Docker For Developers

Second Edition

Chris Tankersley

Docker for Developers

Chris Tankersley

This book is for sale at http://leanpub.com/dockerfordevs

This version was published on 2017-08-07

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Chris Tankersley by spreading the word about this book on Twitter!

The suggested hashtag for this book is #dockerForDevs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#dockerForDevs

To my wonderful wife Giana, who puts up with all the stuff I do, and my two boys who put up with all of my travel.

Thank you to the people that read this book early and provided valuable feedback, including Ed Finkler, Beau Simensen, Gary Hockin, and M. Craig Amiano.

I'd also like to thank the entire PHP community, for without them I wouldn't be where I am today as a developer.

The book cover contains elements found on Freepik

Contents

Getting Started

Before we begin using Docker, we are going to need to install it. There will be a few caveats that we are going to discuss as we go through the installation because, unless you are on Linux, we're going to need some extra software to utilize Docker. This will create some extra issues down the road, but rest assured I'll keep you abreast of the more disasterous pitfalls that you may encounter, or various issues that might arise on non-Linux systems.

The installation is normally fairly easy no matter what OS you are going to use, so let's get cracking. We're going to install Docker Community Edition 17.03. I'll go over some basic installation, but you can always refer to https://docs.docker.com/installation/[1] for anything special or other Operating Systems if you aren't using Windows, OSX, or Ubuntu.

Throughout this book, I'm going to be using Ubuntu for all of the examples because it not only gives us a full operating system to work with as a host, but it is also very easy to set up. There are smaller Linux distributions that are designed for running Docker, but we are more worried about development at this stage. Since we're using containers it doesn't really matter what the host OS is.

In 2016 Docker released Docker for Mac and Docker for Windows, which brings a much more native feel to working with containers on those operating systems. Everything that is discussed in this book should work fine on Mac or Windows. The few caveats that still exist are detailed in the section for those operating systems.

Docker Community Edition vs Docker Enterprise Edition

In March of 2017, Docker split the project into two versions - Docker Community Edition (CE) and Docker Enterprise Edition (EE). Functionally they are currently the same, with EE having a certification process for its releases and a different support plan than CE. You will not lose out on anything by using the CE edition of Docker, and as such we will be using it for this book. If you see references to Docker 1.13 in this book and online, that is essentially the same thing as Docker CE 17.03, as Docker changed the version numbers when releasing CE and EE.

[1] https://docs.docker.com/installation/

Installing Docker

Ubuntu

Since Ubuntu ships with Long Term Release releases, I would recommend installing Ubuntu 16.04 and using that. The following instructions should work just fine for 14.04 or higher as well. Ubuntu does have Docker in it's repositories but it is generally out of date pretty quickly so we're going to use an apt repository from Docker that will keep us up-to-date. Head on over to http://www.ubuntu.com/download/server[2] and download the 16.04 LTS Server ISO and install it like normal. If you'd like a GUI, grab the desktop version. Either one will work. I'll be using the Desktop version with the intent to deploy to Ubuntu 16.04 Server.

If you've never installed Ubuntu before, Ubuntu provides a quick tutorial on what to do. Server instructions can be found at http://www.ubuntu.com/download/server/install-ubuntu-server[3] and Desktop instructions can be found at http://www.ubuntu.com/download/desktop/install-ubuntu-desktop[4].

There's a few commands we can run to set up the Docker repository. Open up a terminal and install Docker:

```
1 $ sudo -i
2 $ wget -qO- https://get.docker.com/ | sh
3 $ usermod -a -G docker [username]
4 $ exit
5 $ sg docker
```

Line 1 switches us to the `root` user to make things easier. Lines 2 runs a script that adds the repository to Ubuntu, updates apt, and install Docker for us. Line 3 sets up your user to use Docker so that we do not have to be `root` all of the time, so replace `[username]` with your actual user you will use.

We can make sure that the Docker engine is working by running `docker -v` to see what version we are at:

```
1 $ docker -v
2 Docker version 1.13.0, build 49bf474
```

To make sure that the container system is working, we can run a small contaier.

[2] http://www.ubuntu.com/download/server
[3] http://www.ubuntu.com/download/server/install-ubuntu-server
[4] http://www.ubuntu.com/download/desktop/install-ubuntu-desktop

```
$ docker run --rm hello-world
```

Ubuntu is all set up!

