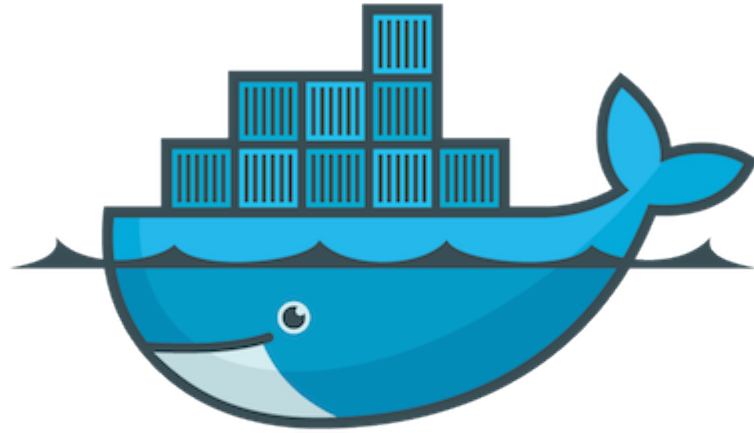




# Introduction to Docker

Version: a2622f1



# docker

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

# Logistics

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

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

# Part 1

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

## Part 2

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

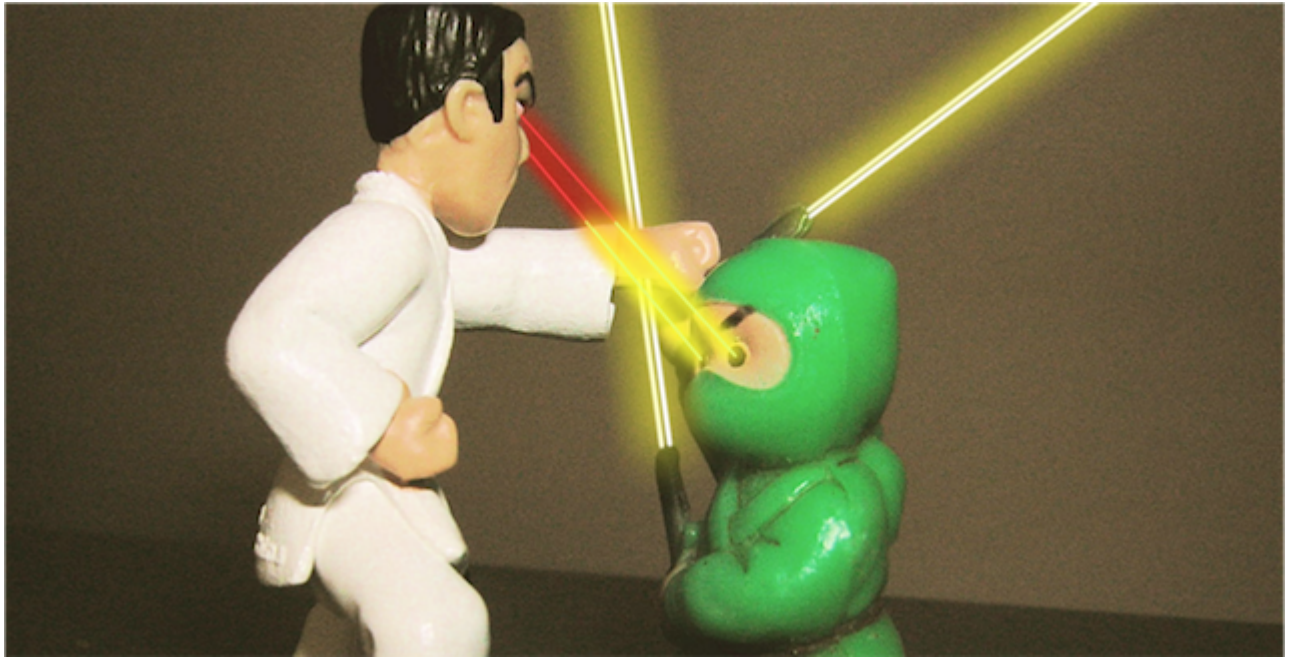
# Extra material

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

# Table of Contents

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

# About Docker



# Lesson 1: Docker 30,000ft overview

## Objectives

In this lesson, we will learn about:

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

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

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



# The origins of the Docker Project

- dotCloud was operating a PaaS, using a custom container engine.
- This engine was based on OpenVZ (and later, LXC) and AUFS.
- It started (circa 2008) as a single Python script.
- By 2012, the engine had multiple (~10) Python components. (and ~100 other micro-services!)
- End of 2012, dotCloud refactors this container engine.
- The codename for this project is "Docker!"

# First public release

- March 2013, PyCon, Santa Clara:  
"Docker" is shown to a public audience for the first time.
- It is released with an open source license.
- Very positive reactions and feedback!
- The dotCloud team progressively shifts to Docker development.
- The same year, dotCloud changes name to Docker.
- In 2014, the PaaS activity is sold.

# The Docker Project

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

# About Docker Inc.

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

## How does Docker Inc. make money?

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

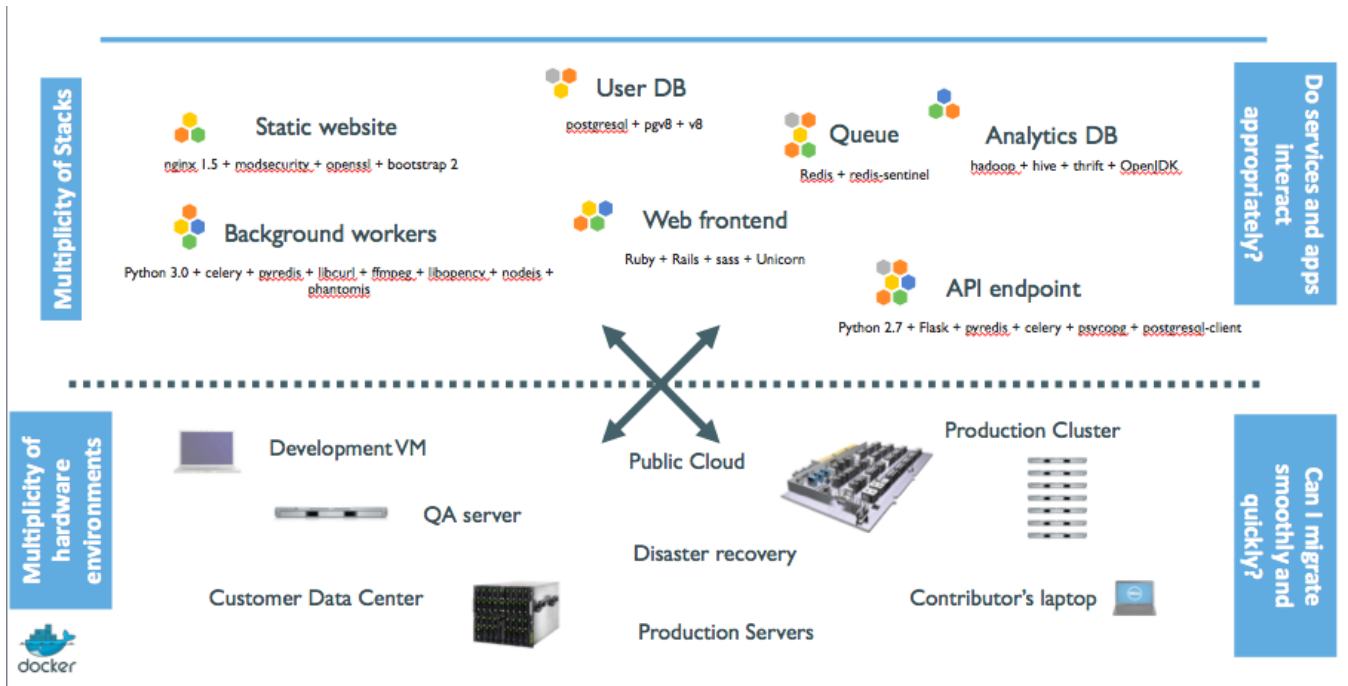
## OK... Why the buzz around containers?

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

# Deployment becomes very complex














- Many different stacks.
- Many different targets.

