

Docker

Self-paced version

Remarks

- Adapted from the free training initiated by Jérôme Petazzone
- Download [slides.zip](#)
- If you find a typo or a non-working command please contribute to the [slides sources](#)
- [Tutorials](#)
- [keystore-rest-server](#)
- We included as much information as possible for you in these slides, we will skip some

Practical Sessions

- Each of you has 2 VMs: 1 worker, 1 manager
- Use exclusively the manager until swarm tutorial, the last one.
- If you did not receive an e-mail with credentials let me know asap

References

- Docker documentation
- Docker forums
- StackOverflow
- The Docker Book
- freeCodeCamp - The Docker Handbook 2021 Edition

Tutorials

- 11% Install docker
- 25% Basic docker commands
- 29% Working with images
- 51% Building images
- 56% Volumes
- 67% Network
- 72% docker-compose
- 94% Swarm mode

Final demo

- Docker on Ryax CI/CD
- Deployment with kubernetes

Part 1

- Docker 30,000ft overview
- History of containers ... and Docker
- Installing Docker

Part 2

- Our first containers
- Background containers
- Restarting and attaching to containers
- Naming and inspecting containers
- Labels
- Getting inside a container

Part 3

- Understanding Docker images
- Building images interactively
- Building Docker images with a Dockerfile
- `CMD` and `ENTRYPOINT`
- Copying files during the build
- Exercise — writing Dockerfiles

Part 4

- Reducing image size
- Multi-stage builds
- Publishing images to the Docker Hub
- Tips for efficient Dockerfiles
- Dockerfile examples
- Hosting our own registry

Part 5

- Working with volumes

Part 6

- Container networking basics
- Container network drivers
- The Container Network Model
- Service discovery with containers

Part 7

- Compose for development stacks
- Managing hosts with Docker Machine
- Exercise — writing a Compose file

Part 8

- SwarmKit
- Declarative vs imperative
- Swarm mode
- Creating our first Swarm
- Running our first Swarm service
- Swarm Stacks

Part 9

- CI/CD for Docker and orchestration
- Updating services
- Rolling updates
- Health checks and auto-rollback
- Secrets management and encryption at rest



Docker 30,000ft overview

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

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

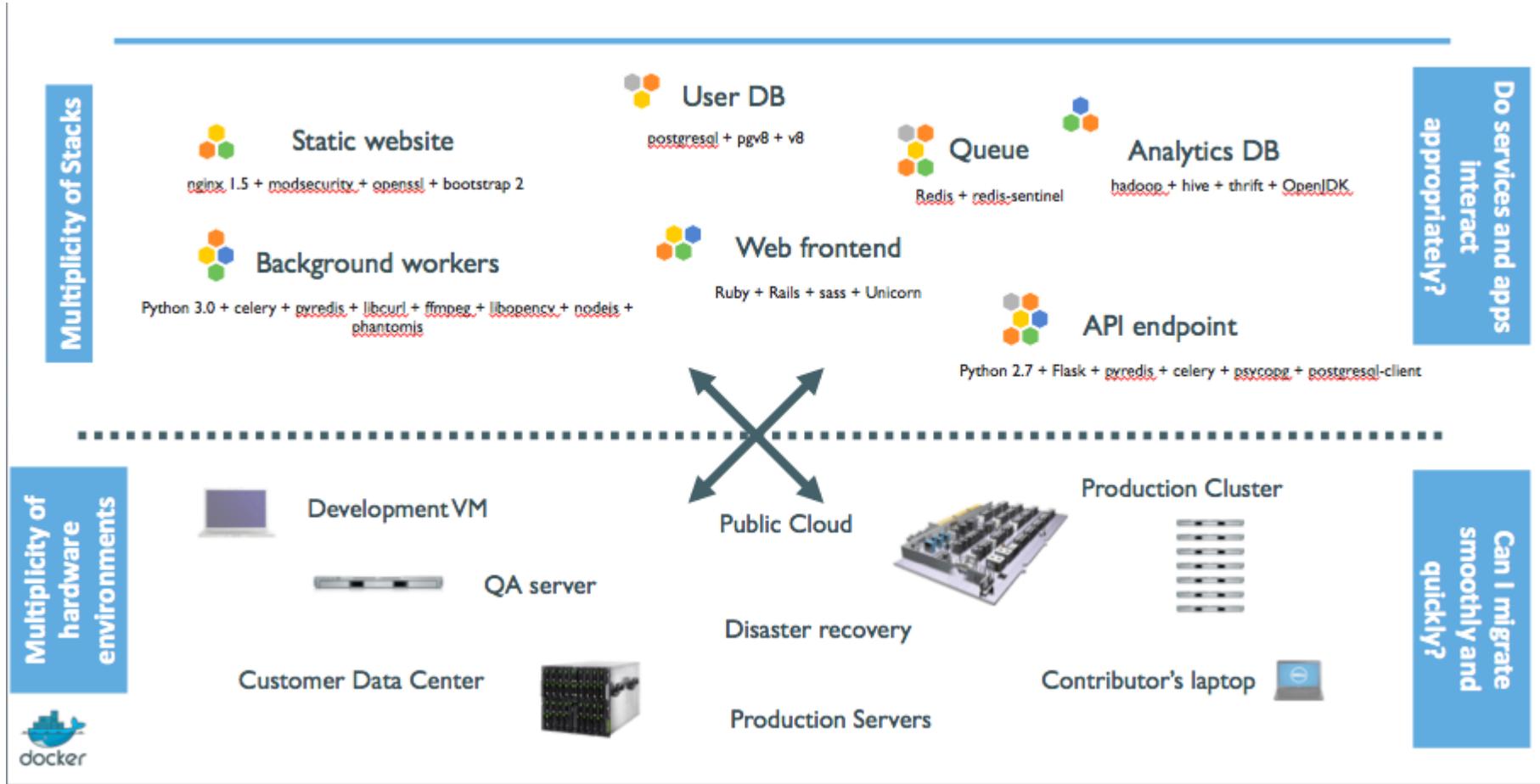
- Now:

- 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

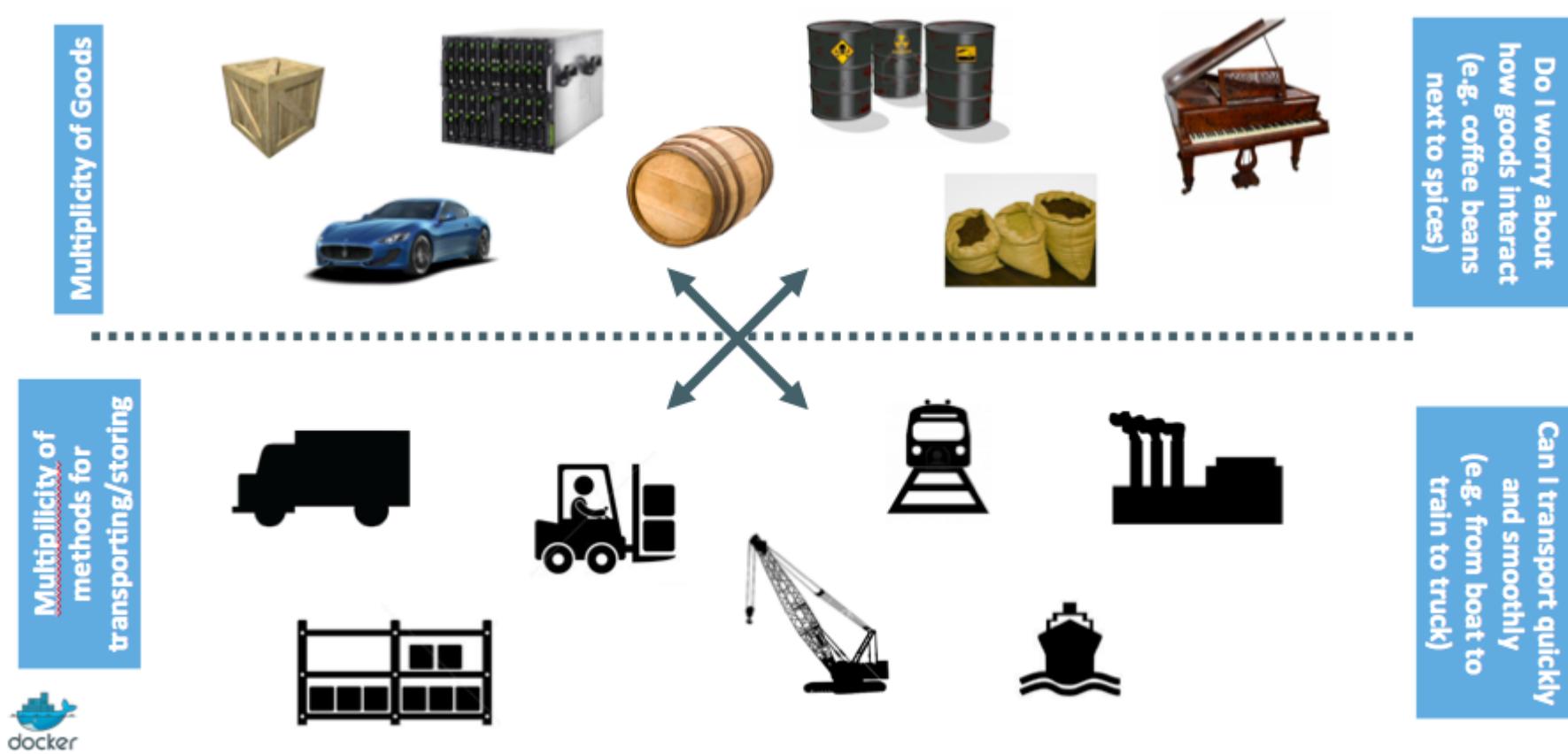


The matrix from hell

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



The parallel with the shipping industry



Intermodal shipping containers



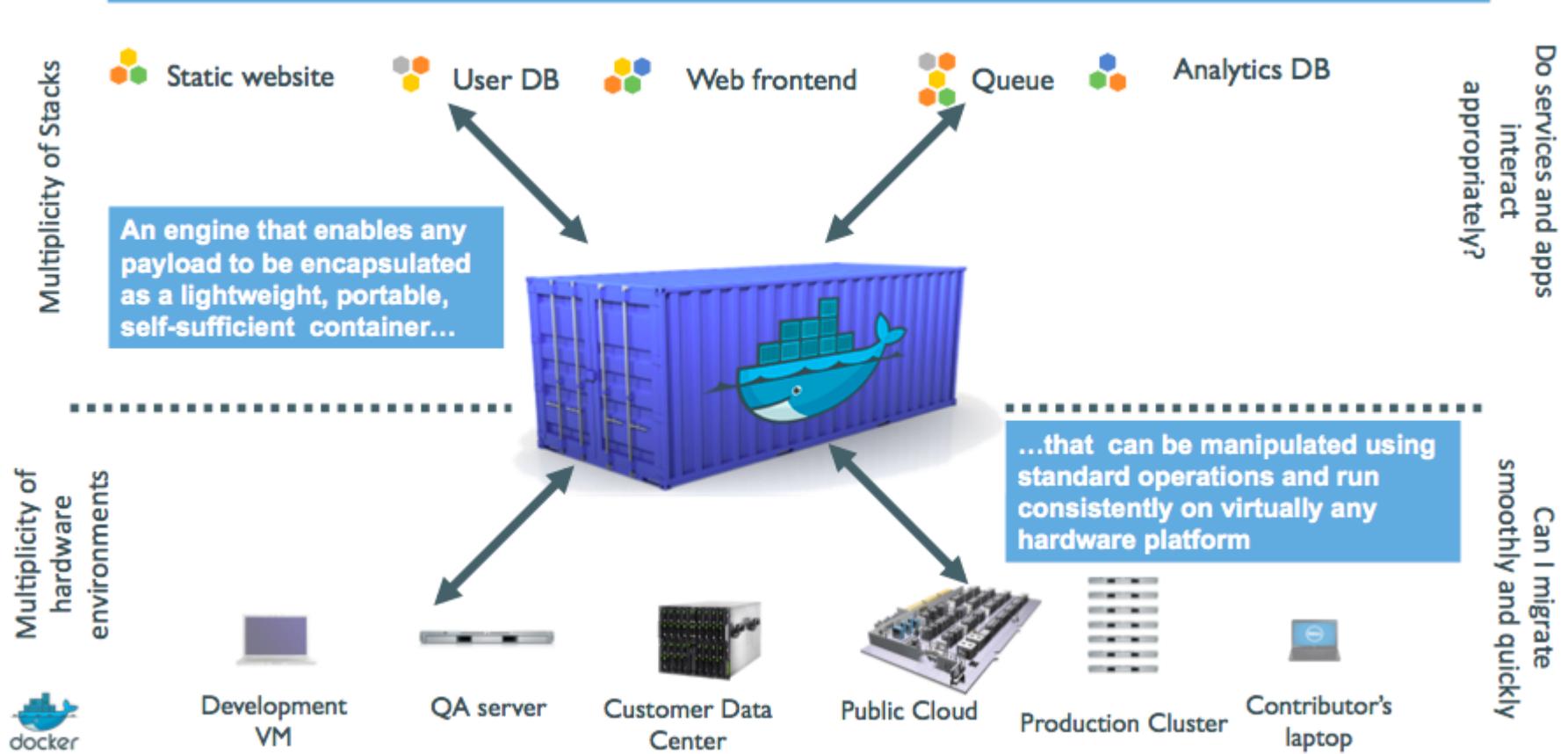
A new shipping ecosystem



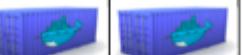
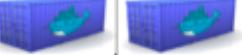
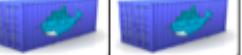
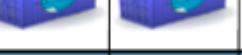
- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)
→ massive globalization
- 5000 ships deliver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



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

History of containers ... and Docker

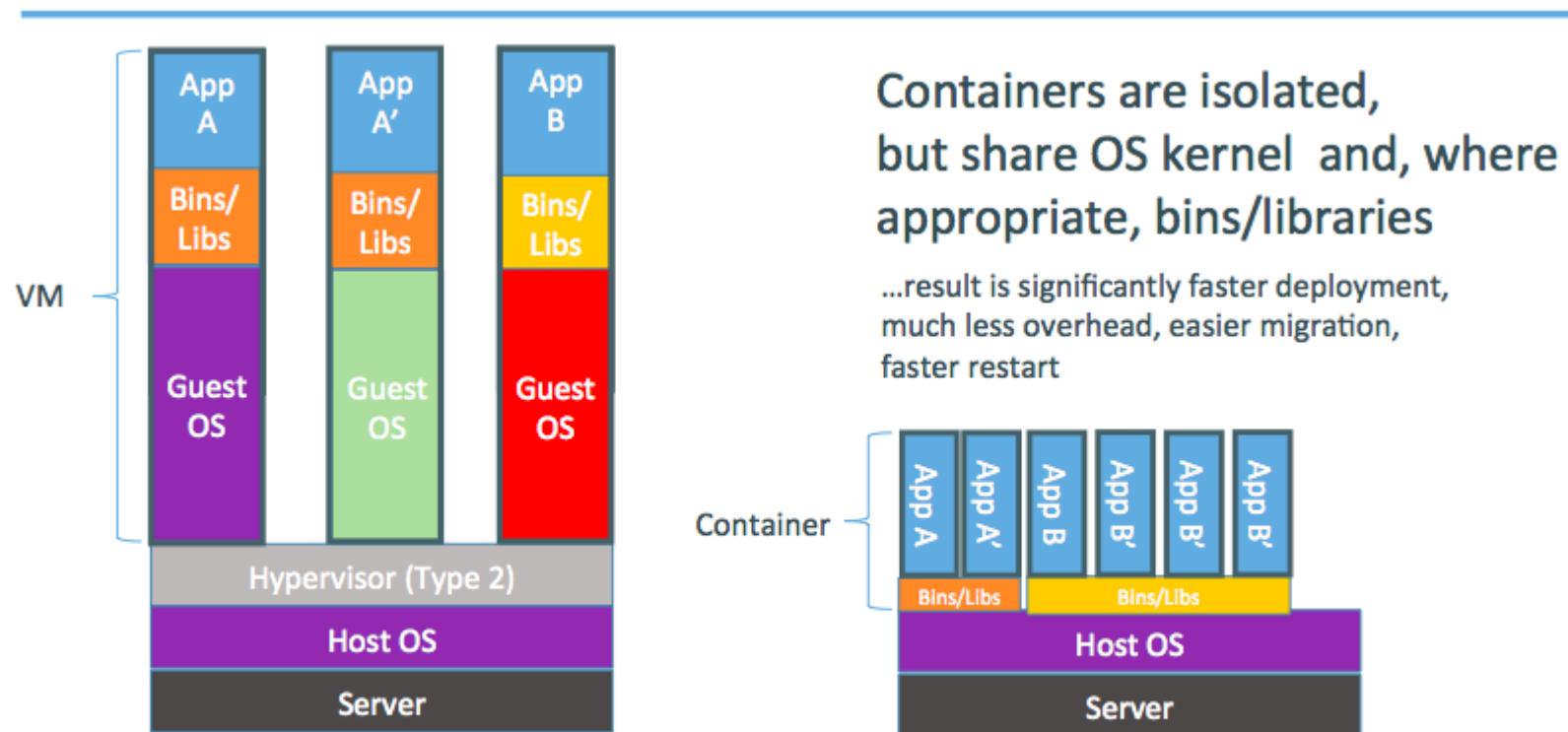
First experimentations

- 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 = cheaper than VMs

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

The PAAS period (2008-2013)

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, 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.

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 ...



Installing Docker

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

Installing Docker



Objectives

At the end of this lesson, you will know:

- How to install Docker.
- When to use `sudo` when running Docker commands.

Note: if you were provided with a training VM for a hands-on tutorial, you can skip this chapter, since that VM already has Docker installed, and Docker has already been setup to run without `sudo`.

Installing Docker

There are many ways to install Docker.

We can arbitrarily distinguish:

- Installing Docker on an existing Linux machine (physical or VM)
- Installing Docker on macOS or Windows
- Installing Docker on a fleet of cloud VMs

Installing Docker on Linux

- The recommended method is to install the packages supplied by Docker Inc :
 - add Docker Inc.'s package repositories to your system configuration
 - install the Docker Engine
- Detailed installation instructions (distro by distro) are available on:

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

- You can also install from binaries (if your distro is not supported):

<https://docs.docker.com/engine/installation/linux/docker-ce/binaries/>

- To quickly setup a dev environment, Docker provides a convenience install script:

```
curl -fsSL get.docker.com | sh
```



Docker Inc. packages vs distribution packages

- Docker Inc. releases new versions monthly (edge) and quarterly (stable)
- Releases are immediately available on Docker Inc.'s package repositories
- Linux distros don't always update to the latest Docker version
 - (Sometimes, updating would break their guidelines for major/minor upgrades)
- Sometimes, some distros have carried packages with custom patches
- Sometimes, these patches added critical security bugs ☹
- Installing through Docker Inc.'s repositories is a bit of extra work ...
 - ... but it is generally worth it!

Installing Docker on macOS and Windows

- On macOS, the recommended method is to use Docker Desktop for Mac:

<https://docs.docker.com/docker-for-mac/install/>

- On Windows 10 Pro, Enterprise, and Education, you can use Docker Desktop for Windows:

<https://docs.docker.com/docker-for-windows/install/>

- On older versions of Windows, you can use the Docker Toolbox:

https://docs.docker.com/toolbox/toolbox_install_windows/

- On Windows Server 2016, you can also install the native engine:

<https://docs.docker.com/install/windows/docker-ee/>

Docker Desktop

- Special Docker edition available for Mac and Windows
- Integrates well with the host OS:
 - installed like normal user applications on the host
 - provides user-friendly GUI to edit Docker configuration and settings
- Only support running one Docker VM at a time ...
... but we can use `docker-machine`, the Docker Toolbox, VirtualBox, etc. to get a cluster.



Docker Desktop internals

- Leverages the host OS virtualization subsystem
(e.g. the [Hypervisor API](#) on macOS)
- Under the hood, runs a tiny VM
(transparent to our daily use)
- Accesses network resources like normal applications
(and therefore, plays better with enterprise VPNs and firewalls)
- Supports filesystem sharing through volumes
(we'll talk about this later)

Running Docker on macOS and Windows

When you execute `docker version` from the terminal:

- the CLI connects to the Docker Engine over a standard socket,
- the Docker Engine is, in fact, running in a VM,
- ... but the CLI doesn't know or care about that,
- the CLI sends a request using the REST API,
- the Docker Engine in the VM processes the request,
- the CLI gets the response and displays it to you.

All communication with the Docker Engine happens over the API.

This will also allow to use remote Engines exactly as if they were local.

Important PSA about security

- If you have access to the Docker control socket, you can take over the machine
(Because you can run containers that will access the machine's resources)
- Therefore, on Linux machines, the `docker` user is equivalent to `root`
- You should restrict access to it like you would protect `root`
- By default, the Docker control socket belongs to the `docker` group
- You can add trusted users to the `docker` group
- Otherwise, you will have to prefix every `docker` command with `sudo`, e.g.:

```
sudo docker version
```



Our first containers

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

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:

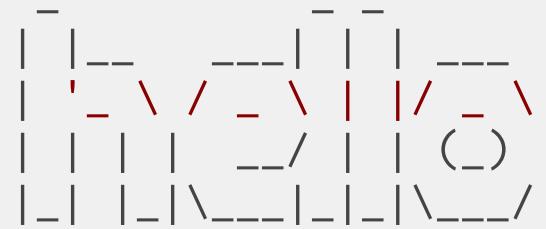
```
root@04c0bb0a6c07:/# apt-get update
...
Fetched 1514 kB in 14s (103 kB/s)
Reading package lists... Done
root@04c0bb0a6c07:/# apt-get install figlet
Reading package lists... Done
...
```

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! 😍

Comparing the container and the host

Exit the container by logging out of the shell, with ^D or exit.

Now try to run figlet. Does that work?

(It shouldn't; except if, by coincidence, you are running on a machine where figlet was installed before.)

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 can only run on Windows hosts; at least for now.)

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.

Where's my container?

- Can we reuse that container that we took time to customize?

We can, but that's not the default workflow with Docker.

- What's the default workflow, then?

Always start with a fresh container.

If we need something installed in our container, build a custom image.

- That seems complicated!

We'll see that it's actually pretty easy!

- And what's the point?

This puts a strong emphasis on automation and repeatability. Let's see why ...

Pets vs. Cattle

- In the "pets vs. cattle" metaphor, there are two kinds of servers.
- Pets:
 - have distinctive names and unique configurations
 - when they have an outage, we do everything we can to fix them
- Cattle:
 - have generic names (e.g. with numbers) and generic configuration
 - configuration is enforced by configuration management, golden images ...
 - when they have an outage, we can replace them immediately with a new server
- What's the connection with Docker and containers?

Local development environments

- When we use local VMs (with e.g. VirtualBox or VMware), our workflow looks like this:
 - create VM from base template (Ubuntu, CentOS...)
 - install packages, set up environment
 - work on project
 - when done, shut down VM
 - next time we need to work on project, restart VM as we left it
 - if we need to tweak the environment, we do it live
- Over time, the VM configuration evolves, diverges.
- We don't have a clean, reliable, deterministic way to provision that environment.

Local development with Docker

- With Docker, the workflow looks like this:
 - create container image with our dev environment
 - run container with that image
 - work on project
 - when done, shut down container
 - next time we need to work on project, start a new container
 - if we need to tweak the environment, we create a new image
- We have a clear definition of our environment, and can share it reliably with others.
- Let's see in the next chapters how to bake a custom image with **figlet!**



Background containers

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

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.

When ^C doesn't work...

Sometimes, ^C won't be enough.

Why? And how can we stop the container in that case?

What happens when we hit ^C

SIGINT gets sent to the container, which means:

- SIGINT gets sent to PID 1 (default case)
- SIGINT gets sent to *foreground processes* when running with -ti

But there is a special case for PID 1: it ignores all signals!

- except SIGKILL and SIGSTOP
- except signals handled explicitly

TL,DR: there are many circumstances when ^C won't stop the container.



Why is PID 1 special?

- PID 1 has some extra responsibilities:
 - it starts (directly or indirectly) every other process
 - when a process exits, its processes are "reparented" under PID 1
- When PID 1 exits, everything stops:
 - on a "regular" machine, it causes a kernel panic
 - in a container, it kills all the processes
- We don't want PID 1 to stop accidentally
- That's why it has these extra protections

How to stop these containers, then?

- Start another terminal and forget about them
(for now!)
- We'll shortly learn about `docker kill`

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.

Extra options

Useful options, more on that later.

- `-v <srcFile>:<dstFile>`: copy local file `<srcFile>` to `<dstFile>` inside the container
- `-p <localPort>:<ctnPort>` : redirect `localPort` to container `ctnPort`

Example of a quick custom webserver:

```
cho "Salut mamie" > ~/index.html
ocker run -dti -p 8080:80 -v $PWD/index.html:/usr/local/apache2/htdocs/index.html httpd
```

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.

(Using `docker ps -lq` is prone to race conditions: what happens if someone else, or another program or script, starts another container just before we run `docker ps -lq`?)

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-second 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 part](#) | [Back to table of contents](#) | [Next part](#)

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 (Linux/macOS)

- 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."

