image or free up disk space.

Removing unnecessary files

Various techniques are available to obtain smaller images:

- collapsing layers,
- adding binaries that are built outside of the Dockerfile,
- squashing the final image,
- multi-stage builds.

Let's review them quickly.

Collapsing layers

You will frequently see Dockerfiles like this:

```
FROM ubuntu
RUN apt-get update && apt-get install xxx && ... && apt-get remove xxx && ...
```

Or the (more readable) variant:

```
FROM ubuntu
RUN apt-get update \
&& apt-get install xxx \
&& ... \
&& apt-get remove xxx \
&& ...
```

This `RUN` command gives us a single layer.

The files that are added, then removed in the same layer, do not grow the layer size.

Collapsing layers: pros and cons

Pros:

- works on all versions of Docker
- doesn't require extra tools

Cons:

- not very readable
- some unnecessary files might still remain if the cleanup is not thorough
- that layer is expensive (slow to build)

Building binaries outside of the Dockerfile

This results in a Dockerfile looking like this:

```
FROM ubuntu
COPY xxx /usr/local/bin
```

Of course, this implies that the file `xxx` exists in the build context.

That file has to exist before you can run `docker build`.

For instance, it can:

- exist in the code repository,
- be created by another tool (script, Makefile...),
- be created by another container image and extracted from the image.

See for instance the [busybox official image](#) or this [older busybox image](#).

Building binaries outside: pros and cons

Pros:

- final image can be very small

Cons:

- requires an extra build tool
- we're back in dependency hell and "works on my machine"

Cons, if binary is added to code repository:

- breaks portability across different platforms
- grows repository size a lot if the binary is updated frequently

Squashing the final image

The idea is to transform the final image into a single-layer image.

This can be done in (at least) two ways.

- Activate experimental features and squash the final image:

```
docker image build --squash ...
```

- Export/import the final image.

```
docker build -t temp-image .
docker run --entrypoint true --name temp-container temp-image
docker export temp-container | docker import - final-image
docker rm temp-container
docker rmi temp-image
```

Squashing the image: pros and cons

Pros:

- single-layer images are smaller and faster to download
- removed files no longer take up storage and network resources

Cons:

- we still need to actively remove unnecessary files
- squash operation can take a lot of time (on big images)
- squash operation does not benefit from cache
(even if we change just a tiny file, the whole image needs to be re-squashed)

Multi-stage builds

Multi-stage builds allow us to have multiple *stages*.

Each stage is a separate image, and can copy files from previous stages.

We're going to see how they work in more detail.



Multi-stage builds

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Multi-stage builds

- At any point in our Dockerfile, we can add a new FROM line.
- This line starts a new stage of our build.
- Each stage can access the files of the previous stages with COPY --from=....
- When a build is tagged (with docker build -t ...), the last stage is tagged.
- Previous stages are not discarded: they will be used for caching, and can be referenced.

Multi-stage builds in practice

- Each stage is numbered, starting at `0`
- We can copy a file from a previous stage by indicating its number, e.g.:

```
COPY --from=0 /file/from/first/stage /location/in/current/stage
```

- We can also name stages, and reference these names:

```
FROM golang AS builder
RUN ...
FROM alpine
COPY --from=builder /go/bin/mylittlebinary /usr/local/bin/
```

Multi-stage builds for our C program

We will change our Dockerfile to:

- give a nickname to the first stage: `compiler`
- add a second stage using the same `ubuntu` base image
- add the `hello` binary to the second stage
- make sure that `CMD` is in the second stage

The resulting Dockerfile is on the next slide.

Multi-stage build Dockerfile

Here is the final Dockerfile:

```
FROM ubuntu AS compiler
RUN apt-get update
RUN apt-get install -y build-essential
COPY hello.c /
RUN make hello
FROM ubuntu
COPY --from=compiler /hello /hello
CMD /hello
```

Let's build it, and check that it works correctly:

```
docker build -t hellomultistage .
docker run hellomultistage
```

Comparing single/multi-stage build image sizes

List our images with `docker images`, and check the size of:

- the `ubuntu` base image,
- the single-stage `hello` image,
- the multi-stage `hellomultistage` image.

We can achieve even smaller images if we use smaller base images.

However, if we use common base images (e.g. if we standardize on `ubuntu`), these common images will be pulled only once per node, so they are virtually "free."



Publishing images to the Docker Hub

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Publishing images to the Docker Hub

We have built our first images.

We can now publish it to the Docker Hub!

You don't have to do the exercises in this section, because they require an account on the Docker Hub, and we don't want to force anyone to create one.

Note, however, that creating an account on the Docker Hub is free (and doesn't require a credit card), and hosting public images is free as well.

Logging into our Docker Hub account

- This can be done from the Docker CLI:

```
docker login
```

 When running Docker for Mac/Windows, or Docker on a Linux workstation, it can (and will when possible) integrate with your system's keyring to store your credentials securely. However, on most Linux servers, it will store your credentials in `~/.docker/config`.

Image tags and registry addresses

- Docker images tags are like Git tags and branches.
- They are like *bookmarks* pointing at a specific image ID.
- Tagging an image doesn't *rename* an image: it adds another tag.
- When pushing an image to a registry, the registry address is in the tag.

Example: `registry.example.net:5000/image`

- What about Docker Hub images?

Image tags and registry addresses

- Docker images tags are like Git tags and branches.
- They are like *bookmarks* pointing at a specific image ID.
- Tagging an image doesn't *rename* an image: it adds another tag.
- When pushing an image to a registry, the registry address is in the tag.

Example: `registry.example.net:5000/image`

- What about Docker Hub images?
- `jpetazzo/clock` is, in fact, `index.docker.io/jpetazzo/clock`
- `ubuntu` is, in fact, `library/ubuntu`, i.e. `index.docker.io/library/ubuntu`

Tagging an image to push it on the Hub

- Let's tag our `figlet` image (or any other to our liking):

```
docker tag figlet jpetazzo/figlet
```

- And push it to the Hub:

```
docker push jpetazzo/figlet
```

- That's it!

Tagging an image to push it on the Hub

- Let's tag our `figlet` image (or any other to our liking):

```
docker tag figlet jpetazzo/figlet
```

- And push it to the Hub:

```
docker push jpetazzo/figlet
```

- That's it!
- Anybody can now `docker run jpetazzo/figlet` anywhere.

The goodness of automated builds

- You can link a Docker Hub repository with a GitHub or BitBucket repository
- Each push to GitHub or BitBucket will trigger a build on Docker Hub
- If the build succeeds, the new image is available on Docker Hub
- You can map tags and branches between source and container images
- If you work with public repositories, this is free



Setting up an automated build

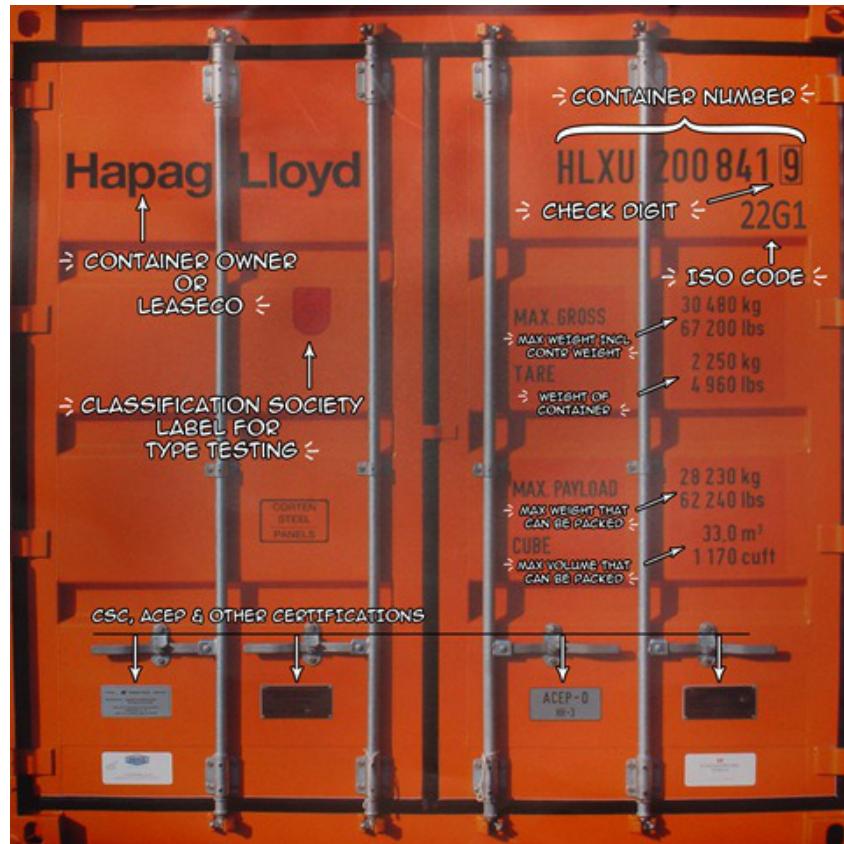
- We need a Dockerized repository!
- Let's go to <https://github.com/jpetazzo/trainingwheels> and fork it.
- Go to the Docker Hub (<https://hub.docker.com/>).
- Select "Create" in the top-right bar, and select "Create Automated Build."
- Connect your Docker Hub account to your GitHub account.
- Select your user and the repository that we just forked.
- Create.
- Then go to "Build Settings."
- Put `/www` in "Dockerfile Location" (or whichever directory the Dockerfile is in).
- Click "Trigger" to build the repository immediately (without waiting for a git push).
- Subsequent builds will happen automatically, thanks to GitHub hooks.



Naming and inspecting containers

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Naming and inspecting containers



Objectives

In this lesson, we will learn about an important Docker concept: container *naming*.

Naming allows us to:

- Reference easily a container.
- Ensure unicity of a specific container.

We will also see the `inspect` command, which gives a lot of details about a container.

Naming our containers

So far, we have referenced containers with their ID.

We have copy-pasted the ID, or used a shortened prefix.

But each container can also be referenced by its name.

If a container is named `thumbnail-worker`, I can do:

```
$ docker logs thumbnail-worker  
$ docker stop thumbnail-worker  
etc.
```

Default names

When we create a container, if we don't give a specific name, Docker will pick one for us.

It will be the concatenation of:

- A mood (furious, goofy, suspicious, boring...)
- The name of a famous inventor (tesla, darwin, wozniak...)

Examples: happy_curié, clever_hopper, jovial_lovelace ...

Specifying a name

You can set the name of the container when you create it.

```
$ docker run --name ticktock jpetazzo/clock
```

If you specify a name that already exists, Docker will refuse to create the container.

This lets us enforce unicity of a given resource.

Renaming containers

- You can rename containers with `docker rename`.
- This allows you to "free up" a name without destroying the associated container.

Inspecting a container

The `docker inspect` command will output a very detailed JSON map.

```
$ docker inspect <containerID>
[{
...
(many pages of JSON here)
...
}
```

There are multiple ways to consume that information.

Parsing JSON with the Shell

- You *could* grep and cut or awk the output of `docker inspect`.
- Please, don't.
- It's painful.
- If you really must parse JSON from the Shell, use JQ! (It's great.)

```
$ docker inspect <containerID> | jq .
```

- We will see a better solution which doesn't require extra tools.

Using --format

You can specify a format string, which will be parsed by Go's text/template package.

```
$ docker inspect --format '{{ json .Created }}' <containerID>
"2015-02-24T07:21:11.712240394Z"
```

- The generic syntax is to wrap the expression with double curly braces.
- The expression starts with a dot representing the JSON object.
- Then each field or member can be accessed in dotted notation syntax.
- The optional `json` keyword asks for valid JSON output.
(e.g. here it adds the surrounding double-quotes.)



Labels

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Labels

- Labels allow to attach arbitrary metadata to containers.
- Labels are key/value pairs.
- They are specified at container creation.
- You can query them with `docker inspect`.
- They can also be used as filters with some commands (e.g. `docker ps`).

Using labels

Let's create a few containers with a label `owner`.

```
docker run -d -l owner=alice nginx
docker run -d -l owner=bob nginx
docker run -d -l owner nginx
```

We didn't specify a value for the `owner` label in the last example.

This is equivalent to setting the value to be an empty string.