Windows 10 with Hyper-V

In 2016 Docker formally released a beta of Docker that runs on Windows, if your version of Windows includes Hyper-V. This includes Windows 10 Professional, Enterprise, and Education. If you have Home, you will need to upgrade to Windows 10 Professional to be able to use Hyper-V.

Head on over to the Docker Products[5] and download the package for Windows. The Docker for Windows package includes Docker Engine, Compose, Machine, and Swarm all ready to be installed, and the installer will also enable Hyper-V.

Launch the Installer. Accept the install agreement and let Docker install. Once the installer is done, make sure the 'Launch Docker' option is selected and finish the installation. That's it! See Figure 2-1.

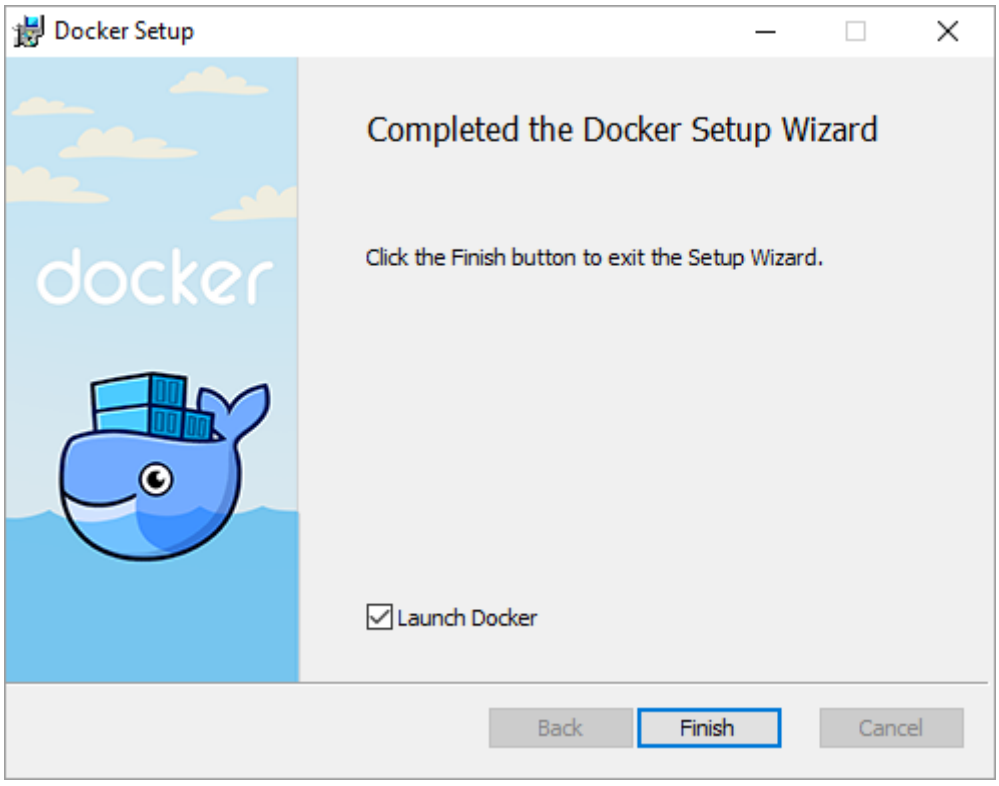

Figure 2-1

Docker will start itself up. If Hyper-V is not installed, it will prompt you to install Hyper-V and then restart the PC (Figure 2-2). This may take a few additional moments, and your PC may restart a few times.

