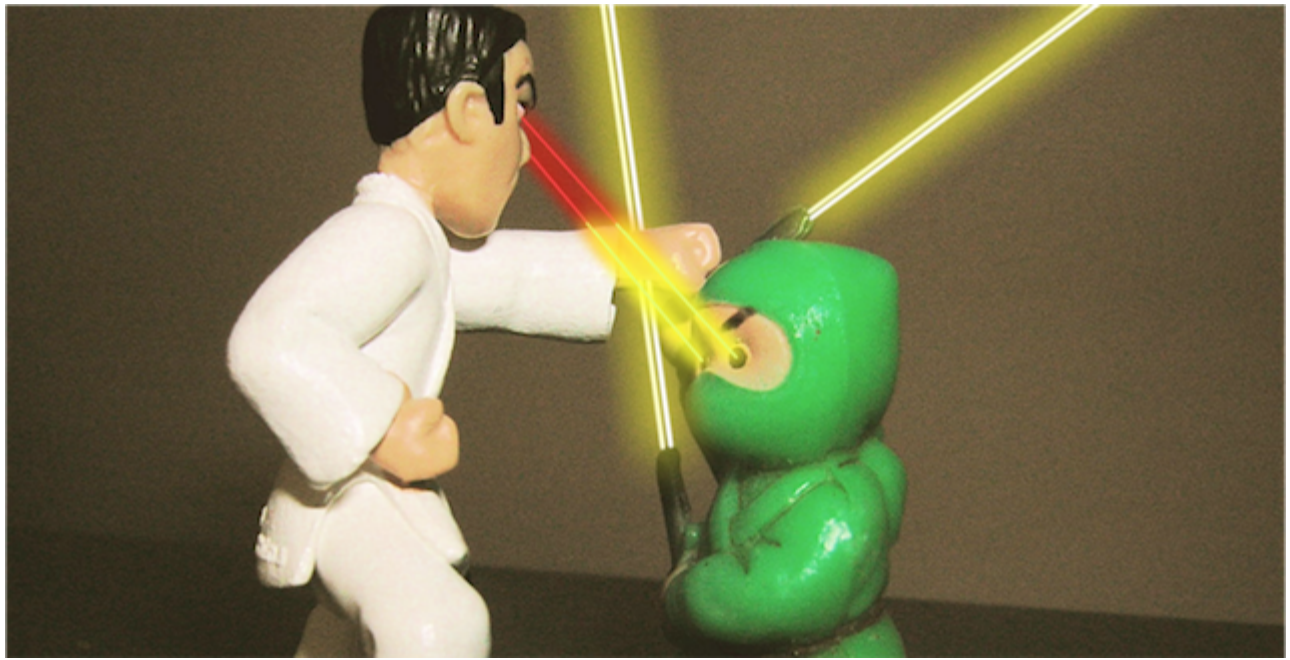

# 1-HOUR TRAINING MASTERING DOCKER

# About Docker



# Elevator pitch

(for your manager, boss…)