Querying labels

We can view the labels with `docker inspect`.

```
$ docker inspect $(docker ps -lq) | grep -A3 Labels
  "Labels": {
    "maintainer": "NGINX Docker Maintainers <docker-maint@nginx.com>",
    "owner": ""
  },
```

We can use the `--format` flag to list the value of a label.

```
$ docker inspect $(docker ps -q) --format 'OWNER={{.Config.Labels.owner}}'
```

Using labels to select containers

We can list containers having a specific label.

```
$ docker ps --filter label=owner
```

Or we can list containers having a specific label with a specific value.

```
$ docker ps --filter label=owner=alice
```

Use-cases for labels

- HTTP vhost of a web app or web service.

(The label is used to generate the configuration for NGINX, HAProxy, etc.)

- Backup schedule for a stateful service.

(The label is used by a cron job to determine if/when to backup container data.)

- Service ownership.

(To determine internal cross-billing, or who to page in case of outage.)

- etc.



Getting inside a container

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Getting inside a container



Objectives

On a traditional server or VM, we sometimes need to:

- log into the machine (with SSH or on the console),
- analyze the disks (by removing them or rebooting with a rescue system).

In this chapter, we will see how to do that with containers.

Getting a shell

Every once in a while, we want to log into a machine.

In an perfect world, this shouldn't be necessary.

- You need to install or update packages (and their configuration)?

Use configuration management. (e.g. Ansible, Chef, Puppet, Salt...)

- You need to view logs and metrics?

Collect and access them through a centralized platform.

In the real world, though ... we often need shell access!

Not getting a shell

Even without a perfect deployment system, we can do many operations without getting a shell.

- Installing packages can (and should) be done in the container image.
- Configuration can be done at the image level, or when the container starts.
- Dynamic configuration can be stored in a volume (shared with another container).
- Logs written to stdout are automatically collected by the Docker Engine.
- Other logs can be written to a shared volume.
- Process information and metrics are visible from the host.

Let's save logging, volumes ... for later, but let's have a look at process information!

Viewing container processes from the host

If you run Docker on Linux, container processes are visible on the host.

```
$ ps faux | less
```

- Scroll around the output of this command.
- You should see the `jpetazzo/clock` container.
- A containerized process is just like any other process on the host.
- We can use tools like `lsof`, `strace`, `gdb` ... To analyze them.



What's the difference between a container process and a host process?

- Each process (containerized or not) belongs to *namespaces* and *cgroups*.
- The namespaces and cgroups determine what a process can "see" and "do".
- Analogy: each process (containerized or not) runs with a specific UID (user ID).
- UID=0 is root, and has elevated privileges. Other UIDs are normal users.

We will give more details about namespaces and cgroups later.

Getting a shell in a running container

- Sometimes, we need to get a shell anyway.
- We *could* run some SSH server in the container ...
- But it is easier to use `docker exec`.

```
$ docker exec -ti ticktock sh
```

- This creates a new process (running `sh`) *inside* the container.
- This can also be done "manually" with the tool `nsenter`.

Caveats

- The tool that you want to run needs to exist in the container.
- Some tools (like `ip netns exec`) let you attach to *one* namespace at a time.
(This lets you e.g. setup network interfaces, even if you don't have `ifconfig` or `ip` in the container.)
- Most importantly: the container needs to be running.
- What if the container is stopped or crashed?

Getting a shell in a stopped container

- A stopped container is only *storage* (like a disk drive).
- We cannot SSH into a disk drive or USB stick!
- We need to connect the disk to a running machine.
- How does that translate into the container world?

Analyzing a stopped container

As an exercise, we are going to try to find out what's wrong with `jpetazzo/crashtest`.

```
docker run jpetazzo/crashtest
```

The container starts, but then stops immediately, without any output.

What would MacGyver™ do?

First, let's check the status of that container.

```
docker ps -l
```

Viewing filesystem changes

- We can use `docker diff` to see files that were added / changed / removed.

```
docker diff <container_id>
```

- The container ID was shown by `docker ps -l`.
- We can also see it with `docker ps -lq`.
- The output of `docker diff` shows some interesting log files!

Accessing files

- We can extract files with `docker cp`.

```
docker cp <container_id>:/var/log/nginx/error.log .
```

- Then we can look at that log file.

```
cat error.log
```

(The directory `/run/nginx` doesn't exist.)

Exploring a crashed container

- We can restart a container with `docker start ...`
- ... But it will probably crash again immediately!
- We cannot specify a different program to run with `docker start`
- But we can create a new image from the crashed container

```
docker commit <container_id> debugimage
```

- Then we can run a new container from that image, with a custom entrypoint

```
docker run -ti --entrypoint sh debugimage
```



Obtaining a complete dump

- We can also dump the entire filesystem of a container.
- This is done with `docker export`.
- It generates a tar archive.

```
docker export <container_id> | tar tv
```

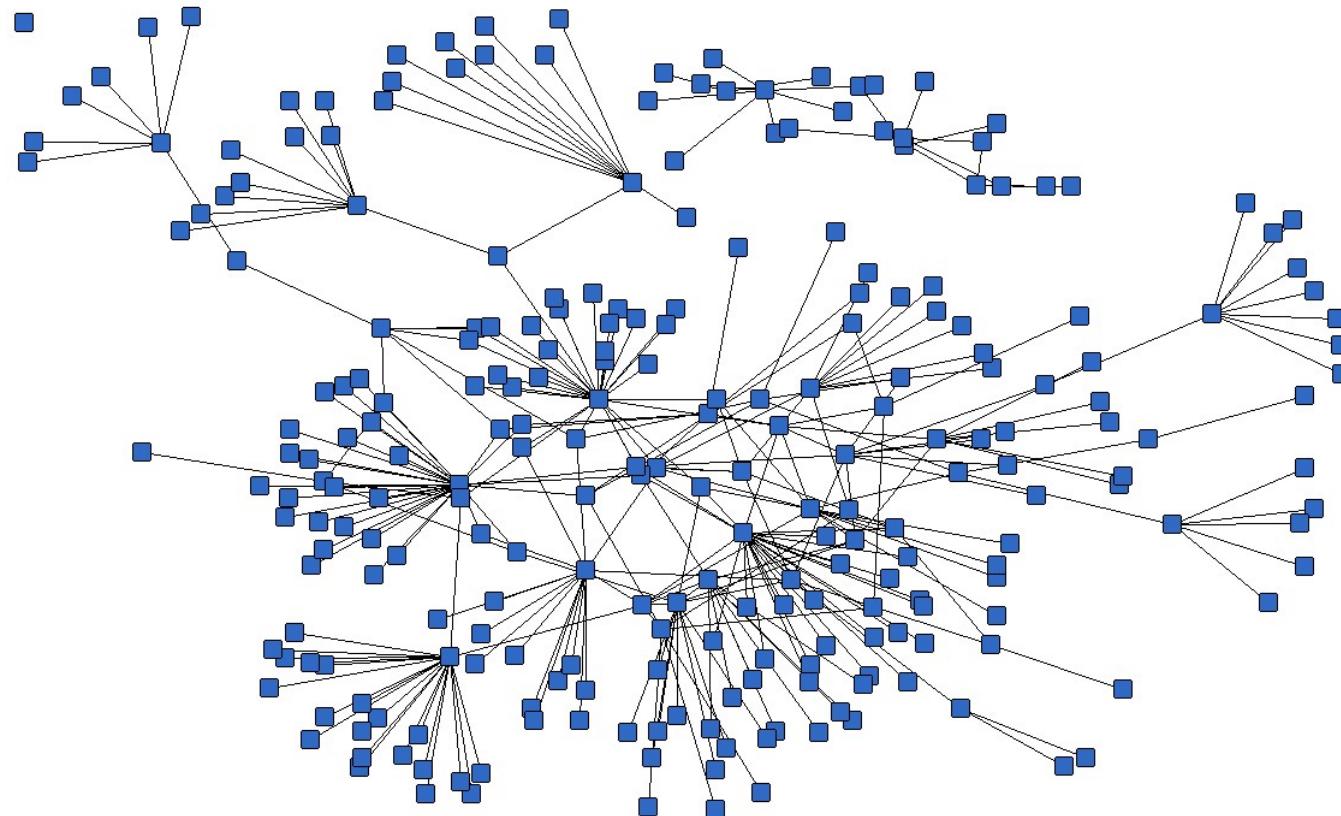
This will give a detailed listing of the content of the container.



Container networking basics

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Container networking basics



Objectives

We will now run network services (accepting requests) in containers.

At the end of this section, you will be able to:

- Run a network service in a container.
- Manipulate container networking basics.
- Find a container's IP address.

We will also explain the different network models used by Docker.

A simple, static web server

Run the Docker Hub image `nginx`, which contains a basic web server:

```
$ docker run -d -P nginx  
66b1ce719198711292c8f34f84a7b68c3876cf9f67015e752b94e189d35a204e
```

- Docker will download the image from the Docker Hub.
- `-d` tells Docker to run the image in the background.
- `-P` tells Docker to make this service reachable from other computers.
(`-P` is the short version of `--publish-all`.)

But, how do we connect to our web server now?

Finding our web server port

We will use `docker ps`:

```
$ docker ps
CONTAINER ID  IMAGE      ...  PORTS          ...
e40ffb406c9e  nginx      ...  0.0.0.0:32768->80/tcp  ...
```

- The web server is running on port 80 inside the container.
- This port is mapped to port 32768 on our Docker host.

We will explain the whys and hows of this port mapping.

But first, let's make sure that everything works properly.

Connecting to our web server (GUI)

Point your browser to the IP address of your Docker host, on the port shown by `docker ps` for container port 80.



Connecting to our web server (CLI)

You can also use `curl` directly from the Docker host.

Make sure to use the right port number if it is different from the example below:

```
$ curl localhost:32768
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

How does Docker know which port to map?

- There is metadata in the image telling "this image has something on port 80".
- We can see that metadata with `docker inspect`:

```
$ docker inspect --format '{{.Config.ExposedPorts}}' nginx
map[80/tcp:{}]
```

- This metadata was set in the Dockerfile, with the `EXPOSE` keyword.
- We can see that with `docker history`:

```
$ docker history nginx
IMAGE          CREATED          CREATED BY
7f70b30f2cc6  11 days ago      /bin/sh -c #(nop)  CMD ["nginx" "-g" ...
<missing>      11 days ago      /bin/sh -c #(nop)  STOPSIGNAL [SIGTERM]
<missing>      11 days ago      /bin/sh -c #(nop)  EXPOSE 80/tcp
```

Why are we mapping ports?

- We are out of IPv4 addresses.
- Containers cannot have public IPv4 addresses.
- They have private addresses.
- Services have to be exposed port by port.
- Ports have to be mapped to avoid conflicts.

Finding the web server port in a script

Parsing the output of `docker ps` would be painful.

There is a command to help us:

```
$ docker port <containerID> 80  
32768
```

Manual allocation of port numbers

If you want to set port numbers yourself, no problem:

```
$ docker run -d -p 80:80 nginx
$ docker run -d -p 8000:80 nginx
$ docker run -d -p 8080:80 -p 8888:80 nginx
```

- We are running three NGINX web servers.
- The first one is exposed on port 80.
- The second one is exposed on port 8000.
- The third one is exposed on ports 8080 and 8888.

Note: the convention is `port-on-host:port-on-container`.

Plumbing containers into your infrastructure

There are many ways to integrate containers in your network.

- Start the container, letting Docker allocate a public port for it.
Then retrieve that port number and feed it to your configuration.
- Pick a fixed port number in advance, when you generate your configuration.
Then start your container by setting the port numbers manually.
- Use a network plugin, connecting your containers with e.g. VLANs, tunnels...
- Enable *Swarm Mode* to deploy across a cluster.
The container will then be reachable through any node of the cluster.

When using Docker through an extra management layer like Mesos or Kubernetes, these will usually provide their own mechanism to expose containers.

Finding the container's IP address

We can use the `docker inspect` command to find the IP address of the container.

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' <yourContainerID>
172.17.0.3
```

- `docker inspect` is an advanced command, that can retrieve a ton of information about our containers.
- Here, we provide it with a format string to extract exactly the private IP address of the container.

Pinging our container

We can test connectivity to the container using the IP address we've just discovered. Let's see this now by using the `ping` tool.

```
$ ping <ipAddress>
64 bytes from <ipAddress>: icmp_req=1 ttl=64 time=0.085 ms
64 bytes from <ipAddress>: icmp_req=2 ttl=64 time=0.085 ms
64 bytes from <ipAddress>: icmp_req=3 ttl=64 time=0.085 ms
```

Section summary

We've learned how to:

- Expose a network port.
- Manipulate container networking basics.
- Find a container's IP address.