Detaching cont. (Win PowerShell and cmd.exe)

- Docker for Windows has a different detach experience due to shell features.
- `^P^Q` does not work.
- `^C` will detach, rather than stop the container.
- Using Bash, Subsystem for Linux, etc. on Windows behaves like Linux/macOS shells.
- Both PowerShell and Bash work well in Win 10; just be aware of differences.



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

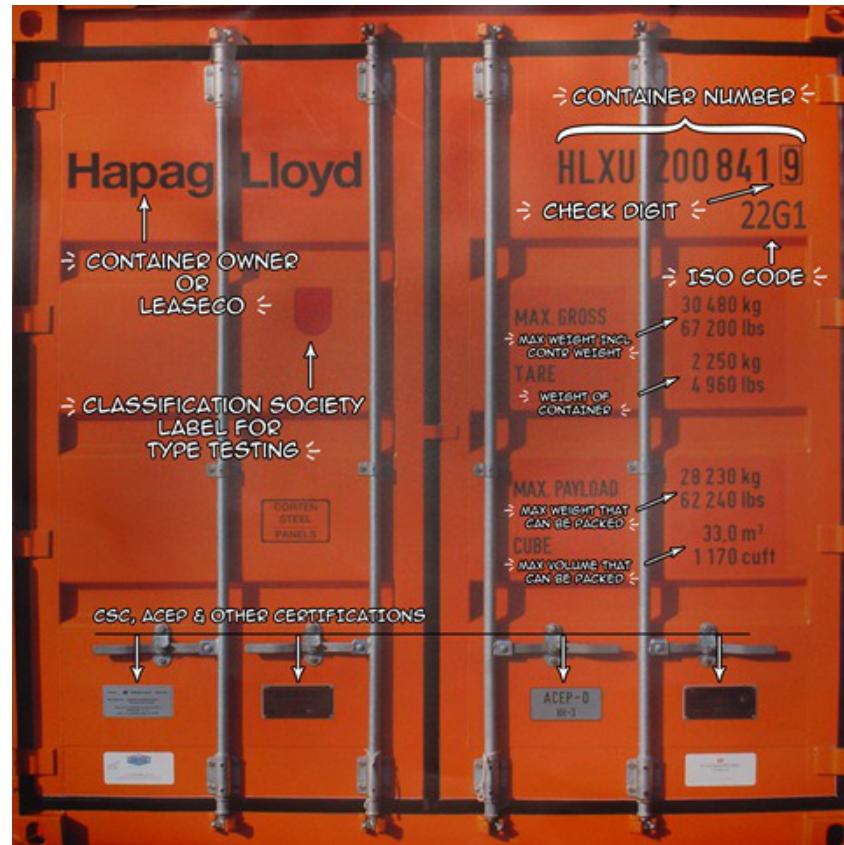
- When you `docker attach`, the Docker Engine sends SIGWINCH signals to the container.
- SIGWINCH = WINdow CHange; indicates a change in window size.
- This will cause some CLI and TUI programs to redraw the screen.
- But not all of them.



Naming and inspecting containers

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

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_curie, 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 part](#) | [Back to table of contents](#) | [Next part](#)

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

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
```

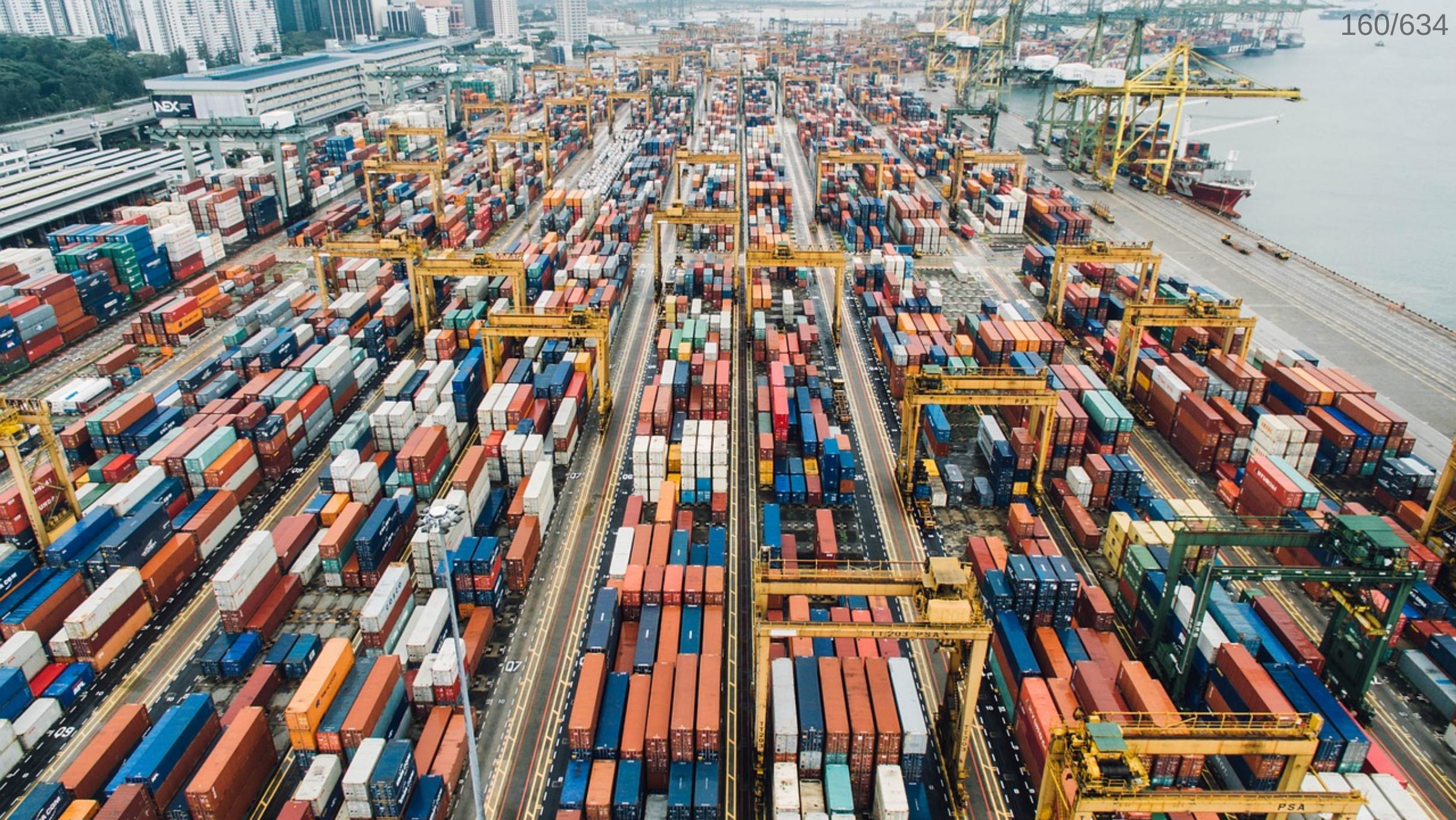


Obtaining a complete dump

- 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
```

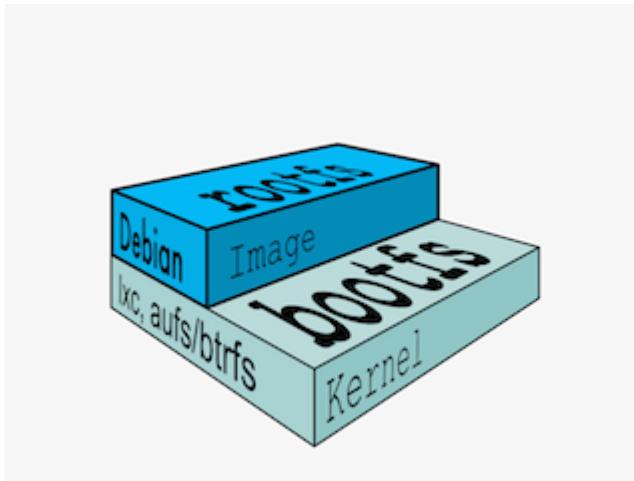
This will give a detailed listing of the content of the container.



Understanding Docker images

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

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.

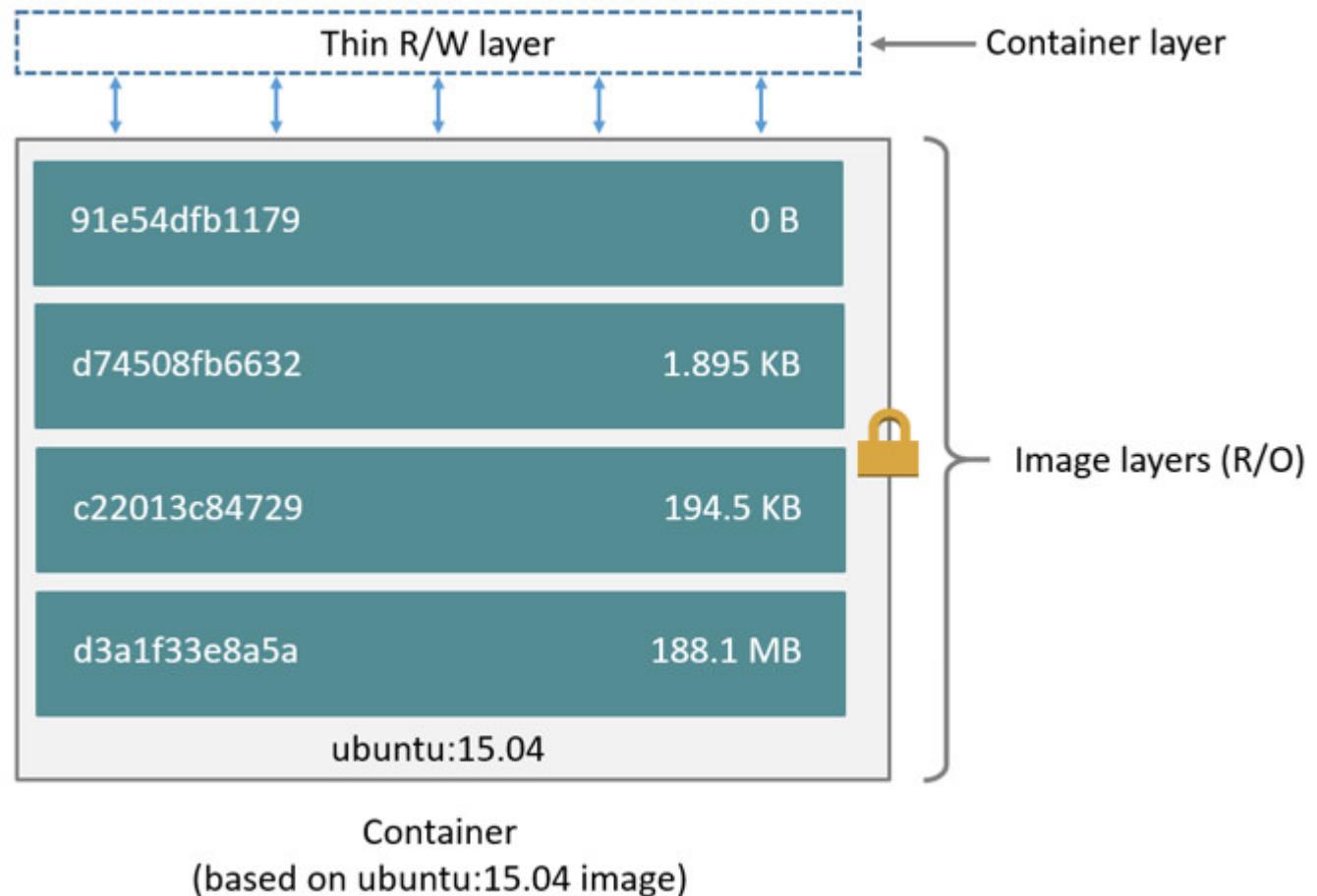
Example for a Java webapp

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

(Note: app config is generally added by orchestration facilities.)

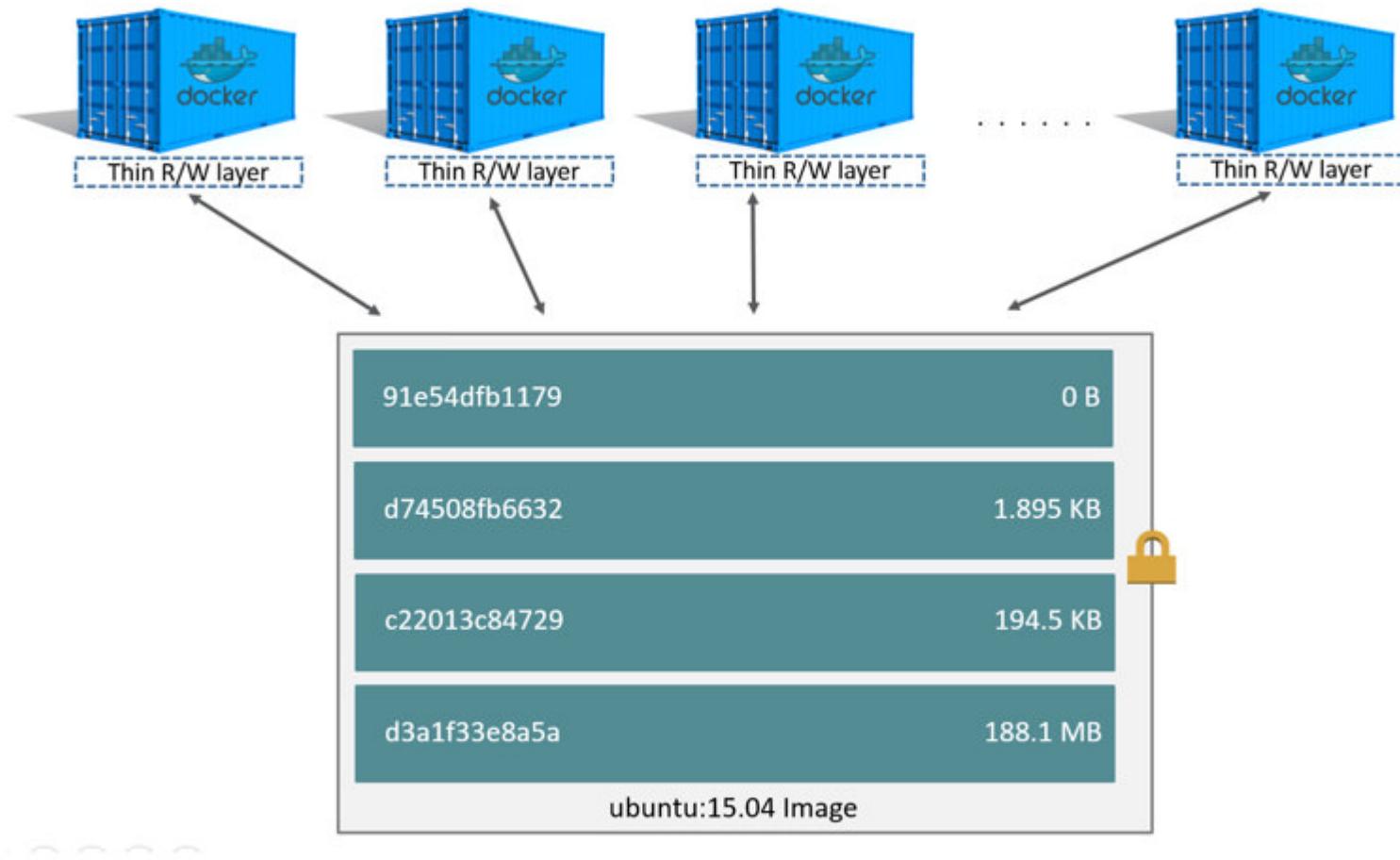
The read-write layer



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.

Multiple containers sharing the same 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 gated by Docker Inc.

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 150 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

Other examples:

```
quay.io/coreos/etcd  
gcr.io/google-containers/hugo
```

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.

This is similar to what we would do with `pip install`, `npm install`, etc.



Multi-arch images

- An image can support multiple architectures
- More precisely, a specific *tag* in a given *repository* can have either:
 - a single *manifest* referencing an image for a single architecture
 - a *manifest list* (or *fat manifest*) referencing multiple images
- In a *manifest list*, each image is identified by a combination of:
 - `os` (linux, windows)
 - `architecture` (amd64, arm, arm64...)
 - optional fields like `variant` (for arm and arm64), `os.version` (for windows)



Working with multi-arch images

- The Docker Engine will pull "native" images when available
(images matching its own os/architecture/variant)
- We can ask for a specific image platform with `--platform`
- The Docker Engine can run non-native images thanks to QEMU+binfmt
(automatically on Docker Desktop; with a bit of setup on Linux)

Section summary

We've learned how to:

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



Building images interactively

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

Building images interactively

In this section, we will create our first container image.

It will be a basic distribution image, but we will pre-install the package `figlet`.

We will:

- Create a container from a base image.
- Install software manually in the container, and turn it into a new image.
- Learn about new commands: `docker commit`, `docker tag`, and `docker diff`.

The plan

1. Create a container (with `docker run`) using our base distro of choice.
2. Run a bunch of commands to install and set up our software in the container.
3. (Optionally) review changes in the container with `docker diff`.
4. Turn the container into a new image with `docker commit`.
5. (Optionally) add tags to the image with `docker tag`.

Setting up our container

Start an Ubuntu container:

```
$ docker run -it ubuntu  
root@<yourContainerId>:/#
```

Run the command `apt-get update` to refresh the list of packages available to install.

Then run the command `apt-get install figlet` to install the program we are interested in.

```
root@<yourContainerId>:/# apt-get update && apt-get install figlet  
.... OUTPUT OF APT-GET COMMANDS ....
```

Inspect the changes

Type `exit` at the container prompt to leave the interactive session.

Now let's run `docker diff` to see the difference between the base image and our container.

```
$ docker diff <yourContainerId>
C /root
A /root/.bash_history
C /tmp
C /usr
C /usr/bin
A /usr/bin/figlet
...
```

Docker tracks filesystem changes

As explained before:

- An image is read-only.
- When we make changes, they happen in a copy of the image.
- Docker can show the difference between the image, and its copy.
- For performance, Docker uses copy-on-write systems.
(i.e. starting a container based on a big image doesn't incur a huge copy.)

Copy-on-write security benefits

- `docker diff` gives us an easy way to audit changes
(à la Tripwire)
- Containers can also be started in read-only mode
(their root filesystem will be read-only, but they can still have read-write data volumes)

Commit our changes into a new image

The `docker commit` command will create a new layer with those changes, and a new image using this new layer.

```
$ docker commit <yourContainerId>  
<newImageId>
```

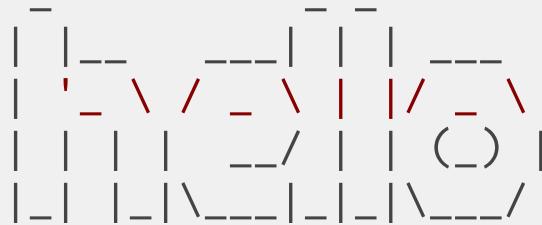
The output of the `docker commit` command will be the ID for your newly created image.

We can use it as an argument to `docker run`.

Testing our new image

Let's run this image:

```
$ docker run -it <newImageId>
root@fcfb62f0bfde:/# figlet hello
```



It works! 🎉

Tagging images

Referring to an image by its ID is not convenient. Let's tag it instead.

We can use the `tag` command:

```
$ docker tag <newImageId> figlet
```

But we can also specify the tag as an extra argument to `commit`:

```
$ docker commit <containerId> figlet
```

And then run it using its tag:

```
$ docker run -it figlet
```

Backup/Restore images

You can backup restore images using `docker save` and `docker load` commands.

This is not the recommended way for continuous integration.

```
$ docker save figlet > figlet.tgz
```

To restore the image use:

```
$ docker load < figlet.tgz
```

What's next?

Manual process = bad.

Automated process = good.

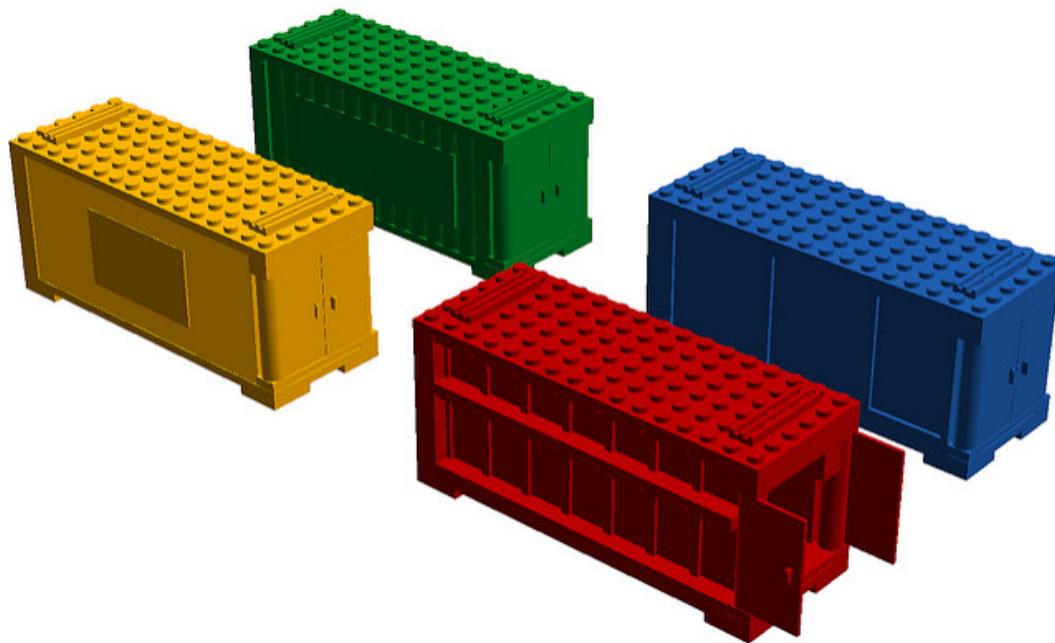
In the next chapter, we will learn how to automate the build process by writing a [Dockerfile](#).



Building Docker images with a Dockerfile

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

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?

It depends if we're using BuildKit or not!

If there are lots of blue lines and the first line looks like this:

```
[+] Building 1.8s (4/6)
```

... then we're using BuildKit.

If the output is mostly black-and-white and the first line looks like this:

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

... then we're using the "classic" or "old-style" builder.

To BuildKit or Not To BuildKit

Classic builder:

- copies the whole "build context" to the Docker Engine
- linear (processes lines one after the other)
- requires a full Docker Engine

BuildKit:

- only transfers parts of the "build context" when needed
- will parallelize operations (when possible)
- can run in non-privileged containers (e.g. on Kubernetes)

With the classic builder

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.
- You can speed up the process with a `.dockerignore` file
 - It tells docker to ignore specific files in the directory
 - Only ignore files that you won't need in the build context!

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.

With BuildKit

```
[+] Building 7.9s (7/7) FINISHED
=> [internal] load build definition from Dockerfile                               0.0s
=> => transferring dockerfile: 98B                                              0.0s
=> [internal] load .dockerignore                                               0.0s
=> => transferring context: 2B                                                 0.0s
=> [internal] load metadata for docker.io/library/ubuntu:latest                1.2s
=> [1/3] FROM docker.io/library/ubuntu@sha256:cf31af331f38d1d7158470e095b132acd126a7180a54f263d386 3.2s
=> => resolve docker.io/library/ubuntu@sha256:cf31af331f38d1d7158470e095b132acd126a7180a54f263d386 0.0s
=> => sha256:cf31af331f38d1d7158470e095b132acd126a7180a54f263d386da88eb681d93 1.20kB / 1.20kB   0.0s
=> => sha256:1de4c5e2d8954bf5fa9855f8b4c9d3c3b97d1d380efe19f60f3e4107a66f5cae 943B / 943B      0.0s
=> => sha256:6a98cbe39225dadebc当地04e21dbe5900ad604739b07a9fa351dd10a6ebad4c1b 3.31kB / 3.31kB    0.0s
=> => sha256:80bc30679ac1fd798f3241208c14accd6a364cb8a6224d1127dfb1577d10554f 27.14MB / 27.14MB  2.3s
=> => sha256:9bf18fab4cfbf479fa9f8409ad47e2702c63241304c2cdd4c33f2a1633c5f85e 850B / 850B     0.5s
=> => sha256:5979309c983a2adeff352538937475cf961d49c34194fa2aab142effe19ed9c1 189B / 189B    0.4s
=> => extracting sha256:80bc30679ac1fd798f3241208c14accd6a364cb8a6224d1127dfb1577d10554f 0.7s
=> => extracting sha256:9bf18fab4cfbf479fa9f8409ad47e2702c63241304c2cdd4c33f2a1633c5f85e 0.0s
=> => extracting sha256:5979309c983a2adeff352538937475cf961d49c34194fa2aab142effe19ed9c1 0.0s
=> [2/3] RUN apt-get update                                         2.5s
=> [3/3] RUN apt-get install figlet                                0.9s
=> exporting to image                                              0.1s
=> => exporting layers                                             0.1s
=> => writing image sha256:3b8aee7b444ab775975dfba691a72d8ac24af2756e0a024e056e3858d5a23f7c 0.0s
=> => naming to docker.io/library/figlet                            0.0s
```

Understanding BuildKit output

- BuildKit transfers the Dockerfile and the *build context*
(these are the first two [internal] stages)
- Then it executes the steps defined in the Dockerfile
([1/3], [2/3], [3/3])
- Finally, it exports the result of the build
(image definition + collection of layers)



BuildKit plain output

- When running BuildKit in e.g. a CI pipeline, its output will be different
- We can see the same output format by using `--progress=plain`

The caching system

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

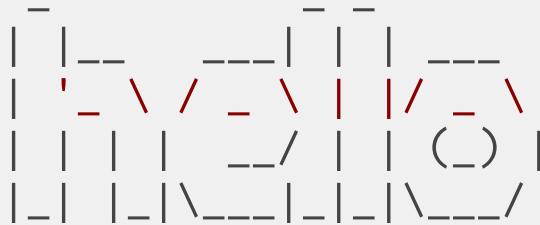
- After each build step, Docker takes a snapshot of the resulting image.
- Before executing a step, Docker checks if it has already built the same sequence.
- Docker uses the exact strings defined in your Dockerfile, so:
 - `RUN apt-get install figlet cowsay`
is different from
`RUN apt-get install cowsay 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



Why sh -c?

