These Dockerfiles are the best place to learn the Dockerfile as its verified from Docker Inc.
One of the best practice is that we combine multiple commands in RUN instruction by using **&&**
Every RUN instruction creates layer on the images, so if you have ten RUN instruction that creates ten extra layers in your image. To avoid this, we can right all the commands in single RUN instruction by combining all the commands with && as shown in above screenshot.

A sample mongodb Dockerfile.

```
###########################################################

# Dockerfile to build MongoDB container images

# Based on Ubuntu

###########################################################


# Set the base image to Ubuntu

FROM ubuntu


# File Author / Maintainer

MAINTAINER Example McAuthor


# Update the repository sources list

RUN apt-get update


################## BEGIN INSTALLATION ######################

# Install MongoDB Following the Instructions at MongoDB Docs

# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/


# Add the package verification key

RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10


# Add MongoDB to the repository sources list

RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist
10gen' | tee /etc/apt/sources.list.d/mongodb.list


# Update the repository sources list once more

RUN apt-get update
```

```
# Install MongoDB package (.deb)

RUN apt-get install -y mongodb-10gen


# Create the default data directory

RUN mkdir -p /data/db


##################### INSTALLATION END #####################


# Expose the default port

EXPOSE 27017


# Default port to execute the entrypoint (MongoDB)

CMD ["--port 27017"]


# Set default container command

ENTRYPOINT usr/bin/mongod
```

11. Container Networking Basics

A simple, static web server

Run the Docker Hub image nginx, which contains a basic web server:

```
imran@DevOps:~$ docker run -d -P nginx

b53c99839ccf662b17ff96424352701cc494b788e4d97525d930406b0cfdb237
```

Docker will download the image from the Docker Hub.

-d tells Docker to run the image in the background.

-P tells Docker to make this service reachable from other computers.
(-P is the short version of --publish-all.)

But, how do we connect to our web server now?

Finding our web server port

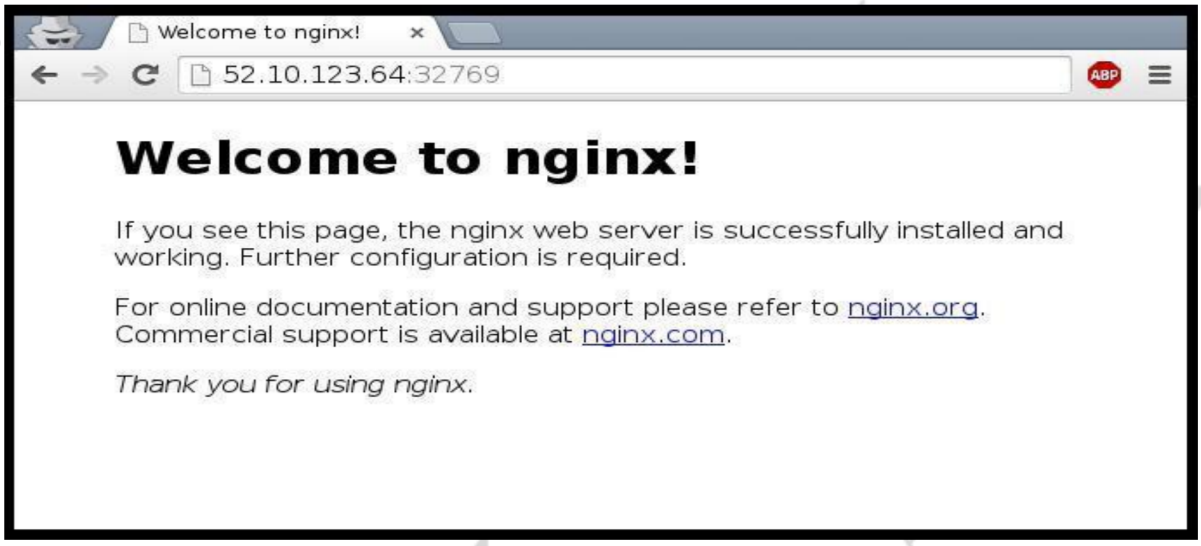
We will use docker ps:

```
imran@DevOps:~$ docker ps

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                   NAMES

b53c99839ccf        nginx               "nginx -g 'daemon ..."   About a minute ago   Up About a
minute   0.0.0.0:32769->80/tcp   distracted_bassi
```

The web server is running on ports 80 and 443 inside the container.Those ports are mapped to ports 32769 and 32768 on our Docker host.We will explain the whys and hows of this port mapping.
But first, let's make sure that everything works properly.