# The deployment problem



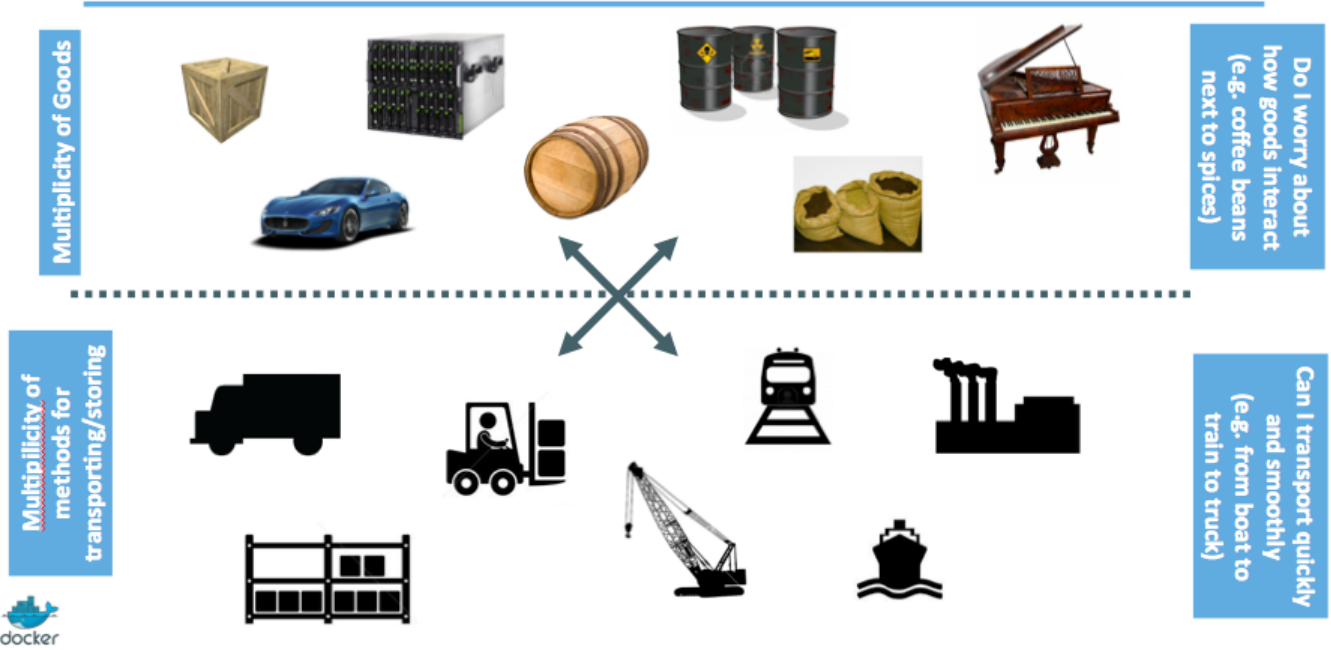


# The Matrix from Hell

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



# An inspiration and some ancient history!



# Intermodal shipping containers



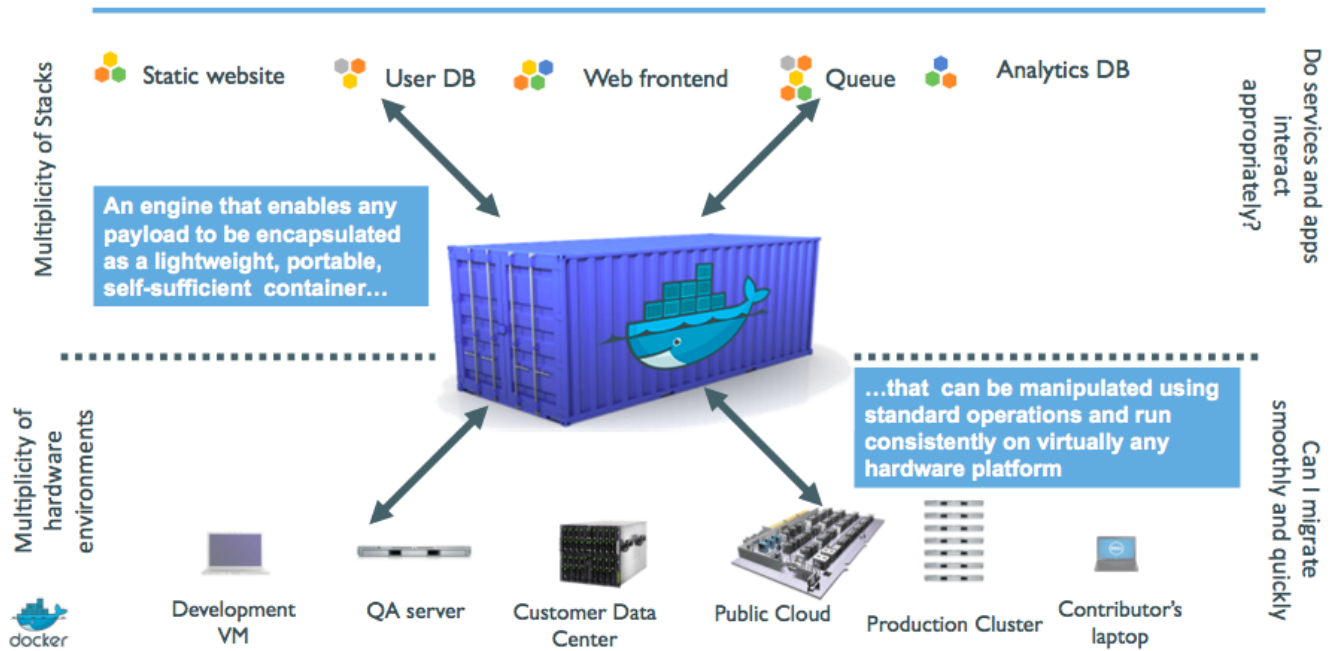
# This spawned a Shipping Container Ecosystem!



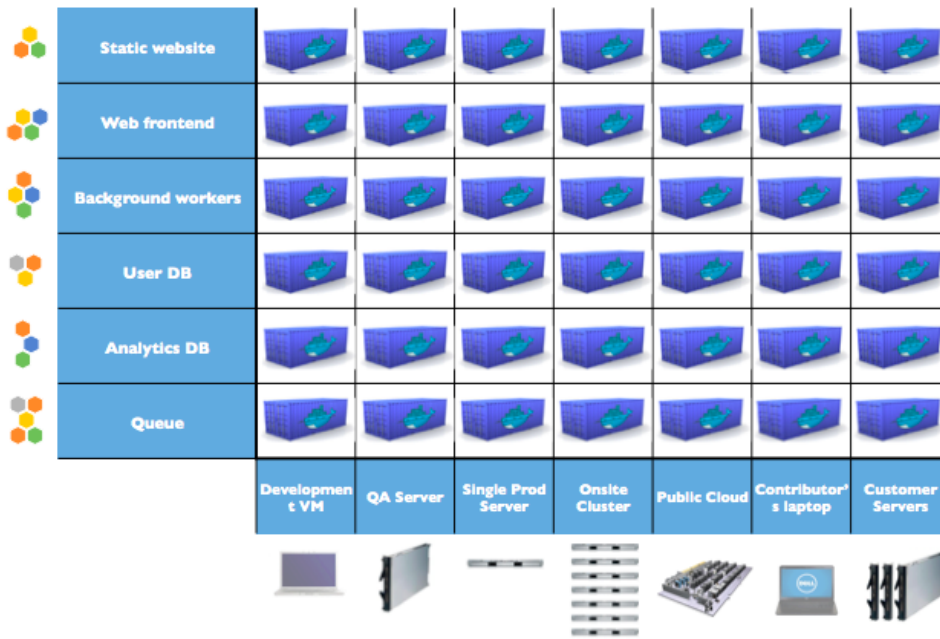
- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)
- massive globalization
- 5000 ships deliver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



# A shipping container system for applications



# Eliminate the matrix from Hell



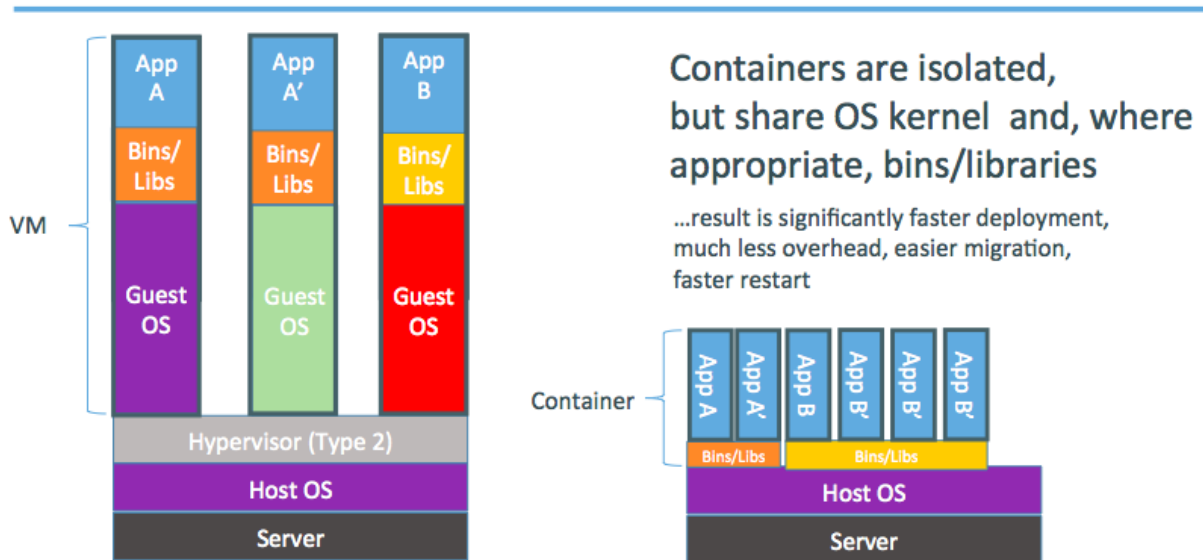
# From lightweight VMs to application containers

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

# Step 1: containers as lightweight VMs



# Less overhead!



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

## Step 2: commoditization of containers

# Containers before Docker

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

## Analogy:

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

## Containers after Docker

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

## Positive feedback loop

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

## Step 3: shipping containers efficiently

# Before Docker

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

# After Docker

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



# Example

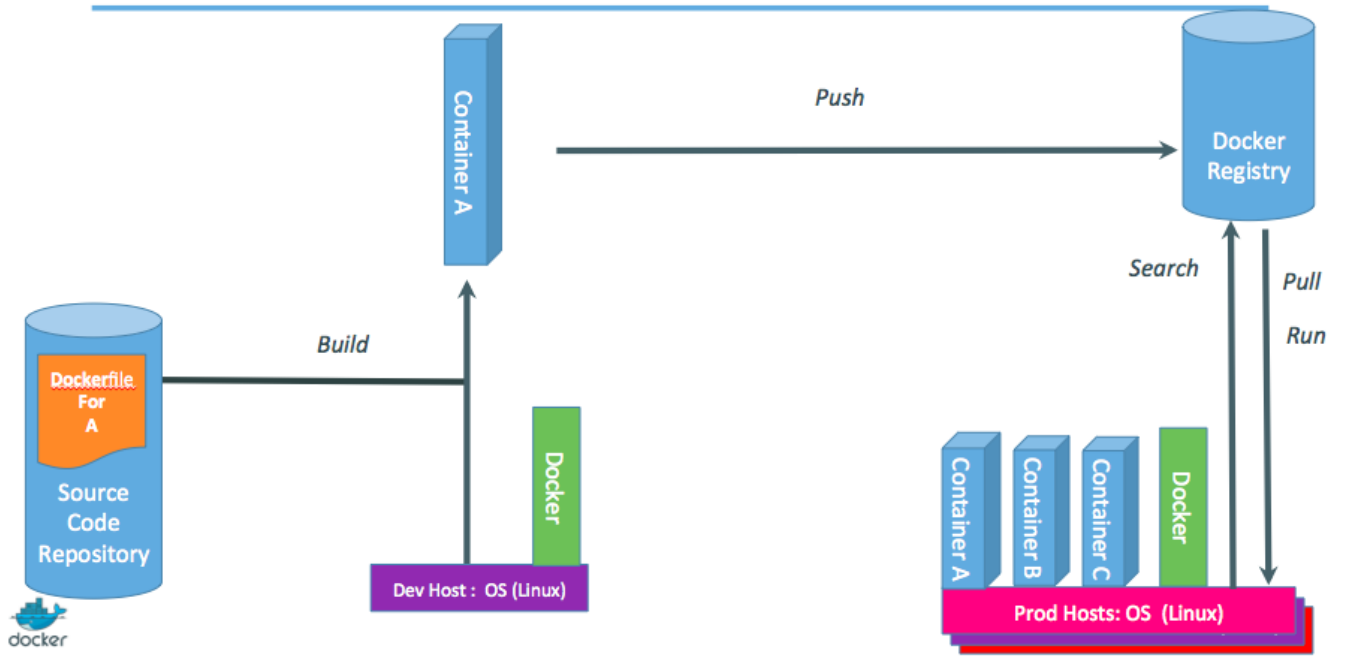
Layers:

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

# Step 4: containers in a modern software factory

# Container image as build artifact

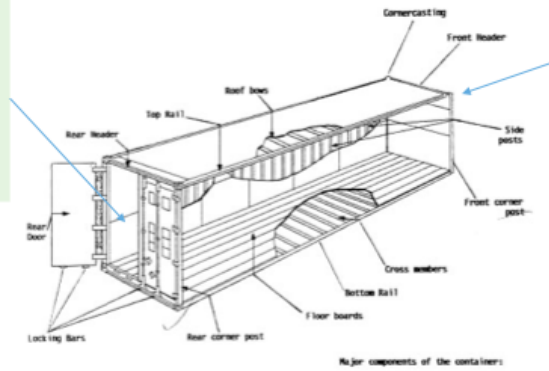
The same container can go from dev, to test, to QA, to prod.



# Technical & cultural revolution: separation of concerns

- Dan the Developer

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



- Oscar the Ops Guy

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



# Your training Virtual Machine



# Lesson 2: Your training Virtual Machine

## Objectives

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

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

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

# Your training Virtual Machine

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

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

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

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

# Connecting to your Virtual Machine

You need an SSH client.

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

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

- On Windows, if you don't have an SSH client, you can download Putty from [www.putty.org](http://www.putty.org).



