**Getting Started with Docker**

Mar 20, 2014

[deployment](https://serversforhackers.com/tag/deployment) [container](https://serversforhackers.com/tag/container)

If you're interested in more of this type of content, check out the [Servers for Hackers eBook](https://book.serversforhackers.com/)!

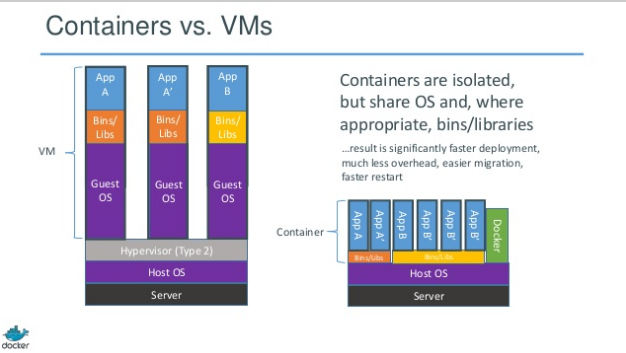
**What is Docker?**

Docker is a Container.

While a Virtual Machine is a whole other guest computer running on top of your host computer (sitting on top of a layer of virtualization), Docker is an isolated portion of the host computer, sharing the host kernel (OS) and even its bin/libraries if appropriate.

To put it in an over-simplified way, if I run a CoreOS host server and have a guest Docker Container based off of Ubuntu, the Docker Container contains the parts that make Ubuntu different from CoreOS.

This is one of my favorite images which describes the difference:



This image is found on [these slides](http://www.slideshare.net/dotCloud/docker-intro-november) provided by Docker.

**Getting Docker**

Docker isn't compatible with Macintosh's kernel unless you install [boot2docker](http://docs.docker.io/en/latest/installation/mac/#macosx). I avoid that and use CoreOS in Vagrant, which comes with Docker installed.

I highly recommend [CoreOS](https://coreos.com/) as a host machine for your play-time with Docker. They are building a lot of [awesome tooling](https://coreos.com/using-coreos/) around Docker.

My Vagrantfile for CoreOS is as follows:

config.vm.box = "coreos"

config.vm.box\_url = "http://storage.core-os.net/coreos/amd64-generic/dev-channel/coreos\_production\_vagrant.box"

config.vm.network "private\_network",

ip: "172.12.8.150"

If you like NFS, then perhaps use these settings, which share with CoreOS's writable directory:

# This will require sudo access when using "vagrant up"

config.vm.synced\_folder ".", "/home/core/share",

id: "core",

:nfs => true,

:mount\_options => ['nolock,vers=3,udp']

If you have VMWare instead of Virtualbox:

config.vm.provider :vmware\_fusion do |vb, override|

override.vm.box\_url = "http://storage.core-os.net/coreos/amd64-generic/dev-channel/coreos\_production\_vagrant\_vmware\_fusion.box"

end

And finally, fixing a Vagrant plugin conflict:

# plugin conflict

if Vagrant.has\_plugin?("vagrant-vbguest") then

config.vbguest.auto\_update = false

end

If you're **not** using CoreOS, then check out [this page](https://www.docker.io/gettingstarted/) with install instructions for other flavors of Linux. **Note:** Ubuntu is what Docker develops on, so that's a safe bet.

If you are using CoreOS, dont be dismayed when it tries to restart on you. It's a "feature", done during auto-updates. You may, however, need to run vagrant reload to restart the server so Vagrant set up file sync and networking again.

**Your First Container**

This is the unfortunate baby-step which everyone needs to take to first get their feet wet with Docker. This *won't* show what makes Docker powerful, but it does illustrate some important points.

Docker has a concept of "base containers", which you use to build off of. After you make changes to a base container, you can save those change and commit them. You can even push your boxes up to [index.docker.io](http://index.docker.io).

One of Docker's most basic images is just called "Ubuntu". Let's run an operation on it.

If the image is not already downloaded in your system, it will download it first from the "Ubuntu repository". Note the use of similar terminology to version control systems such as Git.

Run Bash:

docker run ubuntu /bin/bash

Nothing happened! Well, actually it did. Run docker ps (similar to our familiar Linux ps) - you'll see no containers listed, as none are currently running. Run docker ps -a, however, and you'll see an entry!

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

8ea31697f021 ubuntu:12.04 /bin/bash About a minute ago Exit 0 loving\_pare

So, we can see that docker did run /bin/bash, but there wasn't any running process to keep it alive. **A Docker container only stays alive as long as there is an active process being run in it**.