Connecting to our web server (GUI)

Point your browser to the IP address of your Docker host, on the port shown by docker ps for container port 80.



Connecting to our web server (CLI)

You can also use curl directly from the Docker host.

Make sure to use the right port number if it is different from the example below:

```
$ curl localhost:32769

<!DOCTYPE html> <html>

<head>

<title>Welcome to nginx!</title>
```

Why are we mapping ports?
We are out of IPv4 addresses. Containers cannot have public IPv4 addresses. They have private addresses. Services have to be exposed port by port. Ports have to be mapped to avoid conflicts.

Finding the web server port in a script

Parsing the output of docker ps would be painful.There is a command to help us:
$ docker port <containerID>
80 32769

Manual allocation of port numbers

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

```
$ docker run -d -p 80:80 nginx

$ docker run -d -p 8000:80 nginx

$ docker run -d -p 8080:80 -p 8888:80 nginx
```

We are running two NGINX web servers. The first one is exposed on port 80.The second one is exposed on port 8000.The third one is exposed on ports 8080 and 8888.Note: the convention is port-on-host: port-on-container.

Plumbing containers into your infrastructure

There are many ways to integrate containers in your network. Start the container, letting Docker allocate a public port for it. Then retrieve that port number and feed it to your configuration.

Pick a fixed port number in advance, when you generate your configuration. Then start your container by setting the port numbers manually.

Use a network plugin, connecting your containers with e.g. VLANs, tunnels...
Enable *Swarm Mode* to deploy across a cluster.

The container will then be reachable through any node of the cluster.

Finding the container's IP address

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

```
$ docker inspect --format '{{


.NetworkSettings.IPAddress }}' <yourContainerID> 172.17.0.3
```

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

Pinging our container

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

```
$ ping <ipAddress>


64 bytes from <ipAddress>: icmp_req=1 ttl=64 time=0.085 ms 64 bytes from
```

The different network drivers

A container can use one of the following drivers:

bridge (default)

none

host

container

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

The default bridge

By default, the container gets a virtual eth0 interface. (In addition to its own private lo loopback interface.)

That interface is provided by a veth pair. It is connected to the Docker bridge.
(Named docker0 by default; configurable with --bridge.)

Addresses are allocated on a private, internal subnet.
(Docker uses 172.17.0.0/16 by default; configurable with –bip.)

Outbound traffic goes through an iptables MASQUERADE rule. Inbound traffic goes through an iptables DNAT rule. The container can have its own routes, iptables rules, etc.

12. The Container Network Model

The Container Network Model

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

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

What's in a network?

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

Creating a network

Let's create a network called dev.

```
$ docker network create dev

27c06defdb9d99002e670935d8fbcd24ef7f24eb3fd1c22b18f383005722a3c4
```

The network is now visible with the network ls command:

```
$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE

d7dfc8dc0972        bridge              bridge              local

27c06defdb9d        dev                 bridge              local

708d4a2432fd        host                host                local

0e356843766c        none                null                local
```

Placing containers on a network

We will create a *named* container on this network. It will be reachable with its name, search.

```
$ docker run -d --name search --net dev elasticsearch
aec474a67e5d12490f503a9ab209cd6074f0b7ddcb12ee783476b3476ad2764a
```

Communication between containers

Now, create another container on this network.

```
$ docker run -ti --net dev alpine sh

Unable to find image 'alpine:latest' locally

latest: Pulling from library/alpine

2aecc7e1714b: Pull complete

Digest:
sha256:0b94d1d1b5eb130dd0253374552445b39470653fb1a1ec2d81490948876e462c

Status: Downloaded newer image for alpine:latest

/ #
```

