

Deployment

Your code runs on your laptop



Your code runs somewhere else
(probably on a server)

Deployment: what

Just copy your code!

Not enough...

- Your code

- ECMAScript dependencies

- Binary/FFI libraries (maybe)

- NodeJS (of a correct version!)

- NodeJS dependencies

- Maybe a database?

- Maybe cache, queue...

- Basic OS libraries

- Kernel

Good news: Linux kernel «must not break userspace»

Bad news: the only such component in your system

Deployment: what

Just copy your code!

Not enough...

- Your code

- ECMAScript dependencies

- Binary/FFI libraries (maybe)

- NodeJS (of a correct version!)

- NodeJS dependencies

- Maybe a database?

- Maybe cache, queue...

- Basic OS libraries

- Kernel

Good news: Linux kernel «must not break userspace»

Bad news: the only such component in your system

Deployment: what

Just copy your code!

Not enough...

Your code

ECMAScript dependencies

Binary/FFI libraries (maybe)

NodeJS (of a correct version!)

NodeJS dependencies

Maybe a database?

Maybe cache, queue...

Basic OS libraries

Kernel

Good news: Linux kernel «must not break userspace»

Bad news: the only such component in your system

Deployment: what

Just copy your code!

Not enough...

Your code

ECMAScript dependencies

Binary/FFI libraries (maybe)

NodeJS (of a correct version!)

NodeJS dependencies

Maybe a database?

Maybe cache, queue...

Basic OS libraries

Kernel

Good news: Linux kernel «must not break userspace»

Bad news: the only such component in your system

Deployment: what

Just copy your code!

Not enough...

- Your code

- ECMAScript dependencies

- Binary/FFI libraries (maybe)

- NodeJS (of a correct version!)

- NodeJS dependencies

- Maybe a database?

- Maybe cache, queue...

- Basic OS libraries

- Kernel

Good news: Linux kernel «must not break userspace»

Bad news: the only such component in your system

Deployment: what

Just copy your code!

Not enough...

- Your code

- ECMAScript dependencies

- Binary/FFI libraries (maybe)

- NodeJS (of a correct version!)

- NodeJS dependencies

- Maybe a database?

- Maybe cache, queue...

- Basic OS libraries

- Kernel

Good news: Linux kernel «must not break userspace»

Bad news: the only such component in your system

Deployment: what

Just copy your code!

Not enough...

- Your code

- ECMAScript dependencies

- Binary/FFI libraries (maybe)

- NodeJS (of a correct version!)

- NodeJS dependencies

- Maybe a database?

- Maybe cache, queue...

- Basic OS libraries

- Kernel

Good news: Linux kernel «must not break userspace»

Bad news: the only such component in your system

Deployment: more than once

All that long list?

Version conflicts everywhere!

Deployment: options

- Specific manual configuration

- Same, but not by you

 - Target a distribution's version set

- Every project has own copies of libraries

 - How NPM works...

 - Binary dependencies?

- Every version of **everything** into its own prefix

 - Nix, Guix, ...

- Every project has its own OS installation

 - Virtual machines, probably

- Like VMs but less heavy: kernel is shared.

 - Containers

 - Docker sits here

Deployment: options

- Specific manual configuration

- Same, but not by you

 - Target a distribution's version set

- Every project has own copies of libraries

 - How NPM works...

 - Binary dependencies?

- Every version of **everything** into its own prefix

 - Nix, Guix, ...

- Every project has its own OS installation

 - Virtual machines, probably

- Like VMs but less heavy: kernel is shared.

 - Containers

 - Docker sits here

Deployment: options

- Specific manual configuration

- Same, but not by you

 - Target a distribution's version set

- Every project has own copies of libraries

 - How NPM works...

 - Binary dependencies?

- Every version of **everything** into its own prefix

 - Nix, Guix, ...

- Every project has its own OS installation

 - Virtual machines, probably

- Like VMs but less heavy: kernel is shared.

 - Containers

 - Docker sits here

Deployment: options

- Specific manual configuration

- Same, but not by you

 - Target a distribution's version set

- Every project has own copies of libraries

 - How NPM works...

 - Binary dependencies?

- Every version of **everything** into its own prefix

 - Nix, Guix, ...

- Every project has its own OS installation

 - Virtual machines, probably

- Like VMs but less heavy: kernel is shared.

 - Containers

 - Docker sits here

Deployment: virtualisation

Bare metal

- Least overhead
- Least flexible management
- Well-suited to scaling up

Virtual machine

- Pretends to be a computer
- Is a slice of computer resources
- More flexibility
 - Might have live migration etc.
- Well-suited to scaling out
- More overhead
- Noisy neighbours

Paravirtualisation

- Less support from hardware
- More collaboration from guest OS
- Mostly obsolete

Deployment: virtualisation

Bare metal

- Least overhead
- Least flexible management
- Well-suited to scaling up

Virtual machine

- Pretends to be a computer
- Is a slice of computer resources
- More flexibility
 - Might have live migration etc.
- Well-suited to scaling out
- More overhead
- Noisy neighbours

Paravirtualisation

- Less support from hardware
- More collaboration from guest OS
- Mostly obsolete

Deployment: virtualisation

Bare metal

- Least overhead
- Least flexible management
- Well-suited to scaling up

Virtual machine

- Pretends to be a computer
- Is a slice of computer resources
- More flexibility
 - Might have live migration etc.
- Well-suited to scaling out
- More overhead
- Noisy neighbours

Paravirtualisation

- Less support from hardware
- More collaboration from guest OS
- Mostly obsolete

Deployment: containers

Containers

- Same kernel

- Pretend to be a VM

- Need something to run over

Plain chroot

- Only limit filesystem view

- Very low overhead

Deployment: to where

Basically, on which level you give up

- Own server room (or more)

- Rented physical server

- Rented virtual machines

 - Manual management

- Virtual machine fleet with management API

 - Infrastructure as a Service (IaaS)

- Container runtime

- Special framework: just write the core code

 - Deployment is unified

 - Platform as a service (PaaS)

- Complete software solution on subscription

 - Software as a Service (SaaS)

Deployment: to where

Basically, on which level you give up

- Own server room (or more)

