Docker JumpStart

Andrew Odewahn

Table of Contents

1 Introduction

Docker is a tool for packaging and shipping apps. Based on the idea of a shipping container, it provides a standardized way for developers or system administrators to create lightweight images, or collections of images, for each element of an application, and then easily and quickly deploy the image. Since the image is standardized, it can be uniformly deployed on development or production, leading to a much simpler workflow, faster development time for the dev team, and lower management overhead for the ops team.

First, a quick overview of a few things Docker *is*:

- An open source tool that places a layer on top of Linux containers (LXC) to make it simple to package and ship complex apps
- A tool for creating a layered filesystem; each layer is versioned and can be shared across running instances, making for much more lightweight deployments
- A company behind the project, as well as a site called the "Docker Hub" for sharing containers

And, a few things Docker *isn't*:

- A virtual machine. Unlike a true VM, a docker container does not also require a host OS, meaning it's much slimmer than a real VM
- An infrastructure automation tool, like Puppet or Chef. Like those other tools, Docker is a major player in the DevOps space, but its focus is around running apps in a container environment, as opposed to representing a machine state.

This guide introduces the key ideas you'll use again and again in Docker, such as images, layers, containers, commits, tags, and so forth. The main things to understand include:

- an *image* is a specific state of a filesystem
- an image is composed of *layers* representing changes in the filesystem at various points in time; layers are a bit like the commit history of a git repository

- a *container* is a running process that is started based on an image
- you can change the state of the filesystem on a container and commit it to create a new image
- changes in memory / state are not committed -- only changes on the file-system

Docker uses the git-style command format:

```
$ docker [OPTIONS] COMMAND [arg...]
```

The following table, taken from the "docker help" command, provides a quick summary of the commands.

Command	Description
attach	Attach to a running container
build	Build an image from a Dockerfile
commit	Create a new image from a container's changes
cp	Copy files/folders from a container's filesystem to the host path
diff	Inspect changes on a container's filesystem
events	Get real time events from the server
export	Stream the contents of a container as a tar archive
history	Show the history of an image
images	List images
import	Create a new filesystem image from the contents of a tarball
info	Display system-wide information
inspect	Return low-level information on a container
kill	Kill a running container
load	Load an image from a tar archive
login	Register or log in to the Docker registry server
logs	Fetch the logs of a container
port	Lookup the public-facing port that is NAT-ed to PRIVATE_PORT
pause	Pause all processes within a container
ps	List containers

Command	Description
pull	Pull an image or a repository from a Docker registry server
push	Push an image or a repository to a Docker registry server
restart	Restart a running container
rm	Remove one or more containers
rmi	Remove one or more images
run	Run a command in a new container
save	Save an image to a tar archive
search	Search for an image on the Docker Hub
start	Start a stopped container
stop	Stop a running container
tag	Tag an image into a repository
top	Lookup the running processes of a container
unpause	Unpause a paused container
version	Show the Docker version information
wait	Block until a container stops, then print its exit code

As you scan the list (which I've left in alphabetical order), you'll notice 3 key groups of commands:

- Commands related to managing *images*, such as *images*, *build*, *save*, *rmi*, and *tag*.
- Commands related to *containers*, such as *run*, *ps*, *kill*, *restart*, *top*, and *pause*.
- Commands related to the **Docker hub** (https://hub.docker.com/), such as *login*, *search*, *pull*, and *push*. (More on this in the next chapter.)

This guide is organized along these same basic lines. First, we'll cover how to get Docker up and running on a Mac or Windows machine using a tool called boot2docker. (If you're using a Linux-based machine, you can skip this chapter.) Next, we'll talk about images and layers, and then talk about containers, which are the "running" instances of an image. Next, we'll talk about the Docker Hub, which is a place where you can post your images, as well as find images maintained by other Docker users.

Although the examples might be a bit contrived, the goal of the guide is to give you a quick introduction to the ideas you'll use again and again as you explore Docker in depth.

Using boot2docker to run Docker on a Mac or Windows

2

Docker's underlying containerization technology only works on Linux. To use it on a Mac or Windows, you must use a command-line tool called **boot2docker** (http://boot2docker.io/) that installs a Linux virtual machine (or "VM") on your system. If you're unfamiliar with VMs, they let you run a "guest" operating system (like Linux) on your "host" operating system (like a Mac or Windows). For all intents and purposes, the VM is another computer running "inside" your computer -- it has its own operating system, filesystem, and network that can piggyback on the resources of the host OS in an isolated environment.

So, the idea behind boot2docker is really quite clever: since we can't run Docker natively on Windows or a Mac, we install a bare-bones Linux VM that *can* run Docker, and then we communicate with it using a Docker client running on our host (i.e., the terminal on our main OS). If you've ever used a package like Postgres or MySql, it's exactly the same idea: you have a client tool that you use to issue commands to another server. The main difference is that our "docker" process is running inside a VM, rather than as a native service.

This chapter will walk you through the installation process. Since both Docker and boot2docker are changing so rapidly, rather than describe the detailed steps here, I've provided an overview of the process so that you'll understand what's happening.

Install boot2docker

Before you start, it's a good idea to first set up SSH on your host machine so that boot2docker can install your login credentials on the VM. Otherwise, you'll be prompted for a password whenever you log in. If you're unfamiliar with SSH or SSH keys, Appendix ??? will give you details of how to get set up.

First, download the installation package for your system. You can find **Mac** (http://docs.docker.com/installation/mac/) and **Windows** (http://docs.docker.com/installation/windows/) installation instructions on the Docker's **docu-**

mentation site (http://docs.docker.com/). Next, you run the install procedure, which will install the boot2docker command line tool as well **VirtualBox** (https://www.virtualbox.org/), a free tool from Oracle that allows your computer to run virtual machines.

