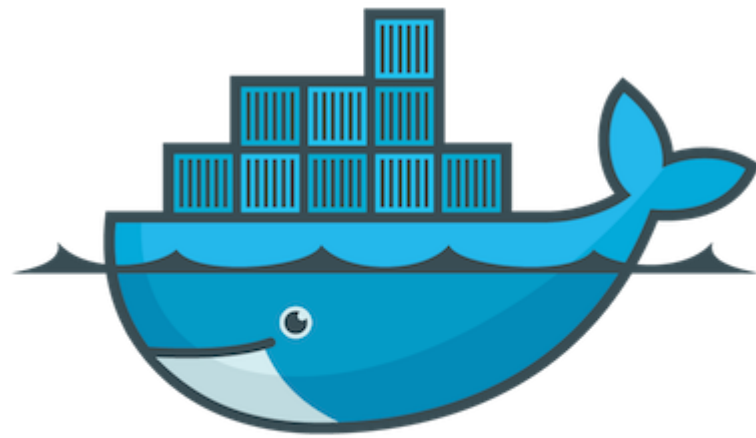


Introduction to Docker

Version: d7590d1



docker

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

Logistics

- Updated copy of the slides: <http://container.training/>
- Note: the PDF version has extra slides
- You should have a little piece of paper, with your training VM IP address + credentials
- Can't find the paper? Raise your hand and ask for one!
- Backchannel:
container.training/chat
 - students: feel free to ask questions there
 - TAs: join the channel to answer questions

Part 3

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

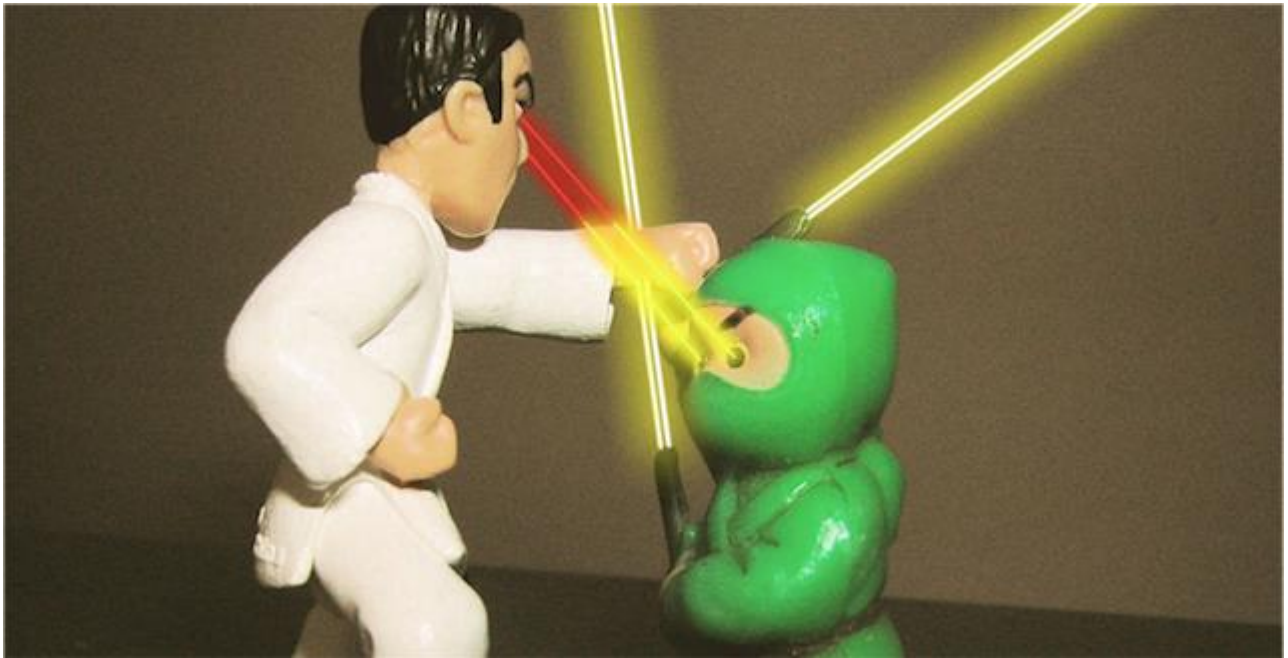
Part 4

- Naming and inspecting containers
- Introduction to Container Networking
- Container Network Model
- (Connecting Containers with Links)
- (Ambassadors)
- Local Development Workflow with Docker
- Working with Volumes
- Compose For Development Stacks

Table of Contents

About Docker	6
Your training Virtual Machine	51
Install Docker	56
Our First Containers	75
Background Containers	86
Restarting and Attaching to Containers	100
Understanding Docker Images	111
Building Images Interactively	133
Building Docker images	143
CMD and ENTRYPOINT	157
Copying files during the build	171
Advanced Dockerfiles	178
A quick word about the Docker Hub	208
Naming and inspecting containers	210
Container Networking Basics	219
The Container Network Model	237
Connecting Containers With Links	264
Ambassadors	278
Local Development Workflow with Docker	290
Working with Volumes	309
Compose For Development Stacks	333
Course Conclusion	351

About Docker



Lesson 1: Docker 30,000ft overview

Objectives

In this lesson, we will learn about:

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

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

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

Elevator pitch

(for your manager, boss...)

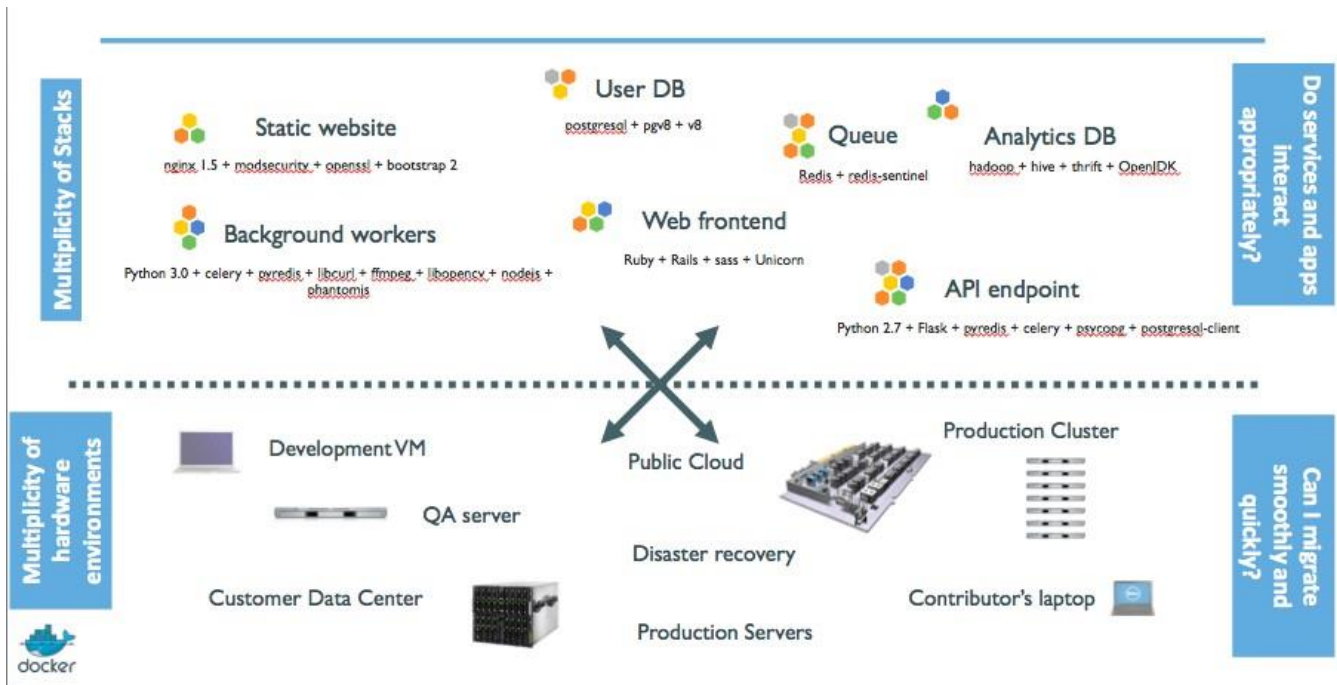
OK... Why the buzz around containers?

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














Deployment becomes very complex

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

The deployment problem

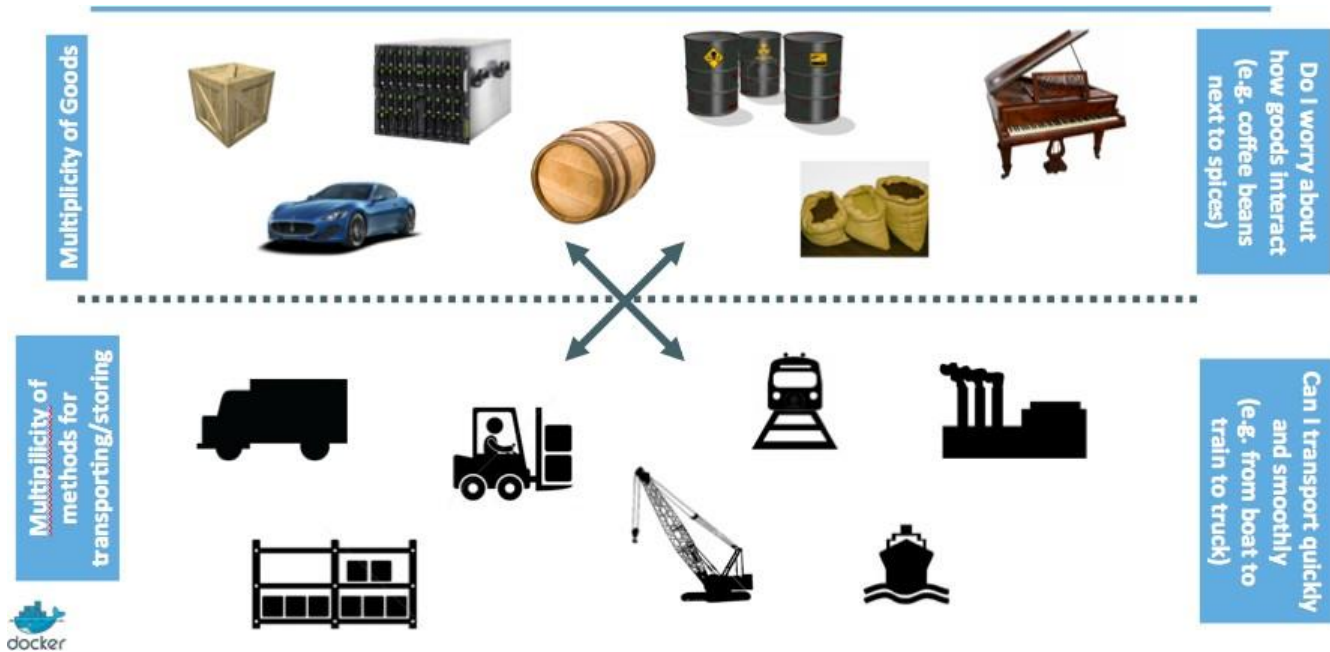


The Matrix from Hell

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



An inspiration and some ancient history!



