Introduction to Containers

Self-paced version

These slides are open source

- You are welcome to use, re-use, share these slides
- These slides are written in markdown
- The sources of these slides are available in a public GitHub repository:

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

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

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

Extra details

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


13/373
SANTA CRUZ
MONROVIA
HAMBURG SÜD

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!

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!

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/

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 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/

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.

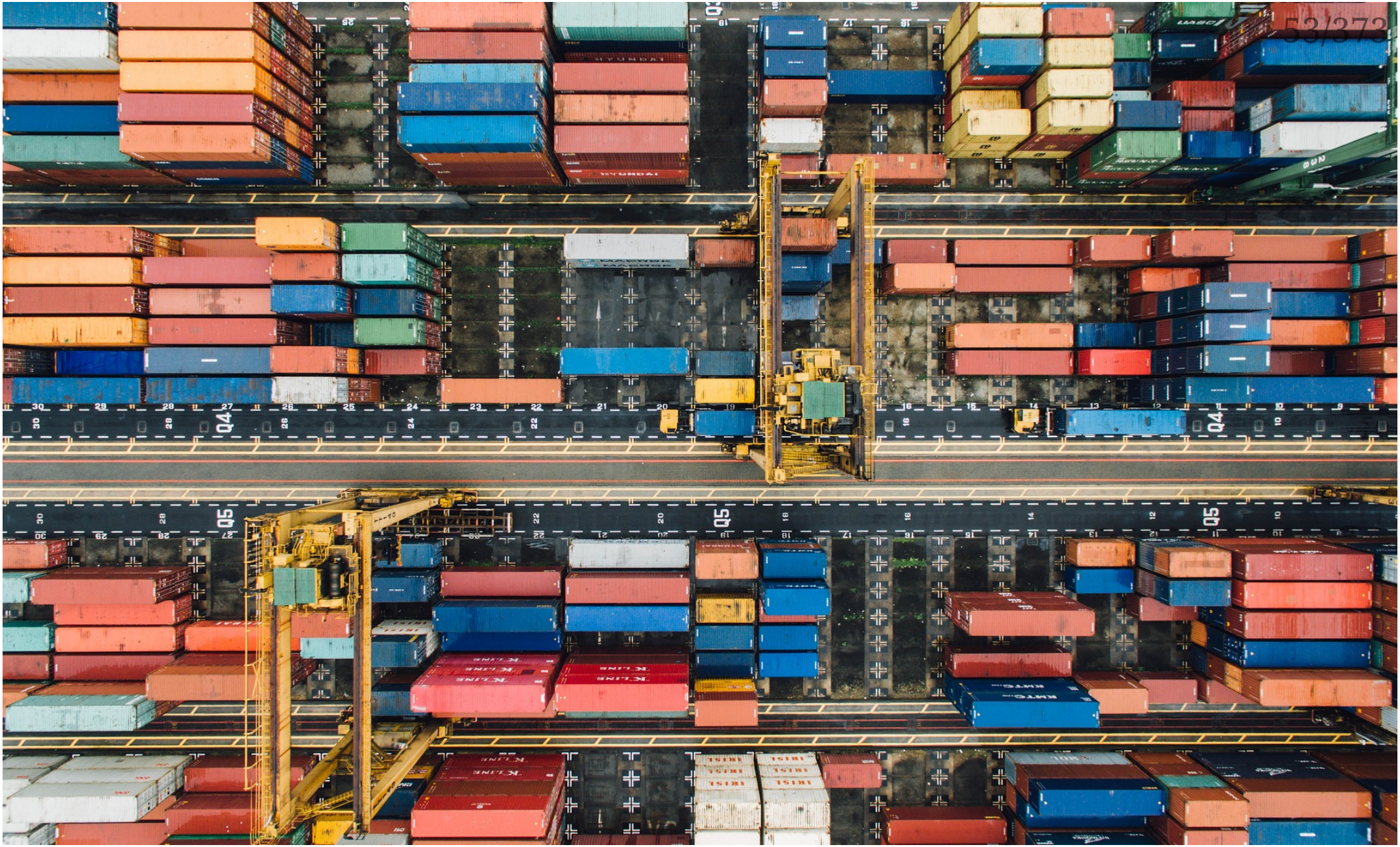
This will also allow to use remote Engines exactly as if they were local.

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.:


53/373

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 cannot run on Linux machines; at least not yet.)

Where's our container?

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

Starting another container

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

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

- We started a *brand new container*.
- The basic Ubuntu image was used, and `figlet` is not here.

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

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

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.

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.