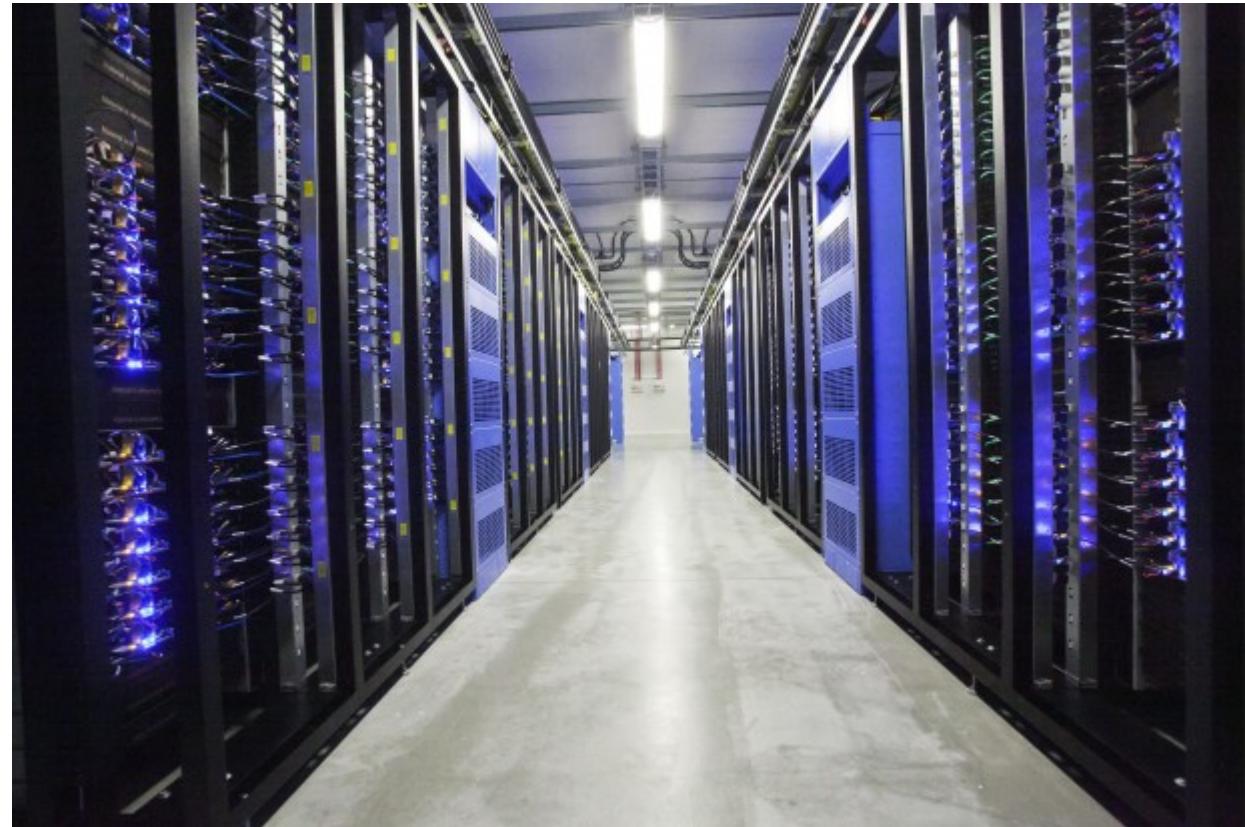
In the next chapter, we will see how to connect containers together without exposing their ports.



Working with volumes

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Working with volumes



Objectives

At the end of this section, you will be able to:

- Create containers holding volumes.
- Share volumes across containers.
- Share a host directory with one or many containers.

Working with volumes

Docker volumes can be used to achieve many things, including:

- Bypassing the copy-on-write system to obtain native disk I/O performance.
- Bypassing copy-on-write to leave some files out of `docker commit`.
- Sharing a directory between multiple containers.
- Sharing a directory between the host and a container.
- Sharing a *single file* between the host and a container.
- Using remote storage and custom storage with "volume drivers".

Volumes are special directories in a container

Volumes can be declared in two different ways.

- Within a Dockerfile, with a VOLUME instruction.

```
VOLUME /uploads
```

- On the command-line, with the -v flag for docker run.

```
$ docker run -d -v /uploads myapp
```

In both cases, /uploads (inside the container) will be a volume.



Volumes bypass the copy-on-write system

Volumes act as passthroughs to the host filesystem.

- The I/O performance on a volume is exactly the same as I/O performance on the Docker host.
- When you `docker commit`, the content of volumes is not brought into the resulting image.
- If a `RUN` instruction in a `Dockerfile` changes the content of a volume, those changes are not recorded either.
- If a container is started with the `--read-only` flag, the volume will still be writable (unless the volume is a read-only volume).



Volumes can be shared across containers

You can start a container with *exactly the same volumes* as another one.

The new container will have the same volumes, in the same directories.

They will contain exactly the same thing, and remain in sync.

Under the hood, they are actually the same directories on the host anyway.

This is done using the `--volumes-from` flag for `docker run`.

We will see an example in the following slides.



Sharing app server logs with another container

Let's start a Tomcat container:

```
$ docker run --name webapp -d -p 8080:8080 -v /usr/local/tomcat/logs tomcat
```

Now, start an `alpine` container accessing the same volume:

```
$ docker run --volumes-from webapp alpine sh -c "tail -f /usr/local/tomcat/logs/*"
```

Then, from another window, send requests to our Tomcat container:

```
$ curl localhost:8080
```

Volumes exist independently of containers

If a container is stopped or removed, its volumes still exist and are available.

Volumes can be listed and manipulated with `docker volume` subcommands:

```
$ docker volume ls
DRIVER      VOLUME NAME
local      5b0b65e4316da67c2d471086640e6005ca2264f3...
local      pgdata-prod
local      pgdata-dev
local      13b59c9936d78d109d094693446e174e5480d973...
```

Some of those volume names were explicit (pgdata-prod, pgdata-dev).

The others (the hex IDs) were generated automatically by Docker.

Naming volumes

- Volumes can be created without a container, then used in multiple containers.

Let's create a couple of volumes directly.

```
$ docker volume create webapps  
webapps
```

```
$ docker volume create logs  
logs
```

Volumes are not anchored to a specific path.

Using our named volumes

- Volumes are used with the `-v` option.
- When a host path does not contain a `/`, it is considered to be a volume name.

Let's start a web server using the two previous volumes.

```
$ docker run -d -p 1234:8080 \
  -v logs:/usr/local/tomcat/logs \
  -v webapps:/usr/local/tomcat/webapps \
  tomcat
```

Check that it's running correctly:

```
$ curl localhost:1234
... (Tomcat tells us how happy it is to be up and running) ...
```

Using a volume in another container

- We will make changes to the volume from another container.
- In this example, we will run a text editor in the other container.

(But this could be a FTP server, a WebDAV server, a Git receiver...)

Let's start another container using the `webapps` volume.

```
$ docker run -v webapps:/webapps -w /webapps -ti alpine vi ROOT/index.jsp
```

Vandalize the page, save, exit.

Then run `curl localhost:1234` again to see your changes.

Using custom "bind-mounts"

In some cases, you want a specific directory on the host to be mapped inside the container:

- You want to manage storage and snapshots yourself.
(With LVM, or a SAN, or ZFS, or anything else!)
- You have a separate disk with better performance (SSD) or resiliency (EBS) than the system disk, and you want to put important data on that disk.
- You want to share your source directory between your host (where the source gets edited) and the container (where it is compiled or executed).

Wait, we already met the last use-case in our example development workflow! Nice.

```
$ docker run -d -v /path/on/the/host:/path/in/container image ...
```



Migrating data with `--volumes-from`

The `--volumes-from` option tells Docker to re-use all the volumes of an existing container.

- Scenario: migrating from Redis 2.8 to Redis 3.0.
- We have a container (`myredis`) running Redis 2.8.
- Stop the `myredis` container.
- Start a new container, using the Redis 3.0 image, and the `--volumes-from` option.
- The new container will inherit the data of the old one.
- Newer containers can use `--volumes-from` too.
- Doesn't work across servers, so not usable in clusters (Swarm, Kubernetes).



Data migration in practice

Let's create a Redis container.

```
$ docker run -d --name redis28 redis:2.8
```

Connect to the Redis container and set some data.

```
$ docker run -ti --link redis28:redis busybox telnet redis 6379
```

Issue the following commands:

```
SET counter 42
INFO server
SAVE
QUIT
```



Upgrading Redis

Stop the Redis container.

```
$ docker stop redis28
```

Start the new Redis container.

```
$ docker run -d --name redis30 --volumes-from redis28 redis:3.0
```



Testing the new Redis

Connect to the Redis container and see our data.

```
docker run -ti --link redis30:redis busybox telnet redis 6379
```

Issue a few commands.

```
GET counter  
INFO server  
QUIT
```

Volumes lifecycle

- When you remove a container, its volumes are kept around.
- You can list them with `docker volume ls`.
- You can access them by creating a container with `docker run -v`.
- You can remove them with `docker volume rm` or `docker system prune`.

Ultimately, *you* are the one responsible for logging, monitoring, and backup of your volumes.



Checking volumes defined by an image

Wondering if an image has volumes? Just use `docker inspect`:

```
$ # docker inspect training/datavol
[{
  "config": {
    . . .
    "Volumes": {
      "/var/webapp": {}
    },
    . . .
  }]
}
```



Checking volumes used by a container

To look which paths are actually volumes, and to what they are bound, use `docker inspect` (again):

```
$ docker inspect <yourContainerID>
[{
  "ID": "<yourContainerID>",
  . . .
  "Volumes": {
    "/var/webapp": "/var/lib/docker/vfs/dir/f4280c5b6207ed531efd4cc673ff620cef2a7980f
  },
  "VolumesRW": {
    "/var/webapp": true
  },
}]
```

- We can see that our volume is present on the file system of the Docker host.

Sharing a single file

The same `-v` flag can be used to share a single file (instead of a directory).

One of the most interesting examples is to share the Docker control socket.

```
$ docker run -it -v /var/run/docker.sock:/var/run/docker.sock docker sh
```

From that container, you can now run `docker` commands communicating with the Docker Engine running on the host. Try `docker ps`!

 Since that container has access to the Docker socket, it has root-like access to the host.

Volume plugins

You can install plugins to manage volumes backed by particular storage systems, or providing extra features. For instance:

- [REX-Ray](#) - create and manage volumes backed by an enterprise storage system (e.g. SAN or NAS), or by cloud block stores (e.g. EBS, EFS).
- [Portworx](#) - provides distributed block store for containers.
- [Gluster](#) - open source software-defined distributed storage that can scale to several petabytes. It provides interfaces for object, block and file storage.
- and much more at the [Docker Store!](#)

Volumes vs. Mounts

- Since Docker 17.06, a new option is available: `--mount`.
- It offers a new, richer syntax to manipulate data in containers.
- It makes an explicit difference between:
 - volumes (identified with a unique name, managed by a storage plugin),
 - bind mounts (identified with a host path, not managed).
- The former `-v / --volume` option is still usable.

--mount syntax

Binding a host path to a container path:

```
$ docker run \
--mount type=bind,source=/path/on/host,target=/path/in/container alpine
```

Mounting a volume to a container path:

```
$ docker run \
--mount source=myvolume,target=/path/in/container alpine
```

Mounting a tmpfs (in-memory, for temporary files):

```
$ docker run \
--mount type=tmpfs,destination=/path/in/container,tmpfs-size=1000000 alpine
```

Section summary

We've learned how to:

- Create and manage volumes.
- Share volumes across containers.
- Share a host directory with one or many containers.



Limiting resources

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Limiting resources

- So far, we have used containers as convenient units of deployment.
- What happens when a container tries to use more resources than available?
(RAM, CPU, disk usage, disk and network I/O...)
- What happens when multiple containers compete for the same resource?
- Can we limit resources available to a container?
(Spoiler alert: yes!)