- Rented physical server

- Rented virtual machines

 - Manual management

- Virtual machine fleet with management API

 - Infrastructure as a Service (IaaS)

- Container runtime

- Special framework: just write the core code

 - Deployment is unified

 - Platform as a service (PaaS)

- Complete software solution on subscription

 - Software as a Service (SaaS)

Deployment: to where

Basically, on which level you give up

- Own server room (or more)

- Rented physical server

- Rented virtual machines

 - Manual managemetn

- Virtual machine fleet with management API

 - Infrastructure as a Service (IaaS)

- Container runtime

- Special framework: just write the core code

 - Deployment is unified

 - Platform as a service (PaaS)

- Complete software solution on subscription

 - Software as a Service (SaaS)

Deployment: to where

Basically, on which level you give up

- Own server room (or more)

- Rented physical server

- Rented virtual machines

 - Manual management

- Virtual machine fleet with management API

 - Infrastructure as a Service (IaaS)

- Container runtime

- Special framework: just write the core code

 - Deployment is unified

 - Platform as a service (PaaS)

- Complete software solution on subscription

 - Software as a Service (SaaS)

Deployment: management

Public commercial infrastructure

Private servers

Private cloud

- API as if public IaaS

- Organisational control

Hybrid

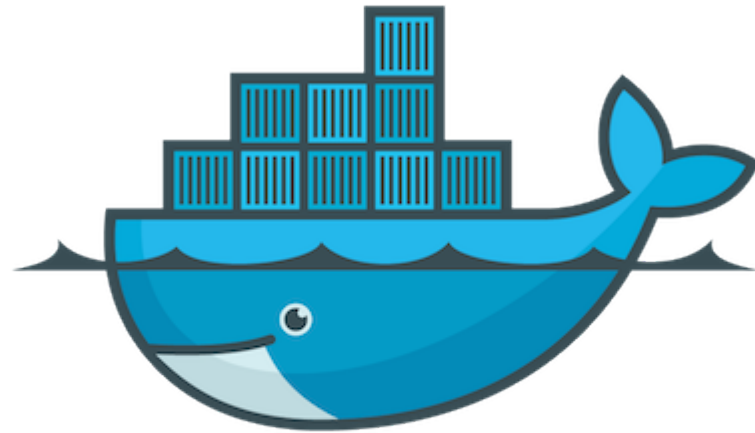
- Sensitive or predictable load workloads locally

- Public infrastructure for peak load



Introduction to Docker

Version: a2622f1



docker

An Open Platform to Build, Ship, and Run Distributed Applications

Logistics

- Updated copy of the slides: <http://lisa.dckr.info/>
- I'm Jérôme Petazzoni
- I work for Docker Inc.
- You should have a little piece of paper, with your training VM IP address + credentials
- Can't find the paper? Come get one here!
- We will make a break halfway through
- Don't hesitate to use the LISA Slack (#docker channel)
- This will be fast-paced, but DON'T PANIC
- To contact me: jerome@docker.com / Twitter: @jpetazzo

Those slides were made possible by Leon Licht, Markus Meinhardt, Ninette, Yetti Messner, and a plethora of other great artist of the Berlin techno music scene, alongside with what is probably an unhealthy amount of Club Mate.

Part 1

- About Docker
- Your training Virtual Machine
- Install Docker
- Our First Containers
- Background Containers
- Restarting and Attaching to Containers
- Understanding Docker Images
- Building Docker images
- A quick word about the Docker Hub

Part 2

- Naming and inspecting containers
- Container Networking Basics
- Local Development Work flow with Docker
- Working with Volumes
- Connecting Containers
- Ambassadors
- Compose For Development Stacks

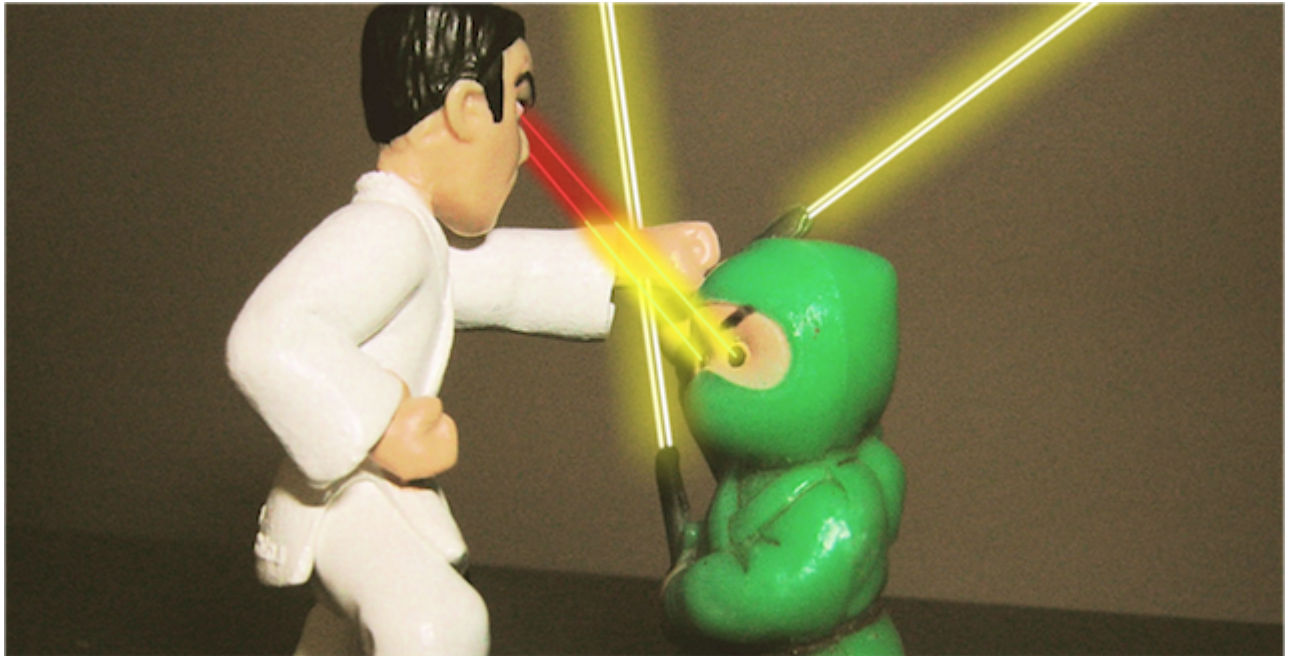
Extra material

- Advanced Dockerfiles
- Security
- Dealing with Vulnerabilities
- Securing Docker with TLS
- The Docker API