Background containers

Objectives

Our first containers were *interactive*.

We will now see how to:

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

A non-interactive container

We will run a small custom container.

This container just displays the time every second.

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

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

Run a container in the background

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

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

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

List running containers

How can we check that our container is still running?

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

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

Docker tells us:

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

Starting more containers

Let's start two more containers.

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

```
$ docker run -d jpetazzo/clock
068cc994ffd0190bbe025ba74e4c0771a5d8f14734af772ddee8dc1aaf20567d
```

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

Viewing only the last container started

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

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

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

View only the IDs of the containers

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

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

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

Combining flags

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

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

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

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

(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. ag
57ad9bdfc06b  jpetazzo/clock  ...  21 min. ago  Exited (137) 3 min. ag
47d677dcfba4  jpetazzo/clock  ...  23 min. ago  Exited (137) 3 min. ag
5c1dfd4d81f1  jpetazzo/clock  ...  40 min. ago  Exited (0) 40 min. ago
b13c164401fb  ubuntu          ...  55 min. ago  Exited (130) 53 min. a
```

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.

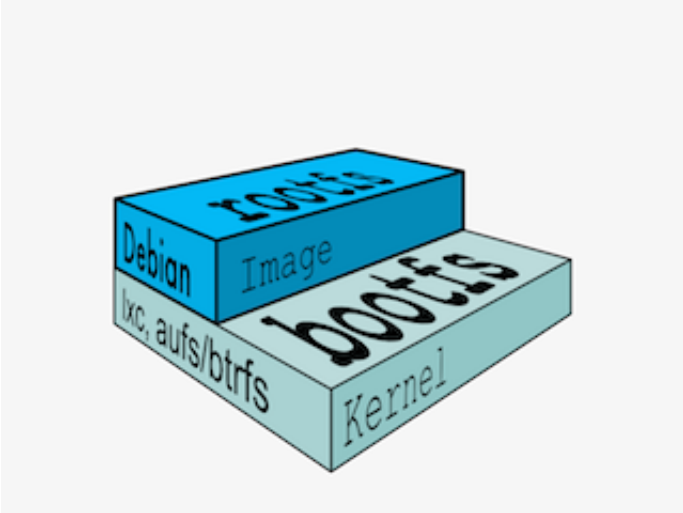
Attaching to a REPL

- 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.

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?

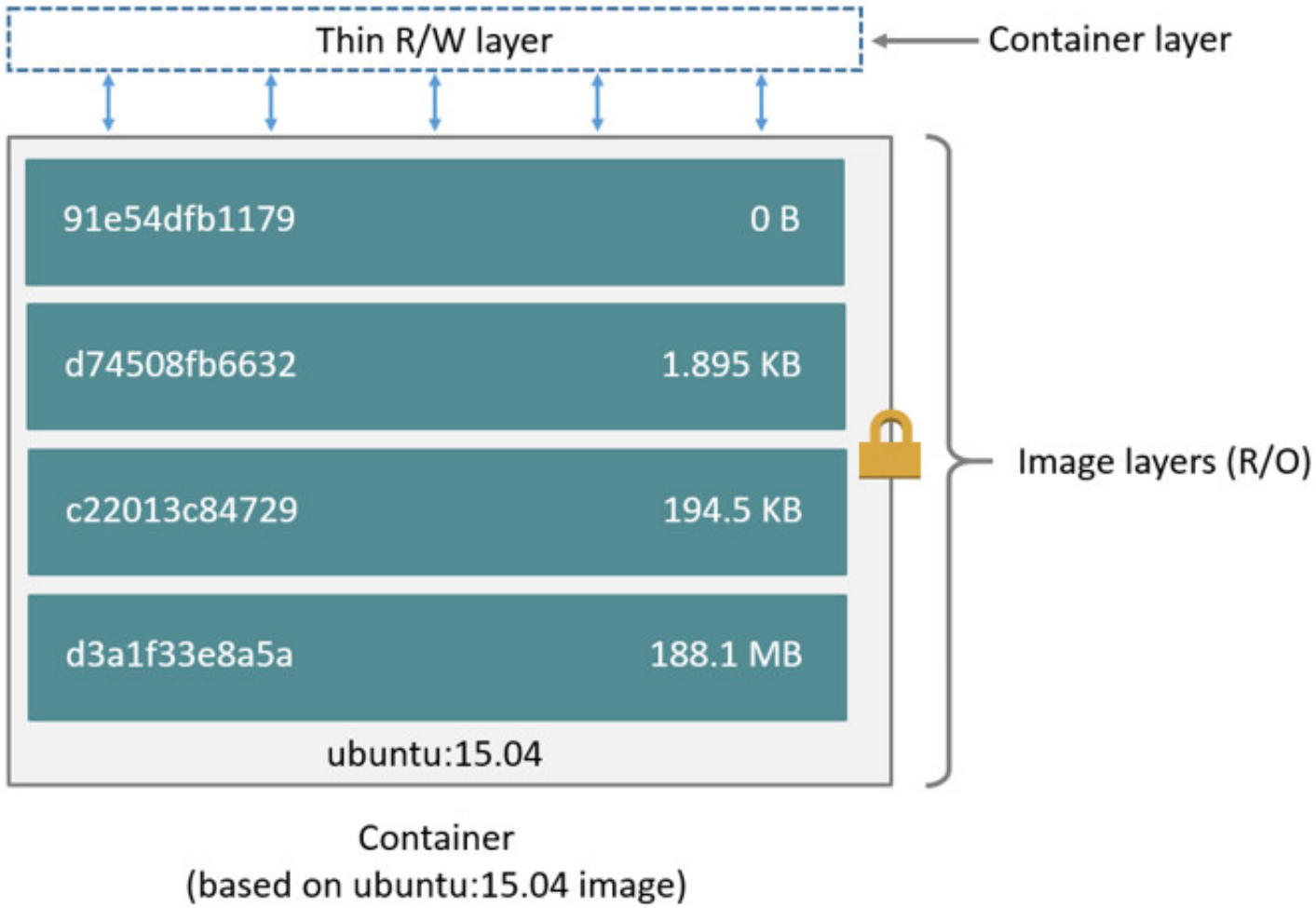
- Image = files + metadata
- These files form the root filesystem of our container.
- The metadata can indicate a number of things, e.g.:
  - the author of the image
  - the command to execute in the container when starting it
  - environment variables to be set
  - etc.
- Images are made of *layers*, 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

Example for a Java webapp

Each of the following items will correspond to one layer:

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

The read-write layer

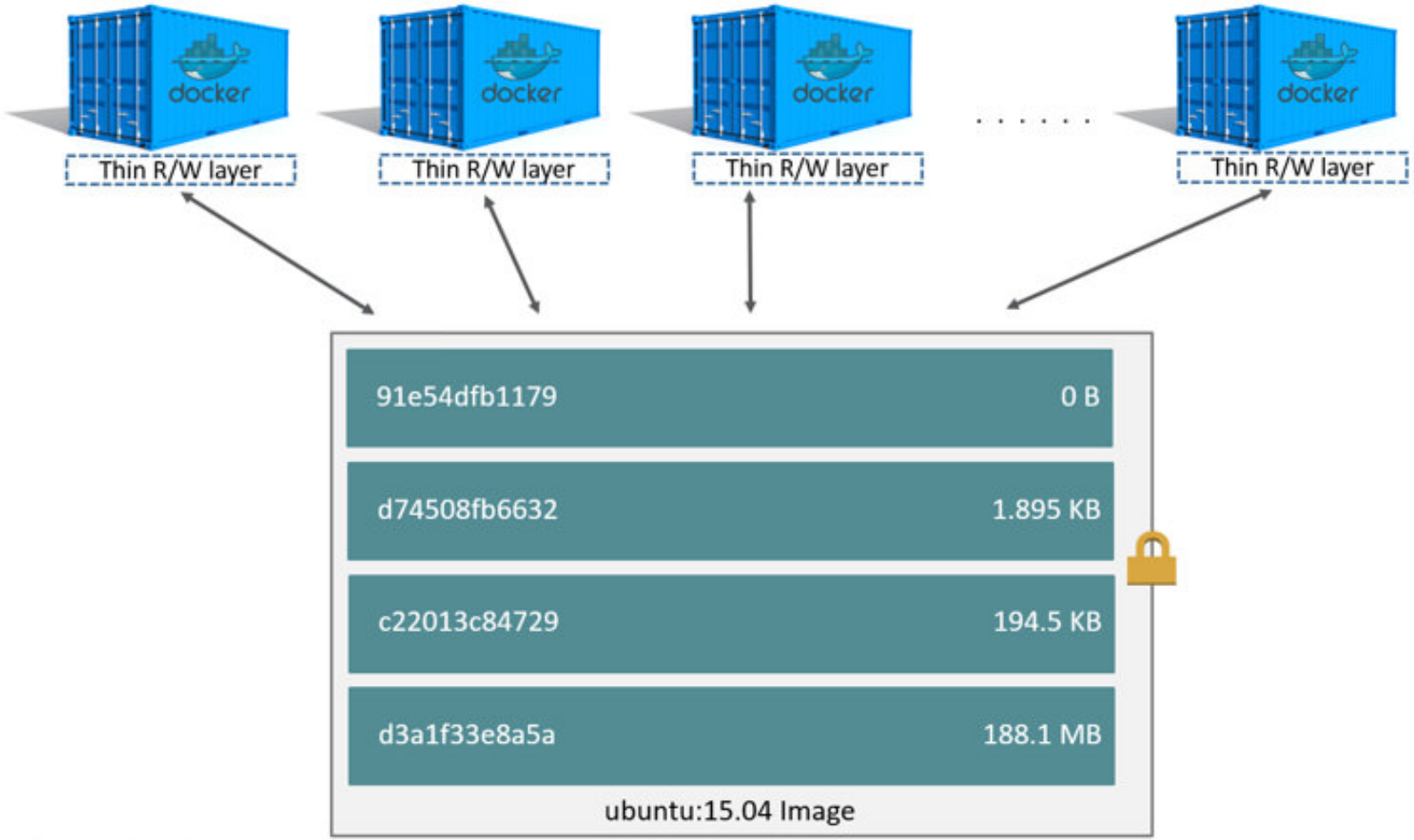


Differences between containers and images

- 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:

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  OFFICI
mesosphere/marathon      A cluster-wide init and co...   105
mesoscloud/marathon      Marathon                        31
mesosphere/marathon-lb   Script to update haproxy b...   22
tobilg/mongodb-marathon  A Docker image to start a ...   4
```