Intermodal shipping containers



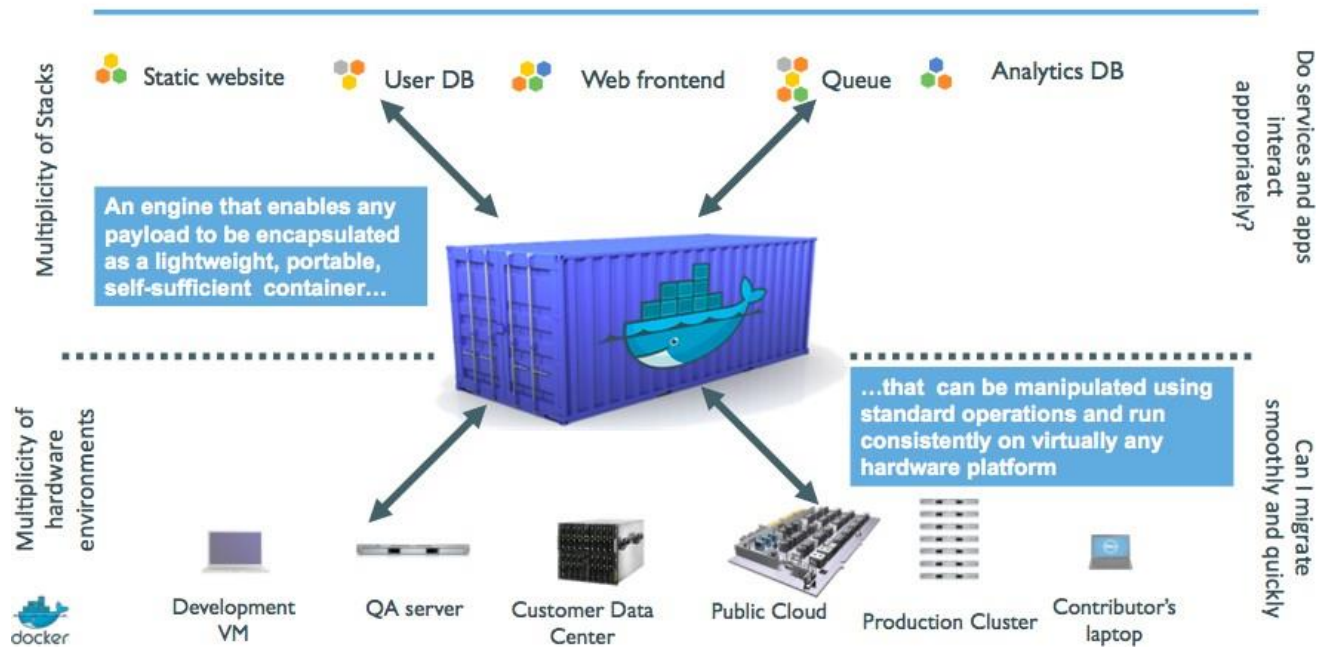
This spawned a Shipping Container Ecosystem!



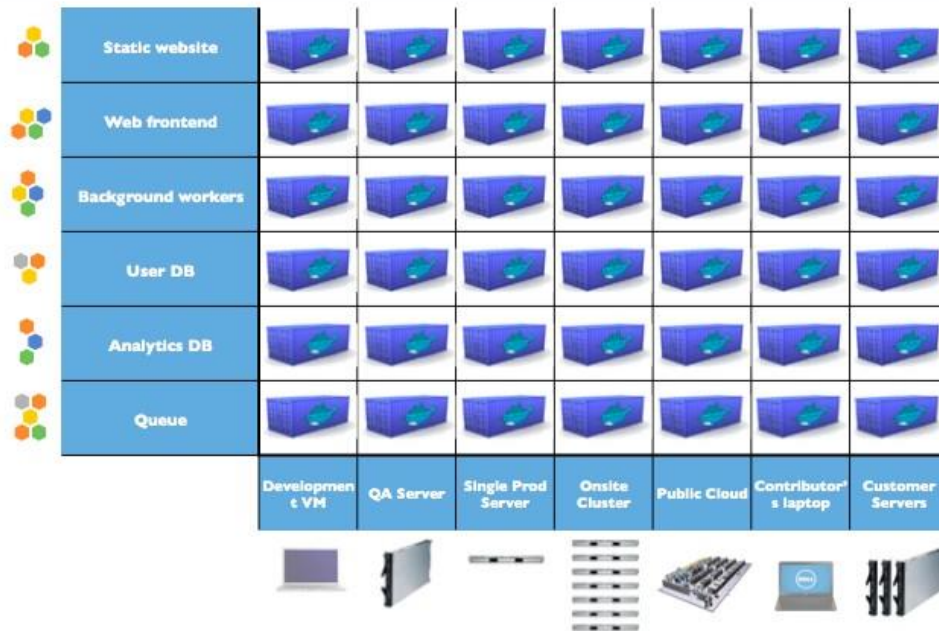
- 90% of all cargo now shipped in a standard container
 - Order of magnitude reduction in cost and time to load and unload ships
 - Massive reduction in losses due to theft or damage
 - Huge reduction in freight cost as percent of final goods (from >25% to <3%)
- massive globalization
- 5000 ships deliver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



A shipping container system for applications



Eliminate the matrix from Hell



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!

Use container images as build artefacts

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

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

Decouple "plumbing" from application logic

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

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

What did Docker bring to the table?

Docker before/after

Formats and APIs, before Docker

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

Analogy:

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

Formats and APIs, after Docker

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

Shipping, before Docker

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

Shipping, after Docker

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

Example

Layers:

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

Devs vs Ops, before Docker

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

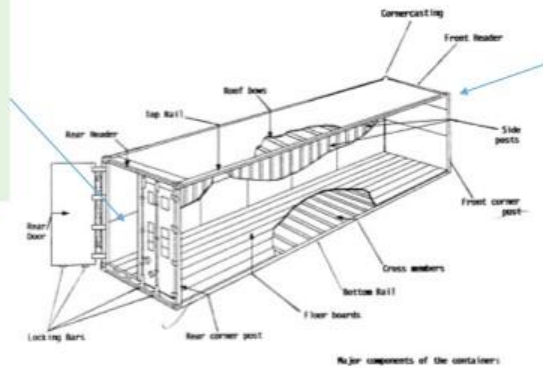
Devs vs Ops, after Docker

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

Clean 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~~ ^{connect}
- All containers start, stop, copy, attach, migrate, etc. the same way



History of containers ... and Docker

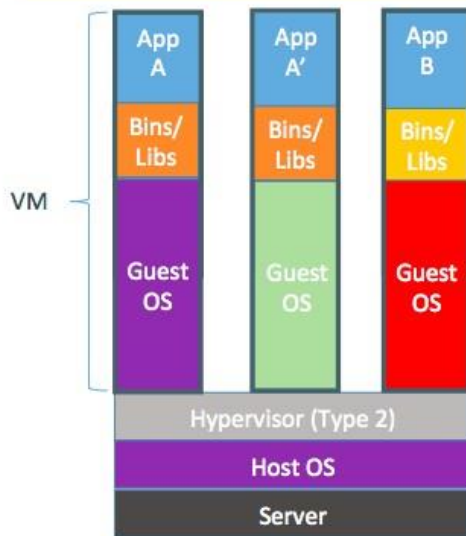
First experimentations

- [IBM VM/370 \(1972\)](#)
- [Linux VServers \(2001\)](#)
- [Solaris Containers \(2004\)](#)
- [FreeBSD jails \(1999\)](#)

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

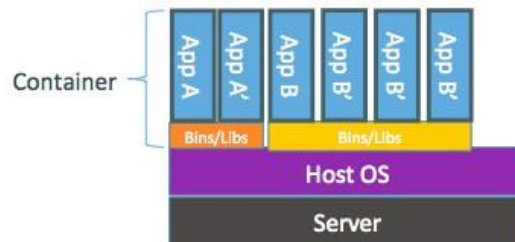
VPS-olitic period (until 2007-2008)