Table of Contents

About Docker.....	7
Your training Virtual Machine	37
Install Docker	42
Our First Containers.....	57
Background Containers	68
Restarting and Attaching to Containers.....	80
Understanding Docker Images.....	87
Building Images Interactively	110
Building Docker images.....	120
CMD and ENTRYPOINT	132
Copying files during the build	145
A quick word about the Docker Hub.....	152
Naming and inspecting containers.....	154
Container Networking Basics	163
Local Development Workflow with Docker.....	190
Working with Volumes.....	209
Connecting Containers.....	228
Ambassadors	245
Compose For Development Stacks	253
Advanced Dockerfiles	270
Security	297
Dealing with Vulnerabilities	311
Securing Docker with TLS	317
The Docker API	329
Course Conclusion.....	347

About Docker



Lesson 1: Docker 30,000ft overview

Objectives

In this lesson, we will learn about:

- Docker (the Open Source project)
- Docker Inc. (the company)
- Containers (how and why they are useful)

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

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

The origins of the Docker Project

- 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

- 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.

The Docker Project

- The initial container engine is now known as "Docker Engine."
- Other tools have been added:
 - Docker Compose (formerly "Fig")
 - Docker Machine
 - Docker Swarm
 - Kitematic (acquisition)
 - Tutum (recent acquisition)

About Docker Inc.

- Founded in 2009.
- Formerly dotCloud Inc.
- Primary sponsor of the Docker Project.
 - Hires maintainers and contributors.
 - Provides infrastructure for the project.
 - Runs the Docker Hub.
- HQ in San Francisco.
- Backed by more than 100M in venture capital.

How does Docker Inc. make money?

- Docker Hub has free and paid services.
- DTR (Docker Trusted Registry).
- Enterprise support for Engine and other products.
- Training and professional services.

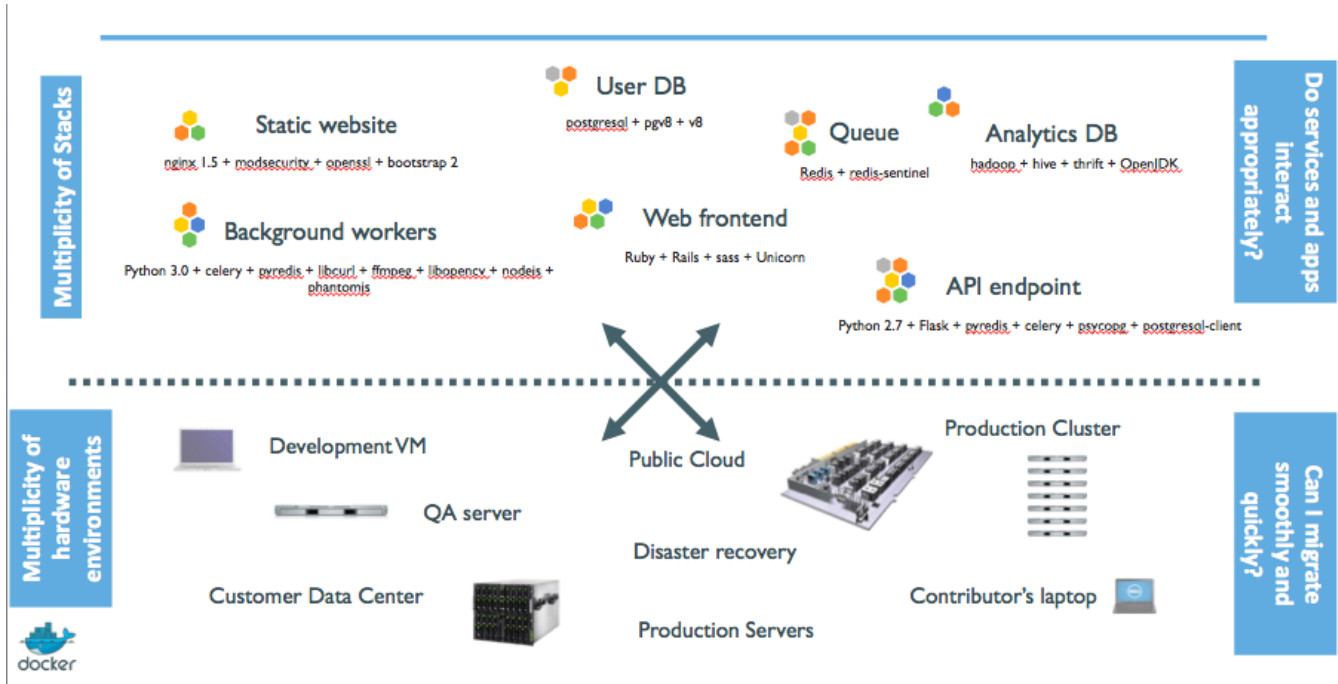
OK... Why the buzz around containers?

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

Deployment becomes very complex

- Many different stacks.
- Many different targets.