- "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.

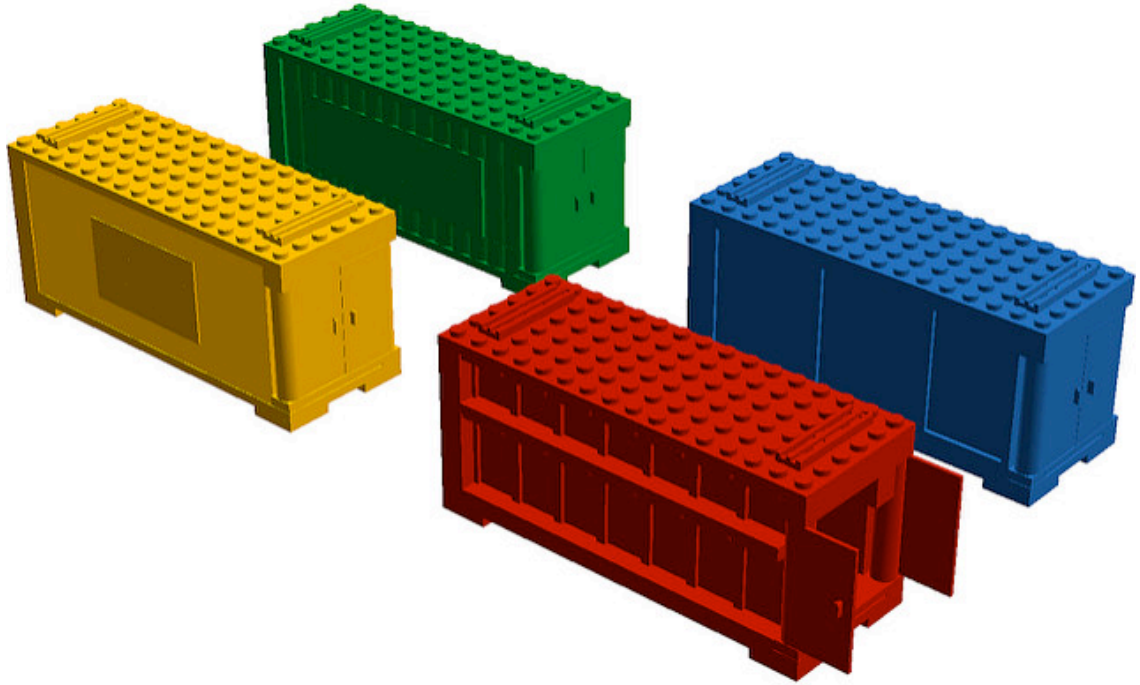
This is similar to what we would do with `pip install`, `npm install`, etc.

Section summary

We've learned how to:

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

Building Docker images with a Dockerfile



Objectives

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

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

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

`Dockerfile` overview

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

Writing our first `Dockerfile`

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

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

```
$ mkdir myimage
```

1. Create a `Dockerfile` inside this directory.

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

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

Type this into our Dockerfile...

