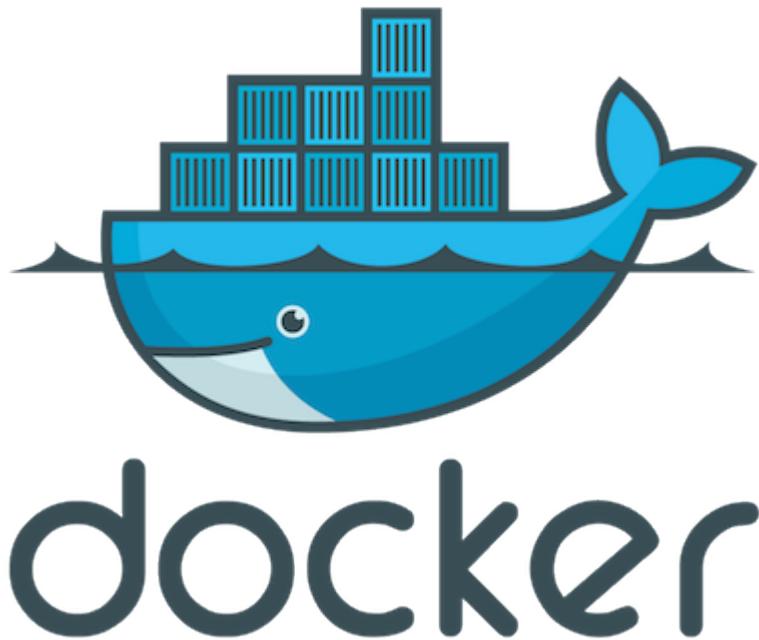


DeepMinds Training

Introduction to Docker



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

Other sessions on this topic, at SCALE...

Today

:

- Containerization in Production: The Good, The Bad, The Ugly
- Building a Minimal Host for Docker

Containers Tomorrow:

- Using Docker, CoreOS, and git hooks to deploy applications
- NGINX 101 - now with more Docker
- Docker and

Microservices Sunday:

- Docker for Multi-Cloud

Apps Non exhaustive list!

Course

You will:

- Learn what Docker is and what it is for.
- Learn the terminology, and define images, containers, etc.
- Learn how to install Docker.
- Use Docker to run containers.
- Use Docker to build containers.
- Learn how to orchestrate Docker containers.
- Interact with the Docker Hub website.
- Acquire tips and good practice.
- Know what's next: the future of Docker, and how to get help.

Course Agenda

- About Docker
- Using the training virtual machines
- Installing Docker
- Our first containers
- Running containers in the background
- Images and containers
- Building images manually
- Building images automatically
- Basic networking
- Local development workflow

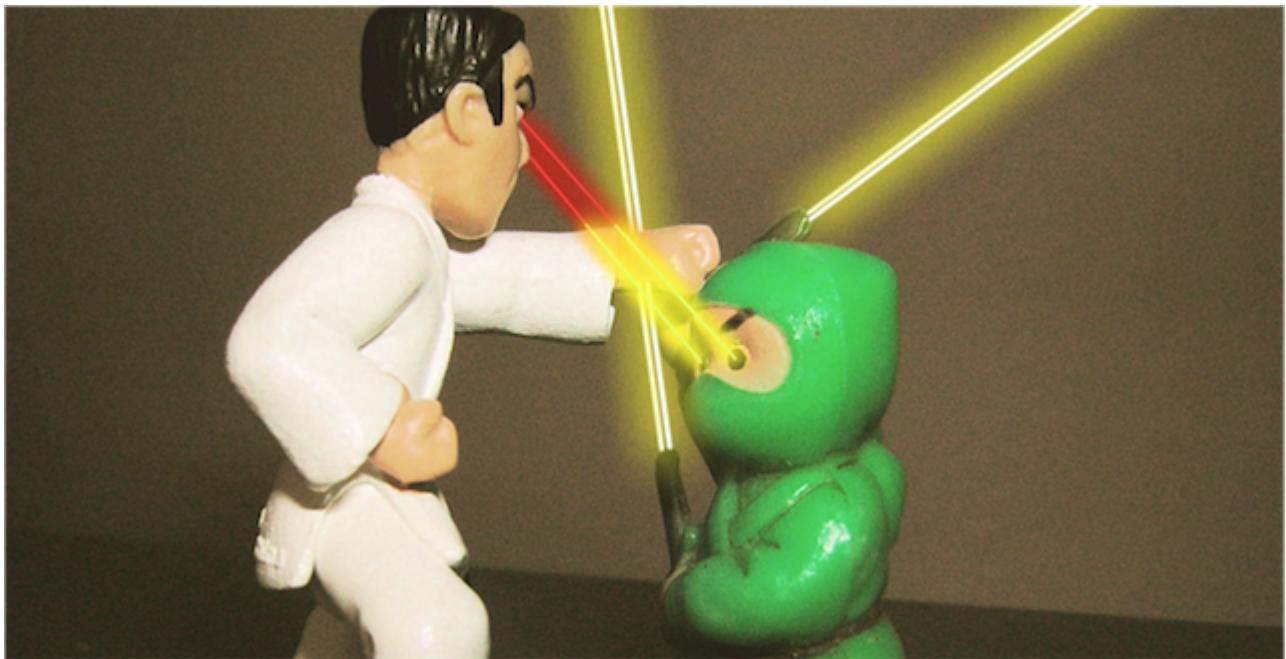
Course Agenda

- Working with volumes
- Connecting containers
- Advanced Dockerfiles
- Orchestration
- Ambassadors
- The Docker Hub
- Automated builds
- Security
- The Docker API

Table of

The table of contents is empty because you haven't selected paragraph styles to appear in it.

About Docker



Overview: About Docker

Objectives

In this lesson, we will learn about:

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

We won't actually run Docker or containers in this chapter (yet!), but don't worry, we will get to that fast enough!

About Docker Inc.

Focused on Docker and growing the Docker ecosystem:

- Founded in 2009.
- Formerly dotCloud Inc.
- Released Docker in 2013.

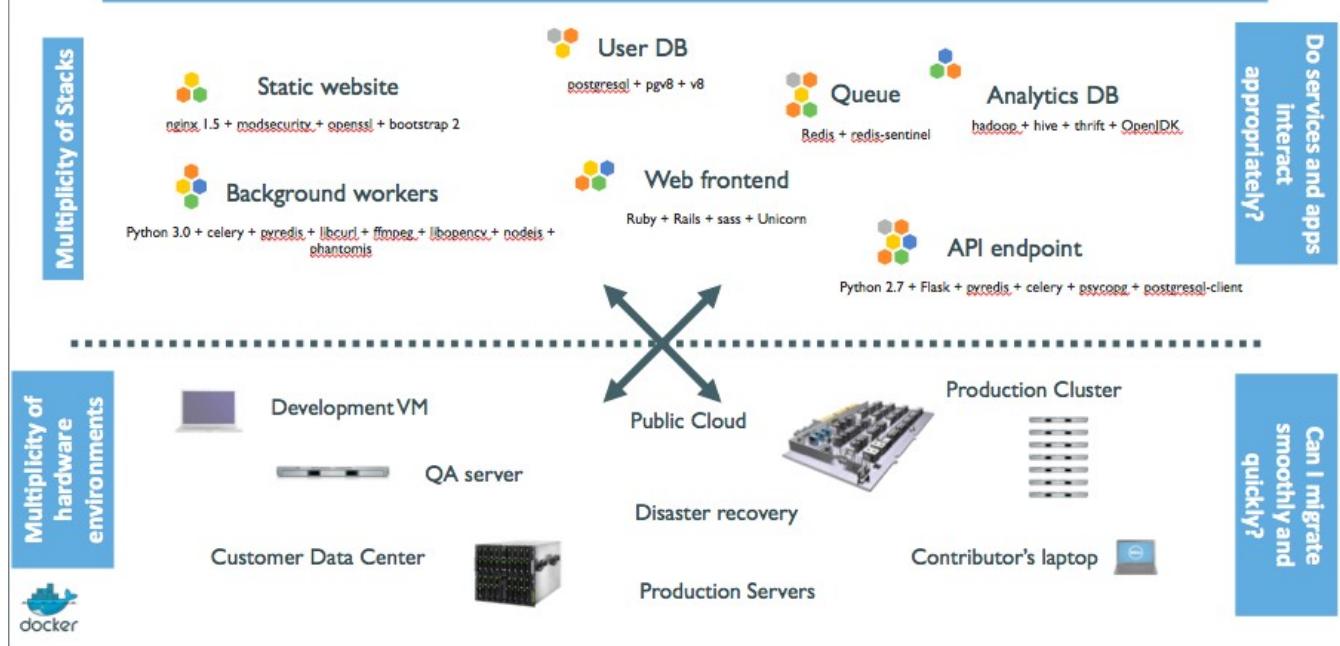
What does Docker Inc. do?

- Docker Engine - open source container management.
- Docker Hub - online home and hub for managing your Docker containers.
- Docker Enterprise Support - commercial support for Docker.
- Docker Services & Training - professional services and training to help you get the best out of Docker.

Why Docker?

- The software industry has changed.
- Applications used to be monolithic, with long lifecycles, scaled up.
- Today, applications are decoupled, built iteratively, scaled out.
- As a result, deployment is though!

The problem in 2015

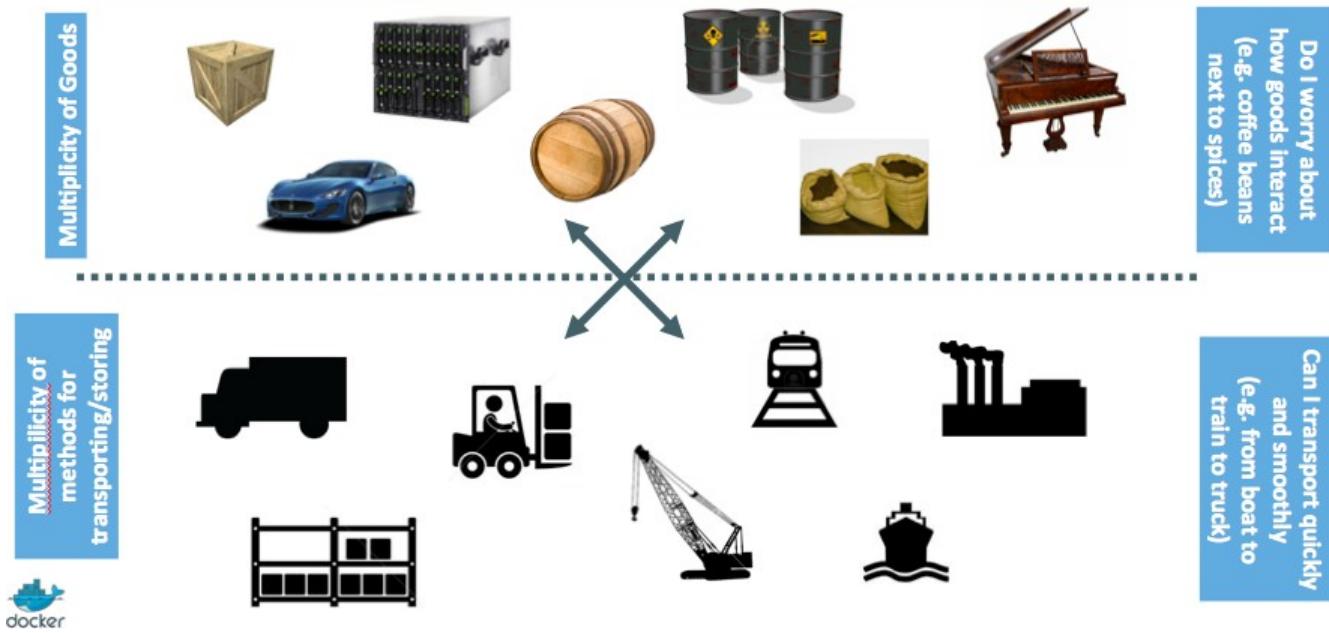


The Matrix from Hell

	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	

docker

An inspiration and some ancient history!



Intermodal shipping containers



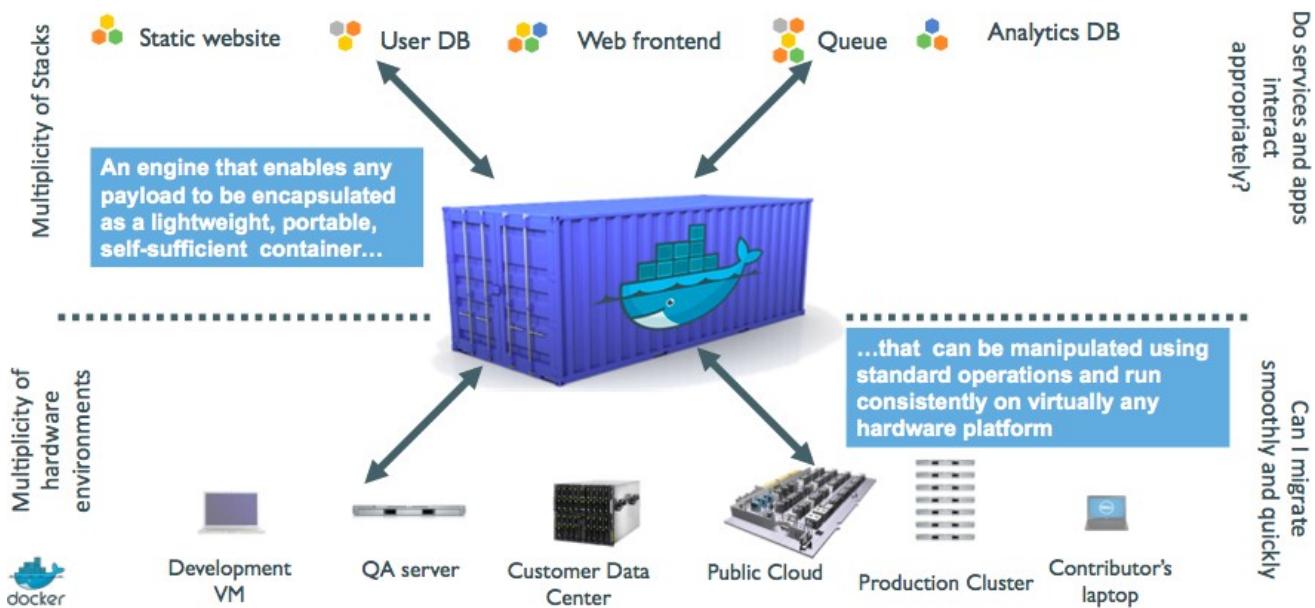
This spawned a Shipping Container Ecosystem!



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



A shipping container system for applications



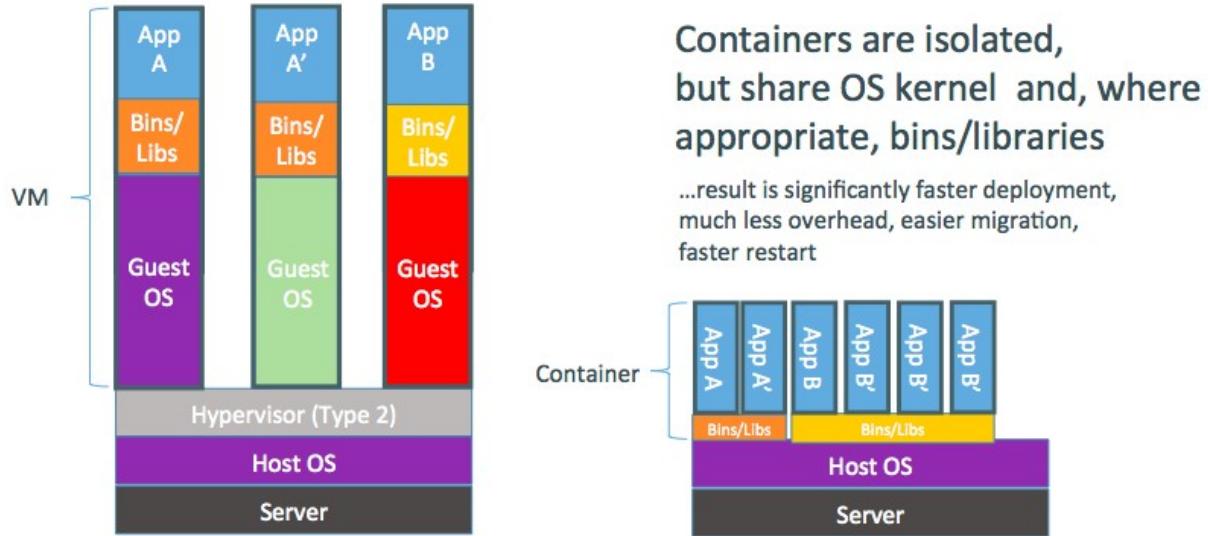
Eliminate the matrix from Hell

	Static website							
	Web frontend							
	Background workers							
	User DB							
	Analytics DB							
	Queue							
		Developmen t VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
								

Docker high-level roadmap

- Step 1: containers as lightweight VMs
- Step 2: commoditization of containers
- Step 3: shipping containers efficiently
- Step 4: containers in a modern software factory

Step 1: containers as lightweight VMs



- This drew attention from hosting and PaaS industry.
- Highly technical audience with strong ops culture.

Step 2: commoditization of containers

Container technology has been around for a while. (c.f. LXC, Solaris Zones, BSD Jails, LPAR...)

So what's new?

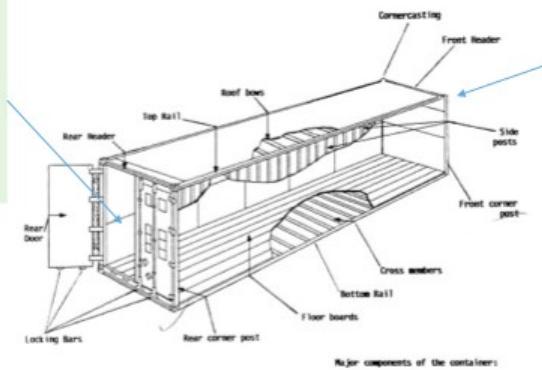
- Standardize the container format, because containers were not portable.
- Analogy:
 - shipping containers are not just steel boxes
 - they are steel boxes that are a standard size, with the same hooks and holes
- Make containers easy to use for developers.
- Emphasis on re-usable components, APIs, ecosystem of standard tools.
- Improvement over ad-hoc, in-house, specific tools.

Running containers everywhere

- Maturity of underlying technology (cgroups, namespaces, copy-on-write systems).
- Ability to run on any Linux server today: physical, virtual, VM, cloud, OpenStack...
- Ability to move between any of the above in a matter of seconds-no modification or delay.
- Ability to share containerized components.
- Self contained environment - no dependency hell.
- Tools for how containers work together: linking, discovery, orchestration...

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



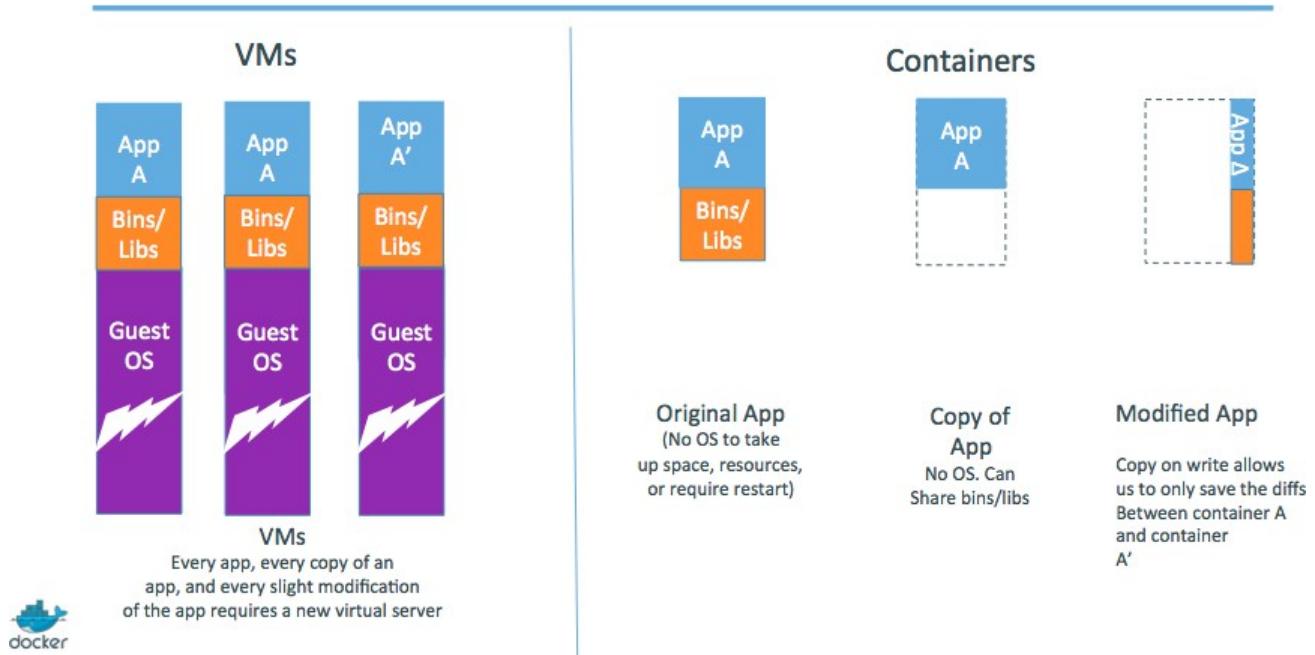
- Oscar the Ops Guy
 - Worries about what's "outside" the container
 - Logging
 - Remote access
 - Monitoring
 - Network config
 - All containers start, stop, copy, attach, migrate, etc. the same way



Step 3: shipping containers efficiently

Ship container images, made of reusable shared layers.