The deployment problem

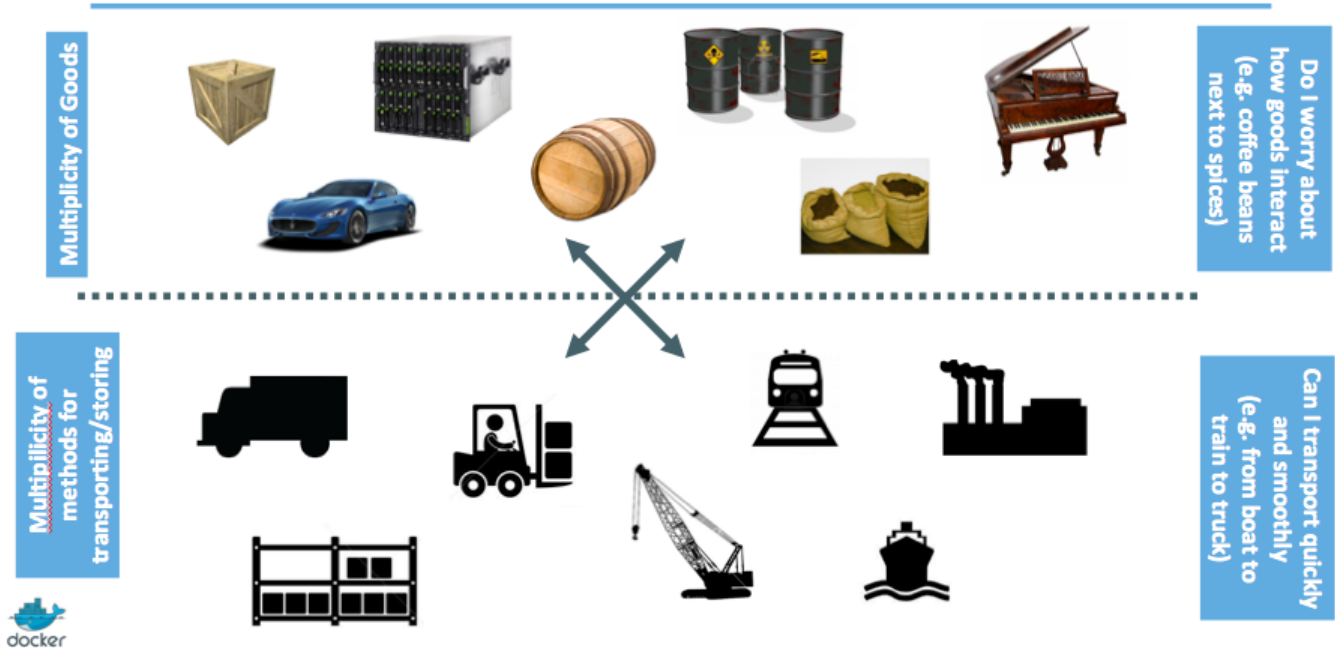


The Matrix from Hell

	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
								



An inspiration and some ancient history!



Intermodal shipping containers



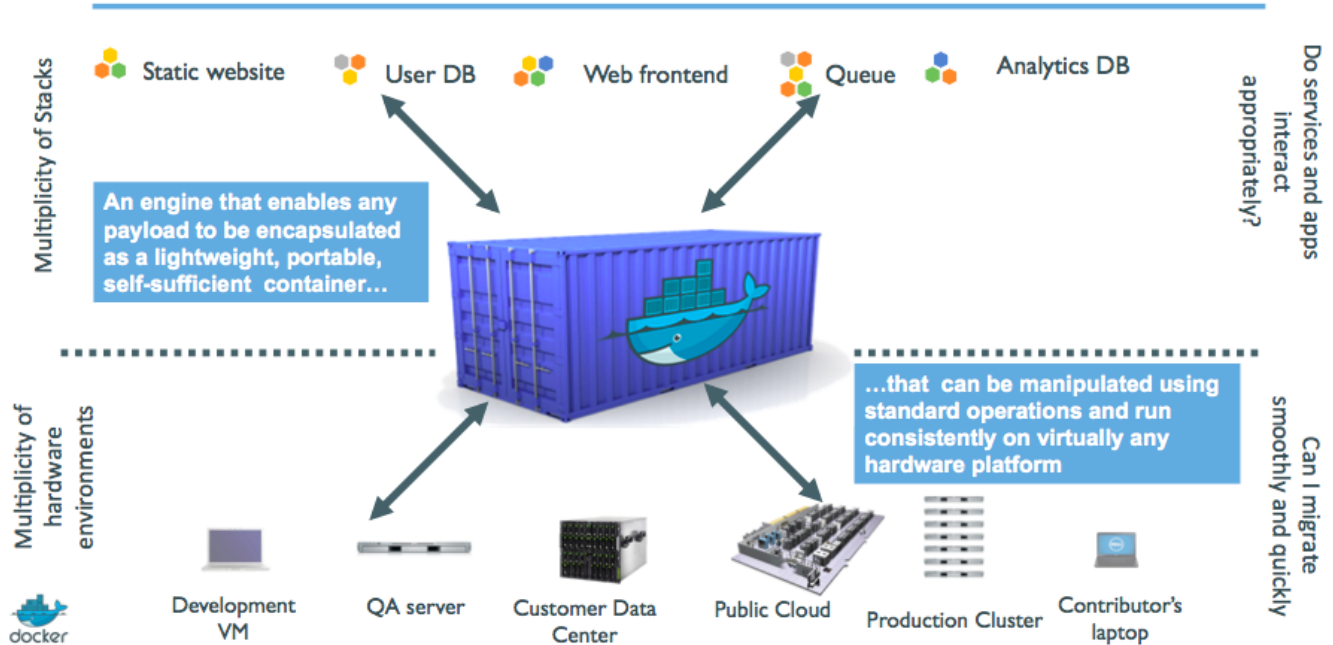
This spawned a Shipping Container Ecosystem!