[5] https://www.docker.com/products/docker#windows

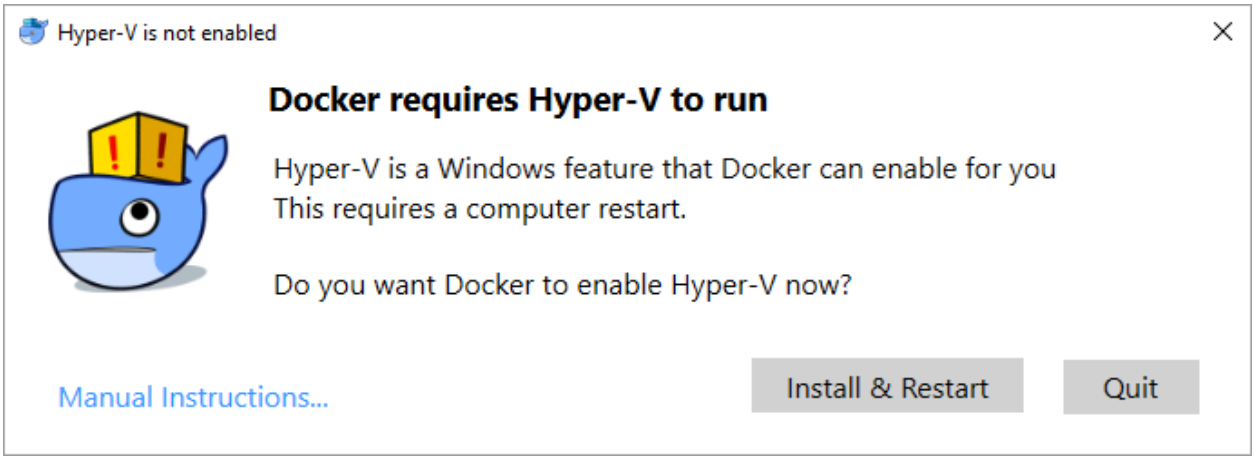

Figure 2-2

Once that is all finished you will have a 'Docker for Windows' icon on your desktop, and a Docker icon in your notification tray. As long as it is not red, you should be able to power up a Powershell window and run `docker -v` to make sure everything is working.

```
1 $ docker -v
2 Docker version 17.03.0-ce, build 60ccb22
```

Out of Memory Issues

On my test laptop, which has 4GB of RAM, I had to lower the VM memory usage all the way down to 1024 MB before it would run. I am not 100% sure why exactly, as I had more than 2 GB of available RAM at startup. If you get this error, right-click on the Docker icon in your notification tray, select 'Settings', and then 'Advanced.' Lower the Memory slider until you are able to start Docker.

`pwd` In Code Samples

Various shells in OSX and Linux allow you to specify a shortcut for the current directory by using 'pwd'. You will see this throughout the book. If you are on Windows, just replace 'pwd' with the full path of the folder that you want.

Windows 7/8/8.1

Docker does not support Docker Toolbox as well as Docker for Windows, and as such I cannot guarantee that everything in this book will work properly. Your mileage may vary.

Docker and Microsoft have released a version of Docker that runs under Hyper-V, but only for users of Windows 10 Professional or higher. For users of older versions, or lower versions of Windows 10, you will still need to download the Docker Toolbox[6], which will set everything up for us. The Toolbox includes the Doker client, Docker Machine, Docker Compose, Kitematic, and VirtualBox. It does not come with Docker Swarm.

Start up the installer. When you get to Figure 2-3 you can install VirtualBox and git if needed. I've already got them installed so I'll be skipping them but feel free to select those if needed. You should be good with the rest of the default options that the installer provides.

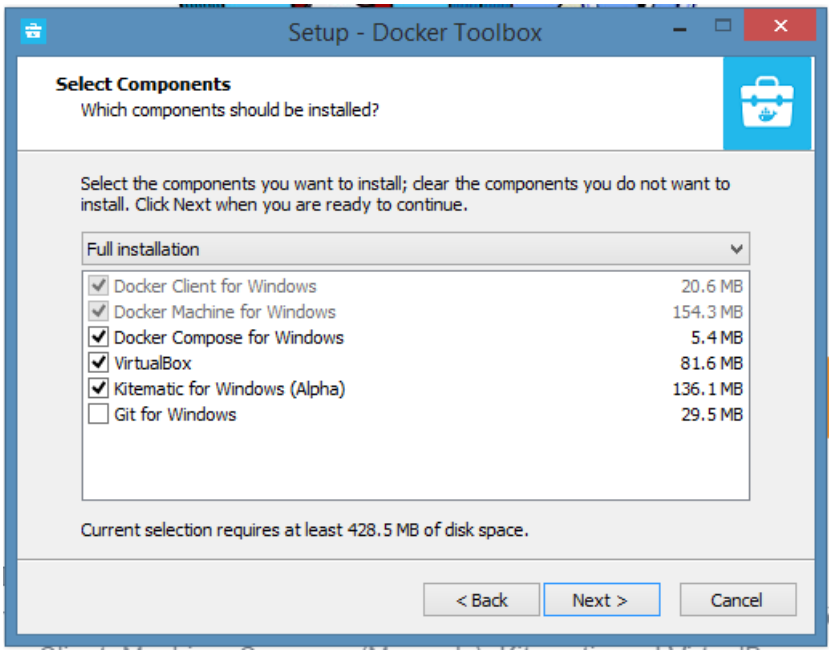

Figure 2-3

Since this changes your `PATH`, you will probably want to reboot Windows once it is all finished.