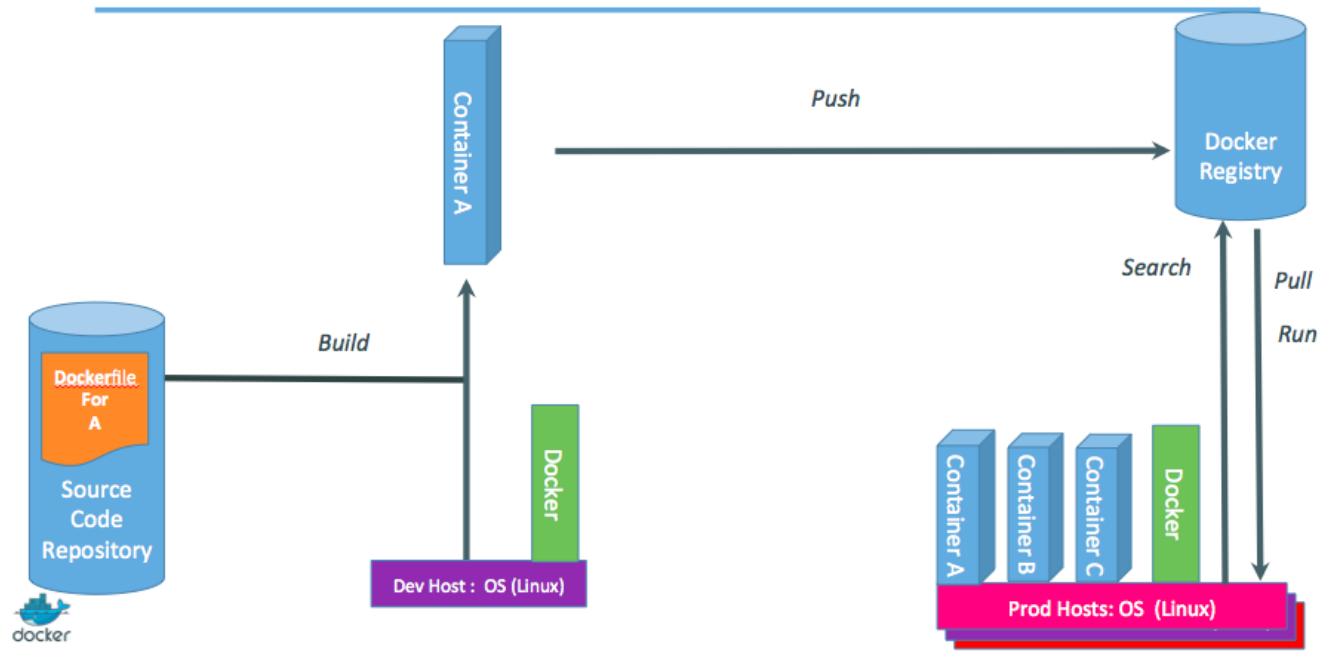
Optimizes disk usage, memory usage, network usage.



Step 4: containers in a modern software factory

The container becomes the new build artefact.

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



Docker architecture

Docker is a client-server application.

- **The Docker daemon**

Receives and processes incoming Docker API requests.

- **The Docker client**

Command line tool - the dockerbinary.

Talks to the Docker daemon via the Docker API.

- **Docker Hub Registry**

Public image registry.

The Docker daemon talks to it via the registry API.

Your training Virtual Machine



Lesson 2: 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

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.

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.

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

Connecting to your Virtual Machine

You need an SSH client.

- On OS X, Linux, and other UNIX systems, just use ssh:
- On Windows, if you don't have an SSH client, you can download Putty from www.putty.org.

```
$ docker version
Client
version: 1.4.1
Client API version: 1.16
Go version (client): go1.3.3 Git commit
(client): 5bc2ff8 OS/Arch (client): linux/
amd64 Server version: 1.4.1
Server API version: 1.16
Go version (server): go1.3.3 Git commit
(server): 5bc2ff8
```

Checking your Virtual Machine

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

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

Install

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).

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

Installing Docker on your Linux distribution

On Fedora:

On CentOS 7:

On Debian and derivatives:

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

Installation script from Docker

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

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.

To install Docker on these platforms we run a small virtual machine using a tool called [Boot2Docker](#).



```
$ docker version
Client
version: 1.5.0
Client API version: 1.17
Go version (client): go1.4.1 Git commit
(client): a8a31ef OS/Arch (client): linux/
amd64 Server version: 1.5.0
Server API version: 1.17
Go version (server): go1.4.1 Git commit
(server): a8a31ef
```

Check that Docker is working

Using the dockerclient:

Install

Su-su-sudo



The dockergroup

Warning

!

The dockeruser is rootequivalent.

It provides root-level access to the host.

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

Add the Docker group

Add ourselves to the group

Restart the Docker daemon

Log out

```
$ docker version
Client
version: 1.5.0
Client API version: 1.17
Go version (client): go1.4.1 Git commit
(client): a8a31ef OS/Arch (client): linux/
amd64 Server version: 1.5.0
Server API version: 1.17
Go version (server): go1.4.1 Git commit
(server): a8a31ef
```

Check that Docker works without sudo

Section summary

We've learned how
to:

- Install Docker.
- Run Docker without sudo.

Our First Containers



Overview: Our First Containers

Objectives

At the end of this lesson, you will have:

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

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

Hello World

In your Docker environment, just run the following command:

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.

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

A more useful container

Let's run a more exciting container:

- This is a brand new container.
- It runs a bare-bones, no-frills ubuntusystem.

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

Do something in our container

Try to run curl in our container.

Told you it was bare-bones!

Let's check how many packages are installed here.

- `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.

```
# curl ifconfig.me/ip  
64.134.229.24
```

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

Install a package in our container

We want curl, so let's install it:

Answer Y or just press Enter.

One minute later, curl is installed!

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

Exiting our container

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

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

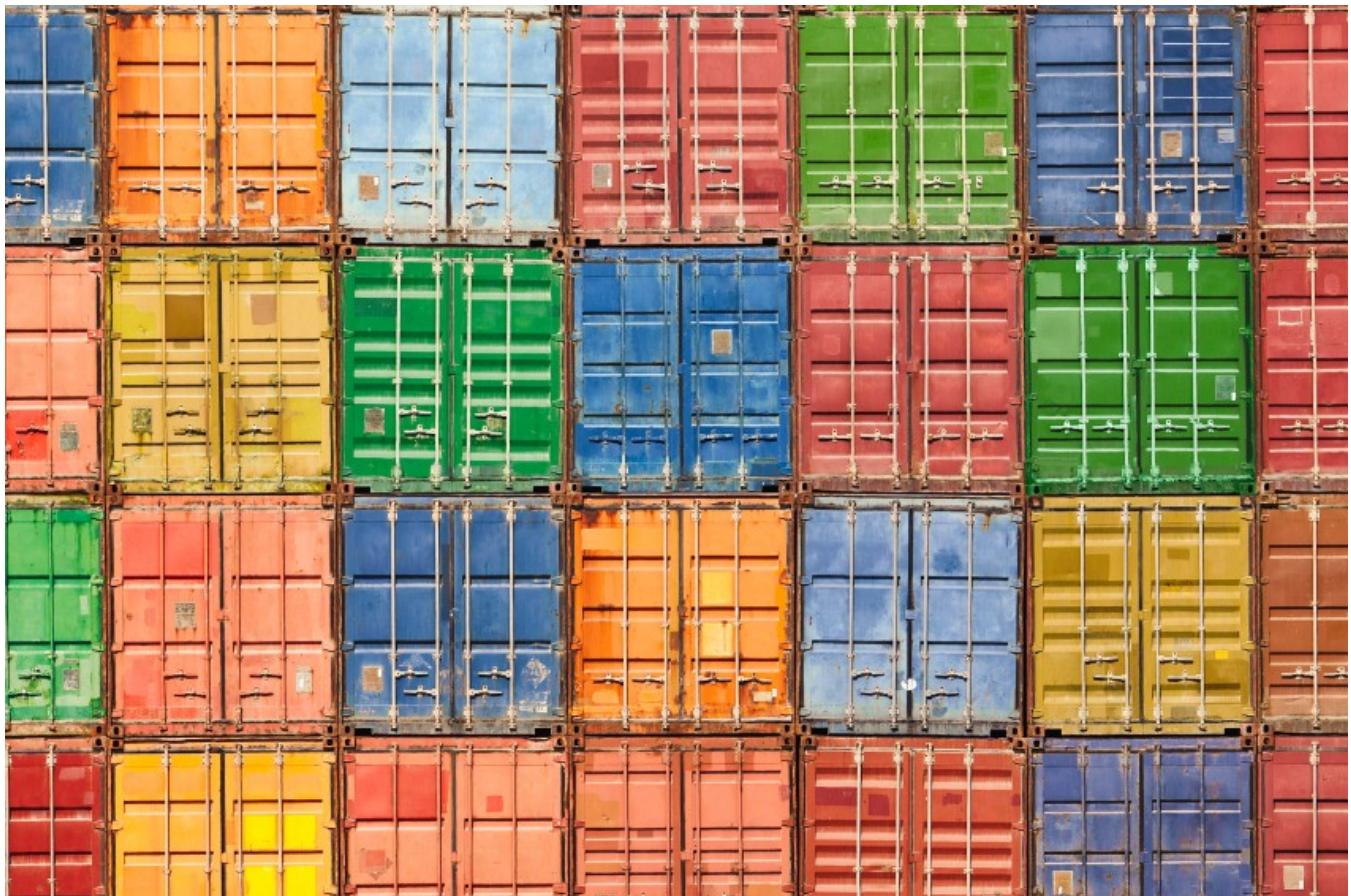
```
$ docker run -it ubuntu bash
root@b13c164401fb:/# curl bash: curl:
command not found
```

Starting another container

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

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

Background Containers



Background Containers

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.

```
$ 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  
...
```

A non-interactive container

We will run a small custom container.

This container just displays the time every second.

- 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.

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

Run a container in the background

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

- 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.

\$ docker ps CONTAINER					
ID	COMMAND	STATUS	...		
47d677dcfba4	ipetazzo/clock:latest	... 2 minutes ago	Up 2 minutes	...	

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 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.

```
$ docker ps -q  
47d677dcfba4
```

```
$ docker ps -q
```

Two useful flags for docker ps

To see only the last container that was started:

To see only the ID of containers:

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

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

View the logs of a container

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

- 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.)

```
$ 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
```

View only the tail of the logs

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

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

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

Follow the logs in real time

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

- 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.

```
$ docker ps
```

Killing it

Let's kill our container:

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

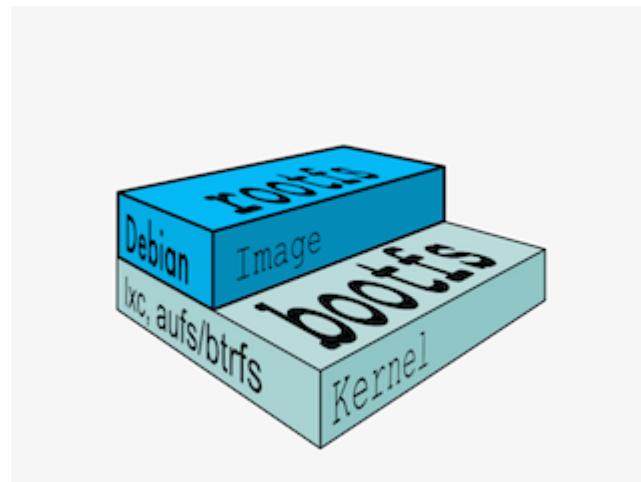
Docker will echo the ID of the container we've just stopped. Let's check that our container doesn't show up anymore:

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
47d677dcfba4      jpetazzo/clock:latest   "..."              23 min. ago       Exited (0) 4 min. ago
5c1dfd4d81f1      jpetazzo/clock:latest   "..."              40 min. ago       Exited (0) 40 min. ago
```

List stopped containers

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

Understanding Docker Images



Lesson 6: Understanding Docker Images

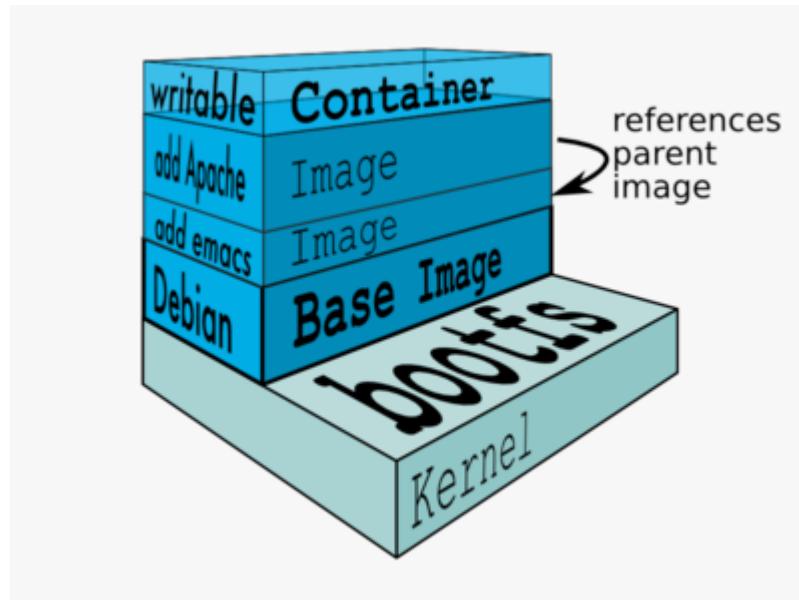
Objectives

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 runstarts 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.

In practice

There are multiple ways to create new images.

- docker commit: creates a new layer (and a new image) from a container.
- docker build: performs a repeatable build sequence.
- docker import: loads a tarball into Docker, as a standalone base layer.

We will explain commit and build in later chapters.

import can be used for various hacks, but its main purpose is to bootstrap the creation of base images.

Images namespaces

There are three namespaces:

- Root-like
- User (and organizations)
- Self-Hosted

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...

jpetazzo/clock

User namespace

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

The Docker Hub user is:

The image name is:

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:

The remote host and port is:

The image name is:

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.

```
$ docker images
```

REPOSITORY

TA

IMAGE ID

CREATE

VIRTUAL SIZE

Let's look at the output:

ubuntu	13.10	9f676bd305a4	7	wee ks	ag o	178 MB
ubuntu	saucy	9f676bd305a4	7	wee ks	ag o	178 MB
ubuntu	raring	eb601b8965b 8	7	wee ks	ag o	166.5 MB
ubuntu	13.04	eb601b8965b 8	7	wee ks	ag o	166.5 MB
ubuntu	12.10	5ac751e8d62 3	7	wee ks	ag o	161 MB
ubuntu	quantal	5ac751e8d62 3	7	wee ks	ag o	161 MB
ubuntu	10.04	9cc9ea5ea54 0	7	wee ks	ag o	180.8 MB
ubuntu	lucid	9cc9ea5ea54 0	7	wee ks	ag o	180.8 MB
ubuntu	12.04	9cd978db300 e	7	wee ks	ag o	204.4 MB
ubuntu	latest	9cd978db300 e	7	wee ks	ag o	204.4 MB
ubuntu	precise	9cd978db300 e	7	wee ks	ag o	204.4 MB

See

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

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

Downloading images

There are two ways to download images.

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

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

Pulling an image

- 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 variants.
- docker pull ubuntu will refer to ubuntu:latest.
- The :latest tag can be updated frequently.
- When using images it is always best to be specific.

Section summary

We've learned how
to:

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

Building Images Interactively



Lesson 7: Building Images Interactively

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

We will install software manually in a container, and turn it into a new image.

We will introduce 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 wget. First, we will do it manually with docker commit.

Then, in an upcoming chapter, we will use a Dockerfile and dockerbuild.

Building from a base

Our base will be the `ubuntuimage`.

If you prefer `debian`, `centos`, or `fedora`, feel free to use them instead. (You will have to adapt `apt` to `yum`, of course.)

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

Create a new container and make some changes

Start an Ubuntu container:

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

Then run the command apt-get install -y wget to install the program we are interested in.

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

Inspect the changes

Type exitat 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 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.)

```
$ docker run -it <newImageId> bash  
root@fcfb62f0bfde:/# wget  
wget: missing URL  
...
```

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.

The output of the dockercommitcommand will be the ID for your newly created image.

We can run this image:

```
$ docker run -it mydistro bash
```

```
$ docker tag <newImageId> mydistro
```

Tagging images

Referring to an image by its ID is not convenient. Let's tag it instead. We can use the tag command:

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

And then run it using its tag:

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 8: Building Docker Images

Objectives

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.

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

Writing our first Dockerfile

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

1. Create a directory to hold our Dockerfile.
2. Create a Dockerfile inside this directory.

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

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget
```

Type this into our Dockerfile...

- `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.)

Build it!

Save our file, then execute:

```
$ docker build -t myimage .
```

- `-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.)

```
$ docker build -t myimage .
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 -y wget
--> Running in 2b44df762a2f
--> f9e8f1642759
Removing intermediate container 2b44df762a2f Successfully built
```

What happens when we build the image?

The output of docker build looks like this:

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

Sending the build context to Docker

- The build context is the . directory given to dockerbuild.
- 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.

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

Executing each step

- A container (840cb3533193) is created from the base image.
- The RUNcommand 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.

```
$ docker run -ti myimage bash  
root@91f3c974c9a1:/# wget wget:  
missing URL
```

Running the image

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

- Sweet is the taste of success!

IMAGE	CREATED	CREATED	SIZE
f9e8f1642759	About an hour ago	/bin/sh -c apt-get install -y	6.062 MB
e54ca5efa2e9	8 months ago	/bin/sh -c apt-get update &&	8.549 MB
6c37f792ddac	8 months ago	/bin/sh -c apt-get update &&	8 B
83ff768040a0	8 months ago	/bin/sh -c sed -i s/'#\s*\d	83.43 MB
2f4b4d6a4a06	8 months ago	/bin/sh -c echo #!/bin/sh >	1.903 kB
d7ac5e4f1812	8 months ago	/bin/sh -c #(nop) ADD file:ad	194.5 kB
			192.5 MB

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.

CMD and ENTRYPOINT



Lesson 9: 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.

```
wget -O- -q http://ifconfig.me/ip
```

Defining a default command

When people run our container, we want to automatically execute wget to retrieve our public IP address, using ifconfig.me.

For that, we will execute:

- -O-tells wget to output to standard output instead of a file.
- -qtells wget to skip verbose output and give us only the data.
- http://ifconfig.me/ips is the URL we want to retrieve.