Containers = cheaper than VMs



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

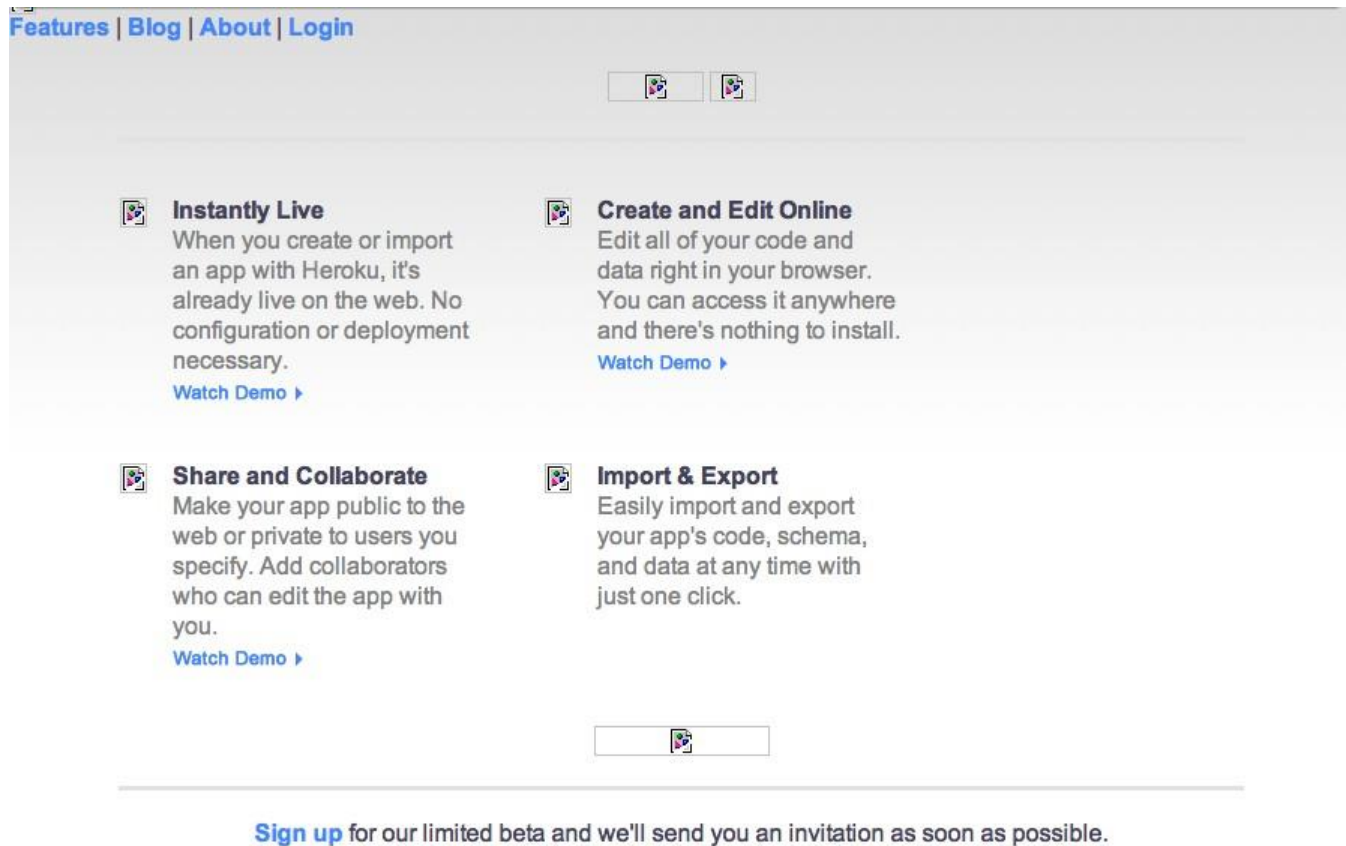
...result is significantly faster deployment, much less overhead, easier migration, faster restart



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

PaaS-olithic period (2008-2013)

Containers = easier than VMs



- I can't speak for Heroku, but containers were (one of) dotCloud's secret weapon

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.

Docker early days (2013-2014)

First users of Docker

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

Positive feedback loop

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

Maturity (2015-2016)

Docker becomes an industry standard

- Docker reaches the symbolic 1.0 milestone.
- Existing systems like Mesos and Cloud Foundry add Docker support.
- Standards like OCI, CNCF appear.
- Other container engines are developed.

Docker becomes a platform

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

Docker Inc. (the company)

About Docker Inc.

- Docker Inc. used to be dotCloud Inc.
- dotCloud Inc. used to be a French company.
- Docker Inc. is the primary sponsor and contributor to 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?

- SAAS
 - Docker Hub (per private repo)
 - Docker Cloud (per node)
- Subscription
 - on-premise stack (Docker Datacenter)
 - DTR (Docker Trusted Registry)
 - UCP (Universal Control Plane)
 - CS (Commercially Supported Engine)
- Support
- Training and professional services

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 (www.putty.org)
 - Git BASH (<https://git-for-windows.github.io/>)
 - MobaXterm (<http://moabaxterm.mobatek.net>)

Checking your Virtual Machine

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

```
$ docker version
Client:
 Version:      1.11.1
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   5604cbe
 Built:        Tue Apr 26 23:38:55 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.11.1
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   5604cbe
 Built:        Tue Apr 26 23:38:55 2016
 OS/Arch:      linux/amd64
```

- 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 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

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 with upstream packages

- Preferred method to install Docker on Linux.
- Upstream's packages are more up-to-date than distros'.
- Instructions per distro:
<https://docs.docker.com/engine/installation/linux/>
- Package will be named `docker-engine`.

Installing Docker with distros packages

On Red Hat derivatives (Fedora, CentOS):

```
$ 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.

There are three ways to get Docker on OS X or Windows:

- Using Docker Mac or Docker Windows (recommended);
- Using the Docker Toolbox (formerly recommended);
- Rolling your own with e.g. Parallels, VirtualBox, VMware...

Running Docker on OS X 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.

Rolling your own install

- Good luck, you're on your own!
- There is (almost?) no good reason to do that.
- If you want to do something very custom, the Docker Toolbox is probably better anyway.

Using the Docker Toolbox

The Docker Toolbox installs the following components:

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

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.



Docker Mac and Docker Windows

- Docker Mac and Docker Windows are newer products
- They let you run Docker without VirtualBox
- They are installed like normal applications (think QEMU, but faster)
- They provide better integration with enterprise VPNs
- They support filesystem sharing through volumes (we'll talk about this later)

Only downside (for now): only one instance at a time; so if you want to run a full cluster on your local machine, you can fallback on the Docker Toolbox (it can coexist with Docker Mac/Windows just fine).

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.

If your user is not in the `docker` group, you will need to prefix every command with `sudo`; e.g. `sudo docker version`.

Reminder ...

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`.

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:      1.11.1
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   5604cbe
 Built:        Tue Apr 26 23:38:55 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.11.1
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   5604cbe
 Built:        Tue Apr 26 23:38:55 2016
 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 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 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'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.

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
```

```
h e l l o
```

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
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 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.

Two useful flags for `docker ps`

To see only the last container that was started:

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

To see only the ID of containers:

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

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

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

View the logs of a container

We told you that Docker was logging the container output.

Let's see that now.

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

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

View only the tail of the logs

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

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

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

Follow the logs in real time

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

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

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

Stop our container

There are two ways we can terminate our detached container.

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

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

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

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

Stopping our containers

Let's stop one of those containers:

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

This will take 10 seconds:

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

Killing the remaining containers

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

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

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

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

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

```
$ docker ps
```

List stopped containers

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

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

Restarting and Attaching to Containers

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.

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

Detaching from a container

- If you have started an *interactive* container (with option `-it`), you can detach from it.
- The "detach" sequence is $^P^Q$.
- Otherwise you can detach by killing the Docker client.
(But not by hitting C , as this would deliver `SIGINT` to the container.)

What does `-it` stand for?

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

Specifying a custom detach sequence

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

Start a container with a custom detach command:

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

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

Check that our container is still running:

```
$ docker ps -l
```


Attaching to a container

You can attach to a container:

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

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

Try it on our previous container:

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

Check that `^X` 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.

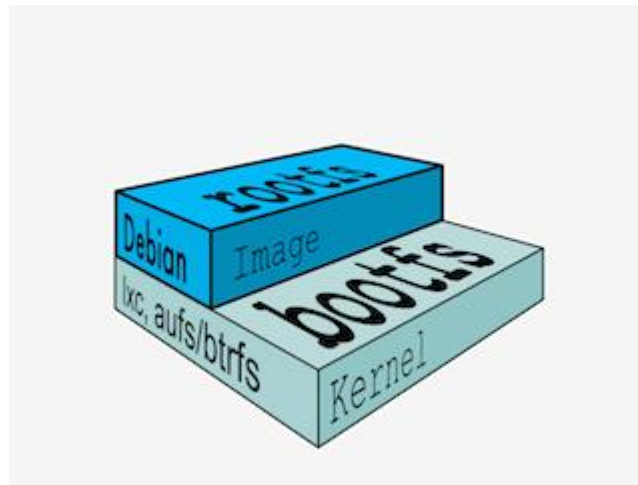
Attaching to a REPL

- 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 a couple of 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



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.
- 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

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:

- Official images
e.g. `ubuntu`, `busybox` ...
- User (and organizations) images
e.g. `jpetazzo/clock`
- Self-hosted images
e.g. `registry.example.com:5000/my-private/image`

Let's explain each of them.

Root namespace

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

Those images include:

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

User namespace

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

For example:

jpetazzo/clock

The Docker Hub user is:

jpetazzo

The image name is:

clock

Self-Hosted namespace

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

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

For example:

```
localhost:5000/wordpress
```

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

How do you store and manage images?

Images can be stored:

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

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

To be more accurate: you can use the Docker client to tell a Docker 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
```

Searching for images

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

```
$ docker search zookeeper
NAME                DESCRIPTION                STARS   OFFICIAL   AUTOMATED
jplock/zookeeper    Builds a docker image ...  103
mesoscloud/zookeeper ZooKeeper                  42
springxd/zookeeper  A Docker image that ca...  5
elevy/zookeeper     ZooKeeper configured t...  3
```

- "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  
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

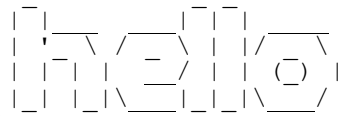
The `docker commit` command will create a new layer with those changes, and a new image using this new layer.