Once the basic tools are installed, you're ready to create the box by running *boot2docker init*. (Think of this as purchasing a new computer.) This will create a new Linux "box" called "boot2docker-vm" on your machine that has Docker installed, as well as configure some of the basic things you'll need to communicate with it from your host. The command will produce something like this:

```
$ boot2docker init
2014/08/11 13:30:58 Creating VM boot2docker-vm...
2014/08/11 13:30:58 Apply interim patch to VM boot2docker-vm
(https://www.virtualbox.org/ticket/12748)
2014/08/11 13:30:58 Setting NIC #1 to use NAT network...
2014/08/11 13:30:58 Port forwarding [ssh] tcp://127.0.0.1:2022 --
> :22
2014/08/11 13:30:58 Port forwarding [docker] tcp://127.0.0.1:2375
--> :2375
2014/08/11 13:30:58 Setting NIC #2 to use host-only network "vbox-
net4"...
2014/08/11 13:30:59 Setting VM storage...
2014/08/11 13:31:07 Done. Type `boot2docker up` to start the VM.
```

Once the box is created (it can take a few minutes because it has to download the image), you run *boot2docker up* to boot up the box. Think of this as turning your the new computer on. The box will start the Docker daemon in the VM that listens for requests from the our client on our host. Here's an example:

```
$ boot2docker up
2014/08/11 13:38:00 Waiting for VM to be started...
............
2014/08/11 13:38:36 Started.
2014/08/11 13:38:36 To connect the Docker client to the Docker dae-
mon, please set:
2014/08/11 13:38:36     export DOCKER_HOST=tcp://
192.168.59.104:2375
```

Finally, you'll need to set an environment variable called DOCKER_HOST that will tell your Docker client on your host machine the URI for the Docker daemon running on the VM. This address is the last line returned from the "boot2docker up" command. For example, here's what we'd run based on the output we got in the previous step:

```
$ export DOCKER_HOST=tcp://192.168.59.104:2375
```

Since running the export command in the shell will only set the environment variable temporarily, you should follow the process on your OS to make it permanent. On a Mac, **add the following line to the ~/.bash_profile** (http://stackoverflow.com/questions/22502759/mac-os-x-10-9-setting-permanent-environment-variables):

```
export DOCKER_HOST=tcp://$(boot2docker ip 2>/dev/null):2375
```

On windows, **use the "environment variables" setting of the "Advanced system settings" tab** (http://stackoverflow.com/questions/17312348/how-do-i-set-windows-environment-variables-permanently)

Once you've completed all these steps, you should have the Docker client installed and a VM capable of starting Docker containers. To test it out, try the command "docker run hello-world" from your host:

```
$ docker run hello-world
Unable to find image 'hello-world' locally
Pulling repository hello-world
565a9d68a73f: Pulling image (latest) from hello-world, endpoint:
https://cdn-reg565a9d68a73f: Download complete
511136ea3c5a: Download complete
2505d942a91d: Download complete
Hello from Docker.
...
```

As you can see from the output, this command has pulled down the **hello-world** (https://registry.hub.docker.com/_/hello-world/) repository from (more on this in a later chapter) and printed the text "Hello from Docker." If you're seeing this message, then you've successfully installed boot2docker. Alternatively, if you get a message like the following, you should doublecheck that you've set the DOCKER_HOST environment variable correctly:

```
$ docker run hello-world
2014/08/11 15:05:55 Post http:///var/run/docker.sock/v1.13/contain-
ers/create: dial unix /var/run/docker.sock: no such file or direc-
tory
```

Upgrading from an older install

If you've installed an older version of boot2docker, you can (and should!) use boot2docker itself to update the package. The commands are:

```
$ boot2docker stop
$ boot2docker download
$ boot2docker start
```

boot2docker Quick Reference

boot2docker uses the increasingly common “git"-style format:

```
$ boot2docker [<options>] <command> [<args>]
```

For example, here’s how to get the VM’s current status:

```
$ boot2docker status
running
```

The following table, taken from “boot2docker help”, summarizes the various commands.

Command	Description
init	Create a new boot2docker VM. You should only do this once to create a box.
up	Starts the VM and the Docker daemon
status	Get the current state of VM, such as “running,” “saved,” or “poweroff”.
ssh [ssh-command]	Start a shell on the VM using SSH. If you have not set up an SSH key, you’ll be prompted for a password. (It’s “tcuser”)
save / suspend	Suspend the VM and save state to disk.
down / stop / halt	Gracefully shutdown the VM.
restart	Gracefully reboot the VM.
delete / destroy	Delete the boot2docker VM and its disk image.
config / cfg	Displays selected settings for the VM, such as its memory size or IP address
help	Displays the commands in this table
info	A json dump of profile settings
ip	The IP address of the VM’s Host-only network; this can be used to set the DOCKER_HOST environment variable
download	Download’s the current boot2docker ISO image. (An ISO is an archive format based on old CDROM standard.)
upgrade	Upgrade the boot2docker ISO image (if vm is running it will be stopped and started).

Command	Description
version	Display the current version of boot2docker. It changes often, so be sure to update frequently!

You can also provide a variety of options to further customize the VM, such as setting the amount of memory or the name of the box itself. Most of these options appear to only work when the image is created (i.e., they only apply then you're doing "boos2docker init"). The following table summarizes the options from "boot2docker help". (The actual output from the command shows the default settings for each option.)

Option	Description
--basevmdk	Path to VMDK to use as base for persistent partition
--dhcp	enable VirtualBox host-only network DHCP.
--dhcpip	VirtualBox host-only network DHCP server address.
--disksize	boot2docker disk image size (in MB).
--dockerport	host Docker port (forward to port 2375 in VM).
--hostip	VirtualBox host-only network IP address.
--iso	path to boot2docker ISO image.
--lowerip	VirtualBox host-only network DHCP lower bound.
--memory	virtual machine memory size (in MB).
--netmask	VirtualBox host-only network mask.
--serial	try serial console to get IP address (experimental)
--serialfile	path to the serial socket/pipe.
--ssh	path to SSH client utility.
--ssh-keygen	path to ssh-keygen utility.
--sshkey	path to SSH key to use.
--sshport	host SSH port (forward to port 22 in VM).
--upperip	VirtualBox host-only network DHCP upper bound.
--vbm	path to VirtualBox management utility.
--verbose	display verbose command invocations.
--vm	virtual machine name.

Here's an example of how you'd created an image that used 4GB, rather than the default 2GB.

```
$ boot2docker --memory=4096 init
2014/08/11 17:15:50 Creating VM boot2docker-vm...
2014/08/11 17:15:50 Apply interim patch to VM boot2docker-vm
(https://www.virtualbox.org/ticket/12748)
2014/08/11 17:15:50 Setting NIC #1 to use NAT network...
2014/08/11 17:15:51 Port forwarding [ssh] tcp://127.0.0.1:2022 --
> :22
2014/08/11 17:15:51 Port forwarding [docker] tcp://127.0.0.1:2375
--> :2375
2014/08/11 17:15:51 Setting NIC #2 to use host-only network "vbox-
net4"...
2014/08/11 17:15:51 Setting VM storage...
2014/08/11 17:15:59 Done. Type `boot2docker up` to start the VM.

$ boot2docker info | python -m json.tool
{
    "BaseFolder": "/Users/odewahn/VirtualBox/VMs/boot2docker-vm",
    "BootOrder": null,
    "CPUs": 4,
    "CfgFile": "/Users/odewahn/VirtualBox/VMs/boot2docker-vm/
boot2docker-vm.vbox",
    "DockerPort": 2375,
    "Flag": 0,
    "Memory": 4096,
    "Name": "boot2docker-vm",
    "OSType": "",
    "SSHPort": 2022,
    "SerialFile": "/Users/odewahn/.boot2docker/boot2docker-
vm.sock",
    "State": "poweroff",
    "UUID": "db24beb1-5b83-4744-9f5f-ce05b41eda91",
    "VRAM": 8
}
```