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget
CMD wget -O- -q http://ifconfig.me/ip
```

Adding CMDto our Dockerfile

Our new Dockerfile will look like this:

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

```
$ docker run ifconfigme 64.134.229.24
```

Build and test our image

Let's build
it:

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

And run it:

```
$ docker run -it ifconfigme bash root@7ac86a641116:/#
```

Overriding CMD

If we want to get a shell into our container (instead of running wget), we just have to specify a different program to run:

- We specified bash.
- It replaced the value of CMD.

```
$ docker run ifconfigme http://ifconfig.me/ua Wget/1.12 (linux-gnu)
```

Using ENTRYPPOINT

We want to be able to specify a different URL on the command line, while retaining wget and some default parameters.

In other words, we would like to be able to do this:

We will use the ENTRYPPOINT verb in Dockerfile.

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget ENTRYPOINT
["wget", "-O-", "-q"]
```

Adding ENTRYPOINT to our Dockerfile

Our new Dockerfile will look like this:

- 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.

```
$ docker run ifconfigme http://ifconfig.me/ua Wget/1.12 (linux-gnu)
```

Build and test our image

Let's build it:

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

And run it:

Great success!

Using CMDand ENTRYPOINTtogether

What if we want to define a default URL for our container? Then we will use ENTRYPOINTand CMD together.

- ENTRYPOINTwill define the base command for our container.
- CMDwill define the default parameter(s) for this command.

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget ENTRYPOINT
["wget", "-O-", "-q"]
CMD http://ifconfig.me/ip
```

CMDand ENTRYPOINTtogether

Our new Dockerfile will look like this:

- ENTRYPOINTdefines 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 CMDis appended.
- Otherwise, our extra command-line arguments are used instead of CMD.

```
$ docker run ifconfigme 64.134.229.24  
$ docker run ifconfigme http://ifconfig.me/ua Wget/1.12 (linux-gnu)
```

Build and test our image

Let's build it:

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

And run it:

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

Overriding ENTRYPOINT

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

We cannot just do docker run ifconfigme bash because that would try to fetch the URL bash(which is not a valid URL, obviously).

We use the --entrypoint parameter:

Container Networking Basics



Lesson 10: Container Networking Basics

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 network model used by Docker.

```
$ docker run -d -P jpetazzo/web 66b1ce719198711292c8f34f84a7b68c3876cf9f67015e752b94e189d35a204e
```

A simple, static web server

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

- 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?

```
$ docker ps CONTAINER
ID          IMAGE
66b1ce719198  ipetazzo/web:latest
```

Finding our web server port

We will use docker ps:

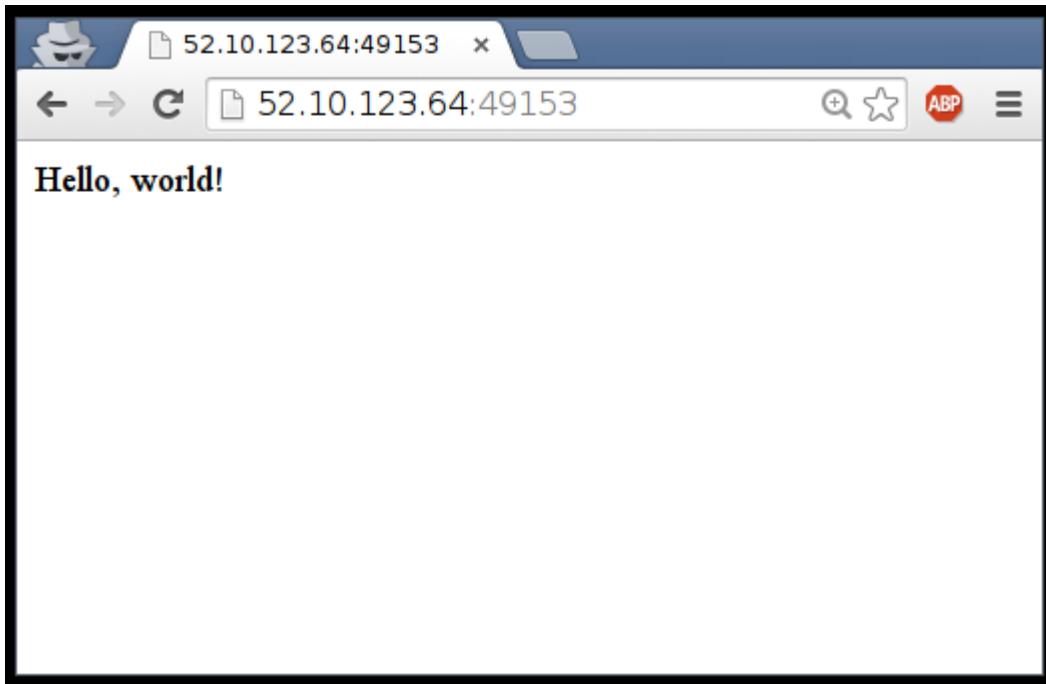
- The web server is running on port 8000 inside the container.
- That port is exposed on port 49153 on our Docker host.

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.



```
$ curl localhost:49153 Hello, world!
```

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:

Docker network model

- 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> 8000 49153
```

```
$ docker run -t -p 80:8000 jpetazzo/web
```

Manual allocation of port numbers

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

- This time, we are running our container in the foreground. (That way, we can kill it easily with ^C.)
- We mapped port 80 on the host, to port 8000 in the 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...

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' <yourContainerID> 172.17.0.3
```

Finding the container's IP address

We can use the docker inspect command to find the IP address of the container.

- 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.

```
$ 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
```

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 pingtool.

Section summary

We've learned how to:

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

NOTE: Later on we'll see how to network containers without exposing ports using the link primitive.

Local Development Work flow with Docker



Lesson 11: 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.

```
$ docker pull training/namer
```

Using a Docker container for local development

Docker containers are perfect for local development.

Let's grab an image with a web application and see how this works.

Our namer image

Our training/namer image is based on the Ubuntu image. It contains:

- Ruby.
- Sinatra.
- Required dependencies.

```
$ git clone https://github.com/docker-training/namer.git  
$ cd namer  
$ ls  
company_name_generator.rb config.ru Dockerfile Gemfile README.md
```

Adding our source code

Let's download our application's source code.

```
$ docker run -d \
-v $(pwd):/opt/namer \
-p 80:9292 \
training/namer
```

Creating a container from our image

We've got an image, some source code and now we can add a container to run that code.

- 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.
- `training/namer` is the name of the image we will run.

More on these later.

We've launched the application with the `training/namer` image and the `rackup` command. `rackup` has been set as the `CMD` in the Dockerfile.

Mounting volumes inside containers

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

- If [host-path] or [container-path] doesn't exist it is created.
- You can control the write status of the volume with the roand rwoptions.
- If you don't specify rwor ro, it will be rwby default.

```
$ docker ps CONTAINER  
ID          COMMAND      STATUS  
PORTS      NAMES  
045885b68bc5  training/namer:latest backup      3 seconds ago Up 3 seconds
```

Checking our new container

Now let us see if our new container is running.

Viewing our application

Now let's browse to our web application on:

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.

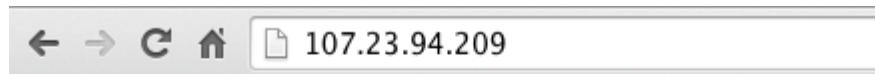
And change

To:

Refreshing our application

Now let's refresh our browser:

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



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 -vflag 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.

```
$ # 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
```

docker exec example

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

Stopping the container

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

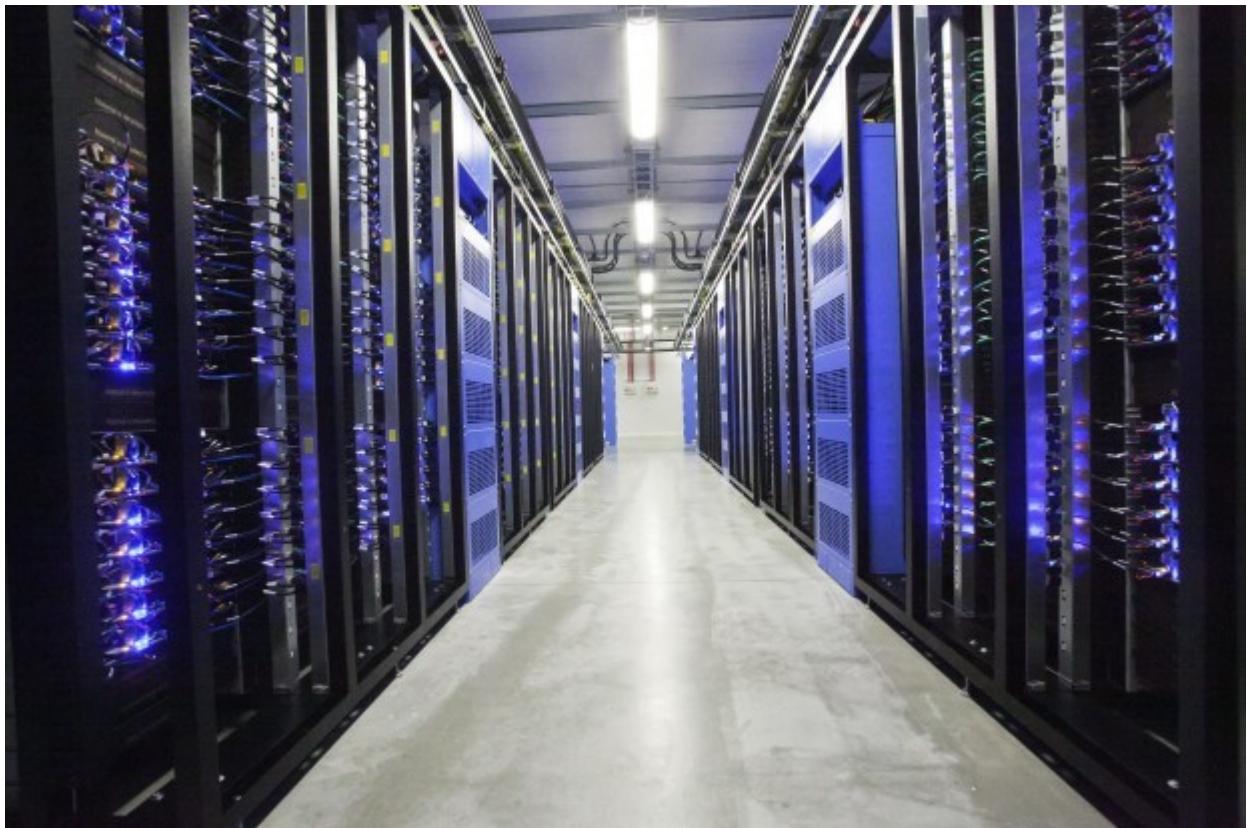
And remove it.

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 12: 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 dockercommit.
- 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.

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

Volumes are special directories in a container

Volumes can be declared in two different ways.

- Within a Dockerfile, with a VOLUME instruction.
- On the command-line, with the -v flag for docker run.

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 either.

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

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

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 `dockerrun`.

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

Volumes exist independently of containers

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

In the last exemple, it doesn't matter if container alphais running or not.

```
$ docker run --name wwwdata -v /var/lib/www busybox true  
$ docker run --name wwwlogs -v /var/log/www busybox true
```

Data containers

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:

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

Try it out!

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 wwwdata --volumes-from wwwlogs webserver  
$ docker run -d --volumes-from wwwdata ftpserver  
$ docker run -d --volumes-from wwwlogs logstash
```

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

Managing volumes yourself (instead of letting Docker do it)

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

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

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

```
$ docker run -it -v $(pwd)/bindthis:/var/www/html/webapp:ro ubuntu bash  
$ cd  
$ mkdir bindthis  
$ ls bindthis  
$ docker run -it -v $(pwd)/bindthis:/var/www/html/webapp ubuntu bash root@<yourContainerID>:# touch /var/www/html/webapp/index.html root@<yourContainerID>:# exit
```

Sharing a directory between the host and a container

This will mount the bindthisdirectory into the container at /var/www/html/webapp.

Note that the paths must be absolute.

It defaults to mounting read-write but we can also mount read-only.

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

```
root@<yourContainerID># exit
```

Chaining container volumes together

Let's see how to put both pieces together.

1. Create an initial container.
2. Create some data in our data volume.
3. Exit the container.

```
root@<yourContainerID># exit
```

Use a data volume from our container.

1. Create a new container.
2. Let's view our data.
3. Let's make a change to our data.
4. Exit the container.

```
$ docker rm -v appdata appserver1 appserver2
```

```
root@179c063af97a# cat/var/appvolume/data Hello
```

Chain containers with data volumes

1. Create a third container.
2. Let's view our data.
3. Exit the container.
4. Tidy up your containers.

What happens when you remove containers with volumes?

- As long as a volume is referenced by at least one container, you will be able to access it.
- When you remove the last container referencing a volume, that volume will be orphaned.
- Orphaned volumes are not deleted (as of Docker 1.2).
- The data is not lost, but you will not be able to access it.
(Unless you do some serious archeology in /var/lib/docker.)

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

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

Checking volumes defined by an image

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

```
$ docker inspect <yourContainerID> [{
  "ID": "<yourContainerID>",
  ...
  "Volumes": {
    "/var/webapp": "/var/lib/docker/vfs/dir/
f4280c5b6207ed531efd4cc673ff620cef2a7980f747dbbccca001db61de04468"
  },
  "VolumesRW": { "/var/webapp": true
  },
}]
}
```

Checking volumes used by a container

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

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

```
$ docker run -it -v /var/run/docker.sock:/docker.sock ubuntu bash  
$ echo 4815162342 > /tmp/numbers  
$ docker run -it -v /tmp/numbers:/numbers ubuntu bash
```

Sharing a single file between the host and a container

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

- All modifications done to `/numbers` in the container will also change `/tmp/numbers` on the host!

It can also be used to share a *socket*.

- This pattern is frequently used to give access to the Docker socket to a given container.

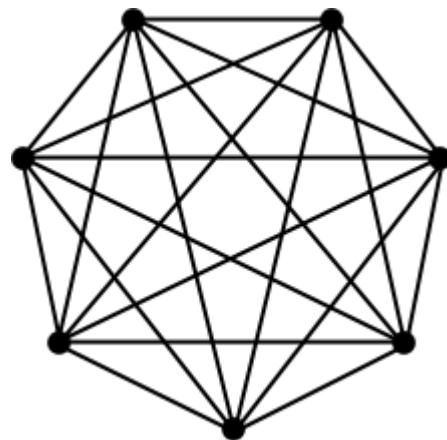
Warning: when using such mounts, the container gains root-like access to the host. It can potentially do bad things.

Section summary

We've learned how
to:

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

Connecting Containers



Lesson 13: Connecting containers

Objectives

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

- Launch named containers.
- 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.

```
$ docker images redis
```

REPOSITORY	IMAGE ID	CREATED
VIRTUAL SIZE	b73cdc045d3c	2 weeks ago
redis		

Our Redis database image

Let's start by pulling down our database image.

- There are multiple versions of the redisimage.
- To save time and network resources, we will pull only the one we need (latest).

To do that, we will specify the exact version tag to be used.

Let's review the result.

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

Launch a container from the redisimage.

Let's launch a container from the redisimage.

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.

```
$ docker images nathanleclaire/redisonrails
```

Our Rails application image

Let's start by pulling down our Rails application image.

And reviewing it.

```
FROM ruby
RUN apt-get update -qq && apt-get install -y build-essential libpq-dev
WORKDIR /myapp
ADD Gemfile /myapp/Gemfile
RUN bundle install -j8 EXPOSE 3000
ADD . /myapp
CMD ["bundle", "exec", "rails", "s"]
```

The nathanleclaire/redisonrailsDockerfile

Let's look at the Dockerfile that builds this image.

The nathanleclaire/redisonrailsDockerfile in detail

- Based on the rubybase image from Docker Hub (provided by Docker Inc.)
- Installs the required packages with bundle install.
- Adds the Rails application itself to the /myappdirectory.
- Exposes port 3000.
- Runs Ruby on Rails when a container is launched from the image.

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

Connecting to redis in the container

The following Ruby code will be used in /myapp/config/initializers/ redis.rb to connect to the running Redis container.

As we'll see in more detail later, Links provide a DNS entry for the linked container as well as information about how to connect (IP address, ports, etc.) populated in environment variables.

```
$ 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'
....
```

Launch a container from the nathanleclaire/redisonrailsimage.

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

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

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

```
$ 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>
```

Launch and link a container

Let's try again but this time we'll link our container to our existing Redis container.

Woot! That's more like it.

- The --linkflag 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 ActiveRecordclass to create data models that have the speed on in-memory lookups.

```
$ docker run --link mycache:redis nathanleclaire/redisondrails 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
```

More about our link - Environment variables

The link provides a secure tunnel between containers. On our mycachecontainer port 6379(the default Redis port) has been exposed to the linked container.

Docker will automatically set environment variables in our container, and populate an /etc/hostsentry for DNS lookup, to indicate connection information.

Let's see that information:

- Each variables is prefixed with the link alias: redis.
- Includes connection information PLUS any environment variables set in the mycachecontainer via ENVinstructions.

```
$ 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/
```

DNS

Links also provides you with a DNS entry corresponding to the name of the container, which is what we've used in this sample application.

```
$ docker ps -l
```

Starting our Rails application

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

Now let's check the container is running.

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

Starting our Rails application

Our homepage controller contains the following code.

Viewing our Rails application

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



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:

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

Advanced Dockerfiles



Lesson 14: Advanced 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:

A user image:

Or self-hosted image:

```
FROM ubuntu:14.04
```

```
...  
FROM fedora:20
```

```
...
```

More about FROM

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

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 MAINTAINERinstruction

The MAINTAINERinstruction tells you who wrote the Dockerfile.

It's optional but recommended.

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

The RUNinstruction

The RUNinstruction can be specified in two ways.

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

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

More about the RUNinstruction

RUNwill do the following:

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

RUNwill 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 CMDand/or ENTRYPPOINT.

The EXPOSE instruction

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

- 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.

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

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

The ADDinstruction

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

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 ADDinstruction can also be used to get remote files.

This would download the webappfile and place it in the /optdirectory.

More about the ADDinstruction

- 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 ADDinstruction are owned by rootwith permissions of 0755.

More on ADD[here](#).

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

The VOLUMEinstruction

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

- 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 WORKDIRinstruction

The WORKDIRinstruction sets the working directory for subsequent instructions.

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

You can specify WORKDIRagain to change the working directory for further operations.

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

The ENV instruction

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

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

You can also specify environment variables when you use dockerrun.

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.

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

The CMD instruction

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

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

Instead of:

We can just do:

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

More about the CMDinstruction

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

The second executes directly, without shell processing:

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

Overriding the CMDinstruction

The CMDcan be overridden when you run a container.

Will run bashinstead of nginx -g "daemon off;".

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

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 (["..."]).

If we were to run:

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

```
$ 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:/#
```

Overriding the ENTRYPOINT instruction

The entry point can be overridden as well.

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

How CMDand ENTRYPOINTinteract

The CMDand ENTRYPOINTinstructions work best when used together.

The ENTRYPOINTspecifies the command to be run and the CMDspecifies its options. On the command line we can then potentially override the options when needed.

This will override the options CMDprovided with new flags.

The ONBUILDinstruction

The ONBUILDinstruction 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.

- You can't chain ONBUILDinstructions with ONBUILD.
- ONBUILDcan't be used to trigger FROMand MAINTAINERinstructions.

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.
- ADD dependency lists (`package.json`, `requirements.txt`, etc.) by themselves to avoid reinstalling unchanged dependencies every time.

```
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
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
RUN pip install -qr requirements.txt EXPOSE 5000
CMD ["python", "app.py"]
```

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
ADD ./webapp/requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
EXPOSE 5000
```

Fixed Dockerfile

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

Container Orchestration



Lesson 15: Orchestration

Objectives

In this lesson, you will:

- Understand what orchestration is and why it is needed.
- Use fig, our recommended tool, to bootstrap a development environment.
- Understand some of the options and when you would use each one.

What is Orchestration?

- Orchestration is a loosely defined term that people use to describe the set of practices around managing multiple Docker containers.
- It can be as simple as using tools to ease the "wiring together" of multiple Docker containers (one for application, one for DB, one for key-value store etc.), or it can be as complex as the coordinating and scheduling of many resources and containers across large clusters of computers.
- For instance, what we did in the last section was very handy for learning, but it is difficult / tedious to remember and type in long docker run commands all the time. So we want a better, more automatic way.

Getting started with Orchestration

We're going to get our hands dirty with an orchestration tool officially endorsed by Docker. It's called Fig.



Fig

With fig, we define a set of set of containers to boot up, and their runtime properties, in a YAML file. Then we run fig up, and watch as fig boots the containers, assigns the appropriate links, and multiplexes the logs' outputs together.