```
$ 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?

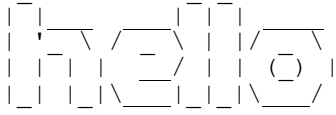
- After each build step, Docker takes a snapshot of the resulting image.
- Before executing a step, Docker checks if it has already built the same sequence.
- Docker uses the exact strings defined in your Dockerfile, so:
 - `RUN apt-get install figlet cowsay`
is different from
`RUN apt-get install cowsay 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 1.627 MB
7257c37726a1   About an hour ago /bin/sh -c apt-get update      21.58 MB
07c86167cdc4    4 days ago       /bin/sh -c #(nop) CMD ["/bin 0 B
<missing>       4 days ago       /bin/sh -c sed -i 's/^#\s*\(( 1.895 kB
<missing>       4 days ago       /bin/sh -c echo '#!/bin/sh'    194.5 kB
<missing>       4 days ago       /bin/sh -c #(nop) ADD file:b    187.8 MB
```

Introducing JSON syntax

Most Dockerfile arguments can be passed in two forms:

- plain string:
`RUN apt-get install figlet`
- JSON list:
`RUN ["apt-get", "install", "figlet"]`

Let's change our Dockerfile as follows!

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

Then build the new Dockerfile.

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

JSON syntax vs string syntax

Compare the new history:

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

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

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.


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

```
$ 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, you must 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
```

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

```
$ docker run figlet salut
```

Using CMD and ENTRYPOINT together

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

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

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

CMD and ENTRYPOINT together

Our new Dockerfile will look like this:

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

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

Build and test our image

Let's build it:

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

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.

Advanced Dockerfiles



Lesson 12: Advanced Dockerfiles

Objectives

We have seen simple Dockerfiles to illustrate how Docker build container images. In this chapter, we will see:

- The syntax and keywords that can be used in Dockerfiles.
- Tips and tricks to write better Dockerfiles.

Dockerfile usage summary

- `Dockerfile` instructions are executed in order.
- Each instruction creates a new layer in the image.
- Instructions are cached. If no changes are detected then the instruction is skipped and the cached layer used.
- The `FROM` instruction **MUST** be the first non-comment instruction.
- Lines starting with `#` are treated as comments.
- You can only have one `CMD` and one `ENTRYPOINT` instruction in a `Dockerfile`.

The FROM instruction

- Specifies the source image to build this image.
- Must be the first instruction in the `Dockerfile`, except for comments.

The FROM instruction

Can specify a base image:

```
FROM ubuntu
```

An image tagged with a specific version:

```
FROM ubuntu:12.04
```

A user image:

```
FROM training/sinatra
```

Or self-hosted image:

```
FROM localhost:5000/funtoo
```

More about FROM

- The FROM instruction can be specified more than once to build multiple images.

```
FROM ubuntu:14.04
. . .
FROM fedora:20
. . .
```

Each FROM instruction marks the beginning of the build of a new image. The `-t` command-line parameter will only apply to the last image.

- If the build fails, existing tags are left unchanged.
- An optional version tag can be added after the name of the image.
E.g.: `ubuntu:14.04.`

A use case for multiple FROM lines

- Integrate CI and unit tests in the build system

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

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

The MAINTAINER instruction

The MAINTAINER instruction tells you who wrote the Dockerfile.

```
MAINTAINER Docker Education Team <education@docker.com>
```

It's optional but recommended.

The RUN instruction

The `RUN` instruction can be specified in two ways.

With shell wrapping, which runs the specified command inside a shell, with `/bin/sh -c`:

```
RUN apt-get update
```

Or using the `exec` method, which avoids shell string expansion, and allows execution in images that don't have `/bin/sh`:

```
RUN [ "apt-get", "update" ]
```

More about the `RUN` instruction

`RUN` will do the following:

- Execute a command.
- Record changes made to the filesystem.
- Work great to install libraries, packages, and various files.

`RUN` will NOT do the following:

- Record state of *processes*.
- Automatically start daemons.

If you want to start something automatically when the container runs, you should use `CMD` and/or `ENTRYPOINT`.

Collapsing layers

It is possible to execute multiple commands in a single step:

```
RUN apt-get update && apt-get install -y wget && apt-get clean
```

It is also possible to break a command onto multiple lines:

It is possible to execute multiple commands in a single step:

```
RUN apt-get update \  
&& apt-get install -y wget \  
&& apt-get clean
```

The EXPOSE instruction

The `EXPOSE` instruction tells Docker what ports are to be published in this image.

```
EXPOSE 8080
EXPOSE 80 443
EXPOSE 53/tcp 53/udp
```

- All ports are private by default.
- The `Dockerfile` doesn't control if a port is publicly available.
- When you `docker run -p <port> . . .`, that port becomes public. (Even if it was not declared with `EXPOSE`.)
- When you `docker run -P . . .` (without port number), all ports declared with `EXPOSE` become public.

A *public port* is reachable from other containers and from outside the host.

A *private port* is not reachable from outside.

The COPY instruction

The `COPY` instruction adds files and content from your host into the image.

```
COPY . /src
```

This will add the contents of the *build context* (the directory passed as an argument to `docker build`) to the directory `/src` in the container.

Note: you can only reference files and directories *inside* the build context. Absolute paths are taken as being anchored to the build context, so the two following lines are equivalent:

```
COPY . /src  
COPY / /src
```

Attempts to use `..` to get out of the build context will be detected and blocked with Docker, and the build will fail.

Otherwise, a `Dockerfile` could succeed on host A, but fail on host B.

ADD

ADD works almost like COPY, but has a few extra features.

ADD can get remote files:

```
ADD http://www.example.com/webapp.jar /opt/
```

This would download the `webapp.jar` file and place it in the `/opt` directory.

ADD will automatically unpack zip files and tar archives:

```
ADD ./assets.zip /var/www/html/assets/
```

This would unpack `assets.zip` into `/var/www/html/assets`.

However, ADD will not automatically unpack remote archives.

ADD, COPY, and the build cache

- For most Dockerfiles instructions, Docker only checks if the line in the Dockerfile has changed.
- For `ADD` and `COPY`, Docker also checks if the files to be added to the container have been changed.
- `ADD` always need to download the remote file before it can check if it has been changed. (It cannot use, e.g., ETags or If-Modified-Since headers.)

VOLUME

The `VOLUME` instruction tells Docker that a specific directory should be a *volume*.

```
VOLUME /var/lib/mysql
```

Filesystem access in volumes bypasses the copy-on-write layer, offering native performance to I/O done in those directories.

Volumes can be attached to multiple containers, allowing to "port" data over from a container to another, e.g. to upgrade a database to a newer version.

It is possible to start a container in "read-only" mode. The container filesystem will be made read-only, but volumes can still have read/write access if necessary.

The WORKDIR instruction

The `WORKDIR` instruction sets the working directory for subsequent instructions.

It also affects `CMD` and `ENTRYPOINT`, since it sets the working directory used when starting the container.

```
WORKDIR /src
```

You can specify `WORKDIR` again to change the working directory for further operations.

The `ENV` instruction

The `ENV` instruction specifies environment variables that should be set in any container launched from the image.

```
ENV WEBAPP_PORT 8080
```

This will result in an environment variable being created in any containers created from this image of

```
WEBAPP_PORT=8080
```

You can also specify environment variables when you use `docker run`.

```
$ docker run -e WEBAPP_PORT=8000 -e WEBAPP_HOST=www.example.com ...
```

The `USER` instruction

The `USER` instruction sets the user name or UID to use when running the image.

It can be used multiple times to change back to root or to another user.

The CMD instruction

The `CMD` instruction is a default command run when a container is launched from the image.

```
CMD [ "nginx", "-g", "daemon off;" ]
```

Means we don't need to specify `nginx -g "daemon off;"` when running the container.

Instead of:

```
$ docker run <dockerhubUsername>/web_image nginx -g "daemon off;"
```

We can just do:

```
$ docker run <dockerhubUsername>/web_image
```

More about the CMD instruction

Just like RUN, the CMD instruction comes in two forms. The first executes in a shell:

```
CMD nginx -g "daemon off;"
```

The second executes directly, without shell processing:

```
CMD [ "nginx", "-g", "daemon off;" ]
```

Overriding the CMD instruction

The CMD can be overridden when you run a container.

```
$ docker run -it <dockerhubUsername>/web_image bash
```

Will run `bash` instead of `nginx -g "daemon off;"`.

The ENTRYPOINT instruction

The `ENTRYPOINT` instruction is like the `CMD` instruction, but arguments given on the command line are *appended* to the entry point.

Note: you have to use the "exec" syntax (`["..."]`).

```
ENTRYPOINT [ "/bin/ls" ]
```

If we were to run:

```
$ docker run training/ls -l
```

Instead of trying to run `-l`, the container will run `/bin/ls -l`.

Overriding the ENTRYPOINT instruction

The entry point can be overridden as well.

```
$ docker run -it training/ls
bin  dev  home  lib64  mnt  proc  run   srv  tmp  var
boot etc  lib   media  opt  root  sbin  sys  usr
$ docker run -it --entrypoint bash training/ls
root@d902fb7b1fc7:/#
```

How CMD and ENTRYPOINT interact

The CMD and ENTRYPOINT instructions work best when used together.

```
ENTRYPOINT [ "nginx" ]  
CMD [ "-g", "daemon off;" ]
```

The ENTRYPOINT specifies the command to be run and the CMD specifies its options. On the command line we can then potentially override the options when needed.