# Checking your Virtual Machine

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

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

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

# Install Docker



# Lesson 3: Installing Docker

## Objectives

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

- Install Docker.
- Run Docker without sudo.

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

# Installing Docker

Docker is easy to install.

It runs on:

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

# Installing Docker on Linux

It can be installed via:

- Distribution-supplied packages on virtually all distros.

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

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

# Installing Docker on your Linux distribution

On Fedora:

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

On CentOS 7:

```
$ sudo yum install docker
```

On Debian and derivatives:

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

# Installation script from Docker

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

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

This currently works on:

- Ubuntu
- Debian
- Fedora
- Gentoo

# Installing on OS X and Microsoft Windows

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

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

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



# Running Docker on OS X and Windows

When you execute `docker version` from the terminal:

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

Reminder: all communication happens over the API!

## About boot2docker

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

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

Boot2Docker is not a "lite" version of Docker.



# Check that Docker is working

Using the docker client:

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

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

# Su-su-sudo



## Important PSA about security

The `docker` user is `root` equivalent.

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

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

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

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

# The docker group

## Add the Docker group

```
$ sudo groupadd docker
```

## Add ourselves to the group

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

## Restart the Docker daemon

```
$ sudo service docker restart
```

## Log out

```
$ exit
```

## Check that Docker works without sudo

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

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

## Section summary

We've learned how to:

- Install Docker.
- Run Docker without sudo.



# Our First Containers



# Lesson 4: Our First Containers

## Objectives

At the end of this lesson, you will have:

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

# Docker architecture

Docker is a client-server application.

- **The Docker daemon (or "Engine")**  
Receives and processes incoming Docker API requests.
- **The Docker client**  
Talks to the Docker daemon via the Docker API.  
We'll use mostly the CLI embedded within the docker binary.
- **Docker Hub Registry**  
Collection of public images.  
The Docker daemon talks to it via the registry API.

# Hello World

In your Docker environment, just run the following command:

```
$ docker run busybox echo hello world  
hello world
```

## That was our first container!

- We used one of the smallest, simplest images available: busybox.
- busybox is typically used in embedded systems (phones, routers...)
- We ran a single process and echo'ed `hello world`.

## A more useful container

Let's run a more exciting container:

```
$ docker run -it ubuntu bash
root@04c0bb0a6c07:/#
```

- This is a brand new container.
- It runs a bare-bones, no-frills ubuntu system.
- `-it` is shorthand for `-i -t`.
  - `-i` tells Docker to connect us to the container's stdin.
  - `-t` tells Docker that we want a pseudo-terminal.

# Do something in our container

Try to run `figlet` in our container.

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

Alright, we need to install it.

## An observation

Let's check how many packages are installed here.

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

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



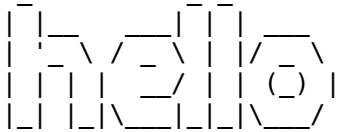
# Install a package in our container

We want `figlet`, so let's install it:

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

One minute later, `figlet` is installed!

```
# figlet hello
```



## Exiting our container

Just exit the shell, like you would usually do.

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

```
root@04c0bb0a6c07:/# exit
```

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

## Starting another container

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

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

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

# Background Containers



# Lesson 5: Background Containers

## Objectives

Our first containers were *interactive*.

We will now see how to:

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

# A non-interactive container

We will run a small custom container.

This container just displays the time every second.

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

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

# Run a container in the background

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

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

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

# List running containers

How can we check that our container is still running?

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

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

Docker tells us:

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



## Two useful flags for docker ps

To see only the last container that was started:

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

To see only the ID of containers:

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

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

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

## View the logs of a container

We told you that Docker was logging the container output.

Let's see that now.

```
$ docker logs 47d6
Fri Feb 20 00:39:52 UTC 2015
Fri Feb 20 00:39:53 UTC 2015
...
```

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

## View only the tail of the logs

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

```
$ docker logs --tail 3 47d6
Fri Feb 20 00:55:35 UTC 2015
Fri Feb 20 00:55:36 UTC 2015
Fri Feb 20 00:55:37 UTC 2015
```

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

## Follow the logs in real time

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

```
$ docker logs --tail 1 --follow 47d6
Fri Feb 20 00:57:12 UTC 2015
Fri Feb 20 00:57:13 UTC 2015
^C
```

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

# Stop our container

There are two ways we can terminate our detached container.

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

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

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

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

# Killing it

Let's kill our container:

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

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

Let's check that our container doesn't show up anymore:

```
$ docker ps
```

# List stopped containers

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

```
$ docker ps -a
CONTAINER ID   IMAGE                                ...   CREATED        STATUS
47d677dcfba4   jpetazzo/clock:latest              ...   23 min. ago    Exited (0) 4 min. ago
5c1dfd4d81f1   jpetazzo/clock:latest              ...   40 min. ago    Exited (0) 40 min. ago
b13c164401fb   ubuntu:latest                       ...   55 min. ago    Exited (130) 53 min. ago
```

---

# Restarting and Attaching to Containers



# Lesson 6: Restarting and Attaching to Containers

## Objectives

We have started containers in the foreground, and in the background.

In this chapter, we will see how to:

- Put a container in the background.
- Attach to a background container to bring it to the foreground.
- Restart a stopped container.

## Background and foreground

The distinction between foreground and background containers is arbitrary.

From Docker's point of view, all containers are the same.

All containers run the same way, whether there is a client attached to them or not.

It is always possible to detach from a container, and to reattach to a container.

## Detaching from a container

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

What does `-i t` stand for?

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

# Attaching to a container

You can attach to a container:

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

- The container must be running.
- There *can* be multiple clients attached to the same container.
- **Warning:** if the container was started without `-it`...
  - You won't be able to detach with `^P^Q`.
  - If you hit `^C`, the signal will be proxied to the container.
- Remember: you can always detach by killing the Docker client.

## Checking container output

- Use `docker attach` if you intend to send input to the container.
- If you just want to see the output of a container, use `docker logs`.

```
$ docker logs --tail 1 --follow  
<containerID>
```

## Restarting a container

When a container has exited, it is in stopped state.

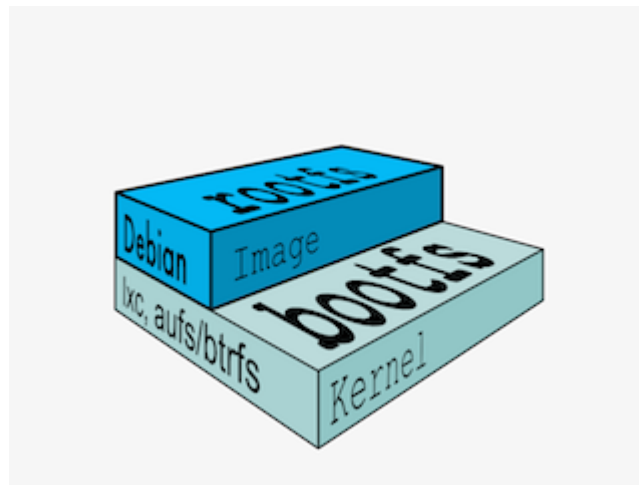
It can then be restarted with the `start` command.

```
$ docker start <yourContainerID>
```

The container will be restarted using the same options you launched it with.

You can re-attach to it if you want to interact with it.

# Understanding Docker Images



# Lesson 7: Understanding Docker Images

## Objectives

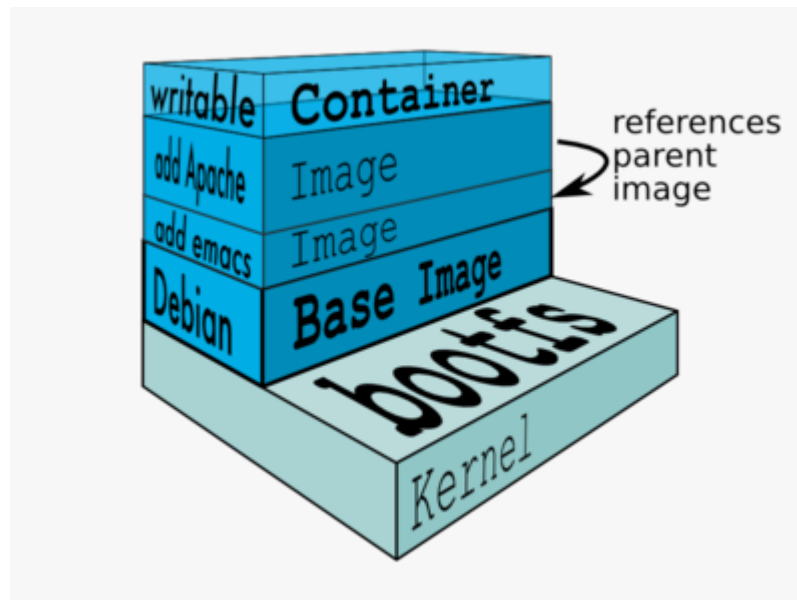
In this lesson, we will explain:

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



# What is an image?

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



# Differences between containers and images

- An image is a read-only filesystem.
- A container is an encapsulated set of processes running in a read-write copy of that filesystem.
- To optimize container boot time, *copy-on-write* is used instead of regular copy.
- `docker run` starts a container from a given image.

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

# Image as stencils

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



# Object-oriented programming

- Images are conceptually similar to *classes*.
- Layers are conceptually similar to *inheritance*.
- Containers are conceptually similar to *instances*.

## Wait a minute...

If an image is read-only, how do we change it?

- We don't.
- We create a new container from that image.
- Then we make changes to that container.
- When we are satisfied with those changes, we transform them into a new layer.
- A new image is created by stacking the new layer on top of the old image.

## A chicken-and-egg problem

- The only way to create an image is by "freezing" a container.
- The only way to create a container is by instantiating an image.
- Help!

## Creating the first images

There is a special empty image called `scratch`.

- It allows to *build from scratch*.

The `docker import` command loads a tarball into Docker.

- The imported tarball becomes a standalone image.
- That new image has a single layer.

Note: you will probably never have to do this yourself.

## Creating other images

`docker commit`

- Saves all the changes made to a container into a new layer.
- Creates a new image (effectively a copy of the container).

`docker build`

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

We will explain both methods in a moment.



# Images namespaces

There are three namespaces:

- Root-like

```
ubuntu
```

- User (and organizations)

```
jpetazzo/clock
```

- Self-Hosted

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

Let's explain each of them.

# Root namespace

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

Those images include:

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

# User namespace

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

For example:

jpetazzo/clock

The Docker Hub user is:

jpetazzo

The image name is:

clock

## Self-Hosted namespace

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

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

For example:

```
localhost:5000/wordpress
```

The remote host and port is:

```
localhost:5000
```

The image name is:

```
wordpress
```

## Historical detail

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

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

# How do you store and manage images?

Images can be stored:

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

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

To be more accurate: you can use the Docker client to tell a Docker server to push and pull images to and from a registry.