Container processes are normal processes

- Containers are closer to "fancy processes" than to "lightweight VMs".
- A process running in a container is, in fact, a process running on the host.
- Let's look at the output of `ps` on a container host running 3 containers :

```
 0  2662  0.2  0.3 /usr/bin/dockerd -H fd://
 0  2766  0.1  0.1  \_ docker-containerd --config /var/run/docker/containe
 0 23479  0.0  0.0      \_ docker-containerd-shim -namespace moby -workdir
 0 23497  0.0  0.0          \_ nginx: master process nginx -g daemon off;
101 23543  0.0  0.0            \_ nginx: worker process
 0 23565  0.0  0.0      \_ docker-containerd-shim -namespace moby -workdir
102 23584  9.4 11.3          \_ /docker-java-home/jre/bin/java -Xms2g -Xmx2
 0 23707  0.0  0.0      \_ docker-containerd-shim -namespace moby -workdir
 0 23725  0.0  0.0          \_ /bin/sh
```

- The highlighted processes are containerized processes.
(That host is running nginx, elasticsearch, and alpine.)

By default: nothing changes

- What happens when a process uses too much memory on a Linux system?

By default: nothing changes

- What happens when a process uses too much memory on a Linux system?
- Simplified answer:
 - swap is used (if available);
 - if there is not enough swap space, eventually, the out-of-memory killer is invoked;
 - the OOM killer uses heuristics to kill processes;
 - sometimes, it kills an unrelated process.

By default: nothing changes

- What happens when a process uses too much memory on a Linux system?
- Simplified answer:
 - swap is used (if available);
 - if there is not enough swap space, eventually, the out-of-memory killer is invoked;
 - the OOM killer uses heuristics to kill processes;
 - sometimes, it kills an unrelated process.
- What happens when a container uses too much memory?
- The same thing!

(i.e., a process eventually gets killed, possibly in another container.)

Limiting container resources

- The Linux kernel offers rich mechanisms to limit container resources.
- For memory usage, the mechanism is part of the *cgroup* subsystem.
- This subsystem allows to limit the memory for a process or a group of processes.
- A container engine leverages these mechanisms to limit memory for a container.
- The out-of-memory killer has a new behavior:
 - it runs when a container exceeds its allowed memory usage,
 - in that case, it only kills processes in that container.

Limiting memory in practice

- The Docker Engine offers multiple flags to limit memory usage.
- The two most useful ones are `--memory` and `--memory-swap`.
- `--memory` limits the amount of physical RAM used by a container.
- `--memory-swap` limits the total amount (RAM+swap) used by a container.
- The memory limit can be expressed in bytes, or with a unit suffix.
(e.g.: `--memory 100m` = 100 megabytes.)
- We will see two strategies: limiting RAM usage, or limiting both

Limiting RAM usage

Example:

```
docker run -ti --memory 100m python
```

If the container tries to use more than 100 MB of RAM, *and* swap is available:

- the container will not be killed,
- memory above 100 MB will be swapped out,
- in most cases, the app in the container will be slowed down (a lot).

If we run out of swap, the global OOM killer still intervenes.

Limiting both RAM and swap usage

Example:

```
docker run -ti --memory 100m --memory-swap 100m python
```

If the container tries to use more than 100 MB of memory, it is killed.

On the other hand, the application will never be slowed down because of swap.

When to pick which strategy?

- Stateful services (like databases) will lose or corrupt data when killed
- Allow them to use swap space, but monitor swap usage
- Stateless services can usually be killed with little impact
- Limit their mem+swap usage, but monitor if they get killed
- Ultimately, this is no different from "do I want swap, and how much?"

Limiting CPU usage

- There are no less than 3 ways to limit CPU usage:
 - setting a relative priority with `--cpu-shares`,
 - setting a CPU% limit with `--cpus`,
 - pinning a container to specific CPUs with `--cpuset-cpus`.
- They can be used separately or together.

Setting relative priority

- Each container has a relative priority used by the Linux scheduler.
- By default, this priority is 1024.
- As long as CPU usage is not maxed out, this has no effect.
- When CPU usage is maxed out, each container receives CPU cycles in proportion of its relative priority.
- In other words: a container with `--cpu-shares 2048` will receive twice as much than the default.

Setting a CPU% limit

- This setting will make sure that a container doesn't use more than a given % of CPU.
- The value is expressed in CPUs; therefore:

`--cpus 0.1` means 10% of one CPU,

`--cpus 1.0` means 100% of one whole CPU,

`--cpus 10.0` means 10 entire CPUs.

Pinning containers to CPUs

- On multi-core machines, it is possible to restrict the execution on a set of CPUs.
- Examples:

`--cpuset-cpus 0` forces the container to run on CPU 0;

`--cpuset-cpus 3,5,7` restricts the container to CPUs 3, 5, 7;

`--cpuset-cpus 0-3,8-11` restricts the container to CPUs 0, 1, 2, 3, 8, 9, 10, 11.

- This will not reserve the corresponding CPUs!

(They might still be used by other containers, or uncontainerized processes.)

Limiting disk usage

- Most storage drivers do not support limiting the disk usage of containers.
(With the exception of devicemapper, but the limit cannot be set easily.)
- This means that a single container could exhaust disk space for everyone.
- In practice, however, this is not a concern, because:
 - data files (for stateful services) should reside on volumes,
 - assets (e.g. images, user-generated content...) should reside on object stores or on volume,
 - logs are written on standard output and gathered by the container engine.
- Container disk usage can be audited with `docker ps -s` and `docker diff`.



Deep dive into container internals

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Deep dive into container internals

In this chapter, we will explain some of the fundamental building blocks of containers.

This will give you a solid foundation so you can:

- understand "what's going on" in complex situations,
- anticipate the behavior of containers (performance, security...) in new scenarios,
- implement your own container engine.

The last item should be done for educational purposes only!

There is no container code in the Linux kernel

- If we search "container" in the Linux kernel code, we find:
 - generic code to manipulate data structures (like linked lists, etc.),
 - unrelated concepts like "ACPI containers",
 - *nothing* relevant to "our" containers!
- Containers are composed using multiple independent features.
- On Linux, containers rely on "namespaces, cgroups, and some filesystem magic."
- Security also requires features like capabilities, seccomp, LSMs...



Namespaces

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Namespaces

- Provide processes with their own view of the system.
- Namespaces limit what you can see (and therefore, what you can use).
- These namespaces are available in modern kernels:
 - pid
 - net
 - mnt
 - uts
 - ipc
 - user

(We are going to detail them individually.)

- Each process belongs to one namespace of each type.

Namespaces are always active

- Namespaces exist even when you don't use containers.
- This is a bit similar to the UID field in UNIX processes:
 - all processes have the UID field, even if no user exists on the system
 - the field always has a value / the value is always defined
(i.e. any process running on the system has some UID)
 - the value of the UID field is used when checking permissions
(the UID field determines which resources the process can access)
- You can replace "UID field" with "namespace" above and it still works!
- In other words: even when you don't use containers,
there is one namespace of each type, containing all the processes on the system.



Manipulating namespaces

- Namespaces are created with two methods:
 - the `clone()` system call (used when creating new threads and processes),
 - the `unshare()` system call.
- The Linux tool `unshare` allows to do that from a shell.
- A new process can re-use none / all / some of the namespaces of its parent.
- It is possible to "enter" a namespace with the `setns()` system call.
- The Linux tool `nsenter` allows to do that from a shell.



Namespaces lifecycle

- When the last process of a namespace exits, the namespace is destroyed.
- All the associated resources are then removed.
- Namespaces are materialized by pseudo-files in `/proc/<pid>/ns`.

```
ls -l /proc/self/ns
```

- It is possible to compare namespaces by checking these files.
(This helps to answer the question, "are these two processes in the same namespace?")
- It is possible to preserve a namespace by bind-mounting its pseudo-file.



Namespaces can be used independently

- As mentioned in the previous slides:

A new process can re-use none / all / some of the namespaces of its parent.

- We are going to use that property in the examples in the next slides.
- We are going to present each type of namespace.
- For each type, we will provide an example using only that namespace.

UTS namespace

- `gethostname` / `sethostname`
- Allows to set a custom hostname for a container.
- That's (mostly) it!
- Also allows to set the NIS domain.

(If you don't know what a NIS domain is, you don't have to worry about it!)

- If you're wondering: UTS = UNIX time sharing.
- This namespace was named like this because of the `struct utsname`, which is commonly used to obtain the machine's hostname, architecture, etc.

(The more you know!)



Creating our first namespace

Let's use `unshare` to create a new process that will have its own UTS namespace:

```
$ sudo unshare --uts
```

- We have to use `sudo` for most `unshare` operations.
- We indicate that we want a new uts namespace, and nothing else.
- If we don't specify a program to run, a `$SHELL` is started.



Demonstrating our uts namespace

In our new "container", check the hostname, change it, and check it:

```
# hostname  
nodeX  
# hostname tupperware  
# hostname  
tupperware
```

In another shell, check that the machine's hostname hasn't changed:

```
$ hostname  
nodeX
```

Exit the "container" with `exit` or `Ctrl-D`.

Net namespace overview

- Each network namespace has its own private network stack.
- The network stack includes:
 - network interfaces (including `lo`),
 - routing tables (as in `ip rule` etc.),
 - iptables chains and rules,
 - sockets (as seen by `ss`, `netstat`).
- You can move a network interface from a network namespace to another:

```
ip link set dev eth0 netns PID
```

Net namespace typical use

- Each container is given its own network namespace.
- For each network namespace (i.e. each container), a `veth` pair is created.
(Two `veth` interfaces act as if they were connected with a cross-over cable.)
- One `veth` is moved to the container network namespace (and renamed `eth0`).
- The other `veth` is moved to a bridge on the host (e.g. the `docker0` bridge).



Creating a network namespace

Start a new process with its own network namespace:

```
$ sudo unshare --net
```

See that this new network namespace is unconfigured:

```
# ping 1.1
connect: Network is unreachable
# ifconfig
# ip link ls
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 100
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```



Creating the veth interfaces

In another shell (on the host), create a `veth` pair:

```
$ sudo ip link add name in_host type veth peer name in_netns
```

Configure the host side (`in_host`):

```
$ sudo ip link set in_host master docker0 up
```



Moving the veth interface

In the process created by `unshare`, check the PID of our "network container":

```
# echo $$  
533
```

On the host, move the other side (`in_netns`) to the network namespace:

```
$ sudo ip link set in_netns netns 533
```

(Make sure to update "533" with the actual PID obtained above!)



Basic network configuration

Let's set up `lo` (the loopback interface):

```
# ip link set lo up
```

Activate the `veth` interface and rename it to `eth0`:

```
# ip link set in_netns name eth0 up
```



Allocating IP address and default route

On the host, check the address of the Docker bridge:

```
$ ip addr ls dev docker0
```

(It could be something like 172.17.0.1.)

Pick an IP address in the middle of the same subnet, e.g. 172.17.0.99.

In the process created by unshare, configure the interface:

```
# ip addr add 172.17.0.99/24 dev eth0  
# ip route add default via 172.17.0.1
```

(Make sure to update the IP addresses if necessary.)



Validating the setup

Check that we now have connectivity:

```
# ping 1.1
```

Note: we were able to take a shortcut, because Docker is running, and provides us with a `docker0` bridge and a valid `iptables` setup.

If Docker is not running, you will need to take care of this!



Cleaning up network namespaces

- Terminate the process created by `unshare` (with `exit` or `Ctrl-D`).
- Since this was the only process in the network namespace, it is destroyed.
- All the interfaces in the network namespace are destroyed.
- When a `veth` interface is destroyed, it also destroys the other half of the pair.
- So we don't have anything else to do to clean up!

Other ways to use network namespaces

- `--net none` gives an empty network namespace to a container.
(Effectively isolating it completely from the network.)
- `--net host` means "do not containerize the network".
(No network namespace is created; the container uses the host network stack.)
- `--net container` means "reuse the network namespace of another container".
(As a result, both containers share the same interfaces, routes, etc.)

Mnt namespace

- Processes can have their own root fs (à la chroot).
- Processes can also have "private" mounts. This allows to:
 - isolate `/tmp` (per user, per service...)
 - mask `/proc`, `/sys` (for processes that don't need them)
 - mount remote filesystems or sensitive data,
but make it visible only for allowed processes
- Mounts can be totally private, or shared.
- At this point, there is no easy way to pass along a mount from a namespace to another.



Setting up a private `/tmp`

Create a new mount namespace:

```
$ sudo unshare --mount
```

In that new namespace, mount a brand new `/tmp`:

```
# mount -t tmpfs none /tmp
```

Check the content of `/tmp` in the new namespace, and compare to the host.

The mount is automatically cleaned up when you exit the process.

PID namespace

- Processes within a PID namespace only "see" processes in the same PID namespace.
- Each PID namespace has its own numbering (starting at 1).
- When PID 1 goes away, the whole namespace is killed.

(When PID 1 goes away on a normal UNIX system, the kernel panics!)

- Those namespaces can be nested.
- A process ends up having multiple PIDs (one per namespace in which it is nested).



PID namespace in action

Create a new PID namespace:

```
$ sudo unshare --pid --fork
```

(We need the `--fork` flag because the PID namespace is special.)

Check the process tree in the new namespace:

```
# ps faux
```



PID namespace in action

Create a new PID namespace:

```
$ sudo unshare --pid --fork
```

(We need the `--fork` flag because the PID namespace is special.)

Check the process tree in the new namespace:

```
# ps faux
```

🤔 Why do we see all the processes?!?



PID namespaces and /proc

- Tools like `ps` rely on the `/proc` pseudo-filesystem.
- Our new namespace still has access to the original `/proc`.
- Therefore, it still sees host processes.
- But it cannot affect them.

(Try to `kill` a process: you will get `No such process`.)



PID namespaces, take 2

- This can be solved by mounting `/proc` in the namespace.
- The `unshare` utility provides a convenience flag, `--mount-proc`.
- This flag will mount `/proc` in the namespace.
- It will also unshare the mount namespace, so that this mount is local.

Try it:

```
$ sudo unshare --pid --fork --mount-proc  
# ps faux
```



OK, really, why do we need --fork?

It is not necessary to remember all these details.

This is just an illustration of the complexity of namespaces!

The `unshare` tool calls the `unshare` syscall, then `execs` the new binary.

A process calling `unshare` to create new namespaces is moved to the new namespaces...

... Except for the PID namespace.

(Because this would change the current PID of the process from X to 1.)

The processes created by the new binary are placed into the new PID namespace.

The first one will be PID 1.

If PID 1 exits, it is not possible to create additional processes in the namespace.

(Attempting to do so will result in `ENOMEM`.)

Without the `--fork` flag, the first command that we execute will be PID 1 ...

... And once it exits, we cannot create more processes in the namespace!

Check `man 2 unshare` and `man pid_namespaces` if you want more details.

IPC namespace

IPC namespace

- Does anybody know about IPC?

IPC namespace

- Does anybody know about IPC?
- Does anybody *care* about IPC?

IPC namespace

- Does anybody know about IPC?
- Does anybody *care* about IPC?
- Allows a process (or group of processes) to have own:
 - IPC semaphores
 - IPC message queues
 - IPC shared memory

... without risk of conflict with other instances.
- Older versions of PostgreSQL cared about this.

No demo for that one.

User namespace

- Allows to map UID/GID; e.g.:
 - UID 0→1999 in container C1 is mapped to UID 10000→11999 on host
 - UID 0→1999 in container C2 is mapped to UID 12000→13999 on host
 - etc.
- UID 0 in the container can still perform privileged operations in the container.
(For instance: setting up network interfaces.)
- But outside of the container, it is a non-privileged user.
- It also means that the UID in containers becomes unimportant.
(Just use UID 0 in the container, since it gets squashed to a non-privileged user outside.)
- Ultimately enables better privilege separation in container engines.



User namespace challenges

- UID needs to be mapped when passed between processes or kernel subsystems.
- Filesystem permissions and file ownership are more complicated.

(E.g. when the same root filesystem is shared by multiple containers running with different UIDs.)

- With the Docker Engine:
 - some feature combinations are not allowed
(e.g. user namespace + host network namespace sharing)
 - user namespaces need to be enabled/disabled globally
(when the daemon is started)
 - container images are stored separately
(so the first time you toggle user namespaces, you need to re-pull images)

No demo for that one.



Control groups

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Control groups

- Control groups provide resource *metering* and *limiting*.
- This covers a number of "usual suspects" like:
 - memory
 - CPU
 - block I/O
 - network (with cooperation from iptables/tc)
- And a few exotic ones:
 - huge pages (a special way to allocate memory)
 - RDMA (resources specific to InfiniBand / remote memory transfer)

Crowd control

- Control groups also allow to group processes for special operations:
 - freezer (conceptually similar to a "mass-SIGSTOP/SIGCONT")
 - perf_event (gather performance events on multiple processes)
 - cpuset (limit or pin processes to specific CPUs)
- There is a "pids" cgroup to limit the number of processes in a given group.
- There is also a "devices" cgroup to control access to device nodes.
(i.e. everything in `/dev`.)

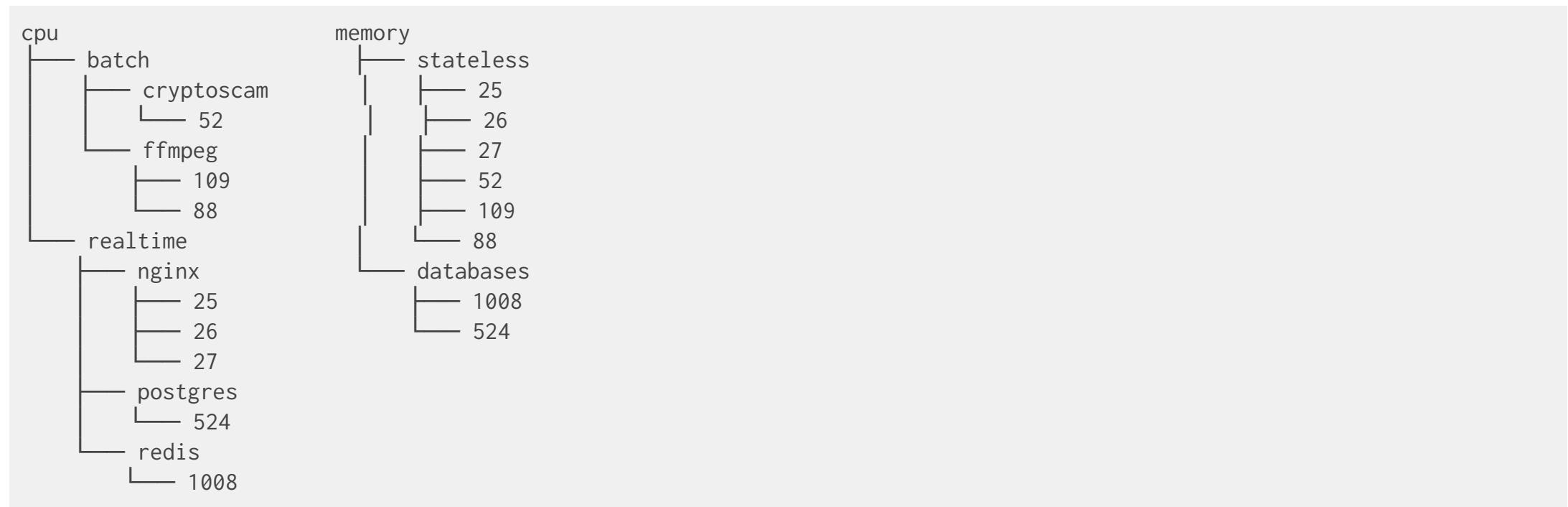
Generalities

- Cgroups form a hierarchy (a tree).
- We can create nodes in that hierarchy.
- We can associate limits to a node.
- We can move a process (or multiple processes) to a node.
- The process (or processes) will then respect these limits.
- We can check the current usage of each node.
- In other words: limits are optional (if we only want accounting).
- When a process is created, it is placed in its parent's groups.

Example

The numbers are PIDs.

The names are the names of our nodes (arbitrarily chosen).





Cgroups v1 vs v2

- Cgroups v1 are available on all systems (and widely used).
- Cgroups v2 are a huge refactor.

(Development started in Linux 3.10, released in 4.5.)

- Cgroups v2 have a number of differences:
 - single hierarchy (instead of one tree per controller),
 - processes can only be on leaf nodes (not inner nodes),
 - and of course many improvements / refactorings.

Memory cgroup: accounting

- Keeps track of pages used by each group:
 - file (read/write/mmap from block devices),
 - anonymous (stack, heap, anonymous mmap),
 - active (recently accessed),
 - inactive (candidate for eviction).
- Each page is "charged" to a group.
- Pages can be shared across multiple groups.

(Example: multiple processes reading from the same files.)

- To view all the counters kept by this cgroup:

```
$ cat /sys/fs/cgroup/memory/memory.stat
```

Memory cgroup: limits

- Each group can have (optional) hard and soft limits.
- Limits can be set for different kinds of memory:
 - physical memory,
 - kernel memory,
 - total memory (including swap).

Soft limits and hard limits

- Soft limits are not enforced.

(But they influence reclaim under memory pressure.)

- Hard limits *cannot* be exceeded:

- if a group of processes exceeds a hard limit,
- and if the kernel cannot reclaim any memory,
- then the OOM (out-of-memory) killer is triggered,
- and processes are killed until memory gets below the limit again.



Avoiding the OOM killer

- For some workloads (databases and stateful systems), killing processes because we run out of memory is not acceptable.
- The "oom-notifier" mechanism helps with that.
- When "oom-notifier" is enabled and a hard limit is exceeded:
 - all processes in the cgroup are frozen,
 - a notification is sent to user space (instead of killing processes),
 - user space can then raise limits, migrate containers, etc.,
 - once the memory usage is below the hard limit, unfreeze the cgroup.



Overhead of the memory cgroup

- Each time a process grabs or releases a page, the kernel update counters.
- This adds some overhead.
- Unfortunately, this cannot be enabled/disabled per process.
- It has to be done system-wide, at boot time.
- Also, when multiple groups use the same page:
 - only the first group gets "charged",
 - but if it stops using it, the "charge" is moved to another group.



Setting up a limit with the memory cgroup

Create a new memory cgroup:

```
$ CG=/sys/fs/cgroup/memory/onehundredmegs  
$ sudo mkdir $CG
```

Limit it to approximately 100MB of memory usage:

```
$ sudo tee $CG/memory.memsw.limit_in_bytes <<< 100000000
```

Move the current process to that cgroup:

```
$ sudo tee $CG/tasks <<< $$
```

The current process *and all its future children* are now limited.

(Confused about <<<? Look at the next slide!)



What's <<<?

- This is a "here string". (It is a non-POSIX shell extension.)
- The following commands are equivalent:

```
foo <<< hello
```

```
echo hello | foo
```

```
foo <<EOF  
hello  
EOF
```

- Why did we use that?



Writing to cgroups pseudo-files requires root

Instead of:

```
sudo tee $CG/tasks <<< $$
```

We could have done:

```
sudo sh -c "echo $$ > $CG/tasks"
```

The following commands, however, would be invalid:

```
sudo echo $$ > $CG/tasks
```

```
sudo -i # (or su)  
echo $$ > $CG/tasks
```



Testing the memory limit

Start the Python interpreter:

```
$ python
Python 3.6.4 (default, Jan  5 2018, 02:35:40)
[GCC 7.2.1 20171224] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Allocate 80 megabytes:

```
>>> s = "!" * 1000000 * 80
```

Add 20 megabytes more:

```
>>> t = "!" * 1000000 * 20
Killed
```

CPU cgroup

- Keeps track of CPU time used by a group of processes.

(This is easier and more accurate than `getrusage` and `/proc`.)

- Keeps track of usage per CPU as well.

(i.e., "this group of process used X seconds of CPU0 and Y seconds of CPU1".)

- Allows to set relative weights used by the scheduler.

Cpuset cgroup

- Pin groups to specific CPU(s).
- Use-case: reserve CPUs for specific apps.
- Warning: make sure that "default" processes aren't using all CPUs!
- CPU pinning can also avoid performance loss due to cache flushes.
- This is also relevant for NUMA systems.
- Provides extra dials and knobs.

(Per zone memory pressure, process migration costs...)

Blkio cgroup

- Keeps track of I/Os for each group:
 - per block device
 - read vs write
 - sync vs async
- Set throttle (limits) for each group:
 - per block device
 - read vs write
 - ops vs bytes
- Set relative weights for each group.
- Note: most writes go through the page cache.
(So classic writes will appear to be unthrottled at first.)

Net_cls and net_prio cgroup

- Only works for egress (outgoing) traffic.
- Automatically set traffic class or priority for traffic generated by processes in the group.
- Net_cls will assign traffic to a class.
- Classes have to be matched with tc or iptables, otherwise traffic just flows normally.
- Net_prio will assign traffic to a priority.
- Priorities are used by queuing disciplines.