```
$ docker run -d <dockerhubUsername>/web_image -t
```

This will override the options CMD provided with new flags.

Advanced Dockerfile instructions

- **ONBUILD** lets you stash instructions that will be executed when this image is used as a base for another one.
- **LABEL** adds arbitrary metadata to the image.
- **ARG** defines build-time variables (optional or mandatory).
- **STOPSIGNAL** sets the signal for `docker stop` (`TERM` by default).
- **HEALTHCHECK** defines a command assessing the status of the container.
- **SHELL** sets the default program to use for string-syntax `RUN`, `CMD`, etc.

The ONBUILD instruction

The `ONBUILD` instruction is a trigger. It sets instructions that will be executed when another image is built from the image being build.

This is useful for building images which will be used as a base to build other images.

```
ONBUILD COPY . /src
```

- You can't chain `ONBUILD` instructions with `ONBUILD`.
- `ONBUILD` can't be used to trigger `FROM` and `MAINTAINER` instructions.

Building an efficient Dockerfile

- Each line in a `Dockerfile` creates a new layer.
- Build your `Dockerfile` to take advantage of Docker's caching system.
- Combine multiple similar commands into one by using `&&` to continue commands and `\` to wrap lines.
- `COPY` dependency lists (`package.json`, `requirements.txt`, etc.) by themselves to avoid reinstalling unchanged dependencies every time.

Example "bad" Dockerfile

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

```
FROM python
MAINTAINER Docker Education Team <education@docker.com>
COPY . /src/
WORKDIR /src
RUN pip install -qr requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```

Fixed Dockerfile

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

```
FROM python
MAINTAINER Docker Education Team <education@docker.com>
COPY ./requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
COPY . /src/
WORKDIR /src
EXPOSE 5000
CMD ["python", "app.py"]
```

A quick word about the Docker Hub

Lesson 13: Uploading our images to the Docker Hub

We have built our first images.

If we were so inclined, we could share those images through the Docker Hub.

We won't do it since we don't want to force everyone to create a Docker Hub account (although it's free, yay!) but the steps would be:

- have an account on the Docker Hub
- tag our image accordingly (i.e. `username/imagename`)
- `docker push username/imagename`

Anybody can now `docker run username/imagename` from any Docker host.

Images can be set to be private as well.

Naming and inspecting containers



Lesson 14: Naming and inspecting containers

Objectives

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

Naming allows us to:

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

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

Naming our containers

So far, we have referenced containers with their ID.

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

But each container can also be referenced by its name.

If a container is named `prod-db`, I can do:

```
$ docker logs prod-db  
$ docker stop prod-db  
etc.
```

Default names

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

It will be the concatenation of:

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

Examples: `happy_curie`, `clever_hopper`, `jovial_lovelace` ...

Specifying a name

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

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

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

This lets us enforce unicity of a given resource.

Renaming containers

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

Inspecting a container

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

```
$ docker inspect <containerID>
[{"AppArmorProfile": "",
  "Args": [],
  "Config": {
    "AttachStderr": true,
    "AttachStdin": false,
    "AttachStdout": true,
    "Cmd": [
      "bash"
    ],
    "CpuShares": 0,
    ...
```

There are multiple ways to consume that information.

Parsing JSON with the Shell

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

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

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

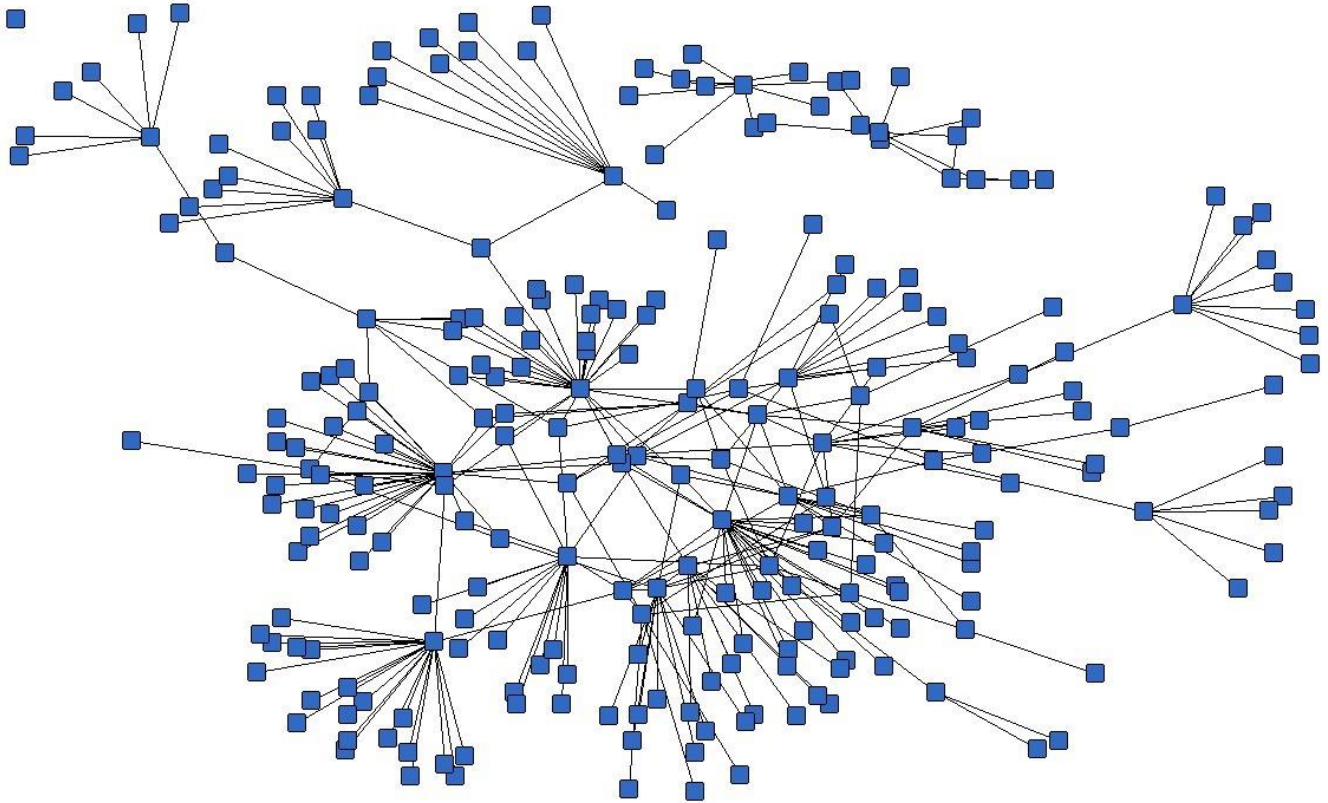
Using `--format`

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

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

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

Container Networking Basics



Lesson 15: Container Networking Basics

Objectives

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

At the end of this lesson, 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    0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp
```

- The web server is running on ports 80 and 443 inside the container.
- Those ports are mapped to ports 32769 and 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:32769
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```


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
32769
```

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 two 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 }}' <yourContainerID>  
172.17.0.3
```

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

Pinging our container

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

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

The different network drivers

A container can use one of the following drivers:

- `bridge` (default)
- `none`
- `host`
- `container`

The driver is selected with `docker run --net`

The default bridge

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

The null driver

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

The host driver

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

Use cases:

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

The container driver

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

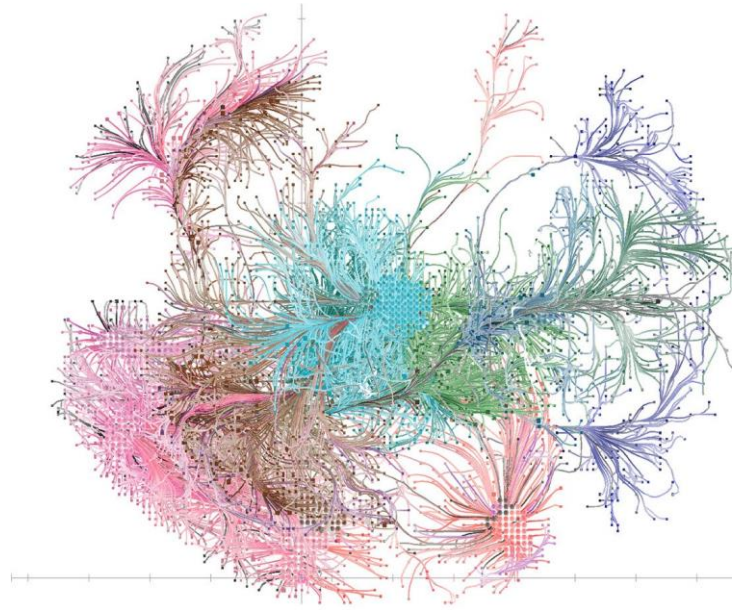
Section summary

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.

The Container Network Model



Lesson 16: The Container Network Model

Objectives

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

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

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

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

The Container Network Model

The CNM was introduced in Engine 1.9.0 (November 2015).

The CNM adds the notion of a *network*, and a new top-level command to manipulate and see those networks: `docker network`.

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

What's in a network?

- Conceptually, a network is a virtual switch.
- It can be local (to a single Engine) or global (across multiple hosts).
- A network has an IP subnet associated to it.
- A network is managed by a *driver*.
- A network can have a custom IPAM (IP allocator).
- Containers with explicit names are discoverable via DNS.
- All the drivers that we have seen before are available.
- A new multi-host driver, *overlay*, is available out of the box.
- More drivers can be provided by plugins (OVS, VLAN...)

Creating a network

Let's create a network called `dev`.

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

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

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

Placing containers on a network

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

It will be reachable with its name, `search`.

```
$ docker run -d --name search --net dev elasticsearch  
8abb80e229ce8926c7223beb69699f5f34d6f1d438bfc5682db893e798046863
```

Communication between containers

Now, create another container on this network.

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

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

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

Resolving container addresses

In Docker Engine 1.9, name resolution is implemented with `/etc/hosts`, and updating it each time containers are added/removed.