## Showing current images

Let's look at what images are on our host now.

```
$ docker images
REPOSITORY          TAG          IMAGE ID      CREATED       VIRTUAL SIZE
ubuntu              13.10       9f676bd305a4 7 weeks ago  178 MB
ubuntu              saucy       9f676bd305a4 7 weeks ago  178 MB
ubuntu              raring     eb601b8965b8 7 weeks ago  166.5 MB
ubuntu              13.04      eb601b8965b8 7 weeks ago  166.5 MB
ubuntu              12.10      5ac751e8d623 7 weeks ago  161 MB
ubuntu              quantal    5ac751e8d623 7 weeks ago  161 MB
ubuntu              10.04      9cc9ea5ea540 7 weeks ago  180.8 MB
ubuntu              lucid      9cc9ea5ea540 7 weeks ago  180.8 MB
ubuntu              12.04      9cd978db300e 7 weeks ago  204.4 MB
ubuntu              latest     9cd978db300e 7 weeks ago  204.4 MB
ubuntu              precise    9cd978db300e 7 weeks ago  204.4 MB
```

# Searching for images

Searches your registry for images:

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

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



# Downloading images

There are two ways to download images.

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

## Pulling an image

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

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

## Image and tags

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

## When to (not) use tags

Don't specify tags:

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

Do specify tags:

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

## Section summary

We've learned how to:

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

# Building Images Interactively



# Lesson 8: Building Images Interactively

## Objectives

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

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

We will:

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

# Building Images Interactively

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

How do we want to construct our own images?

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

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

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



# Building from a base

Our base will be the ubuntu image.

# Create a new container and make some changes

Start an Ubuntu container:

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

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

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

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

# Inspect the changes

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

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

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

# Docker tracks filesystem changes

As explained before:

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

## Commit and run your image

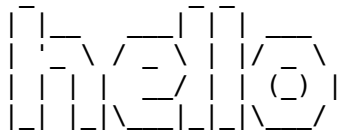
The `docker commit` command will create a new layer with those changes, and a new image using this new layer.

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

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

We can run this image:

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



## Tagging images

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

We can use the `tag` command:

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

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

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

And then run it using its tag:

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

## What's next?

Manual process = bad.

Automated process = good.

In the next chapter, we will learn how to automate the build process by writing a Dockerfile.

# Building Docker images





# Lesson 9: Building Images With A Dockerfile

## Objectives

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

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

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

# Dockerfile overview

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

# Writing our first Dockerfile

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

1. Create a directory to hold our Dockerfile.

```
$ mkdir myimage
```

2. Create a Dockerfile inside this directory.

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

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

## Type this into our Dockerfile...

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

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

# Build it!

Save our file, then execute:

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

- `-t` indicates the tag to apply to the image.
- `.` indicates the location of the *build context*.  
(We will talk more about the build context later; but to keep things simple: this is the directory where our Dockerfile is located.)

# What happens when we build the image?

The output of `docker build` looks like this:

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

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

# Sending the build context to Docker

Sending build context to Docker daemon 2.048 kB

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

## Executing each step

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

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



# The caching system

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

Why?

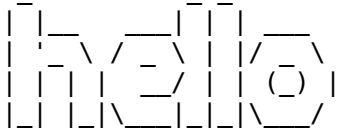
- After each build step, Docker takes a snapshot of the resulting image.
- Before executing a step, Docker checks if it has already built the same sequence.
- Docker uses the exact strings defined in your Dockerfile, so:
  - `RUN apt-get install figlet cowsay` is different from `RUN apt-get install cowsay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

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

## Running the image

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

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



- Sweet is the taste of success!

# Using image and viewing history

The `history` command lists all the layers composing an image.

For each layer, it shows its creation time, size, and creation command.

When an image was built with a Dockerfile, each layer corresponds to a line of the Dockerfile.

```
$ docker history figlet
IMAGE          CREATED          CREATED BY          SIZE
f9e8f1642759  About an hour ago /bin/sh -c apt-get install fi 6.062 MB
7257c37726a1  About an hour ago /bin/sh -c apt-get update     8.549 MB
e54ca5efa2e9   8 months ago    /bin/sh -c apt-get update &&  8 B
6c37f792ddac  8 months ago    /bin/sh -c apt-get update &&  83.43 MB
83ff768040a0  8 months ago    /bin/sh -c sed -i s/^#\s*\(\d 1.903 kB
2f4b4d6a4a06  8 months ago    /bin/sh -c echo #!/bin/sh >  194.5 kB
d7ac5e4f1812  8 months ago    /bin/sh -c #(nop) ADD file:ad 192.5 MB
511136ea3c5a  20 months ago   
```

# CMD and ENTRYPOINT



# Lesson 10: CMD and ENTRYPOINT

## Objectives

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

CMD and ENTRYPOINT.

Those commands allow us to set the default command to run in a container.

# Defining a default command

When people run our container, we want to greet them with a nice hello message, and using a custom font.

For that, we will execute:

```
figlet -f script hello
```

- `-f script` tells `figlet` to use a fancy font.
- `hello` is the message that we want it to display.

# Adding CMD to our Dockerfile

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install figlet
CMD figlet -f script hello
```

- CMD defines a default command to run when none is given.
- It can appear at any point in the file.
- Each CMD will replace and override the previous one.
- As a result, while you can have multiple CMD lines, it is useless.





## 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
```



```
salut
```

We will use the `ENTRYPOINT` verb in Dockerfile.

# Adding ENTRYPOINT to our Dockerfile

Our new Dockerfile will look like this:

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

- ENTRYPOINT defines a base command (and its parameters) for the container.
- The command line arguments are appended to those parameters.
- Like CMD, ENTRYPOINT can appear anywhere, and replaces the previous value.

# Build and test our image

Let's build it:

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

And run it:

```
$ docker run figlet salut
```

```
 /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_
 /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_
 /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_
 /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_  /_
```

Great success!

## Using CMD and ENTRYPOINT together

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

Then we will use ENTRYPOINT and CMD together.

- ENTRYPOINT will define the base command for our container.
- CMD will define the default parameter(s) for this command.

# CMD and ENTRYPOINT together

Our new Dockerfile will look like this:

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

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

# Build and test our image

Let's build it:

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

And run it:

```
$ docker run figlet  
  _  
 / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  
 |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |  
  _ _ _ _ _  
$ docker run figlet hola mundo  
 _  
 / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  / \  
 |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |   | |  
  _ _ _ _ _
```

# Overriding ENTRYPOINT

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

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

We use the `--entrypoint` parameter:

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



## Copying files during the build



# Lesson 11: Copying files during the build

## Objectives

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

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

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

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

## Build some C code

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

Here is the program, `hello.c`:

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

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

Then we will write the Dockerfile.

## The Dockerfile

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

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

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

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

Create this Dockerfile.

## Testing our C program

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

Success!

## COPY and the build cache

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

## Details

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

---

## A quick word about the Docker Hub



## Lesson 12: 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.

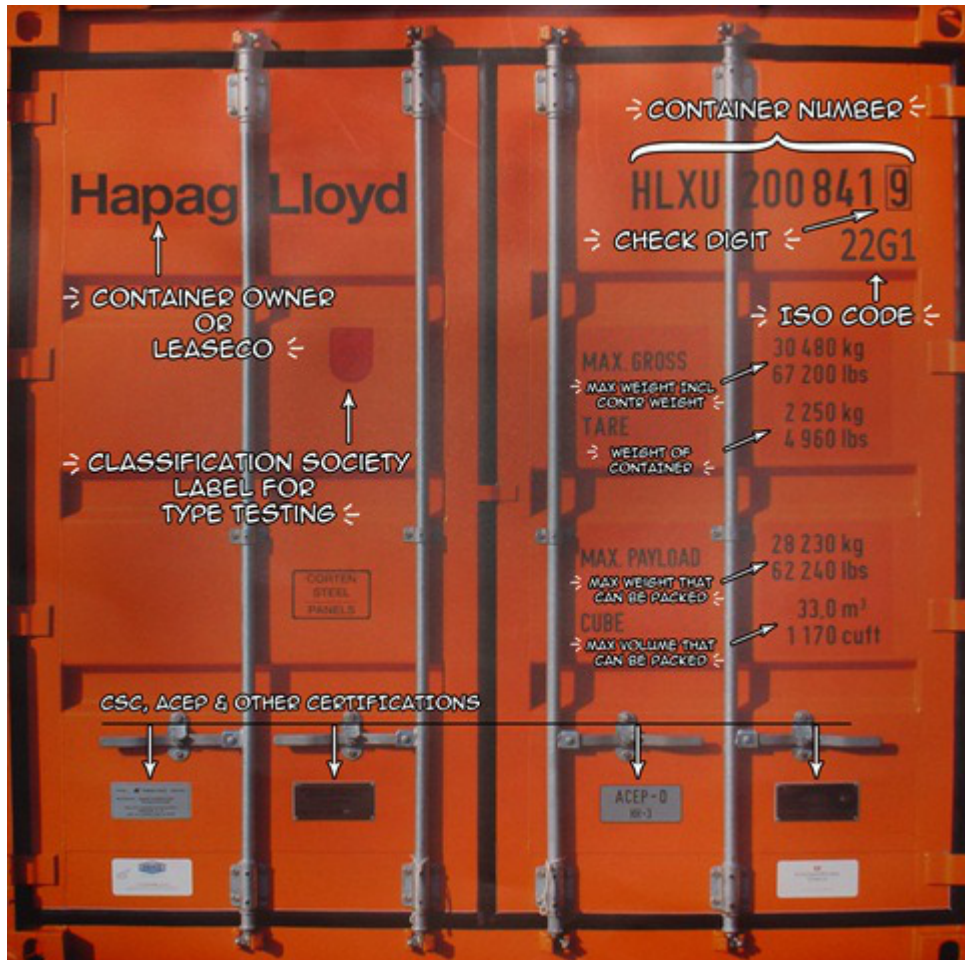
We won't do it since we don't want to force everyone to create a Docker Hub account (although it's free, yay!) but the steps would be:

- 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 13: 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

Since Docker 1.5 (released February 2015), you can rename containers with `docker rename`.

This allows you to "free up" a name without destroying the associated container, for instance.