Opening a port with VirtualBox

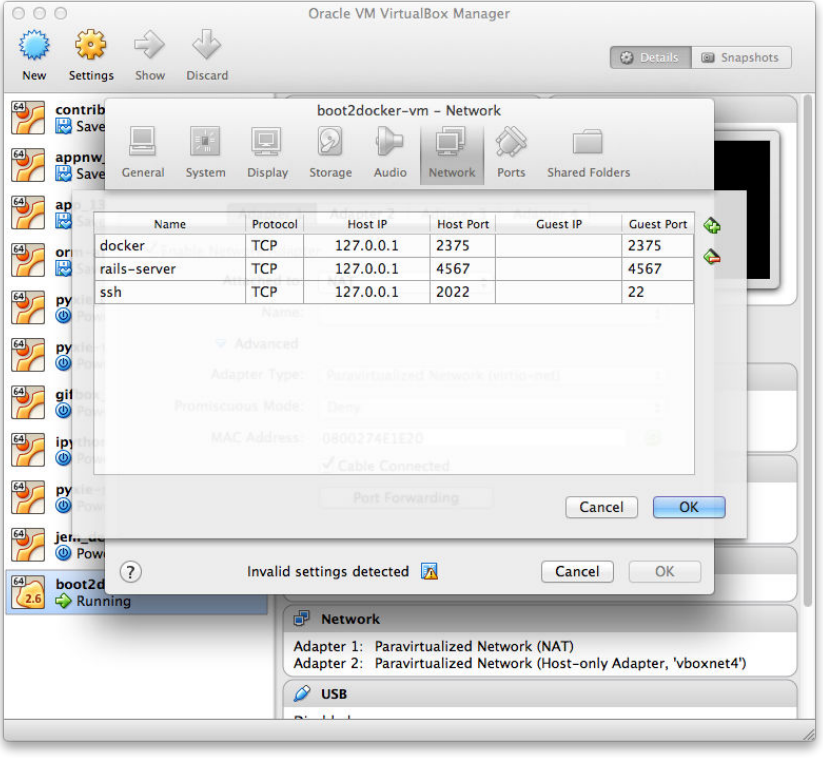
By and large, boot2docker hides the nitty gritty details of working directly with **VirtualBox** (https://www.virtualbox.org/), the underlying software that allows us to run the VMs. However, there's one particular situation -- opening a port from your VM to your host machine -- where you'll need to use the VirtualBox itself. The most common use case for doing this is when you are developing some custom app that is running within Docker, but you want to connect with it in your browser.

For example, say you're writing a Rails application. You might pull down the a **Docker image that has a Ruby and Rails environment** (https://registry.hub.docker.com/u/stackbrew/rails/) installed, start the local development server, and then try to connect to **localhost:4567** to see the output, like you would in a "normal" environment. By default, however, boot2docker only exposes a few ports on the VM (port 2022 for SSH and port 2375 for the boot2docker API), so instead of seeing your app, you'll only see a "Cannot connect" page.

Fortunately, the fix for this is quite simple. VirtualBox has a command line utility called "VBoxManage" that you can use to customize the box as it runs. To expose a port, type something like this into your terminal:

```
$ VBoxManage controlvm boot2docker-vm natpf1 "rails-server,tcp,
127.0.0.1,4567,,4567"
```

This will create a port forwarding rule that in VirtualBox that will last until you destroy the box. You can also create this rule in the VirtualBox GUI tool in the "Settings -> Network -> Port Forwarding" screen, as shows in the next figure.



You can find out more about this tool in the extensive **VirtualBox documentation** (https://www.virtualbox.org/manual/UserManual.html) site.

Images: Layered filesystems 3

A Docker *image* represents a snapshot of a filesystem at a certain point in time. As mentioned in the introduction, the image is composed of layers that progressively stack on top of each other; containers (running instances of an image) can share these layers among them, which is one reason Docker is so much lighter weight than a full VM, where nothing is generally shared.

Perhaps the best way to start (after you get Docker installed, of course!) is to use "docker pull" to grab the latest release of Ubuntu (we'll talk more about where you're actually pulling from in the chapter on the Docker Hub):

```
$ docker pull ubuntu
Pulling repository ubuntu
c4ff7513909d: Download complete
511136ea3c5a: Download complete
1c9383292a8f: Download complete
9942dd43ff21: Download complete
d92c3c92fa73: Download complete
0ea0d582fd90: Download complete
cc58e55aa5a5: Download complete
...
```

As you pull the image, you'll see the progress of each dependent layer being downloaded. Once all the layers are finished downloading (and there are a LOT of layers for Ubuntu and all the versions), you can run `docker images` to get information about the images on your system. Here's an example:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             VIRTUAL SIZE
ubuntu              14.04               c4ff7513909d        2 days
ago          225.4 MB
ubuntu              latest              c4ff7513909d        2 days
ago          225.4 MB
ubuntu              14.04.1             c4ff7513909d        2 days
ago          225.4 MB
```

```
ubuntu              trusty              c4ff7513909d        2 days
ago         225.4 MB
ubuntu              14.10               75204fdb260b        2 days
ago         230.1 MB
ubuntu              utopic              75204fdb260b        2 days
ago         230.1 MB
ubuntu              12.04.5             822a01ae9a15        2 days
ago         108.1 MB
ubuntu              precise             822a01ae9a15        2 days
ago         108.1 MB
ubuntu              12.04               822a01ae9a15        2 days
ago         108.1 MB
```

As you can see, the command returns the following columns:

- REPOSITORY. The name of the repository, which in this case is “ubuntu”.
- TAG. We’ll talk more about tags in a bit, but tags are similar to those found in git or other version control systems, and represent a specific set point in the repositories’ commit history. As you can see from the list, we’ve pulled down a bunch of different versions of ubuntu: 14.04, 14.10, 12.04, etc. Each of these versions is tagged with a version number, a name, and there’s even a special tag called “latest” which represents the latest version.
- IMAGE ID. This is like the primary key for the image. Sometimes, such as when you commit a container without specifying a name or tag, the repository or the tag is *<NONE>*, but you can always refer to a specific image or container using its ID.
- CREATED. The date the repository was created, as opposed to when it was pulled. This can help you assess how “fresh” a particular build is. Docker appears to update their master images on a fairly frequent basis.
- VIRTUAL SIZE. The size of the image.

If you want a granular view of the layers in in an image, you can use `docker history`:

```
$ docker history ubuntu:latest
IMAGE               CREATED             CREATED
BY                                      SIZE
c4ff7513909d        7 days ago          /bin/sh -c #(nop) CMD
[/bin/bash]               0 B
cc58e55aa5a5        7 days ago          /bin/sh -c apt-get update
&& apt-get dist-upg   32.67 MB
0ea0d582fd90        7 days ago          /bin/sh -c sed -i 's/^#\s*\
(deb.*universe\)$/   1.895 kB
d92c3c92fa73        7 days ago          /bin/sh -c rm -
rf /var/lib/apt/lists/*          0 B
9942dd43ff21        7 days ago          /bin/sh -c echo '#!/bin/
sh' > /usr/sbin/polic   194.5 kB
```

```
1c9383292a8f        7 days ago          /bin/sh -c #(nop) ADD
file:c1472c26527df28498   192.5 MB
511136ea3c5a        14 months
ago                                                       0 B
```

Each line in the history corresponds to a commit of the image's filesystem. The values in the SIZE column add up to the corresponding VIRTUAL SIZE column for the image in `docker image`. (If you decide to double check this, remember to that the column has units, so be sure to convert all values to MB.)

There are a couple of key things to understand about the layers in a docker images:

- They can be reused. Docker keeps track of all the layers you've pulled. So, if two images happen to have a layer in common (for example, if two images are built form the same base box), Docker will reuse the common parts, and only pull the diffs.
- The layers are always additive, which can lead to really big sizes if you're not careful. For example, if you download a large file, make a commit, delete the file, and then make another commit, that large file will still be present in the layer history. We'll come back to this idea again, so don't worry if it doesn't make too much sense right now. Just remember that layers are always additive.

Containers: Running instances of an Image

4

A container is a running image that you start with the `docker run` command, like this:

```
$ docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

The command consists of:

- OPTIONS. There are a LOT of options for the run command. We'll cover this in a bit more depth in the next shortly.
- IMAGE. The name of the image that the container should start from.
- COMMAND. The command that should run *on the container when it starts*.

Let's try a simple command to to start a new container using the latest version of Ubuntu. Once this container starts, you'll be at a bash shell where you can do `cat /etc/os-release` to see the release information about the OS:

```
$ docker run -it ubuntu:latest /bin/bash
root@4afa46473802:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="14.04.1 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04.1 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
```

Next, type this command in your running container to make a simple change on the filesystem:

```
root@4afa46473802:/# echo "hello world" > /tmp/hello.txt
```

Now that you've verified the OS release and created a test file, here's a bit more detail about the options we've used in the RUN command:

- The OPTIONS "-it" tell Docker to make the container interactive and provide a tty (i.e., attach a terminal)
- The IMAGE is the "ubuntu:latest" image we pulled down. Note the *repository:tag* format, which is used throughout docker. By default, the "latest" tag is used, but if we were to use "ubuntu:precise" we'd be using Ubuntu 12.04.
- The COMMAND is "/bin/bash", which starts a shell where you can log in

The cool thing is that once you're in an interactive docker shell, you can do anything you want, such as installing packages (via apt-get), code bases (using git), or dependencies (using pip, gem, npm, or any other package manager du jour.) You can then commit the container at any point to create a new image that has your changes.

To see this in action, open a new terminal on your host machine and use "docker ps" command to see some information about the running container:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS               NAMES
4afa46473802        ubuntu:14.04        /bin/bash           17
hours ago        Up 39 seconds                          jol-
ly_perlman
```

This command will tell you:

- The CONTAINER_ID, a unique identifier you can use to refer to the container in other commands (this is kind of like a process id in Linux)
- The IMAGE that was used when the container started
- The COMMAND used to start the container
- The time the underlying image was CREATED
- The uptime STATUS
- The exposed ports (lot's more on this!)
- A human readable NAME that you can use in place of the ID (this is something you can also assign yourself, which we'll get into shortly.)

Now try the `docker diff` command:

```
$ docker diff jolly_perlman
C /tmp
A /tmp/hello.txt
```

As you can see, this command returns the changes that have happened on the filesystem, which in our case is that we've changed the "/tmp" directory by

adding a “hello.txt” file. Now let’s commit the changes we’ve made into a new container:

```
$ docker commit -m "Adding hello world" jolly\_perlman hello\_dock-
er
5af244124dd8656...
```

Taking another look at `docker images` shows the effect; we now have a new image called “hello_docker” that includes our “/tmp/hello.txt” file. Taking a quick look at the container’s history shows how our change added a whopping new 12 byte layer on top of the original layers of ubuntu:latest.

```
$ docker history hello_docker
IMAGE               CREATED             CREATED
BY                                      SIZE
5af244124dd8        18 hours ago        /bin/
bash                                       12 B
c4ff7513909d        2 days ago          /bin/sh -c #(nop) CMD
[/bin/bash]               0 B
cc58e55aa5a5        2 days ago          /bin/sh -c apt-get update
&& apt-get dist-upg   32.67 MB
0ea0d582fd90        2 days ago          /bin/sh -c sed -i 's/^#\s*\
(deb.*universe\)$/   1.895 kB
d92c3c92fa73        2 days ago          /bin/sh -c rm -
rf /var/lib/apt/lists/*          0 B
9942dd43ff21        2 days ago          /bin/sh -c echo '#!/bin/
sh' > /usr/sbin/polic   194.5 kB
1c9383292a8f        2 days ago          /bin/sh -c #(nop) ADD
file:c1472c26527df28498   192.5 MB
511136ea3c5a        14 months
ago                                                     0 B
```

Finally, we can kill our docker process like this:

```
$ docker kill jolly_perlman
jolly_perlman
```

If you do *docker ps* again, you’ll see that the process has stopped.

Docker RUN Quick Reference

The next chapter will provide much more detail on how to create your own containers and images using *docker run*, one of the most feature-rich commands in Docker. Taken directly from *docker help*, this table is a useful quick reference you can refer to as you go further with Docker.

Option	Description
-a, --attach=[]	Attach to stdin, stdout or stderr.
-c, --cpu-shares=0	CPU shares (relative weight)
--cidfile=""	Write the container ID to the file
--cpuset=""	CPUs in which to allow execution (0-3, 0,1)
-d, --detach=false	Detached mode: Run container in the background, print new container id
--dns=[]	Set custom dns servers
--dns-search=[]	Set custom dns search domains
-e, --env=[]	Set environment variables
--entrypoint=""	Overwrite the default entrypoint of the image
--env-file=[]	Read in a line delimited file of ENV variables
--expose=[]	Expose a port from the container without publishing it to your host
-h, --hostname=""	Container host name
-i, --interactive=false	Keep stdin open even if not attached
--link=[]	Add link to another container (name:alias)
--lxc-conf=[]	(lxc exec-driver only) Add custom lxc options --lxc-conf="lxc.cgroup.cpuset.cpus = 0,1"
-m, --memory=""	Memory limit (format: , where unit = b, k, m or g)
--name=""	Assign a name to the container
--net="bridge"	Set the Network mode for the container
	'bridge': creates a new network stack for the container on the docker bridge
	'none': no networking for this container
	'container:': reuses another container network stack
	'host': use the host network stack inside the container. Note: the host mode gives the container full access to local system services such as D-bus and is therefore considered insecure.
-P, --publish-all=false	Publish all exposed ports to the host interfaces
-p, --publish=[]	Publish a container's port to the host

Option	Description
	format: ip:hostPort:containerPort or ip::containerPort or hostPort:containerPort
	(use ‘docker port’ to see the actual mapping)
--privileged=false	Give extended privileges to this container
--rm=false	Automatically remove the container when it exits (incompatible with -d)
--sig-proxy=true	Proxify received signals to the process (even in non-tty mode). SIGCHLD is not proxied.
-t, --tty=false	Allocate a pseudo-tty
-u, --user=""	Username or UID
-v, --volume=[]	Bind mount a volume (e.g., from the host: -v /host:/container, from docker: -v /container)
--volumes-from=[]	Mount volumes from the specified container(s)
-w, --workdir=""	Working directory inside the container

Building images with Dockerfiles 5

As we saw in the Docker Walkthrough chapter, the general Docker workflow is:

- start a container based on an image in a known state
- add things to the filesystem, such as packages, codebases, libraries, files, or anything else
- commit the changes as layers to make a new image

In the walkthrough, we took a very simple approach of just starting a container interactively, running the commands we wanted (like "apt-get install" and "pip install"), and then committing the container into a new image.

In this chapter, we'll look at a more robust way to build an image. Rather than just running commands and adding files with tools like *wget*, we'll put our instructions in a special file called the ***Dockerfile*** *(https://docs.docker.com/reference/builder/)*. A Dockerfile is similar in concept to the recipes and manifests found in infrastructure automation (IA) tools like **Chef** (http://www.getchef.com/) or **Puppet** (http://puppetlabs.com/).

Overall, a Dockerfile is much more stripped down than the IA tools, consisting of a single file with a DSL that has a handful of instructions. The format looks like this:

```
# Comment
INSTRUCTION arguments
```

The following table summarizes the instructions; many of these options map directly to option in the "docker run" command:

Command	Description
ADD	Copies a file from the host system onto the container
CMD	The command that runs when the container starts

Command	Description
ENTRYPOINT	
ENV	Sets an environment variable in the new container
EXPOSE	Opens a port for linked containers
FROM	The base image to use in the build. This is mandatory and must be the first command in the file.
MAINTAINER	An optional value for the maintainer of the script
ONBUILD	A command that is triggered when the image in the Dcokerfile is used as a base for another image
RUN	Executes a command and save the result as a new layer
USER	Sets the default user within the container
VOLUME	Creates a shared volume that can be shared among containers or by the host machine
WORKDIR	Set the default working directory for the container

Once you've created a Dockerfile and added all your instructions, you can use it to build an image using the `docker build` command. The format for this command is:

```
docker build [OPTIONS] PATH | URL | -
```

The build command results in a new image that you can start using `docker run`, just like any other image. Each line in the Dockerfile will correspond to a layer in the images' commit history.

Example of building an image from a Dockerfile

Perhaps the best way to understand a Dockerfile is to dive into an example. Let's take a look at the example we went through in our overview chapter and condense it into a Dockerfile:

```
#
# Super simple example of a Dockerfile
#
FROM ubuntu:latest
MAINTAINER Andrew Odewahn "odewahn@oreilly.com"

RUN apt-get update
```

```
RUN apt-get install -y python python-pip wget
RUN pip install Flask

ADD hello.py /home/hello.py

WORKDIR /home
```

As you can see, it's pretty straightforward: we start from "ubuntu:latest," install dependencies with the RUN command, add our code file with the ADD command, and then set the default directory for when the container starts. Once we have a Dockerfile itself, we can build an image using `docker build`, like this:

```
$ docker build -t "simple_flask:dockerfile" .
```

The "-t" flag adds a tag to the image so that it gets a nice repository name and tag. Also not the final ".", which tells Docker to use the Dockerfile in the current directory. Once you start the build, you'll see it churn away for a while installing things, and when it completes, you'll have a brand new image. Running `docker history` will show you the effect of each command has on the overall size of the file:

```
$ docker history simple_flask:dockerfile
IMAGE               CREATED             CREATED
BY                                      SIZE
9ada423c0a60        3 days ago          /bin/sh -c #(nop) WORKDIR /
home                 0 B
5c3625267cd9        3 days ago          /bin/sh -c #(nop) ADD file:
96e699cd177f1a3f3c   163 B
9c20a6548fbe        3 days ago          /bin/sh -c pip install
Flask                    4.959 MB
7195370ae6e1        3 days ago          /bin/sh -c apt-get install
-y python python-p   136.1 MB
761bf82875cc        3 days ago          /bin/sh -c apt-get up-
date                       19.94 MB
40b29df1d2c2        3 days ago          /bin/sh -c #(nop) MAINTAIN-
ER Andrew Odewahn "   0 B
c4ff7513909d        9 days ago          /bin/sh -c #(nop) CMD
[/bin/bash]               0 B
cc58e55aa5a5        9 days ago          /bin/sh -c apt-get update
&& apt-get dist-upg   32.67 MB
0ea0d582fd90        9 days ago          /bin/sh -c sed -i 's/^#\s*\
(deb.*universe\)$/   1.895 kB
d92c3c92fa73        9 days ago          /bin/sh -c rm -
rf /var/lib/apt/lists/*          0 B
9942dd43ff21        9 days ago          /bin/sh -c echo '#!/bin/
sh' > /usr/sbin/polic   194.5 kB
1c9383292a8f        9 days ago          /bin/sh -c #(nop) ADD
```