```
[root@0eccccdfa45ef /]# cat /etc/hosts
172.18.0.3    0eccccdfa45ef
127.0.0.1     localhost
::1          localhost ip6-localhost ip6-loopback
fe00::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters
172.18.0.2    search
172.18.0.2    search.dev
```

In Docker Engine 1.10, this has been replaced by a dynamic resolver.

(This avoids race conditions when updating `/etc/hosts`.)

Connecting multiple containers together

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

Running the web server

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

Start the container, exposing all its ports:

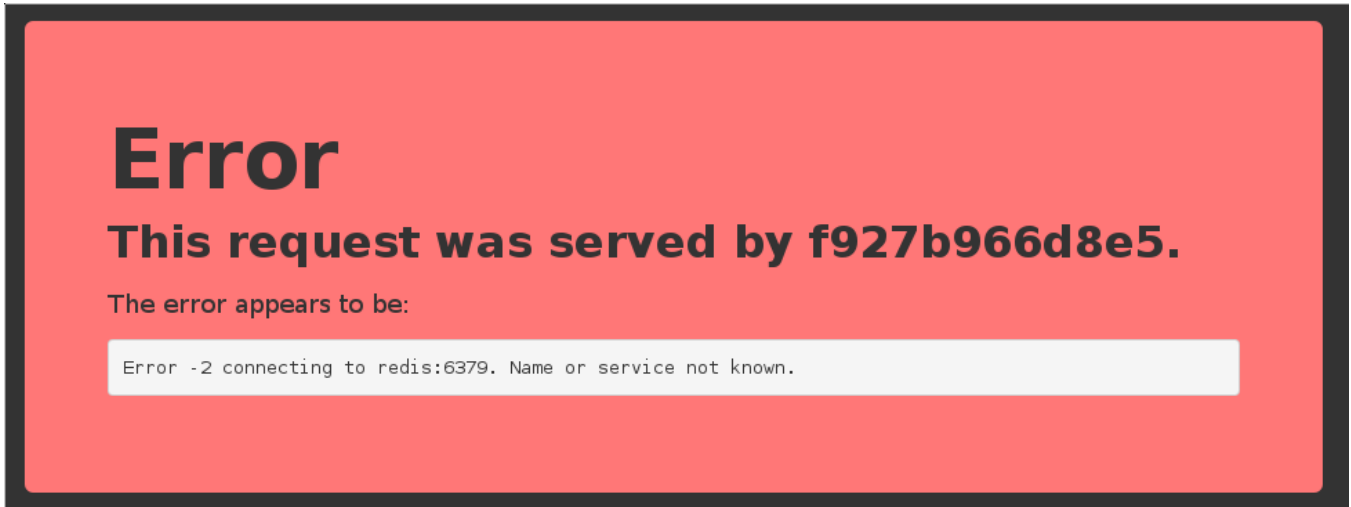
```
$ docker run --net dev -d -P jpetazzo/trainingwheels
```

Check the port that has been allocated to it:

```
$ docker ps -l
```

Test the web server

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



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

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

Start the data store

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

Start the container:

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


Test the web server again

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

Training wheels

**This request was served by f927b966d8e5.
f927b966d8e5 served 1 request so far.**

The current ladder is:

- f927b966d8e5 → 1 request

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

A few words on *scope*

- What if we want to run multiple copies of our application?
- Since names are unique, there can be only one container named `redis` at a time.
- We can specify `--net-alias` to define network-scoped aliases, independently of the container name.

Let's remove the `redis` container:

```
$ docker rm -f redis
```

And create one that doesn't block the `redis` name:

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

Check that the app still works (but the counter is back to 1, since we wiped out the old Redis container).

Names are *local* to each network

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

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

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

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

Network aliases

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

We will use *network aliases*.

A container can have multiple network aliases.

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

Multiple containers can have the same network alias (even on the same network). In Docker Engine 1.11, resolving a network alias yields the IP addresses of all containers holding this alias.

Creating containers on another network

Create the `prod` network.

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

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

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

Resolving network aliases

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

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

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

Connecting to aliased containers

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

Try the following command a few times:

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

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

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

Good to know ...

- Docker will not create network names and aliases on the default `bridge` network.
- Therefore, if you want to use those features, you have to create a custom network first.
- Network aliases are *not* unique: you can give multiple containers the same alias *on the same network*.
 - In Engine 1.10: one container will be selected and only its IP address will be returned when resolving the network alias.
 - In Engine 1.11: when resolving the network alias, the DNS reply includes the IP addresses of all containers with this network alias. This allows crude load balancing across multiple containers (but is not a substitute for a real load balancer).
 - In Engine 1.12: enabling *Swarm Mode* gives access to clustering features, including an advanced load balancer using Linux IPVS.
- Creation of networks and network aliases is generally automated with tools like Compose (covered in a few chapters).

A few words about round robin DNS

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

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

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

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

Custom networks

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

Setting containers' IP address

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

A full example would look like this.

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

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

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

Overlay networks

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

Multi-host networking (overlay)

Out of the scope for this intro-level workshop!

Very short instructions:

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

See <http://jpetazzo.github.io/orchestration-workshop> for all the deets about clustering!

Multi-host networking (plugins)

Out of the scope for this intro-level workshop!

General idea:

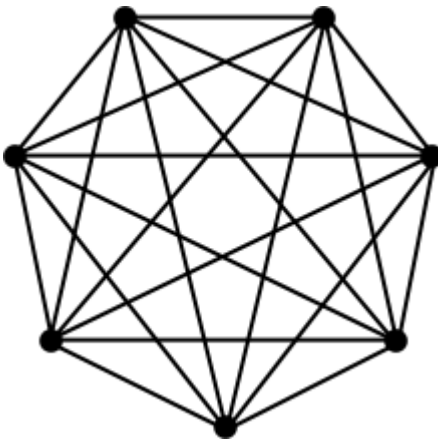
- install the plugin (they often ship within containers)
- run the plugin (if it's in a container, it will often require extra parameters; don't just `docker run` it blindly!)
- some plugins require configuration or activation (creating a special file that tells Docker "use the plugin whose control socket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

Section summary

We've learned how to:

- Create private networks for groups of containers.
- Assign IP addresses to containers.
- Use container naming to implement service discovery.

Connecting Containers With Links



Lesson 17: Connecting containers with links

Objectives

Links were the "legacy" way of connecting containers (before the implementation of the CNM).

They are still useful in some scenarios.

How *links* work

- Links are created *between two containers*
- Links are created *from the client to the server*
- Links associate an arbitrary name to an existing container
- Links exist *only in the context of the client*

The plan

- We will create the `redis` container first.
- Then, we will create the `www` container, *with a link to the previous container*.
- We don't need to use a custom network for this to work.

Create the `redis` container

Let's launch a container from the `redis` image.

```
$ docker run -d --name datastore redis  
<yourContainerID>
```

Let's check the container is running:

```
$ docker ps -l  
CONTAINER ID    IMAGE           COMMAND          ...   PORTS           NAMES  
9efd72a4f320    redis:latest    redis-server     ...   6379/tcp        datastore
```

- Our container is launched and running an instance of Redis.
- We used the `--name` flag to reference our container easily later.
- We could have used *any name we wanted*.

Create the `WWW` container

If we create the web container without any extra option, it will not be able to connect to redis.

```
$ docker run -dP jpetazzo/trainingwheels
```

Check the port number with `docker ps`, and connect to it.

We get the same red error page as before.

How our app connects to Redis

Remember, in the code, we connect to the name `redis`:

```
redis = redis.Redis("redis")
```

- This means "try to connect to 'redis'".
- Not 192.168.123.234.
- Not redis.prod.mycompany.net.

Obviously it doesn't work.

Creating a linked container

Docker allows to specify *links*.

Links indicate an intent: "this container will connect to this other container."

Here is how to create our first link:

```
$ docker run -ti --link datastore:redis alpine sh
```

In this container, we can communicate with `datastore` using the `redis` DNS alias.

DNS

Docker has created a DNS entry for the container, resolving to its internal IP address.

```
$ docker run -it --link datastore:redis alpine ping redis
PING redis (172.17.0.29): 56 data bytes
64 bytes from 172.17.0.29: icmp_seq=0 ttl=64 time=0.164 ms
64 bytes from 172.17.0.29: icmp_seq=1 ttl=64 time=0.122 ms
64 bytes from 172.17.0.29: icmp_seq=2 ttl=64 time=0.086 ms
^C--- redis ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.086/0.124/0.164/0.032 ms
```

- The `--link` flag connects one container to another.
- We specify the name of the container to link to, `datastore`, and an alias for the link, `redis`, in the format `name:alias`.

Starting our application

Now that we've poked around a bit let's start the application itself in a fresh container:

```
$ docker run -d -P --link datastore:redis jpetazzo/trainingwheels
```

Now let's check the port number associated to the container.

```
$ docker ps -l
```

Confirming that our application works properly

Finally, let's browse to our application and confirm it's working.

Links and environment variables

In addition to the DNS information, Docker will automatically set environment variables in our container, giving extra details about the linked container.