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

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

Build it!

Save our file, then execute:

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

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

We will talk more about the build context later.

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

What happens when we build the image?

The output of `docker build` looks like this:

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

- The output of the `RUN` commands has been omitted.

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.

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

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
7257c37726a1   About an hour ago   /bin/sh -c apt-get update       21.58
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
<missing>      4 days ago          /bin/sh -c echo '#!/bin/sh'     194.5
<missing>      4 days ago          /bin/sh -c #(nop) ADD file:b    187.8
```

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.)

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
7257c37726a1   About an hour ago    /bin/sh -c apt-get update                       21.58
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
<missing>      4 days ago           /bin/sh -c echo '#!/bin/sh'                     194.5
<missing>      4 days ago           /bin/sh -c #(nop) ADD file:b                    187.8
```

- 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

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 `ENTRYPOINT`

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

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:/#
```

Copying files during the build

Objectives

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

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

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

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

Build some C code

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

Here is the program, `hello.c`:

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

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

Then we will write the Dockerfile.

The Dockerfile

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

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

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

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

Create this Dockerfile.

Testing our C program

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

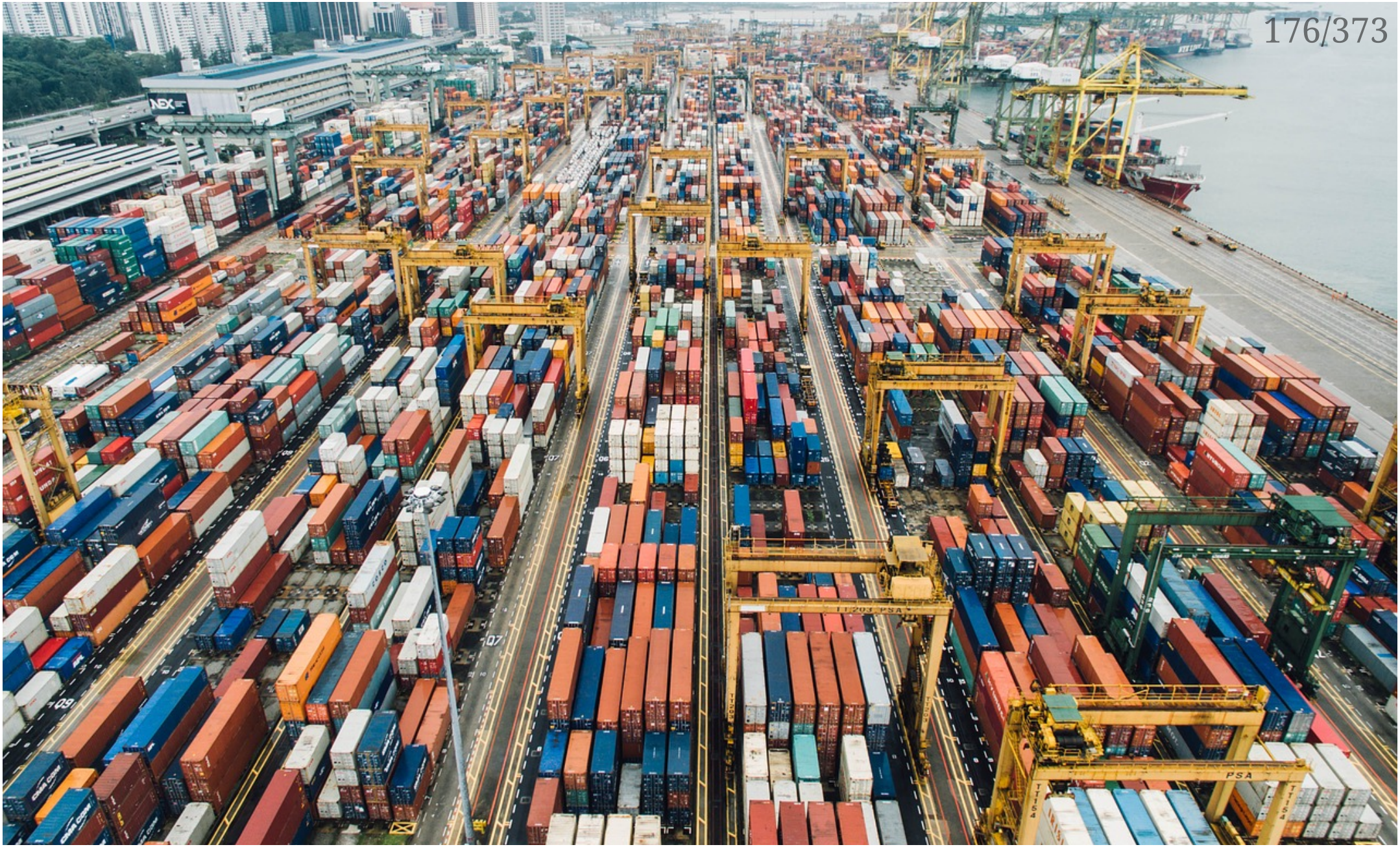
Success!

`COPY` and the build cache

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

Details

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


176/373
PSA

Exercise — writing Dockerfiles

Let's write Dockerfiles for an existing application!

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

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!

Getting a shell in a running container

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

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

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

Caveats

- The tool that you want to run needs to exist in the container.
- Some tools (like `ip netns exec`) let you attach to *one* namespace at a time.

  (This lets you e.g. setup network interfaces, even if you don't have `ifconfig` or `ip` in the container.)
- Most importantly: the container needs to be running.
- What if the container is stopped or crashed?

Getting a shell in a stopped container

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

Analyzing a stopped container

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