Devices cgroup

- Controls what the group can do on device nodes
- Permissions include read/write/mknod
- Typical use:
 - allow `/dev/{tty,zero,random,null} ...`
 - deny everything else
- A few interesting nodes:
 - `/dev/net/tun` (network interface manipulation)
 - `/dev/fuse` (filesystems in user space)
 - `/dev/kvm` (VMs in containers, yay inception!)
 - `/dev/dri` (GPU)



Security features

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Security features

- Namespaces and cgroups are not enough to ensure strong security.
- We need extra mechanisms: capabilities, seccomp, LSMs.
- These mechanisms were already used before containers to harden security.
- They can be used together with containers.
- Good container engines will automatically leverage these features.

(So that you don't have to worry about it.)

Capabilities

- In traditional UNIX, many operations are possible if and only if UID=0 (root).
- Some of these operations are very powerful:
 - changing file ownership, accessing all files ...
- Some of these operations deal with system configuration, but can be abused:
 - setting up network interfaces, mounting filesystems ...
- Some of these operations are not very dangerous but are needed by servers:
 - binding to a port below 1024.
- Capabilities are per-process flags to allow these operations individually.

Some capabilities

- `CAP_CHOWN`: arbitrarily change file ownership and permissions.
- `CAP_DAC_OVERRIDE`: arbitrarily bypass file ownership and permissions.
- `CAP_NET_ADMIN`: configure network interfaces, iptables rules, etc.
- `CAP_NET_BIND_SERVICE`: bind a port below 1024.

See `man capabilities` for the full list and details.

Using capabilities

- Container engines will typically drop all "dangerous" capabilities.
- You can then re-enable capabilities on a per-container basis, as needed.
- With the Docker engine: `docker run --cap-add ...`
- If you write your own code to manage capabilities:
 - make sure that you understand what each capability does,
 - read about *ambient* capabilities as well.

Seccomp

- Seccomp is secure computing.
- Achieve high level of security by restricting drastically available syscalls.
- Original seccomp only allows `read()`, `write()`, `exit()`, `sigreturn()`.
- The seccomp-bpf extension allows to specify custom filters with BPF rules.
- This allows to filter by syscall, and by parameter.
- BPF code can perform arbitrarily complex checks, quickly, and safely.
- Container engines take care of this so you don't have to.

Linux Security Modules

- The most popular ones are SELinux and AppArmor.
- Red Hat distros generally use SELinux.
- Debian distros (in particular, Ubuntu) generally use AppArmor.
- LSMs add a layer of access control to all process operations.
- Container engines take care of this so you don't have to.



Copy-on-write filesystems

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Copy-on-write filesystems

Container engines rely on copy-on-write to be able to start containers quickly, regardless of their size.

We will explain how that works, and review some of the copy-on-write storage systems available on Linux.

What is copy-on-write?

- Copy-on-write is a mechanism allowing to share data.
- The data appears to be a copy, but is only a link (or reference) to the original data.
- The actual copy happens only when someone tries to change the shared data.
- Whoever changes the shared data ends up using their own copy instead of the shared data.

A few metaphors

A few metaphors

- First metaphor:
white board and tracing paper

A few metaphors

- First metaphor:
white board and tracing paper
- Second metaphor:
magic books with shadowy pages

A few metaphors

- First metaphor:
white board and tracing paper
- Second metaphor:
magic books with shadowy pages
- Third metaphor:
just-in-time house building

Copy-on-write is *everywhere*

- Process creation with `fork()`.
- Consistent disk snapshots.
- Efficient VM provisioning.
- And, of course, containers.

Copy-on-write and containers

Copy-on-write is essential to give us "convenient" containers.

- Creating a new container (from an existing image) is "free".
(Otherwise, we would have to copy the image first.)
- Customizing a container (by tweaking a few files) is cheap.
(Adding a 1 KB configuration file to a 1 GB container takes 1 KB, not 1 GB.)
- We can take snapshots, i.e. have "checkpoints" or "save points" when building images.

AUFS overview

- The original (legacy) copy-on-write filesystem used by first versions of Docker.
- Combine multiple *branches* in a specific order.
- Each branch is just a normal directory.
- You generally have:
 - at least one read-only branch (at the bottom),
 - exactly one read-write branch (at the top).

(But other fun combinations are possible too!)

AUFS operations: opening a file

- With `O_RDONLY` - read-only access:
 - look it up in each branch, starting from the top
 - open the first one we find
- With `O_WRONLY` or `O_RDWR` - write access:
 - if the file exists on the top branch: open it
 - if the file exists on another branch: "copy up"
(i.e. copy the file to the top branch and open the copy)
 - if the file doesn't exist on any branch: create it on the top branch

That "copy-up" operation can take a while if the file is big!

AUFS operations: deleting a file

- A *whiteout* file is created.
- This is similar to the concept of "tombstones" used in some data systems.

```
# docker run ubuntu rm /etc/shadow
```

```
# ls -la /var/lib/docker/aufs/diff/$(docker ps --no-trunc -lq)/etc
total 8
drwxr-xr-x 2 root root 4096 Jan 27 15:36 .
drwxr-xr-x 5 root root 4096 Jan 27 15:36 ..
-r--r--r-- 2 root root    0 Jan 27 15:36 .wh.shadow
```

AUFS performance

- AUFS `mount()` is fast, so creation of containers is quick.
- Read/write access has native speeds.
- But initial `open()` is expensive in two scenarios:
 - when writing big files (log files, databases ...),
 - when searching many directories (PATH, classpath, etc.) over many layers.
- Protip: when we built dotCloud, we ended up putting all important data on *volumes*.
- When starting the same container multiple times:
 - the data is loaded only once from disk, and cached only once in memory;
 - but `dentries` will be duplicated.

Device Mapper

Device Mapper is a rich subsystem with many features.

It can be used for: RAID, encrypted devices, snapshots, and more.

In the context of containers (and Docker in particular), "Device Mapper" means:

"the Device Mapper system + its *thin provisioning target*"

If you see the abbreviation "thinp" it stands for "thin provisioning".

Device Mapper principles

- Copy-on-write happens on the *block* level (instead of the *file* level).
- Each container and each image get their own block device.
- At any given time, it is possible to take a snapshot:
 - of an existing container (to create a frozen image),
 - of an existing image (to create a container from it).
- If a block has never been written to:
 - it's assumed to be all zeros,
 - it's not allocated on disk.

(That last property is the reason for the name "thin" provisioning.)

Device Mapper operational details

- Two storage areas are needed: one for *data*, another for *metadata*.
- "data" is also called the "pool"; it's just a big pool of blocks.

(Docker uses the smallest possible block size, 64 KB.)

- "metadata" contains the mappings between virtual offsets (in the snapshots) and physical offsets (in the pool).
- Each time a new block (or a copy-on-write block) is written, a block is allocated from the pool.
- When there are no more blocks in the pool, attempts to write will stall until the pool is increased (or the write operation aborted).
- In other words: when running out of space, containers are frozen, but operations will resume as soon as space is available.

Device Mapper performance

- By default, Docker puts data and metadata on a loop device backed by a sparse file.
- This is great from a usability point of view, since zero configuration is needed.
- But it is terrible from a performance point of view:
 - each time a container writes to a new block,
 - a block has to be allocated from the pool,
 - and when it's written to,
 - a block has to be allocated from the sparse file,
 - and sparse file performance isn't great anyway.
- If you use Device Mapper, make sure to put data (and metadata) on devices!

BTRFS principles

- BTRFS is a filesystem (like EXT4, XFS, NTFS...) with built-in snapshots.
- The "copy-on-write" happens at the filesystem level.
- BTRFS integrates the snapshot and block pool management features at the filesystem level.

(Instead of the block level for Device Mapper.)

- In practice, we create a "subvolume" and later take a "snapshot" of that subvolume.

Imagine: `mkdir` with Super Powers and `cp -a` with Super Powers.

- These operations can be executed with the `btrfs` CLI tool.

BTRFS in practice with Docker

- Docker can use BTRFS and its snapshotting features to store container images.
- The only requirement is that `/var/lib/docker` is on a BTRFS filesystem.
(Or, the directory specified with the `--data-root` flag when starting the engine.)



BTRFS quirks

- BTRFS works by dividing its storage in *chunks*.
- A chunk can contain data or metadata.
- You can run out of chunks (and get `No space left on device`) even though `df` shows space available.

(Because chunks are only partially allocated.)

- Quick fix:

```
# btrfs filesystem balance start -dusage=1 /var/lib/docker
```

Overlay2

- Overlay2 is very similar to AUFS.
- However, it has been merged in "upstream" kernel.
- It is therefore available on all modern kernels.
(AUFS was available on Debian and Ubuntu, but required custom kernels on other distros.)
- It is simpler than AUFS (it can only have two branches, called "layers").
- The container engine abstracts this detail, so this is not a concern.
- Overlay2 storage drivers generally use hard links between layers.
- This improves `stat()` and `open()` performance, at the expense of inode usage.

ZFS

- ZFS is similar to BTRFS (at least from a container user's perspective).
- Pros:
 - high performance
 - high reliability (with e.g. data checksums)
 - optional data compression and deduplication
- Cons:
 - high memory usage
 - not in upstream kernel
- It is available as a kernel module or through FUSE.

Which one is the best?

- Eventually, overlay2 should be the best option.
- It is available on all modern systems.
- Its memory usage is better than Device Mapper, BTRFS, or ZFS.
- The remarks about *write performance* shouldn't bother you:
data should always be stored in volumes anyway!



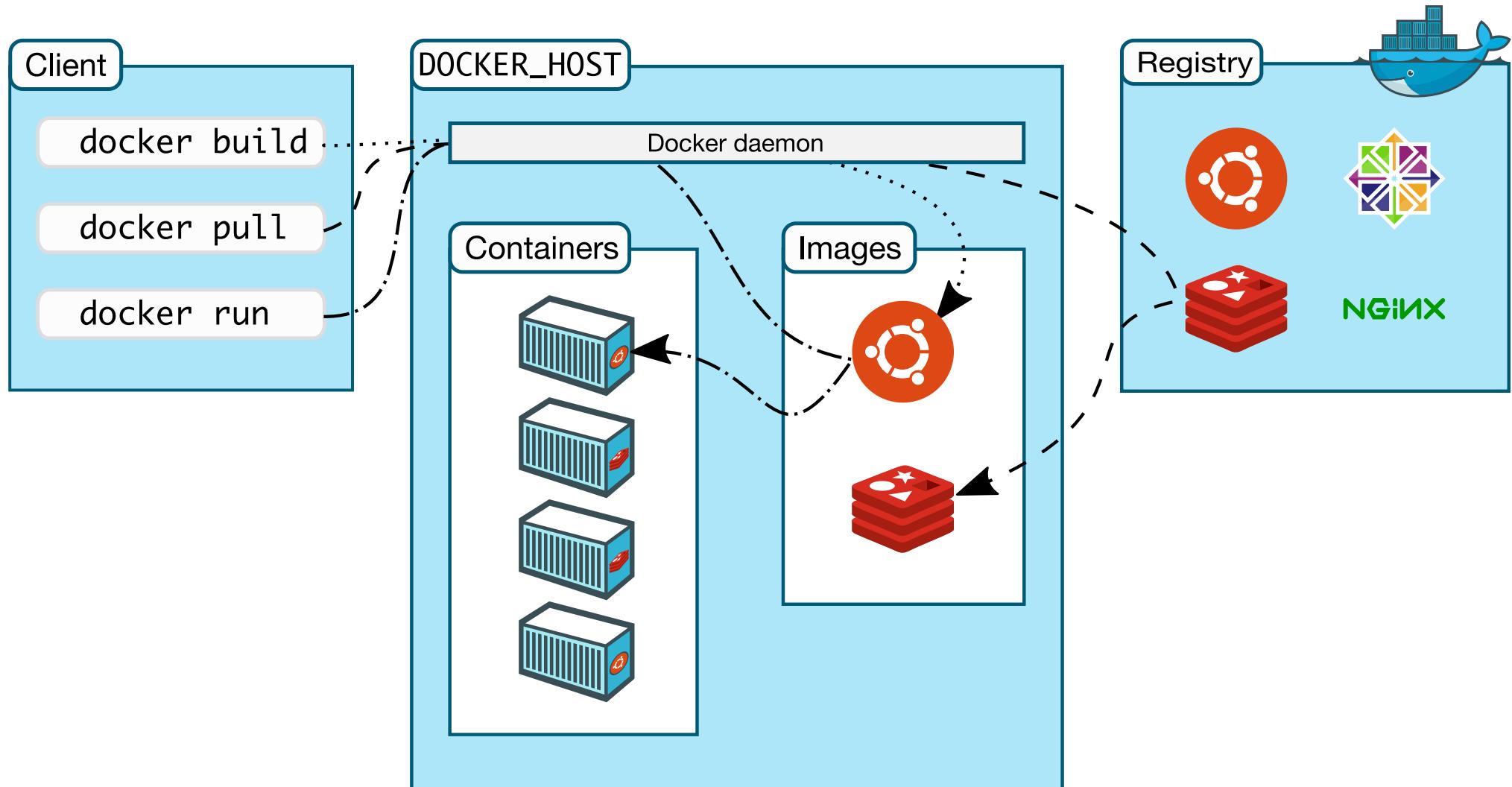
Docker Engine and other container engines

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Docker Engine and other container engines

- We are going to cover the architecture of the Docker Engine.
- We will also present other container engines.