```
$ docker run --link datastore:redis alpine env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=0738e57b771e
REDIS_PORT=tcp://172.17.0.120:6379
REDIS_PORT_6379_TCP=tcp://172.17.0.120:6379
REDIS_PORT_6379_TCP_ADDR=172.17.0.120
REDIS_PORT_6379_TCP_PORT=6379
REDIS_PORT_6379_TCP_PROTO=tcp
REDIS_NAME=/dreamy_wilson/redis
REDIS_ENV_REDIS_VERSION=2.8.13
REDIS_ENV_REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-2.8.13.tar.gz
REDIS_ENV_REDIS_DOWNLOAD_SHA1=a72925a35849eb2d38a1ea076a3db82072d4ee43
HOME=/
RUBY_MAJOR=2.1
RUBY_VERSION=2.1.2
```

- Each variables is prefixed with the link alias: `redis`.
- Includes connection information PLUS any environment variables set in the `datastore` container via `ENV` instructions.

Differences between network aliases and links

- With network aliases, you can start containers in *any order*.
- With links, you have to start the server (in our example: Redis) first.
- With network aliases, you cannot change the name of the server once it is running. If you want to add a name, you have to create a new container.
- With links, you can give new names to an existing container.
- Network aliases require the use of a custom network.
- Links can be used on the default bridge network.
- Network aliases work across multi-host networking.
- Links (as of Engine 1.11) only work with local containers (but this might be changed in the future).
- Network aliases don't populate environment variables.
- Links give access to the environment of the target container.

Section summary

We've learned how to:

- Create links between containers.
- Use names and links to communicate across containers.

Ambassadors



Lesson 18: Ambassadors

Objectives

At the end of this lesson, you will be able to understand the ambassador pattern.

Ambassadors abstract the connection details for your services:

- discovery (where is my service actually running?)
- migration (what if my service has to be moved while I use it?)
- fail over (what if my service has a replication system, and I need to connect to the right instance?)
- load balancing (what if there are multiple instances of my service?)
- authentication (what if my service requires credentials, certificates, or otherwise?)

Ambassadors

We've already seen a couple of ways we can manage our application architecture in Docker.

- With network aliases.
- With links.

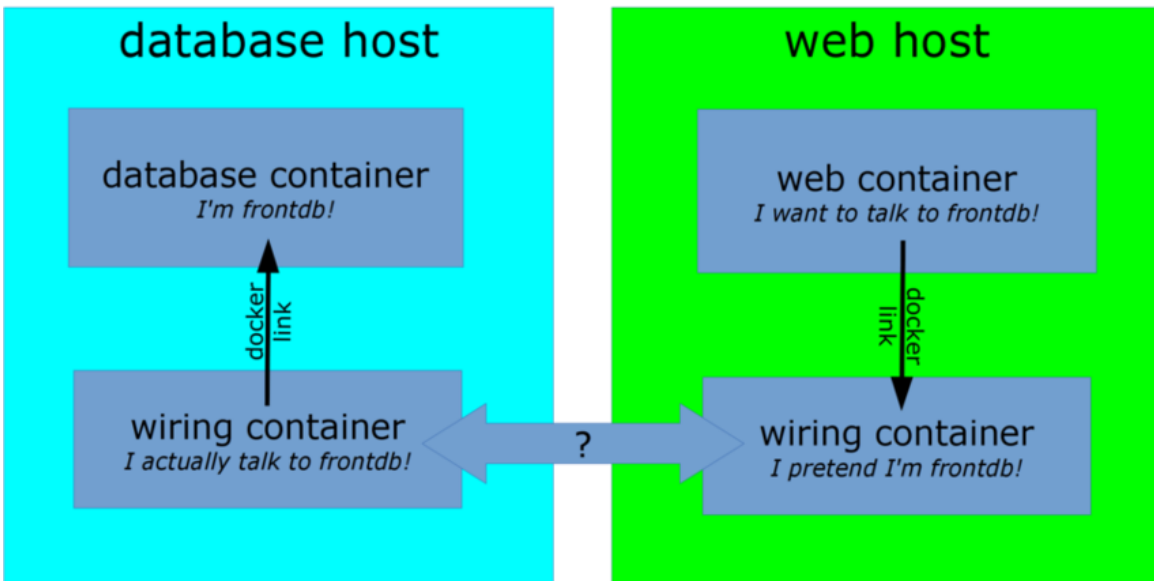
We're now going to see a pattern for service portability we call: ambassadors.

Introduction to Ambassadors

The ambassador pattern:

- Takes advantage of Docker's per-container naming system and abstracts connections between services.
- Allows you to manage services without hard-coding connection information inside applications.

To do this, instead of directly connecting containers you insert ambassador containers.



Interacting with ambassadors

- The web application container uses normal Docker networking to connect to the ambassador.
- The database container also talks with an ambassador.
- For both containers, there is no difference between normal operation and operation with ambassador containers.
- If the database container is moved (or a failover happens), its new location will be tracked by the ambassador containers, and the web application container will still be able to connect, without reconfiguration.

Ambassadors for simple service discovery

Use case:

- my application code connects to `redis` on the default port (6379),
- my Redis service runs on another machine, on a non-default port (e.g. 12345),
- I want to use an ambassador to let my application connect without modification.

The ambassador will be:

- a container running right next to my application,
- using the name `redis` (or linked as `redis`),
- listening on port 6379,
- forwarding connections to the actual Redis service.

Ambassadors for service migration

Use case:

- my application code still connects to `redis`,
- my Redis service runs somewhere else,
- my Redis service is moved to a different host+port,
- the location of the Redis service is given to me via e.g. DNS SRV records,
- I want to use an ambassador to automatically connect to the new location, with as little disruption as possible.

The ambassador will be:

- the same kind of container as before,
- running an additional routine to monitor DNS SRV records,
- updating the forwarding destination when the DNS SRV records are updated.

Ambassadors for credentials injection

Use case:

- my application code still connects to `redis`,
- my application code doesn't provide Redis credentials,
- my production Redis service requires credentials,
- my staging Redis service requires different credentials,
- I want to use an ambassador to abstract those credentials.

The ambassador will be:

- a container using the name `redis` (or a link),
- passed the credentials to use,
- running a custom proxy that accepts connections on Redis default port,
- performing authentication with the target Redis service before forwarding traffic.

Ambassadors for load balancing

Use case:

- my application code connects to a web service called `api`,
- I want to run multiple instances of the `api` backend,
- those instances will be on different machines and ports,
- I want to use an ambassador to abstract those details.

The ambassador will be:

- a container using the name `api` (or a link),
- passed the list of backends to use (statically or dynamically),
- running a load balancer (e.g. HAProxy or NGINX),
- dispatching requests across all backends transparently.

"Ambassador" is a *pattern*

There are many ways to implement the pattern.

Different deployments will use different underlying technologies.

- On-premise deployments with a trusted network can track container locations in e.g. Zookeeper, and generate HAproxy configurations each time a location key changes.
- Public cloud deployments or deployments across unsafe networks can add TLS encryption.
- Ad-hoc deployments can use a master-less discovery protocol like avahi to register and discover services.
- It is also possible to do one-shot reconfiguration of the ambassadors. It is slightly less dynamic but has much less requirements.
- Ambassadors can be used in addition to, or instead of, overlay networks.

Section summary

We've learned how to:

- Understand the ambassador pattern and what it is used for (service portability).

For more information about the ambassador pattern, including demos on Swarm and ECS:

- AWS re:invent 2015 [DVO317](#)
- [SwarmWeek video about Swarm+Compose](#)

Local Development Workflow with Docker



Lesson 19: Local Development Workflow with Docker

Objectives

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

- Share code between container and host.
- Use a simple local development workflow.

Using a Docker container for local development

Never again:

- "Works on my machine"
- "Not the same version"
- "Missing dependency"

By using Docker containers, we will get a consistent development environment.

Our "namer" application

- The code is available on <https://github.com/jpetazzo/namer>.
- The image jpetazzo/namer is automatically built by the Docker Hub.

Let's run it with:

```
$ docker run -dP jpetazzo/namer
```

Check the port number with `docker ps` and open the application.

Let's look at the code

Let's download our application's source code.

```
$ git clone https://github.com/jpetazzo/namer
$ cd namer
$ ls -l
company_name_generator.rb
config.ru
docker-compose.yml
Dockerfile
Gemfile
```

Where's my code?

According to the Dockerfile, the code is copied into `/src` :

```
FROM ruby
MAINTAINER Education Team at Docker <education@docker.com>

COPY . /src
WORKDIR /src
RUN bundler install

CMD ["rackup", "--host", "0.0.0.0"]
EXPOSE 9292
```

We want to make changes *inside the container* without rebuilding it each time.

For that, we will use a *volume*.

Our first volume

We will tell Docker to map the current directory to `/src` in the container.

```
$ docker run -d -v $(pwd):/src -p 80:9292 jpetazzo/namer
```

- The `-d` flag indicates that the container should run in detached mode (in the background).
- The `-v` flag provides volume mounting inside containers.
- The `-p` flag maps port `9292` inside the container to port `80` on the host.
- `jpetazzo/namer` is the name of the image we will run.
- We don't need to give a command to run because the Dockerfile already specifies `rackup`.

Mounting volumes inside containers

The `-v` flag mounts a directory from your host into your Docker container. The flag structure is:

```
[host-path]:[container-path]:[rw|ro]
```

- If `[host-path]` or `[container-path]` doesn't exist it is created.
- You can control the write status of the volume with the `ro` and `rw` options.
- If you don't specify `rw` or `ro`, it will be `rw` by default.

There will be a full chapter about volumes!

Testing the development container

Now let us see if our new container is running.

```
$ docker ps
```

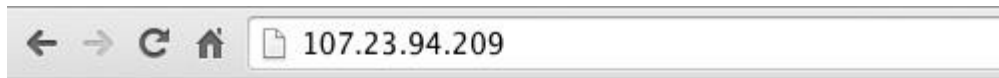
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
045885b68bc5	tra...	rackup	3 seconds ago	Up ...	0.0.0.0:80->9292/tcp	...

Viewing our application

Now let's browse to our web application on:

`http://<yourHostIP>:80`

We can see our company naming application.



Jast-Schiller
unleash customized web-readiness

Making a change to our application

Our customer really doesn't like the color of our text. Let's change it.