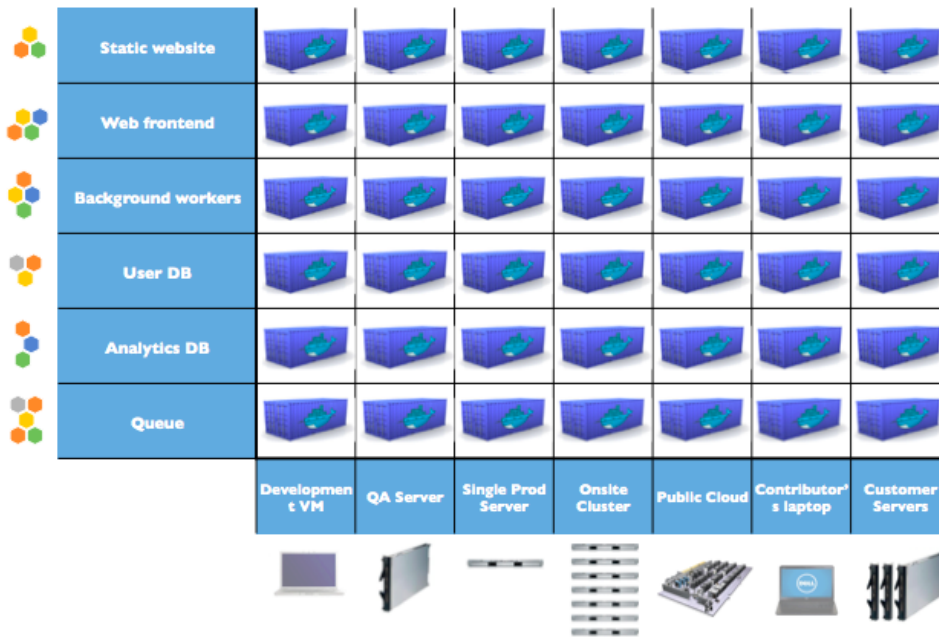
- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)
- massive globalization
- 5000 ships deliver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



A shipping container system for applications



Eliminate the matrix from Hell

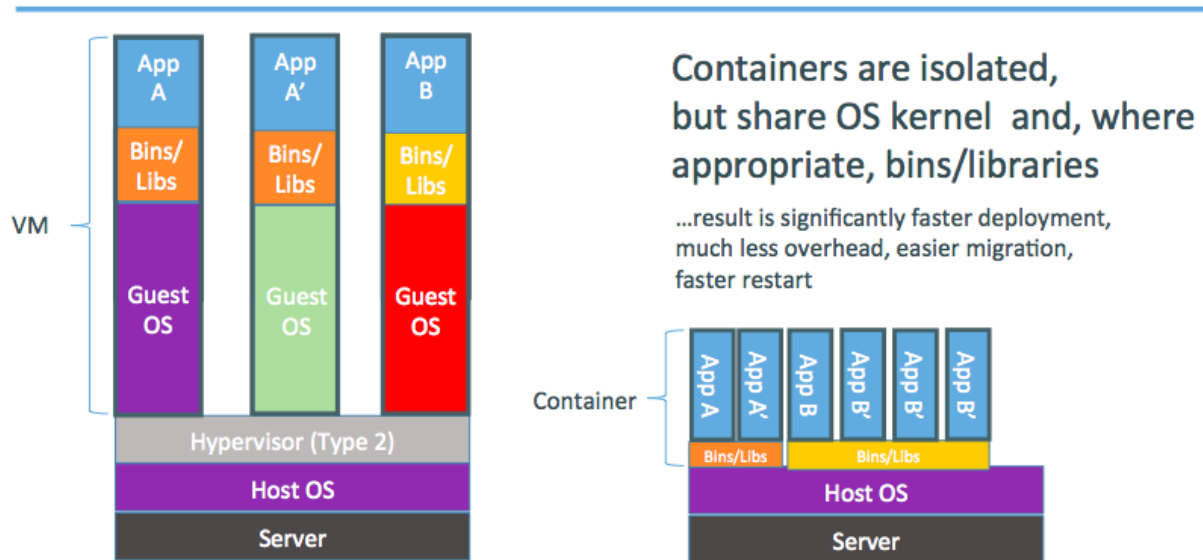


From lightweight VMs to application containers

- Containers have been around for a *very long time*.
(c.f. LXC, Solaris Zones, BSD Jails, LPAR...)
- Why are they trending now?
- What does Docker bring to the table?

Step 1: containers as lightweight VMs

Less overhead!



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

Step 2: commoditization of containers

Containers 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`?)
- 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.

Containers 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.

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.

Step 3: shipping containers efficiently

Before Docker

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

After Docker

- Ship container images with all their dependencies.
- Break image into layers.
- Only ship layers that have changed.
- Save disk, network, memory usage.

Example

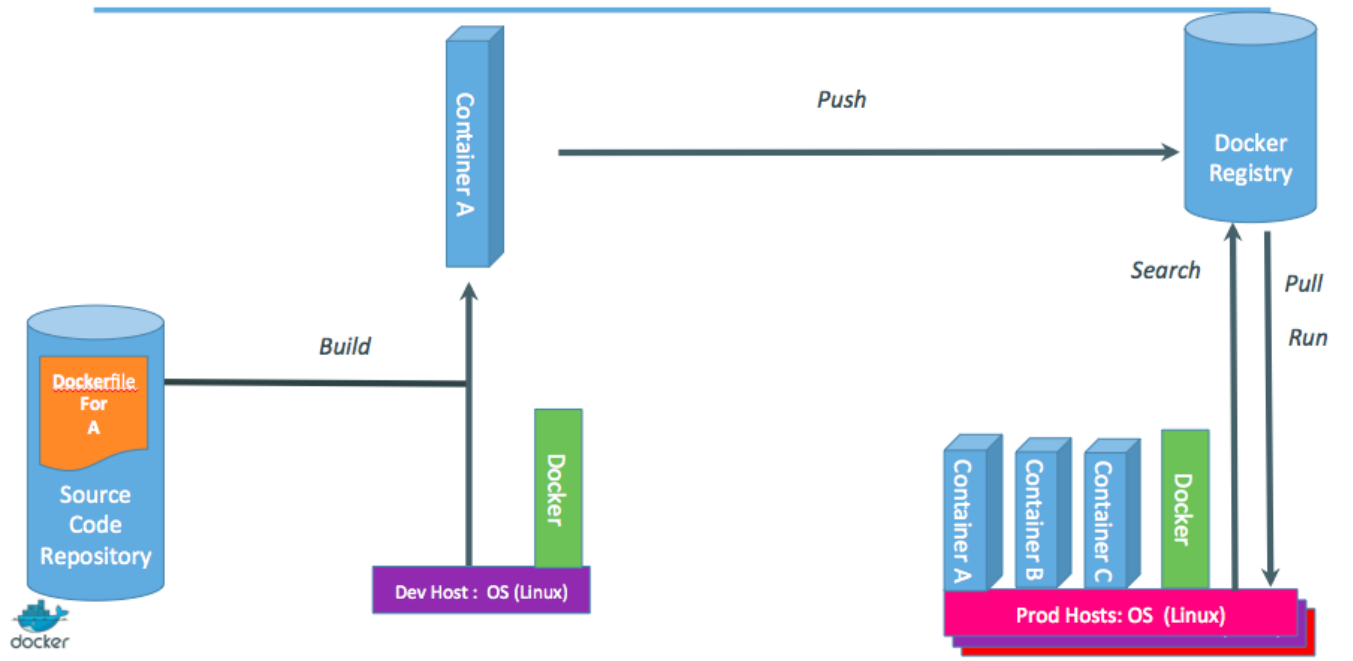
Layers:

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

Step 4: containers in a modern software factory

Container image as build artifact

The same container can go from dev, to test, to QA, to prod.