- On UNIX, to start a new program, we need two system calls:
 - `fork()`, to create a new child process;
 - `execve()`, to replace the new child process with the program to run.
- Conceptually, `execve()` works like this:

```
execve(program, [list, of, arguments])
```

- When we run a command, e.g. `ls -l /tmp`, something needs to parse the command.
(i.e. split the program and its arguments into a list.)
- The shell is usually doing that.
(It also takes care of expanding environment variables and special things like `~`.)



Why sh -c?

- When we do `RUN ls -l /tmp`, the Docker builder needs to parse the command.
- Instead of implementing its own parser, it outsources the job to the shell.
- That's why we see `sh -c ls -l /tmp` in that case.
- But we can also do the parsing jobs ourselves.
- This means passing `RUN` a list of arguments.
- This is called the *exec syntax*.

Shell syntax vs exec syntax

Dockerfile commands that execute something can have two forms:

- plain string, or *shell syntax*:

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

- JSON list, or *exec syntax*:

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

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

Using exec 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 .
```

History with exec 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

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

When to use exec syntax and shell syntax

- shell 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

- exec 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

Pro-tip: the `exec` shell built-in

POSIX shells have a built-in command named `exec`.

`exec` should be followed by a program and its arguments.

From a user perspective:

- it looks like the shell exits right away after the command execution,
- in fact, the shell exits just *before* command execution;
- or rather, the shell gets *replaced* by the command.

Example using exec

```
CMD exec figlet -f script hello
```

In this example, `sh -c` will still be used, but `figlet` will be PID 1 in the container.

The shell gets replaced by `figlet` when `figlet` starts execution.

This allows to run processes as PID 1 without using JSON.



CMD and ENTRYPOINT

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

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.

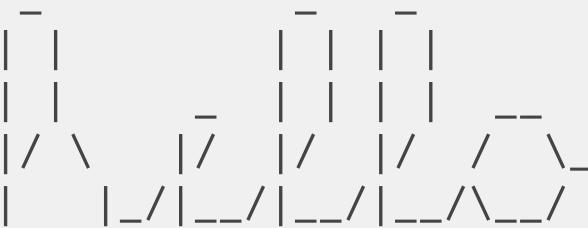
Build and test our image

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
```

```
      _  
     || |  
 ,---, | / | -|_  
 / \_/_|_|/_/ \_/_/|_/_/
```

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
```

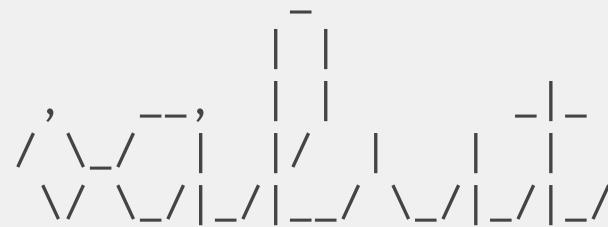
Build and test our image

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 myfiglet .  
...  
Successfully built 6e0b6a048a07  
Successfully tagged myfiglet:latest
```

Run it without parameters:

```
$ docker run myfiglet
```



Overriding the image default parameters

Now let's pass extra arguments to the image.

```
$ docker run myfiglet hola mundo
```



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

Overriding ENTRYPOINT

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

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

We use the `--entrypoint` parameter:

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

CMD and ENTRYPOINT recap

- `docker run myimage` executes `ENTRYPOINT + CMD`
- `docker run myimage args` executes `ENTRYPOINT + args` (overriding `CMD`)
- `docker run --entrypoint prog myimage` executes `prog` (overriding both)

Command	ENTRYPOINT	CMD	Result
<code>docker run figlet</code>	none	none	Use values from base image (bash)
<code>docker run figlet hola</code>	none	none	Error (executable <code>hola</code> not found)
<code>docker run figlet</code>	<code>figlet -f script</code>	none	<code>figlet -f script</code>
<code>docker run figlet hola</code>	<code>figlet -f script</code>	none	<code>figlet -f script hola</code>
<code>docker run figlet</code>	none	<code>figlet -f script</code>	<code>figlet -f script</code>
<code>docker run figlet hola</code>	none	<code>figlet -f script</code>	Error (executable <code>hola</code> not found)
<code>docker run figlet</code>	<code>figlet -f script</code>	<code>hello</code>	<code>figlet -f script hello</code>
<code>docker run figlet hola</code>	<code>figlet -f script</code>	<code>hello</code>	<code>figlet -f script hola</code>

When to use ENTRYPPOINT vs CMD

ENTRYPOINT is great for "containerized binaries".

Example: docker run consul --help

(Pretend that the docker run part isn't there!)

CMD is great for images with multiple binaries.

Example: docker run busybox ifconfig

(It makes sense to indicate *which* program we want to run!)



Copying files during the build

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

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

- We can `COPY` whole directories recursively
- It is possible to do e.g. `COPY . .`
(but it might require some extra precautions to avoid copying too much)
- In older Dockerfiles, you might see the `ADD` command; consider it deprecated
(it is similar to `COPY` 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



.dockerignore

- We can create a file named `.dockerignore`
(at the top-level of the build context)
- It can contain file names and globs to ignore
- They won't be sent to the builder
(and won't end up in the resulting image)
- See the [documentation](#) for the little details
(exceptions can be made with `!`, multiple directory levels with `**...`)



Exercise — writing Dockerfiles

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

Exercise — writing Dockerfiles

Let's write Dockerfiles for an existing application!

1. Check out the code repository
2. Read all the instructions
3. Write Dockerfiles
4. Build and test them individually

Code repository

Clone the repository available at:

<https://github.com/jpetazzo/wordsmith>

It should look like this:

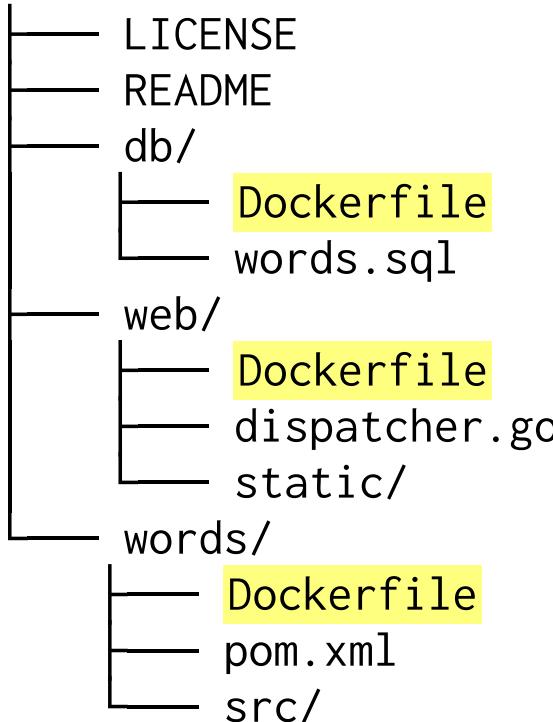
```
├── LICENSE
├── README
└── db/
    └── words.sql
├── web/
    ├── dispatcher.go
    └── static/
└── words/
    ├── pom.xml
    └── src/
```

Instructions

The repository contains instructions in English and French.

For now, we only care about the first part (about writing Dockerfiles).

Place each Dockerfile in its own directory, like this:



Build and test

Build and run each Dockerfile individually.

For db, we should be able to see some messages confirming that the data set was loaded successfully (some INSERT lines in the container output).

For web and words, we should be able to see some message looking like "server started successfully".

That's all we care about for now!

Bonus question: make sure that each container stops correctly when hitting Ctrl-C.



Reducing image size

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

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

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."

Build targets

- We can also tag an intermediary stage with the following command:

```
docker build --target STAGE --tag NAME
```

- This will create an image (named `NAME`) corresponding to stage `STAGE`
- This can be used to easily access an intermediary stage for inspection
(instead of parsing the output of `docker build` to find out the image ID)
- This can also be used to describe multiple images from a single Dockerfile
(instead of using multiple Dockerfiles, which could go out of sync)



Publishing images to the Docker Hub

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

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/>) and sign-in. Select "Repositories" in the blue navigation menu.
- Select "Create" in the top-right bar, and select "Create Repository+".
- Connect your Docker Hub account to your GitHub account.
- Click "Create" button.
- Then go to "Builds" folder.
- Click on Github icon and select your user and the repository that we just forked.
- In "Build rules" block near page bottom, put /www in "Build Context" column (or whichever directory the Dockerfile is in).
- Click "Save and Build" to build the repository immediately (without waiting for a git push).
- Subsequent builds will happen automatically, thanks to GitHub hooks.

Building on the fly

- Some services can build images on the fly from a repository
- Example: [ctr.run](#)



- Use `ctr.run` to automatically build a container image and run it:

```
docker run ctr.run/github.com/undefinedlabs/hello-world
```

There might be a long pause before the first layer is pulled, because the API behind `docker pull` doesn't allow to stream build logs, and there is no feedback during the build.

It is possible to view the build logs by setting up an account on [ctr.run](#).



Tips for efficient Dockerfiles

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

Tips for efficient Dockerfiles

We will see how to:

- Reduce the number of layers.
- Leverage the build cache so that builds can be faster.
- Embed unit testing in the build process.

Reducing the number of layers

- Each line in a Dockerfile creates a new layer.
- Build your Dockerfile to take advantage of Docker's caching system.
- Combine commands by using && to continue commands and \ to wrap lines.

Note: it is frequent to build a Dockerfile line by line:

```
RUN apt-get install thisthing  
RUN apt-get install andthatthing andthatotherone  
RUN apt-get install somemorestuff
```

And then refactor it trivially before shipping:

```
RUN apt-get install thisthing andthatthing andthatotherone somemorestuff
```

Avoid re-installing dependencies at each build

- Classic Dockerfile problem:

"each time I change a line of code, all my dependencies are re-installed!"

- Solution: `COPY` dependency lists (`package.json`, `requirements.txt`, etc.) by themselves to avoid reinstalling unchanged dependencies every time.

Example "bad" Dockerfile

The dependencies are reinstalled every time, because the build system does not know if requirements.txt has been updated.

```
FROM python
WORKDIR /src
COPY . .
RUN pip install -qr requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```

Fixed Dockerfile

Adding the dependencies as a separate step means that Docker can cache more efficiently and only install them when requirements.txt changes.

```
FROM python
WORKDIR /src
COPY requirements.txt .
RUN pip install -qr requirements.txt
COPY . .
EXPOSE 5000
CMD ["python", "app.py"]
```

Be careful with chown, chmod, mv

- Layers cannot store efficiently changes in permissions or ownership.
- Layers cannot represent efficiently when a file is moved either.
- As a result, operations like chown, chmod, mv can be expensive.
- For instance, in the Dockerfile snippet below, each RUN line creates a layer with an entire copy of some-file.

```
COPY some-file .
RUN chown www-data:www-data some-file
RUN chmod 644 some-file
RUN mv some-file /var/www
```

- How can we avoid that?

Put files on the right place

- Instead of using `mv`, directly put files at the right place.
- When extracting archives (tar, zip...), merge operations in a single layer.

Example:

```
...
RUN wget http://.../foo.tar.gz \
&& tar -zxf foo.tar.gz \
&& mv foo/foectl /usr/local/bin \
&& rm -rf foo
...
```

Use COPY --chown

- The Dockerfile instruction `COPY` can take a `--chown` parameter.

Examples:

```
...
COPY --chown=1000 some-file .
COPY --chown=1000:1000 some-file .
COPY --chown=www-data:www-data some-file .
```

- The `--chown` flag can specify a user, or a user:group pair.
- The user and group can be specified as names or numbers.
- When using names, the names must exist in `/etc/passwd` or `/etc/group`.

(In the container, not on the host!)

Set correct permissions locally

- Instead of using `chmod`, set the right file permissions locally.
- When files are copied with `COPY`, permissions are preserved.

Embedding unit tests in the build process

```
FROM <baseimage>
RUN <install dependencies>
COPY <code>
RUN <build code>
RUN <install test dependencies>
COPY <test data sets and fixtures>
RUN <unit tests>
FROM <baseimage>
RUN <install dependencies>
COPY <code>
RUN <build code>
CMD, EXPOSE ...
```

- The build fails as soon as an instruction fails
- If `RUN <unit tests>` fails, the build doesn't produce an image
- If it succeeds, it produces a clean image (without test libraries and data)



Dockerfile examples

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

Dockerfile examples

There are a number of tips, tricks, and techniques that we can use in Dockerfiles.

But sometimes, we have to use different (and even opposed) practices depending on:

- the complexity of our project,
- the programming language or framework that we are using,
- the stage of our project (early MVP vs. super-stable production),
- whether we're building a final image or a base for further images,
- etc.

We are going to show a few examples using very different techniques.

When to optimize an image

When authoring official images, it is a good idea to reduce as much as possible:

- the number of layers,
- the size of the final image.

This is often done at the expense of build time and convenience for the image maintainer; but when an image is downloaded millions of time, saving even a few seconds of pull time can be worth it.

```
RUN apt-get update && apt-get install -y libpng12-dev libjpeg-dev && rm -rf /var/lib/apt/lists/* \
&& docker-php-ext-configure gd --with-png-dir=/usr --with-jpeg-dir=/usr \
&& docker-php-ext-install gd
...
RUN curl -o wordpress.tar.gz -SL https://wordpress.org/wordpress- ${WORDPRESS_UPSTREAM_VERSION}.tar.gz \
&& echo "$WORDPRESS_SHA1 *wordpress.tar.gz" | sha1sum -c - \
&& tar -xzf wordpress.tar.gz -C /usr/src/ \
&& rm wordpress.tar.gz \
&& chown -R www-data:www-data /usr/src/wordpress
```

(Source: [Wordpress official image](#))

When to *not* optimize an image

Sometimes, it is better to prioritize *maintainer convenience*.

In particular, if:

- the image changes a lot,
- the image has very few users (e.g. only 1, the maintainer!),
- the image is built and run on the same machine,
- the image is built and run on machines with a very fast link ...

In these cases, just keep things simple!

(Next slide: a Dockerfile that can be used to preview a Jekyll / github pages site.)

```
FROM debian:sid

RUN apt-get update -q
RUN apt-get install -yq build-essential make
RUN apt-get install -yq zlib1g-dev
RUN apt-get install -yq ruby ruby-dev
RUN apt-get install -yq python-pygments
RUN apt-get install -yq nodejs
RUN apt-get install -yq cmake
RUN gem install --no-rdoc --no-ri github-pages

COPY . /blog
WORKDIR /blog

VOLUME /blog/_site

EXPOSE 4000
CMD ["jekyll", "serve", "--host", "0.0.0.0", "--incremental"]
```

Multi-dimensional versioning systems

Images can have a tag, indicating the version of the image.

But sometimes, there are multiple important components, and we need to indicate the versions for all of them.

This can be done with environment variables:

```
ENV PIP=9.0.3 \
    ZC_BUILDOUT=2.11.2 \
    SETUPTOOLS=38.7.0 \
    PLONE_MAJOR=5.1 \
    PLONE_VERSION=5.1.0 \
    PLONE_MD5=76dc6fcf1c749d763c32ffff3a9870d8d
```

(Source: [Plone official image](#))

Entrypoints and wrappers

It is very common to define a custom entrypoint.

That entrypoint will generally be a script, performing any combination of:

- pre-flights checks (if a required dependency is not available, display a nice error message early instead of an obscure one in a deep log file),
- generation or validation of configuration files,
- dropping privileges (with e.g. `su` or `gosu`, sometimes combined with `chown`),
- and more.

A typical entrypoint script

```
#!/bin/sh
set -e

# first arg is '-f' or '--some-option'
# or first arg is 'something.conf'
if [ "${1#-}" != "$1" ] || [ "${1%.conf}" != "$1" ]; then
    set -- redis-server "$@"
fi

# allow the container to be started with '--user'
if [ "$1" = 'redis-server' -a "$(id -u)" = '0' ]; then
    chown -R redis .
    exec su-exec redis "$0" "$@"
fi

exec "$@"
```

(Source: [Redis official image](#))

Factoring information

To facilitate maintenance (and avoid human errors), avoid to repeat information like:

- version numbers,
- remote asset URLs (e.g. source tarballs) ...

Instead, use environment variables.

```
ENV NODE_VERSION 10.2.1
...
RUN ...
  && curl -fsSLO --compressed "https://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION.tar.xz" \
  && curl -fsSLO --compressed "https://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
  && gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \
  && grep " node-v$NODE_VERSION.tar.xz\$" SHASUMS256.txt | sha256sum -c - \
  && tar -xf "node-v$NODE_VERSION.tar.xz" \
  && cd "node-v$NODE_VERSION" \
...
```

(Source: [Nodejs official image](#))

Overrides

In theory, development and production images should be the same.

In practice, we often need to enable specific behaviors in development (e.g. debug statements).

One way to reconcile both needs is to use Compose to enable these behaviors.

Let's look at the [trainingwheels](#) demo app for an example.

Production image

This Dockerfile builds an image leveraging gunicorn:

```
FROM python
RUN pip install flask
RUN pip install gunicorn
RUN pip install redis
COPY . /src
WORKDIR /src
CMD gunicorn --bind 0.0.0.0:5000 --workers 10 counter:app
EXPOSE 5000
```

(Source: [trainingwheels Dockerfile](#))

Development Compose file

This Compose file uses the same image, but with a few overrides for development:

- the Flask development server is used (overriding `CMD`),
- the `DEBUG` environment variable is set,
- a volume is used to provide a faster local development workflow.

```
services:  
  www:  
    build: www  
    ports:  
      - 8000:5000  
    user: nobody  
    environment:  
      DEBUG: 1  
    command: python counter.py  
    volumes:  
      - ./www:/src
```

(Source: [trainingwheels Compose file](#))

How to know which best practices are better?

- The main goal of containers is to make our lives easier.
- In this chapter, we showed many ways to write Dockerfiles.
- These Dockerfiles use sometimes diametrically opposed techniques.
- Yet, they were the "right" ones *for a specific situation*.
- It's OK (and even encouraged) to start simple and evolve as needed.
- Feel free to review this chapter later (after writing a few Dockerfiles) for inspiration!



Hosting our own registry

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

Hosting our own registry

- We need to run a `registry` container
- It will store images and layers to the local filesystem
(but you can add a config file to use S3, Swift, etc.)
- Docker *requires* TLS when communicating with the registry
 - unless for registries on `127.0.0.0/8` (i.e. `localhost`)
 - or with the Engine flag `--insecure-registry`
- Our strategy: publish the registry container on port 5000,
so that it's available through `127.0.0.1:5000` on each node

Deploying the registry

- We will create a single-instance service, publishing its port on the whole cluster



- Create the registry service:

```
docker service create --name registry --publish 5000:5000 registry
```

- Now try the following command; it should return {"repositories":[]}:

```
curl 127.0.0.1:5000/v2/_catalog
```

Testing our local registry

- We can retag a small image, and push it to the registry



- Make sure we have the busybox image, and retag it:

```
docker pull busybox
docker tag busybox 127.0.0.1:5000/busybox
```

- Push it:

```
docker push 127.0.0.1:5000/busybox
```

Checking what's on our local registry

- The registry API has endpoints to query what's there



- Ensure that our busybox image is now in the local registry:

```
curl http://127.0.0.1:5000/v2/_catalog
```

The curl command should now output:

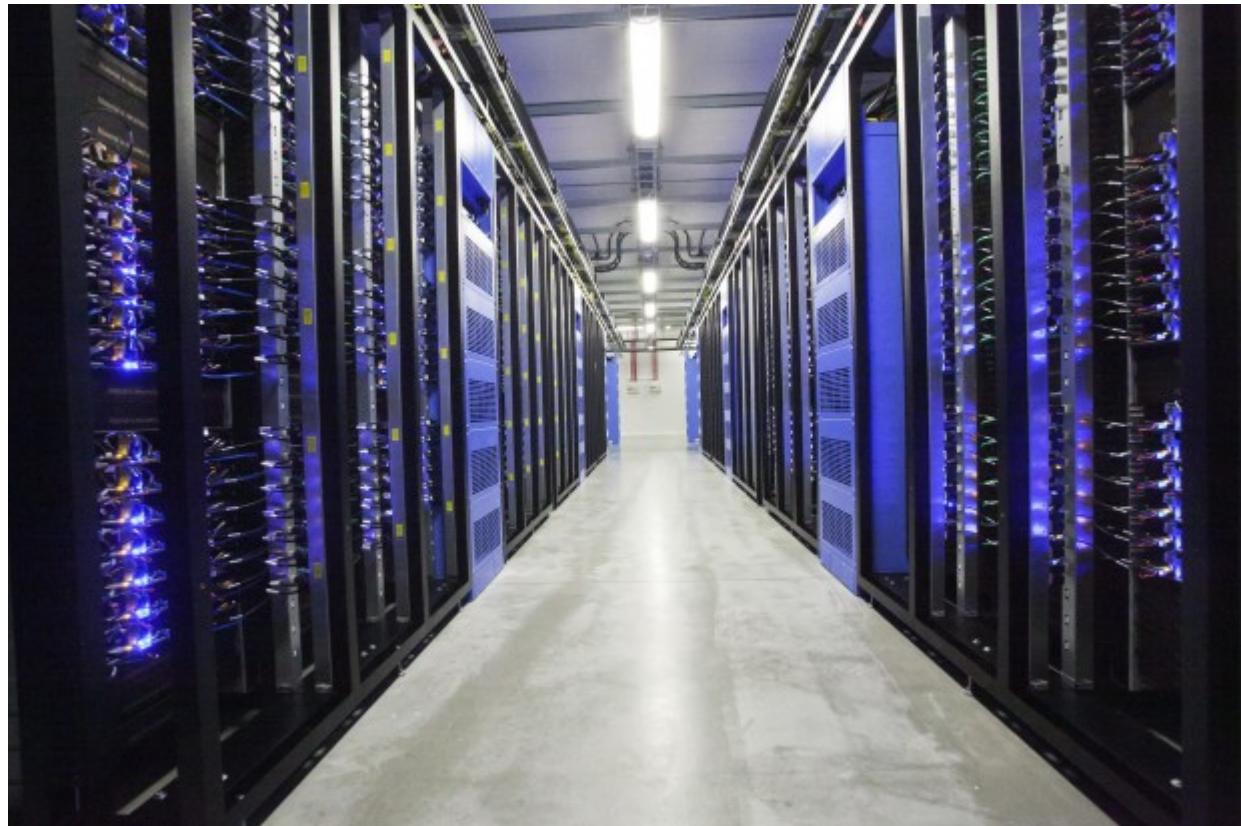
```
{"repositories":["busybox"]}
```



Working with volumes

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

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.

Populating volumes

- When an empty volume is mounted on a non-empty directory, the directory is copied to the volume.
- This makes it easy to "promote" a normal directory to a volume.
- Non-empty volumes are always mounted as-is.

Let's populate the webapps volume with the webapps.dist directory from the Tomcat image.