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

```
/ # ping search

PING search (172.18.0.2): 56 data bytes

64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.136 ms

64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.063 ms

64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.092 ms


3 packets transmitted, 3 received, 0% packet loss, time 2000ms rtt
min/avg/max/mdev = 0.114/0.149/0.221/0.052 ms

root@0ecccdfa45ef:/#
```

Resolving container addresses

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

[root@0ecccdfa45ef /]# cat /etc/hosts
.0ecccdfa45ef
.localhost
::1 localhost ip6-localhost ip6-loopback fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix ff02::1 ip6-allnodes ff02::2 ip6-allrouters
.search
.search.dev

In Docker Engine 1.10, this has been replaced by a dynamic resolver.
(This avoids race conditions when updating /etc/hosts.)

Connecting multiple containers together

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

Running the web server

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

Start the container, exposing all its ports:

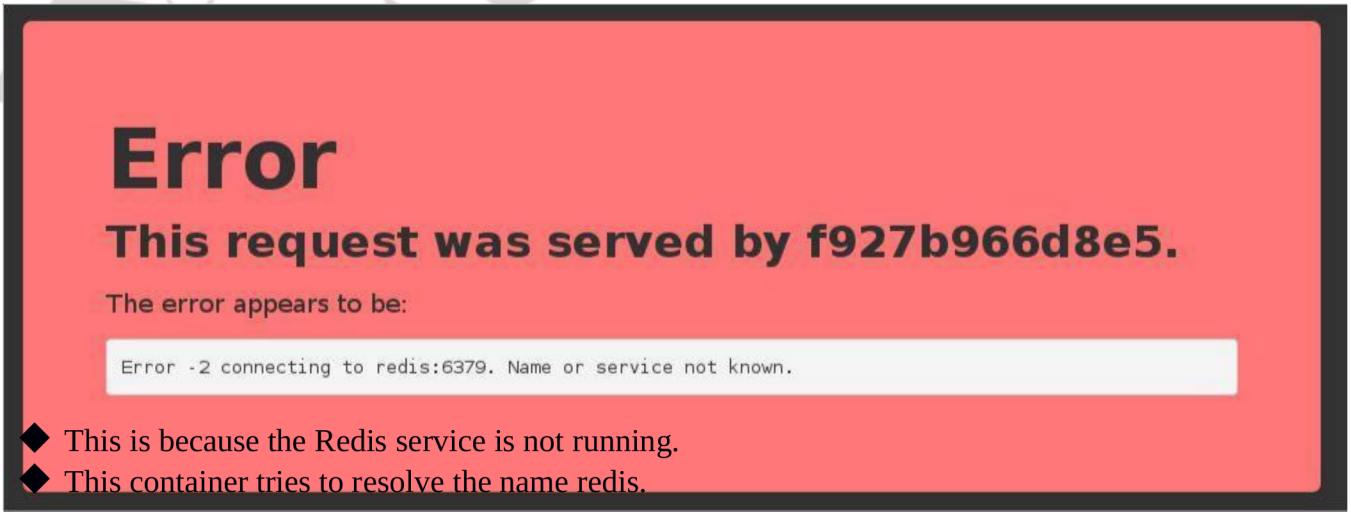
```
$ docker run --net dev -d -P jpetazzo/trainingwheels
```