# 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.

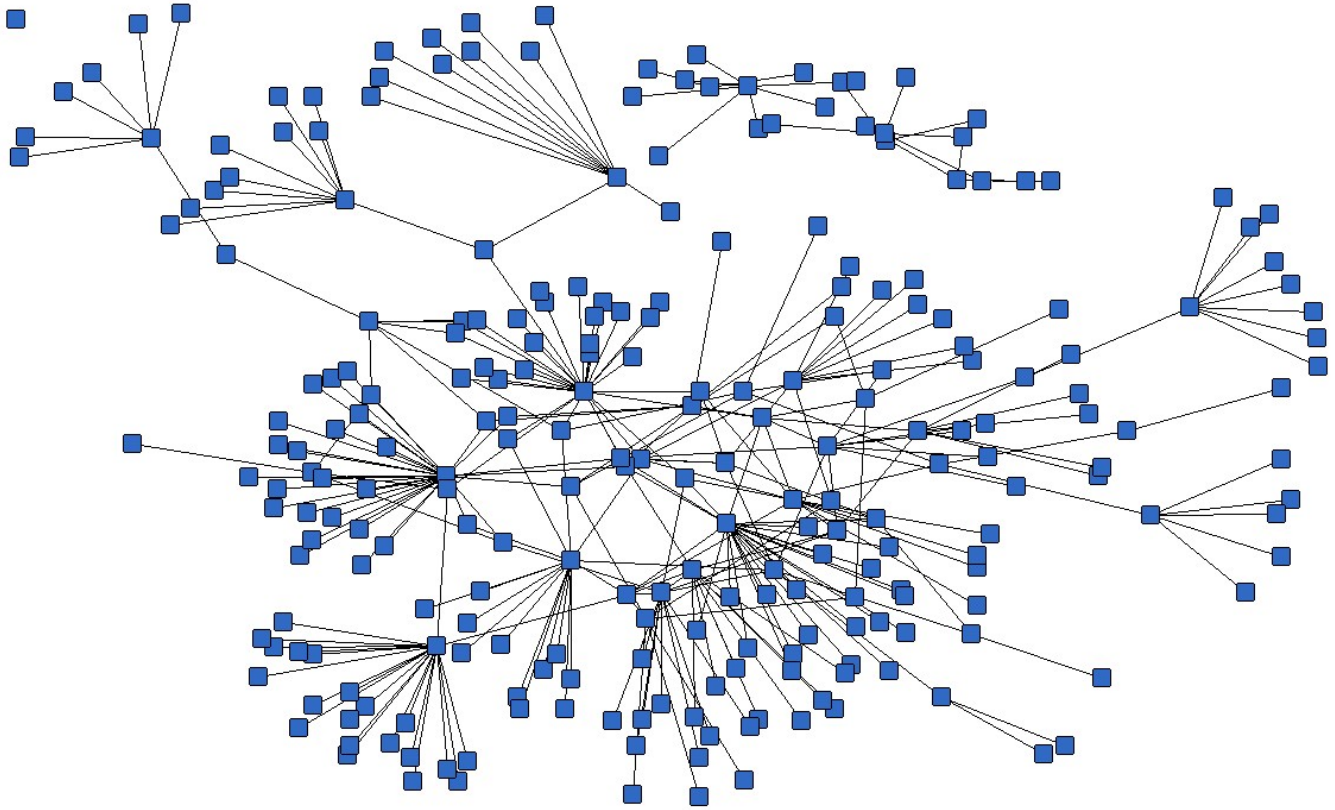
## Using `--format`

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 14: 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
```

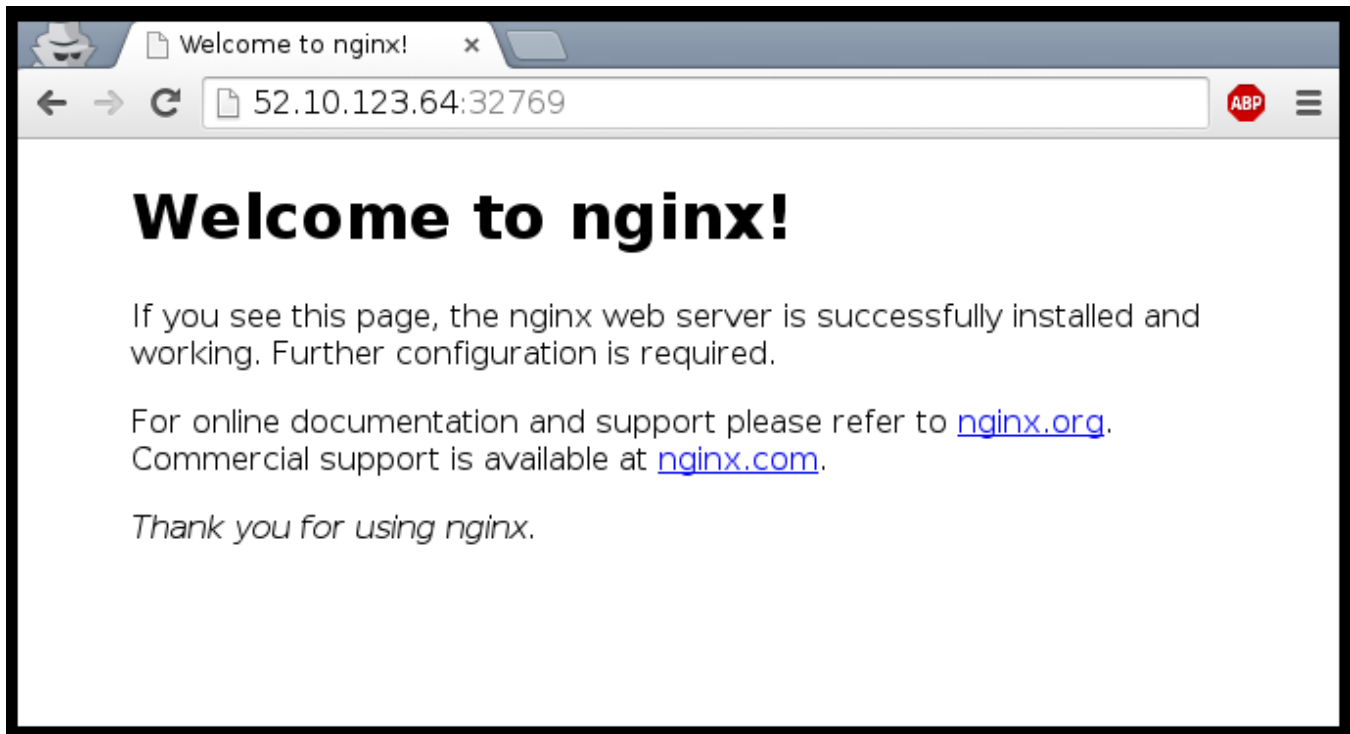
- The web server is running on ports 80 and 443 inside the container.
- Those ports are mapped to ports 32769 and 32768 on our Docker host.

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
```

- We are running two NGINX web servers.
- The first one is exposed on port 80.
- The second one is exposed on port 8000.

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

# Plumbing containers into your infrastructure

There are (at least) three ways to integrate containers in your network.

- Start the container, letting Docker allocate a public port for it. Then retrieve that port number and feed it to your configuration.
- Pick a fixed port number in advance, when you generate your configuration. Then start your container by setting the port numbers manually.
- Use an overlay network, connecting your containers with e.g. VLANs, tunnels...

## 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 old model (before Engine 1.9)

A container could use one of the following drivers:

- bridge (default)
- none
- host
- container

## 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!

## The container driver

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

## The new model (since Engine 1.9.0)

DON'T PANIC: all the previous drivers are still available.

Docker now has 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       skynet            bridge
228a4355d548       darknet           overlay
3f1733d3e233       darkernet         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.
- More drivers can be provided by plugins (OVS, VLAN...)

# Creating a network

Let's create a network.

```
$ docker network create skynet  
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       skynet            bridge
```

# Placing containers on a network

We will create two *named* containers on this network.

First, let's create this container in the background.

```
$ docker run -dti --name t800 --net skynet alpine sh  
8abb80e229ce8926c7223beb69699f5f34d6f1d438bfc5682db893e798046863
```

Now, create this other container in the foreground.

```
$ docker run -ti --name t1000 --net skynet ubuntu  
root@0eccdfa45ef:/#
```

# Communication between containers

From our new container (t1000), we can resolve and ping the other one, using its assigned name:

```
root@0ecccdfa45ef:/# ping t800
PING t800 (172.18.0.2) 56(84) bytes of data.
64 bytes from t800 (172.18.0.2): icmp_seq=1 ttl=64 time=0.221 ms
64 bytes from t800 (172.18.0.2): icmp_seq=2 ttl=64 time=0.114 ms
64 bytes from t800 (172.18.0.2): icmp_seq=3 ttl=64 time=0.114 ms
^C
--- t800 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:/#
```

How did that work?



# Resolving container addresses

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

```
root@0ecccdfa45ef:/# cat /etc/hosts
172.18.0.3 0ecccdfa45ef
127.0.0.1  localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0  ip6-localnet
ff00::0  ip6-mcastprefix
ff02::1  ip6-allnodes
ff02::2  ip6-allrouters
172.18.0.2  t800
172.18.0.2  t800.skynet
```

In the future, this will *probably* be replaced by a dynamic resolver.

# Connecting to multiple networks

Let's create another network.

```
$ docker network create resistance  
955b84336816b8e2265a156905aa716f5d1d880516ceaba48b9331f8f4e706aa
```

Create a container in this network.

```
$ docker run --net resistance -ti --name sarahconnor ubuntu  
root@4937d654a579:/#
```

This container cannot ping t800 (try it).

Now, from another terminal, connect t800 to the resistance:

```
$ docker network connect resistance t800
```

Then try again to ping t800 from sarahconnor. It works!

# Implementation details

With the "bridge" network driver, each container joining a network receives a new virtual interface.

Each container receives a new virtual interface:

```
$ docker run --net container:t800 alpine ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
73: eth0@if74: <BROADCAST,MULTICAST,...> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe12:2/64 scope link
        valid_lft forever preferred_lft forever
84: eth1@if85: <BROADCAST,MULTICAST,...> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:13:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.19.0.3/16 scope global eth1
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe13:3/64 scope link
        valid_lft forever preferred_lft forever
```

# Multi-host networking

Out of the scope for this intro-level workshop!

Very short instructions:

- deploy a key/value store (Consul, Etc, Zookeeper)
- add two extra flags to your Docker Engine
- you can now create networks using the overlay driver!

When you create a network on one host with the overlay driver, it appears automatically on all other hosts.

Containers placed on the same networks are able to resolve and ping as if they were local.

The overlay network is based on VXLAN and store neighbor info in a key/value store.

## Section summary

We've learned how to:

- Expose a network port.
- Manipulate container networking basics.
- Find a container's IP address.
- Create private networks for groups of containers.

**NOTE:** Later we will see another mechanism to interconnect containers using the `link` primitive.

# Local Development Workflow with Docker



# Lesson 15: 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:master
```

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:master
```

- 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  training/namer:latest              rackup                   3 seconds ago   Up 3 seconds
0.0.0.0:80->9292/tcp  condescending_shockley
```

## Viewing our application

Now let's browse to our web application on:

`http://<yourHostIP>:80`

We can see our company naming application.



# 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.



# 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:  
    - 9292:9292
```

# 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.

It is not meant to be used for production (except in emergencies, as a sort of pseudo-SSH), but it is handy for development.

You can get a shell prompt inside an existing container this way.

## 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 16: 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 /var/lib/postgresql
```

- On the command-line, with the `-v` flag for `docker run`.

```
$ docker run -d -v /var/lib/postgresql \
  training/postgresql
```

In both cases, `/var/lib/postgresql` (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.

## 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.

In the last example, it doesn't matter if container `alpha` is running or not.

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

```
$ docker volume ls
DRIVER          VOLUME NAME
local          5b0b65e4316da67c2d471086640e6005ca2264f3...
local          vol02
local          vol04
local          13b59c9936d78d109d094693446e174e5480d973...
```

Some of those volume names were explicit (`vol02`, `vol04`).

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 volumes directly (without data containers).

```
$ docker volume create --name=files  
files  
$ docker volume create --name=logs  
logs
```

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 the same containers as before:

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

Again: volumes are not anchored to a specific path.

(This can be a good or a bad thing.)

## 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.

# 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`

- 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.

# 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 archeology 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
```

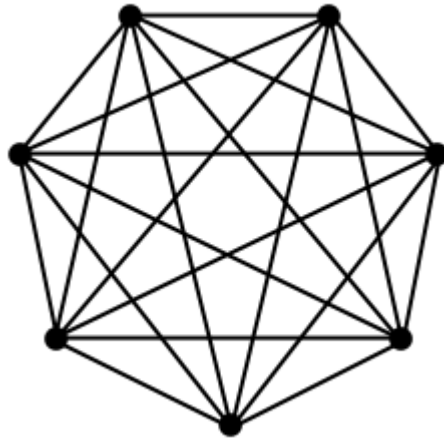
**Warning:** when using such mounts, the container gains root-like access to the host. It can potentially do bad things.