```
$ docker run -v webapps:/usr/local/tomcat/webapps.dist tomcat true
```

Note: running `true` will cause the container to exit successfully once the `webapps.dist` directory has been copied to the `webapps` volume, instead of starting `tomcat`.

Using our named volumes

- Volumes are used with the `-v` option.
- When a host path does not contain a `/`, it is considered 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 an 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:

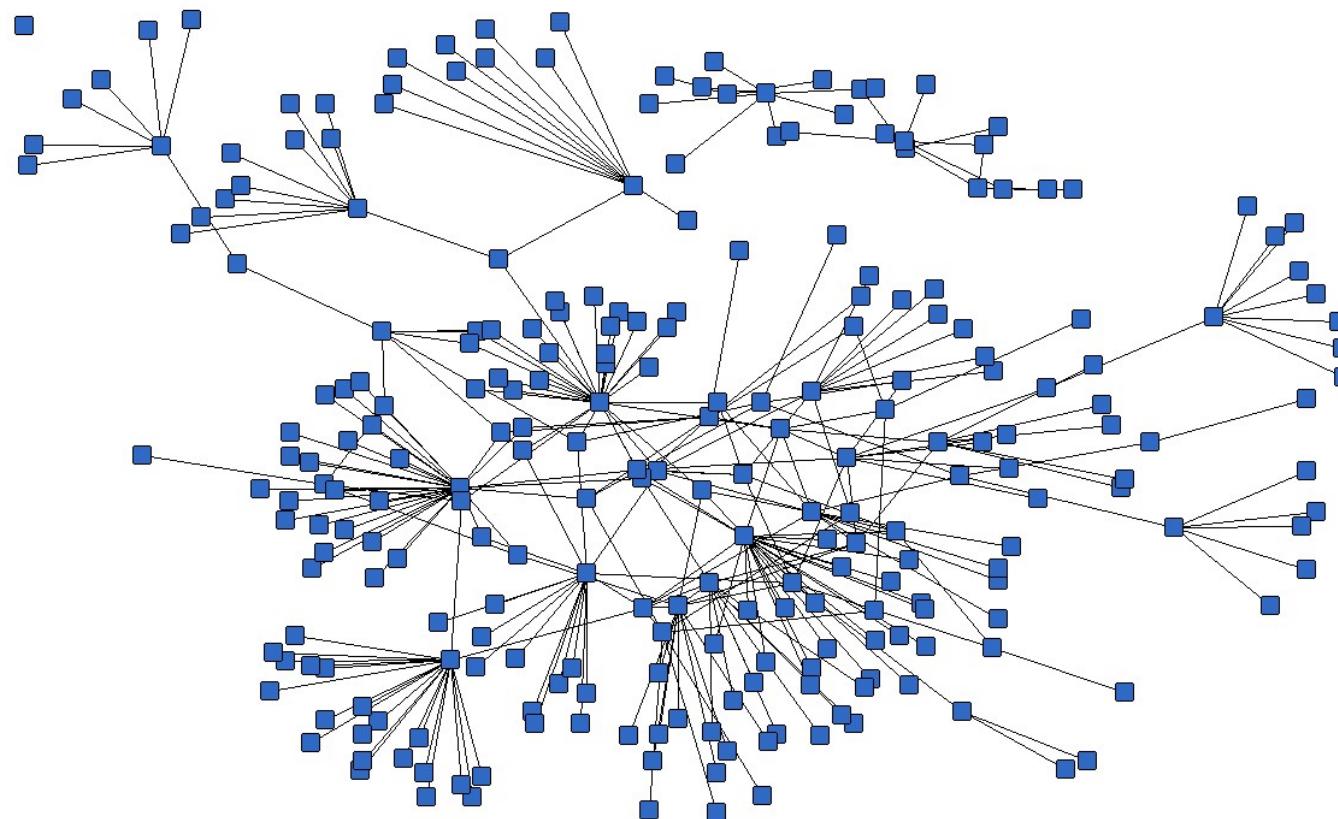
- Create and manage volumes.
- Share volumes across containers.
- Share a host directory with one or many containers.



Container networking basics

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

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.
- Connect to that network service.
- Find a container's IP address.

Running a very simple service

- We need something small, simple, easy to configure
(or, even better, that doesn't require any configuration at all)
- Let's use the official NGINX image (named `nginx`)
- It runs a static web server listening on port 80
- It serves a default "Welcome to nginx!" page

Runing an NGINX server

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

- Docker will automatically pull the `nginx` image from the Docker Hub
- `-d / --detach` tells Docker to run it in the background
- `P / --publish-all` tells Docker to publish all ports
 - (publish = make them reachable from other computers)
- ...OK, how do we connect to our web server now?

Finding our web server port

- First, we need to find the *port number* used by Docker
(the NGINX container listens on port 80, but this port will be *mapped*)
- We can use `docker ps`:

```
$ docker ps
CONTAINER ID  IMAGE      ...  PORTS          ...
e40ffb406c9e  nginx      ...  0.0.0.0:12345->80/tcp  ...
```

- This means:
port 12345 on the Docker host is mapped to port 80 in the container
- Now we need to connect to the Docker host!

Finding the address of the Docker host

- When running Docker on your Linux workstation:

use `localhost`, or any IP address of your machine

- When running Docker on a remote Linux server:

use any IP address of the remote machine

- When running Docker Desktop on Mac or Windows:

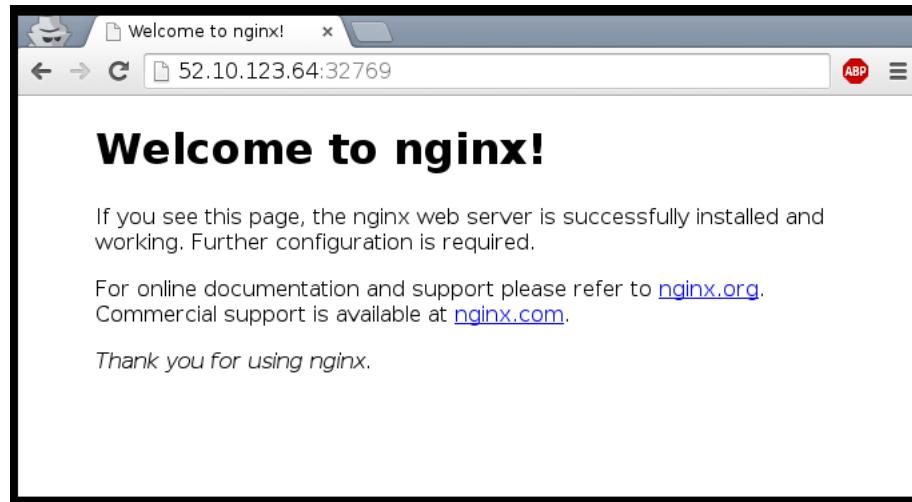
use `localhost`

- In other scenarios (`docker-machine`, local VM...):

use the IP address of the Docker VM

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:12345
<!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)  STOP SIGNAL [SIGTERM]
<missing>      11 days ago      /bin/sh -c #(nop)  EXPOSE 80/tcp
```

Why can't we just connect to port 80?

- Our Docker host has only one port 80
- Therefore, we can only have one container at a time on port 80
- Therefore, if multiple containers want port 80, only one can get it
- By default, containers *do not* get "their" port number, but a random one
(not "random" as "crypto random", but as "it depends on various factors")
- We'll see later how to force a port number (including port 80!)



Using multiple IP addresses

Hey, my network-fu is strong, and I have questions...

- Can I publish one container on 127.0.0.2:80, and another on 127.0.0.3:80?
- My machine has multiple (public) IP addresses, let's say A.A.A.A and B.B.B.B.
Can I have one container on A.A.A.A:80 and another on B.B.B.B:80?
- I have a whole IPV4 subnet, can I allocate it to my containers?
- What about IPV6?

You can do all these things when running Docker directly on Linux.

(On other platforms, *generally not*, but there are some exceptions.)

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  
0.0.0.0:12345
```

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 an orchestrator like Kubernetes or Swarm.
The orchestrator will provide its own networking facilities.

Orchestrators typically provide mechanisms to enable direct container-to-container communication across hosts, and publishing/load balancing for inbound traffic.

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

Let's try to ping our container *from another container*.

```
docker run alpine ping <ipaddress>
PING 172.17.0.X (172.17.0.X): 56 data bytes
64 bytes from 172.17.0.X: seq=0 ttl=64 time=0.106 ms
64 bytes from 172.17.0.X: seq=1 ttl=64 time=0.250 ms
64 bytes from 172.17.0.X: seq=2 ttl=64 time=0.188 ms
```

When running on Linux, we can even ping that IP address directly!

(And connect to a container's ports even if they aren't published.)

How often do we use -p and -P ?

- When running a stack of containers, we will often use Compose
- Compose will take care of exposing containers
(through a `ports:` section in the `docker-compose.yml` file)
- It is, however, fairly common to use `docker run -P` for a quick test
- Or `docker run -p ...` when an image doesn't `EXPOSE` a port correctly

Section summary

We've learned how to:

- Expose a network port.
- Connect to an application running in a container.
- Find a container's IP address.



Container network drivers

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

Container network drivers

The Docker Engine supports different network drivers.

The built-in drivers include:

- bridge (default)
- null (for the special network called none)
- host (for the special network called host)
- container (that one is a bit magic!)

The network is selected with `docker run --net ...`.

Each network is managed by a driver.

The different drivers are explained with more details on the following slides.

The default bridge

- By default, the container gets a virtual `eth0` interface.
(In addition to its own private `lo` loopback interface.)
- That interface is provided by a `veth` pair.
- It is connected to the Docker bridge.
(Named `docker0` by default; configurable with `--bridge`.)
- Addresses are allocated on a private, internal subnet.
(Docker uses `172.17.0.0/16` by default; configurable with `--bip`.)
- Outbound traffic goes through an iptables MASQUERADE rule.
- Inbound traffic goes through an iptables DNAT rule.
- The container can have its own routes, iptables rules, etc.

The null driver

- Container is started with `docker run --net none ...`
- It only gets the `lo` loopback interface. No `eth0`.
- It can't send or receive network traffic.
- Useful for isolated/untrusted workloads.

The host driver

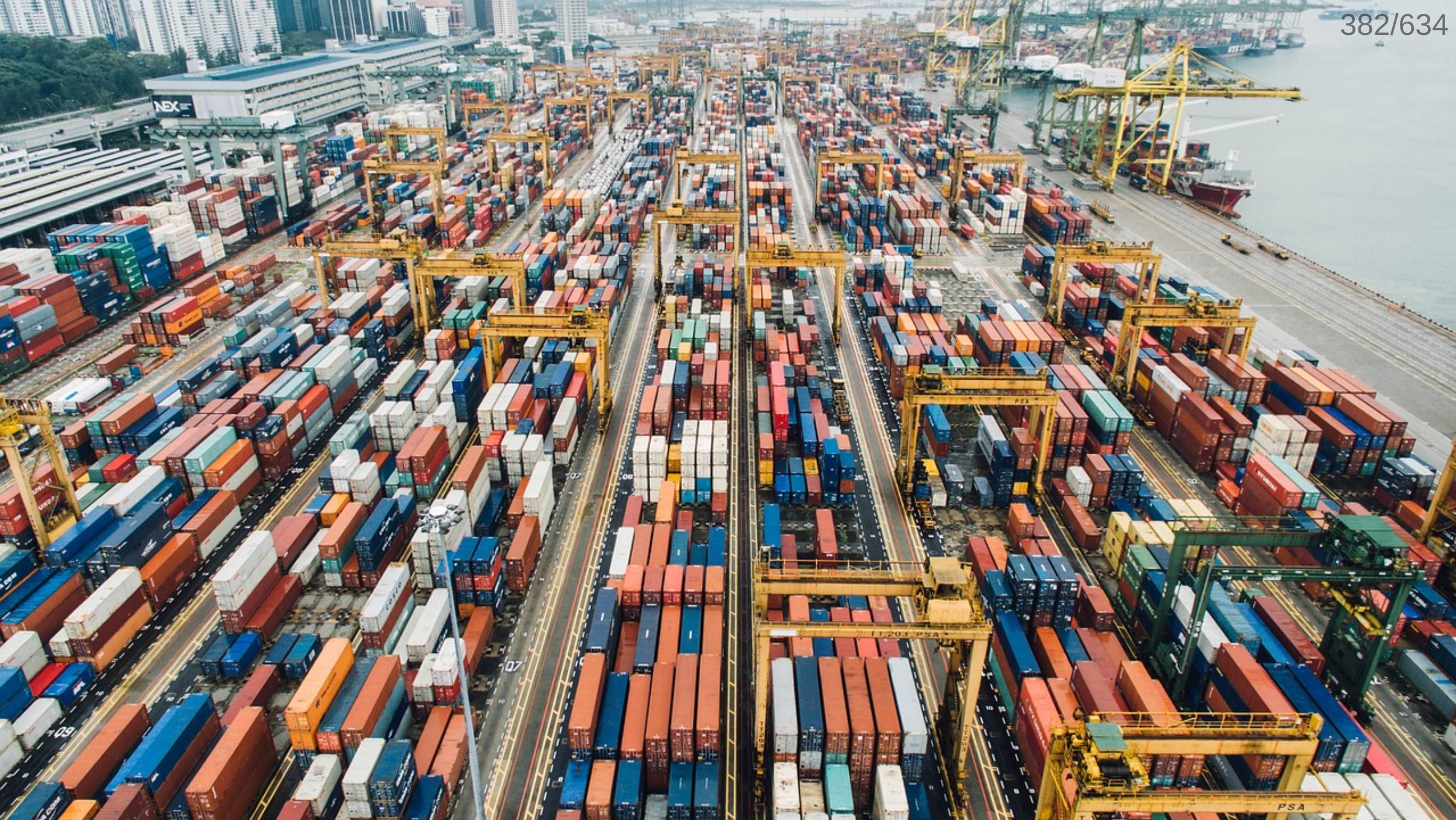
- Container is started with `docker run --net host ...`
- It sees (and can access) the network interfaces of the host.
- It can bind any address, any port (for ill and for good).
- Network traffic doesn't have to go through NAT, bridge, or veth.
- Performance = native!

Use cases:

- Performance sensitive applications (VOIP, gaming, streaming...)
- Peer discovery (e.g. Erlang port mapper, Raft, Serf...)

The container driver

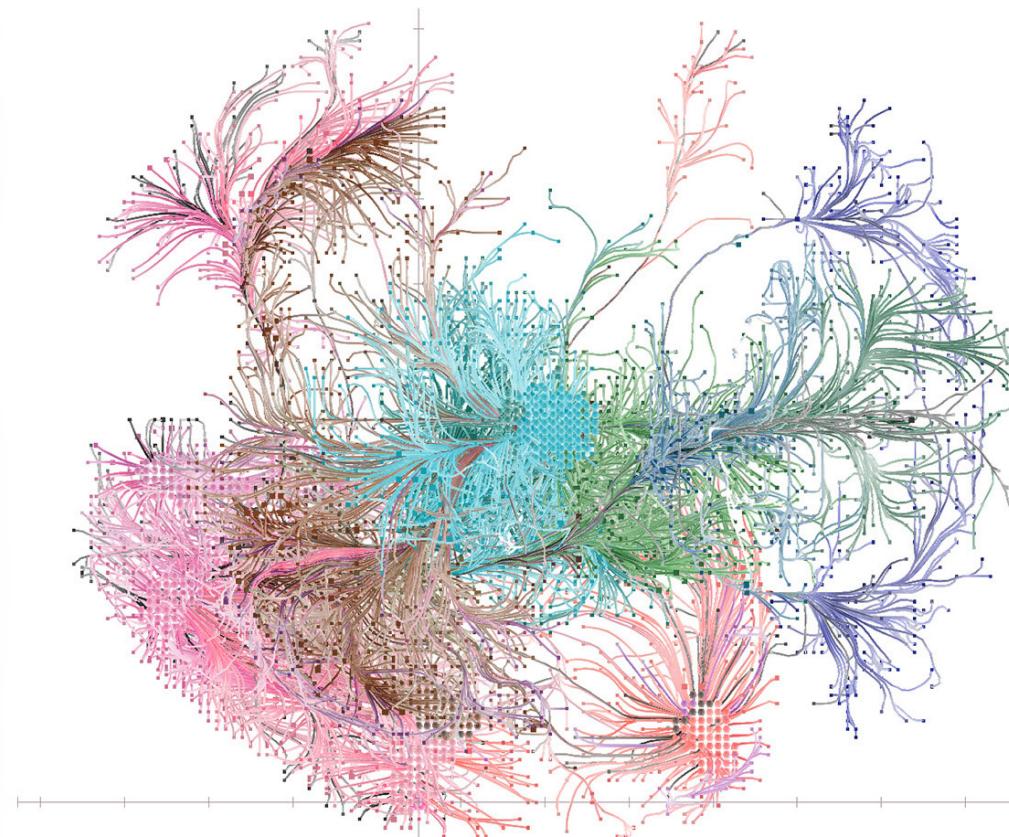
- Container is started with `docker run --net container:id ...`
- It re-uses the network stack of another container.
- It shares with this other container the same interfaces, IP address(es), routes, iptables rules, etc.
- Those containers can communicate over their `lo` interface.
(i.e. one can bind to 127.0.0.1 and the others can connect to it.)



The Container Network Model

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

The Container Network Model



Objectives

We will learn about the CNM (Container Network Model).

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

- Create a private network for a group of containers.
- Use container naming to connect services together.
- Dynamically connect and disconnect containers to networks.
- Set the IP address of a container.

We will also explain the principle of overlay networks and network plugins.

The Container Network Model

Docker has "networks".

We can manage them with the `docker network` commands; for instance:

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER
6bde79dfcf70	bridge	bridge
8d9c78725538	none	null
eb0eeab782f4	host	host
4c1ff84d6d3f	blog-dev	overlay
228a4355d548	blog-prod	overlay

New networks can be created (with `docker network create`).

(Note: networks `none` and `host` are special; let's set them aside for now.)

What's a network?

- Conceptually, a Docker "network" is a virtual switch
(we can also think about it like a VLAN, or a WiFi SSID, for instance)
- By default, containers are connected to a single network
(but they can be connected to zero, or many networks, even dynamically)
- Each network has its own subnet (IP address range)
- A network can be local (to a single Docker Engine) or global (span multiple hosts)
- Containers can have *network aliases* providing DNS-based service discovery
(and each network has its own "domain", "zone", or "scope")

Service discovery

- A container can be given a network alias
(e.g. with `docker run --net some-network --net-alias db ...`)
- The containers running in the same network can resolve that network alias
(i.e. if they do a DNS lookup on `db`, it will give the container's address)
- We can have a different `db` container in each network
(this avoids naming conflicts between different stacks)
- When we name a container, it automatically adds the name as a network alias
(i.e. `docker run --name xyz ...` is like `docker run --net-alias xyz ...`)

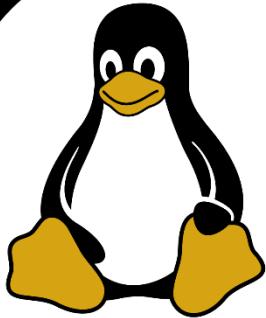
Network isolation

- Networks are isolated
- By default, containers in network A cannot reach those in network B
- A container connected to both networks A and B can act as a router or proxy
- Published ports are always reachable through the Docker host address
(`docker run -P ...` makes a container port available to everyone)

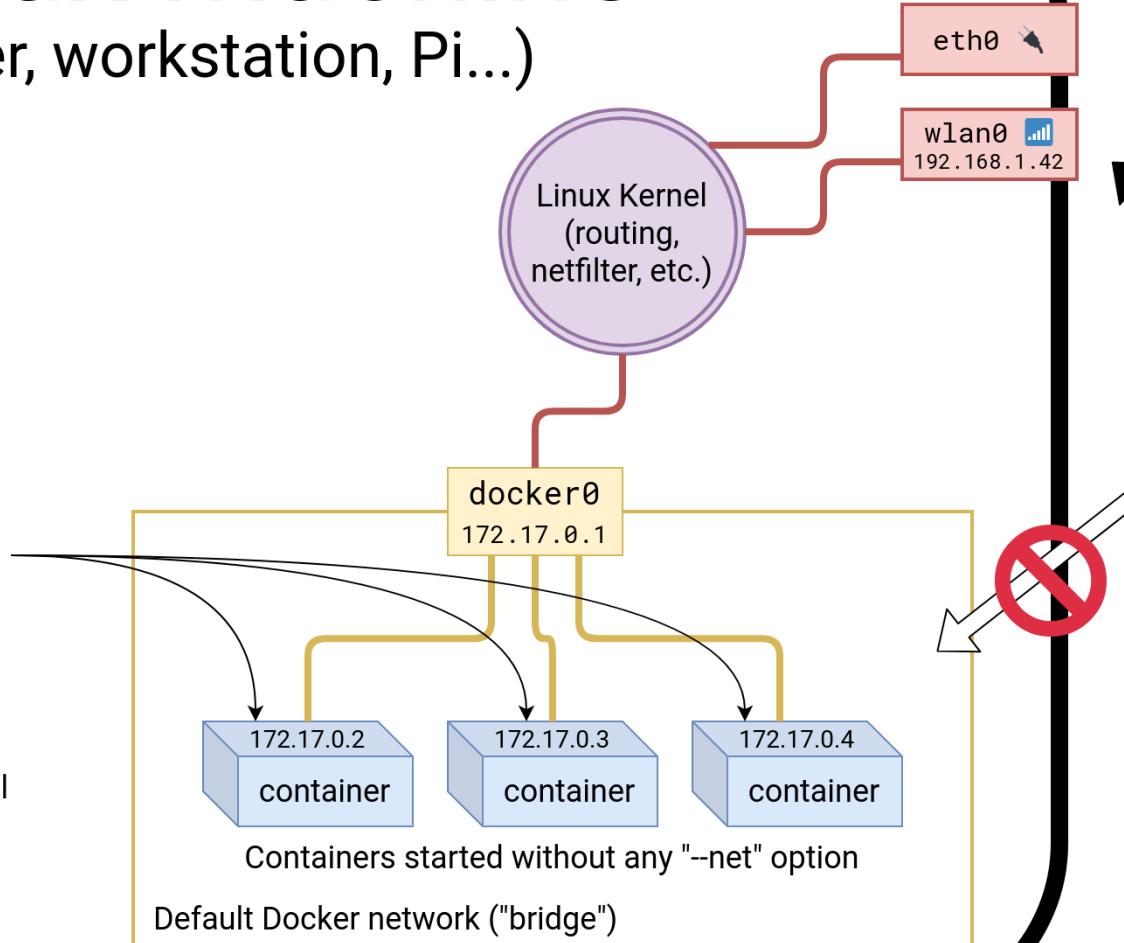
How to use networks

- We typically create one network per "stack" or app that we deploy
- More complex apps or stacks might require multiple networks
(e.g. frontend, backend, ...)
- Networks allow us to deploy multiple copies of the same stack
(e.g. prod, dev, pr-442,)
- If we use Docker Compose, this is managed automatically for us

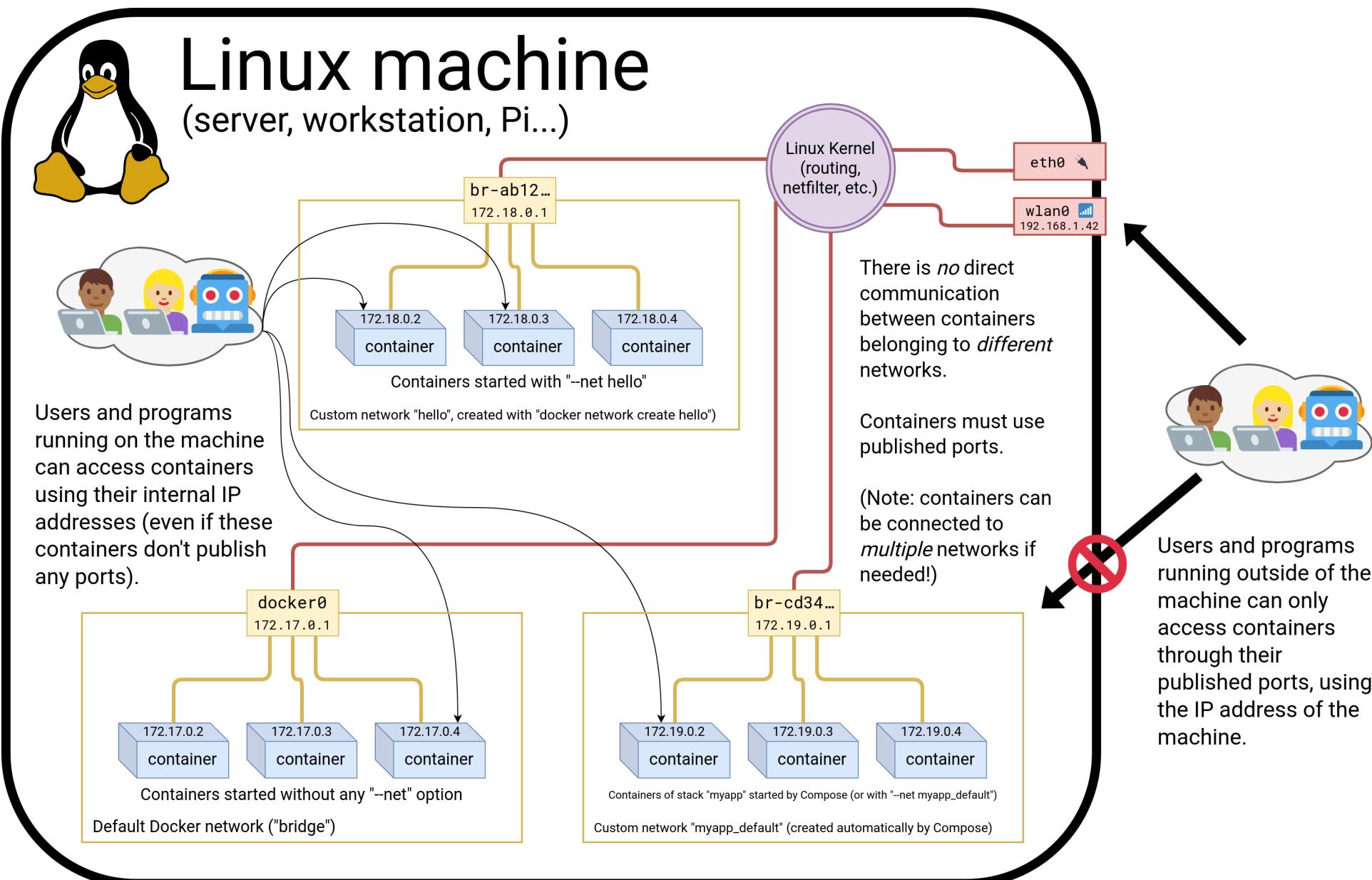
Linux machine (server, workstation, Pi...)

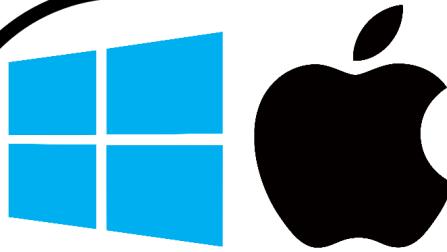


Users and programs running on the machine can access containers using their internal IP addresses (even if these containers don't publish any ports).



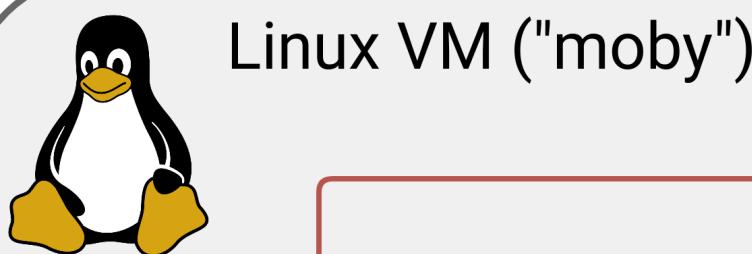
Users and programs running outside of the machine can only access containers through their published ports, using the IP address of the machine.



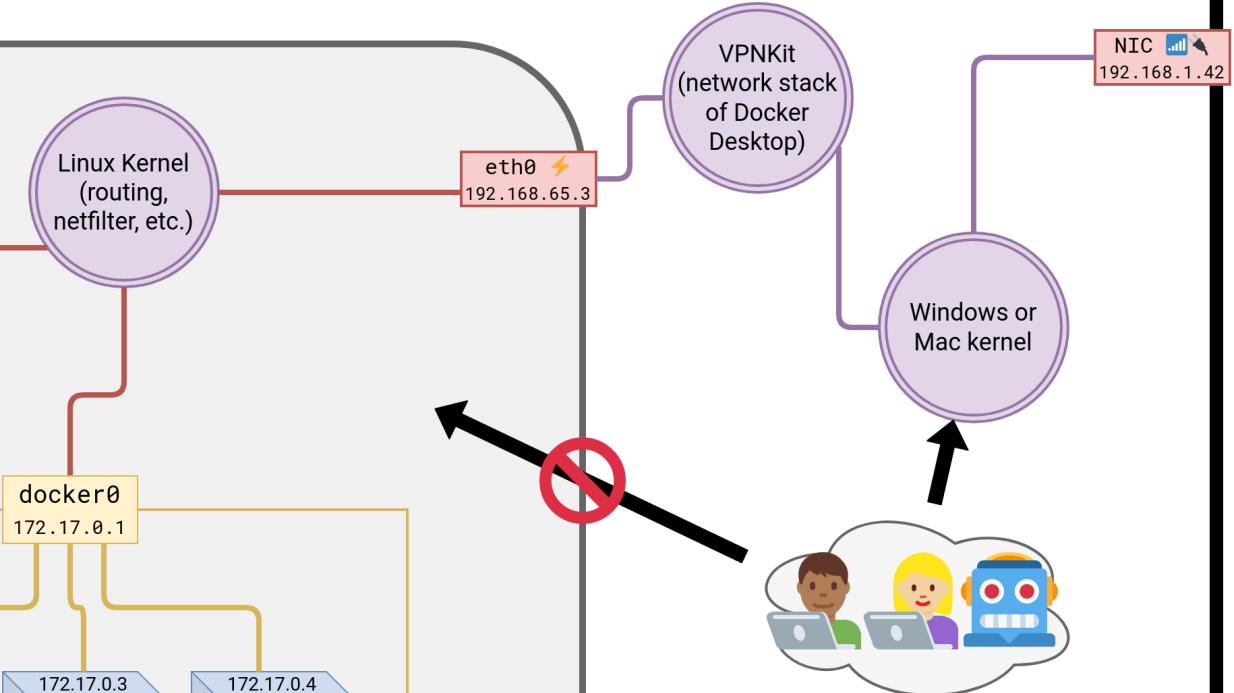
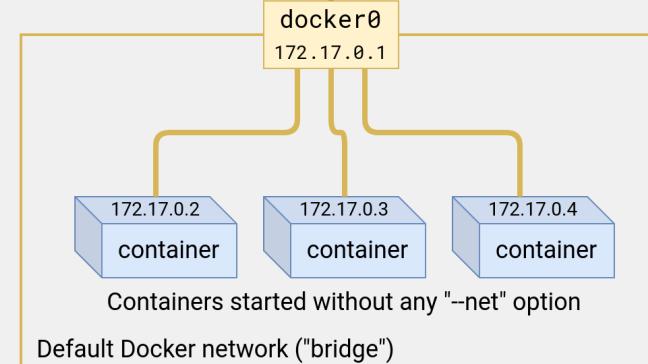
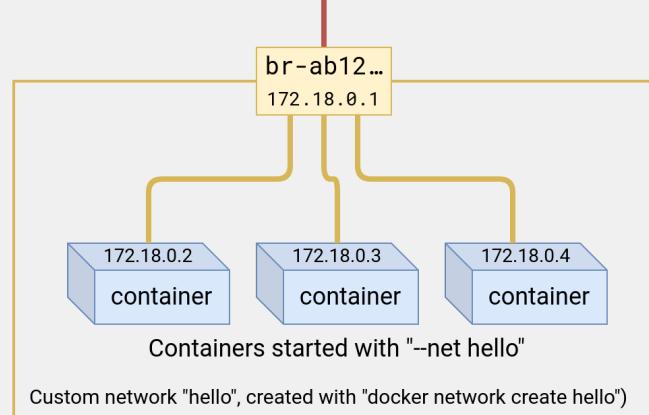


Windows or Mac machine

(with Docker Desktop; note: WSL might be different!)



Linux VM ("moby")



Users and programs running on the machine can only access containers through their published ports.



CNM vs CNI

- CNM is the model used by Docker
- Kubernetes uses a different model, architected around CNI
(CNI is a kind of API between a container engine and *CNI plugins*)
- Docker model:
 - multiple isolated networks
 - per-network service discovery
 - network interconnection requires extra steps
- Kubernetes model:
 - single flat network
 - per-namespace service discovery
 - network isolation requires extra steps (Network Policies)

Creating a network

Let's create a network called `dev`.