```
sudo pip install -U fig
```

Installation

To install fig:

```
curl -L https://github.com/docker/fig/releases/download/0.5.2/linux \
> /usr/local/bin/fig chmod +x /
/usr/local/bin/fig
```

You can also use pip if you prefer:

```
cd  
git clone https://github.com/docker-training/simplefig cd simplefig
```

Gettin' Figgy With It

Clone the source code for the app we will be working on.

Gettin' Figgy With It

Create this
Dockerfile:

```
FROM python:2.7
ADD requirements.txt /code/requirements.txt WORKDIR /code
RUN pip install -r requirements.txt ADD . /code
```

```
web:  
  build: .  
  command: python app.py  
  ports:  
    - "5000:5000"  
  volumes:  
    - ./code  
  links:  
    - redis  
redis:  
  image: orchardup/redis
```

Gettin' Figgy With It

Now create a fig.yml to store the runtime properties of the app.

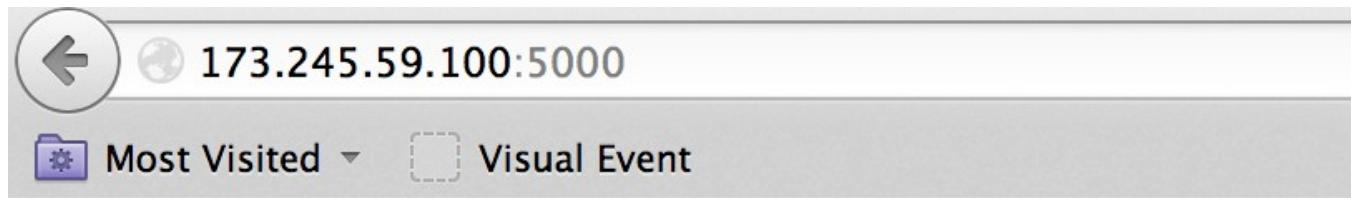
Gettin' Figgy With It

Run fig up in the directory, and watch fig build and run your app with the correct parameters, including linking the relevant containers together.



Gettin' Figgy With It

Verify that the app is running at <http://<yourHostIP>:5000>.



Hello Docker Training! I have been seen 8 times

Some more useful fig commands

fig introduces a unit of abstraction called a "service" (mostly, a container that interacts with other containers in some way and has specific runtime properties).

To rebuild all the services in your fig.yml:

To run fig up in the background instead of the foreground:

To see currently running services:

To remove the existing services:

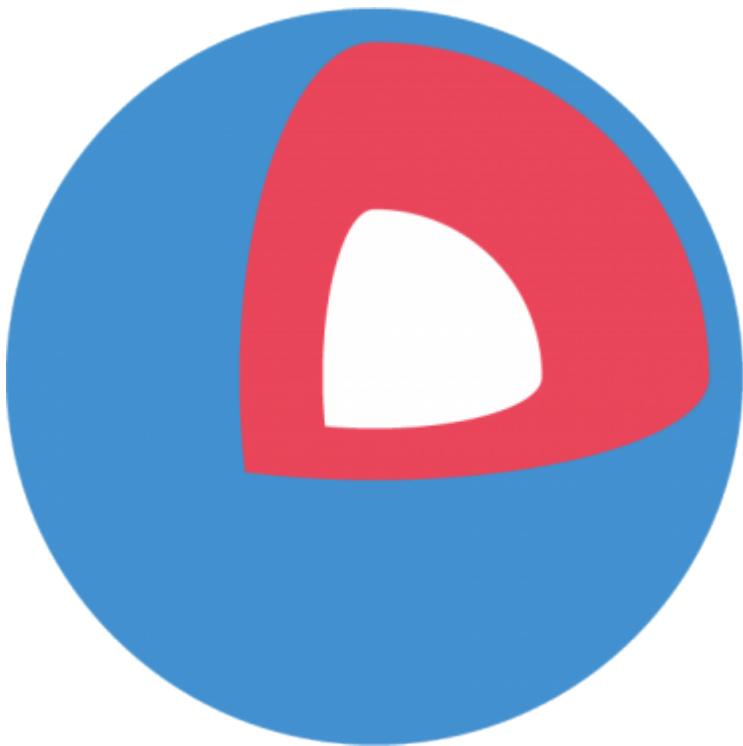
Other Orchestration Options

Now to rattle off some of the other options very quickly. Many of these have a different focus than fig, so they may be useful in different ways.



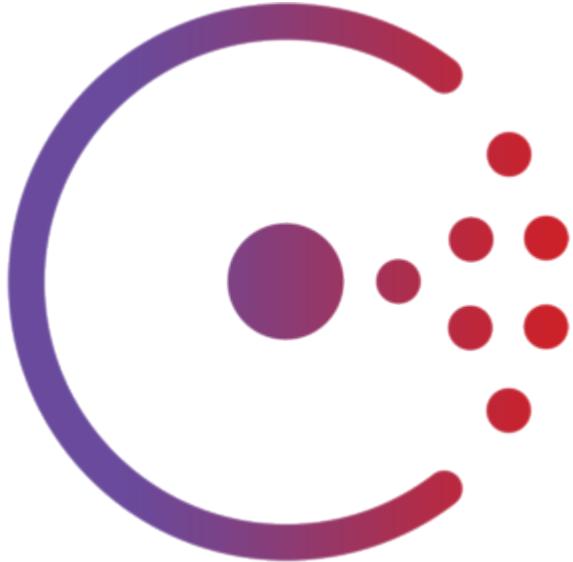
Fleet / etcd

Go - from CoreOS



Consul

Go - from HashiCorp



Kubernetes

Go - from Google



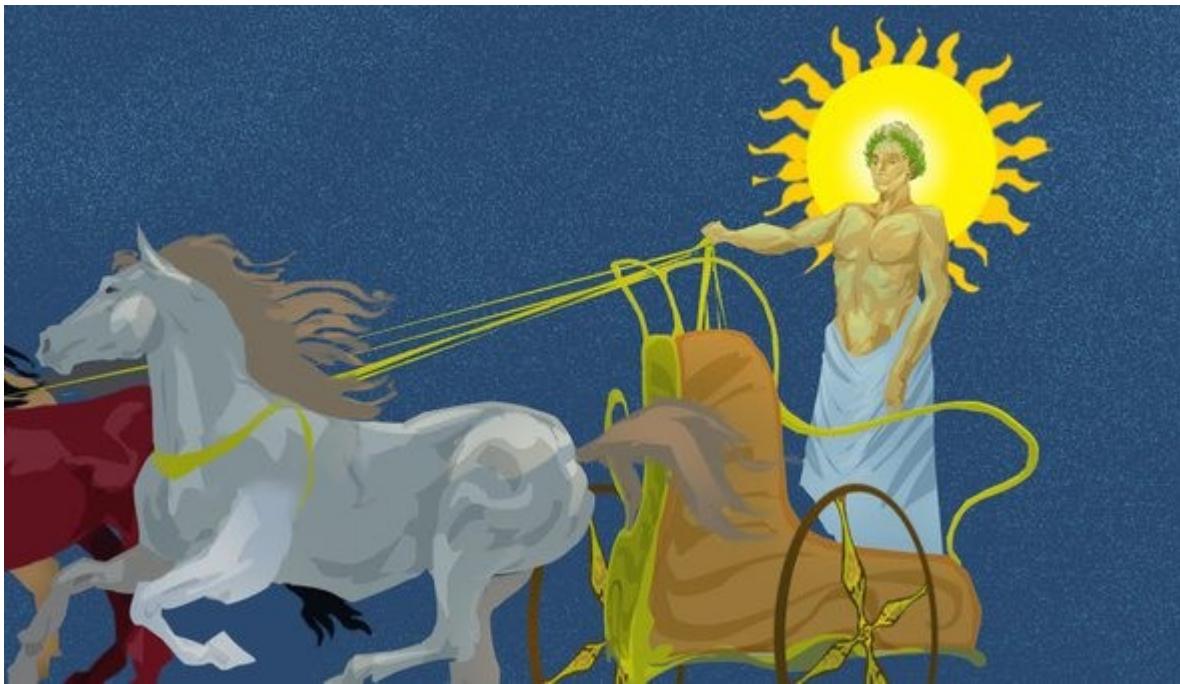
Mesos

C++ - from Apache/Twitter



Helios

Java - from Spotify



Centurion

Ruby - from New Relic



Future

There are many choices - the dream is to eventually be able to swap these choices out seamlessly using libswarm, an ongoing project Docker has around orchestration, and use whatever fits your needs.

<https://github.com/docker/libswarm>



Section Summary

We've learned how to:

- Understand what orchestration is and why it is needed.
- Use fig, our recommended tool, to bootstrap a development environment.
- Understand some of the other options and when you would use each one.

Ambassadors



Lesson 16: 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.

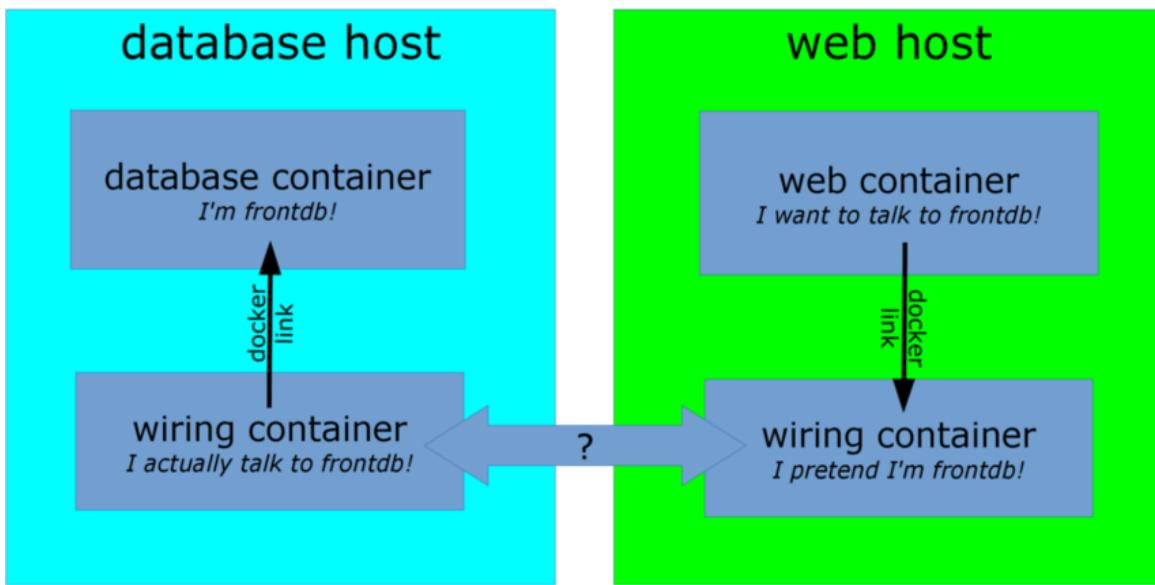
We're now going to see a pattern for service portability we call: ambassadors.

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.

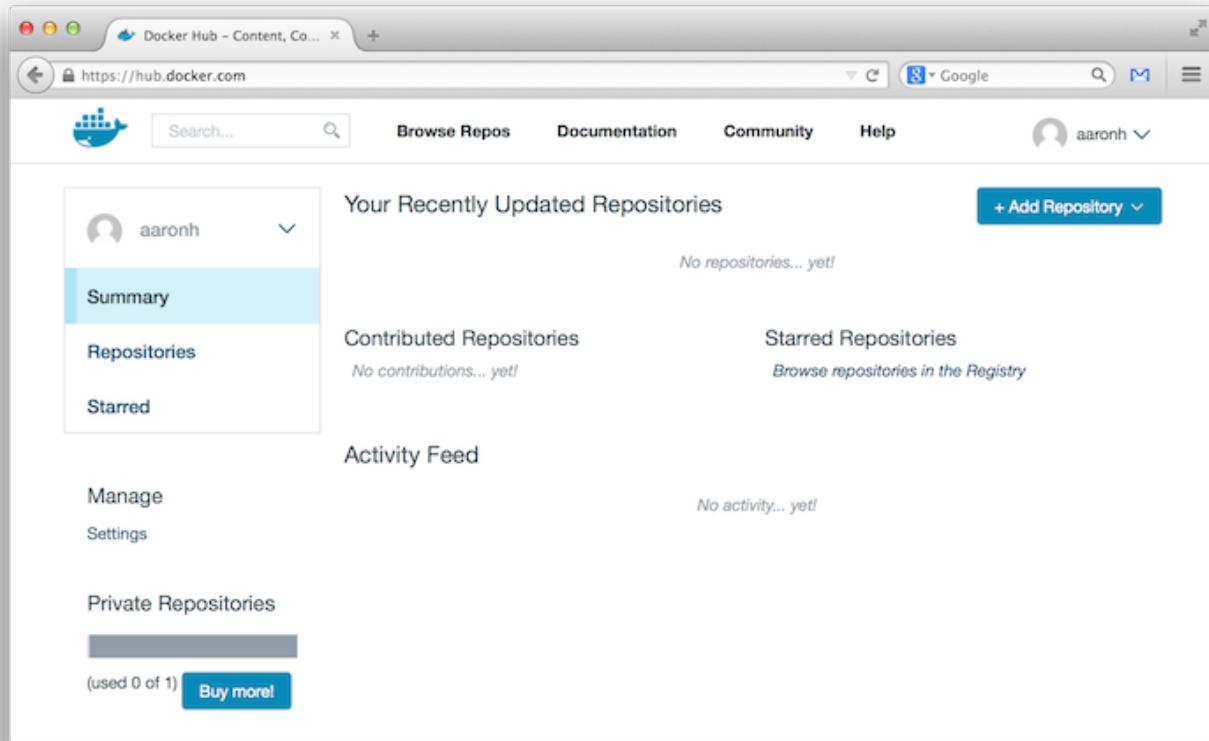
- On-premise deployments with a trusted network can track container locations in e.g. Zookeeper, and generate HAProxy configurations each time a location key changes.
- Public cloud deployments or deployments across unsafe networks can add TLS encryption.
- Ad-hoc deployments can use a master-less discovery protocol like avahi to register and discover services.

Section summary

We've learned how
to:

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

Introducing Docker Hub



Lesson 17: Introducing Docker Hub Objectives

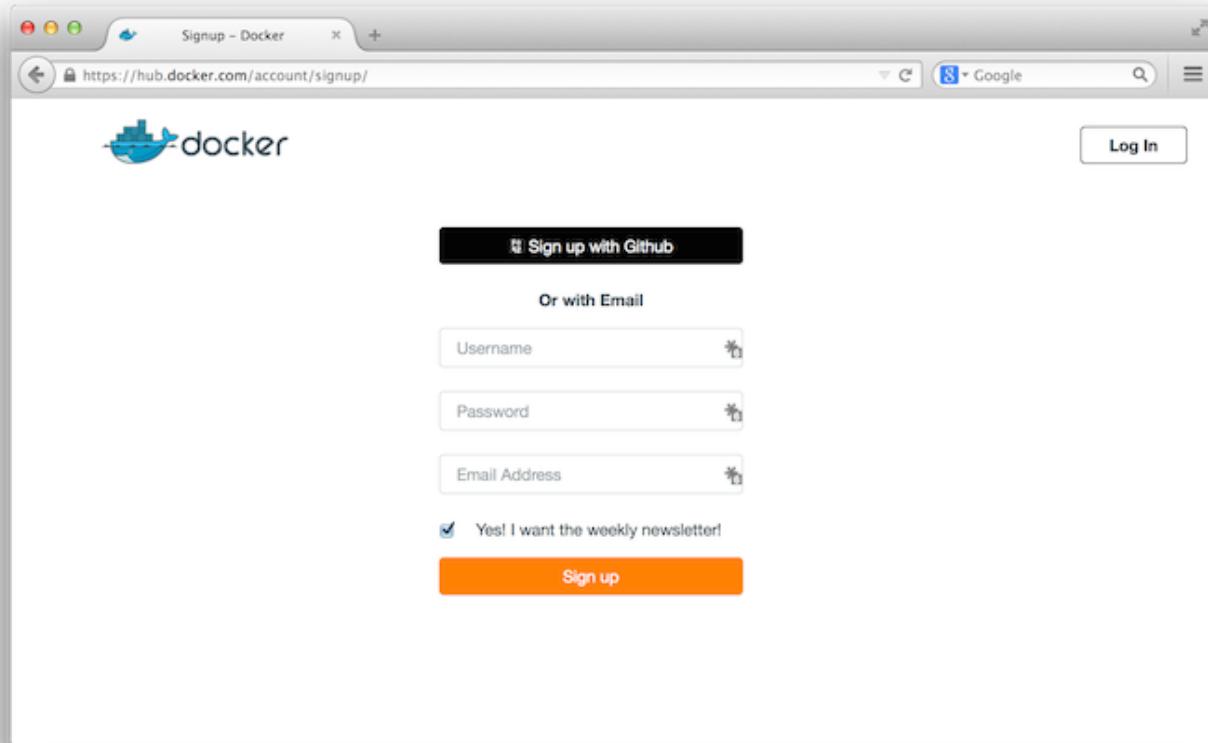
At the end of this lesson, you will be able to:

- Register for an account on Docker Hub.
- Login to your account from the command line.
- Learn about how Docker Hub works.
- Learn about how to integrate Docker Hub into your development workflow.

Sign up for a Docker Hub account

Note: if you already have an account on the Index/Hub, don't create another one.

- Having a Docker Hub account will allow us to store our images in the registry.
- To sign up, you'll go to [hub.docker.com](https://hub.docker.com/account/signup/) and fill out the form.
- Note: your Docker Hub username has to be all lowercase.



Activate your account through e-mail.

- Check your e-mail and click the confirmation link.

```
$ docker login  
Username: my_docker_hub_login  
Password:  
Email: my@email.com Login  
Succeeded
```

Login

Let's use our new account to login to the Docker Hub!

Our credentials will be stored in `~/.dockercfg`.