```
docker run jpetazzo/crashtest
```

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

What would MacGyver™ do?

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

```
docker ps -l
```

Viewing filesystem changes

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

```
docker diff <container_id>
```

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

Accessing files

- We can extract files with `docker cp`.

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

- Then we can look at that log file.

```
cat error.log
```

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

Exploring a crashed container

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

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

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

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

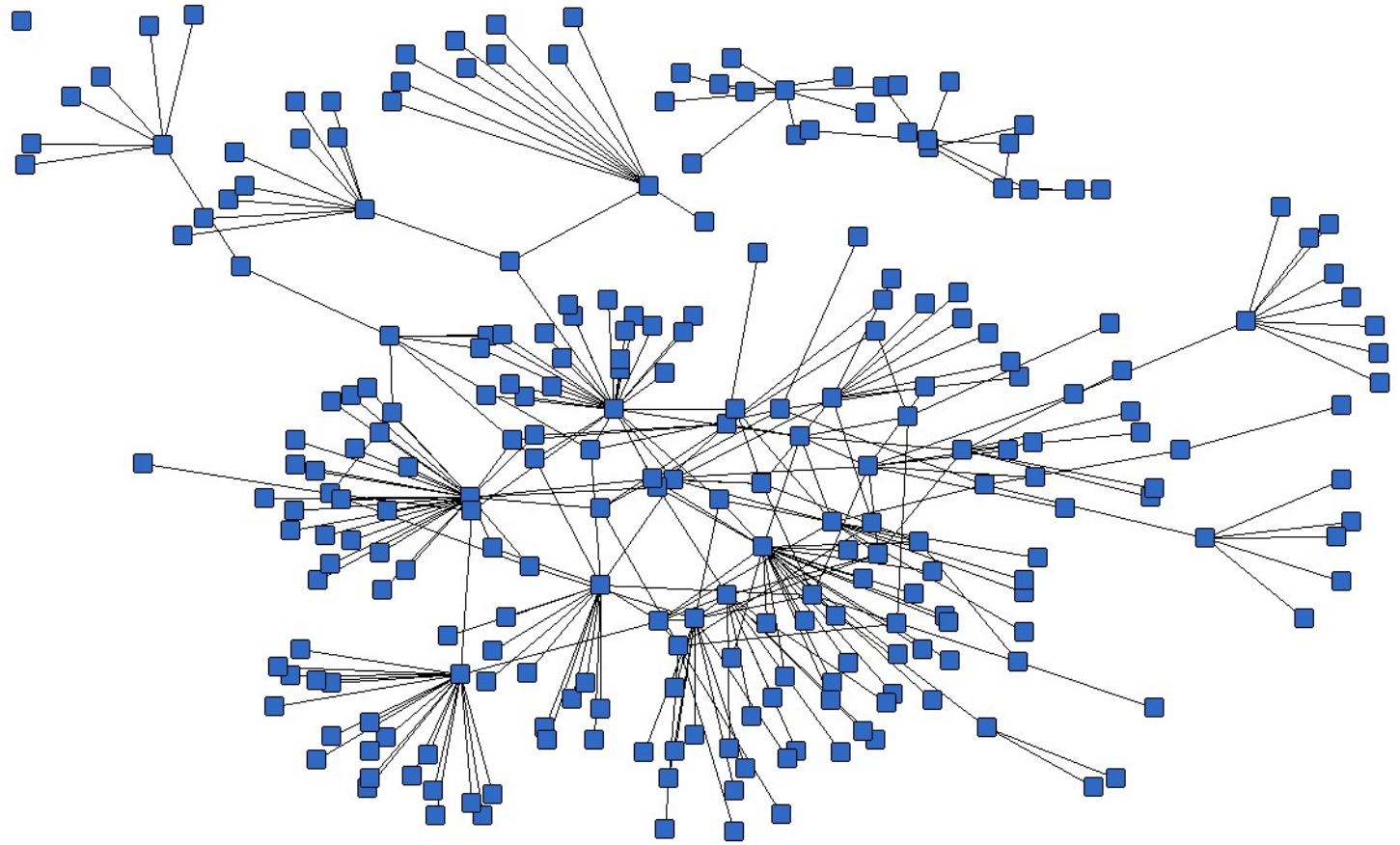
Obtaining a complete dump

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

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

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

Container networking basics



Objectives

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

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

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

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

A simple, static web server

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

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

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

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

Finding our web server port

We will use `docker ps`:

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

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

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

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

Connecting to our web server (GUI)

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

Connecting to our web server (CLI)

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

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

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

How does Docker know which port to map?

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

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

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