## Section summary

We've learned how to:

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

# Connecting Containers



# Lesson 17: Connecting containers

## Objectives

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

- Create links between containers.
- Use names and links to communicate across containers.
- Use these features to decouple app dependencies and reduce complexity.

# Connecting containers

- We will learn how to use names and links to expose one container's port(s) to another.
- Why? So each component of your app (e.g., DB vs. web app) can run independently with its own dependencies.

## What we've got planned

- We're going to get two images: a Redis (key-value store) image and a Ruby on Rails application image.
- We're going to start containers from each image.
- We're going to link the container running our Rails application and the container running Redis using Docker's link primitive.

# Launch a container from the `redis` image.

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

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

Let's check the container is running:

```
$ docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	
STATUS	PORTS	NAMES		
9efd72a4f320	redis:latest	redis-server	5 seconds ago	Up
4 seconds	6379/tcp	mycache		

- Our container is launched and running an instance of Redis.
- Using the `--name` flag we've given it a name: `mycache`. Remember that! Container names are unique. We're going to use that name shortly.



## Trying our Rails app

Try to run the application, without any other preparation:

```
$ docker run -dP nathanleclaire/redisonrails
```

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

It doesn't work!

## How our app connects to Redis

If we pull the code, we will see the following line:

```
$redis = Redis.new(:host => 'redis', :port => 6379)
```

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

*Obviously* it doesn't work.

# Launch a container from the nathanleclaire/redisonrails image.

Let's launch a container from the nathanleclaire/redisonrails image, without links to start.

In the Rails console we can see that `$redis` exists, but we did not link to any actual Redis instance.

```
$ docker run -it nathanleclaire/redisonrails rails console
Loading development environment (Rails 4.0.2)
irb(main):001:0> $redis
=> #<Redis client v3.1.0 for redis://redis:6379/0>
irb(main):002:0> $redis.set('foo', 'bar')
SocketError: getaddrinfo: Name or service not known
  from /usr/local/lib/ruby/gems/2.1.0/gems/redis-3.1.0/lib/redis/connection/
ruby.rb:152:in `getaddrinfo'
  from /usr/local/lib/ruby/gems/2.1.0/gems/redis-3.1.0/lib/redis/connection/
ruby.rb:152:in `connect'
  from /usr/local/lib/ruby/gems/2.1.0/gems/redis-3.1.0/lib/redis/connection/
ruby.rb:211:in `connect'
  from /usr/local/lib/ruby/gems/2.1.0/gems/redis-3.1.0/lib/redis/client.rb:304:in
`establish_connection'
.....
```

Without access to a Redis server at the proper location the initialized `$redis` object will not work.

## Launch and link a 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 mycache:redis alpine sh
```

In this container, we can communicate with `mycache` 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 mycache:redis nathanleclaire/redisonrails 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
```

# Access our container with the Rails console

You can skip this example if you're not comfortable with Ruby.

```
$ docker run -it --link mycache:redis \
  nathanleclaire/redisonrails rails console
Loading development environment (Rails 4.0.2)
irb(main):001:0> $redis
=> #<Redis client v3.1.0 for redis://redis:6379/0>
irb(main):002:0> $redis.set('a', 'b')
=> "OK"
irb(main):003:0> $redis.get('a')
=> "b"
irb(main):004:0> $redis.set('someHash', {:foo => 'bar', :spam => 'eggs'})
=> "OK"
irb(main):005:0> $redis.get('someHash')
=> "{:foo=>\"bar\", :spam=>\"eggs\"}"
irb(main):006:0> $redis.set('users', ['Aaron', 'Jerome', 'Nathan'])
=> "OK"
irb(main):007:0> $redis.get('users')
=> "[\"Aaron\", \"Jerome\", \"Nathan\"]"
irb(main):008:0> exit
```

Woot! That's more like it.

- The `--link` flag connects one container to another.
- We specify the name of the container to link to, `mycache`, and an alias for the link, `redis`, in the format `name:alias`.
- We can use `$redis` in an `ActiveRecord` class to create data models that have the speed on in-memory lookups.

## More about our link - Environment variables

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

Let's see that information:

```
$ docker run --link mycache: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 `mycache` container via `ENV` instructions.

## Starting our Rails application

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

```
$ docker run -d -p 80:3000 --link mycache:redis nathanleclaire/redisonrails
```

Now let's check the container is running.

```
$ docker ps -l
```



# Starting our Rails application

Our homepage controller contains the following code.

```
class WelcomeController < ApplicationController
  def index
    views = $redis.get('views').to_i()
    views += 1
    $redis.set('views', views)
    @page_views = views
  end
end
```

## Viewing our Rails application

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

http://<yourHostIP>



**This is an app connected to Redis.**

It has been viewed 40 times.

## Tidying up

Finally let's tidy up our database.

```
$ docker kill mycache  
.  
.  
.  
$ docker rm mycache
```

- We can use the container name to stop and remove them.
- We removed it so we can re-use its name later if we want.

(Remember container names are unique!)

## Section summary

We've learned how to:

- Create links between containers.
- Use names and links to communicate across containers.
- Use these features to decouple app dependencies and reduce complexity.

# Ambassadors



# Lesson 18: Ambassadors

## Objectives

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

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

# Ambassadors

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

- With links.
- Using host-based volumes.
- Using data volumes shared between containers.

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

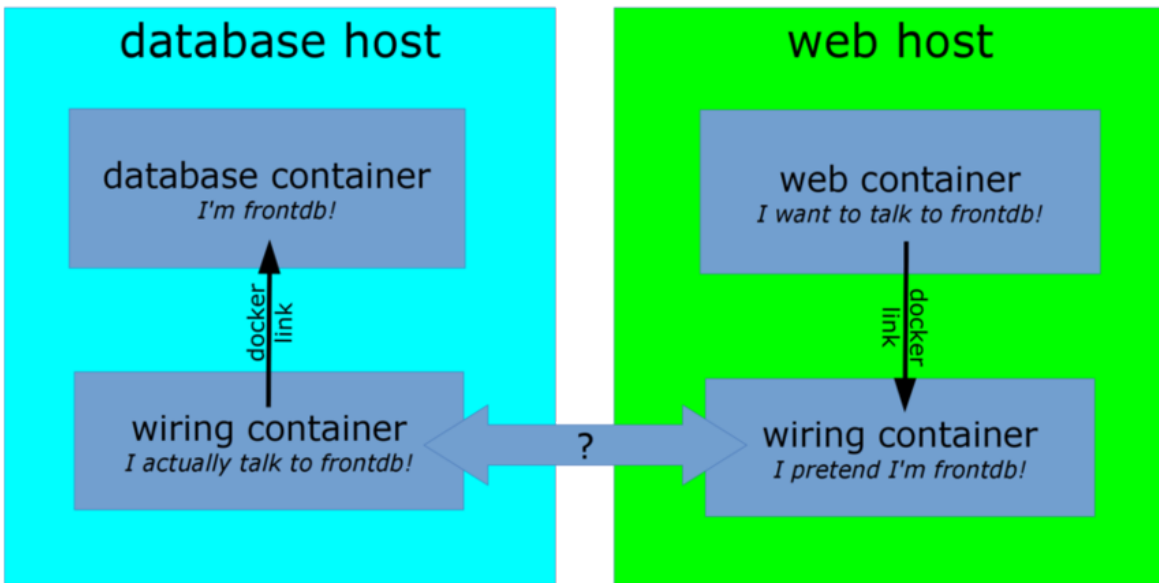
# Introduction to Ambassadors

The ambassador pattern:

- Takes advantage of Docker's lightweight linkages 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 a normal link to connect to the ambassador.
- The database container is linked with an ambassador as well.
- For both containers, there is no difference between normal operation and operation with ambassador containers.
- If the database container is moved, its new location will be tracked by the ambassador containers, and the web application container will still be able to connect, without reconfiguration.

# Implementing the ambassador 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.

## 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: look for [DVO317](#) (AWS re:invent talk).

# Compose For Development Stacks



# Lesson 19: 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

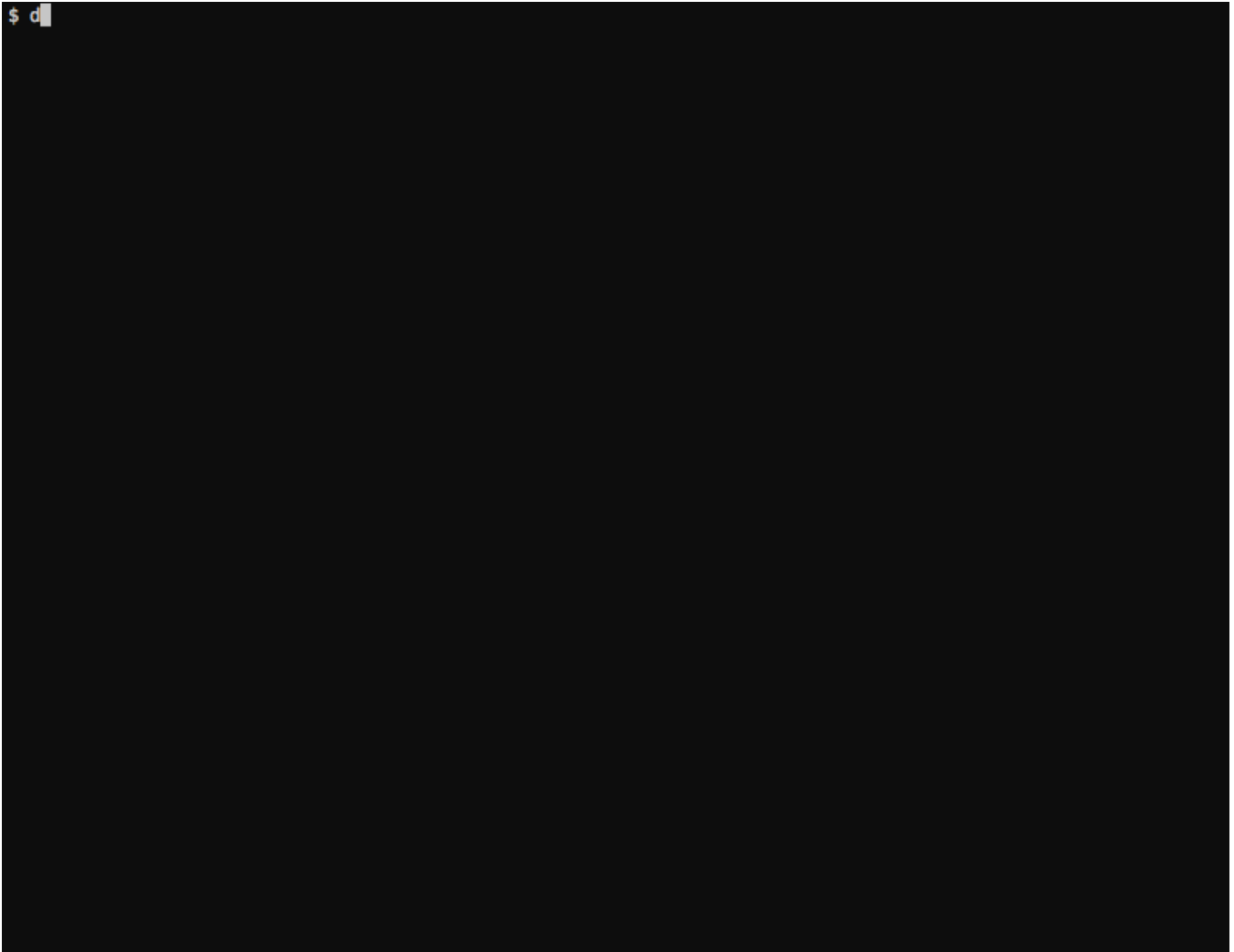
This is how you work with Compose:

- 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/1.2.0/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
```

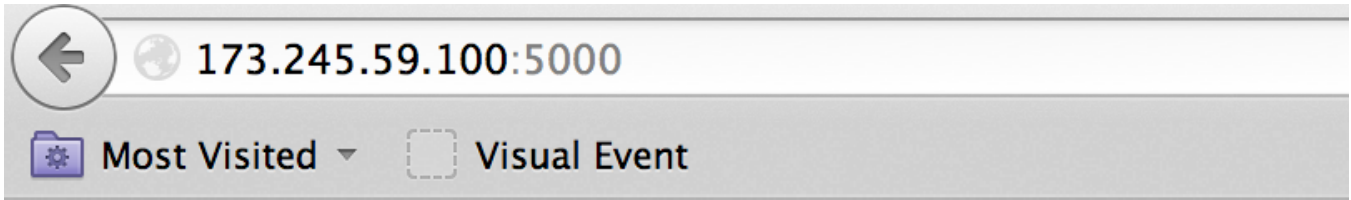
Second step: start your app.