```
{  
  "https://index.docker.io/v1/":  
    { "auth":"amFtdHVyMDE6aTliMUw5ckE=",  
      "email":"education@docker.com"  
    }  
}
```

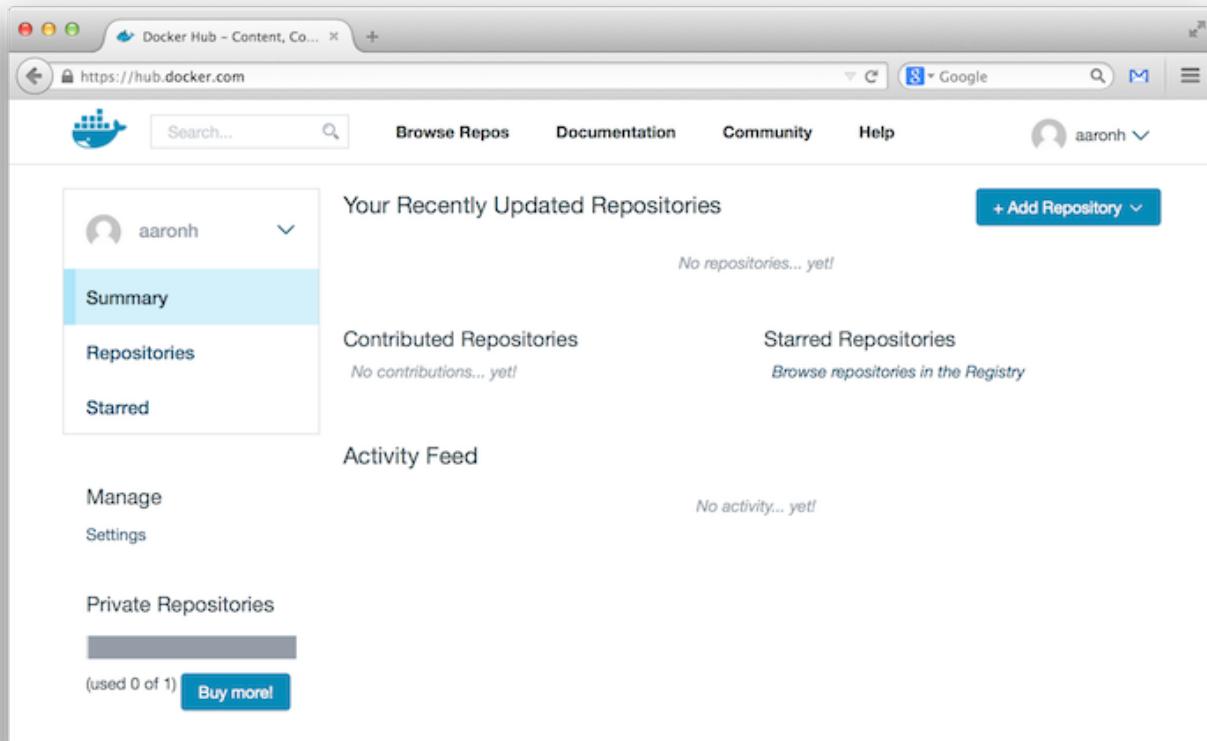
The .dockercfg configuration file

The `~/.dockercfg` configuration file holds our Docker registry authentication credentials.

The authsection is Base64 encoding of your user name and password. It should be owned by your user with permissions of 0600.

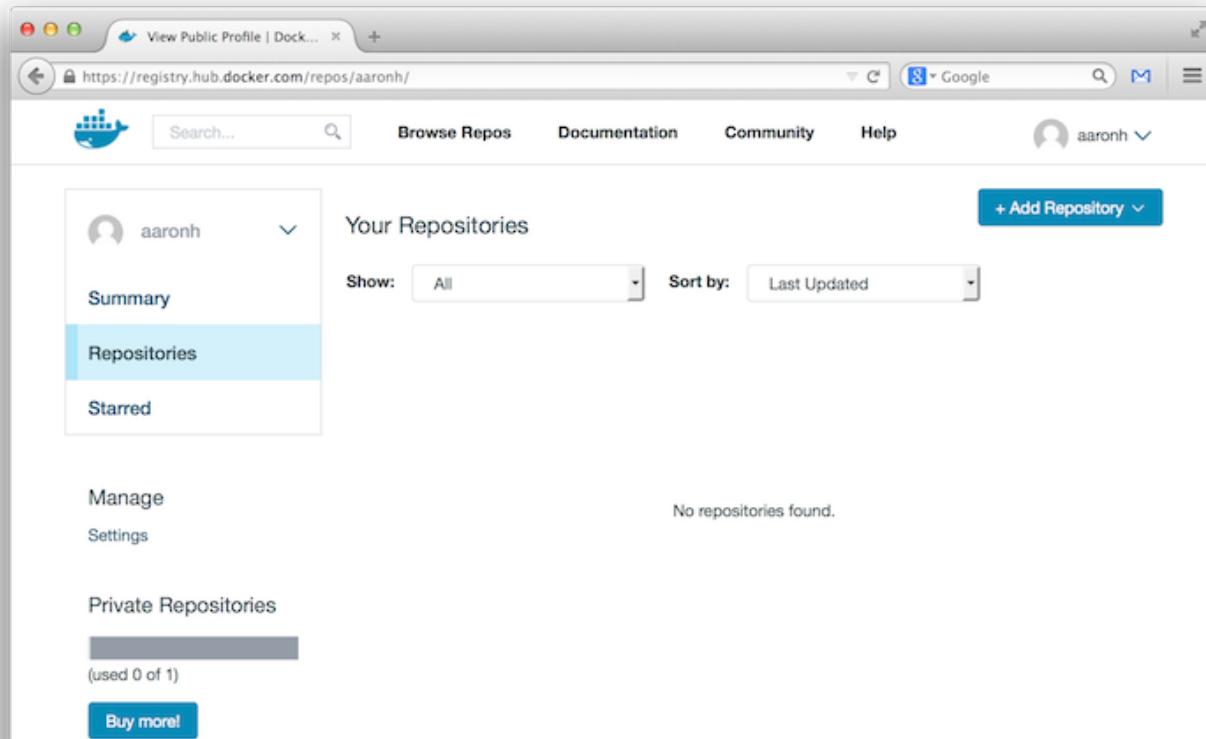
You should protect this file!

Navigating Docker Hub



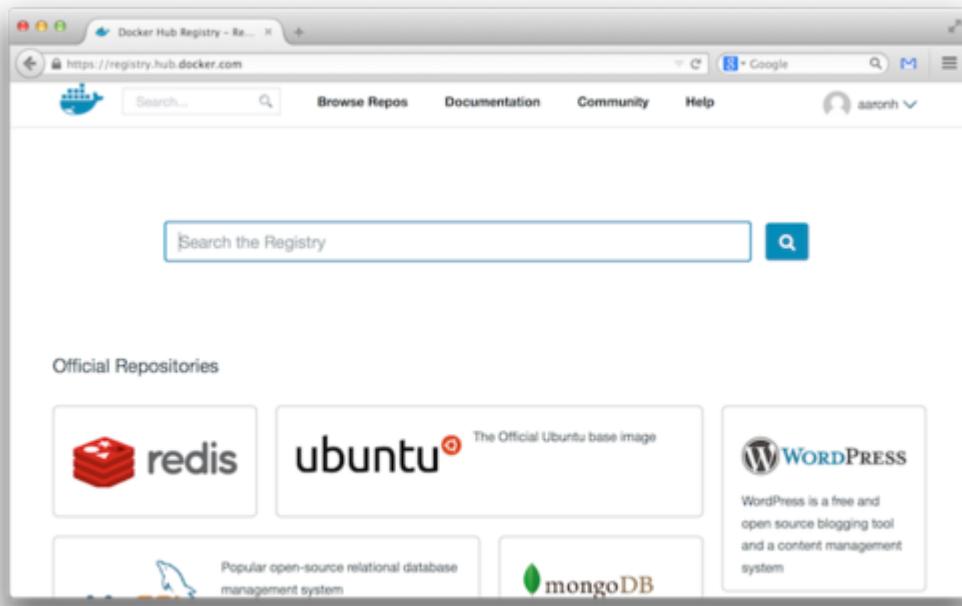
Repositories

- Store all public and private images in the registry
- Apply to your namespace
- Empty!



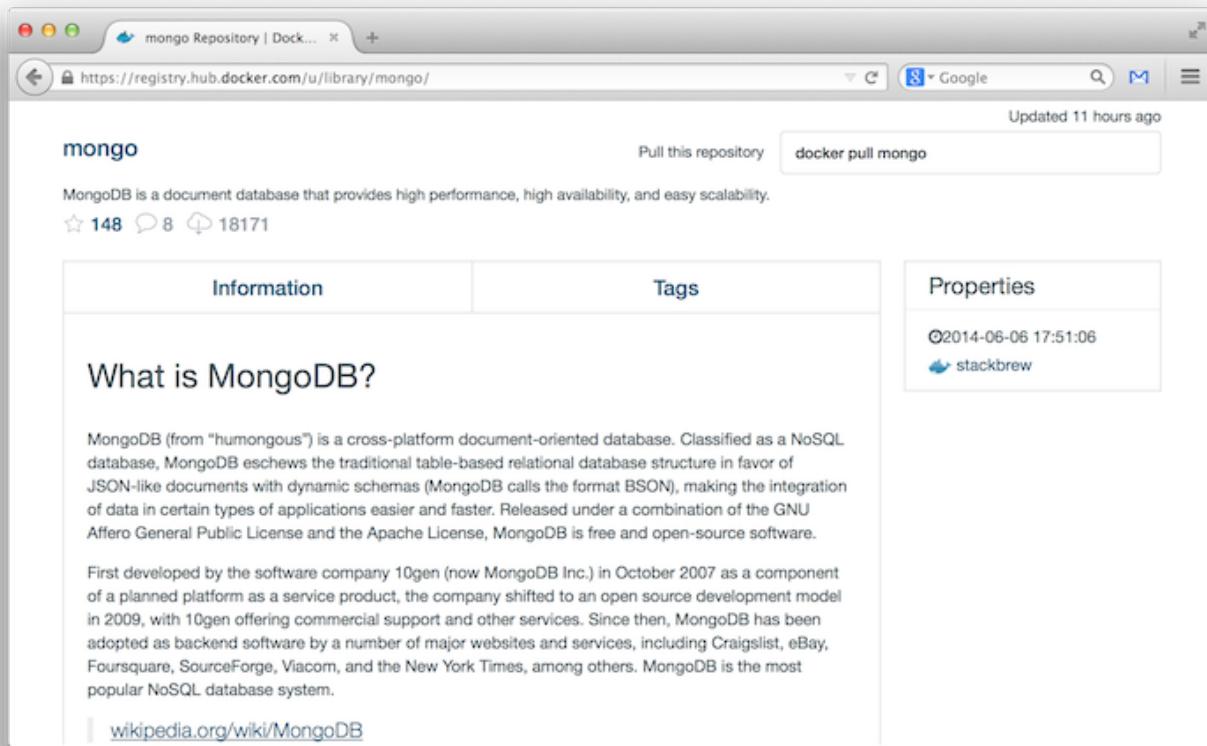
Public Repositories

- Docker Hub provides access to tens of thousands of pre-made images that you can build from.
- Some of these are official builds and live in the root namespace.
- Most are community contributed and maintained.



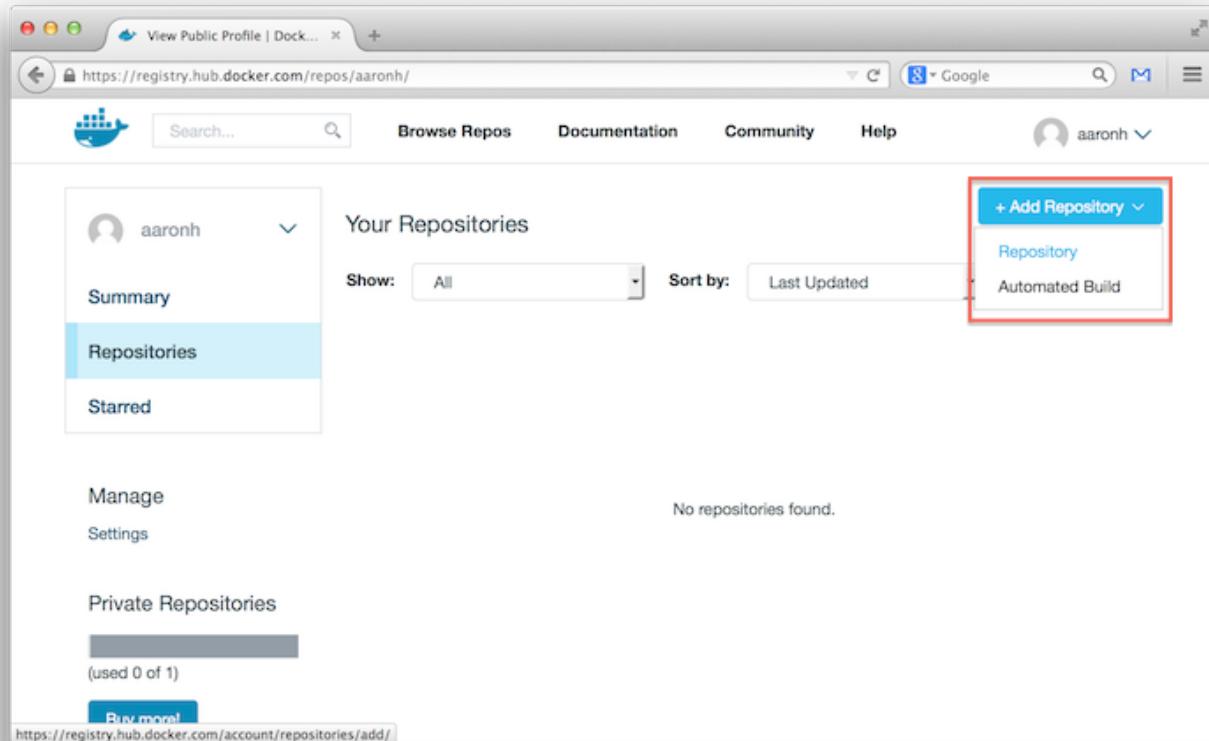
Official Repositories

- Are maintained by the product owners
- Blessed by Docker



New Repository (1/3)

- Pull down Add Repository menu and select Repository



New Repository (2/3)

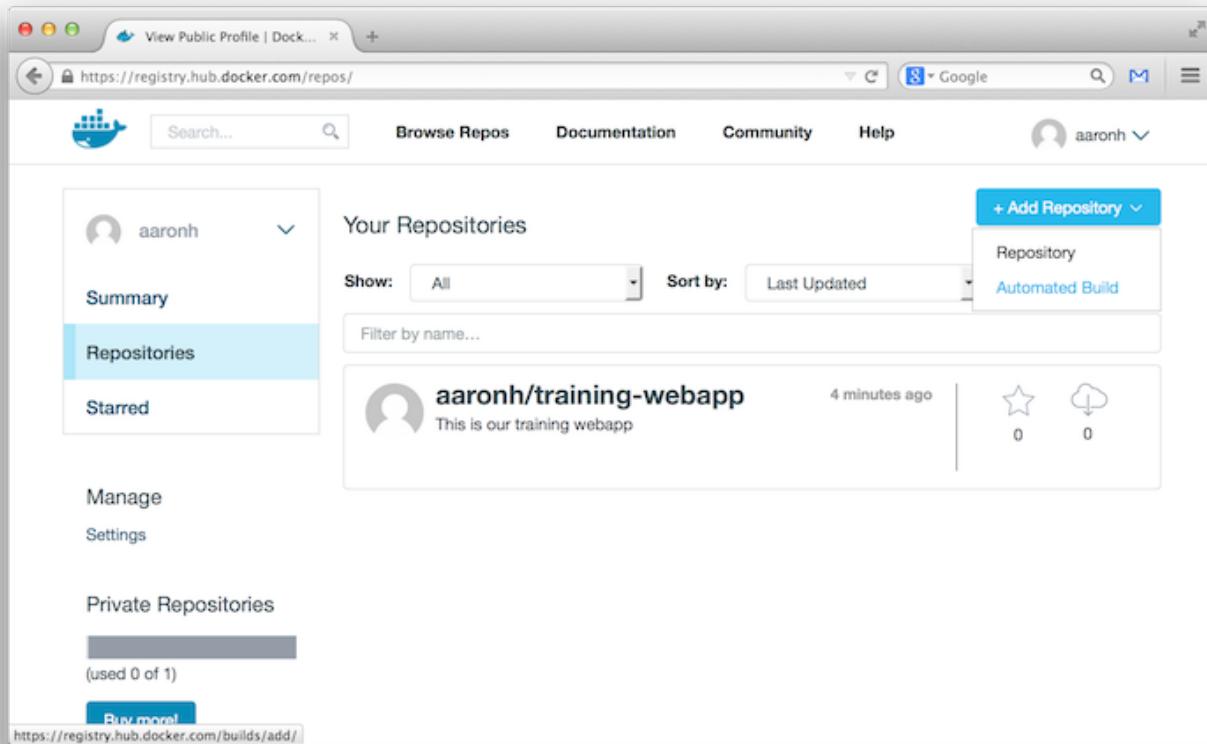
- Leave namespace at the default (your username)
- Give your repository a name
- Type a brief description so people know what it is
- Leave Public selected
- Submit the form with Add Repository button (not shown)

The screenshot shows a web browser window with the URL <https://registry.hub.docker.com/account/repositories/add/>. The page title is "Add Repository".
The form fields are:

- Namespace (optional) and Repository Name:** The input field contains "aaronh / training-webapp". Below it is a note: "New unique Repo name; 3 - 30 characters. Only lowercase letters, digits and _ - . characters are allowed."
- Description:** A text area containing "This is our training webapp". Below it is a note: "Limit 100 Characters".
- Repository Type:** A radio button group with "Public" selected and "Private" as an option.

New Repository (3/3)

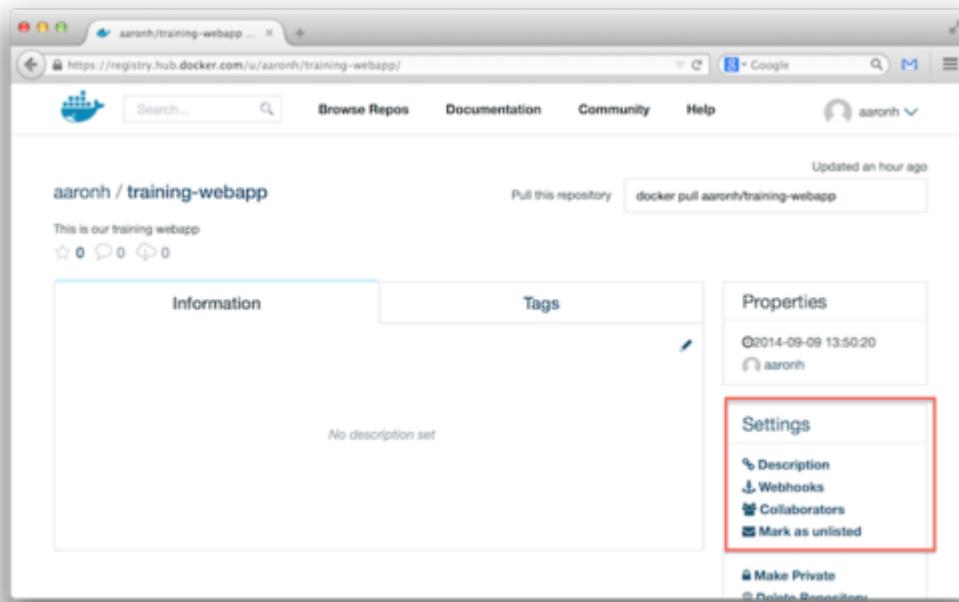
- Click **Repositories** and you will see your new repository.
- You can push images to this repository from the docker commandline.
- More on this later.



Repository Settings

You can change the following:

- Repository Description
- Webhooks
- Collaborators
- Mark as unlisted in the global search (NOT a private repository)



Collaborators

You can invite other Docker Hub users to collaborate on your projects.

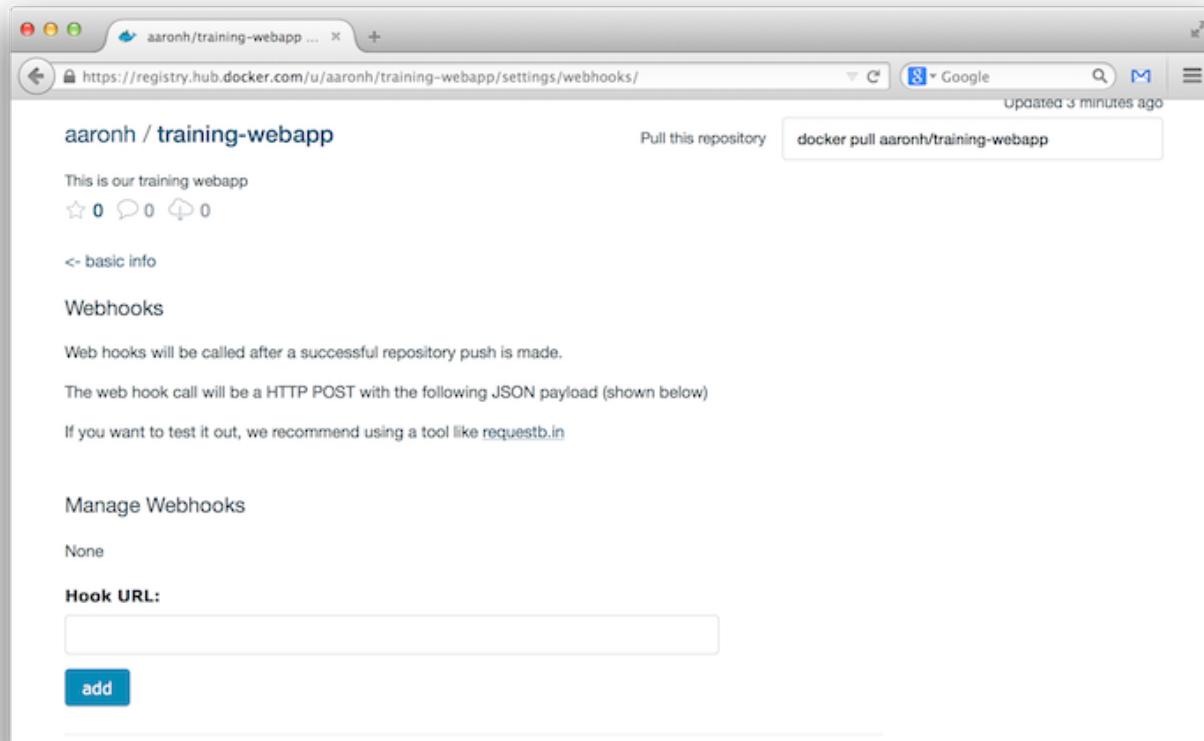
- Collaborators cannot change settings in the repository.
- Collaborators can push images to the repository.

The screenshot shows a web browser window with the URL <https://registry.hub.docker.com/u/aaronh/training-webapp/settings/collaborators/>. The page displays basic repository information: 'aaronh / training-webapp', a description 'This is our training webapp', and statistics '0 stars', '0 comments', and '0 forks'. A 'Pull this repository' button and a command 'docker pull aaronh/training-webapp' are also present. The 'Collaborators' section indicates that users can push and pull images but have no administrative rights. Below this, a 'Manage Users' table lists two users: 'aaronh' (access: owner) and 'huslage' (access: collaborator). A blue 'add' button is available to invite new users. The page is updated 3 minutes ago.

Username	Access	Action
aaronh <small>You!</small>	owner	
huslage	collaborator	[remove]

Webhooks

- Notify external applications that an image has been uploaded to the repository.
- Powerful tool for integrating with your development workflow.
- Even more powerful when used with Automated Builds.



Automated Builds

- Automatically build an image when source code is changed.
- Integrated with Github and Bitbucket
- Work with public and private repositories
- Add the same as a regular repository, select Automated Build from the Add Repository menu
- We'll set one of these up later!
 - You will need a Github account to follow along later, so go ahead and create one now if you don't have one yet.

Section summary

We've learned how to:

- Register for an account on Docker Hub.
- Login to your account from the command line.
- Access some special Docker Hub features for better workflows.

Working with Images

Lesson 18: Working with Images

Objectives

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

- Pull and push images to the Docker Hub.
- Explore the Docker Hub.
- Understand and create *Automated Builds*.

Working with images

In the last section we created a new image for our web application. This image would be useful to the whole team but how do we share it? Using the [Docker Hub](#)!