Once everything is all finished, there will be two new icons on your desktop or in your Start menu. Open up "Docker Quickstart Terminal." At this time Powershell and CMD support are somewhat lacking, so this terminal will be the best way to work with Docker. Once the terminal is up, run `docker -v` to see if you get a version number back.

[6] https://www.docker.com/products/docker#windows

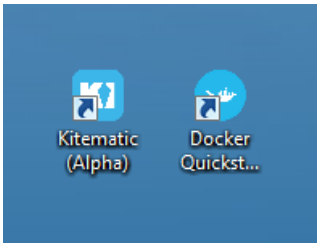

Figure 2-4

```
$ docker -v
Docker version 1.9.0, build 76d6bc9
```

Since this is the first time you've opened up the terminal, you should have also seen it create a VM for your automatically. If you open VirtualBox you'll see a new 'default' VM, which is what Docker will use. Let's start a Docker container to make sure all that underlying software is working.

```
$ docker-machine run --rm hello-world
```

You should get a nice little output from Docker. If so, move on, you are all set!

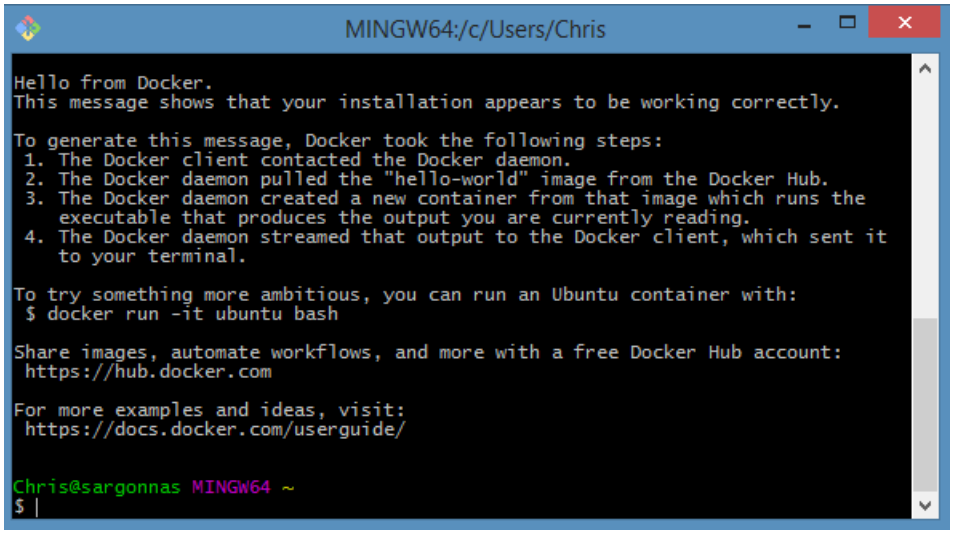

Figure 2-5

Potential Problems

If the creation of the default VM seems to hang, you can cancel it with a CTRL+C. Open up a Powershell session and run the following commands:

```
$ docker-machine rm default
$ docker-machine create --driver=virtualbox default
```

This will create the virtual machine for you, and the Quickstart Terminal will work with it after that. For some reason either Docker or VirtualBox seems to hang sometimes creating the default VM inside the terminal, but creating it through Powershell will work. You may need to accept some UAC prompts from VirtualBox to create network connections during the creation process as well.

If you are trying to run any commands throughout this book and they are failing or not working correctly, it may be due to a big in the terminal. Try running the command by adding `winpty` to the beginning, like this:

```
$ winpty docker run --rm hello-world
```

`winpty` should be pre-installed in the Quickstart tutorial, and does a better job of passing commands correctly to the various Docker programs.

You may also want to use the Docker CLI bundled with Kitematic. Simply open Kitematic, and click on 'Docker CLI' in the lower left-hand corner. You'll get a custom Powershell that is configured to work with Docker. Most commands should work fine through here, and throughout the book I've noted any specific changes needed for Windows.