```
$ 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>:5000`.



**Hello Docker Training! I have been seen 8 times**

## 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:

```
www:
  build: www
  ports:
    - 8000:5000
  links:
    - redis
  user: nobody
  command: python counter.py
  volumes:
    - ./www:/src
```

redis: image: redis

Each section of the YAML file (web, redis) corresponds to a container.

Let's see what can be in a section.

# Containers in `docker-compose.yml`

Each section of 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.
- `links` translates to one (or multiple) `--link` options. You can refer to other Compose containers by their name.

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
-----
figdemo_redis_1      redis-server        Exit 0
figdemo_www_1        gunicorn --bi       Exit 0
```

# 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...
```

## 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.

# Advanced Dockerfiles



# Lesson 20: 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`.

# 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 on 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
```

- 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 ADD instruction

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

```
ADD /src/webapp /opt/webapp
```

This will add the contents of the `/src/webapp/` directory to the `/opt/webapp` directory in the image.

Note: `/src/webapp/` is not relative to the host filesystem, but to the directory containing the `Dockerfile`.

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

The ADD instruction can also be used to get remote files.

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

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

## More about the ADD instruction

- ADD is cached. If you recreate the image and no files have changed then a cache is used.
- If the local source is a zip file or a tarball it'll be unpacked to the destination.
- Sources that are URLs and zipped will not be unpacked.
- Any files created by the ADD instruction are owned by root with permissions of 0600.

More on ADD [here](#).

# The VOLUME instruction

The VOLUME instruction will create a data volume mount point at the specified path.

```
VOLUME [ "/opt/webapp/data" ]
```

- Data volumes bypass the union file system.  
In other words, they are not captured by `docker commit`.
- Data volumes can be shared and reused between containers.  
We'll see how this works in a subsequent lesson.
- It is possible to share a volume with a stopped container.
- Data volumes persist until all containers referencing them are destroyed.

# 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 /opt/webapp
```

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.

# 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 . /app/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 ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y -q \
    python-all python-pip
COPY ./webapp /opt/webapp/
WORKDIR /opt/webapp
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 ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y -q \
    python-all python-pip
COPY ./webapp/requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
COPY ./webapp /opt/webapp/
WORKDIR /opt/webapp
EXPOSE 5000
CMD ["python", "app.py"]
```



# Security



# Lesson 21: Security

## Objectives

At the end of this lesson, you will know:

- The security implications of exposing Docker's API
- How to take basic steps to make containers more secure
- Where to find more information on Docker security

## What can we do with Docker API access?

Someone who has access to the Docker API will have full root privileges on the Docker host.

If you give root privileges to someone, assume that they can do *anything they like* on the host, including:

- Accessing all data.
- Changing all data.
- Creating new user accounts and changing passwords.
- Installing stealth rootkits.
- Shutting down the machine.

# Accessing the host filesystem

To do that, we will use `-v` to expose the host filesystem inside a container:

```
$ docker run -v /:/hostfs ubuntu cat /hostfs/etc/passwd  
...This shows the content of /etc/passwd on the host...
```

If you want to explore freely the host filesystem:

```
$ docker run -it -v /:/hostfs -w /hostfs ubuntu bash
```

# Modifying the host filesystem

Volumes are read-write by default, so let's create a dummy file on the host filesystem:

```
$ docker run -it -v /:/hostfs ubuntu touch /hostfs/hi-there
$ ls -l /
...You will see the hi-there file, created on the host...
```

Note: if you are using boot2docker or a remote Docker host, you won't see the `hi-there` file. It will be in the boot2docker VM, or on the remote Docker host instead.

## Privileged containers

If you start a container with `--privileged`, it will be able to access all devices and perform all operations.

For instance, it will be able to access the whole kernel memory by reading (and even writing!) `/dev/kcore`.

A container could also be started with `--net host` and `--privileged` together, and be able to sniff all the traffic going in and out of the machine.

## Other harmful operations

We won't explain how to do this (because we don't want you to break your Docker machines), but with access to the Docker API, you can:

- Add user accounts.
- Change password of existing accounts.
- Add SSH key authentication to existing accounts.
- Insert kernel modules.
- Run malicious processes and insert special kernel code to hide them.

## What to do?

- Do not expose the Docker API to the general public.
- If you expose the Docker API, secure it with TLS certificates.
- TLS certificates will be presented in the next section.
- Make sure that your users are trained to not give away credentials.



# Security of containers themselves

- "Containers Do Not Contain!"
- Containers themselves do not have security features.
- Security is ensured by a number of other mechanisms.
- We will now review some of those mechanisms.

## Do not run processes as root

- By default, Docker runs everything as root.
- This is a security risk.
- Docker might eventually drop root privileges automatically, but until then, you should specify USER in your Dockerfiles, or use su or sudo.

## Don't colocate security-sensitive containers

- If a container contains security-sensitive information, put it on its own Docker host, without other containers.
- Other containers (private development environments, non-sensitive applications...) can be put together.

## Run AppArmor or SELinux

- Both of these will provide you with an additional layer of protection if an attacker is able to gain elevated access.

# Learn more about containers and security

- Presentation given at LinuxCon 2014 (Chicago)

<http://www.slideshare.net/jpetazzo/docker-linux-containers-lxc-and-security>

## Section summary

We have learned:

- The security implications of exposing Docker's API
- How to take basic steps to make containers more secure
- Where to find more information on Docker security

# Dealing with Vulnerabilities



# Lesson 22: Dealing with Vulnerabilities

## Objectives

We will discuss how to address security vulnerabilities disclosures:

- Concerning Docker itself.
- Concerning the dependencies pulled into your container images.



# Vulnerabilities

When a vulnerability is discovered, it is common practice for the vendor or community to issue a *security advisory*.

The security advisory will typically indicate:

- The affected software.
- The specific scenarios (if relevant) where the vulnerability should be a concern.
- The specific versions affected by the vulnerability.
- The severity of the vulnerability (e.g. can it lead to malfunction, or prevent others from using the software or service, or allow unauthorized access or modification to data).
- How to remediate it (typically by upgrading affected software).

# Vulnerabilities in Docker

The typical method to deal with Docker vulnerabilities is:

- Stop all containers.
- Stop the Docker daemon.
- Upgrade the Docker daemon.
- Start the Docker daemon.
- Start the container.

## Vulnerabilities in images

If a vulnerability is announced concerning a package that you are using in your images, you should upgrade those images.

Assuming that updated packages are available, you should:

- Force a `docker pull` of all your base images.
- Re-execute a `docker build --no-cache` of all your built images.
- Re-start all containers using the updated images.

## Detecting vulnerabilities

You can use traditional auditing systems, but Docker provides new, efficient, non-intrusive ways to perform security audit and vulnerability reporting.

The use of the `--read-only` flag lets you perform offline analysis of images to detect those which have vulnerable software.

---

## Securing Docker with TLS

# Lesson 23: Securing Docker with TLS

## Objectives

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

- Understand how Docker uses TLS to secure and authorize remote clients
- Create a TLS Certificate Authority
- Create TLS Keys
- Sign TLS Keys
- Use these keys with Docker

## Why should I care?

- Docker does not have any access controls on its network API unless you use TLS!

# What is TLS

- TLS is Transport Layer Security.
- The protocol that secures websites with `https` URLs.
- Uses Public Key Cryptography to encrypt connections.
- Keys are signed with Certificates which are maintained by a trusted party.
- These Certificates indicate that a trusted party believes the server is who it says it is.
- Each transaction is therefor encrypted *and* authenticated.



# How Docker Uses TLS

- Docker provides mechanisms to authenticate both the server the client to each other.
- Provides strong authentication, authorization and encryption for any API connection over the network.
- Client keys can be distributed to authorized clients

# Environment Preparation

- You need to make sure that OpenSSL version 1.0.1 is installed on your machine.
- Make a directory for all of the files to reside.
- Make sure that the directory is protected and backed up!
- *Treat these files the same as a root password.*

## Creating a Certificate Authority

First, initialize the CA serial file and generate CA private and public keys:

```
$ echo 01 > ca.srl  
$ openssl genrsa -des3 -out ca-key.pem 2048  
$ openssl req -new -x509 -days 365 -key ca-key.pem -out ca.pem
```

We will use the `ca . pem` file to sign all of the other keys later.

## Create and Sign the Server Key

Now that we have a CA, we can create a server key and certificate signing request. Make sure that CN matches the hostname you run the Docker daemon on:

```
$ openssl genrsa -des3 -out server-key.pem 2048
$ openssl req -subj '/CN=**<Your Hostname Here>**' -new -key server-key.pem -out
server.csr
$ openssl rsa -in server-key.pem -out server-key.pem
```

Next we're going to sign the key with our CA:

```
$ openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pem \
-out server-cert.pem
```

## Create and Sign the Client Key

```
$ openssl genrsa -des3 -out client-key.pem 2048  
$ openssl req -subj '/CN=client' -new -key client-key.pem -out client.csr  
$ openssl rsa -in client-key.pem -out client-key.pem
```

To make the key suitable for client authentication, create a extensions config file:

```
$ echo extendedKeyUsage = clientAuth > extfile.cnf
```

Now sign the key:

```
$ openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.pem \  
-out client-cert.pem -extfile extfile.cnf
```

# Configuring the Docker Daemon for TLS

- By default, Docker does not listen on the network at all.
- To enable remote connections, use the `-H` flag.
- The assigned port for Docker over TLS is 2376.

```
$ sudo docker -d --tlsverify  
--tlscacert=ca.pem  
--tlscert=server-cert.pem  
--tlskey=server-key.pem -H=0.0.0.0:2376
```

Note: You will need to modify the startup scripts on your server for this to be permanent! The keys should be placed in a secure system directory, such as `/etc/docker`.

# Configuring the Docker Client for TLS

If you want to secure your Docker client connections by default, you can move the key files to the `.docker` directory in your home directory. Set the `DOCKER_HOST` variable as well.