Check the port that has been allocated to it:

```
$ docker ps -l
```

Test the web server

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



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

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

Start the data store

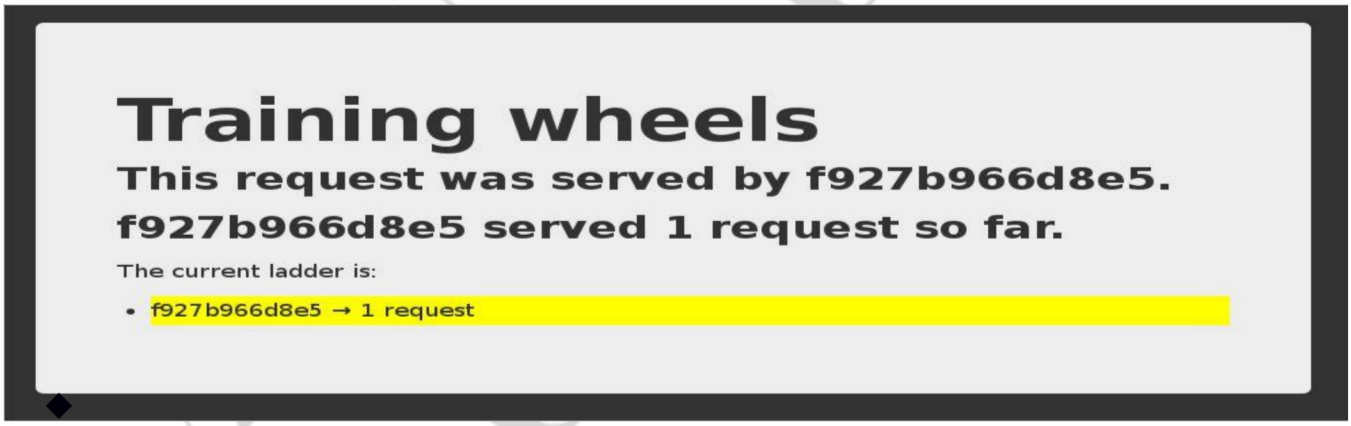
- ✓ We need to start a Redis container.
- ✓ That container must be on the same network as the web server.
- ✓ It must have the right name (redis) so the application can find it.

Start the container:

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

Test the web server again

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



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

A few words on scope

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

Let's remove the redis container:

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

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

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

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

Names are local to each network

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

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

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

Network aliases

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

A container can have multiple network aliases. Network aliases are *local* to a given network (only exist in this network). Multiple containers can have the same network alias (even on the same network). In Docker Engine 1.11, resolving a network alias yields the IP addresses of all containers holding this alias.

Creating containers on another network

Create the prod network.

```
$ docker create network prod

5a41562fecf2d8f115bedc16865f7336232a04268bdf2bd816aecca01b68d5 0c
```

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

```
$ docker run -d --name prod-es-1 --net-alias search --net prod elasticsearch


38079d21caf0c5533a391700d9e9e920724e89200083df73211081c8a356d771
'

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

Resolving network aliases

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

```
$ docker run --net prod --rm alpine nslookup search Name: search
```

Address 1: 172.23.0.3 prod-es-2.prod Address 2: 172.23.0.2 prod-es-1.prod
(You can ignore the can't resolve '(null)' errors.)

Connecting to aliased containers

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

Try the following command a few times:

```
$ docker run --rm --net dev centos curl -s search:9200

{

"name" : "Tarot",

...

}
```

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

```
$ docker run --rm --net prod centos curl -s search:9200

{

"name" : "The Symbiote",

...

}
```

Good to know...

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

13. Local Development Workflow with Docker

Using a Docker container for local development

Never again:

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

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

Our "namer" application

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

Let's run it with:

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

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

Let's look at the code

Let's download our application's source code.

```
$ git clone https://github.com/jpetazzo/namer

$ cd namer $ ls -1 company_name_generator.rbconfig.ru docker-compose.yml
Dockerfile Gemfile
```

Where's my code?

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

```
FROM ruby

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

COPY . /src

WORKDIR /src

RUN bundler install
```

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

EXPOSE 9292
```

We want to make changes *inside the container* without rebuilding it each time.
For that, we will use a volume.

Our first volume

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

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

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

Mounting volumes inside containers

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

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

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

There will be a full chapter about volumes!

Testing the development container

Now let us see if our new container is running.

```
$ docker ps
```

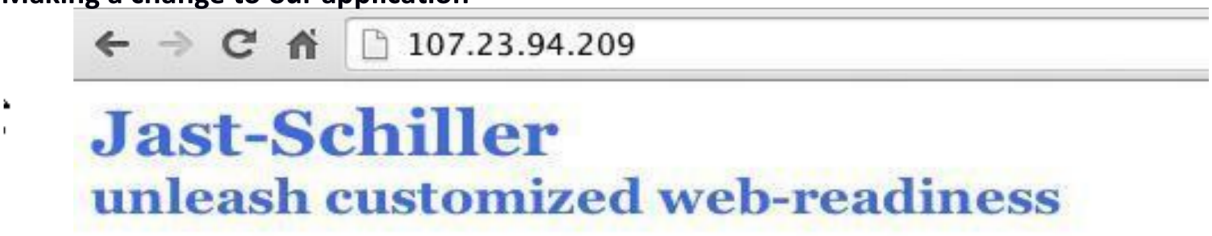
Viewing our application

Now let's browse to our web application on:

```
http://<yourHostIP>:80
```

We can see our company naming application.