```
file:c1472c26527df28498   192.5 MB
511136ea3c5a        14 months
ago                                                    0 B
```

Finally, you can start the container itself with the following command:

```
$ docker run -p 5000:5000 simple_flask:dockerfile python hello.py
```

Notice that in this example we're running the Flask app directly when we start the container, rather than just running the bash shell and starting it as we've done in other examples.

Dockerfiles vs. Infrastructure Automation (IA)

Dockerfiles provide a relatively simple way to create a base image. And, because you can use the FROM command to chain Dockerfiles together into increasingly complex images, you can do quite a lot, even with Docker's (refreshingly!) minimal command set. But, if you already have an existing IA tool (and you should!), such as **Chef** (http://www.getchef.com/), **Puppet** (http://puppetlabs.com/), **Ansible** (http://www.ansible.com/home), **Salt** (http://www.saltstack.com/), it's very unlikely you could or even should rewrite everything. So, if you're in this situation what can you do?

I HAVE NO IDEA! I NEED TO RESEARCH THIS MORE, BUT I THINK YOU CAN USE A TOOL LIKE **Packer](http://www.packer.io/). MAYBE I CAN CONVINCE Jeroen Janssens to do something with his Ansible stuff** (https://github.com/jeroenjanssens/data-science-at-the-command-line/tree/master/dst/build)

Creating your own Docker Image

6

Now that we've got the basics, let's make something a tiny bit more realistic: a **Flask** (http://flask.pocoo.org/) app. In this section we'll:

- Start a basic container named "simple_flask" based on Ubuntu
- Install the dependencies
- Install the app itself (which is just the "Hello, World!" example, so it's nothing exciting)
- Commit the container to a new image for our app
- Start a new container based on our image
- Access our app using a browser

Again, the point here is not to show the best way to set up an environment, but instead to illustrate the Docker commands and what they do. I'll code development environments in more detail later.

Start the "simple_flask" container

Our first step is to start a new container based on "ubuntu:latest." As we did in the previous example, we'll make it interactive using the "-it" options, but this time we'll give it our own name:

```
$ docker run -it --name="simple_flask" ubuntu:latest /bin/bash
```

The nice thing about using a name is that you can use it in other docker commands in place of the container id. For example, you can use this command to see the top process running on the container. (Also, it's worth noting that you need to run this on a terminal on your host, not in the container itself.)

```
$ docker top simple_flask
PID                 USER                COMMAND
969                 root                /bin/bash
```

Install the dependencies

Now that we've got the container running, we install the dependencies we need. First, since Flask is a Python micro framework, let's check out what version of Python we have:

```
root@2f5ada6523c4:/home# python --version
bash: python: command not found
```

"Wait a minute!" you might be asking yourself, "What gives? Shouldn't Python be installed on Ubuntu?" And, the answer is "Yes," on a "real" Ubuntu distribution it is. But, to save on size, the Docker base images are stripped down versions of the official images. This is important to keep this in mind as you build new containers: don't take it for granted that everything on the official distribution will be present on the Docker base image.

So, let's get a minimal python environment going. First, we'll update our apt repository:

```
root@2f5ada6523c4:/home# apt-get update
```

This will churn away for a while updating various packages. Once it's complete, install python and pip:

```
root@2f5ada6523c4:/home# apt-get install -y python python-pip wget
```

This churns even longer, but once it's done, you can install Flask:

```
root@2f5ada6523c4:/home# pip install Flask
```

Whew. That's a lot of stuff to install, so let's commit the image with a simple commit message:

```
$ docker commit -m "installed python and flask" simple_flask
c21c062b...
```

So, now if you check your "docker images" you'll see the new repository:

```
admins-air-5:~ odewahn$ docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             VIRTUAL SIZE
```

```
<none>              <none>              c21c062b086a        2 days
ago          385.7 MB
ubuntu              latest              c4ff7513909d        7 days
ago          225.4 MB
```

But, ugh, it doesn't have a name -- it just shows up as "<none>". We can use *docker tag* to give our image a nice name, like this:

```
$ docker tag c21c062b086a simple_flask
$ docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             VIRTUAL SIZE
simple_flask        latest              c21c062b086a        2 days
ago          385.7 MB
ubuntu              latest              c4ff7513909d        7 days
ago          225.4 MB
```

Finally, taking a quick look at the history shows that we've loaded about 160MB worth of new stuff onto our image:

```
$ docker history simple_flask
IMAGE               CREATED             CREATED
BY                                      SIZE
c21c062b086a        2 days ago          /bin/
bash                                       160.3 MB
c4ff7513909d        7 days ago          /bin/sh -c #(nop) CMD
[/bin/bash]                 0 B
cc58e55aa5a5        7 days ago          /bin/sh -c apt-get update
&& apt-get dist-upg   32.67 MB
0ea0d582fd90        7 days ago          /bin/sh -c sed -i 's/^#\s*\
(deb.*universe\)$/   1.895 kB
d92c3c92fa73        7 days ago          /bin/sh -c rm -
rf /var/lib/apt/lists/*          0 B
9942dd43ff21        7 days ago          /bin/sh -c echo '#!/bin/
sh' > /usr/sbin/polic   194.5 kB
1c9383292a8f        7 days ago          /bin/sh -c #(nop) ADD
file:c1472c26527df28498   192.5 MB
511136ea3c5a        14 months
ago                                                       0 B
```

Install our code

Let's get our code installed. First, fire up a new container:

```
$ docker run -it -p 5000:5000 simple_flask /bin/bash
```

As before, we'll use the "-it" options to make it interactive, but we're now adding a couple of new settings:

- The "-p 5000:5000" option exposes port 5000 on the container and routes it to port 5000 on the host. If you're running Docker on a Mac, the confusing thing to remember is that from the Docker container's perspective, the virtual machine is the host, so you'll also need to expose port 5000 from the VM in order to see it on your "real" host OS. We'll cover this **Inception** (http://en.wikipedia.org/wiki/Inception)-like step in the next section.
- The "-w /home" option sets the working directory used when the container starts. Note that this must be an absolute path.

Once we're in, we'll grab our simple hello world file from this projects github repo:

```
# wget https://raw.githubusercontent.com/odewahn/docker-jumpstart/
master/examples/hello.py
--2014-08-17 05:38:02--  https://raw.githubusercontent.com/odewahn/
docker-jumpstart/master/examples/hello.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
199.27.78.133
Connecting to raw.githubusercontent.com (raw.githubusercon-
tent.com)|199.27.78.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 149 [text/plain]
Saving to: 'hello.py'