Keep that in mind for later. Let's see how we can run Bash and poke around the Container. This time run:

docker run -t -i ubuntu /bin/bash

You'll see you are now logged in as user "root" and can poke around the container!

What's that command doing?

* docker run - Run a container
* -t - Allocate a (pseudo) [tty](http://en.wikipedia.org/wiki/Computer_terminal)
* -i - Keep stdin open (so we can interact with it)
* ubuntu - use the Ubuntu base image
* /bin/bash - Run the bash shell

**Tracking Changes**

Exit out of that shell (ctrl+d or type exit) and run docker ps -a again. You'll see some more output similar to before:

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

30557c9017ec ubuntu:12.04 /bin/bash About a minute ago Exit 127 elegant\_pike

8ea31697f021 ubuntu:12.04 /bin/bash 22 minutes ago Exit 0 loving\_pare

Copy and paste the most recent Container ID (30557c9017ec in my case). Use that ID and run docker diff <container id>. For me, I see:

core@localhost ~ $ docker diff 30557c9017ec

A /.bash\_history

C /dev

A /dev/kmsg

We can see that just by logging into bash, we created a .bash\_history file, a /dev directory and a /dev/kmsg file. Minor changes, but tracked changes never the less! Docker tracks any changes we make to a container. In fact, Docker allows us make changes to an image, commit those changes, and then push those changes out somehwere. **This is the basis of how we can deploy with Docker.**

Let's install some base items into this Ubuntu install and save it as our own base image.

# Get into Bash

docker run -t -i ubuntu /bin/bash

# Install some stuff

apt-get update

apt-get install -y git ack-grep vim curl wget tmux build-essential python-software-properties

Once that finishes running, exit and run docker ps -a again. Grab the latest container ID and run another diff (docker diff <Container ID>):

core@localhost ~ $ docker diff 5d4bdae290a4

> A TON OF FILE LISTED HERE

There were, of course, lots of new files added. Let's save this version of our base image so we can use it later. We'll commit these changes, name this image and tag it in one go. We'll use: docker commit <Container ID> <Name>:<Tag>

core@localhost ~ $ docker commit 5d4bdae290a4 fideloper/docker-example:0.1

c07e8dc7ab1b1fbdf2f58c7ff13007bc19aa1288add474ca358d0428bc19dba6 # You'll get a long hash as a Success message

Let's see this image we just created. Run docker images:

core@localhost ~ $ docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

fideloper/docker-example 0.1 c07e8dc7ab1b 22 seconds ago 455.1 MB

ubuntu 13.10 9f676bd305a4 6 weeks ago 178 MB

ubuntu saucy 9f676bd305a4 6 weeks ago 178 MB

ubuntu 13.04 eb601b8965b8 6 weeks ago 166.5 MB

ubuntu raring eb601b8965b8 6 weeks ago 166.5 MB

ubuntu 12.10 5ac751e8d623 6 weeks ago 161 MB

ubuntu quantal 5ac751e8d623 6 weeks ago 161 MB

ubuntu 10.04 9cc9ea5ea540 6 weeks ago 180.8 MB

ubuntu lucid 9cc9ea5ea540 6 weeks ago 180.8 MB

ubuntu 12.04 9cd978db300e 6 weeks ago 204.4 MB

ubuntu latest 9cd978db300e 6 weeks ago 204.4 MB

ubuntu precise 9cd978db300e 6 weeks ago 204.4 MB

You'll notice a ton of Ubuntu's here. Since I first used the Ubuntu base image, it loaded knowledge of the variously tagged versions of Ubuntu they have available on the [Docker index](https://index.docker.io/_/ubuntu/).

More excitingly, however, is that we also have our own image fideloper/docker-example and its tag 0.1!

**Building a Server with Dockerfile**

Let's move onto building a static web server with a Dockerfile. The Dockerfile provides a set of instructions for Docker to run on a container. This lets us automate installing items - we could have used a Dockerfile to install git, curl, wget and everything else we installed above.

Create a new directory and cd into it. Because we're installing Nginx, let's create a default configuration file that we'll use for it.

Create a file called default and add:

server {

root /var/www;

index index.html index.htm;

# Make site accessible from http://localhost/

server\_name localhost;

location / {

# First attempt to serve request as file, then

# as directory, then fall back to index.html

try\_files $uri $uri/ /index.html;

}

}

That's about as basic as it gets for Nginx.