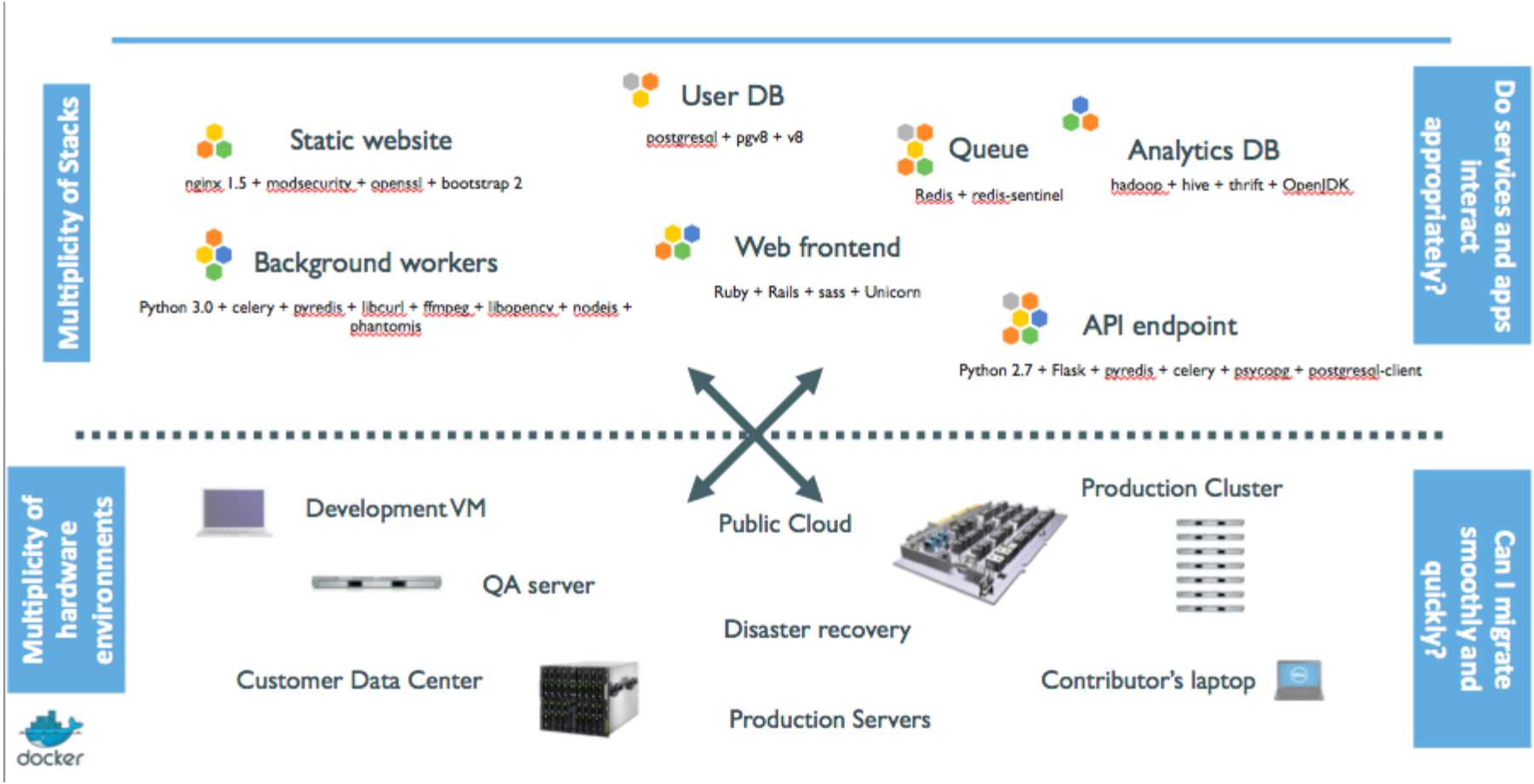
# OK… Why the buzz around containers?