```
$ docker history nginx
```

Why are we mapping ports?

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

Finding the web server port in a script

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

There is a command to help us:

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

Manual allocation of port numbers

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

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

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

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

Plumbing containers into your infrastructure

There are many ways to integrate containers in your network.

- Start the container, letting Docker allocate a public port for it.
  Then retrieve that port number and feed it to your configuration.

- Pick a fixed port number in advance, when you generate your configuration.
  Then start your container by setting the port numbers manually.

- Use a network plugin, connecting your containers with e.g. VLANs, tunnels...

- Enable *Swarm Mode* to deploy across a cluster.
  The container will then be reachable through any node of the cluster.

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 }}' <yourCont
172.17.0.3
```

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

Pinging our container

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

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

Section summary

We've learned how to:

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

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

Working with volumes

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 neither.
- If a container is started with the `--read-only` flag, the volume will still be writable (unless the volume is a read-only volume).

Compose for development stacks

Previous section | Back to table of contents | Next section

Compose for development stacks

Dockerfiles are great to build container images.

But what if we work with a complex stack made of multiple containers?

Eventually, we will want to write some custom scripts and automation to build, run, and connect our containers together.

There is a better way: using Docker Compose.

In this section, you will use Compose to bootstrap a development environment.

What is Docker Compose?

Docker Compose (formerly known as `fig`) is an external tool.

Unlike the Docker Engine, it is written in Python. It's open source as well.

The general idea of Compose is to enable a very simple, powerful onboarding workflow:

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

Compose overview

This is how you work with Compose:

- You describe a set (or stack) of containers in a YAML file called `docker-compose.yml`.
- You run `docker-compose up`.
- Compose automatically pulls images, builds containers, and starts them.
- Compose can set up links, volumes, and other Docker options for you.
- Compose can run the containers in the background, or in the foreground.
- When containers are running in the foreground, their aggregated


$ do

Checking if Compose is installed

If you are using the official training virtual machines, Compose has been pre-installed.

If you are using Docker for Mac/Windows or the Docker Toolbox, Compose comes with them.

If you are on Linux (desktop or server environment), you will need to install Compose from its release page or with `pip install docker-compose`.

You can always check that it is installed by running:

```
$ docker-compose --version
```