Next, create a file named Dockerfile and add the following, changing the FROM section as suitable for whatever you named your image:

FROM fideloper/docker-example:0.1

RUN echo "deb http://archive.ubuntu.com/ubuntu precise main universe" > /etc/apt/sources.list

RUN apt-get update

RUN apt-get -y install nginx

RUN echo "daemon off;" >> /etc/nginx/nginx.conf

RUN mkdir /etc/nginx/ssl

ADD default /etc/nginx/sites-available/default

EXPOSE 80

CMD ["nginx"]

What's this doing?

* FROM will tell Docker what image (and tag in this case) to base this off of
* RUN will run the given command (as user "root") using sh -c "your-given-command"
* ADD will copy a file from the host machine into the container
  + This is handy for configuration files or scripts to run, such as a process watcher like supervisord, systemd, upstart, forever (etc)
* EXPOSE will expose a port to the host machine. You can expose multiple ports like so: EXPOSE 80 443 8888
* CMD will run a command (not using sh -c). This is usually your long-running process. In this case, we're simply starting Nginx.
  + In production, we'd want something watching the nginx process in case it fails

Once that's saved, we can build a new image from this Dockerfile!

docker build -t nginx-example .

If that works, you'll see Successfully built 88ff0cf87aba (your new container ID will be different).

Check out what you have now, run docker images:

core@localhost ~/webapp $ docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

nginx-example latest 88ff0cf87aba 35 seconds ago 468.5 MB

fideloper/docker-example 0.1 c07e8dc7ab1b 29 minutes ago 455.1 MB

...other Ubuntu images below ...

Also run docker ps -a:

core@localhost ~/webapp $ docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

de48fa2b142b 8dc0de13d8be /bin/sh -c #(nop) CM About a minute ago Exit 0 cranky\_turing

84c5b21feefc 2eb367d9069c /bin/sh -c #(nop) EX About a minute ago Exit 0 boring\_babbage

3d3ed53987ec 77ca921f5eef /bin/sh -c #(nop) AD About a minute ago Exit 0 sleepy\_brattain

b281b7bf017f cccba2355de7 /bin/sh -c mkdir /et About a minute ago Exit 0 high\_heisenberg

56a84c7687e9 fideloper/docker-e... /bin/sh -c #(nop) MA 4 minutes ago Exit 0 backstabbing\_turing

... other images ...

What you can see here is that **for each line in the Dockerfile, a new container (and commit sha) is produced** if that line results in a change to the image used. Similar(ish) to version control! (Also, how funny is the name "backstabbing\_turing"?)

**Finally, run the web server**

Let's run this web server! Use docker run -p 80:80 -d nginx-example (assuming you also named yours "nginx-example" when building it).

The -p 80:80 binds the Container's port 80 to the guest machines, so if we curl localhost or go to the server's IP address in our browser, we'll see the results of Nginx processing requests on port 80 in the container.

core@localhost ~/webapp $ docker run -d nginx-example

73750fc2a49f3b7aa7c16c0623703d00463aa67ba22d2108df6f2d37276214cc # Success!

core@localhost ~/webapp $ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

a085a33093f4 nginx-example:latest nginx 2 seconds ago Up 2 seconds 80/tcp determined\_bardeen

Note we ran docker ps instead of docker ps -a - We're seeing a currently active and running Docker container. Let's curl localhost:

core@localhost ~/webapp $ curl localhost/index.htmld

<html>

<head><title>500 Internal Server Error</title></head>

<body bgcolor="white">

<center><h1>500 Internal Server Error</h1></center>

<hr><center>nginx/1.1.19</center>

</body>

</html>

Well, we're sorta there. Nginx is working (woot!) but we're getting a 500 error. That's likely because there's no default index.html file for Nginx to fall back onto. Let's stop this docker instance via docker stop <container id>:

core@localhost ~/webapp $ docker stop a085a33093f4

a085a33093f4

To fix this, let's share a directory in between the Container and our host machine. First, create an index.html page from where we'll share it.