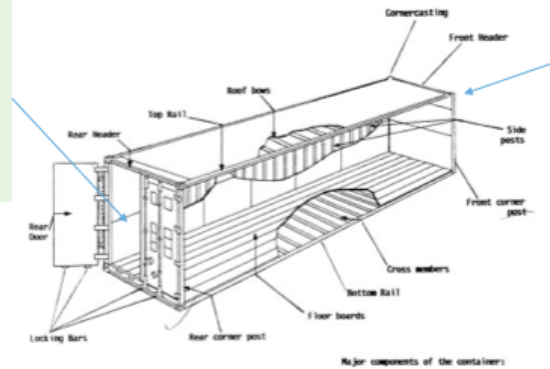
Technical & cultural revolution: separation of concerns

- Dan the Developer

- Worries about what's "inside" the container
 - His code
 - His Libraries
 - His Package Manager
 - His Apps
 - His Data
- All Linux servers look the same

- Oscar the Ops Guy

- Worries about what's "outside" the container
 - Logging
 - Remote access
 - Monitoring
 - Network config
- All containers start, stop, copy, attach, migrate, etc. the same way



Your training Virtual Machine



Lesson 2: Your training Virtual Machine

Objectives

In this section, we will see how to use your training Virtual Machine.

If you are following this course as part of an official Docker training or workshop, you have been given credentials to connect to your own private Docker VM.

If you are following this course on your own, without access to an official training Virtual Machine, just skip this lesson, and check "Installing Docker" instead.

Your training Virtual Machine

This section assumes that you are following this course as part of an official Docker training or workshop, and have been given credentials to connect to your own private Docker VM.

This VM has been created specifically for you, just before the training.

It comes pre-installed with the latest and shiniest version of Docker, as well as some useful tools.

It will stay up and running for the whole training, but it will be destroyed shortly after the training.

Connecting to your Virtual Machine

You need an SSH client.

- On OS X, Linux, and other UNIX systems, just use ssh:

```
$ ssh <login>@<ip-address>
```

- On Windows, if you don't have an SSH client, you can download Putty from www.putty.org.

Checking your Virtual Machine

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

```
$ docker version
Client version: 1.9.0
Client API version: 1.21
Go version (client): go1.4.2
Git commit (client): 76d6bc9
OS/Arch (client): linux/amd64
Server version: 1.9.0
Server API version: 1.21
Go version (server): go1.4.2
Git commit (server): 76d6bc9
```

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

Install Docker



Lesson 3: Installing Docker

Objectives

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

- Install Docker.
- Run Docker without sudo.

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

Docker is easy to install.

It runs on:

- A variety of Linux distributions.
- OS X via a virtual machine.
- Microsoft Windows via a virtual machine.

Installing Docker on Linux

It can be installed via:

- Distribution-supplied packages on virtually all distros.

(Includes at least: Arch Linux, CentOS, Debian, Fedora, Gentoo, openSUSE, RHEL, Ubuntu.)

- Packages supplied by Docker.
- Installation script from Docker.
- Binary download from Docker (it's a single file).

Installing Docker on your Linux distribution

On Fedora:

```
$ sudo yum install docker-io
```

On CentOS 7:

```
$ sudo yum install docker
```

On Debian and derivatives:

```
$ sudo apt-get install docker.io
```

Installation script from Docker

You can use the `curl` command to install on several platforms:

```
$ curl -s https://get.docker.com/ | sudo sh
```

This currently works on:

- Ubuntu
- Debian
- Fedora
- Gentoo

Installing on OS X and Microsoft Windows

Docker doesn't run natively on OS X or Microsoft Windows.

We recommend to use the Docker Toolbox, which installs the following components:

- VirtualBox + Boot2Docker VM image (runs Docker Engine)
- Kitematic GUI
- Docker CLI
- Docker Machine
- Docker Compose
- A handful of clever wrappers

Running Docker on OS X and Windows

When you execute `docker version` from the terminal:

- the CLI prepares a request for the REST API,
- environment variables tell the CLI where to send the request,
- the request goes to the Boot2Docker VM in VirtualBox,
- the Docker Engine in the VM processes the request.

Reminder: all communication happens over the API!

About boot2docker

It is a very small VM image (~30 MB).

It runs on most hypervisors and can also boot on actual hardware.

Boot2Docker is not a "lite" version of Docker.



Check that Docker is working

Using the docker client:

```
$ docker version
Client:
 Version:      1.9.0
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   76d6bc9
 Built:        Tue Nov  3 17:29:38 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.9.0
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   76d6bc9
 Built:        Tue Nov  3 17:29:38 UTC 2015
 OS/Arch:      linux/amd64
```

Su-su-sudo



Important PSA about security

The `docker` user is `root` equivalent.

It provides `root`-level access to the host.

You should restrict access to it like you would protect `root`.

If you give somebody the ability to access the Docker API, you are giving them full access on the machine.

Therefore, the Docker control socket is (by default) owned by the `docker` group, to avoid unauthorized access on multi-user machines.

The docker group

Add the Docker group

```
$ sudo groupadd docker
```

Add ourselves to the group

```
$ sudo gpasswd -a $USER docker
```

Restart the Docker daemon

```
$ sudo service docker restart
```

Log out

```
$ exit
```

Check that Docker works without sudo

```
$ docker version
Client:
 Version:      1.9.0
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   76d6bc9
 Built:        Tue Nov  3 17:29:38 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.9.0
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   76d6bc9
 Built:        Tue Nov  3 17:29:38 UTC 2015
 OS/Arch:      linux/amd64
```

Section summary

We've learned how to:

- Install Docker.
- Run Docker without sudo.