OSX 10.10

OSX Yosemite introduced a new virtualization layer, much like Windows Hyper-V, that allows much better and tigher control of virtual machines. Since Docker needs features that are part of the Linux kernel, Docker has introduced a new client and setup process that uses the native OSX 10.10 virtualization layer. This provides much better performance and ease-of-use compared to Docker Toolbox.

Head on over to the Docker Products[7] and download the package for OSX. Open up the Docker.img file and drag the Docker icon into your Application folder. Once it has copied, simply run the Docker application, and a Docker icon will appear in your notification area. See Figure 2-6.

[7] https://www.docker.com/products/docker

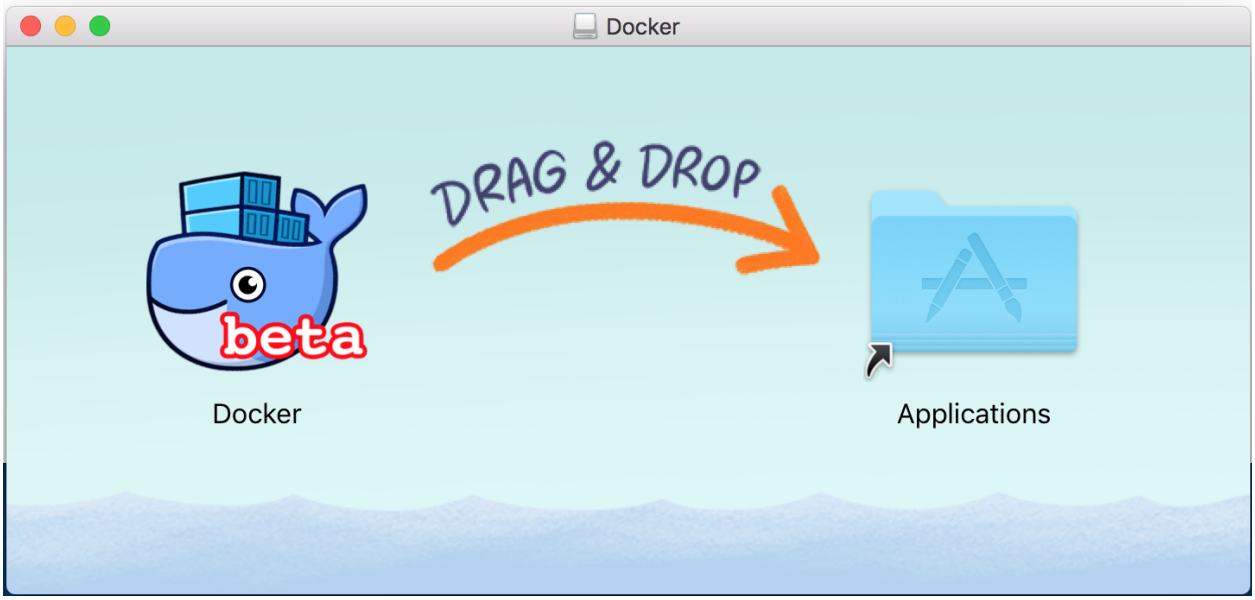

Figure 2-6

Docker may ask for additional privileges, so allow it and type in your password. If it asks to update, feel free to let it go ahead and start the update.

Once that is all done, open up a Terminal session and run `docker -v` to make sure it all works:

```
$ docker -v
Docker version 1.11.2, build b9f10c9
```

OSX 10.9 and Earlier

Docker does not support Docker Toolbox as well as Docker for Mac, and as such I cannot guarantee that everything in this book will work properly. Your mileage may vary.

Since Docker is a native Linux application, we will need to run Docker through some sort of virtualization technology and inside of a Linux VM. OSX prior to 10.10, like many versions of Windows, does not have a native container or even virtualization system, so we are relegated to using a tool like Virtualbox. Like Windows, Docker has the Docker Toolbox[8] available to download and set everything up for us. Go to the web, download the .PKG file, and open it up. There isn't much to change here other than if you want to send statistics to Docker, and which hard disk you want to install to, so the installation is painless.