```
$ docker pull ubuntu:14.04
```

Pulling images

Earlier in this training we saw how to pull images down from the Docker Hub.

This will connect to the Docker Hub and download the ubuntu:14.04 image to allow us to build containers from it.

We can also do the reverse and push an image to the Docker Hub so that others can use it.

Before pushing a Docker image ...

We push images using the docker push command.

Images are uploaded via HTTP and authenticated.

You can only push images to the *user namespace*, and with your own username. This means that you cannot push an image called web.

It has to be called <dockerhubUsername>/web.

```
$ docker build -t <dockerhubUsername>/web\ git://  
github.com/docker-training/  
staticweb.git
```

Name your image properly

Here are different ways to ensure that your image has the right name.

Of course, in the examples below, replace <dockerhubUsername>with your actual login on the Docker Hub.

- If you have previously built the webimage, you can re-tag it:

- Or, you can also rebuild it from scratch:

```
$ docker push <dockerhubUsername>/web
```

```
Please login prior to push: Username:  
<dockerhubUsername> Password:  
*****  
Email: ... Login  
Succeeded
```

Pushing a Docker image to the Docker Hub

Now that the image is named properly, we can push it:

You will be prompted for a user name and password.

(Unless you already did docker loginearlier.)

You will login using your Docker Hub name, account and email address you created earlier in the training.

More about pushing an image

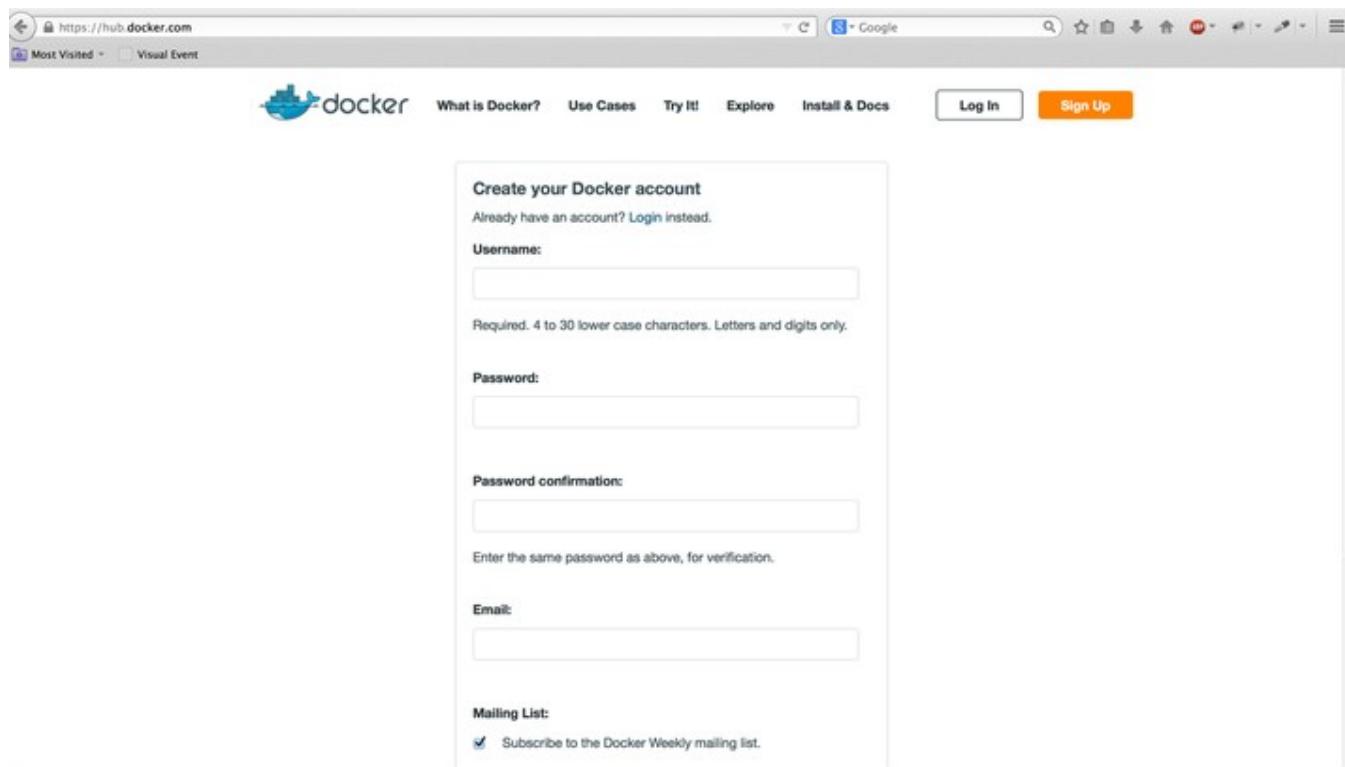
- If the image doesn't exist on the Docker Hub, a new repository will be created.
- You can push an updated image on top of an existing image. Only the layers which have changed will be updated.
- When you pull down the resulting image, only the updates will need to be downloaded.

<https://hub.docker.com/>

Viewing our uploaded image

Let's sign onto the [Docker Hub](#) and review our uploaded image.

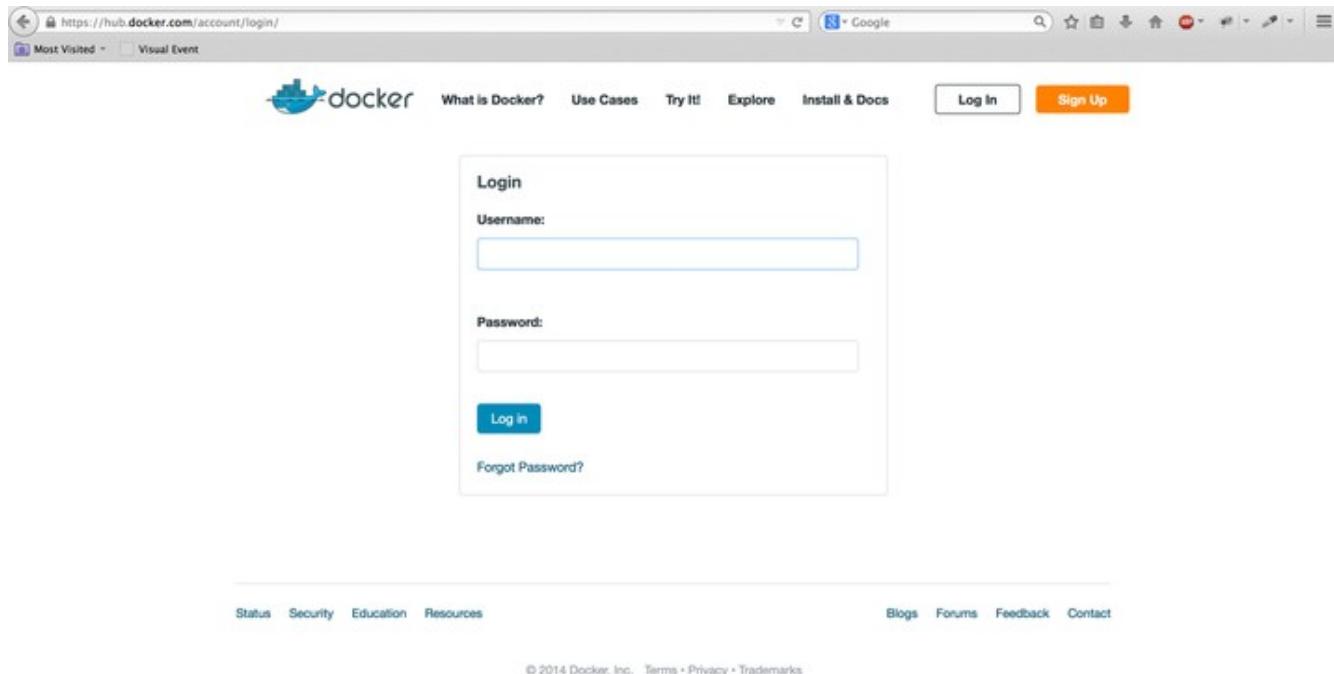
Browse to:



Logging in to the Docker Hub

Now click the Loginlink and fill in the Usernameand Passwordfields.

And clicking the Log Inbutton.



Your account screen

This is the master account screen. Here you can see your repositories and recent activity.

The screenshot shows the Docker Hub account interface for user 'nathanleclaire'. On the left, a sidebar includes links for Summary (selected), Repositories, Starred, Manage, Settings, and Private Repositories (with a 'Buy more!' button). The main area displays 'Your Recently Updated Repositories' with four cards:

- devbox** (6 days ago): A repository full of good stuff for developing applications (Docker in particular). 2 forks, 0 stars.
- webapp** (3 weeks ago): 7 forks, 0 stars.
- pushit** (3 weeks ago): 0 forks, 0 stars.
- serve** (1 months ago): 2 forks, 0 stars.

Below this are sections for 'Contributed Repositories' (dockercn/demo-app) and 'Starred Repositories' (dockercn/demo-app, johnston/apache, johnston/nginx-test).

Review your webapp repository

Click on the link to your <dockerhubUsername>/webrepository.

The screenshot shows a Docker Hub repository page for the user 'nathanleclaire' with the repository name 'webapp'. The page includes a search bar, navigation links for 'Browse Repos', 'Documentation', 'Community', and 'Help', and a user profile for 'nathanleclaire'. The repository details show it is an 'AUTOMATED BUILD REPOSITORY' updated 3 weeks ago. A 'Pull this repository' button contains the command 'docker pull nathanleclaire/webapp'. Below the repository name, there are statistics: 0 stars, 0 comments, and 7 forks. The main content area has tabs for 'Information', 'Build Details', 'Tags', and 'Start a Build (staff)'. The 'Information' tab displays the description 'Docker Fundamentals WebApp' and the note 'The Docker Fundamentals repository contains the example Hello World Python WebApp'. It also shows the 'License' (Apache 2.0) and 'Copyright' (Copyright Docker Inc Education Team 2014 education@docker.com). The 'Comments' section indicates 'No comments available, be the first to comment.' A blue 'Add Comment' button is present. On the right side, a sidebar titled 'Build Details' lists 'Links' (Source Project Page, Source Repository), 'Files' (Build Bundle, Dockerfile), and 'Settings' (Description, Build Details, Webhooks, Collaborators, Build Triggers, Repository Links, Make Private, Delete repository). A 'Start a Build (staff)' button is located at the top right of the main content area.

- You can see the basic information about your image.
- You can also browse to the Tags tab to see image tags, or navigate to a link in the "Settings" sidebar to configure the repo.

Automated Builds

In addition to pushing images to Docker Hub you can also create special images called *Automated Builds*. An *Automated Build* is created from a Dockerfile in a GitHub repository.

This provides a guarantee that the image came from a specific source and allows you to ensure that any downloaded image is built from a Dockerfile you can review.

Creating an Automated build

To create an *Automated Build* click on the Add Repositorybutton on your main account screen and select Automated Build.

The screenshot shows the Docker Hub account interface for user 'nathanleclaire'. On the left, there's a sidebar with 'Summary' (selected), 'Repositories' (disabled), and 'Starred'. The main area displays 'Your Recently Updated Repositories' with two items: 'devbox' (6 days ago) and 'webapp' (3 weeks ago). A dropdown menu titled '+ Add Repository' is open on the right, showing options: 'Repository' (disabled) and 'Automated Build' (selected).

Connecting your GitHub account

If this is your first *Automated Build* you will be prompted to connect your GitHub account to the Docker Hub.

Select the source you want to use for your Automated Build



GitHub

Select



Bitbucket

Select

Select specific GitHub repository

You can then select a specific GitHub repository. It must contain a Dockerfile.

GitHub: Add Automated Build

For more information on Automated Builds, please read the [Automated Build documentation](#).

Select a Repository to build

The screenshot shows a list of GitHub repositories belonging to the user 'nathanleclaire'. The repositories listed are: nathanleclaire/twilio_toy_app, nathanleclaire/unix-command-survey, nathanleclaire/wakeup, and nathanleclaire/webapp. Each repository entry has a 'Select' button to its right. The 'Select' button for 'nathanleclaire/twilio_toy_app' is highlighted in blue, indicating it is selected or about to be selected.

Repository	Select
nathanleclaire/twilio_toy_app	Select
nathanleclaire/unix-command-survey	Select
nathanleclaire/wakeup	Select
nathanleclaire/webapp	Select

If you don't have a repository with a Dockerfile, you can fork <https://github.com/docker-training/staticweb>, for instance.

Configuring Automated Build

You can then configure the specifics of your *Automated Build* and click the Create Repository button.

README.md

If you have a README.md file in your repository, we will use that as the repository full description. We will look for the README.md in the same directory where your Dockerfile lives.

Warning: if you change the full description after a build, it will be rewritten the next time the Automated Build has been built. To make changes, change the README.md in the source repo. For more information please read the [Automated Build documentation](#).

Namespace (optional) and Repository Name

nathanleclaire / webapp

New unique Repo name; 3 - 30 characters. Only lowercase letters, digits and _ - . characters are allowed

Tags

Type	Name	Dockerfile Location	Docker Tag Name
Branch	master	/	latest

Public
Anyone can pull, and is listed and searchable on the docker index.

Private
Only you can pull, and is not listed on the docker index.

Active

When active we will build when new pushes occur

Create Repository

Automated Building

Once configured your *Automated Build* will automatically start building an image from the Dockerfile contained in your Git repository.

Every time you make a commit to that repository a new version of the image will be built.

Section summary

We've learned how
to:

- Pull and push images to the Docker Hub.
- Explore the Docker Hub.
- Understand and create *Automated Builds*.

Using Docker for testing

Lesson 19: Using Docker for testing Objectives

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

- Dockerize a Flask (Python) web application and its tests
- Integrate this Dockerized application with Jenkins for CI
- Use Web Hooks with Automated Builds to automate testing and deployment.



Docker for software testing

One of Docker's popular use cases is to better enable testing.

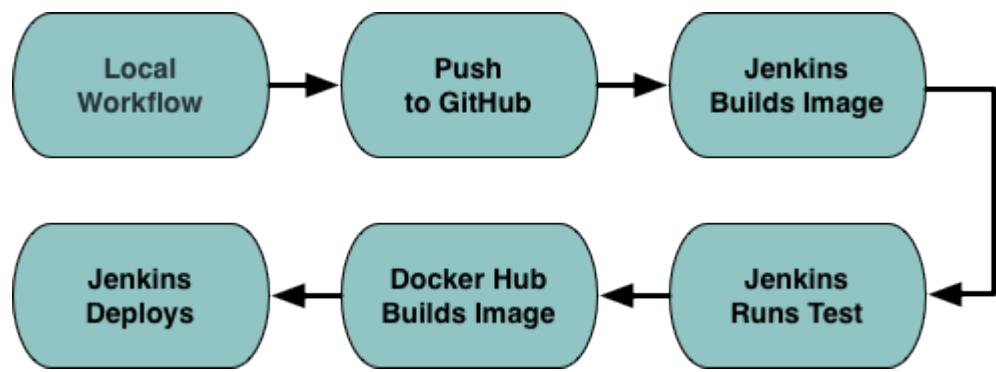
We're going to see how to integrate Docker into a Continuous Integration workflow.

To do this we're going to assume that every time we update an application that we want to rebuild its image and run the application's tests.



What we want

1. Every time our code changes we want our image rebuilt.
2. Every time our code changes we want the tests run against our new image.
3. If the tests are good, deploy the new code.



The plan

- Use a simple web application based on Python Flask.
- Look at the Dockerfile for that application.
- Look at the test suite for that application.
- Create an Automated Build for that application.
- Configure a Web Hook to trigger a Jenkins build.

Why?

This flow is designed to use all of the power of Docker Hub to distribute and maintain your repositories and images. It allows you to maintain the integrity and availability of images to all of your hosts. It also allows you to easily utilize any future Docker Hub features.

- We recommend using Docker Hub's repositories as a known-good source of images for your deployments.
- The CI system (Jenkins in this case) should run tests inside of the same environments as development and production.
- Docker Hub should build all images that will be deployed to production environments.
- The CI system should be able to block Docker Hub from building an image if there is a test failure.
- The deployment system (also Jenkins in this case) should only deploy images that are built on Docker Hub.

Our web application

For simplicity, we are going to use a tiny Python Flask application, called `webapp`.

We're going to use this application as our example of using Docker for continuous integration.

```
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
ADD ./webapp/requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
EXPOSE 5000
```

The webapp Dockerfile

Let's look at the Dockerfile for that application.

You can see the Dockerfile and the application on GitHub:

<https://github.com/docker-training/webapp>

The webapp Dockerfile explained

Our Dockerfile is pretty simple:

- We use an Ubuntu 12.04 image as the base.
- We install basic prerequisites, like Python.
- We add the contents of the webapp directory that holds our application code.
- We specify the working directory of the image as /opt/webapp.
- We install the application's Python-based dependencies using pip.
- We expose port 5000 to serve our application on.
- Finally, our application is launched using python app.py.