100%
[======================================================================>]
149         --.-K/s   in 0s

2014-08-17 05:38:02 (1.34 MB/s) - 'hello.py' saved [149/149]

root@2f5ada6523c4:/home# cat hello.py
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

Finally, once you've got the file, start the app. You should see something like this:

```
root@2f5ada6523c4:/home# python hello.py
 * Running on http://127.0.0.1:5000/
```

View the flask app on your host machine

We're ready for the big reveal. On a terminal on your host, try this command:

```
$ curl localhost:5000
```

If you're running this on Linux as your host OS, you should see "Hello World!" printed. If you're running it on boot2docker on a Mac or Windows, you most likely got this error:

```
curl: (7) Failed connect to localhost:5000; Connection refused
```

As I mentioned earlier, when you started the container and told it to expose port 5000, it exposed it *on the virtual machine*, not on your host. As described in the chapter on boot2docker, you need to use VBoxManage to open the port from the VM onto your host, like this:

```
$ VBoxManage controlvm boot2docker-vm natpf1 "flask-server,tcp,
127.0.0.1,5000,,5000"
```

You should now be able to connect to the simple app, like this:

```
$ curl localhost:5000
Hello World!
```

Cleaning up

If you're really excited about your "Hello World!" Flask app, you should feel free to commit it. Otherwise, let's kill it. First, we need to figure out its ID, which we'll do using "docker ps":

```
$ docker ps -a
CONTAINER ID        IMAGE                 COMMAND
CREATED             STATUS
PORTS                    NAMES
2f5ada6523c4        simple_flask:latest   /bin/bash           3
days ago          Up 34 minutes               0.0.0.0:5000->5000/
tcp   nostalgic_goodall                           simple_flask
```

We can kill our simple_flask app running as container "2f5ada6523c4" like this:

```
$ docker kill 2f5ada6523c4
2f5ada6523c4
```

To verify it's gone, lets run "docker ps" again, but this time add the "-a" option so that we see all containers:

```
$ docker ps -a
CONTAINER ID        IMAGE                COMMAND
CREATED             STATUS
PORTS               NAMES
2f5ada6523c4        simple_flask:latest   /bin/bash           3
days ago          Exited (-1) 2 seconds ago
nostalgic_goodall
55ba5b67cb28        simple_flask:latest   /bin/bash           3
days ago          Exited (0) 40 minutes ago
dreamy_mayer
6029929a1e53        ubuntu:latest         /bin/bash           3
days ago          Exited (0) 13 hours ago
naughty_leakey
ef9bd2df07c6        ubuntu:latest         /bin/bash           4
days ago          Exited (0) 13 hours ago
simple_flask
```

As you'll see, although the container has stopped running (i.e., its status has changed from Up to Exited), the container itself is still there. In fact, unless you use the "-rm" option when you start a container, it will always leave this remnant image behind. And, if left unchecked, after a while you'll consume your entire disk with stopped containers.

Why is it like this, you might ask? The answer lies in Docker's need to get a clean files state for a commit. In a container is running, it can mean that there are open files or processes that could interfere with the ability to save the state of the filesystem. So, rather than destroy the container automatically, Docker saves it to enable you to get a nice, clean commit image. So, killing the image is really the same as just pausing it.

So, to get rid of this ghost container, you need to use "docker rm":

```
$ docker rm 2f5ada6523c4
2f5ada6523c4
```

But, what about all those other stopped containers hanging around? Never fear! This great tip from **Jim Hoskins** (http://jimhoskins.com/) in **Remove Untagged Images From Docker** (http://jimhoskins.com/2013/07/27/remove-untagged-docker-images.html) shows a quick way to remove stopped containers in bulk:

```
$ docker rm $(docker ps -aq)
```

Sharing images on Docker Hub 7

The **Docker Hub** (https://hub.docker.com) is a site where you store and share images you create. Consciously modeled on **GitHub** (https://github.com/), the site offers features like:

- Docker image hosting. Like GitHub, public images are free and private images are paid
- Collaborators who can push and pull the images
- Webhooks that are fired when a new image is pushed
- Statistics tracking, such as downloads and stars

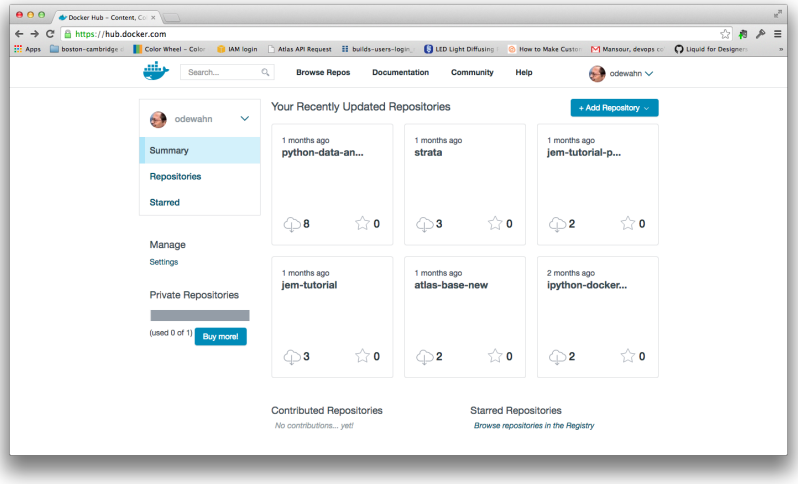
There are two key ways to interact with the Docker Hub:

- The Web interface, where you can register, manage image metadata (description, etc), add or remove collaborators, and so forth. This is similar to GitHub.
- The docker command line tool, where you can pull, push, or search for images. This is similar to git.

There are a number of features specific Docker. One of the most interesting, **trusted builds** (http://blog.docker.com/2013/11/introducing-trusted-builds/), allow you to create signed images based on a Dockerfile in GitHub or BitBucket. As with a CI/CD tool, the trusted build is triggered by a post commit hook on your repo so that your image will also be up to date with your codebase. In addition, the image is flagged with a "trusted build" badge so that you can verity its source.

Create an account

The first to using Docker hub is to sign up, which you can do on their **Signup page** (https://hub.docker.com/account/signup/). You'll be asked for an email address, username, and a password. Once you login, you'll see a project dashboard that's similar to other hosting sites:



Once you've set up your account, you use the `docker login` command from your terminal so that you can pull and push images from your account. (Note that you can also register with this command, as well, if you want to not use the site.) Once you login, your docker will store your credentials in a file called `.dockercfg` in your home directory.