- 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

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

docker

# An inspiration and some ancient history!

# Intermodal shipping containers

# This spawned a Shipping Container Ecosystem!

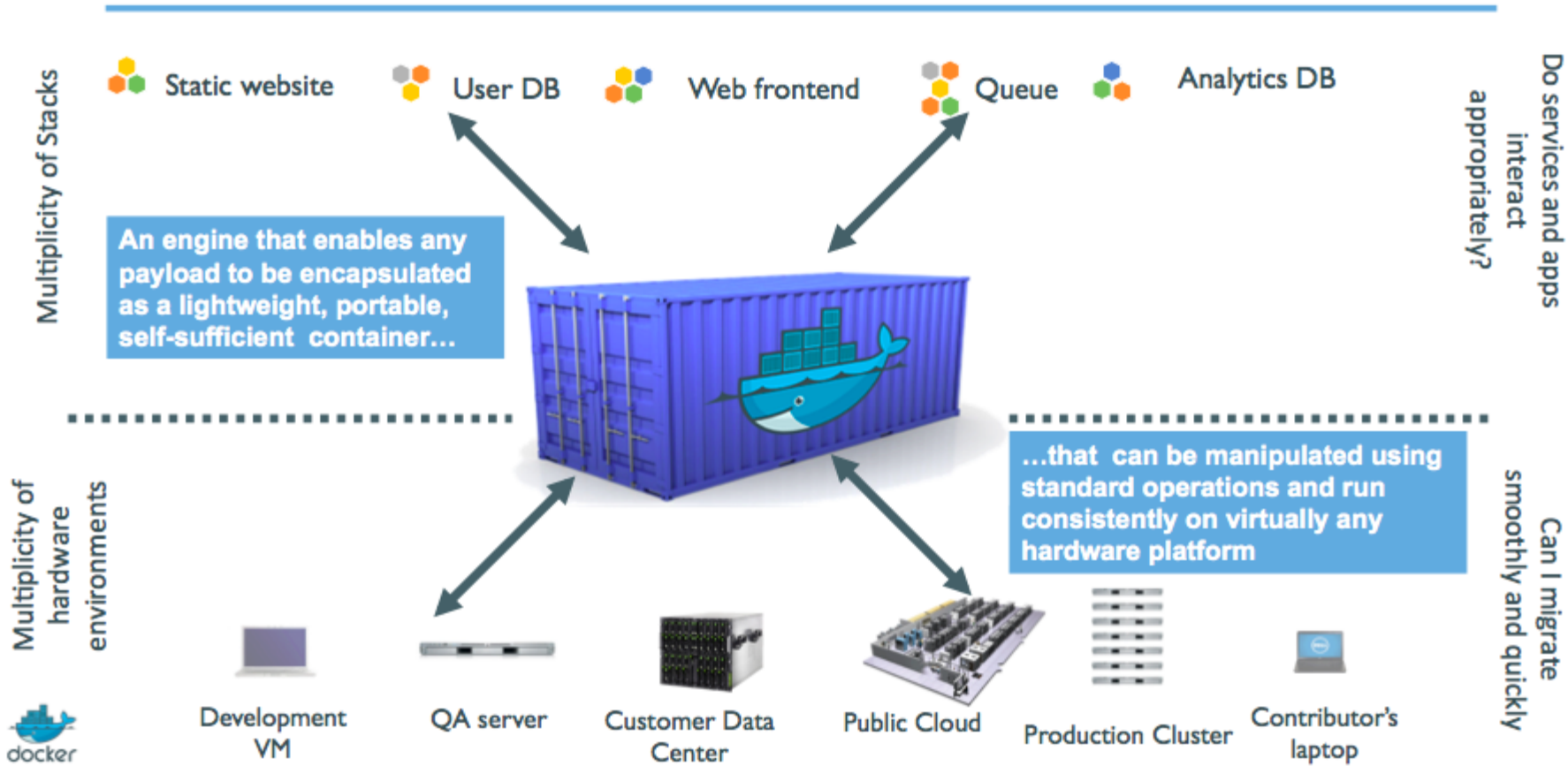
- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)