Docker Engine external architecture



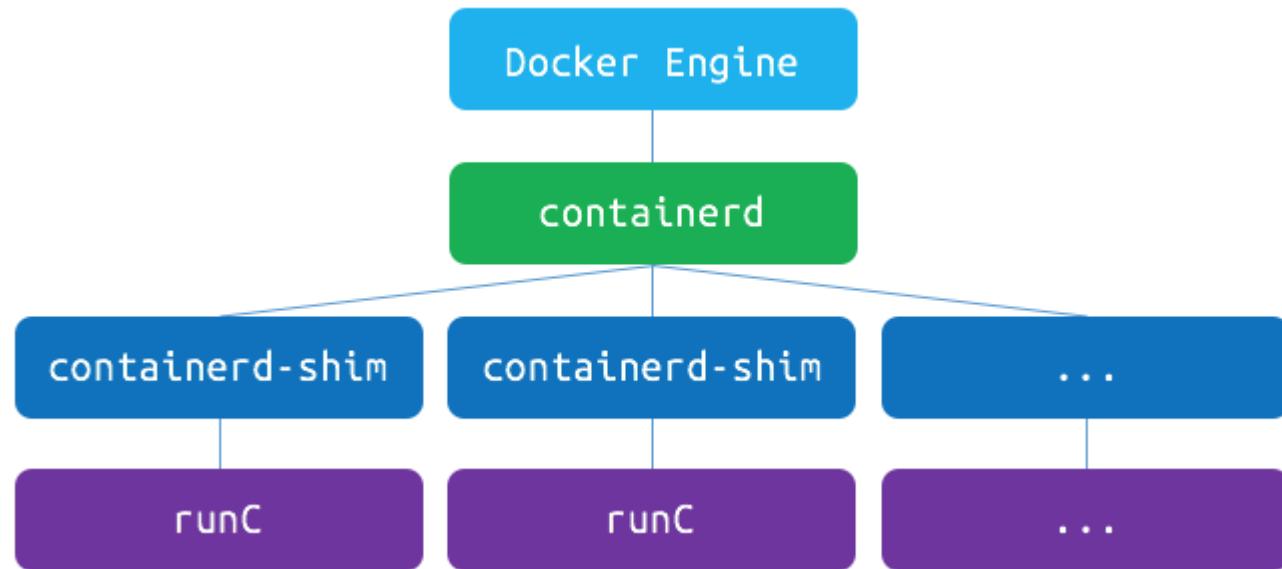
Docker Engine external architecture

- The Engine is a daemon (service running in the background).
- All interaction is done through a REST API exposed over a socket.
- On Linux, the default socket is a UNIX socket: `/var/run/docker.sock`.
- We can also use a TCP socket, with optional mutual TLS authentication.
- The `docker` CLI communicates with the Engine over the socket.

Note: strictly speaking, the Docker API is not fully REST.

Some operations (e.g. dealing with interactive containers and log streaming) don't fit the REST model.

Docker Engine internal architecture



Docker Engine internal architecture

- Up to Docker 1.10: the Docker Engine is one single monolithic binary.
- Starting with Docker 1.11, the Engine is split into multiple parts:
 - `dockerd` (REST API, auth, networking, storage)
 - `containerd` (container lifecycle, controlled over a gRPC API)
 - `containerd-shim` (per-container; does almost nothing but allows to restart the Engine without restarting the containers)
 - `runc` (per-container; does the actual heavy lifting to start the container)
- Some features (like image and snapshot management) are progressively being pushed from `dockerd` to `containerd`.

For more details, check [this short presentation by Phil Estes](#).

Other container engines

The following list is not exhaustive.

Furthermore, we limited the scope to Linux containers.

Containers also exist (sometimes with other names) on Windows, macOS, Solaris, FreeBSD

...

LXC

- The venerable ancestor (first released in 2008).
- Docker initially relied on it to execute containers.
- No daemon; no central API.
- Each container is managed by a `lxc-start` process.
- Each `lxc-start` process exposes a custom API over a local UNIX socket, allowing to interact with the container.
- No notion of image (container filesystems have to be managed manually).
- Networking has to be setup manually.

LXD

- Re-uses LXC code (through liblxc).
- Builds on top of LXC to offer a more modern experience.
- Daemon exposing a REST API.
- Can manage images, snapshots, migrations, networking, storage.
- "offers a user experience similar to virtual machines but using Linux containers instead."

rkt

- Compares to runc.
- No daemon or API.
- Strong emphasis on security (through privilege separation).
- Networking has to be setup separately (e.g. through CNI plugins).
- Partial image management (pull, but no push).

(Image build is handled by separate tools.)

CRI-O

- Designed to be used with Kubernetes as a simple, basic runtime.
- Compares to `containerd`.
- Daemon exposing a gRPC interface.
- Controlled using the CRI API (Container Runtime Interface defined by Kubernetes).
- Needs an underlying OCI runtime (e.g. runc).
- Handles storage, images, networking (through CNI plugins).

We're not aware of anyone using it directly (i.e. outside of Kubernetes).

systemd

- "init" system (PID 1) in most modern Linux distributions.
- Offers tools like `systemd-nspawn` and `machinectl` to manage containers.
- `systemd-nspawn` is "In many ways it is similar to chroot(1), but more powerful".
- `machinectl` can interact with VMs and containers managed by systemd.
- Exposes a DBUS API.
- Basic image support (tar archives and raw disk images).
- Network has to be setup manually.

Overall ...

- The Docker Engine is very developer-centric:
 - easy to install
 - easy to use
 - no manual setup
 - first-class image build and transfer
- As a result, it is a fantastic tool in development environments.
- On servers:
 - Docker is a good default choice
 - If you use Kubernetes, the engine doesn't matter



The container ecosystem

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

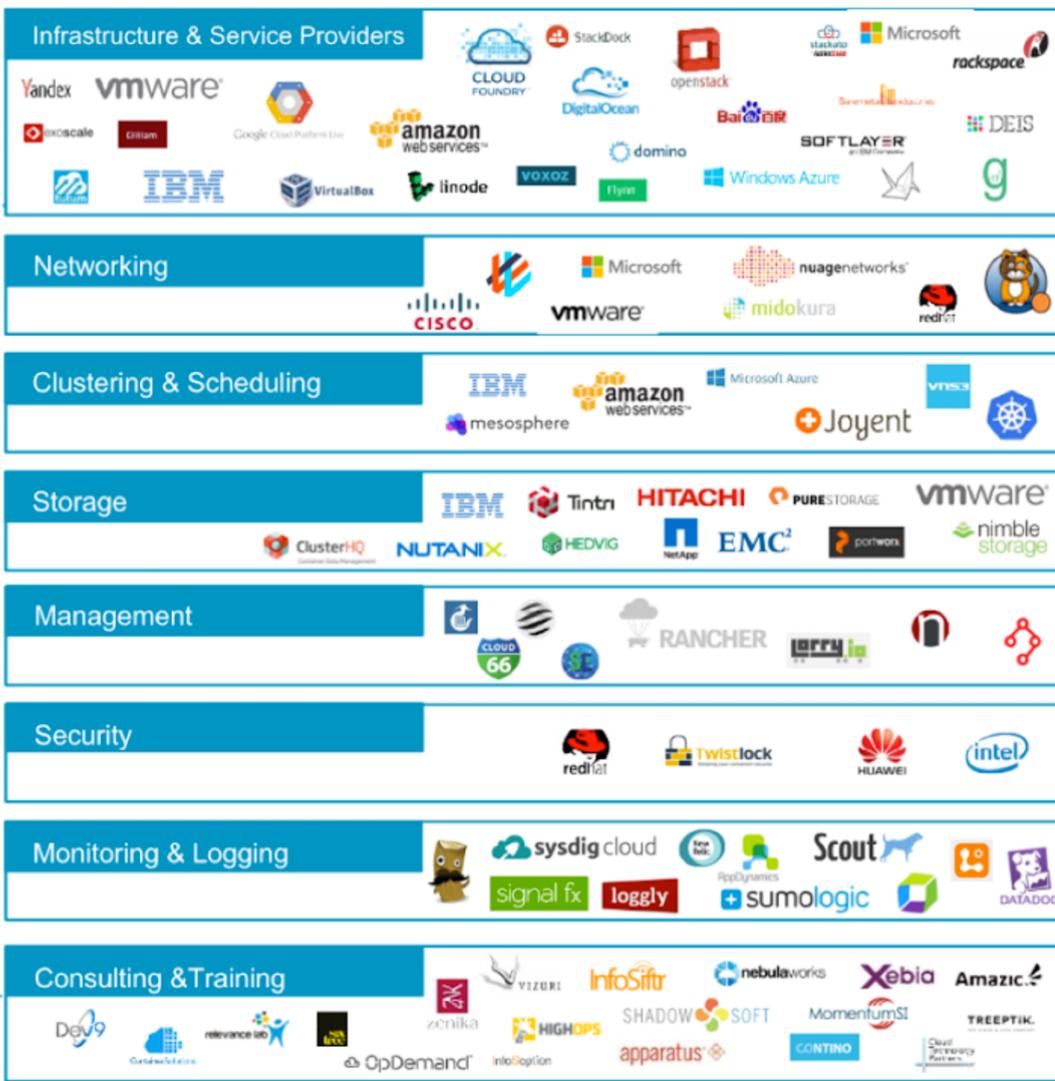
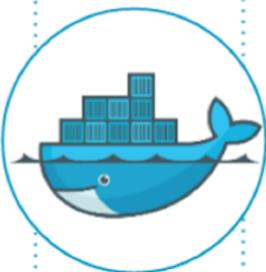
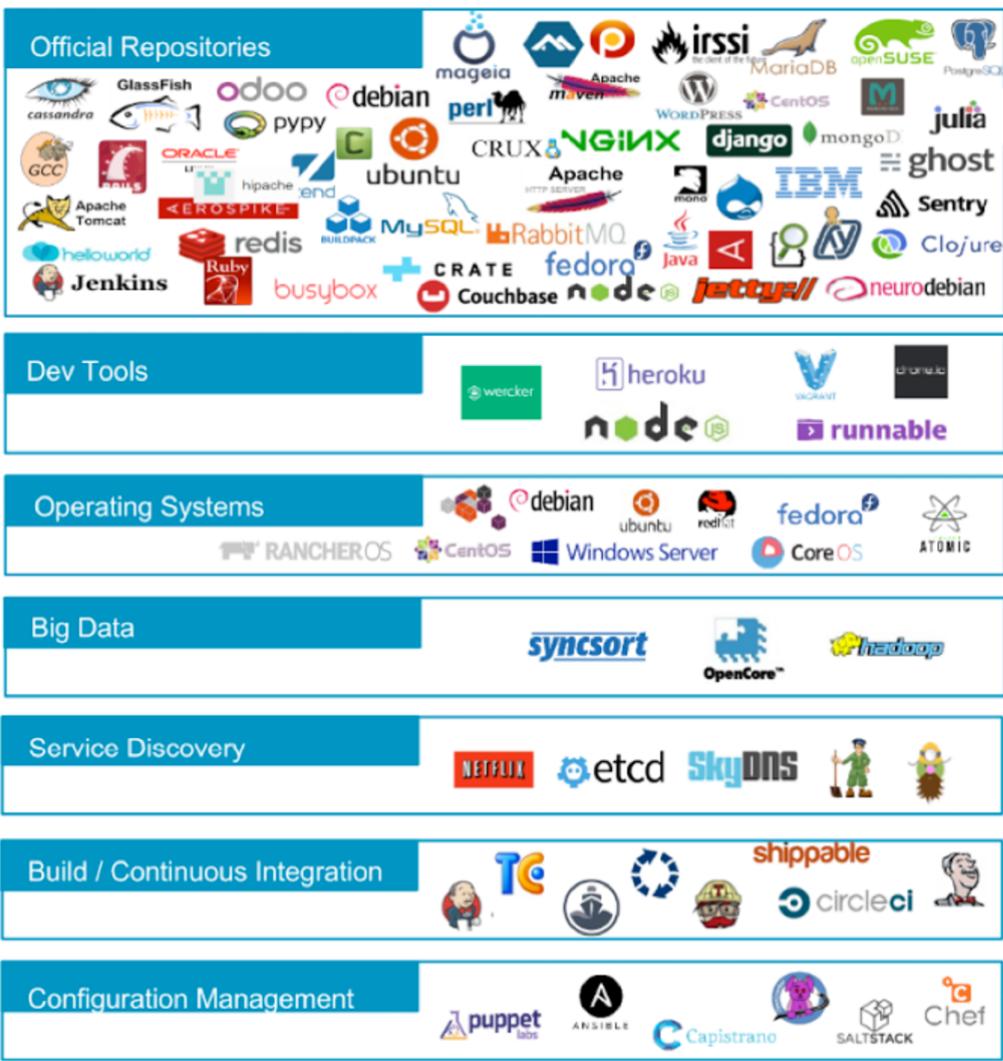
The container ecosystem

In this chapter, we will talk about a few actors of the container ecosystem.