```
$ vi company_name_generator.rb
```

And change

```
color: royalblue;
```

To:

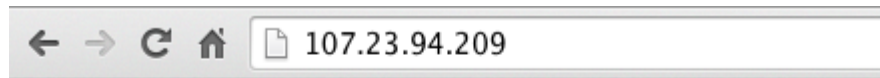
```
color: red;
```

Refreshing our application

Now let's refresh our browser:

`http://<yourHostIP>:80`

We can see the updated color of our company naming application.



Hansen-Koch
streamline B2B infomediaries

Improving the workflow with Compose

- You can also start the container with the following command:

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

- This works thanks to the Compose file, `docker-compose.yml`:

```
www:
  build: .
  volumes:
    - ./src
  ports:
    - 80:9292
```

Why Compose?

- Specifying all those "docker run" parameters is tedious.
- And error-prone.
- We can "encode" those parameters in a "Compose file."
- When you see a `docker-compose.yml` file, you know that you can use `docker-compose up`.
- Compose can also deal with complex, multi-container apps.
(More on this later.)

Workflow explained

We can see a simple workflow:

1. Build an image containing our development environment.
(Rails, Django...)
2. Start a container from that image.
Use the `-v` flag to mount source code inside the container.
3. Edit source code outside the containers, using regular tools.
(vim, emacs, textmate...)
4. Test application.
(Some frameworks pick up changes automatically.
Others require you to Ctrl-C + restart after each modification.)
5. Repeat last two steps until satisfied.
6. When done, commit+push source code changes.
(You *are* using version control, right?)

Debugging inside the container

In 1.3, Docker introduced a feature called `docker exec`.

It allows users to run a new process in a container which is already running.

If sometimes you find yourself wishing you could SSH into a container: you can use `docker exec` instead.

You can get a shell prompt inside an existing container this way, or run an arbitrary process for automation.

docker exec example

```
$ # You can run ruby commands in the area the app is running and more!
$ docker exec -it <yourContainerId> bash
root@5ca27cf74c2e:/opt/namer# irb
irb(main):001:0> [0, 1, 2, 3, 4].map {|x| x ** 2}.compact
=> [0, 1, 4, 9, 16]
irb(main):002:0> exit
```

Stopping the container

Now that we're done let's stop our container.

```
$ docker stop <yourContainerID>
```

And remove it.

```
$ docker rm <yourContainerID>
```

Section summary

We've learned how to:

- Share code between container and host.
- Set our working directory.
- Use a simple local development workflow.

Working with Volumes



Lesson 20: Working with Volumes

Objectives

At the end of this lesson, 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.

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).

Volumes can be shared across containers

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

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

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

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

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

```
$ docker run -it --name alpha -v /var/log ubuntu bash
root@99020f87e695:/# date >/var/log/now
```

In another terminal, let's start another container with the same volume.

```
$ docker run --volumes-from alpha ubuntu cat /var/log/now
Fri May 30 05:06:27 UTC 2014
```

Volumes exist independently of containers

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

Since Docker 1.9, we can see all existing volumes and manipulate them:

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

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

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

Data containers (before Engine 1.9)

A *data container* is a container created for the sole purpose of referencing one (or many) volumes.

It is typically created with a no-op command:

```
$ docker run --name files -v /var/www busybox true
$ docker run --name logs -v /var/log busybox true
```

- We created two data containers.
- They are using the `busybox` image, a tiny image.
- We used the command `true`, possibly the simplest command in the world!
- We named each container to reference them easily later.

Using data containers

Data containers are used by other containers thanks to `--volumes-from`.

Consider the following (fictitious) example, using the previously created volumes:

```
$ docker run -d --volumes-from files --volumes-from logs webserver
$ docker run -d --volumes-from files ftpserver
$ docker run -d --volumes-from logs lumberjack
```

- The first container runs a webserver, serving content from `/var/www` and logging to `/var/log`.
- The second container runs a FTP server, allowing to upload content to the same `/var/www` path.
- The third container collects the logs, and sends them to logstash, a log storage and analysis system, using the lumberjack protocol.

Named volumes (since Engine 1.9)

- We can now create and manipulate volumes as first-class concepts.
- Volumes can be created without a container, then used in multiple containers.

Let's create a volume directly.

```
$ docker volume create --name=website  
website
```

Volumes are not anchored to a specific path.

Using our named volumes

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

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

```
$ docker run -d -p 8888:80 \  
  -v website:/usr/share/nginx/html \  
  -v logs:/var/log/nginx \  
  nginx
```

Check that it's running correctly:

```
$ curl localhost:8888  
<!DOCTYPE html>  
...  
<h1>Welcome to nginx!</h1>  
...
```

Using a volume in another container

- We will make changes to the volume from another container.
- In this example, we will run a text editor in the other container, but this could be a FTP server, a WebDAV server, a Git receiver...

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

```
$ docker run -v website:/website -w /website -ti alpine vi index.html
```

Make changes, save, and exit.

Then run `curl localhost:8888` again to see your changes.

Managing volumes explicitly

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

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

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

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

Sharing a directory between the host and a container

The previous example would become something like this:

```
$ mkdir -p /mnt/files /mnt/logs
$ docker run -d -v /mnt/files:/var/www -v /mnt/logs:/var/log webserver
$ docker run -d -v /mnt/files:/home/ftp ftpserver
$ docker run -d -v /mnt/logs:/var/log lumberjack
```

Note that the paths must be absolute.

Those volumes can also be shared with `--volumes-from`.

Migrating data with `--volumes-from`

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

- Scenario: migrating from Redis 2.8 to Redis 3.0.
- We have a container (`myredis`) running Redis 2.8.
- Stop the `myredis` container.
- Start a new container, using the Redis 3.0 image, and the `--volumes-from` option.
- The new container will inherit the data of the old one.
- Newer containers can use `--volumes-from` too.

Data migration in practice

Let's create a Redis container.

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

Connect to the Redis container and set some data.

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

Issue the following commands:

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

Upgrading Redis

Stop the Redis container.

```
$ docker stop redis28
```

Start the new Redis container.

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

Testing the new Redis

Connect to the Redis container and see our data.

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

Issue a few commands.

```
GET counter  
INFO server  
QUIT
```

What happens when you remove containers with volumes?

- With Engine versions prior 1.9, volumes would be *orphaned* when the last container referencing them is destroyed.
- Orphaned volumes are not deleted, but you cannot access them. (Unless you do some serious archaeology in `/var/lib/docker`.)
- Since Engine 1.9, orphaned volumes can be listed with `docker volume ls` and mounted to containers with `-v`.

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

Checking volumes defined by an image

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

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


Checking volumes used by a container

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

```
$ docker inspect <yourContainerID>
[{"ID": "<yourContainerID>",
  "Volumes": {
    "/var/webapp": "/var/lib/docker/vfs/dir/
f4280c5b6207ed531efd4cc673ff620cef2a7980f747dbbcca001db61de04468"
  },
  "VolumesRW": {
    "/var/webapp": true
  },
}]
```

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

Sharing a single file between the host and a container

The same `-v` flag can be used to share a single file.

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

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

Warning: when using such mounts, the container gains root-like access to the host. It can potentially do bad things.

Volume plugins

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

- [dvol](#) - allows to commit/branch/rollback volumes;
- [Flocker](#), [REX-Ray](#) - create and manage volumes backed by an enterprise storage system (e.g. SAN or NAS), or by cloud block stores (e.g. EBS);
- [Blockbridge](#), [Portworx](#) - provide distributed block store for containers;
- and much more!

Section summary

We've learned how to:

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

Compose For Development Stacks



Lesson 21: Using Docker Compose for Development Stacks

Objectives

Dockerfiles are great to build a single container.

But when you want to start a complex stack made of multiple containers, you need a different tool. This tool is Docker Compose.

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

What is Docker Compose?

Docker Compose (formerly known as fig) is an external tool. It is optional (you do not need Compose to run Docker and containers) but we recommend it highly!

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

1. Clone 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 output is shown.

Before diving in, let's see a small example of Compose in action.

Compose in action



Checking if Compose is installed

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

You can always check that it is installed by running:

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

Installing Compose

If you want to install Compose on your machine, there are (at least) two methods.

Compose is written in Python. If you have `pip` and use it to manage other Python packages, you can install compose with:

```
$ sudo pip install docker-compose
```

(Note: if you are familiar with `virtualenv`, you can also use it to install Compose.)

If you do not have `pip`, or do not want to use it to install Compose, you can also retrieve an all-in-one binary file:

```
$ curl -L \
  https://github.com/docker/compose/releases/download/1.8.0/docker-compose-`uname
-s`-`uname -m` \
  > /usr/local/bin/docker-compose
$ chmod +x /usr/local/bin/docker-compose
```

Launching Our First Stack with Compose

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

```
$ cd
$ git clone git://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`.

Training wheels

This request was served by 5457fb09c174.

5457fb09c174 served 1 request so far.

The current ladder is:

- 5457fb09c174 → 1 request

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 versions

Version 1 directly has the various containers (`www`, `redis`...) at the top level of the file.

Version 2 has multiple sections:

- `version` is mandatory and should be "2".
- `services` is mandatory and corresponds to the content of the version 1 format.
- `networks` is optional and can define multiple networks on which containers can be placed.
- `volumes` is optional and can define volumes to be used (and potentially shared) by the containers.

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).

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 <http://docs.docker.com/compose/yml/>.

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 is common to execute the build and run steps in sequence:

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

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->5000/tcp

Cleaning up

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...
```

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

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

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.

Course Conclusion



Course Summary

During this class, we:

- Installed Docker.
- Launched our first container.
- Learned about images.
- Got an understanding about how to manage connectivity and data in Docker containers.
- Learned how to integrate Docker into your daily workflow.

Questions & Next Steps

Still Learning:

- Docker homepage - <http://www.docker.com/>
- Docker Hub - <https://hub.docker.com>
- Docker blog - <http://blog.docker.com/>
- Docker documentation - <http://docs.docker.com/>
- Docker Getting Started Guide - <http://www.docker.com/gettingstarted/>
- Docker code on GitHub - <https://github.com/docker/docker>
- Docker mailing list - <https://groups.google.com/forum/#!forum/docker-user>
- Docker on IRC: irc.freenode.net and channels `#docker` and `#docker-dev`
- Docker on Twitter - <http://twitter.com/docker>
- Get Docker help on Stack Overflow - <http://stackoverflow.com/search?q=docker>

Thank You