→ massive globalization

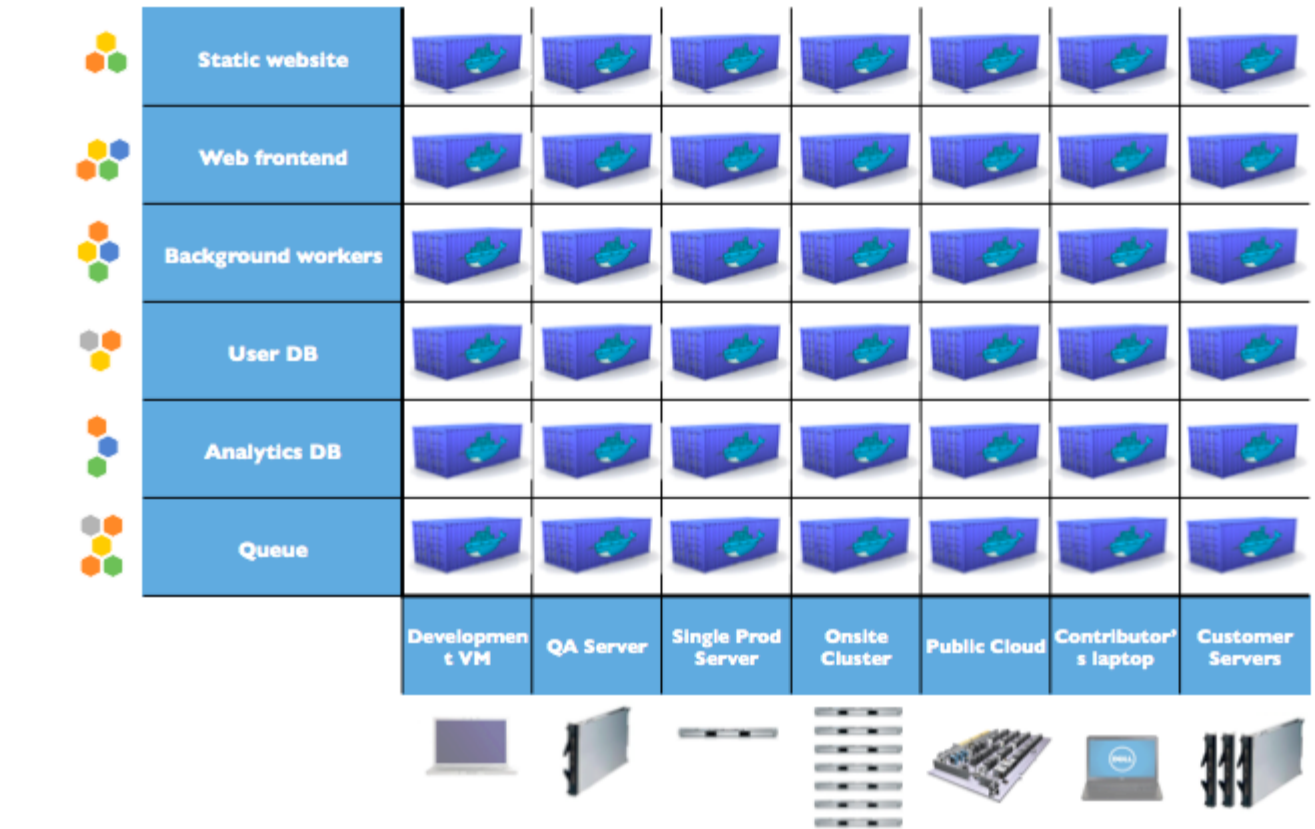
- 5000 ships deliver 200M containers per year

docker

# A shipping container system for applications



# Eliminate the matrix from Hell




docker

# Results

- Dev-to-prod reduced from 9 months to 15 minutes (ING)
- Continuous integration job time reduced by more than 60% (BBC)
- Dev-to-prod reduced from weeks to minutes (GILT)

# 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
```

Also works to create dev, 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!

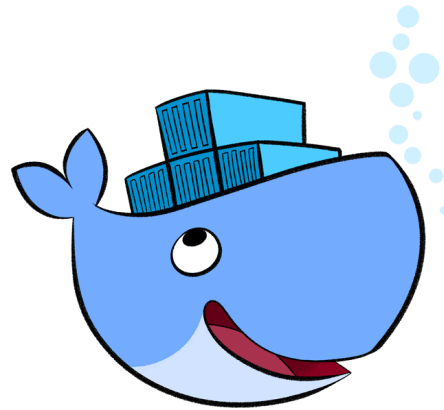
# 1.1 Environment Preparation

# 1.1 Environment Preparation

In this section we'll see how to prepare the environment with Docker

# Docker Playground

- Go to https://labs.play-with-docker.com/
- Sign in with your Docker credentials
- Sign up if you don't already have a user



# Docker Playground

Once started click on `+ Add new Instance` button

- To verify installation

```
$ docker --version
Docker version 20.10.17, build 100c701
```

- Run your first container

```
$ docker run hello-world
...
Hello from Docker!
...
```

# 1.2 Running Docker

# 1.2 Running Docker

At the end of this lesson you'll have learned

- The Docker architecture
- How to run Docker containers

# Docker Architecture

Docker is a client-server application.

- **The Docker Engine** (or "daemon")

  Receives and processes incoming Docker API requests.

- **The Docker client**

  Talks to the Docker daemon via the Docker API.

  We'll use mostly the CLI embedded within the docker binary.

- **Docker Hub Registry**

  Collection of public images.

  The Docker daemon talks to it via the registry API.

# Hello World

- In your Docker environment, just run the following command:

```
$ docker run alpine echo hello world
hello world
```

Notes:

- If `alpine` image was not used before, it will be downloaded automatically
- During presentation we'll execute commands directly to see output on screen

# That was our first container!

- We used one of the smallest, simplest images available: `alpine`.
- `alpine` 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@4f9263507465:/#
```

- 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@4f9263507465:/# 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@4f9263507465:/# apt-get update
...
Fetched 1514 kB in 14s (103 kB/s)
Reading package lists... Done
root@4f9263507465:/# apt-get install figlet
Reading package lists... Done
...
```

- One minute later, `figlet` is installed!