Once the install is finished, click on the 'Docker Quickstart Terminal' icon in the installer to open that up. After it generates a new VM for us to use, you can make sure Docker is working by running `docker -v`:

[8] https://www.docker.com/docker-toolbox

Figure 2-7

```
$ docker -v
Docker version 1.9.0, build 76d6bc9
```

Finally, make sure that the container system is working by running the 'hello-world' container:

```
$ docker run --rm hello-world
```

You should end up with output like in Figure 2-8.

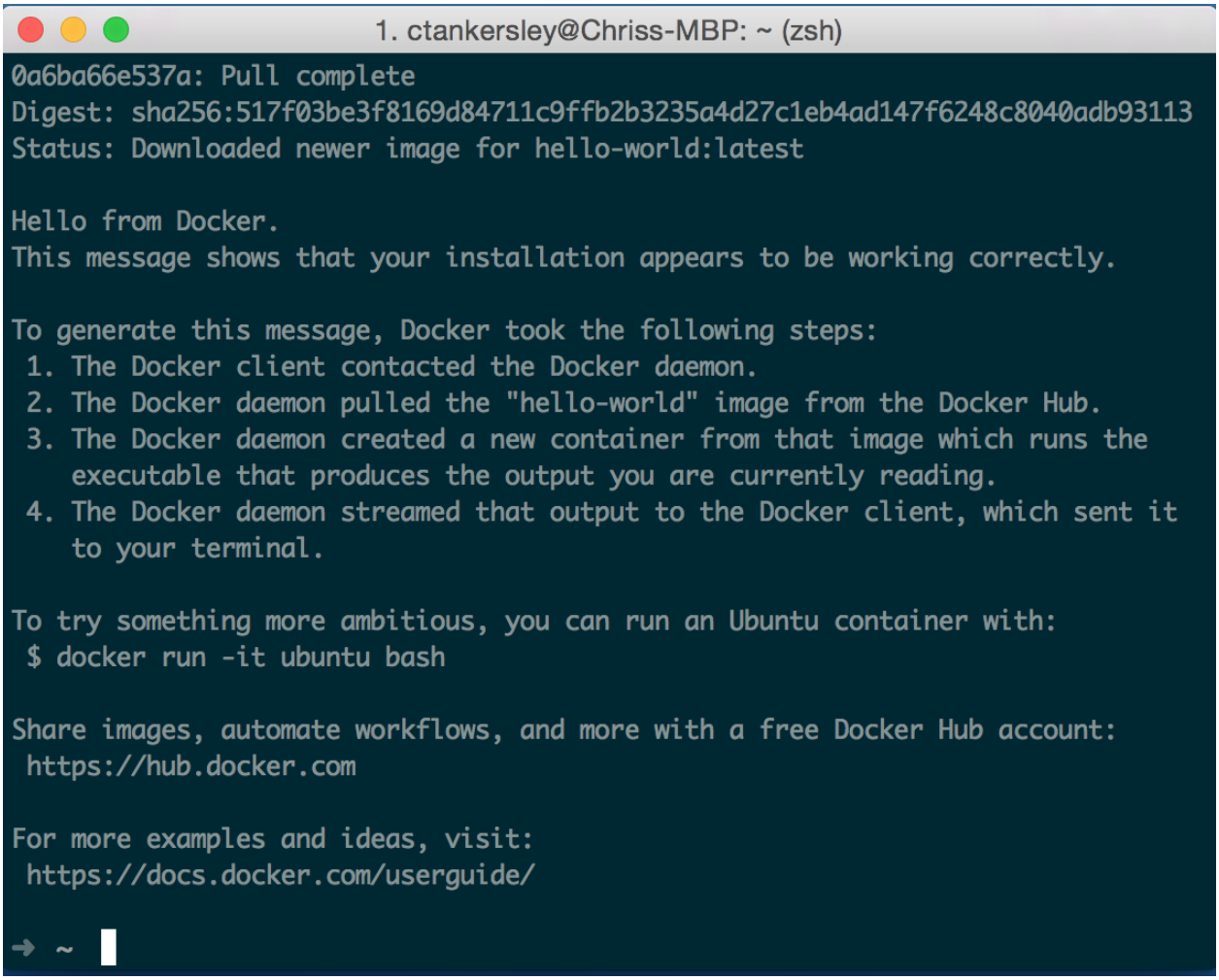

Figure 2-8

Potential Issues

One thing I have noticed, and has also been brought to my attention through readers, is that the \ character used throughout the code examples may not work in the Docker Quickstart Terminal for some reason. You may need to type the full command out instead of copy/pasting the examples.