Launching Our First Stack with Compose

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

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

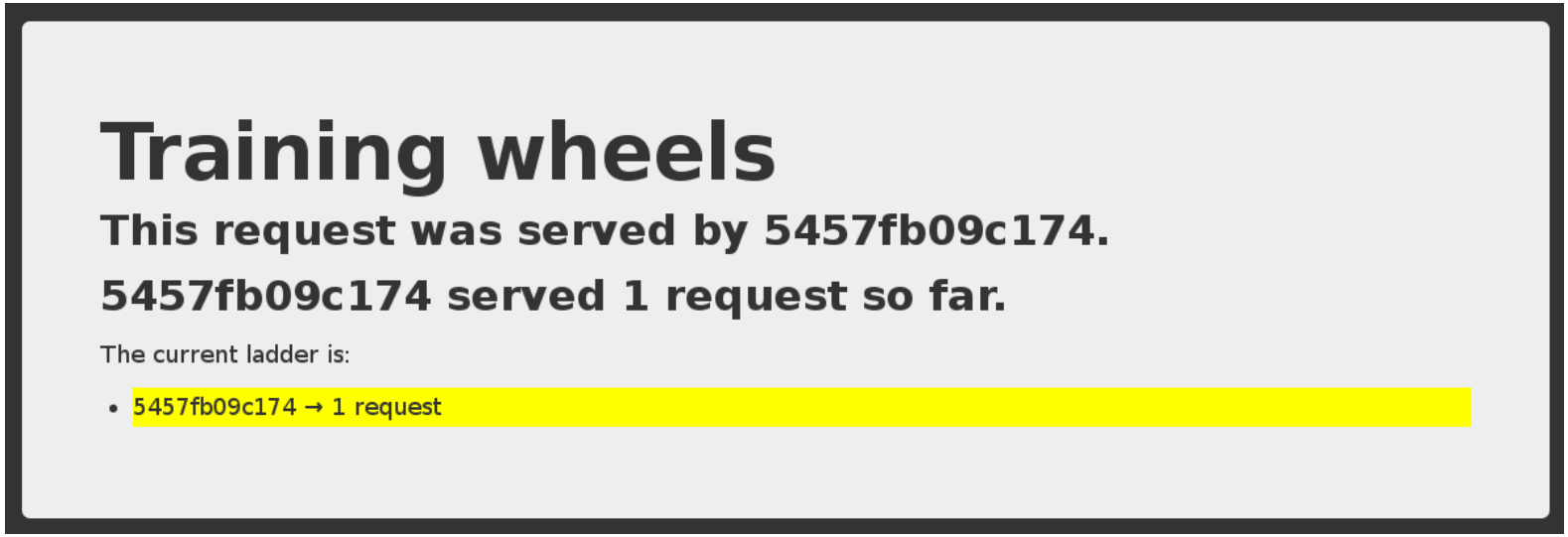
Second step: start your app.

```
$ docker-compose up
```

Watch Compose build and run your app with the correct parameters, including linking the relevant containers together.

Launching Our First Stack with Compose

Verify that the app is running at `http://<yourHostIP>:8000`.



Stopping the app

When you hit `^C`, Compose tries to gracefully terminate all of the containers.

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

The `docker-compose.yml` file

Here is the file used in the demo:

```
version: "2"

services:
  www:
    build: www
    ports:
      - 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. (We should use `"2"` or later; version 1 is deprecated.)
- `services` is mandatory. A service is one or more replicas of the same image running as containers.
- `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).

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/

Compose commands

We already saw `docker-compose up`, but another one is `docker-compose build`.

It will execute `docker build` for all containers mentioning a `build` path.

It can also be invoked automatically when starting the application:

```
docker-compose up --build
```

Another common option is to start containers in the background:

```
docker-compose up -d
```

Check container status

It can be tedious to check the status of your containers with `docker ps`, especially when running multiple apps at the same time.

Compose makes it easier; with `docker-compose ps` you will see only the status of the containers of the current stack:

```
$ docker-compose ps
Name                        Command             State           Ports
-------------------------------------------------------------------------
trainingwheels_redis_1   /entrypoint.sh red   Up      6379/tcp
trainingwheels_www_1     python counter.py    Up      0.0.0.0:8000->50
```

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

Compose is smart. If your container uses volumes, when you restart your application, Compose will create a new container, but carefully re-use the volumes it was using previously.

This makes it easy to upgrade a stateful service, by pulling its new image and just restarting your stack with Compose.

Compose project name

- When you run a Compose command, Compose infers the "project name" of your app.
- By default, the "project name" is the name of the current directory.
- For instance, if you are in `/home/zelda/src/ocarina`, the project name is `ocarina`.
- All resources created by Compose are tagged with this project name.
- The project name also appears as a prefix of the names of the resources.

  E.g. in the previous example, service `www` will create a container `ocarina_www_1`.

Running two copies of the same app

If you want to run two copies of the same app simultaneously, all you have to do is to make sure that each copy has a different project name.

You can:

- copy your code in a directory with a different name
- start each copy with `docker-compose -p myprojname up`

Each copy will run in a different network, totally isolated from the other.

This is ideal to debug regressions, do side-by-side comparisons, etc.


341/373

Exercise — writing a Compose file

Previous section | Back to table of contents | Next section

Exercise — writing a Compose file

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

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


344/373
APL
APL
APL
APL

Thank you!


EN

Links and resources

Previous section | Back to table of contents | Next section

Links and resources

- Docker Community Slack
- Docker Community Forums
- Docker Hub
- Docker Blog
- Docker documentation
- Docker on StackOverflow
- Docker on Twitter
- Play With Docker Hands-On Labs

These slides (and future updates) are on → https://container.training/