Searching for images

One of the most basic things you can do with the Hub is search for other images using `docker search`. Here, for example, are the first few results returned by the `docker search ubuntu` command. As of this writing, we get over 1300 matches, so be prepared to wait a bit for the results.)

```
$ docker search ubuntu
NAME                                                    DESCRIP-
TION                                             STARS     OFFICIAL
AUTOMATED
ubuntu                                                  Official Ubuntu
base image                                       525       [OK]
stackbrew/ubuntu                                        Official Ubuntu
base image                                       40        [OK]
dockerfile/ubuntu                                       Trusted Ubuntu
(http://www.ubuntu.com/) Bu...                    19            [OK]
crashsystems/gitlab-docker                              A trusted, regu-
larly updated build of GitL...                    19            [OK]
ubuntu-upstart                                          Upstart is an
```

```
event-based replacement for ...   10        [OK]
dockerfile/ubuntu-desktop                   Trusted Ubuntu
Desktop (LXDE) (http://lxde...   9               [OK]
lukasz/docker-scala                         Dockerfile for
installing Scala 2.10.3, Ja...   7               [OK]
litaio/ruby                                 Ubuntu 14.04 with
Ruby 2.1.2 compiled from...   7                   [OK]
mbentley/ubuntu-django-uwsgi-
nginx
6                    [OK]
cmfatih/phantomjs                           PhantomJS [ phan-
tomjs 1.9.7, casperjs 1.1....   5               [OK]
zmarcantel/cassandra                        Cluster/Stand-
alone DataStax Cassandra on U...   4              [OK]
suttang/gollum                              The gollum wiki
installed on ubuntu              4              [OK]
...
```

The command returns the following columns:

- Name. This is the name and namespace of the image. (More on namespaces in a bit.)
- Description. This is metadata about the image from Docker Hub.
- Stars. The number of people who have “starred” the image
- Official. Whether or not the image comes from Docker, Inc. An “OK” in this column helps provide some sense of security that the image has been vetted.
- Automated. Whether or not the image is the result of an automated build. While less authoritative than an “Offical” image, an automated image will provide a link to the source GitHub or BitBucket repo on which the image is based. This way you can at least review the contents of the image before you start it on your environment.

As you undoubtedly noticed, there are a couple of formats in the “Name” column. Some images have just a single name, like “ubuntu,” and some have two names separated by a slash, like “crashsystems/gitlab-docker.” This naming convention is a reflection of Docker’s namespace. There are two namespace main levels:

- Root. This is the namespace is reserved for “root” images maintained directly by Docker, Inc. Basically, it’s an “empty” space, and the names are things like “ubuntu” or “debian.”
- User. This namespace is for users within the Docker Hub, such as “crashsystems/gitlab-docker,” “mbentley/ubuntu-django-uwsgi-nginx,” or “suttang/gollum”

Pulling an image

Pulling any of these images is simple: just use `docker pull` and whatever the name is. For example, here's how you pull all layers of the "stackbrew/ubuntu" image:

```
$ $ docker pull stackbrew/ubuntu
Pulling repository stackbrew/ubuntu
822a01ae9a15: Pulling dependent layers
c4ff7513909d: Download complete
195eb90b5349: Pulling dependent layers
bac448df371d: Download complete
dfaad36d8984: Download complete
5796a7edb16b: Download complete
af82eb377801: Downloading
[===========>                                       ] 16.38 MB/
68.7 MB 164h59m35s
3af9d794ad07: Downloading
[========================>                          ] 19.07 MB/
39.53 MB 55h25m0s
93c381d2c255: Downloading
[================>                                  ] 12.58 MB/
39.17 MB 109h9m35s
1c9383292a8f: Download complete
...
```

As discussed in the chapter on images, you can also pull only the layers of the latest build by adding ":latest" to the name. (Otherwise you'll pull the full history.)

Pushing an Image

Let's try to push an image to the hub. Unlike the previous `search` and `pull` commands, you must be logged in to push. If you're not already logged in, Docker will prompt you for your credentials

```
$ docker push simple_flask:latest
2014/09/03 12:42:10 You cannot push a "root" repository. Please re-
name your repository in <user>/<repo> (ex: odewahn/
simple_flask:latest)
```

This rather self-explanatory error occurs because we didn't specify a username when we created the image. To push to the Docker Hub, though, you must specify a username. (Unless you're luck enough to be making Docker's

official images, in which case you probably don't need this book!). So, let's add our username using the `docker tag` command:

```
$ docker tag simple_flask odewahn/simple_flask
```

Once you've retagged the image, you can push it to the Hub:

```
$ docker push odewahn/simple_flask:latest
The push refers to a repository [odewahn/simple_flask] (len: 1)
Sending image list

Please login prior to push:
Username: odewahn
Password:
Email: odewahn@oreilly.com
Login Succeeded
The push refers to a repository [odewahn/simple_flask] (len: 1)
Sending image list
Pushing repository odewahn/simple_flask (1 tags)
511136ea3c5a: Image already pushed, skipping
1c9383292a8f: Image already pushed, skipping
9942dd43ff21: Image already pushed, skipping
d92c3c92fa73: Image already pushed, skipping
0ea0d582fd90: Image already pushed, skipping
cc58e55aa5a5: Image already pushed, skipping
c4ff7513909d: Image already pushed, skipping
c21c062b086a: Image successfully pushed
Pushing tag for rev [c21c062b086a] on {https://cdn-
registry-1.docker.io/v1/repositories/odewahn/simple_flask/tags/
latest}
```

8 Additional Resources

This Jumpstart guide has only scratched the surface of Docker. This page is meant as a kind of living resource, so if you find helpful tools or resources, send me a pull request.

Sites, Blogs, and Books

- **The Docker Documentation Page** (http://docs.docker.com/userguide/). Docker's documentation is great, if somewhat sprawling.
- **The Docker Book** (http://www.dockerbook.com/). An awesome resource from **James Turnbull** (http://www.jamesturnbull.net/), a prolific author and VP of Services at Docker, Inc.
- **CenturyLink Labs** (http://www.centurylinklabs.com/). An amazing collection of how to articles and interviews with leading people in the Docker field.

Other tools in the Docker ecosystem

Docker has a booming ecosystem of tools. Here are a few of the more interesting ones that I'd love to write a chapter about some day:

- **fig** (http://www.fig.sh/). A tool for orchestrating and managing multi-container apps.
- **Kitematic** (https://kitematic.com/). A slick, Mac-only alternative to boot2docker that provides a clean and simple way to install and run Docker containers.
- **Panamax** (http://panamax.io/). A tool for drag-and-drop deployment for containers.