```
from app import app

import os
import unittest

class AppTestCase(unittest.TestCase):
    def test_root_text(self):
        tester = app.test_client(self)
        response = tester.get('/')
        assert 'Hello world!' in response.data

if __name__ == '__main__':
    unittest.main()
```

Our Flask tests

Inside our application, in [tests.py](#), we've also got a simple Python test for our Flask application.

Forking our GitHub repository

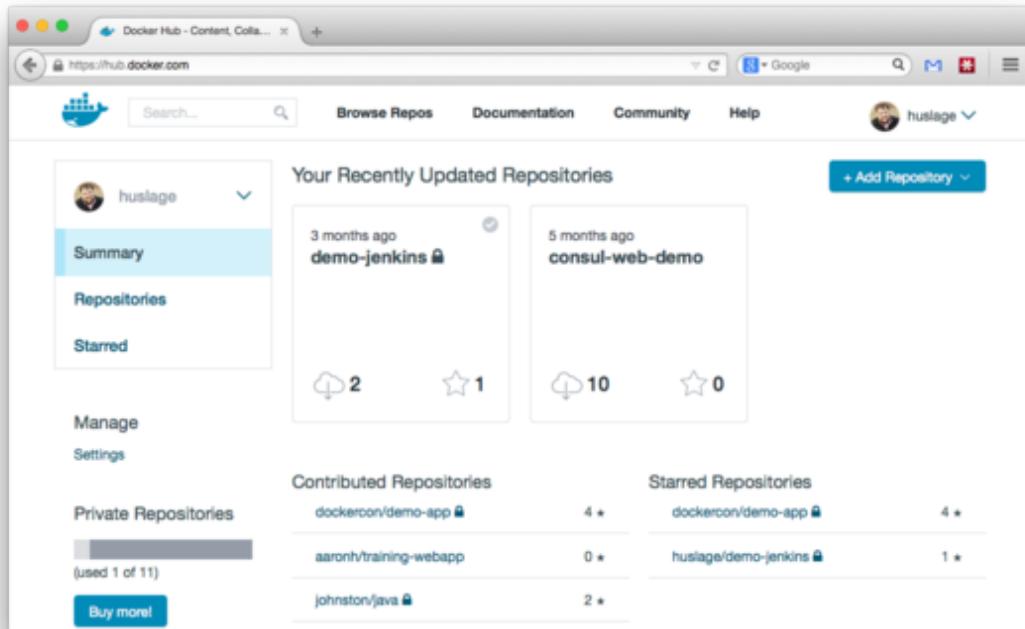
We need a place to put our code changes that will trigger this whole flow. Let's fork the docker-training/webapp repository.

Go to <http://github.com/docker-training/webapp>. Click Fork in the upper-right corner. This will create a new repository under your account to hold changes that you make.

Adding a new Automated Build

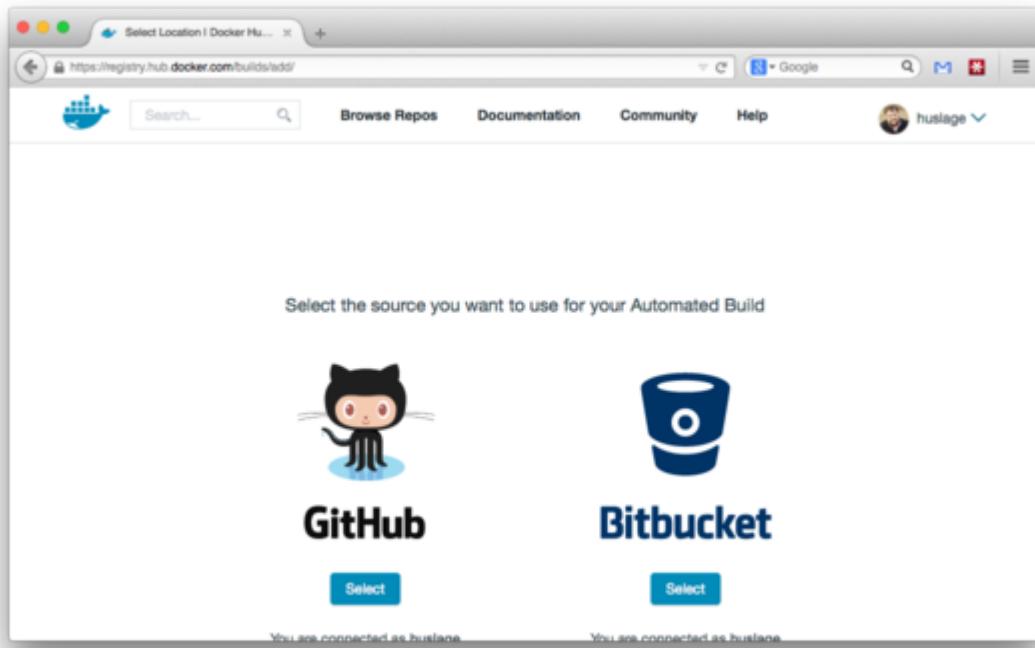
Go to your Docker Hub profile page and click:

Add automated (source) build



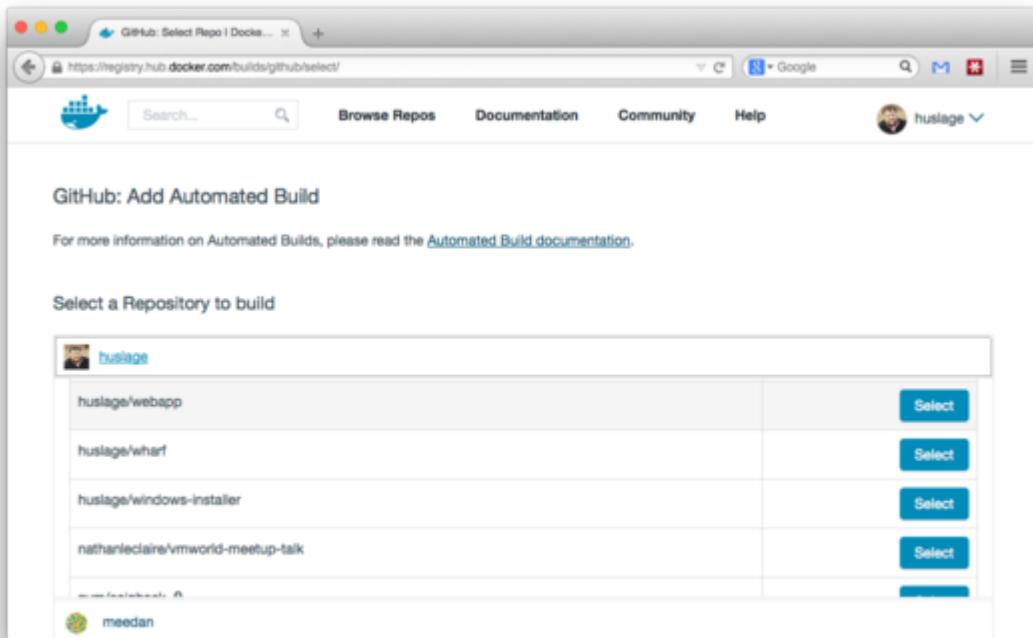
Selecting the Source Code Service

Click on GitHub for the service you want to use.



Selecting the repository

We then select the webapprepository from your GitHub Account.



The screenshot shows a web browser window titled "GitHub: Select Repo | Docker...". The URL in the address bar is <https://registry.hub.docker.com/builds/github/select/>. The page header includes links for "Browse Repos", "Documentation", "Community", "Help", and a user profile for "huslage". Below the header, the title is "GitHub: Add Automated Build" with a sub-instruction: "For more information on Automated Builds, please read the [Automated Build documentation](#)". A section titled "Select a Repository to build" lists repositories under the "huslage" organization. The repositories listed are: huslage/webapp, huslage/wharf, huslage/windows-installer, and nathanleclaire/vmworld-meetup-talk. To the right of each repository name is a blue "Select" button. At the bottom of the list, there is another section for "meedan" which is currently collapsed.

Repository	Select
huslage/webapp	Select
huslage/wharf	Select
huslage/windows-installer	Select
nathanleclaire/vmworld-meetup-talk	Select
... more repositories ...	
meedan	(button collapsed)

Configuring the build

We then configure our Automated Build and click the Create Repositorylink.

The screenshot shows a web browser window with the URL <https://registry.hub.docker.com/builds/github/huslage/webapp/>. The page is titled "Namespace (optional) and Repository Name" and displays the repository name "huslage" followed by a slash and "webapp". Below this, a note states: "New unique Repo name; 3 - 30 characters. Only lowercase letters, digits and _ - . characters are allowed".

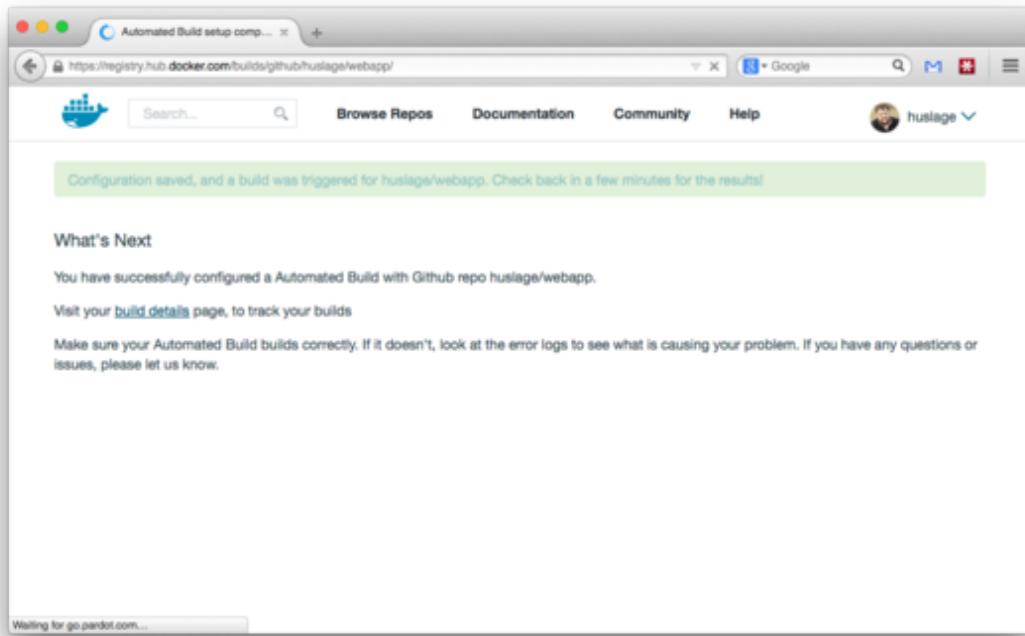
Under the "Tags" section, there is a table with columns: Type, Name, Dockerfile Location, and Docker Tag Name. A row is shown with "Branch" selected in the Type dropdown, "master" in the Name field, an empty Dockerfile Location field, and "latest" in the Docker Tag Name field. A blue "+" button is located to the right of the Docker Tag Name field.

Below the table, there are two radio button options: "Public" (selected) and "Private". The "Public" option is described as "Anyone can pull, and is listed and searchable on the docker index." The "Private" option is described as "Only you can pull, and is not listed on the docker index."

At the bottom, under "Active:", there is a checkbox labeled "When active we will build when new pushes occur" which is checked.

Success

This is what success looks like. Drink it in.



Build status page

If you click on Automated Builds on the left sidebar, you can view the status of your build. When you create a new build, we automatically run it the first time. How handy!

The screenshot shows a web browser window displaying the Docker Hub build status page for the repository `huslage / webapp`. The URL in the address bar is `https://registry.hub.docker.com/u/huslage/webapp/builds_history/66561/`. The page header includes the repository name, a pull button, and a timestamp indicating the build was updated 9 seconds ago.

The main content area is divided into several sections:

- Information:** Shows the repository name `huslage / webapp`, a description field (empty), and social sharing icons (0 stars, 0 comments, 0 forks).
- Dockerfile:** Shows the Dockerfile location as `/`.
- Build Details:** Shows the build type as `Branch`, the name as `master`, the Dockerfile location as `/`, and the tag name as `latest`. There is also a link to `Edit Build Details`.
- Tags:** A section showing available tags.
- Builds History:** A table showing the history of builds, with one entry for build ID `byxjblwygtvkoughtwpuvmf` which is currently `Building`.
- Properties:** A sidebar listing repository metadata: created at `2014-10-13 09:14:59` by user `huslage`.
- Settings:** A sidebar with options for managing the repository: Description, Automated Build, Webhooks, Collaborators, Build Triggers, Repository Links, and Mark as unlisted.