```
# figlet hello
 _          _ _
| |__   ___| | | ___
| '_ \ / _ \ | |/ _ \
| | | |  __/ | | (_) |
|_| |_|\___|_|_|\___/
```

# Exiting our container

- Just exit the shell, like you would usually do (E.g. with `^D` or `exit`)

```
root@4f9263507465:/# exit
```

- Our container is now in a *stopped* state.
- It still exists on disk, but all compute resources have been freed up.

# Starting another container

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

```
$ docker run -it ubuntu
root@31b883f8d6c8:/# figlet
bash: figlet: command not found
```

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

# 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.

```
$ docker ps
CONTAINER ID  IMAGE           ...  CREATED        STATUS        ...
47d677dcfba4  jpetazzo/clock  ...  2 minutes ago  Up 2 minutes  ...
57ad9bdfc06b  jpetazzo/clock  ...  8 seconds ago  Up 6 seconds  ...
068cc994ffd0  jpetazzo/clock  ...  8 seconds ago  Up 6 seconds  ...
```

# View the logs of a container

- We told you that Docker was logging the container output. Let's see that now.

```
$ docker logs -f $(docker ps -lq)
Sat Feb 25 14:55:46 UTC 2022
Sat Feb 25 14:55:47 UTC 2022
...
```

- 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.)
- with flag `-f`, we can follow the logs of our container.
- Use `^C` to exit.

# Stop our container

There are two ways we can terminate our detached container.

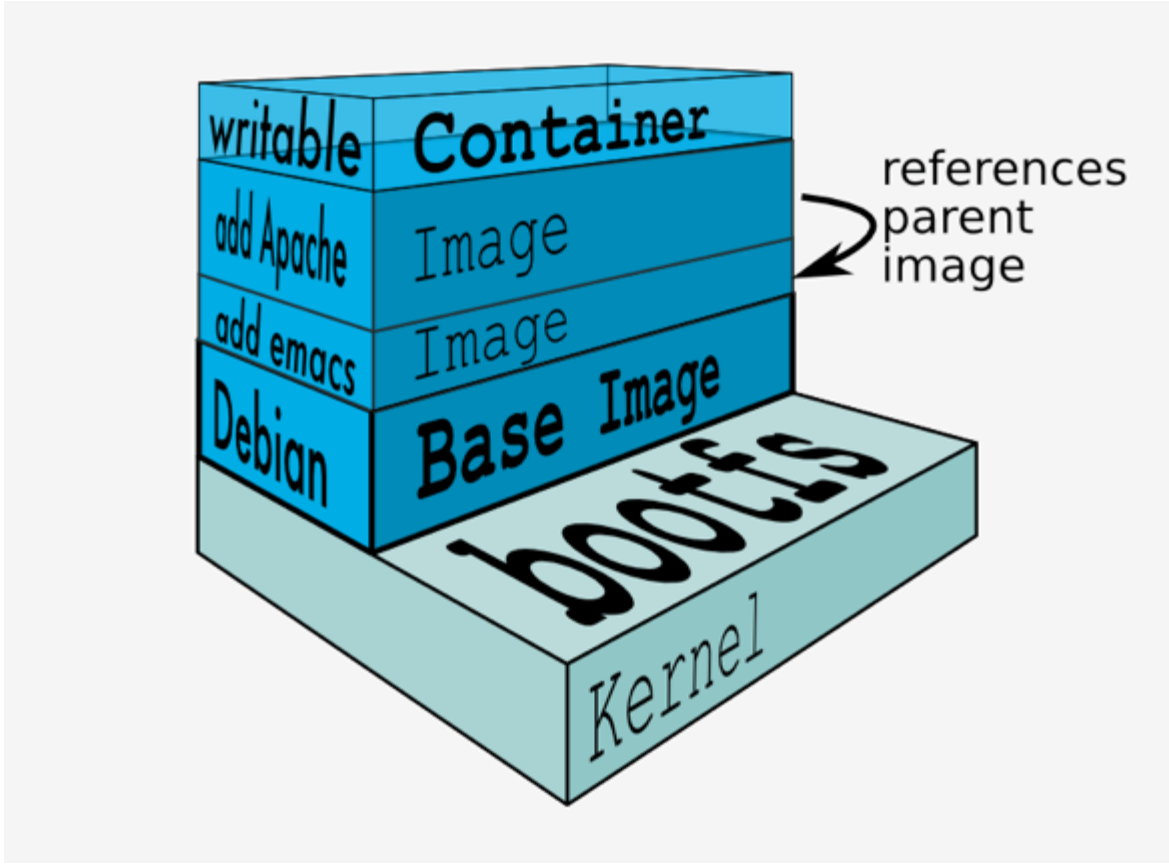
- Killing it using the `` `docker kill` `` command.
- Stopping it using the `` `docker stop` `` command.

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.

# Section 1.3 - Docker Images



# Section 1.3 - Docker Images

In this lesson, we will explain:

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

# What is an Image?

- An image is a collection of files + some meta data.
  (Technically: those files form the root filesystem of a container.)
- Images are made of *layers*, conceptually stacked on top of each other.
- Each layer can add, change, and remove files.
- Images can share layers to optimize disk usage, transfer times, and memory use.
- Example:
  - CentOS
  - JRE
  - Tomcat
  - Dependencies
  - Application JAR
  - Configuration

# Image vs Container

- 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.

Let's give a couple of metaphors to illustrate those concepts.

# Image as stencils

Images are like templates or stencils that you can create containers from.

# 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.

# 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`

- 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.

# 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 server 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
```