Our First Containers



Lesson 4: Our First Containers

Objectives

At the end of this lesson, you will have:

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

Docker architecture

Docker is a client-server application.

- **The Docker daemon (or "Engine")**
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 busybox echo hello world  
hello world
```

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'd `hello world`.

A more useful container

Let's run a more exciting container:

```
$ docker run -it ubuntu bash
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.

An obvservation

Let's check how many packages are installed here.

```
root@04c0bb0a6c07:/# dpkg -l | wc -l  
189
```

- `dpkg -l` lists the packages installed in our container
- `wc -l` counts them
- If you have a Debian or Ubuntu machine, you can run the same command and compare the results.

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!

```
# figlet hello
```



Exiting our container

Just exit the shell, like you would usually do.

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

```
root@04c0bb0a6c07:/# 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 bash
root@b13c164401fb:/# figlet
bash: figlet: command not found
```

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

Background Containers



Lesson 5: 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 tell 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                COMMAND             CREATED          STATUS            ...
47d677dcfba4   jpetazzo/clock:latest ...                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.

Two useful flags for docker ps

To see only the last container that was started:

```
$ docker ps -l
CONTAINER ID   IMAGE                  COMMAND                  CREATED          STATUS          ...
47d677dcfba4   jpetazzo/clock:latest ...                    2 minutes ago   Up 2 minutes    ...
```

To see only the ID of containers:

```
$ docker ps -q
47d677dcfba4
66b1ce719198
ee0255a5572e
```

Combine those flags to see only the ID of the last container started!

```
$ docker ps -lq
47d677dcfba4
```

View the logs of a container

We told you that Docker was logging the container output.

Let's see that now.

```
$ docker logs 47d6
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 47d6
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 47d6
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.

Killing it

Let's kill our container:

```
$ docker kill 47d6  
47d6
```

Docker will echo the ID of the container we've just stopped.

Let's check that our container doesn'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
47d677dcfba4   jpetazzo/clock:latest ...   23 min. ago      Exited (0) 4 min. ago
5c1dfd4d81f1   jpetazzo/clock:latest ...   40 min. ago      Exited (0) 40 min. ago
b13c164401fb   ubuntu:latest         ...   55 min. ago      Exited (130) 53 min. ago
```

Restarting and Attaching to Containers

Lesson 6: Restarting and Attaching to Containers

Objectives

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.

Detaching from a container

- If you have started an *interactive* container (with option `-i t`), 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 `-i t` stand for?

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

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.
- **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.

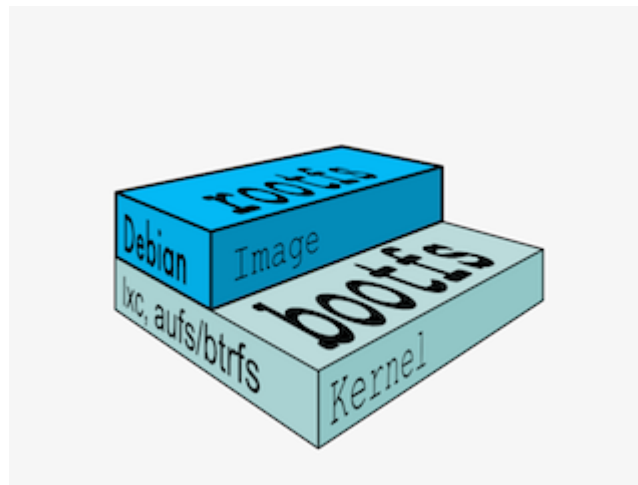
It can then be restarted with the `start` command.

```
$ 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.

Understanding Docker Images



Lesson 7: Understanding Docker Images

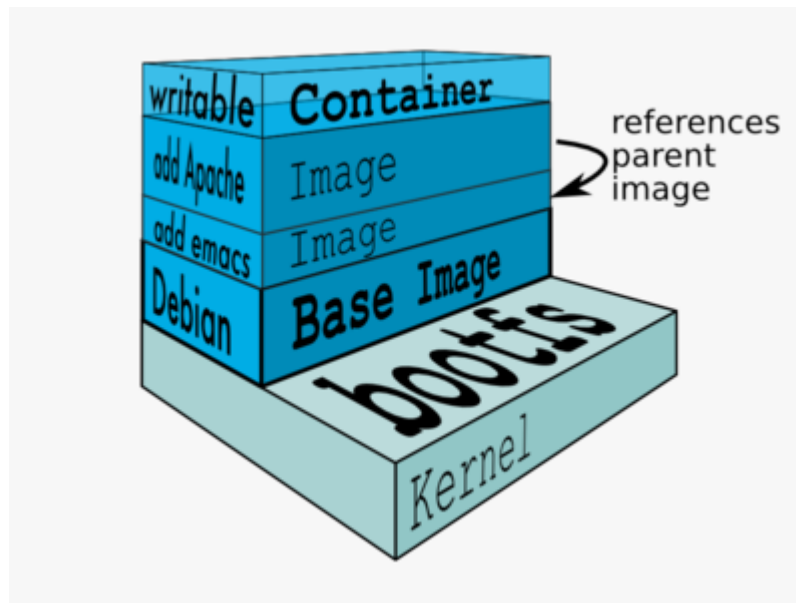
Objectives

In this lesson, we will explain:

- What is an image.
- What is a layer.
- The various image namespaces.
- How to search and download images.

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.



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.

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.

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`

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

We will explain both methods in a moment.

Images namespaces

There are three namespaces:

- Root-like

```
ubuntu
```

- User (and organizations)

```
jpetazzo/clock
```

- Self-Hosted

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

Let's explain each of them.

Root namespace

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

Those images include:

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

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
```

The remote host and port is:

```
localhost:5000
```

The image name is:

```
wordpress
```

Historical detail

Self-hosted registries used to be called *private* registries, but this was misleading!