```
$ docker network create dev  
4c1ff84d6d3f1733d3e233ee039cac276f425a9d5228a4355d54878293a889ba
```

The network is now visible with the `network ls` command:

```
$ docker network ls  
NETWORK ID      NAME      DRIVER  
6bde79dfcf70    bridge    bridge  
8d9c78725538    none     null  
eb0eeab782f4    host     host  
4c1ff84d6d3f    dev      bridge
```

Placing containers on a network

We will create a *named* container on this network.

It will be reachable with its name, es.

```
$ docker run -d --name es --net dev elasticsearch:2  
8abb80e229ce8926c7223beb69699f5f34d6f1d438bfc5682db893e798046863
```

Communication between containers

Now, create another container on this network.

```
$ docker run -ti --net dev alpine sh  
root@0ecccdfa45ef:/#
```

From this new container, we can resolve and ping the other one, using its assigned name:

```
/ # ping es  
PING es (172.18.0.2) 56(84) bytes of data.  
64 bytes from es.dev (172.18.0.2): icmp_seq=1 ttl=64 time=0.221 ms  
64 bytes from es.dev (172.18.0.2): icmp_seq=2 ttl=64 time=0.114 ms  
64 bytes from es.dev (172.18.0.2): icmp_seq=3 ttl=64 time=0.114 ms  
^C  
--- es ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 2000ms  
rtt min/avg/max/mdev = 0.114/0.149/0.221/0.052 ms  
root@0ecccdfa45ef:/#
```



Resolving container addresses

Since Docker Engine 1.10, name resolution is implemented by a dynamic resolver.

Archeological note: when CNM was introduced (in Docker Engine 1.9, November 2015) name resolution was implemented with `/etc/hosts`, and it was updated each time CONTAINERs were added/removed. This could cause interesting race conditions since `/etc/hosts` was a bind-mount (and couldn't be updated atomically).

```
[root@0ecccdfa45ef /]# cat /etc/hosts
172.18.0.3  0ecccdfa45ef
127.0.0.1      localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.18.0.2      es
172.18.0.2      es.dev
```



Service discovery with containers

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

Service discovery with containers

- Let's try to run an application that requires two containers.
- The first container is a web server.
- The other one is a redis data store.
- We will place them both on the `dev` network created before.

Running the web server

- The application is provided by the container image [jpetazzo/trainingwheels](#).
- We don't know much about it so we will try to run it and see what happens!

Start the container, exposing all its ports:

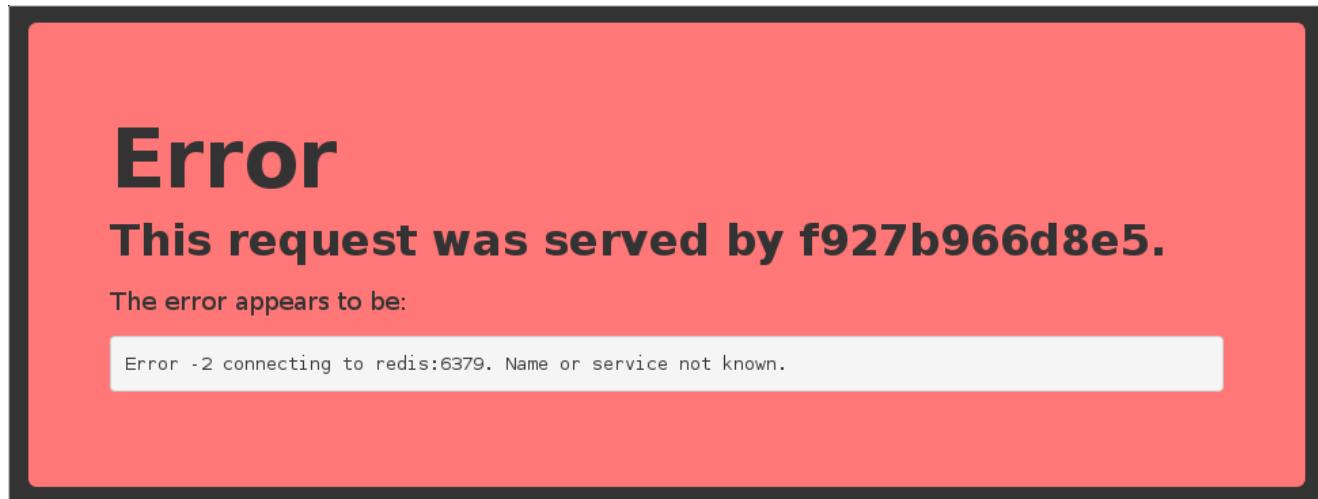
```
$ docker run --net dev -d -P jpetazzo/trainingwheels
```

Check the port that has been allocated to it:

```
$ docker ps -l
```

Test the web server

- If we connect to the application now, we will see an error page:



- This is because the Redis service is not running.
- This container tries to resolve the name `redis`.

Note: we're not using a FQDN or an IP address here; just `redis`.

Start the data store

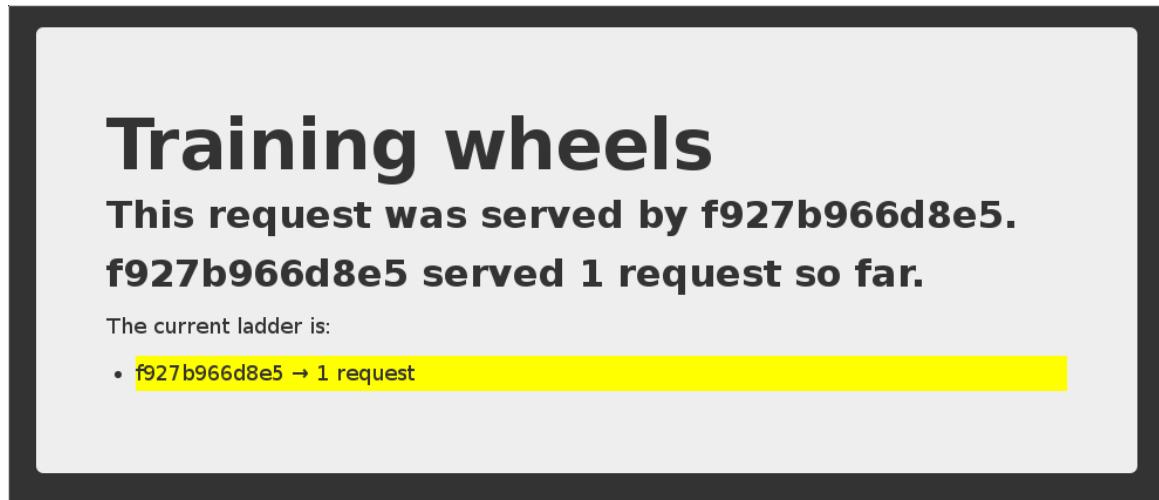
- We need to start a Redis container.
- That container must be on the same network as the web server.
- It must have the right network alias (`redis`) so the application can find it.

Start the container:

```
$ docker run --net dev --net-alias redis -d redis
```

Test the web server again

- If we connect to the application now, we should see that the app is working correctly:



- When the app tries to resolve `redis`, instead of getting a DNS error, it gets the IP address of our Redis container.

A few words on scope

- Container names are unique (there can be only one `--name redis`)
- Network aliases are not unique
- We can have the same network alias in different networks:

```
docker run --net dev --net-alias redis ...
docker run --net prod --net-alias redis ...
```

- We can even have multiple containers with the same alias in the same network
(in that case, we get multiple DNS entries, aka "DNS round robin")



Names are *local* to each network

Let's try to ping our `es` container from another container, when that other container is *not* on the `dev` network.

```
$ docker run --rm alpine ping es
ping: bad address 'es'
```

Names can be resolved only when containers are on the same network.

Containers can contact each other only when they are on the same network (you can try to ping using the IP address to verify).



Network aliases

We would like to have another network, `prod`, with its own `es` container. But there can be only one container named `es`!

We will use *network aliases*.

A container can have multiple network aliases.

Network aliases are *local* to a given network (only exist in this network).

Multiple containers can have the same network alias (even on the same network).

Since Docker Engine 1.11, resolving a network alias yields the IP addresses of all containers holding this alias.



Creating containers on another network

Create the prod network.

```
$ docker network create prod  
5a41562fecf2d8f115bedc16865f7336232a04268bdf2bd816aecca01b68d50c
```

We can now create multiple containers with the es alias on the new prod network.

```
$ docker run -d --name prod-es-1 --net-alias es --net prod elasticsearch:2  
38079d21caf0c5533a391700d9e9e920724e89200083df73211081c8a356d771  
$ docker run -d --name prod-es-2 --net-alias es --net prod elasticsearch:2  
1820087a9c600f43159688050dcc164c298183e1d2e62d5694fd46b10ac3bc3d
```



Resolving network aliases

Let's try DNS resolution first, using the `nslookup` tool that ships with the `alpine` image.

```
$ docker run --net prod --rm alpine nslookup es
Name:      es
Address 1: 172.23.0.3 prod-es-2.prod
Address 2: 172.23.0.2 prod-es-1.prod
```

(You can ignore the `can't resolve '(null)'` errors.)



Connecting to aliased containers

Each ElasticSearch instance has a name (generated when it is started). This name can be seen when we issue a simple HTTP request on the ElasticSearch API endpoint.

Try the following command a few times:

```
$ docker run --rm --net dev centos curl -s es:9200
{
  "name" : "Tarot",
...
}
```

Then try it a few times by replacing `--net dev` with `--net prod`:

```
$ docker run --rm --net prod centos curl -s es:9200
{
  "name" : "The Symbiote",
...
}
```

Good to know ...

- Docker will not create network names and aliases on the default `bridge` network.
- Therefore, if you want to use those features, you have to create a custom network first.
- Network aliases are *not* unique on a given network.
- i.e., multiple containers can have the same alias on the same network.
- In that scenario, the Docker DNS server will return multiple records.
(i.e. you will get DNS round robin out of the box.)
- Enabling *Swarm Mode* gives access to clustering and load balancing with IPVS.
- Creation of networks and network aliases is generally automated with tools like Compose.



A few words about round robin DNS

Don't rely exclusively on round robin DNS to achieve load balancing.

Many factors can affect DNS resolution, and you might see:

- all traffic going to a single instance;
- traffic being split (unevenly) between some instances;
- different behavior depending on your application language;
- different behavior depending on your base distro;
- different behavior depending on other factors (sic).

It's OK to use DNS to discover available endpoints, but remember that you have to re-resolve every now and then to discover new endpoints.



Custom networks

When creating a network, extra options can be provided.

- `--internal` disables outbound traffic (the network won't have a default gateway).
- `--gateway` indicates which address to use for the gateway (when outbound traffic is allowed).
- `--subnet` (in CIDR notation) indicates the subnet to use.
- `--ip-range` (in CIDR notation) indicates the subnet to allocate from.
- `--aux-address` allows specifying a list of reserved addresses (which won't be allocated to containers).



Setting containers' IP address

- It is possible to set a container's address with `--ip`.
- The IP address has to be within the subnet used for the container.

A full example would look like this.

```
$ docker network create --subnet 10.66.0.0/16 pubnet
42fb16ec412383db6289a3e39c3c0224f395d7f85bcb1859b279e7a564d4e135
$ docker run --net pubnet --ip 10.66.66.66 -d nginx
b2887adeb5578a01fd9c55c435cad56bbbe802350711d2743691f95743680b09
```

Note: don't hard code container IP addresses in your code!

I repeat: don't hard code container IP addresses in your code!

Network drivers

- A network is managed by a *driver*.
- The built-in drivers include:
 - `bridge` (default)
 - `none`
 - `host`
 - `macvlan`
 - `overlay` (for Swarm clusters)
- More drivers can be provided by plugins (OVS, VLAN...)
- A network can have a custom IPAM (IP allocator).

Overlay networks

- The features we've seen so far only work when all containers are on a single host.
- If containers span multiple hosts, we need an *overlay* network to connect them together.
- Docker ships with a default network plugin, `overlay`, implementing an overlay network leveraging VXLAN, *enabled with Swarm Mode*.
- Other plugins (Weave, Calico...) can provide overlay networks as well.
- Once you have an overlay network, *all the features that we've used in this chapter work identically across multiple hosts*.



Multi-host networking (overlay)

Out of the scope for this intro-level workshop!

Very short instructions:

- enable Swarm Mode (`docker swarm init` then `docker swarm join` on other nodes)
- `docker network create mynet --driver overlay`
- `docker service create --network mynet myimage`

If you want to learn more about Swarm mode, you can check [this video](#) or [these slides](#).



Multi-host networking (plugins)

Out of the scope for this intro-level workshop!

General idea:

- install the plugin (they often ship within containers)
- run the plugin (if it's in a container, it will often require extra parameters; don't just `docker run` it blindly!)
- some plugins require configuration or activation (creating a special file that tells Docker "use the plugin whose control socket is at the following location")
- you can then `docker network create --driver pluginname`

Connecting and disconnecting dynamically

- So far, we have specified which network to use when starting the container.
- The Docker Engine also allows connecting and disconnecting while the container is running.
- This feature is exposed through the Docker API, and through two Docker CLI commands:
 - `docker network connect <network> <container>`
 - `docker network disconnect <network> <container>`

Dynamically connecting to a network

- We have a container named `es` connected to a network named `dev`.
- Let's start a simple alpine container on the default network:

```
$ docker run -ti alpine sh  
/ #
```

- In this container, try to ping the `es` container:

```
/ # ping es  
ping: bad address 'es'
```

This doesn't work, but we will change that by connecting the container.

Finding the container ID and connecting it

- Figure out the ID of our alpine container; here are two methods:
 - looking at `/etc/hostname` in the container,
 - running `docker ps -lq` on the host.
- Run the following command on the host:

```
$ docker network connect dev <container_id>
```

Checking what we did

- Try again to ping es from the container.
- It should now work correctly:

```
/ # ping es
PING es (172.20.0.3): 56 data bytes
64 bytes from 172.20.0.3: seq=0 ttl=64 time=0.376 ms
64 bytes from 172.20.0.3: seq=1 ttl=64 time=0.130 ms
^C
```

- Interrupt it with Ctrl-C.

Looking at the network setup in the container

We can look at the list of network interfaces with `ifconfig`, `ip a`, or `ip l`:

```
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
18: eth0@if19: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
20: eth1@if21: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:14:00:04 brd ff:ff:ff:ff:ff:ff
    inet 172.20.0.4/16 brd 172.20.255.255 scope global eth1
        valid_lft forever preferred_lft forever
/ #
```

Each network connection is materialized with a virtual network interface.

As we can see, we can be connected to multiple networks at the same time.

Disconnecting from a network

- Let's try the symmetrical command to disconnect the container:

```
$ docker network disconnect dev <container_id>
```

- From now on, if we try to ping `es`, it will not resolve:

```
/ # ping es
ping: bad address 'es'
```

- Trying to ping the IP address directly won't work either:

```
/ # ping 172.20.0.3
... (nothing happens until we interrupt it with Ctrl-C)
```



Network aliases are scoped per network

- Each network has its own set of network aliases.
- We saw this earlier: `es` resolves to different addresses in `dev` and `prod`.
- If we are connected to multiple networks, the resolver looks up names in each of them (as of Docker Engine 18.03, it is the connection order) and stops as soon as the name is found.
- Therefore, if we are connected to both `dev` and `prod`, resolving `es` will **not** give us the addresses of all the `es` services; but only the ones in `dev` or `prod`.
- However, we can lookup `es.dev` or `es.prod` if we need to.



Finding out about our networks and names

- We can do reverse DNS lookups on containers' IP addresses.
- If the IP address belongs to a network (other than the default bridge), the result will be:

name-or-first-alias-or-container-id.network-name

- Example:

```
$ docker run -ti --net prod --net-alias hello alpine
/ # apk add --no-cache drill
...
OK: 5 MiB in 13 packages
/ # ifconfig
eth0      Link encap:Ethernet HWaddr 02:42:AC:15:00:03
          inet addr:172.21.0.3  Bcast:172.21.255.255  Mask:255.255.0.0
                  UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
...
/ # drill -t ptr 3.0.21.172.in-addr.arpa
...
;; ANSWER SECTION:
3.0.21.172.in-addr.arpa.    600    IN    PTR    hello.prod.
```



Building with a custom network

- We can build a Dockerfile with a custom network with `docker build --network NAME`.
- This can be used to check that a build doesn't access the network.
(But keep in mind that most Dockerfiles will fail,
because they need to install remote packages and dependencies!)
- This may be used to access an internal package repository.
(But try to use a multi-stage build instead, if possible!)



Compose for development stacks

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

Compose for development stacks

Dockerfile = great to build *one* container image.

What if we have multiple containers?

What if some of them require particular `docker run` parameters?

How do we connect them all together?

... Compose solves these use-cases (and a few more).

Life before Compose

Before we had Compose, we would typically write custom scripts to:

- build container images,
- run containers using these images,
- connect the containers together,
- rebuild, restart, update these images and containers.

Life with Compose

Compose enables a simple, powerful onboarding workflow:

1. Checkout our code.
2. Run `docker-compose up`.
3. Our app is up and running!

```
$ d[
```

Life after Compose

(Or: when do we need something else?)

- Compose is *not* an orchestrator
- It isn't designed to need to run containers on multiple nodes
(it can, however, work with Docker Swarm Mode)
- Compose isn't ideal if we want to run containers on Kubernetes
 - it uses different concepts (Compose services ≠ Kubernetes services)
 - it needs a Docker Engine (although containerd support might be coming)

First rodeo with Compose

1. Write Dockerfiles
2. Describe our stack of containers in a YAML file called `docker-compose.yml`
3. `docker-compose up` (or `docker-compose up -d` to run in the background)
4. Compose pulls and builds the required images, and starts the containers
5. Compose shows the combined logs of all the containers
(if running in the background, use `docker-compose logs`)
6. Hit Ctrl-C to stop the whole stack
(if running in the background, use `docker-compose stop`)

Iterating

After making changes to our source code, we can:

1. `docker-compose build` to rebuild container images
2. `docker-compose up` to restart the stack with the new images

We can also combine both with `docker-compose up --build`

Compose will be smart, and only recreate the containers that have changed.

When working with interpreted languages:

- don't rebuild each time
- leverage a `volumes` section instead

Launching Our First Stack with Compose

First step: clone the source code for the app we will be working on.