# 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.

# 2.1. Build Docker Images with 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 -p training/figlet/
```

2. Create a `Dockerfile` inside this directory.

```
$ cd training/figlet/
$ vim Dockerfile
```

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

# Type this into our Dockerfile…

- Write the following:

```
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; but to keep things simple: this is the directory where our Dockerfile is located.)

# What happens when we build the image?

- The output of `docker build` looks like this:

```
$ cd training/figlet/
$ docker build -t figlet .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 1/3 : FROM ubuntu
 ---> f49eec89601e
Step 2/3 : RUN apt-get update
 ---> Running in d5167bcc1c45
 ---> 3efa17a47e0c
Removing intermediate container d5167bcc1c45
Step 3/3 : RUN apt-get install figlet
 ---> Running in 442cd64b1a43
 ---> bbf166d0be0b
Removing intermediate container 442cd64b1a43
Successfully built bbf166d0be0b
```

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

- Let's explain what this output means.

# 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 ...` ``.

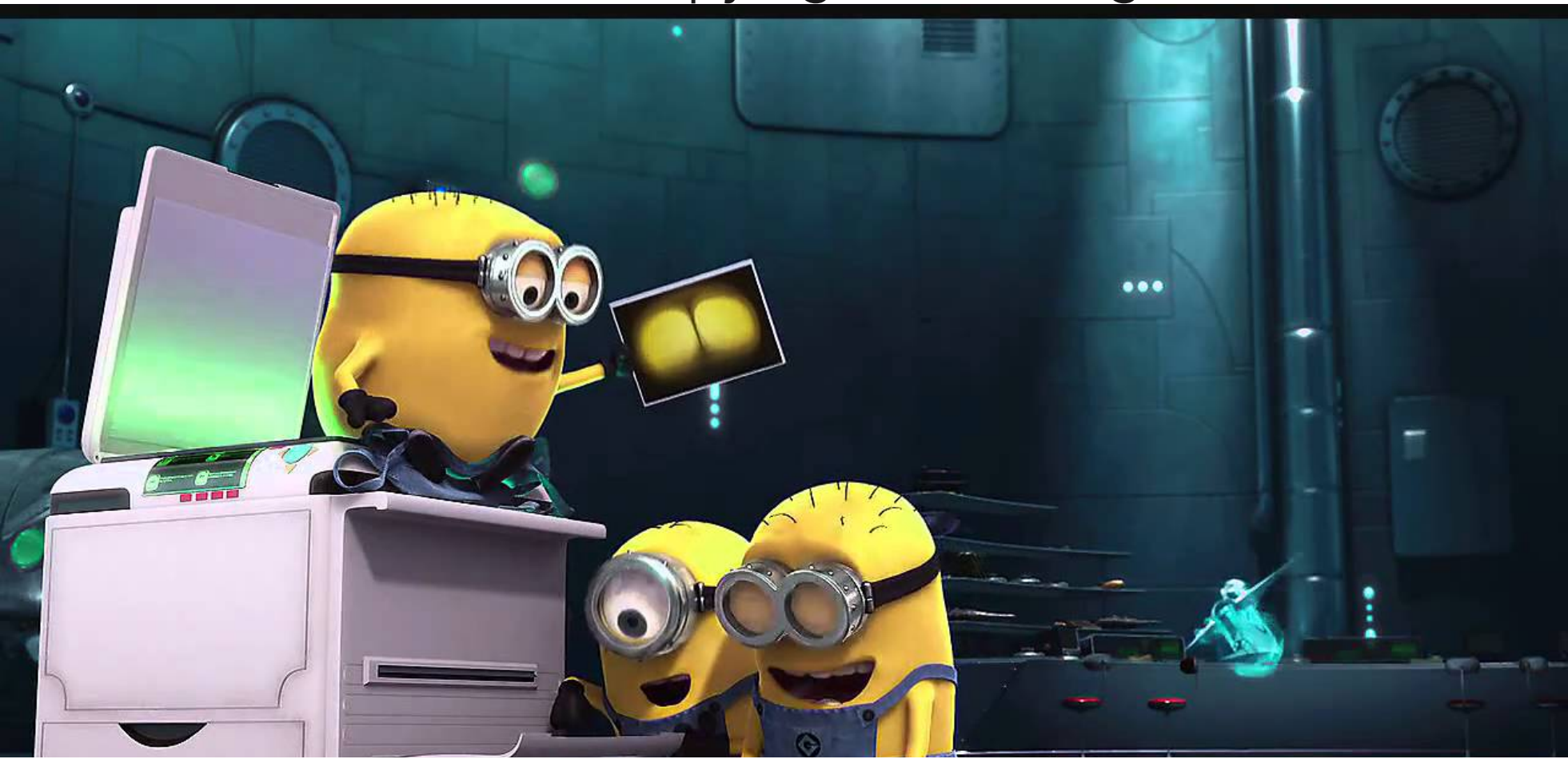
# Running the image

- Let's run it:

```
$ docker run -ti figlet
root@91f3c974c9a1:/# figlet hello
 _          _ _
| |__   ___| | | ___
| '_ \ / _ \ | |/ _ \
| | | |  __/ | | (_) |
|_| |_|\___|_|_|\___/
```

- Sweet is the taste of success!

# Section 2.4 - Copying files during build



# Section 2.4 - Copying files during build

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 direcotry.