- A self-hosted registry can be public or private.
- A registry in the User namespace on Docker Hub can be public or private.

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       VIRTUAL SIZE
ubuntu              13.10       9f676bd305a4 7 weeks ago  178 MB
ubuntu              saucy       9f676bd305a4 7 weeks ago  178 MB
ubuntu              raring      eb601b8965b8 7 weeks ago  166.5 MB
ubuntu              13.04       eb601b8965b8 7 weeks ago  166.5 MB
ubuntu              12.10       5ac751e8d623 7 weeks ago  161 MB
ubuntu              quantal     5ac751e8d623 7 weeks ago  161 MB
ubuntu              10.04       9cc9ea5ea540 7 weeks ago  180.8 MB
ubuntu              lucid       9cc9ea5ea540 7 weeks ago  180.8 MB
ubuntu              12.04       9cd978db300e 7 weeks ago  204.4 MB
ubuntu              latest      9cd978db300e 7 weeks ago  204.4 MB
ubuntu              precise     9cd978db300e 7 weeks ago  204.4 MB
```

Searching for images

Searches your registry for images:

```
$ docker search zookeeper
NAME                                DESCRIPTION                                STARS   ...
jplock/zookeeper                   Builds a docker image for ...             27
thefactory/zookeeper-exhibitor     Exhibitor-managed ZooKeepe...             2
misakai/zookeeper                  ZooKeeper is a service for...             1
digitalwonderland/zookeeper        Latest Zookeeper - cluster...             1
garland/zookeeper                  ...                                         1
raycoding/piggybank-zookeeper       Zookeeper 3.4.6 running on...             1
gregory90/zookeeper                 ...                                         0
```

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

Downloading images

There are two ways to download images.

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

Pulling an image

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

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

Image and tags

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

When to (not) use tags

Don't specify tags:

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

Do specify tags:

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

Section summary

We've learned how to:

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

Building Images Interactively



Lesson 8: Building Images Interactively

Objectives

In this lesson, 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`.

Building Images Interactively

As we have seen, the images on the Docker Hub are sometimes very basic.

How do we want to construct our own images?

As an example, we will build an image that has `figlet`.

First, we will do it manually with `docker commit`.

Then, in an upcoming chapter, we will use a `Dockerfile` and `docker build`.

Building from a base

Our base will be the ubuntu image.

Create a new container and make some changes

Start an Ubuntu container:

```
$ docker run -it ubuntu bash  
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.)

Commit and run your image

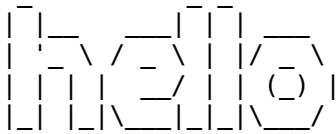
The `docker commit` command will create a new layer with those changes, and a new image using this new layer.

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

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

We can run this image:

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



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
```

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



Lesson 9: Building 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
```

2. 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; 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:

```
$ docker build -t figlet .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
--> e54ca5efa2e9
Step 1 : RUN apt-get update
--> Running in 840cb3533193
--> 7257c37726a1
Removing intermediate container 840cb3533193
Step 2 : RUN apt-get install figlet
--> Running in 2b44df762a2f
--> f9e8f1642759
Removing intermediate container 2b44df762a2f
Successfully built f9e8f1642759
```

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

Sending the build context to Docker

Sending build context to Docker daemon 2.048 kB

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

Executing each step

```
Step 1 : RUN apt-get update
--> Running in 840cb3533193
(...output of the RUN command...)
--> 7257c37726a1
Removing intermediate container 840cb3533193
```

- A container (840cb3533193) is created from the base image.
- The RUN command is executed in this container.
- The container is committed into an image (7257c37726a1).
- The build container (840cb3533193) is removed.
- The output of this step will be the base image for the next one.

The caching system

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

Why?

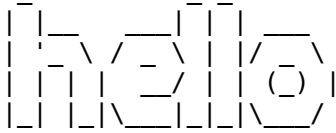
- After each build step, Docker takes a snapshot of the resulting image.
- Before executing a step, Docker checks if it has already built the same sequence.
- Docker uses the exact strings defined in your Dockerfile, so:
 - `RUN apt-get install figlet cowsay` is different from `RUN apt-get install cowsay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

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

Running the image

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

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



- Sweet is the taste of success!

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 6.062 MB
7257c37726a1   About an hour ago /bin/sh -c apt-get update      8.549 MB
e54ca5efa2e9    8 months ago     /bin/sh -c apt-get update &&    8 B
6c37f792ddac    8 months ago     /bin/sh -c apt-get update &&   83.43 MB
83ff768040a0    8 months ago     /bin/sh -c sed -i s/^#\s*\(\d 1.903 kB
2f4b4d6a4a06    8 months ago     /bin/sh -c echo #!/bin/sh >    194.5 kB
d7ac5e4f1812    8 months ago     /bin/sh -c #(nop) ADD file:ad 192.5 MB
511136ea3c5a    20 months ago     
```

CMD and ENTRYPOINT



Lesson 10: CMD and ENTRYPOINT

Objectives

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

CMD and ENTRYPOINT.

Those 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.

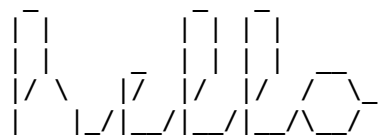
Build and test our image

Let's build it:

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

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.

Build and test our image

Let's build it:

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

And run it:

```
$ docker run figlet salut
```

Great success!

Using CMD and ENTRYPOINT together

What if we want to define a default URL 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.

CMD and ENTRYPOINT together

Our new Dockerfile will look like this:

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

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

Build and test our image

Let's build it:

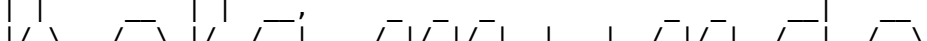
```
$ docker build -t figlet .
...
Successfully built 6e0b6a048a07
```

And run it:

```
$ docker run figlet
```



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



Overriding ENTRYPOINT

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

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

We use the `--entrypoint` parameter:

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

Copying files during the build



Lesson 11: 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 compiler, we would:
 - Place it in a different directory, with the WORKDIR instruction.
 - Even better, use the gcc official image.