```
git clone https://github.com/jpetazzo/trainingwheels  
cd trainingwheels
```

Second step: start the app.

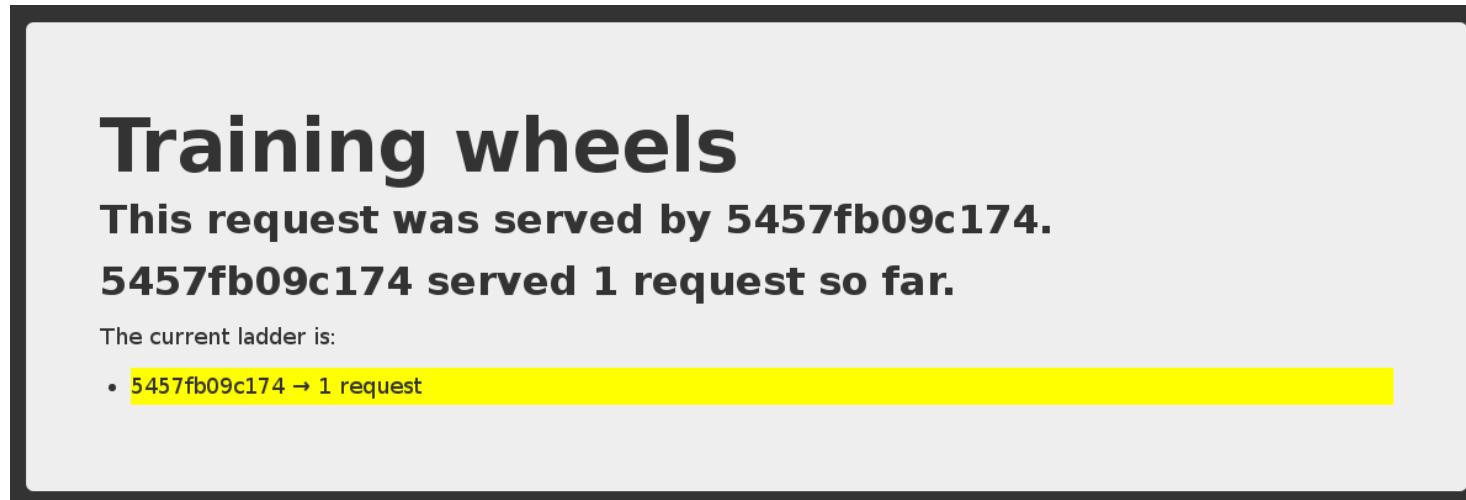
```
docker-compose up
```

Watch Compose build and run the app.

That Compose stack exposes a web server on port 8000; try connecting to it.

Launching Our First Stack with Compose

We should see a web page like this:



Each time we reload, the counter should increase.

Stopping the app

When we hit Ctrl-C, Compose tries to gracefully terminate all of the containers.

After ten seconds (or if we press ^C again) it will forcibly kill them.

The docker-compose.yml file

Here is the file used in the demo:

```
version: "3"

services:
  www:
    build: www
    ports:
      - ${PORT-8000}:5000
    user: nobody
    environment:
      DEBUG: 1
    command: python counter.py
    volumes:
      - ./www:/src

  redis:
    image: redis
```

Compose file structure

A Compose file has multiple sections:

- `version` is mandatory. (Typically use "3".)
- `services` is mandatory. Each service corresponds to a container.
- `networks` is optional and indicates to which networks containers should be connected.
(By default, containers will be connected on a private, per-compose-file network.)
- `volumes` is optional and can define volumes to be used and/or shared by the containers.

Compose file versions

- Version 1 is legacy and shouldn't be used.

(If you see a Compose file without `version` and `services`, it's a legacy v1 file.)

- Version 2 added support for networks and volumes.
- Version 3 added support for deployment options (scaling, rolling updates, etc).
- Typically use `version: "3"`.

The [Docker documentation](#) has excellent information about the Compose file format if you need to know more about versions.

Containers in docker-compose.yml

Each service in the YAML file must contain either `build`, or `image`.

- `build` indicates a path containing a Dockerfile.
- `image` indicates an image name (local, or on a registry).
- If both are specified, an image will be built from the `build` directory and named `image`.

The other parameters are optional.

They encode the parameters that you would typically add to `docker run`.

Sometimes they have several minor improvements.

Container parameters

- `command` indicates what to run (like `CMD` in a Dockerfile).
- `ports` translates to one (or multiple) `-p` options to map ports.
You can specify local ports (i.e. `x:y` to expose public port `x`).
- `volumes` translates to one (or multiple) `-v` options.
You can use relative paths here.

For the full list, check: <https://docs.docker.com/compose/compose-file/>

Environment variables

- We can use environment variables in Compose files
(like `$THIS` or `${THAT}`)
- We can provide default values, e.g. `${PORT-8000}`
- Compose will also automatically load the environment file `.env`
(it should contain `VAR=value`, one per line)
- This is a great way to customize build and run parameters
(base image versions to use, build and run secrets, port numbers...)

Configuring a Compose stack

- Follow [12-factor app configuration principles](#)
(configure the app through environment variables)
- Provide (in the repo) a default environment file suitable for development
(no secret or sensitive value)
- Copy the default environment file to `.env` and tweak it
(or: provide a script to generate `.env` from a template)

Running multiple copies of a stack

- Copy the stack in two different directories, e.g. `front` and `frontcopy`
- Compose prefixes images and containers with the directory name:

`front_www`, `front_www_1`, `front_db_1`

`frontcopy_www`, `frontcopy_www_1`, `frontcopy_db_1`

- Alternatively, use `docker-compose -p frontcopy`
(to set the `--project-name` of a stack, which default to the dir name)
- Each copy is isolated from the others (runs on a different network)

Checking stack status

We have `ps`, `docker ps`, and similarly, `docker-compose ps`:

```
$ docker-compose ps
```

Name	Command	State	Ports
<hr/>			
trainingwheels_redis_1	/entrypoint.sh red	Up	6379/tcp
trainingwheels_www_1	python counter.py	Up	0.0.0.0:8000->5000/tcp

Shows the status of all the containers of our stack.

Doesn't show the other containers.

Cleaning up (1)

If you have started your application in the background with Compose and want to stop it easily, you can use the `kill` command:

```
$ docker-compose kill
```

Likewise, `docker-compose rm` will let you remove containers (after confirmation):

```
$ docker-compose rm
Going to remove trainingwheels_redis_1, trainingwheels_www_1
Are you sure? [yN] y
Removing trainingwheels_redis_1...
Removing trainingwheels_www_1...
```

Cleaning up (2)

Alternatively, `docker-compose down` will stop and remove containers.

It will also remove other resources, like networks that were created for the application.

```
$ docker-compose down
Stopping trainingwheels_www_1 ... done
Stopping trainingwheels_redis_1 ... done
Removing trainingwheels_www_1 ... done
Removing trainingwheels_redis_1 ... done
```

Use `docker-compose down -v` to remove everything including volumes.

Special handling of volumes

- When an image gets updated, Compose automatically creates a new container
- The data in the old container is lost...
- ...Except if the container is using a *volume*
- Compose will then re-attach that volume to the new container
(and data is then retained across database upgrades)
- All good database images use volumes
(e.g. all official images)

Gotchas with volumes

- Unfortunately, Docker volumes don't have labels or metadata
- Compose tracks volumes thanks to their associated container
- If the container is deleted, the volume gets orphaned
- Example: `docker-compose down && docker-compose up`
 - the old volume still exists, detached from its container
 - a new volume gets created
- `docker-compose down -v/-v--volumes` deletes volumes
(but **not** `docker-compose down && docker-compose down -v!`)

Managing volumes explicitly

Option 1: *named volumes*

```
services:  
  app:  
    volumes:  
      - data:/some/path  
volumes:  
  data:
```

- Volume will be named <project>_data
- It won't be orphaned with docker-compose down
- It will correctly be removed with docker-compose down -v

Managing volumes explicitly

Option 2: *relative paths*

```
services:  
  app:  
    volumes:  
      - ./data:/some/path
```

- Makes it easy to colocate the app and its data
(for migration, backups, disk usage accounting...)
- Won't be removed by docker-compose down -v

Managing complex stacks

- Compose provides multiple features to manage complex stacks
(with many containers)
- `-f/-file/$COMPOSE_FILE` can be a list of Compose files
(separated by `:` and merged together)
- Services can be assigned to one or more *profiles*
- `--profile/$COMPOSE_PROFILE` can be a list of comma-separated profiles
(see [Using service profiles](#) in the Compose documentation)
- These variables can be set in `.env`

Dependencies

- A service can have a `depends_on` section
(listing one or more other services)
 - This is used when bringing up individual services
(e.g. `docker-compose up blah` or `docker-compose run foo`)
-  It doesn't make a service "wait" for another one to be up!



A bit of history and trivia

- Compose was initially named "Fig"
- Compose is one of the only components of Docker written in Python
(almost everything else is in Go)
- In 2020, Docker introduced "Compose CLI":
 - `docker compose` command to deploy Compose stacks to some clouds
 - progressively getting feature parity with `docker-compose`
 - also provides numerous improvements (e.g. leverages BuildKit by default)



Managing hosts with Docker Machine

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

Managing hosts with Docker Machine

- Docker Machine is a tool to provision and manage Docker hosts.
- It automates the creation of a virtual machine:
 - locally, with a tool like VirtualBox or VMware;
 - on a public cloud like AWS EC2, Azure, Digital Ocean, GCP, etc.;
 - on a private cloud like OpenStack.
- It can also configure existing machines through an SSH connection.
- It can manage as many hosts as you want, with as many "drivers" as you want.

Docker Machine workflow

- 1) Prepare the environment: setup VirtualBox, obtain cloud credentials ...
- 2) Create hosts with `docker-machine create -d drivername machinename`.
- 3) Use a specific machine with `eval $(docker-machine env machinename)`.
- 4) Profit!

Environment variables

- Most of the tools (CLI, libraries...) connecting to the Docker API can use environment variables.
- These variables are:
 - `DOCKER_HOST` (indicates address+port to connect to, or path of UNIX socket)
 - `DOCKER_TLS_VERIFY` (indicates that TLS mutual auth should be used)
 - `DOCKER_CERT_PATH` (path to the keypair and certificate to use for auth)
- `docker-machine env ...` will generate the variables needed to connect to a host.
- `$(eval docker-machine env ...)` sets these variables in the current shell.

Host management features

With `docker-machine`, we can:

- upgrade a host to the latest version of the Docker Engine,
- start/stop/restart hosts,
- get a shell on a remote machine (with SSH),
- copy files to/from remotes machines (with SCP),
- mount a remote host's directory on the local machine (with SSHFS),
- ...

The generic driver

When provisioning a new host, `docker-machine` executes these steps:

- 1) Create the host using a cloud or hypervisor API.
- 2) Connect to the host over SSH.
- 3) Install and configure Docker on the host.

With the `generic` driver, we provide the IP address of an existing host (instead of e.g. cloud credentials) and we omit the first step.

This allows to provision physical machines, or VMs provided by a 3rd party, or use a cloud for which we don't have a provisioning API.



Exercise — writing a Compose file

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

Exercise — writing a Compose file

Let's write a Compose file for the wordsmith app!

The code is at: <https://github.com/jpetazzo/wordsmith>

469/634



SwarmKit

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

SwarmKit

- [SwarmKit](#) is an open source toolkit to build multi-node systems
- It is a reusable library, like libcontainer, libnetwork, vpnkit ...
- It is a plumbing part of the Docker ecosystem

SwarmKit

- [SwarmKit](#) is an open source toolkit to build multi-node systems
- It is a reusable library, like libcontainer, libnetwork, vpnkit ...
- It is a plumbing part of the Docker ecosystem



Did you know that кит means "whale" in Russian?

SwarmKit features

- Highly-available, distributed store based on [Raft](#)
(avoids depending on an external store: easier to deploy; higher performance)
- Dynamic reconfiguration of Raft without interrupting cluster operations
- Services managed with a *declarative API*
(implementing *desired state* and *reconciliation loop*)
- Integration with overlay networks and load balancing
- Strong emphasis on security:
 - automatic TLS keying and signing; automatic cert rotation
 - full encryption of the data plane; automatic key rotation
 - least privilege architecture (single-node compromise ≠ cluster compromise)
 - on-disk encryption with optional passphrase



Where is the key/value store?

- Many orchestration systems use a key/value store backed by a consensus algorithm (k8s → etcd → Raft, mesos → zookeeper → ZAB, etc.)
- SwarmKit implements the Raft algorithm directly
(Nomad is similar; thanks [@cbednarski](#), [@diptanu](#) and others for pointing it out!)
- Analogy courtesy of [@aluzzardi](#):

It's like B-Trees and RDBMS. They are different layers, often associated. But you don't need to bring up a full SQL server when all you need is to index some data.

- As a result, the orchestrator has direct access to the data
(the main copy of the data is stored in the orchestrator's memory)
- Simpler, easier to deploy and operate; also faster

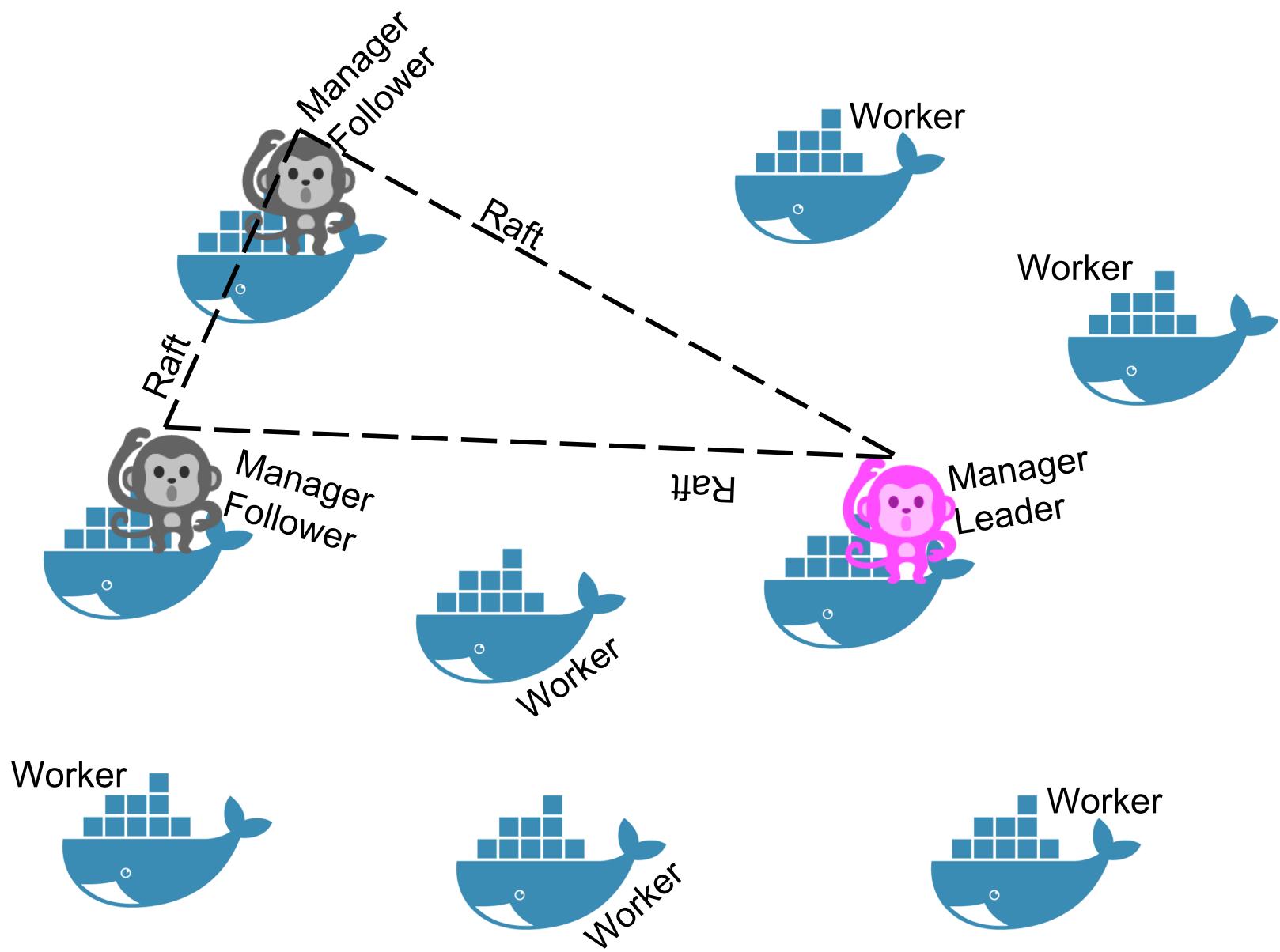
SwarmKit concepts (1/2)

- A *cluster* will be at least one *node* (preferably more)
- A *node* can be a *manager* or a *worker*
- A *manager* actively takes part in the Raft consensus, and keeps the Raft log
- You can talk to a *manager* using the SwarmKit API
- One *manager* is elected as the *leader*; other managers merely forward requests to it
- The *workers* get their instructions from the *managers*
- Both *workers* and *managers* can run containers

Illustration

On the next slide:

- whales = nodes (workers and managers)
- monkeys = managers
- purple monkey = leader
- grey monkeys = followers
- dotted triangle = raft protocol



SwarmKit concepts (2/2)

- The *managers* expose the SwarmKit API
- Using the API, you can indicate that you want to run a *service*
- A *service* is specified by its *desired state*: which image, how many instances...
- The *leader* uses different subsystems to break down services into *tasks*: orchestrator, scheduler, allocator, dispatcher
- A *task* corresponds to a specific container, assigned to a specific *node*
- *Nodes* know which *tasks* should be running, and will start or stop containers accordingly (through the Docker Engine API)

You can refer to the [NOMENCLATURE](#) in the SwarmKit repo for more details.



Declarative vs imperative

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

Declarative vs imperative

- Our container orchestrator puts a very strong emphasis on being *declarative*
- Declarative:

I would like a cup of tea.

- Imperative:

Boil some water. Pour it in a teapot. Add tea leaves. Steep for a while. Serve in a cup.

Declarative vs imperative

- Our container orchestrator puts a very strong emphasis on being *declarative*
- Declarative:

I would like a cup of tea.

- Imperative:

Boil some water. Pour it in a teapot. Add tea leaves. Steep for a while. Serve in a cup.

- Declarative seems simpler at first ...

Declarative vs imperative

- Our container orchestrator puts a very strong emphasis on being *declarative*
- Declarative:

I would like a cup of tea.

- Imperative:

Boil some water. Pour it in a teapot. Add tea leaves. Steep for a while. Serve in a cup.

- Declarative seems simpler at first ...
- ... As long as you know how to brew tea

Declarative vs imperative

- What declarative would really be:

I want a cup of tea, obtained by pouring an infusion¹ of tea leaves in a cup.

Declarative vs imperative

- What declarative would really be:

I want a cup of tea, obtained by pouring an infusion¹ of tea leaves in a cup.

¹An infusion is obtained by letting the object steep a few minutes in hot² water.

Declarative vs imperative

- What declarative would really be:

I want a cup of tea, obtained by pouring an infusion¹ of tea leaves in a cup.

¹An infusion is obtained by letting the object steep a few minutes in hot² water.

²Hot liquid is obtained by pouring it in an appropriate container³ and setting it on a stove.

Declarative vs imperative

- What declarative would really be:

I want a cup of tea, obtained by pouring an infusion¹ of tea leaves in a cup.

¹An infusion is obtained by letting the object steep a few minutes in hot² water.

²Hot liquid is obtained by pouring it in an appropriate container³ and setting it on a stove.

³Ah, finally, containers! Something we know about. Let's get to work, shall we?

Declarative vs imperative

- What declarative would really be:

I want a cup of tea, obtained by pouring an infusion¹ of tea leaves in a cup.

¹An infusion is obtained by letting the object steep a few minutes in hot² water.

²Hot liquid is obtained by pouring it in an appropriate container³ and setting it on a stove.

³Ah, finally, containers! Something we know about. Let's get to work, shall we?

Did you know there was an [ISO standard](#) specifying how to brew tea?

Declarative vs imperative

- Imperative systems:
 - simpler
 - if a task is interrupted, we have to restart from scratch
- Declarative systems:
 - if a task is interrupted (or if we show up to the party half-way through), we can figure out what's missing and do only what's necessary
 - we need to be able to *observe* the system
 - ... and compute a "diff" between *what we have* and *what we want*



Swarm mode

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

Swarm mode

- Since version 1.12, the Docker Engine embeds SwarmKit
- All the SwarmKit features are "asleep" until you enable "Swarm mode"
- Examples of Swarm Mode commands:
 - `docker swarm` (enable Swarm mode; join a Swarm; adjust cluster parameters)
 - `docker node` (view nodes; promote/demote managers; manage nodes)
 - `docker service` (create and manage services)

Swarm mode needs to be explicitly activated

- By default, all this new code is inactive
- Swarm mode can be enabled, "unlocking" SwarmKit functions (services, out-of-the-box overlay networks, etc.)



- Try a Swarm-specific command:

```
docker node ls
```

Swarm mode needs to be explicitly activated

- By default, all this new code is inactive
- Swarm mode can be enabled, "unlocking" SwarmKit functions (services, out-of-the-box overlay networks, etc.)



- Try a Swarm-specific command:

```
docker node ls
```

You will get an error message:

Error response from daemon: This node is not a swarm manager. [...]



Creating our first Swarm

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

Creating our first Swarm

- The cluster is initialized with `docker swarm init`
- This should be executed on a first, seed node
- ⚠ DO NOT execute `docker swarm init` on multiple nodes!

You would have multiple disjoint clusters.



- Create our cluster from node1:

```
docker swarm init
```

Creating our first Swarm

- The cluster is initialized with `docker swarm init`
- This should be executed on a first, seed node
-  DO NOT execute `docker swarm init` on multiple nodes!

You would have multiple disjoint clusters.



- Create our cluster from node1:

```
docker swarm init
```

If Docker tells you that it `could not choose an IP address to advertise`, see next slide!

IP address to advertise

- When running in Swarm mode, each node *advertises* its address to the others (i.e. it tells them "*you can contact me on 10.1.2.3:2377*")
- If the node has only one IP address, it is used automatically
(The addresses of the loopback interface and the Docker bridge are ignored)
- If the node has multiple IP addresses, you **must** specify which one to use
(Docker refuses to pick one randomly)
- You can specify an IP address or an interface name
(in the latter case, Docker will read the IP address of the interface and use it)
- You can also specify a port number
(otherwise, the default port 2377 will be used)

Using a non-default port number

- Changing the *advertised* port does not change the *listening* port
- If you only pass `--advertise-addr eth0:7777`, Swarm will still listen on port 2377
- You will probably need to pass `--listen-addr eth0:7777` as well
- This is to accommodate scenarios where these ports *must* be different (port mapping, load balancers...)

Example to run Swarm on a different port:

```
docker swarm init --advertise-addr eth0:7777 --listen-addr eth0:7777
```

Which IP address should be advertised?

- If your nodes have only one IP address, it's safe to let autodetection do the job

(Except if your instances have different private and public addresses, e.g. on EC2, and you are building a Swarm involving nodes inside and outside the private network: then you should advertise the public address.)
- If your nodes have multiple IP addresses, pick an address which is reachable by *every other node* of the Swarm
- If you are using [play-with-docker](#), use the IP address shown next to the node name

(This is the address of your node on your private internal overlay network. The other address that you might see is the address of your node on the `docker_gwbridge` network, which is used for outbound traffic.)

Examples:

```
docker swarm init --advertise-addr 172.24.0.2
docker swarm init --advertise-addr eth0
```



Using a separate interface for the data path

- You can use different interfaces (or IP addresses) for control and data
- You set the *control plane path* with `--advertise-addr` and `--listen-addr`

(This will be used for SwarmKit manager/worker communication, leader election, etc.)

- You set the *data plane path* with `--data-path-addr`

(This will be used for traffic between containers)

- Both flags can accept either an IP address, or an interface name

(When specifying an interface name, Docker will use its first IP address)

Token generation

- In the output of `docker swarm init`, we have a message confirming that our node is now the (single) manager:

Swarm initialized: current node (8jud...) is now a manager.

- Docker generated two security tokens (like passphrases or passwords) for our cluster
- The CLI shows us the command to use on other nodes to add them to the cluster using the "worker" security token:

To add a worker to this swarm, run the following command:

```
docker swarm join \
--token SWMTKN-1-59f14ak4nqjmao1ofttrc4eprhrola2187... \
172.31.4.182:2377
```



Checking that Swarm mode is enabled



- Run the traditional `docker info` command:

```
docker info
```

The output should include:

Swarm: active

NodeID: 8jud7o8dax3zxbags3f8yox4b

Is Manager: true

ClusterID: 2vcw2oa9rjps3a24m91xhv0c

...

Running our first Swarm mode command

- Let's retry the exact same command as earlier



- List the nodes (well, the only node) of our cluster:

```
docker node ls
```

The output should look like the following:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
8jud...ox4b *	node1	Ready	Active	Leader

Adding nodes to the Swarm

- A cluster with one node is not a lot of fun
- Let's add `node2`!
- We need the token that was shown earlier

Adding nodes to the Swarm

- A cluster with one node is not a lot of fun
- Let's add `node2`!
- We need the token that was shown earlier
- You wrote it down, right?

Adding nodes to the Swarm

- A cluster with one node is not a lot of fun
- Let's add `node2`!
- We need the token that was shown earlier
- You wrote it down, right?
- Don't panic, we can easily see it again 😊

Adding nodes to the Swarm



- Show the token again:

```
docker swarm join-token worker
```

- Log into node2:

```
ssh node2
```

- Copy-paste the docker swarm join ... command
(that was displayed just before)



Check that the node was added correctly

- Stay on node2 for now!