# I'm going to be sharing the /home/core/share directory on my CoreOS machine

echo "Hello, Docker fans!" >> /home/core/share/index.html

Then we can start our Docker container again:

docker run -v /home/core/share:/var/www:rw -p 80:80 -d nginx-example

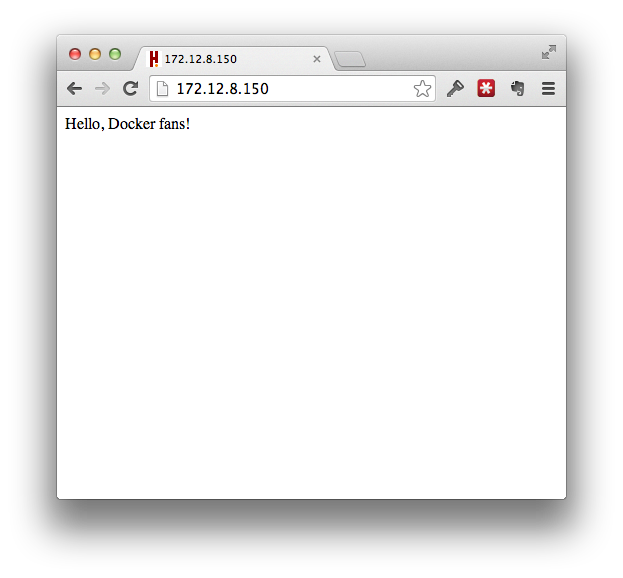
* docker run - Run a container
* -v /path/to/host/dir:/path/to/container/dir:rw - The volumes to share. Note rw is read-write. We can also define ro for "read-only".
* -p 80:80 - Bind the host's port 80 to the containers.
* -d nginx-example Run our image nginx-example, which has the "CMD" setup to run nginx

Now run curl localhost:

core@localhost ~/webapp $ curl localhost

Hello, Docker fans!

...or better yet, point your browser to your server's IP address!



Note that the IP address I used is that of my CoreOS server. I set the IP address in my Vagrantfile. I don't need to know the IP address given to my Container in such a simple example, altho I can find it by running docker inspect <Container ID>.

core@localhost ~/webapp $ docker inspect a0b531aa00f4

[{

"ID": "a0b531aa00f475b0025d8edce09961077eedd82a190f2e2f862592375cad4dd5",

"Created": "2014-03-20T22:38:22.452820503Z",

... a lot of JSON ...

"NetworkSettings": {

"IPAddress": "172.17.0.2",

"IPPrefixLen": 16,

"Gateway": "172.17.42.1",

"Bridge": "docker0",

"PortMapping": null,

"Ports": {

"80/tcp": [

{

"HostIp": "0.0.0.0",

"HostPort": "80"

}

]

}

},

... more JSON ...

}]

**Linking Containers**

Not fully covered here (for now) is the [ability to link two or more containers](http://docs.docker.io/en/latest/use/working_with_links_names/) together. This is handy if containers need to communicate with eachother. For example, if your application container needs to communiate with a database container. Linking lets you have some infrastrcture be separate from your application.

For example:

Start a container and name it something useful (in this case, mysql, via the -name parameter):

docker run -p 3306:3306 -name mysql -d some-mysql-image

Start your web application container and link it to that container via -d name:db (where db is an arbitrary name used in the container's environment variables):

docker run -p 80:80 -link mysql:db -d some-application-image

In this example, the some-application-image will have environment variables available such as DB\_PORT\_3306\_TCP\_ADDR=172.17.0.8 and DB\_PORT\_3306\_TCP\_PORT=3306 which you application can use to know the database location.

Here's an example of a [MySQL Dockerfile](https://github.com/fideloper/docker-mysql)

**The Epic Conclusion**

So, we fairly easily can build servers, add in our application code, and then ship our working applications off to a server. Everything in the environment is under your control.

In this way, we can actually **ship the entire environment instead of just code**.

**P.S. - Tips and Tricks**

**Cleaning Up**

If you're like me and make tons of mistakes and don't want the record of all your broken images and containers lying around, you can clean them them:

* To remove a container: docker rm <Container ID>
* To remove all containers: docker rm $(docker ps -a -q)
* To remove images: docker rmi <Container ID>
* To remove all images: docker rmi $(docker ps -a -q)

Note: You must remove all containers using an image before deleting the image

**Base Images**

I always use [Phusian's Ubuntu base image](https://github.com/phusion/baseimage-docker). It installs/enables a lot of items you may not think of, such as the CRON daemon, logrotate, ssh-server (you want to be able to ssh into your server, right?) and other important items. Read the Readme file of that project to learn more.

Especially note Phusion's use of an "insecure" SSH key to get you started with the ability to SSH into your container and play around, while it runs another process such as Nginx.

[Orchard](https://orchardup.com/), creators of [Fig](http://orchardup.github.io/fig/), also have a ton of [great images to use](https://github.com/orchardup) to learn from. Also, Fig looks like a really nice way to handle Docker images for development environments.

**Want more?**

If articles like this interest you, sign up for the **newsletter**!  
Join nearly 10000 others on the mailing list!

Top of Form

Bottom of Form

Top of Form

Bottom of Form