```
$ cp ca.pem ~/.docker/ca.pem
$ cp client-cert.pem ~/.docker/cert.pem
$ cp client-key.pem ~/.docker/key.pem
$ export DOCKER_HOST=tcp://:2376
```

Then you can run docker with the `--tlsverify` option.

```
$ docker --tlsverify ps
```

# Section Summary

We learned how to:

- Create a TLS Certificate Authority
- Create TLS Keys
- Sign TLS Keys
- Use these keys with Docker



---

# The Docker API

# Lesson 24: The Docker API

## Objectives

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

- Work with the Docker API.
- Create and manage containers with the Docker API.
- Manage images with the Docker API.

# Introduction to the Docker API

So far we've used Docker's command line tools to interact with it. Docker also has a fully fledged RESTful API you can work with.

The API allows:

- To build images.
- Run containers.
- Manage containers.

# Docker API details

The Docker API is:

- Broadly RESTful with some commands hijacking the HTTP connection for STDIN, STDERR, and STDOUT.
- The API binds locally to `unix:///var/run/docker.sock` but can also be bound to a network interface.
- Not authenticated by default.
- Securable with certificates.

In the examples below, we will assume that Docker has been setup so that the API listens on port 2375, because tools like `curl` can't talk to a local UNIX socket directly.

# Testing the Docker API

Let's start by using the `info` endpoint to test the Docker API.

This endpoint returns basic information about our Docker host.

```
$ curl --silent -X GET http://localhost:2375/info \
| python -mjson.tool
{
  "Containers": 68,
  "Debug": 0,
  "Driver": "aufs",
  "DriverStatus": [
    [
      "Root Dir",
      "/var/lib/docker/aufs"
    ],
    [
      "Dirs",
      "711"
    ]
  ],
  "ExecutionDriver": "native-0.2",
  "IPv4Forwarding": 1,
  "Images": 575,
  "IndexServerAddress": "https://index.docker.io/v1/",
  "InitPath": "/usr/bin/docker",
  "InitShal": "",
  "KernelVersion": "3.14.0-1-amd64",
  "MemoryLimit": 1,
  "NEventsListener": 0,
  "NFd": 11,
  "NGoroutines": 11,
  "OperatingSystem": "<unknown>",
  "SwapLimit": 1
}
```

## Doing `docker run` via the API

It is simple to do `docker run` with the CLI, but it is more complex with the API. It involves multiple calls.

We will focus on *detached* containers for now (i.e., running in the background). Interactive containers involve hijacking the HTTP connection. This is easily handled with Docker client libraries, but for now, we will use regular tools like `curl`.

# Container lifecycle with the API

To run a container, you must:

- Create the container. It is then stopped, but ready to go.
- Start the container.
- Optionally, you can wait for the container to exit.
- You can also retrieve the container output (logs) with the API.

Each of those operations corresponds to a specific API call.

## "Create" vs. "Start"

The `create` API call creates the container, and gives us the ID of the newly created container. The container does not run yet, though.

The `start` API call tells Docker to transition the container from "stopped" to "running".

Those are two different calls, so you can attach to the container before starting it, to make sure that you will not miss any output from the container, for instance.

Some parameters (e.g. which image to use, memory limits) must be specified with `create`; others (e.g. ports and volumes mappings) must be specified with `start`.

To see the list of all parameters, check the API reference documentation.



# Creating a new container via the API

Let's use `curl` to create a simple container.

```
$ curl -X POST -H 'Content-Type: application/json' \
  http://localhost:2375/containers/create \
  -d '{
    "Cmd":["echo", "hello world"],
    "Image":"busybox"
  }'
{"Id":"<yourContainerID>","Warnings":null}
```

- You can see the container ID returned by the API.
- The `Cmd` parameter has to be a list.  
  
(If you put `echo hello world` it will try to execute a binary called `echo hello world`.)
- You can add more parameters in the JSON structure.
- The only mandatory parameter is the `Image` to use.

## Starting our new container via the API

In the previous step, the API gave you a container ID.

You will have to copy-paste that ID.

```
$ curl -X POST -H 'Content-Type: application/json' \  
  http://localhost:2375/containers/<yourContainerID>/start \  
  -d '{}'
```

No output will be shown (unless an error happens).

# Inspecting our launched container

We can also inspect our freshly launched container.

```
$ curl --silent \
  http://localhost:2375/containers/<yourContainerID>/json |
  python -mjson.tool
{
  "Args": [
    "hello world"
  ],
  "Config": {
    "AttachStderr": false,
    "AttachStdin": false,
    "AttachStdout": false,
    "Cmd": [
      "echo",
      "hello world"
    ],
    . . .
  }
}
```

- It returns the same hash the `docker inspect` command returns.

# Waiting for our container to exit and check its status code

Our test container will run and exit almost instantly.

But for containers running for a longer period of time, we can call the `wait` endpoint.

The `wait` endpoint also gives the exit status of the container.

```
$ curl --silent -X POST \  
  http://localhost:2375/containers/<yourContainerID>/wait \  
{\"StatusCode\":0}
```

- Note that you have to use a `POST` method here.
- The `StatusCode` of `0` means that the process exited normally, without error.

## Viewing container output (logs)

Our container is supposed to echo `hello world`.

Let's verify that.

```
$ curl --silent \  
  http://localhost:2375/containers/<yourContainerID>/logs?stdout=1  
hello world
```

- There are other options, to select which streams to see (stdout and/or stderr), whether or not to show timestamps, and to follow the logs (like `tail -f` does).
- Check the API reference documentation to see all available options.

# Stopping a container

We can also stop a container using the API.

```
$ curl --silent -X POST \  
http://localhost:2375/containers/<yourContainerID>/stop
```

- Note that you have to use a POST call here.
- If it succeeds it will return a HTTP 204 response code.

# Working with images

We can also work with Docker images.

```
$ curl -X GET http://localhost:2375/images/json?all=0
[
  {
    "Created": 1396291095,
    "Id": "cccdc2d2ec497e814793e8bd952ae76d5d552c8bb7ed927db54aa65579508ffd",
    "ParentId": "9cd978db300e27386baa9dd791bf6dc818f13e52235b26e95703361ec3c94dc6",
    "RepoTags": [
      "training/datavol:latest"
    ],
    "Size": 0,
    "VirtualSize": 204371253
  },
  {
    "Created": 1396117401,
    "Id": "d4faa2107ddab5b22e815759d9a345f1381562ad44d1d95235347d6b006ec713",
    "ParentId": "439aa219e271671919a52a8d5f7a8e7c2b2950c639f09ce763ac3a06c0d15c22",
    . . .
  }
]
```

- Returns a hash of all images.

# Searching the Docker Hub for an image

We can also search the Docker Hub for specific images.

```
$ curl -X GET http://localhost:2375/images/search?term=training
[
  {
    "description": "",
    "is_official": false,
    "is_trusted": true,
    "name": "training/namer",
    "star_count": 0
  },
  {
    "description": "",
    "is_official": false,
    "is_trusted": true,
    "name": "training/postgres",
    "star_count": 0
  }
]
```

This returns a list of images and their metadata.



## Creating an image

We can then add one of these images to our Docker host.

```
$ curl -i -v -X POST \  
http://localhost:2375/images/create?fromImage=training/namer  
{"status":"Pulling repository training/namer"}
```

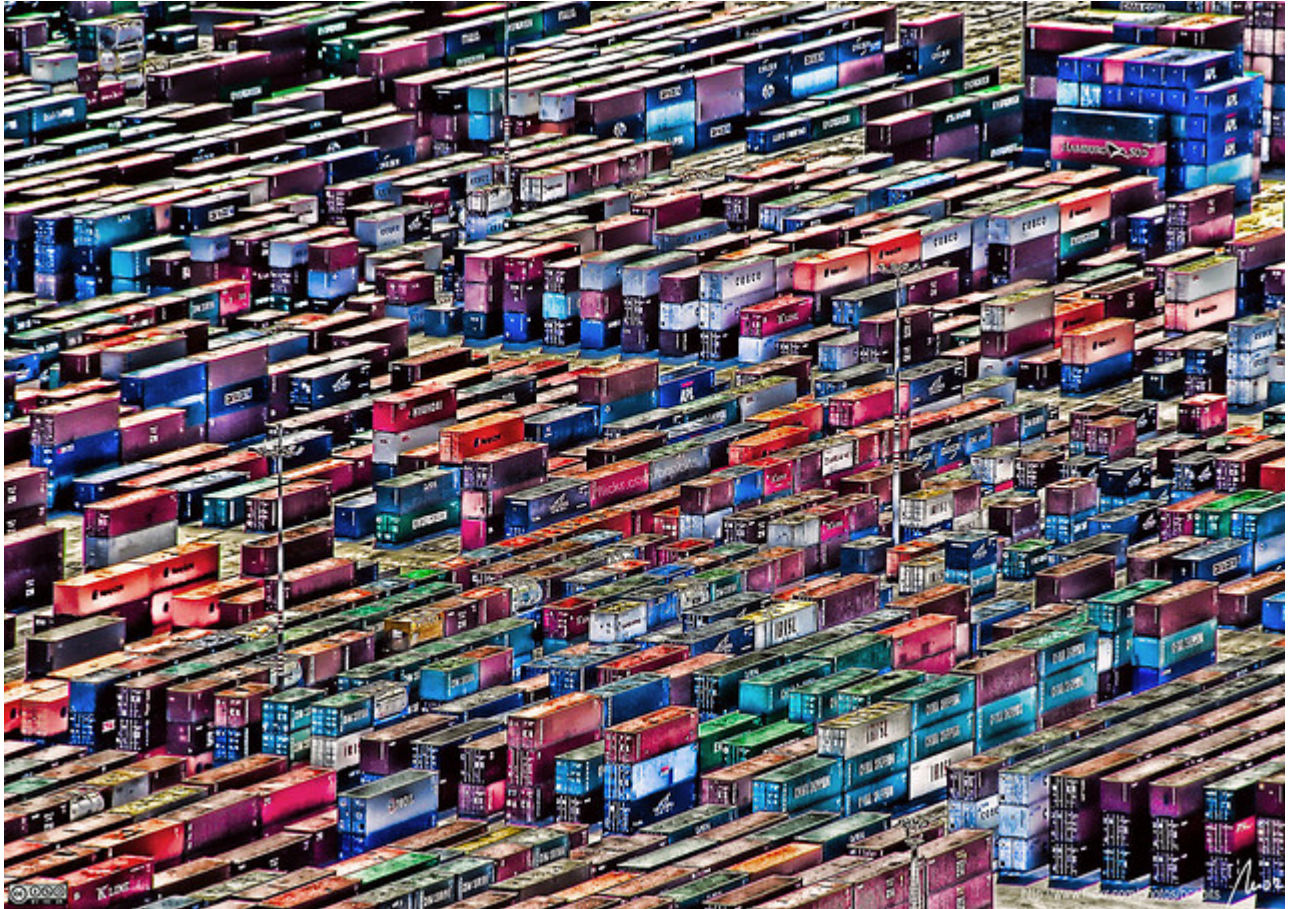
This will pull down the `training/namer` image and add it to our Docker host.

## Section summary

We've learned how to:

- Work with the Docker API.
- Create and manage containers with the Docker API.
- Manage images with the Docker API.

# 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 work flow

# 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