- ```
>
```
- We can still use docker info to verify that the node is part of the Swarm:

```
docker info | grep ^Swarm
```

- However, Swarm commands will not work; try, for instance:

```
docker node ls
```

- This is because the node that we added is currently a *worker*
- Only *managers* can accept Swarm-specific commands

# View our two-node cluster

- Let's go back to node1 and see what our cluster looks like



- ```
>
```
- Switch back to node1 (with exit, Ctrl-D ...)
 - View the cluster from node1, which is a manager:

```
docker node ls
```

The output should be similar to the following:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER	STATUS
8jud...ox4b *	node1	Ready	Active		Leader
ehb0...4fvx	node2	Ready	Active		

Under the hood: docker swarm init

When we do `docker swarm init`:

- a keypair is created for the root CA of our Swarm
- a keypair is created for the first node
- a certificate is issued for this node
- the join tokens are created

Under the hood: join tokens

There is one token to *join as a worker*, and another to *join as a manager*.

The join tokens have two parts:

- a secret key (preventing unauthorized nodes from joining)
- a fingerprint of the root CA certificate (preventing MITM attacks)

If a token is compromised, it can be rotated instantly with:

```
docker swarm join-token --rotate <worker|manager>
```

Under the hood: docker swarm join

When a node joins the Swarm:

- it is issued its own keypair, signed by the root CA
- if the node is a manager:
 - it joins the Raft consensus
 - it connects to the current leader
 - it accepts connections from worker nodes
- if the node is a worker:
 - it connects to one of the managers (leader or follower)

Under the hood: cluster communication

- The *control plane* is encrypted with AES-GCM; keys are rotated every 12 hours
 - (`docker swarm update` allows to change this delay or to use an external CA)
- The *data plane* (communication between containers) is not encrypted by default
 - (but this can be activated on a by-network basis, using IPSEC, leveraging hardware crypto if available)

Under the hood: I want to know more!

Revisit SwarmKit concepts:

- Docker 1.12 Swarm Mode Deep Dive Part 1: Topology ([video](#))
- Docker 1.12 Swarm Mode Deep Dive Part 2: Orchestration ([video](#))

Some presentations from the Docker Distributed Systems Summit in Berlin:

- Heart of the SwarmKit: Topology Management ([slides](#))
- Heart of the SwarmKit: Store, Topology & Object Model ([slides](#)) ([video](#))

And DockerCon Black Belt talks:

- DC17US: Everything You Thought You Already Knew About Orchestration ([video](#))
- DC17EU: Container Orchestration from Theory to Practice ([video](#))

Adding more manager nodes

- Right now, we have only one manager (node1)
- If we lose it, we lose quorum - and that's *very bad!*
- Containers running on other nodes will be fine ...
- But we won't be able to get or set anything related to the cluster
- If the manager is permanently gone, we will have to do a manual repair!
- Nobody wants to do that ... so let's make our cluster highly available

Adding more managers

With Play-With-Docker:

```
TOKEN=$(docker swarm join-token -q manager)
for N in $(seq 3 5); do
    export DOCKER_HOST=tcp://node$N:2375
    docker swarm join --token $TOKEN node1:2377
done
unset DOCKER_HOST
```

Controlling the Swarm from other nodes



- Try the following command on a few different nodes:

```
docker node ls
```

On manager nodes:

you will see the list of nodes, with a * denoting the node you're talking to.

On non-manager nodes:

you will get an error message telling you that the node is not a manager.

As we saw earlier, you can only control the Swarm through a manager node.

Play-With-Docker node status icon

- If you're using Play-With-Docker, you get node status icons
- Node status icons are displayed left of the node name
 - No icon = no Swarm mode detected
 - Solid blue icon = Swarm manager detected
 - Blue outline icon = Swarm worker detected

```
[node1] (local)
$ docker swarm join --token SWMT-1234567890 10.0.16.3:23
To add a worker

docker swarm
--token SWMT
10.0.16.3:23

[node1] (local)
$ 
```

Dynamically changing the role of a node

- We can change the role of a node on the fly:

```
docker node promote nodeX → make nodeX a manager
```

```
docker node demote nodeX → make nodeX a worker
```



- See the current list of nodes:

```
docker node ls
```

- Promote any worker node to be a manager:

```
docker node promote <node_name_or_id>
```

How many managers do we need?

- $2N+1$ nodes can (and will) tolerate N failures
(you can have an even number of managers, but there is no point)

How many managers do we need?

- $2N+1$ nodes can (and will) tolerate N failures
(you can have an even number of managers, but there is no point)
- 1 manager = no failure
- 3 managers = 1 failure
- 5 managers = 2 failures (or 1 failure during 1 maintenance)
- 7 managers and more = now you might be overdoing it for most designs

see [Docker's admin guide](#) on node failure and datacenter redundancy

Why not have *all* nodes be managers?

- With Raft, writes have to go to (and be acknowledged by) all nodes
- Thus, it's harder to reach consensus in larger groups
- Only one manager is Leader (writable), so more managers ≠ more capacity
- Managers should be < 10ms latency from each other
- These design parameters lead us to recommended designs

What would McGyver do?

- Keep managers in one region (multi-zone/datacenter/rack)
- Groups of 3 or 5 nodes: all are managers. Beyond 5, separate out managers and workers
- Groups of 10-100 nodes: pick 5 "stable" nodes to be managers
- Groups of more than 100 nodes: watch your managers' CPU and RAM
 - 16GB memory or more, 4 CPU's or more, SSD's for Raft I/O
 - otherwise, break down your nodes in multiple smaller clusters

Cloud pro-tip: use separate auto-scaling groups for managers and workers

See docker's "[Running Docker at scale](#)" document

What's the upper limit?

- We don't know!
- Internal testing at Docker Inc.: 1000-10000 nodes is fine
 - deployed to a single cloud region
 - one of the main take-aways was "*you're gonna need a bigger manager*"
- Testing by the community: [4700 heterogeneous nodes all over the 'net](#)
 - it just works, assuming they have the resources
 - more nodes require manager CPU and networking; more containers require RAM
 - scheduling of large jobs (70,000 containers) is slow, though ([getting better!](#))

Real-life deployment methods

Real-life deployment methods

Running commands manually over SSH

Real-life deployment methods

Running commands manually over SSH

(lol jk)

Real-life deployment methods

Running commands manually over SSH

(lol jk)

- Using your favorite configuration management tool
- [Docker for AWS](#)
- [Docker for Azure](#)



531/634

Running our first Swarm service

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

Running our first Swarm service

- How do we run services? Simplified version:

```
docker run → docker service create
```



- Create a service featuring an Alpine container pinging Google resolvers:

```
docker service create --name pingpong alpine ping 8.8.8.8
```

- Check the result:

```
docker service ps pingpong
```

Checking service logs

(New in Docker Engine 17.05)

- Just like `docker logs` shows the output of a specific local container ...
- ... `docker service logs` shows the output of all the containers of a specific service



- Check the output of our ping command:

```
docker service logs pingpong
```

Flags `--follow` and `--tail` are available, as well as a few others.

Note: by default, when a container is destroyed (e.g. when scaling down), its logs are lost.



Looking up where our container is running

- The `docker service ps` command told us where our container was scheduled



- Look up the `NODE` on which the container is running:

```
docker service ps pingpong
```

- If you use Play-With-Docker, switch to that node's tab, or set `DOCKER_HOST`
- Otherwise, `ssh` into that node or use `$(eval docker-machine env node...)`



Viewing the logs of the container



- See that the container is running and check its ID:

```
docker ps
```

- View its logs:

```
docker logs containerID
```

- Go back to node1 afterwards

Scale our service

- Services can be scaled in a pinch with the `docker service update` command



- Scale the service to ensure 2 copies per node:

```
docker service update pingpong --replicas 6
```

- Check that we have two containers on the current node:

```
docker ps
```

Monitoring deployment progress with `--detach`

(New in Docker Engine 17.10)

- The CLI monitors commands that create/update/delete services
- In effect, `--detach=false` is the default
 - synchronous operation
 - the CLI will monitor and display the progress of our request
 - it exits only when the operation is complete
 - Ctrl-C to detach at anytime
- `--detach=true`
 - asynchronous operation
 - the CLI just submits our request
 - it exits as soon as the request is committed into Raft

To `--detach` or not to `--detach`

- `--detach=false`
 - great when experimenting, to see what's going on
 - also great when orchestrating complex deployments
(when you want to wait for a service to be ready before starting another)
 - `--detach=true`
 - great for independent operations that can be parallelized
 - great in headless scripts (where nobody's watching anyway)
-  `--detach=true` does not complete *faster*. It just *doesn't wait* for completion.



--detach over time

- Docker Engine 17.10 and later: the default is `--detach=false`
- From Docker Engine 17.05 to 17.09: the default is `--detach=true`
- Prior to Docker 17.05: `--detach` doesn't exist

(You can watch progress with e.g. `watch docker service ps <serviceID>`)

--detach in action



- Scale the service to ensure 3 copies per node:

```
docker service update pingpong --replicas 9 --detach=false
```

- And then to 4 copies per node:

```
docker service update pingpong --replicas 12 --detach=true
```

Expose a service

- Services can be exposed, with two special properties:
 - the public port is available on *every node of the Swarm*,
 - requests coming on the public port are load balanced across all instances.
- This is achieved with option `-p/--publish`; as an approximation:

```
docker run -p → docker service create -p
```

- If you indicate a single port number, it will be mapped on a port starting at 30000 (vs. 32768 for single container mapping)
- You can indicate two port numbers to set the public port number (just like with `docker run -p`)

Expose ElasticSearch on its default port



- Create an ElasticSearch service (and give it a name while we're at it):

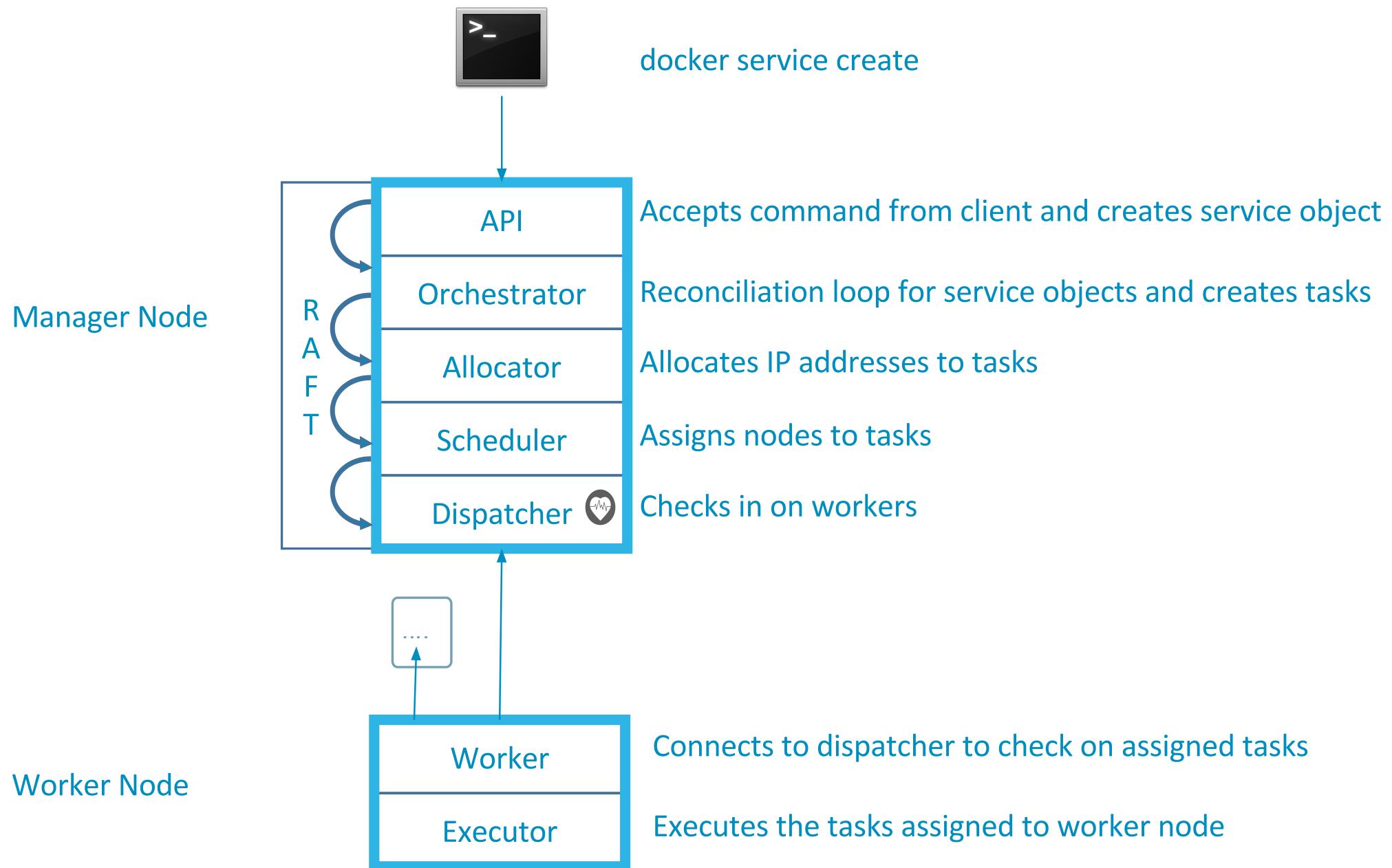
```
docker service create --name search --publish 9200:9200 --replicas 5 \
    elasticsearch:2
```

Note: don't forget the :2!

The latest version of the ElasticSearch image won't start without mandatory configuration.

Tasks lifecycle

- During the deployment, you will be able to see multiple states:
 - assigned (the task has been assigned to a specific node)
 - preparing (this mostly means "pulling the image")
 - starting
 - running
- When a task is terminated (stopped, killed...) it cannot be restarted
(A replacement task will be created)



Test our service

- We mapped port 9200 on the nodes, to port 9200 in the containers
- Let's try to reach that port!



- Try the following command:

```
curl localhost:9200
```

(If you get `Connection refused`: congratulations, you are very fast indeed! Just try again.)

ElasticSearch serves a little JSON document with some basic information about this instance; including a randomly-generated super-hero name.

Test the load balancing

- If we repeat our `curl` command multiple times, we will see different names



- Send 10 requests, and see which instances serve them:

```
for N in $(seq 1 10); do
    curl -s localhost:9200 | jq .name
done
```

Note: if you don't have `jq` on your Play-With-Docker instance, just install it:

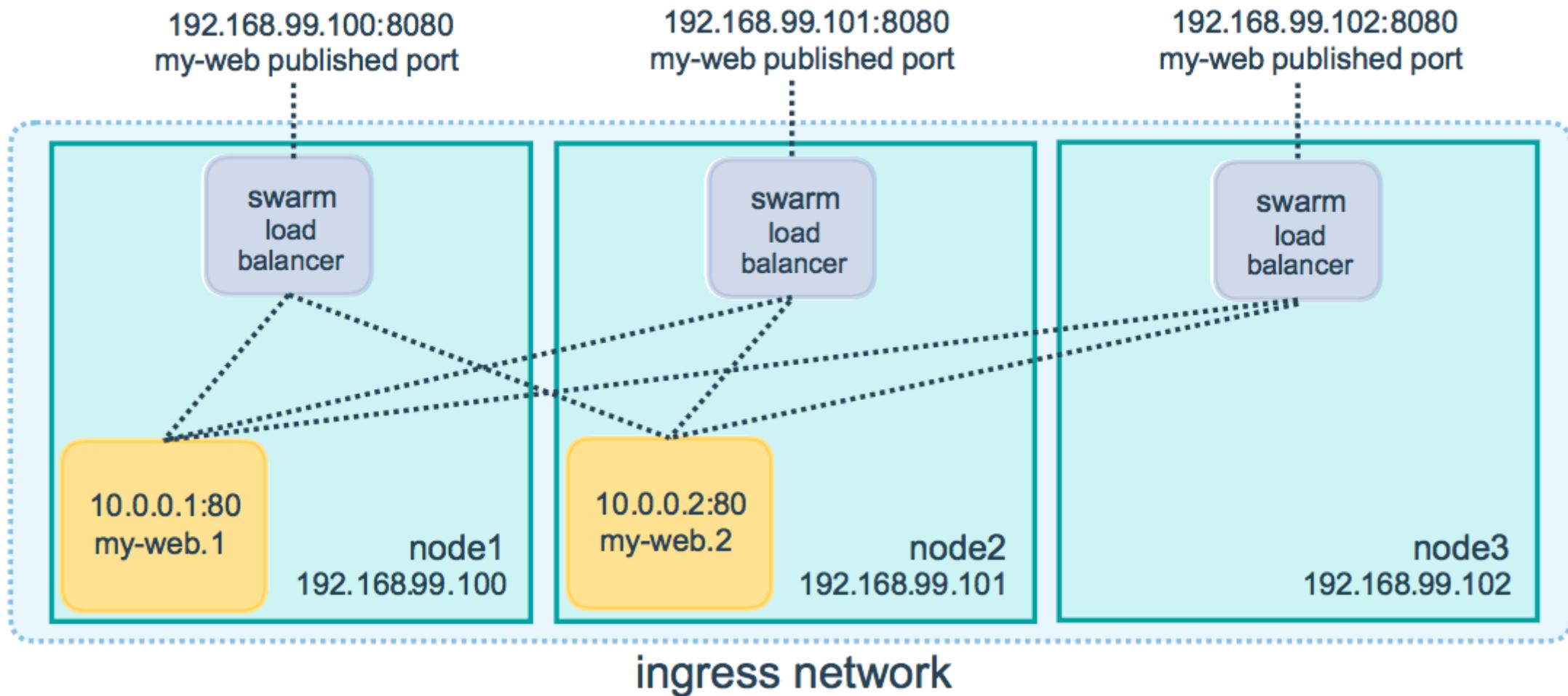
```
apk add --no-cache jq
```

Load balancing results

Traffic is handled by our clusters [routing mesh](#).

Each request is served by one of the instances, in rotation.

Note: if you try to access the service from your browser, you will probably see the same instance name over and over, because your browser (unlike curl) will try to re-use the same connection.



Under the hood of the routing mesh

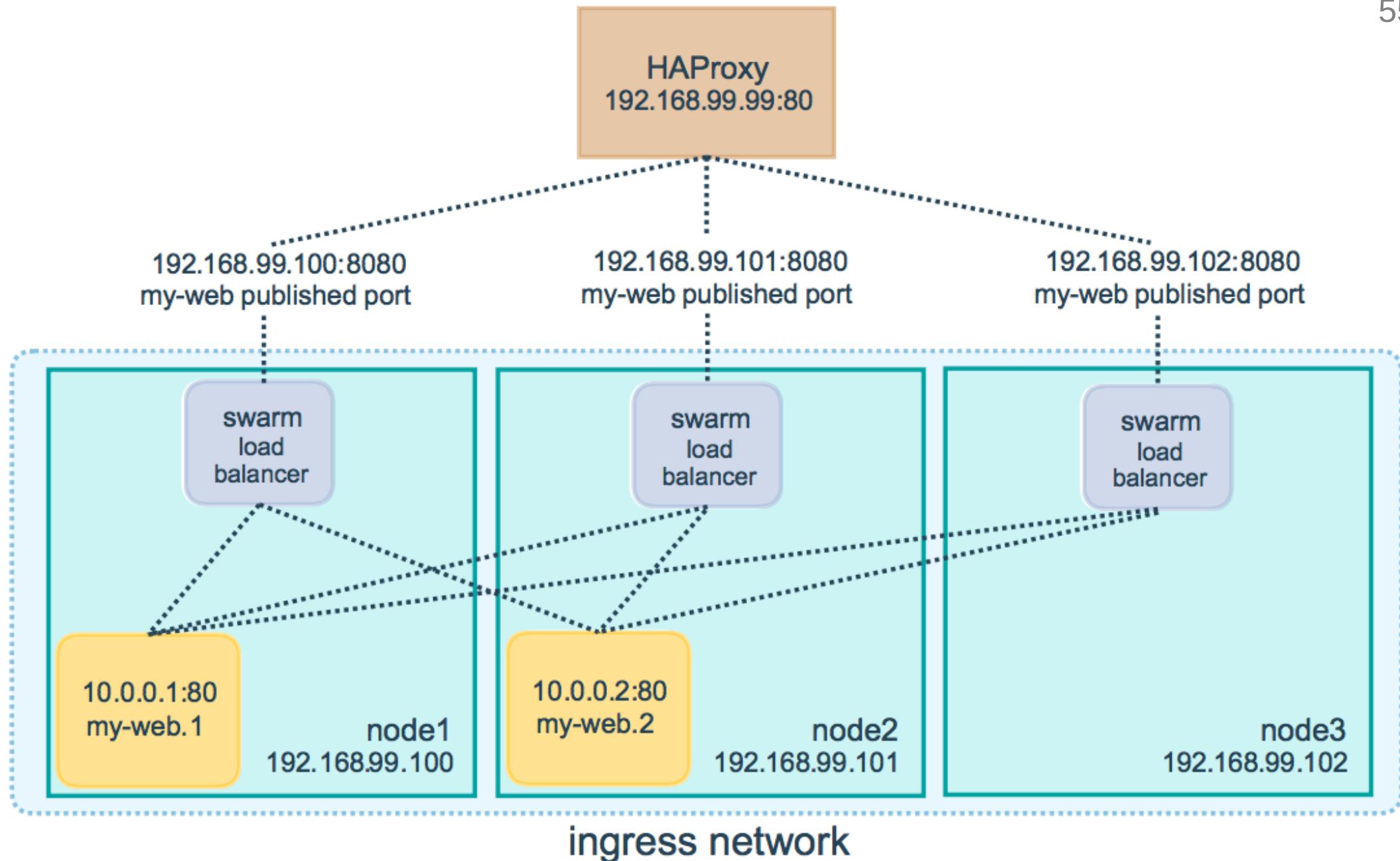
- Load balancing is done by IPVS
- IPVS is a high-performance, in-kernel load balancer
- It's been around for a long time (merged in the kernel since 2.4)
- Each node runs a local load balancer

(Allowing connections to be routed directly to the destination, without extra hops)

Managing inbound traffic

There are many ways to deal with inbound traffic on a Swarm cluster.

- Put all (or a subset) of your nodes in a DNS A record (good for web clients)
- Assign your nodes (or a subset) to an external load balancer (ELB, etc.)
- Use a virtual IP and make sure that it is assigned to an "alive" node
- etc.



Managing HTTP traffic

- The TCP routing mesh doesn't parse HTTP headers
- If you want to place multiple HTTP services on port 80/443, you need something more
- You can set up NGINX or HAProxy on port 80/443 to route connections to the correct Service, but they need to be "Swarm aware" to dynamically update configs

Managing HTTP traffic

- The TCP routing mesh doesn't parse HTTP headers
- If you want to place multiple HTTP services on port 80/443, you need something more
- You can set up NGINX or HAProxy on port 80/443 to route connections to the correct Service, but they need to be "Swarm aware" to dynamically update configs
- Docker EE provides its own [Layer 7 routing](#)
 - Service labels like `com.docker.lb.hosts=<FQDN>` are detected automatically via Docker API and dynamically update the configuration

Managing HTTP traffic

- The TCP routing mesh doesn't parse HTTP headers
- If you want to place multiple HTTP services on port 80/443, you need something more
- You can set up NGINX or HAProxy on port 80/443 to route connections to the correct Service, but they need to be "Swarm aware" to dynamically update configs
- Docker EE provides its own [Layer 7 routing](#)
 - Service labels like `com.docker.lb.hosts=<FQDN>` are detected automatically via Docker API and dynamically update the configuration
- Two common open source options:
 - [Traefik](#) - popular, many features, requires running on managers, needs key/value for HA
 - [Docker Flow Proxy](#) - uses HAProxy, made for Swarm by Docker Captain [@vfarcic](#)

You should use labels

- Labels are a great way to attach arbitrary information to services
- Examples:
 - HTTP vhost of a web app or web service
 - backup schedule for a stateful service
 - owner of a service (for billing, paging...)
 - correlate Swarm objects together (services, volumes, configs, secrets, etc.)

Pro-tip for ingress traffic management

- It is possible to use *local* networks with Swarm services
- This means that you can do something like this:

```
docker service create --network host --mode global traefik ...
```

(This runs the `traefik` load balancer on each node of your cluster, in the `host` network)

- This gives you native performance (no iptables, no proxy, no nothing!)
- The load balancer will "see" the clients' IP addresses
- But: a container cannot simultaneously be in the `host` network and another network

(You will have to route traffic to containers using exposed ports or UNIX sockets)



Using local networks (host, macvlan ...)

- It is possible to connect services to local networks
- Using the host network is fairly straightforward
 - (With the caveats described on the previous slide)
- Other network drivers are a bit more complicated
 - (IP allocation may have to be coordinated between nodes)
- See for instance [this guide](#) to get started on macvlan
- See [this PR](#) for more information about local network drivers in Swarm mode

Visualize container placement

- Let's leverage the Docker API!



- Run this simple-yet-beautiful visualization app:

```
cd ~/container.training/stacks  
docker-compose -f visualizer.yml up -d
```

Connect to the visualization webapp

- It runs a web server on port 8080



- Point your browser to port 8080 of your node1's public ip
(If you use Play-With-Docker, click on the (8080) badge)

- The webapp updates the display automatically (you don't need to reload the page)
- It only shows Swarm services (not standalone containers)
- It shows when nodes go down
- It has some glitches (it's not Carrier-Grade Enterprise-Compliant ISO-9001 software)

Why This Is More Important Than You Think

- The visualizer accesses the Docker API *from within a container*
- This is a common pattern: run container management tools *in containers*
- Instead of viewing your cluster, this could take care of logging, metrics, autoscaling ...
- We can run it within a service, too! We won't do it yet, but the command would look like:

```
docker service create \  
  --mount source=/var/run/docker.sock,type=bind,target=/var/run/docker.sock \  
  --name viz --constraint node.role==manager ...
```

Credits: the visualization code was written by [Francisco Miranda](#).

[Mano Marks](#) adapted it to Swarm and maintains it.

Terminate our services

- Before moving on, we will remove those services
- `docker service rm` can accept multiple services names or IDs
- `docker service ls` can accept the `-q` flag
- A Shell snippet a day keeps the cruft away



- Remove all services with this one liner:

```
docker service ls -q | xargs docker service rm
```

How did we make our app "Swarm-ready"?

This app was written in June 2015. (One year before Swarm mode was released.)

What did we change to make it compatible with Swarm mode?

How did we make our app "Swarm-ready"?

This app was written in June 2015. (One year before Swarm mode was released.)

What did we change to make it compatible with Swarm mode?



- Go to the app directory:

```
cd ~/container.training/dockercoins
```

- See modifications in the code:

```
git log -p --since "4-JUL-2015" -- . ':!*.*' ':!*.*'
```

Which files have been changed since then?

- Compose files
- HTML file (it contains an embedded contextual tweet)
- Dockerfiles (to switch to smaller images)
- That's it!

Which files have been changed since then?

- Compose files
- HTML file (it contains an embedded contextual tweet)
- Dockerfiles (to switch to smaller images)
- That's it!

We didn't change a single line of code in this app since it was written.

Which files have been changed since then?

- Compose files
- HTML file (it contains an embedded contextual tweet)
- Dockerfiles (to switch to smaller images)
- That's it!

We didn't change a single line of code in this app since it was written.

*The images that were [built in June 2015](#) (when the app was written) can still run today ...
... in Swarm mode (distributed across a cluster, with load balancing) ...
... without any modification.*

How did we design our app in the first place?

- [Twelve-Factor App](#) principles
- Service discovery using DNS names
 - Initially implemented as "links"
 - Then "ambassadors"
 - And now "services"
- Existing apps might require more changes!

Integration with Compose

- We saw how to manually build, tag, and push images to a registry
- But ...

Integration with Compose

- We saw how to manually build, tag, and push images to a registry
- But ...

"I'm so glad that my deployment relies on ten nautic miles of Shell scripts"

(No-one, ever)

Integration with Compose

- We saw how to manually build, tag, and push images to a registry
- But ...

"I'm so glad that my deployment relies on ten nautic miles of Shell scripts"

(No-one, ever)

- Let's see how we can streamline this process!



Swarm Stacks

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

Swarm Stacks

- Compose is great for local development
- It can also be used to manage image lifecycle
 - (i.e. build images and push them to a registry)
- Compose files v2 are great for local development
- Compose files v3 can also be used for production deployments!

Compose file version 3

(New in Docker Engine 1.13)

- Almost identical to version 2
- Can be directly used by a Swarm cluster through `docker stack ...` commands
- Introduces a `deploy` section to pass Swarm-specific parameters
- Resource limits are moved to this `deploy` section
- See [here](#) for the complete list of changes
- Supersedes *Distributed Application Bundles*

(JSON payload describing an application; could be generated from a Compose file)

Our first stack

We need a registry to move images around.

Without a stack file, it would be deployed with the following command:

```
docker service create --publish 5000:5000 registry
```

Now, we are going to deploy it with the following stack file:

```
version: "3"

services:
  registry:
    image: registry
    ports:
      - "5000:5000"
```

Checking our stack files

- All the stack files that we will use are in the `stacks` directory



- Go to the `stacks` directory:

```
cd ~/container.training/stacks
```

- Check `registry.yml`:

```
cat registry.yml
```

Deploying our first stack

- All stack manipulation commands start with `docker stack`
- Under the hood, they map to `docker service` commands
- Stacks have a *name* (which also serves as a namespace)
- Stacks are specified with the aforementioned Compose file format version 3



- Deploy our local registry:

```
docker stack deploy --compose-file registry.yml registry
```

Inspecting stacks

- `docker stack ps` shows the detailed state of all services of a stack



- Check that our registry is running correctly:

```
docker stack ps registry
```

- Confirm that we get the same output with the following command:

```
docker service ps registry_registry
```

Specifics of stack deployment

Our registry is not *exactly* identical to the one deployed with `docker service create!`

- Each stack gets its own overlay network
- Services of the stack are connected to this network
(unless specified differently in the Compose file)
- Services get network aliases matching their name in the Compose file
(just like when Compose brings up an app specified in a v2 file)
- Services are explicitly named `<stack_name>_<service_name>`
- Services and tasks also get an internal label indicating which stack they belong to

Testing our local registry

- Connecting to port 5000 *on any node of the cluster* routes us to the registry
- Therefore, we can use `localhost:5000` or `127.0.0.1:5000` as our registry



- Issue the following API request to the registry:

```
curl 127.0.0.1:5000/v2/_catalog
```

It should return:

```
{"repositories":[]}
```

If that doesn't work, retry a few times; perhaps the container is still starting.

Pushing an image to our local registry

- We can retag a small image, and push it to the registry



- Make sure we have the busybox image, and retag it:

```
docker pull busybox
docker tag busybox 127.0.0.1:5000/busybox
```

- Push it:

```
docker push 127.0.0.1:5000/busybox
```

Checking what's on our local registry

- The registry API has endpoints to query what's there



- Ensure that our busybox image is now in the local registry:

```
curl http://127.0.0.1:5000/v2/_catalog
```

The curl command should now output:

```
"repositories": ["busybox"]}
```

Building and pushing stack services

- When using Compose file version 2 and above, you can specify *both* build and image
- When both keys are present:
 - Compose does "business as usual" (uses build)
 - but the resulting image is named as indicated by the image key
(instead of <projectname>_<servicename>:latest)
 - it can be pushed to a registry with docker-compose push
- Example:

```
webfront:  
  build: www  
  image: myregistry.company.net:5000/webfront
```

Using Compose to build and push images



- Try it:

```
docker-compose -f dockercoins.yml build  
docker-compose -f dockercoins.yml push
```

Let's have a look at the `dockercoins.yml` file while this is building and pushing.

```
version: "3"

services:
  rng:
    build: dockercoins/rng
    image: ${REGISTRY-127.0.0.1:5000}/rng:${TAG-latest}
    deploy:
      mode: global
...
  redis:
    image: redis
...
  worker:
    build: dockercoins/worker
    image: ${REGISTRY-127.0.0.1:5000}/worker:${TAG-latest}
...
  deploy:
    replicas: 10
```

Deploying the application

- Now that the images are on the registry, we can deploy our application stack



- Create the application stack:

```
docker stack deploy --compose-file dockercoins.yml dockercoins
```

We can now connect to any of our nodes on port 8000, and we will see the familiar hashing speed graph.

Maintaining multiple environments

There are many ways to handle variations between environments.

- Compose loads `docker-compose.yml` and (if it exists) `docker-compose.override.yml`
- Compose can load alternate file(s) by setting the `-f` flag or the `COMPOSE_FILE` environment variable
- Compose files can *extend* other Compose files, selectively including services:

```
web:  
  extends:  
    file: common-services.yml  
    service: webapp
```

See [this documentation page](#) for more details about these techniques.



Good to know ...

- Compose file version 3 adds the `deploy` section
- Further versions (3.1, ...) add more features (secrets, configs ...)
- You can re-run `docker stack deploy` to update a stack
- You can make manual changes with `docker service update` ...
- ... But they will be wiped out each time you `docker stack deploy`

(That's the intended behavior, when one thinks about it!)

- `extends` doesn't work with `docker stack deploy`
- (But you can use `docker-compose config` to "flatten" your configuration)

Summary

- We've seen how to set up a Swarm
- We've used it to host our own registry
- We've built our app container images
- We've used the registry to host those images
- We've deployed and scaled our application
- We've seen how to use Compose to streamline deployments
- Awesome job, team!

Secret management

- Docker has a "secret safe" (secure key → value store)
- You can create as many secrets as you like
- You can associate secrets to services
- Secrets are exposed as plain text files, but kept in memory only (using `tmpfs`)
- Secrets are immutable (at least in Engine 1.13)
- Secrets have a max size of 500 KB

Creating secrets

- Must specify a name for the secret; and the secret itself



- Assign **one of the four most commonly used passwords** to a secret called `hackme`:

```
echo love | docker secret create hackme -
```

If the secret is in a file, you can simply pass the path to the file.

(The special path `-` indicates to read from the standard input.)

Creating better secrets

- Picking lousy passwords always leads to security breaches



- Let's craft a better password, and assign it to another secret:

```
base64 /dev/urandom | head -c16 | docker secret create arewesecureyet -
```

Note: in the latter case, we don't even know the secret at this point. But Swarm does.

Using secrets

- Secrets must be handed explicitly to services



- Create a dummy service with both secrets:

```
docker service create \  
    --secret hackme --secret arewesecureyet \  
    --name dummyservice \  
    --constraint node.hostname==$HOSTNAME \  
    alpine sleep 100000000
```

We constrain the container to be on the local node for convenience.

(We are going to use `docker exec` in just a moment!)

Accessing secrets

- Secrets are materialized on `/run/secrets` (which is an in-memory filesystem)



- Find the ID of the container for the dummy service:

```
CID=$(docker ps -q --filter label=com.docker.swarm.service.name=dummyservice)
```

- Enter the container:

```
docker exec -ti $CID sh
```

- Check the files in `/run/secrets`

Rotating secrets

- You can't change a secret

(Sounds annoying at first; but allows clean rollbacks if a secret update goes wrong)

- You can add a secret to a service with `docker service update --secret-add`

(This will redeploy the service; it won't add the secret on the fly)

- You can remove a secret with `docker service update --secret-rm`

- Secrets can be mapped to different names by expressing them with a micro-format:

```
docker service create --secret source=secretname,target=filename
```

Changing our insecure password

- We want to replace our `hackme` secret with a better one



- Remove the insecure `hackme` secret:

```
docker service update dummyservice --secret-rm hackme
```

- Add our better secret instead:

```
docker service update dummyservice \
--secret-add source=arewesecureyet,target=hackme
```

Wait for the service to be fully updated with e.g. `watch docker service ps dummyservice`.
(With Docker Engine 17.10 and later, the CLI will wait for you!)

Checking that our password is now stronger

- We will use the power of docker exec!



- Get the ID of the new container:

```
CID=$(docker ps -q --filter label=com.docker.swarm.service.name=dummyservice)
```

- Check the contents of the secret files:

```
docker exec $CID grep -r . /run/secrets
```

Secrets in practice

- Can be (ab)used to hold whole configuration files if needed
- If you intend to rotate secret `foo`, call it `foo.N` instead, and map it to `foo`
(N can be a serial, a timestamp...)

```
docker service create --secret source=foo.N,target=foo ...
```

- You can update (remove+add) a secret in a single command:

```
docker service update ... --secret-rm foo.M --secret-add source=foo.N,target=foo
```

- For more details and examples, [check the documentation](#)



CI/CD for Docker and orchestration

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

CI/CD for Docker and orchestration

A quick note about continuous integration and deployment

- This lab won't have you building out CI/CD pipelines
- We're cheating a bit by building images on server hosts and not in CI tool
- Docker and orchestration works with all the CI and deployment tools

CI/CD general process

- Have your CI build your images, run tests *in them*, then push to registry
- If you security scan, do it then on your images after tests but before push
- Optionally, have CI do continuous deployment if build/test/push is successful
- CD tool would SSH into nodes, or use docker cli against remote engine
- If supported, it could use docker engine TCP API (swarm API is built-in)
- Docker KBase [Development Pipeline Best Practices](#)
- Docker KBase [Continuous Integration with Docker Hub](#)
- Docker KBase [Building a Docker Secure Supply Chain](#)

Continuous Integration & Delivery Workflow

DEVELOPERS

IT OPS

BUILD

Development Environments

Version control



Visual Studio

IntelliJ IDEA



Developers

SHIP

Secure Content & Collaboration



Registry
(Docker hub / DTR)

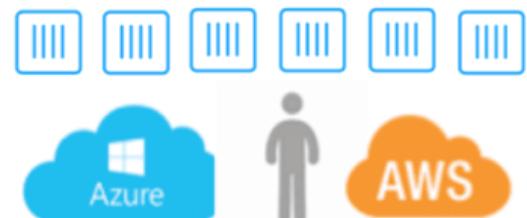


GitLab



RUN

All environments



QA / Staging / Prod

Testers /
Users





Updating services

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

Updating services

- We want to make changes to the web UI
- The process is as follows:
 - edit code
 - build new image
 - ship new image
 - run new image

Updating a single service with service update

- To update a single service, we could do the following:

```
export REGISTRY=127.0.0.1:5000
export TAG=v0.2
IMAGE=$REGISTRY/dockercoins_webui:$TAG
docker build -t $IMAGE webui/
docker push $IMAGE
docker service update dockercoins_webui --image $IMAGE
```

- Make sure to tag properly your images: update the `TAG` at each iteration

(When you check which images are running, you want these tags to be uniquely identifiable)

Updating services with stack deploy

- With the Compose integration, all we have to do is:

```
export TAG=v0.2  
docker-compose -f composefile.yml build  
docker-compose -f composefile.yml push  
docker stack deploy -c composefile.yml nameofstack
```

Updating services with stack deploy

- With the Compose integration, all we have to do is:

```
export TAG=v0.2
docker-compose -f composefile.yml build
docker-compose -f composefile.yml push
docker stack deploy -c composefile.yml nameofstack
```

- That's exactly what we used earlier to deploy the app
- We don't need to learn new commands!
- It will diff each service and only update ones that changed

Changing the code

- Let's make the numbers on the Y axis bigger!



- Update the size of text on our webui:

```
sed -i "s/15px/50px/" dockercoins/webui/files/index.html
```

Build, ship, and run our changes

- Four steps:
 1. Set (and export!) the TAG environment variable
 2. docker-compose build
 3. docker-compose push
 4. docker stack deploy



- Build, ship, and run:

```
export TAG=v0.2
docker-compose -f dockercoins.yml build
docker-compose -f dockercoins.yml push
docker stack deploy -c dockercoins.yml dockercoins
```

- Because we're tagging all images in this demo v0.2, deploy will update all apps, FYI

Viewing our changes

- Wait at least 10 seconds (for the new version to be deployed)
- Then reload the web UI
- Or just mash "reload" frantically
- ... Eventually the legend on the left will be bigger!



Rolling updates

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

Rolling updates

- Let's force an update on hasher to watch it update



- First lets scale up hasher to 7 replicas:

```
docker service scale dockercoins_hasher=7
```

- Force a rolling update (replace containers) to different image:

```
docker service update --image 127.0.0.1:5000 hasher:v0.1 dockercoins_hasher
```

- You can run `docker events` in a separate `node1` shell to see Swarm actions
- You can use `--force` to replace containers without a config change

Changing the upgrade policy

- We can change many options on how updates happen



- Change the parallelism to 2, and the max failed container updates to 25%:

```
docker service update --update-parallelism 2 \
--update-max-failure-ratio .25 dockercoins_hasher
```

- No containers were replaced, this is called a "no op" change
- Service metadata-only changes don't require orchestrator operations

Changing the policy in the Compose file

- The policy can also be updated in the Compose file
- This is done by adding an `update_config` key under the `deploy` key:

```
deploy:  
  replicas: 10  
  update_config:  
    parallelism: 2  
    delay: 10s
```

Rolling back

- At any time (e.g. before the upgrade is complete), we can rollback:
 - by editing the Compose file and redeploying
 - by using the special `--rollback` flag with `service update`
 - by using `docker service rollback`



- Try to rollback the webui service:

```
docker service rollback dockercoins_webui
```

What happens with the web UI graph?

The fine print with rollback

- Rollback reverts to the previous service definition
 - see PreviousSpec in `docker service inspect <servicename>`
- If we visualize successive updates as a stack:
 - it doesn't "pop" the latest update
 - it "pushes" a copy of the previous update on top
 - ergo, rolling back twice does nothing
- "Service definition" includes rollout cadence
- Each `docker service update` command = a new service definition



Timeline of an upgrade

- SwarmKit will upgrade N instances at a time
(following the `update-parallelism` parameter)
- New tasks are created, and their desired state is set to `Ready`
(this pulls the image if necessary, ensures resource availability, creates the container ... without starting it)
- If the new tasks fail to get to `Ready` state, go back to the previous step
(SwarmKit will try again and again, until the situation is addressed or desired state is updated)
- When the new tasks are `Ready`, it sets the old tasks desired state to `Shutdown`
- When the old tasks are `Shutdown`, it starts the new tasks
- Then it waits for the `update-delay`, and continues with the next batch of instances



Health checks and auto- rollbacks

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

Health checks and auto-rollbacks

(New in Docker Engine 1.12)

- Commands that are executed on regular intervals in a container
- Must return 0 or 1 to indicate "all is good" or "something's wrong"
- Must execute quickly (timeouts = failures)
- Example:

```
curl -f http://localhost/_ping || false
```

- the `-f` flag ensures that `curl` returns non-zero for 404 and similar errors
- `|| false` ensures that any non-zero exit status gets mapped to 1
- `curl` must be installed in the container that is being checked

Defining health checks

- In a Dockerfile, with the **HEALTHCHECK** instruction

```
HEALTHCHECK --interval=1s --timeout=3s CMD curl -f http://localhost/ || false
```

- From the command line, when running containers or services

```
docker run --health-cmd "curl -f http://localhost/ || false" ...
docker service create --health-cmd "curl -f http://localhost/ || false" ...
```

- In Compose files, with a per-service **healthcheck** section

```
www:
  image: helloworldapp
  healthcheck:
    test: "curl -f https://localhost/ || false"
    timeout: 3s
```

Using health checks

- With `docker run`, health checks are purely informative
 - `docker ps` shows health status
 - `docker inspect` has extra details (including health check command output)
- With `docker service`:
 - unhealthy tasks are terminated (i.e. the service is restarted)
 - failed deployments can be rolled back automatically
(by setting *at least* the flag `--update-failure-action rollback`)

Enabling health checks and auto-rollbacks

Here is a comprehensive example using the CLI:

```
docker service update \
--update-delay 5s \
--update-failure-action rollback \
--update-max-failure-ratio .25 \
--update-monitor 5s \
--update-parallelism 1 \
--rollback-delay 5s \
--rollback-failure-action pause \
--rollback-max-failure-ratio .5 \
--rollback-monitor 5s \
--rollback-parallelism 0 \
--health-cmd "curl -f http://localhost/ || exit 1" \
--health-interval 2s \
--health-retries 1 \
--image yourimage:newversion yourservice
```

Implementing auto-rollback in practice

We will use the following Compose file (`stacks/dockercoins+healthcheck.yml`):

```
...
hasher:
  build: dockercoins/hasher
  image: ${REGISTRY-127.0.0.1:5000}/hasher:${TAG-latest}
  healthcheck:
    test: curl -f http://localhost/ || exit 1
deploy:
  replicas: 7
  update_config:
    delay: 5s
    failure_action: rollback
    max_failure_ratio: .5
    monitor: 5s
    parallelism: 1
...
...
```

Enabling auto-rollback in dockercoins

We need to update our services with a healthcheck.



- Go to the `stacks` directory:

```
cd ~/container.training/stacks
```

- Deploy the updated stack with healthchecks built-in:

```
docker stack deploy --compose-file dockercoins+healthcheck.yml dockercoins
```

Visualizing an automated rollback

- Here's a good example of why healthchecks are necessary
- This breaking change will prevent the app from listening on the correct port
- The container still runs fine, it just won't accept connections on port 80



- Change the HTTP listening port:

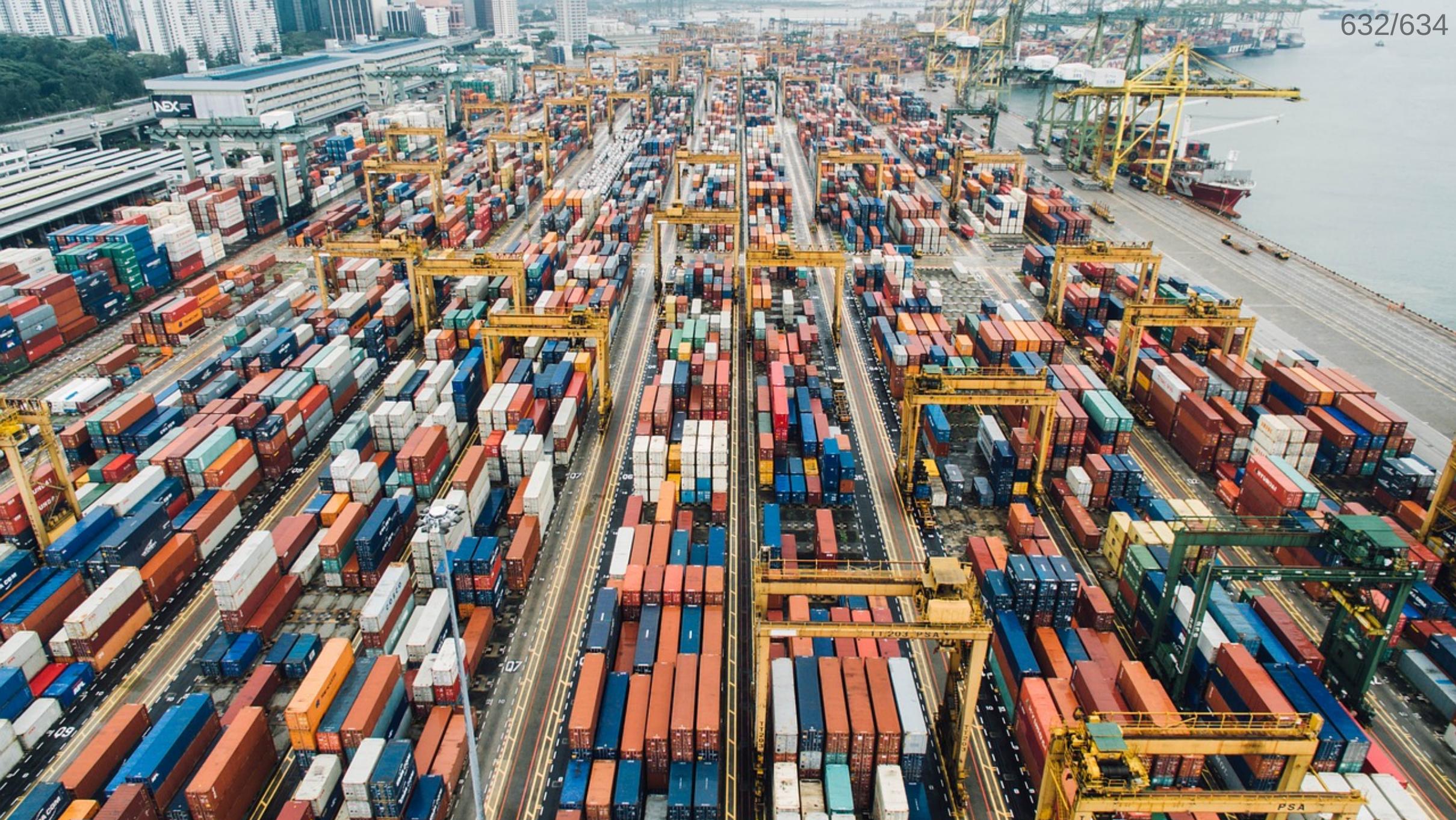
```
sed -i "s/80/81/" dockercoins/hasher/hasher.rb
```

- Build, ship, and run the new image:

```
export TAG=v0.3
docker-compose -f dockercoins+healthcheck.yml build
docker-compose -f dockercoins+healthcheck.yml push
docker service update --image=127.0.0.1:5000/hasher:$TAG dockercoins_hasher
```

CLI flags for health checks and rollbacks

--health-cmd string	Command to run to check health
--health-interval duration	Time between running the check (ms s m h)
--health-retries int	Consecutive failures needed to report unhealthy
--health-start-period duration	Start period for the container to initialize before counting retries towards unstable (ms s m h)
--health-timeout duration	Maximum time to allow one check to run (ms s m h)
--no-healthcheck	Disable any container-specified HEALTHCHECK
--restart-condition string	Restart when condition is met ("none" "on-failure" "any")
--restart-delay duration	Delay between restart attempts (ns us ms s m h)
--restart-max-attempts uint	Maximum number of restarts before giving up
--restart-window duration	Window used to evaluate the restart policy (ns us ms s m h)
--rollback	Rollback to previous specification
--rollback-delay duration	Delay between task rollbacks (ns us ms s m h)
--rollback-failure-action string	Action on rollback failure ("pause" "continue")
--rollback-max-failure-ratio float	Failure rate to tolerate during a rollback
--rollback-monitor duration	Duration after each task rollback to monitor for failure (ns us ms s m h)
--rollback-order string	Rollback order ("start-first" "stop-first")
--rollback-parallelism uint	Maximum number of tasks rolled back simultaneously (0 to roll back all at once)
--update-delay duration	Delay between updates (ns us ms s m h)
--update-failure-action string	Action on update failure ("pause" "continue" "rollback")
--update-max-failure-ratio float	Failure rate to tolerate during an update
--update-monitor duration	Duration after each task update to monitor for failure (ns us ms s m h)
--update-order string	Update order ("start-first" "stop-first")
--update-parallelism uint	Maximum number of tasks updated simultaneously (0 to update all at once)



Secrets management and encryption at rest

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

Secrets management and encryption at rest

(New in Docker Engine 1.13)

- Secrets management = selectively and securely bring secrets to services
- Encryption at rest = protect against storage theft or prying
- Remember:
 - control plane is authenticated through mutual TLS, certs rotated every 90 days
 - control plane is encrypted with AES-GCM, keys rotated every 12 hours
 - data plane is not encrypted by default (for performance reasons),
but we saw earlier how to enable that with a single flag