```
$ docker run -d --name=jenkins -p 8080:8080 \
-v /var/run/docker.sock:/var/run/docker.sock \
-e DOCKERHUB_ID=<Your Docker Hub ID> \
-e DOCKERHUB_EMAIL=<Your Docker Hub Email> \
-e GITHUB_ID=<Your GitHub ID> nathanleclaire/
jenkins
```

Run Jenkins

We can use an existing Docker image to run Jenkins, nathanleclaire/jenkins. Let's run a Jenkins instance now from this image.

This will launch a container from our nathanleclaire/jenkins image and bind it to port 8080 on our local host. The -e options pass environment variables to the script that runs Jenkins. They configure the built-in jobs so that you don't have to.

P.S. If you're interested you can see the [Dockerfile](#) that built this image: <https://github.com/docker-training/jenkins/blob/master/Dockerfile>

Configure Jenkins

Our Jenkins image is already pre-populated with jobs that will run our Webapp tests. Go to the URL of your Jenkins instance

The screenshot shows the Jenkins dashboard at <http://ec2-54-86-232-114.compute-1.amazonaws.com:8080>. The left sidebar includes links for New Item, People, Build History, Manage Jenkins, and Credentials. The main area displays a table of builds:

S	W	Name	Last Success	Last Failure	Last Duration
●	●	training-webapp-build	3 days 20 hr - #2	huslage/webapp-1	N/A 2.7 sec
●	●	training-webapp-deploy	3 days 20 hr - #2		N/A 0.12 sec
●	●	training-webapp-test	3 days 20 hr - #2		N/A 6.3 sec

Build Queue: No builds in the queue.

Build Executor Status: 1 Idle, 2 Idle.

Help us localize this page | Page generated: May 27, 2014 2:14:37 PM | REST API | Jenkins ver. 1.565

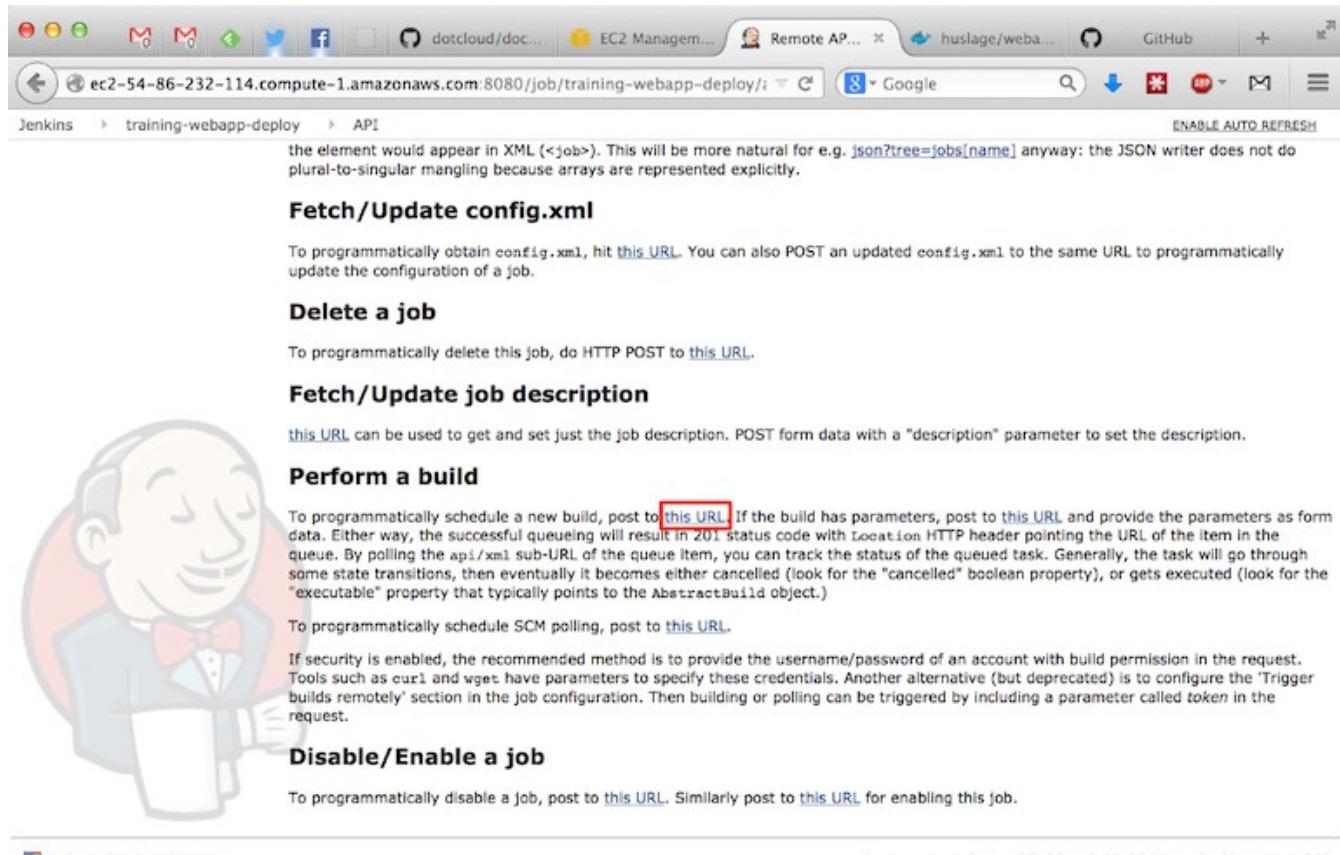
Configuring Deploy Job

We're almost done. Now we need to set up a trigger for the deployment step. From the Jenkins Dashboard click the training-webapp-deploy job. In the bottom-right of the page you will see a link for the REST API. Click that link.

The screenshot shows the Jenkins interface for the 'training-webapp-deploy' project. On the left, there's a sidebar with various project management links like Status, Changes, Workspace, Build Now, Delete Project, Configure, and GitHub. Below that is a 'Build History' section showing two builds: #2 (May 23, 2014 5:16:56 PM) and #1 (May 23, 2014 5:09:25 PM). To the right, there's a main content area titled 'Project training-webapp-deploy'. It features a 'Workspace' link with a folder icon and a 'Recent Changes' link with a document icon. At the top right, there are buttons for 'add description' and 'Disable Project'. Below the main content, under 'Permalinks', is a list of three build links. At the bottom right of the page, there's a red box around the 'REST API' link. The footer includes links for 'Help us localize this page', 'Page generated: May 27, 2014 2:21:36 PM', 'Jenkins ver. 1.565', and the 'REST API' link itself.

Get the build link URL

Scroll down a bit to the Perform a build section. Copy the link where it says Post to this URL.



the element would appear in XML (<job>). This will be more natural for e.g. `json?tree=jobs[name]` anyway: the JSON writer does not do plural-to-singular mangling because arrays are represented explicitly.

Fetch/Update config.xml

To programmatically obtain `config.xml`, hit [this URL](#). You can also POST an updated `config.xml` to the same URL to programmatically update the configuration of a job.

Delete a job

To programmatically delete this job, do HTTP POST to [this URL](#).

Fetch/Update job description

[this URL](#) can be used to get and set just the job description. POST form data with a "description" parameter to set the description.

Perform a build

To programmatically schedule a new build, post to [this URL](#). If the build has parameters, post to [this URL](#) and provide the parameters as form data. Either way, the successful queueing will result in 201 status code with `Location` HTTP header pointing the URL of the item in the queue. By polling the `api/xml` sub-URL of the queue item, you can track the status of the queued task. Generally, the task will go through some state transitions, then eventually it becomes either cancelled (look for the "cancelled" boolean property), or gets executed (look for the "executable" property that typically points to the `AbstractBuild` object.)

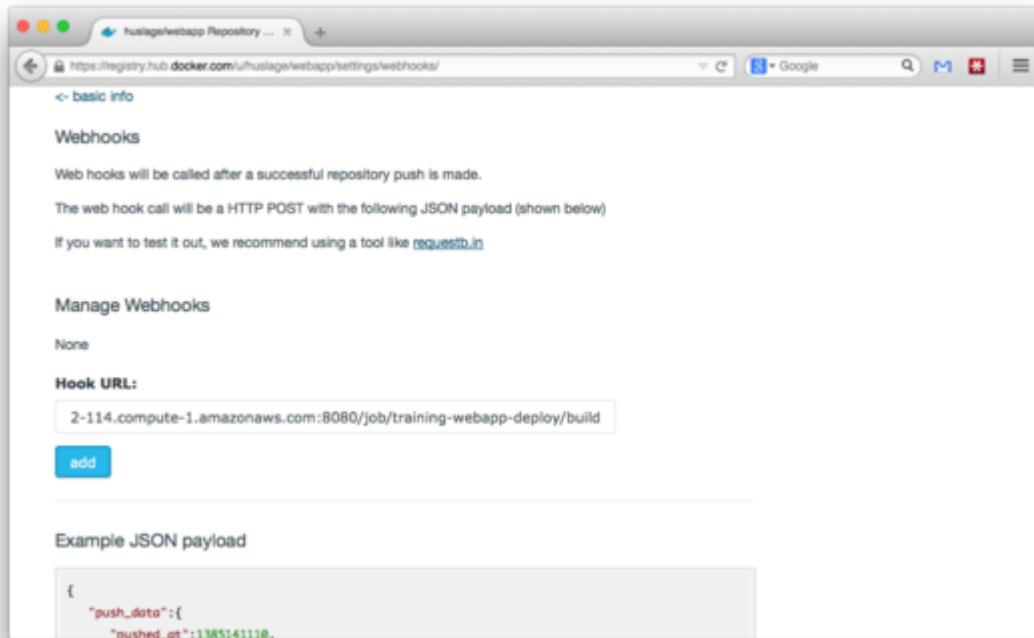
To programmatically schedule SCM polling, post to [this URL](#).

If security is enabled, the recommended method is to provide the username/password of an account with build permission in the request. Tools such as `curl` and `wget` have parameters to specify these credentials. Another alternative (but deprecated) is to configure the 'Trigger builds remotely' section in the job configuration. Then building or polling can be triggered by including a parameter called `token` in the request.

Disable/Enable a job

To programmatically disable a job, post to [this URL](#). Similarly post to [this URL](#) for enabling this job.

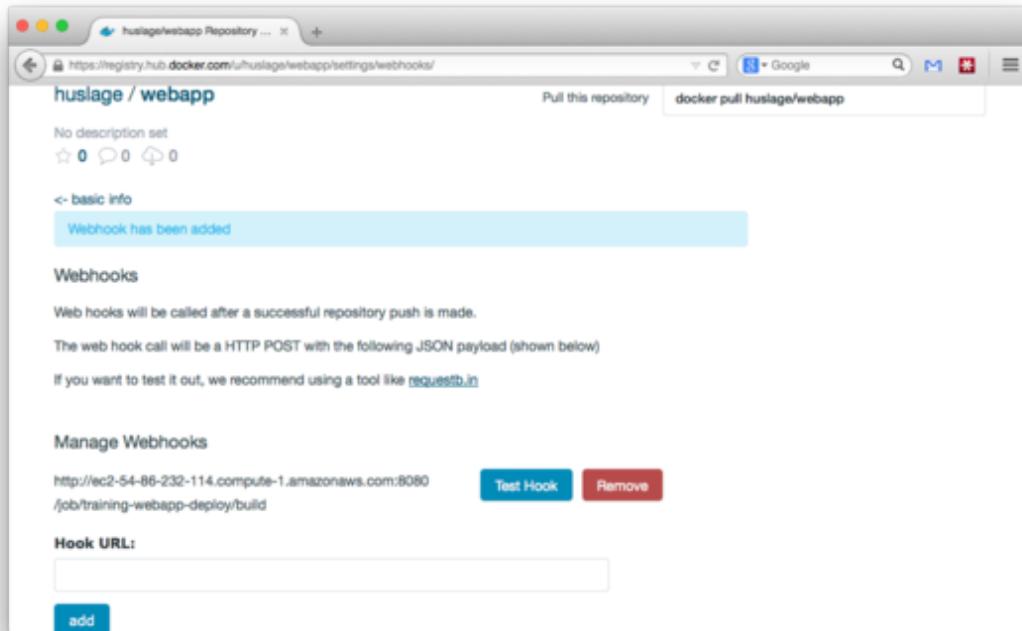
Go back to your webapp Docker Hub repository Settings and click Webhooks on the left sidebar. Paste the URL you just copied into the Hook URL box.



Click Add

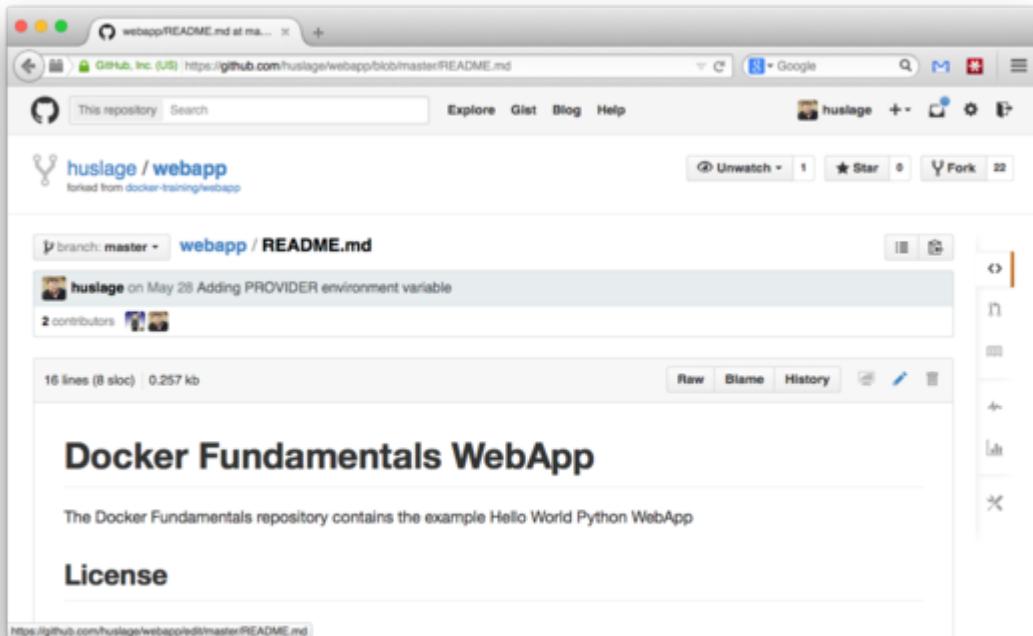
More Success!

We have successfully added a webhook.



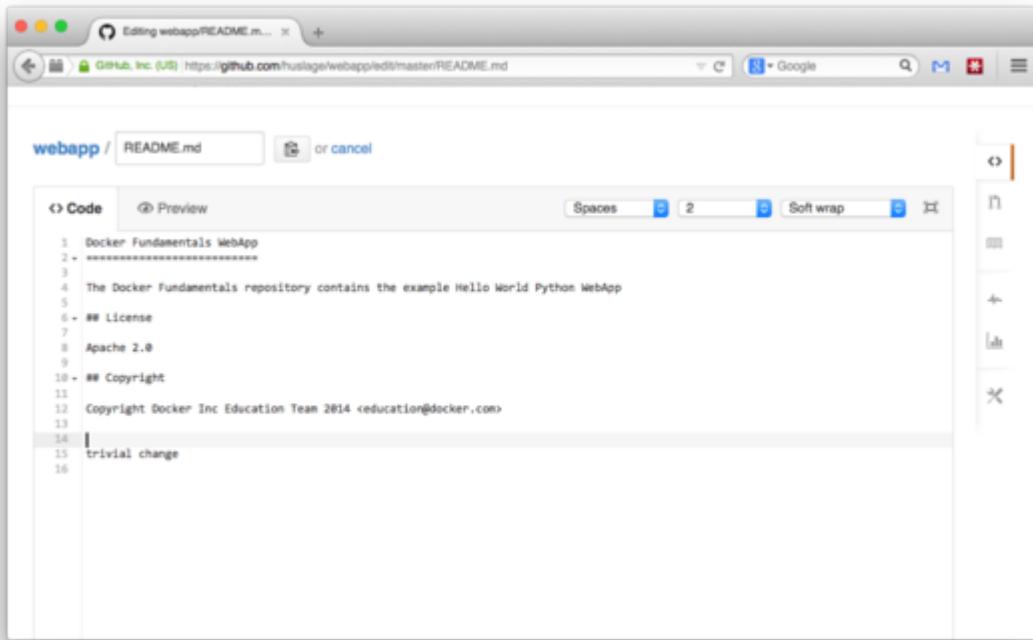
Make some changes

Now we need to make a change to the repository. From the repository home page, click on README.md. Then click Editon the top of the next page.



Trivial changes are best

When the edit box comes up, add some text to the bottom of the file. Then scroll to the bottom of the page and click CommitChanges.



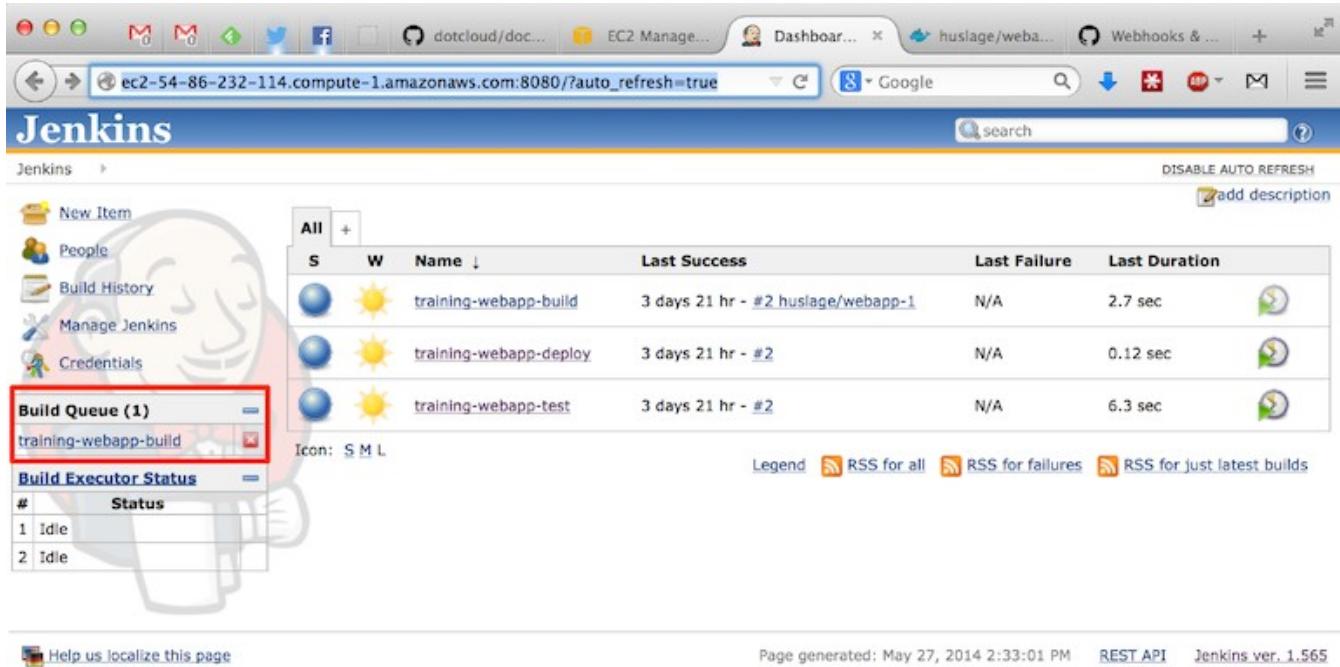
A screenshot of a GitHub README.md editor. The URL in the address bar is <https://github.com/huslage/webapp/edit/master/README.md>. The editor shows the following code:

```
1 Docker Fundamentals WebApp
2 -----
3
4 The Docker Fundamentals repository contains the example Hello World Python WebApp
5
6 ## License
7
8 Apache 2.0
9
10 ## Copyright
11
12 Copyright Docker Inc Education Team 2014 <education@docker.com>
13
14 | trivial change
15
16
```

The Jenkins job will be triggered by this action!

Watch it go

Go back to your Jenkins dashboard. You will see a job running or queued.



The screenshot shows the Jenkins dashboard with a build queue. A specific item, "training-webapp-build", is highlighted in the queue, indicating it is currently running or has recently failed. The dashboard also displays other builds and executor status.

S	W	Name ↓	Last Success	Last Failure	Last Duration
●	☀️	training-webapp-build	3 days 21 hr - #2	N/A	2.7 sec
●	☀️	training-webapp-deploy	3 days 21 hr - #2	N/A	0.12 sec
●	☀️	training-webapp-test	3 days 21 hr - #2	N/A	6.3 sec

Build Queue (1)
Icon: S M L

Build Executor Status
Status
1 Idle
2 Idle

Legend: RSS for all RSS for failures RSS for just latest builds

Help us localize this page Page generated: May 27, 2014 2:33:01 PM REST API Jenkins ver. 1.565

Automation in Action

When the training-webapp-testjob finishes, you can look at the Build Status page on your Automated Build. It should build fine.

You can also try b0rking the source code to see that it fails ;)

The screenshot shows a web browser window displaying a Docker repository build status page. The URL is https://registry.hub.docker.com/u/huslage/webapp/builds/_history/66561/. The page has a green header bar indicating a "Staff Triggered build for 20598". Below this, there are tabs for "Information", "Dockerfile", "Build Details", and "Tags", with "Build Details" selected. The "Build Details" section shows a table with columns "Type", "Name", "Dockerfile Location", and "Tag Name". A single row is present: Type is "Branch", Name is "master", Dockerfile Location is "/", and Tag Name is "latest". To the right of the table is a "Properties" sidebar showing the build date (2014-10-13 09:14:59) and the user "huslage". Below the table is a "Builds History" section with a table showing two builds. The first build has ID "bqhmqjtjoulvppm4ddjfp1" and status "Building", created on 2014-10-13 09:19:25 and last updated on 2014-10-13 09:19:27. The second build has ID "byxjblwygtvkoughwpuvrmf" and status "Building", created on 2014-10-13 09:14:59 and last updated on 2014-10-13 09:15:01. To the right of the history table is a "Settings" sidebar with options: Description, Automated Build, Webhooks, Collaborators, Build Triggers, Repository Links, Make Private, and Delete Repository.

Section summary

We've learned how to:

- Dockerize a Flask (Python) web application and its tests
- Integrate this Dockerized application with Jenkins for CI
- Use Web Hooks with Automated Builds to automate testing.

Security



Lesson 20: 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.

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

Accessing the host filesystem

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

If you want to explore freely 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...
```

Modifying the host filesystem

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

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 `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](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

Securing Docker with TLS

Lesson 21: 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 therefore encrypted *and* authenticated.

How Docker Uses TLS

- Docker provides mechanisms to authenticate both the server and 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.*

```
$ 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
```

Creating a Certificate Authority

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

We will use the ca.pemfile to sign all of the other keys later.

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

```
$ openssl genrsa -des3 -out server-key.pem 2048
```

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 CNmatches the hostname you run the Docker daemon on:

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

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

Create and Sign the Client Key

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

Now sign the key:

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

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.

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.

```
$ docker --tlsverify ps  
$ cp ca.pem ~/.docker/ca.pem  
$ cp client-cert.pem ~/.docker/cert.pem
```

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 `.dockercfg` directory in your home directory. Set the `DOCKER_HOST` variable as well.

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

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 22: 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",
        "InitSha1": "",
        "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.

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

Creating a new container via the API

Let's use curl to create a simple container.

- 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.

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

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.

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

```
$ 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"
    ],
    ...
  }
}
```

Inspecting our launched container

We can also inspect our freshly launched container.

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

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

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.

- 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.

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

Stopping a container

We can also stop a container using the API.

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

```
$ curl -X GET http://localhost:2375/images/json?all=0 [  
 {  
   "Created": 1396291095,  
   "Id": "cccdcc2d2ec497e814793e8bd952ae76d5d552c8bb7ed927db54aa65579508ffd", "ParentId":  
   "9cd978db300e27386baa9dd791bf6dc818f13e52235b26e95703361ec3c94dc6", "RepoTags": [  
     "training/datavol:latest"  
   ],  
   "Size": 0,  
   "VirtualSize": 204371253  
 },  
 {  
   "Created": 1396117401,  
   "Id": "d4faa2107ddab5b22e815759d9a345f1381562ad44d1d95235347d6b006ec713", "ParentId":  
   "439aa219e271671919a52a8d5f7a8e7c2b2950c639f09ce763ac3a06c0d15c22",  
   ...  
 }  
 ]
```

Working with images

We can also work with Docker images.

- Returns a hash of all 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  
 }  
 ]
```

Searching the Docker Hub for an image

We can also search the Docker Hub for specific images.

This returns a list of images and their metadata.

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

Creating an image

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

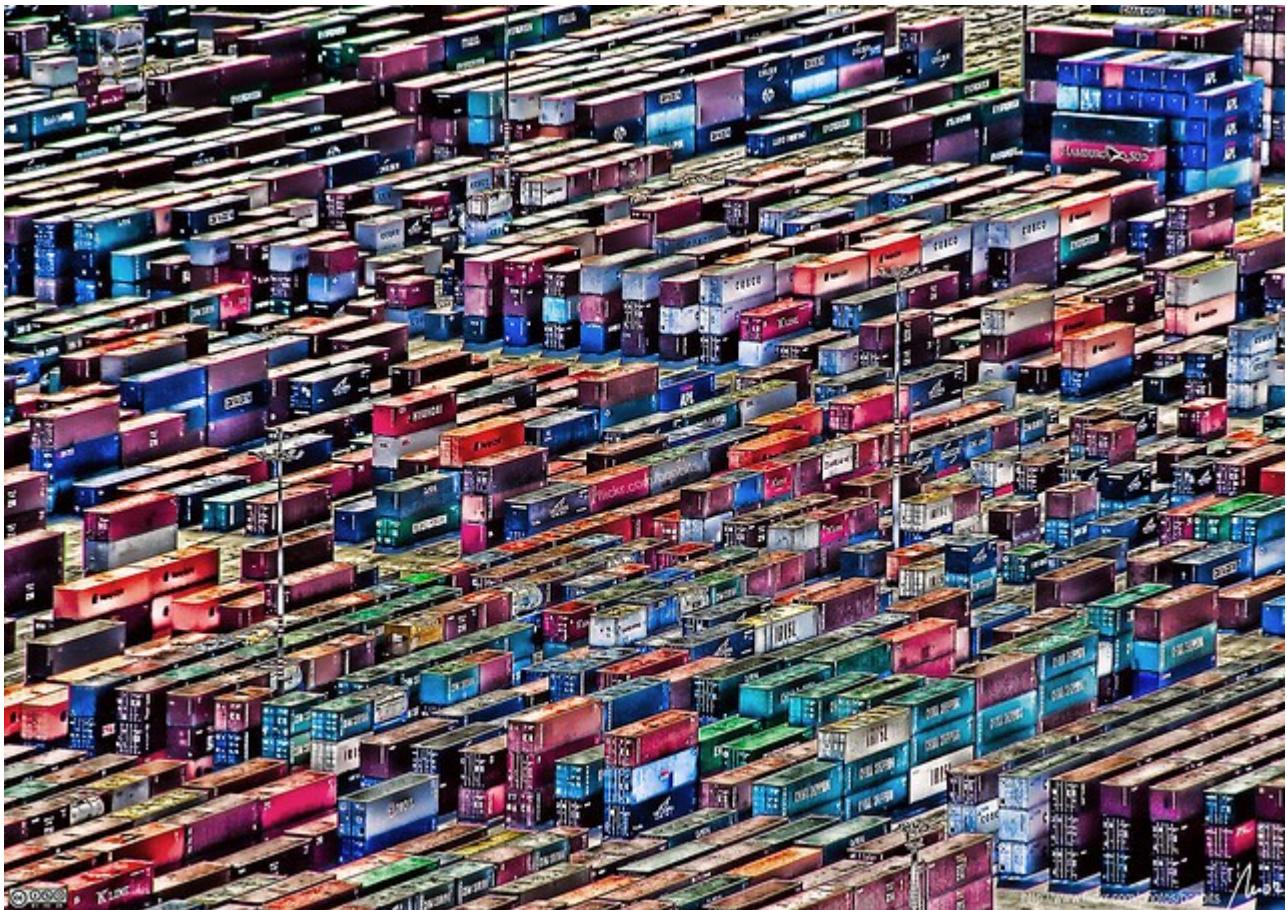
This will pull down the training/namerimage and add it to our Docker host.

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 #dockerand #docker-dev
- Docker on Twitter - <http://twitter.com/docker>
- Get Docker help on Stack Overflow - <http://stackoverflow.com/search?q=docker>

Thank You