We have (arbitrarily) decided to focus on two groups:

- the Docker ecosystem,
- the Cloud Native Computing Foundation (CNCF) and its projects.

The Docker ecosystem



Moby vs. Docker

- Docker Inc. (the company) started Docker (the open source project).
- At some point, it became necessary to differentiate between:
 - the open source project (code base, contributors...),
 - the product that we use to run containers (the engine),
 - the platform that we use to manage containerized applications,
 - the brand.



Exercise in brand management

Questions:

Exercise in brand management

Questions:

- What is the brand of the car on the previous slide?

Exercise in brand management

Questions:

- What is the brand of the car on the previous slide?
- What kind of engine does it have?

Exercise in brand management

Questions:

- What is the brand of the car on the previous slide?
- What kind of engine does it have?
- Would you say that it's a safe or unsafe car?

Exercise in brand management

Questions:

- What is the brand of the car on the previous slide?
- What kind of engine does it have?
- Would you say that it's a safe or unsafe car?
- Harder question: can you drive from the US West to East coasts with it?

Exercise in brand management

Questions:

- What is the brand of the car on the previous slide?
- What kind of engine does it have?
- Would you say that it's a safe or unsafe car?
- Harder question: can you drive from the US West to East coasts with it?

The answers to these questions are part of the Tesla brand.

What if ...

- The blueprints for Tesla cars were available for free.
- You could legally build your own Tesla.
- You were allowed to customize it entirely.

(Put a combustion engine, drive it with a game pad ...)

- You could even sell the customized versions.

What if ...

- The blueprints for Tesla cars were available for free.
- You could legally build your own Tesla.
- You were allowed to customize it entirely.

(Put a combustion engine, drive it with a game pad ...)

- You could even sell the customized versions.
- ... And call your customized version "Tesla".

What if ...

- The blueprints for Tesla cars were available for free.
- You could legally build your own Tesla.
- You were allowed to customize it entirely.

(Put a combustion engine, drive it with a game pad ...)

- You could even sell the customized versions.
- ... And call your customized version "Tesla".

Would we give the same answers to the questions on the previous slide?

From Docker to Moby

- Docker Inc. decided to split the brand.
- Moby is the open source project.
(= Components and libraries that you can use, reuse, customize, sell ...)
- Docker is the product.
(= Software that you can use, buy support contracts ...)
- Docker is made with Moby.
- When Docker Inc. improves the Docker products, it improves Moby.
(And vice versa.)

Other examples

- *Read the Docs* is an open source project to generate and host documentation.
- You can host it yourself (on your own servers).
- You can also get hosted on readthedocs.org.
- The maintainers of the open source project often receive support requests from users of the hosted product ...
- ... And the maintainers of the hosted product often receive support requests from users of self-hosted instances.
- Another example:

WordPress.com is a blogging platform that is owned and hosted online by Automattic. It is run on WordPress, an open source piece of software used by bloggers. (Wikipedia)

Docker CE vs Docker EE

- Docker CE = Community Edition.
- Available on most Linux distros, Mac, Windows.
- Optimized for developers and ease of use.
- Docker EE = Enterprise Edition.
- Available only on a subset of Linux distros + Windows servers.

(Only available when there is a strong partnership to offer enterprise-class support.)

- Optimized for production use.
- Comes with additional components: security scanning, RBAC ...

The CNCF

- Non-profit, part of the Linux Foundation; founded in December 2015.

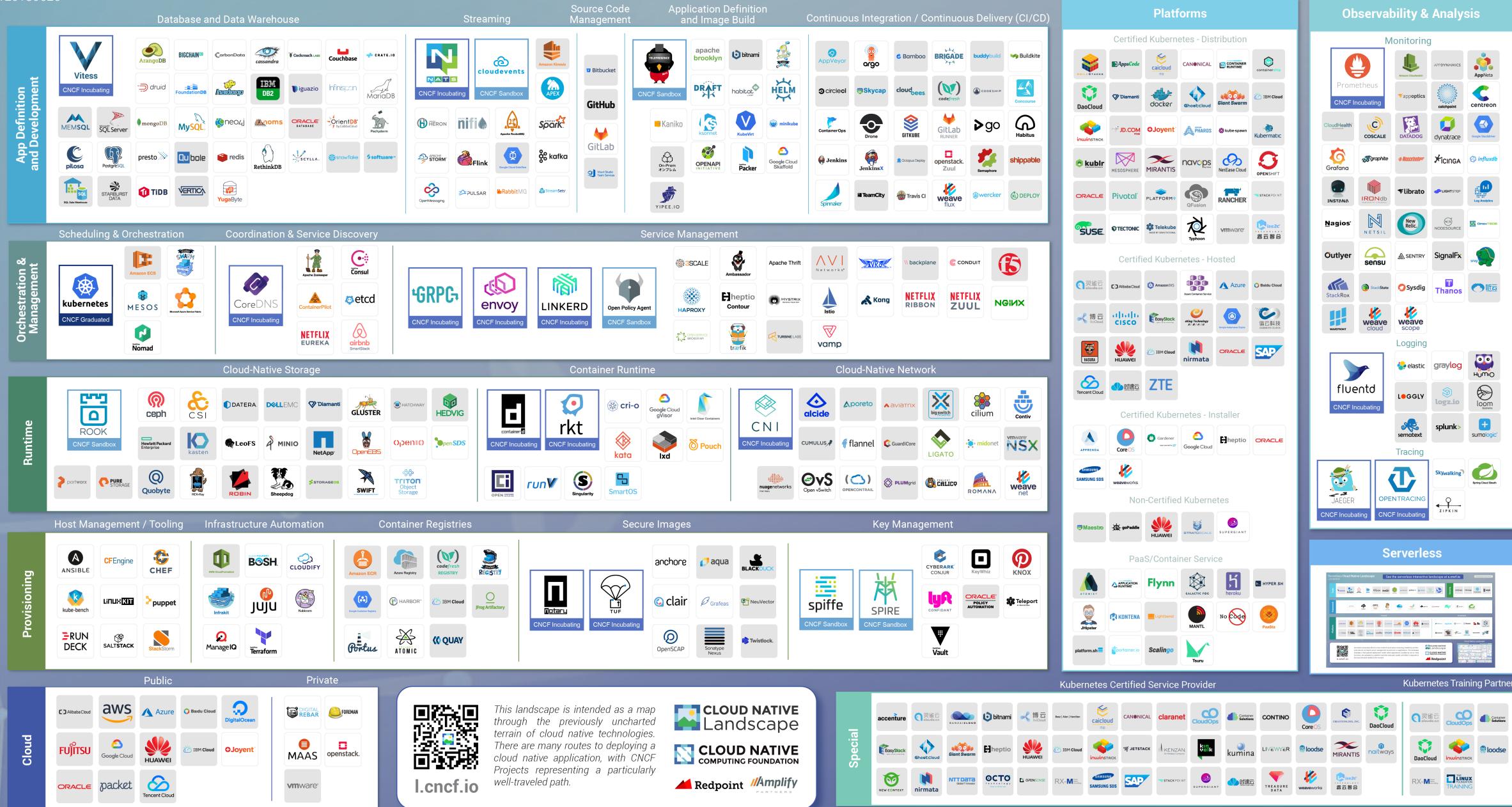
The Cloud Native Computing Foundation builds sustainable ecosystems and fosters a community around a constellation of high-quality projects that orchestrate containers as part of a microservices architecture.

CNCF is an open source software foundation dedicated to making cloud-native computing universal and sustainable.

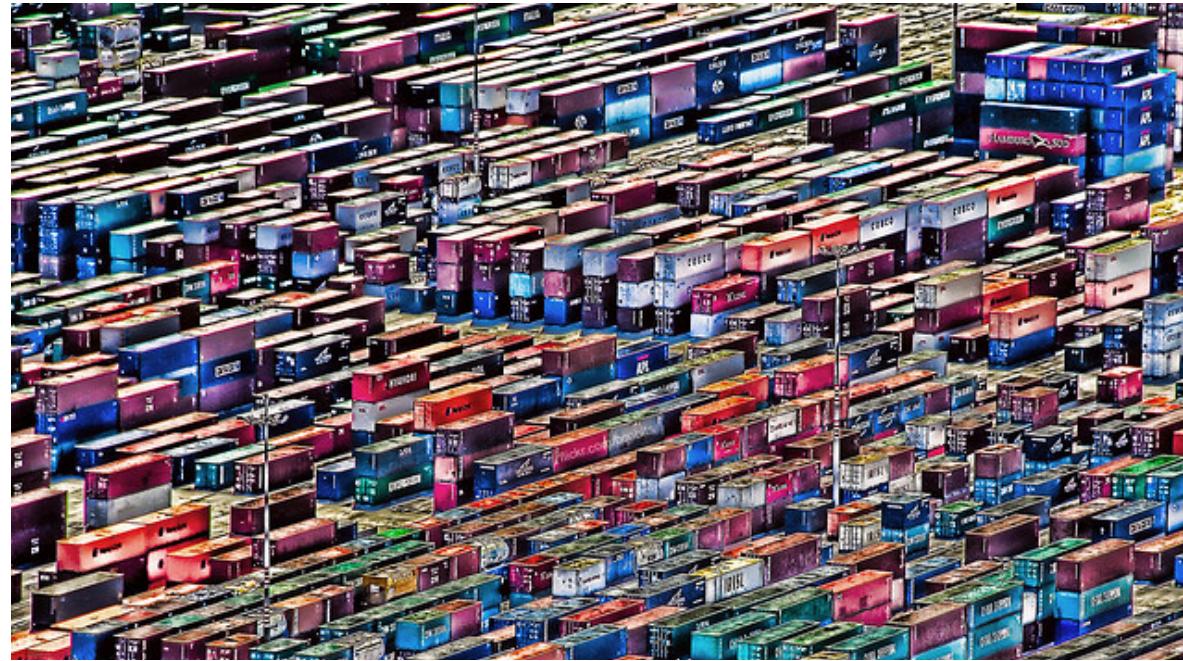
- Home of Kubernetes (and many other projects now).
- Funded by corporate memberships.

See the interactive landscape at l.cncf.io

Cloud Native Landscape



That's all, folks!
Questions?





Links and resources

[Previous section](#) | [Back to table of contents](#) | [Next section](#)

Links and resources

- [Docker Community Slack](#)
- [Docker Community Forums](#)
- [Docker Hub](#)
- [Docker Blog](#)
- [Docker documentation](#)
- [Docker on StackOverflow](#)
- [Docker on Twitter](#)
- [Play With Docker Hands-On Labs](#)

These slides (and future updates) are on → <http://container.training/>