- Run `docker build -t hello .` in this directory.

```
$ cd training/hello-c
$ docker build -t hello .
Successfully built f7ca4c22771f
```

- Run `docker run hello`, you should see `Hello, world!`.

```
$ docker run hello
Hello, world!
```

Success!

# 2.5. Laboratory

- Build a container with a Golang hello example

```
$ docker run hello-go
Hello world from Go!
```

- Here is a Hello world in Go (`hello.go`)

```
package main
import "fmt"
func main() {
    fmt.Printf("Hello world from Go!\n")
}
```

Remember

- You can use as parent image `golang:1.8.0-alpine`
- Your code needs to be on `/go/src/` directory
- To compile the code you need to run `go build SOURCE_CODE`
- Your final binary is located at `/go/hello`
- Build your image with name `hello-go`
- Run a container to print the output

# Section 3.2 - Networking Basics

# 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  ...  PORTS
e40ffb406c9e  ...  0.0.0.0:32769->80/tcp
```

- The web server is running on port 80 inside the container.
- The port is mapped to 32769 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:32769
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

# Build and Run Guestbook

We want to build and run a Guestbook App that uses Redis as backend.

- Clone this git repo:

```
git clone https://github.com/dkapanidis/training
```

## Steps

- Go to `docker/101/guestbook-node`.
- Run `docker compose up -d`.
- Run `docker compose ps` to get the port.
- Open port to see the guestbook running.