You may also find that you have trouble mounting directories when using the Quickstart Terminal. You may be better off using a regular terminal session and accessing the Docker environment like so:

```
1 $ eval $(docker-machine env default)
```

If the above does not work to fix mounting directories, you may need to destroy and re-create the environment. You can rebuild the virtual machine by running:

```
$ docker-machine rm default
$ docker-machine create --driver virtualbox default
$ eval $(docker-machine env default)
```

Running Our First Container

Now that Docker is installed we can run our first container and start to examine what is going on. We'll run a basic shell inside of a container and see what we have. Run the following:

```
$ docker run -ti --rm ubuntu bash
```

You'll see some output similar to Figure 2-9.

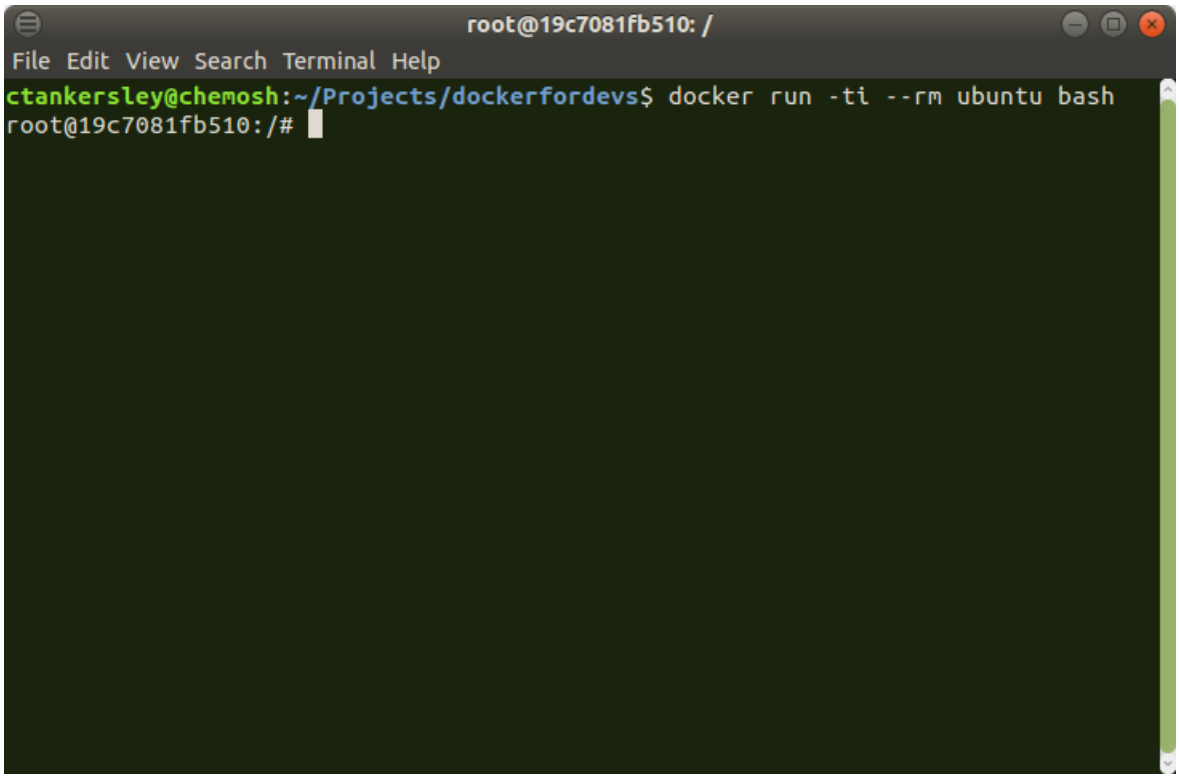

Figure 2-9

We told Docker to do was run a container based on Ubuntu, and inside of that run the command `/bin/bash`. Since we want a shell, run an interactive terminal (`-ti`), and when we are done with this container just delete it completely (`--rm`). Docker is smart enough to know that we do not have the Ubuntu image on our machine so it downloads it from the Docker Repository, which is an online collection of pre-built images, and sets it up on our system. Docker then runs the `/bin/bash` command inside that image and starts the container. We're left with the command line prompt that you can see at the bottom of Figure 2-3.