Making a change to our application



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

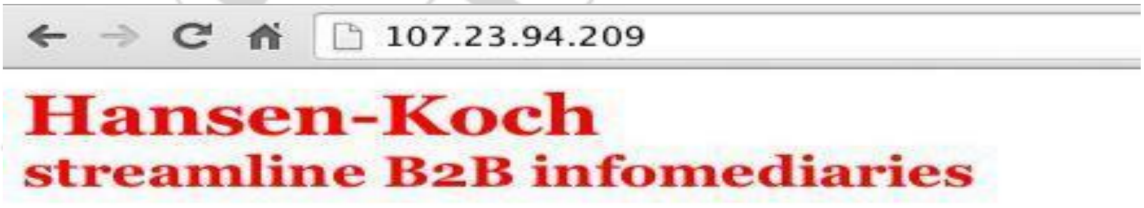
```
$ vi company_name_generator.rb

And change

color: royalblue;

To:  color: red;
```

Refreshing our application

Now let's refresh our browser:

```
http://<yourHostIP>:80
```

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



Improving the workflow with Compose

You can also start the container with the following command: $ docker-compose up -d

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

```
www: build: . volumes:

.:/src

ports:

80:9292
```

Why Compose?

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

Workflow explained

We can see a simple workflow:

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

Debugging inside the container

Docker introduced a feature called docker exec.

It allows users to run a new process in a container which is already running.If sometimes you find yourself wishing you could SSH into a container: you can use docker exec instead.You can get a shell prompt inside an existing container this way, or run an arbitrary process for automation.

docker exec example

```
$ # You can run ruby commands in the area the app is running and more!
```

```
$ docker exec -it <yourContainerId> bash

root@5ca27cf74c2e:/opt/namer# irb

irb(main):001:0> [0, 1, 2, 3, 4].map {|x| x ** 2}.compact => [0, 1, 4, 9, 16]

irb(main):002:0> exit
```

'

Stopping the container

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

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

And remove it

```
$ docker rm ourContainerID>
```

14. Using Docker Compose for Development Stacks

Dockerfiles are great to build a single container. But when you want to start a complex stack made of multiple containers, you need a different tool. This tool is Docker Compose.
In this lesson, you will use Compose to bootstrap a development environment.

Compose for Development Stacks

What is Docker Compose?

Docker Compose (formerly known as fig) is an external tool. It is optional (you do not need Compose to run Docker and containers) but we recommend it highly! The general idea of Compose is to enable a very simple, powerful onboarding workflow:

- Clone your code.
- Run docker-compose up.
- Your app is up and running!

Compose overview

This is how you work with Compose:

- You describe a set (or stack) of containers in a YAML file called docker-compose.yml.
- You run docker-compose up.
- Compose automatically pulls images, builds containers, and starts them.
- Compose can set up links, volumes, and other Docker options for you.
- Compose can run the containers in the background, or in the foreground.
- When containers are running in the foreground, their aggregated output is shown.

Checking if Compose is installed

If you are using the official training virtual machines, Compose has been pre-installed.
You can always check that it is installed by running:

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

Installing Compose

If you want to install Compose on your machine, there are (at least) two methods.
Compose is written in Python. If you have pip and use it to manage other Python packages, you can install compose with:

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

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

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

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

```
$ chmod +x /usr/local/bin/docker-compose
```

Launching Our First Stack with Compose

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

```
$ cd

$ git clone git://github.com/jpetazzo/trainingwheels

...

$ cd trainingwheels
```

Second step: start your app.

```
$ docker-compose up
```

Watch Compose build and run your app with the correct parameters, including linking the relevant containers together.

Launching Our First Stack with Compose

Verify that the app is running at http://<yourHostIP>:8000.