This container is a basic installation of Ubuntu so you get access to all of the GNU tools that Ubuntu ships with as well as all the normal commands you are used to. We can install new software in here using `apt`, download files off the internet, or do whatever we want. When we exit the shell provided, Docker will just delete everything, just like if you delete a virtual machine. It is as if the container never existed.

You'll notice that we are running as the `root` user. This is a root user that is specific to this container and does not impart any specific privileges to the container, or your user, at all. It is root only in the context of this container. If you want to add new users you can using the `adduser` command, just like regular Ubuntu, because everything that is part of a normal Ubuntu installation is here.

Now run `ls /home`. Assuming you haven't created any users this folder should be empty. Contrast this to your host machine that will have, at the very least, a directory for your normal user. This container isn't part of your host machine, it is it's own little world.

From a file and command perspective, we are working just like we did in the chroot. We are sectioned off into our own corner on the host machine and cannot see anything on the host machine.

Remember though, we aren't actually running Ubuntu inside the container though, because we aren't virtualizing and running an operating system inside a container. Exit the container by typing `exit`, and the container will disappear.

How Containers Work

For demonstration purposes, I'm running Ubuntu as the host machine. For the above example I'm using a Ubuntu-based image only because that was easy to do. We can run anything as a base image inside the container. Let's run a CentOS-based image:

```
1 $ docker run -ti --rm centos /bin/bash
```

The same thing happens as before - Docker realizes we do not have a CentOS image locally so it downloads it, unpacks it, and runs the `/bin/bash` command inside the newly downloaded image. The new container is 100% CentOS.

```
1 [root@90a244e62ee3 /]# cat /etc/redhat-release
2 CentOS Linux release 7.1.1503 (Core)
3 [root@90a244e62ee3 /]#
```

The important thing to keep in mind is that we are not running an operating system inside of the container. This second container is using an image of CentOS's files to execute commands, while the first example was using an image of Ubuntu's files to run commands.

When Unix executes a command, it starts to look around for things like other executables or library files. When we run these things normally on a machine, the OS tells it to look in specific folders, like `/usr/lib/`, for those files. The libraries are loaded and the program executes.

What a container does is tell a process to look somewhere else for those files. In our latter example, we run the command under the *context* of a CentOS machine, and feed the executable CentOS libaries. The executable then runs as if it is on CentOS, even though the host operating system is Ubuntu. If we go back to Chapter 1 and look at the basic chroot example, it is the same idea. The container is just looking in a different spot for files, much like Bob has to look in a different spot for `/bin/bash`.

The other thing that happens is that, by virtue of some things built into Docker and the Linux kernel, the process is segregated off from the other processes. For the most part, the only thing that can be seen inside the container are the processes that are booted up inside the container. If we run `ps aux`, we can see that only the `bash` process we started with, as well as our `ps` command show up:

```
[root@cc10adc8847c /]# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.3  0.1  11752  2836 ?        Ss   00:51   0:00 /bin/bash
root        24  0.0  0.1  19772  2156 ?        R+   00:51   0:00 ps aux
```

If we run `ps aux` on the host machine we will see all the processes that are running on the host machine, including this bash process. If we spin up a second container, it will behave like the first container - it will only be able to see the processes that live inside of it.

Containers, normally, only contain the few processes needed to run a single service, and most of the time you should strive to run as few processes inside a single container as possible. As I've mentioned before, we aren't running full VMs. We're running processes, and by limiting the number of processes in a single container we will end up with greater flexibility. We will have more containers, but we will be able to swap them out as needed without too much distruption.

Containers, especially those running under Docker, will also have their own networking space. This means that the containers will generally be running on an internal network to the host. Docker provides wrappers for doing port forwarding to the containers which we will explore later in this book. Keep in mind though that the containers can reach the outside world, but the outside world may not necessarily be able to reach your containers.

Now, there's a whole lot of techincal things I'm leaving out here, but that's the gist of what we need to worry about. We're not running a full operating systems inside containers, just telling executables to use a different set of libraries and files to run against, and using various walls put up by the container system to make the process only see the files inside the image. Unlike a traditional virtual machine we are not booting up an entire kernel and all the related processes that will handle our application, we're simply running a program inside our PC and pointing it to a specific set of libraries to use (like using CentOS's libraries, or Debian's libraries).