

Docker For Developers



Chris Tankersley

Docker for Developers

Chris Tankersley

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

This version was published on 2016-04-28



* * * * *

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.

* * * * *

© 2015 - 2016 Chris Tankersley

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

Table of Contents

[Preface](#)

[Assumptions](#)

[Style Conventions](#)

[Containers](#)

[A Basic Container](#)

[Beyond Basic Containers](#)

[Along Comes Docker](#)

[Why We Should Care as Developers](#)

[Getting Started](#)

[Installing Docker](#)

[Running Our First Container](#)

[How Containers Work](#)

[Working With Containers](#)

[Images](#)

[Controlling Containers](#)

[Container Data](#)

[Networking](#)

[Containerizing Your Application](#)

[Getting PHP to Run](#)

[Getting a Database](#)

[Linking PHP to the MySQL Server](#)

[Getting a Web Server](#)

[Testing the application](#)

[Thinking About Architecture](#)

[Creating Custom Containers](#)

[Dockerfiles](#)

[Building a Custom Image](#)

[Docker Tools](#)

[Docker Machine](#)

[Docker Swarm](#)

[Docker Compose](#)

[Command Cheatsheets](#)

[Images](#)

[Containers](#)

[docker-machine](#)

[docker-compose](#)

Preface

This book, for me, is a long time coming. I've been following containerization for a long time, and got very interested in it in 2013 after seeing a presentation on dotCloud at the first MidwestPHP conference. I'd had previously used things like chroots and jails in my career, but containers took most of that a step further.

It wasn't too long after that that the Docker project was announced, in fact it was only about a week later, and was started by the very same company I had just seen a presentation on. I started playing with it right away. It was primitive, but it made building Linux containers much, much easier than working in straight LXC.

About a year ago I started thinking about writing this book. In March 2015 I threw up a basic website just to see what sort of interest there would be and I got a pretty decent response. My only concern was the Docker ecosystem. It was, and still is, very Linux-centric and the tooling was... well, it wasn't user friendly outside of Linux.

Over the last year there has been an explosion of software and Software-as-a-Services pop out of the woodwork to deal with Docker, and Docker finally announced what I thought were three major tools for working with Docker - being able to provision boxes easily, being able to cluster them, and being able to easily bring up multi-container environments. Yes, there were tools that did this made by third parties, but in this technological age I did not want to be bound to a particular vendor, and I did not want my readers to be bound by that as well. I wanted everything native.

Today we have almost everything a user needs in Docker, minus a few features that are still being worked on. I feel confident that I do not need to burden my readers with a specific vendor and that you, the reader of this book, can get Docker up and running for what you need.

I've written plenty of articles and taught plenty of classes, but I'm proud for this to be my first book. I hope you enjoy it.

Assumptions

I know, I know. Making assumptions makes... well, you know the rest. In any event, I'm writing this book for a developer who is familiar with PHP and is looking at Docker to either help with deployment issues, or development issues introduced through manually managing servers or keeping development environments in line. Maybe you are not happy with using a full virtualization stack like Vagrant, or are having issues with maintaining virtual machines across a wide variety of developers. This book will help you learn Docker, and possibly how it can help you deploy applications and make a better development environment for everyone.

This book will not teach you PHP. In fact, this book really doesn't care what kind of application you are building. I am only using PHP as an example since it is a well known and understood programming language that very heavily leans toward a multi-tiered application. Many modern PHP applications have a web tier, such as nginx, an application tier (PHP), and a database tier like MySQL. These types of applications are well suited for what we will be doing.

This book should work for any type of application that you want to set up in Docker, be it in Ruby, Python, or Go. This book focuses on full stack applications but will easily work with contained applications, such as C programs that are meant to be run from the command line. The ideas are all the same.

At the very least, read through this book if you want to know how to use Docker.

Style Conventions

Throughout the book there will be many times where a command is given as an example. Each command will have a \$ preceding the command, the command itself, and then possibly a result of the command output. These sections will be in a monospace font, and look similar to the following:

```
1 $ docker -v
2 Docker version 1.9.1, build a34a1d5
```

`docker -v` is the command to run, so if you copy and paste the example into your own terminal make sure to leave off the preceding \$. The second line is the output of the command. Some commands may have more lines of output, some none at all or have been omitted for brevity.

Many commands will be too long to properly fit on a single line on the book page and will be broken up using the `\` character. You can still copy and paste these commands into most terminals as they will properly process the `\` character. If you are manually typing these commands, feel free to leave out the `\` character and put the entire command on one line. For example:

```
1 $ docker run \  
2   --rm -ti ubuntu \  
3   bash
```

is equivalent to running:

```
1 $ docker run --rm -ti ubuntu bash
```


Containers

The development world goes through many different changes, just like technology in general. New ideas and concepts are brought out from various different places and we integrate them into our workflow. Some of these things are flashes in the pan, and other help revolutionize how we work and make our development lives easier.

About eight years ago virtualization started taking hold in the server room, and that eventually led to the need to run virtual machines locally. It was expensive and slow, and eventually technology caught up. Once virtual machines became disposable, we needed a way to maintain them, and we ended up with Vagrant.

Vagrant, like many things developers use, is a wrapper around an existing technology. Vagrant makes it easier to download, create, maintain, and destroy virtual machines, but it's not virtualization itself. That's handled by mature programs like Virtualbox or VMWare Workstation/Fusion. Vagrant puts a single interface on those technologies and allows us to share environments.

This is all well and good, but virtualization takes a heavy toll. Whenever we boot up a virtual machine, we are turning on a second computer (or third or fourth) inside our existing computer. Each virtual machine needs enough RAM and CPU to run that full little machine, and it takes it from your host machine. See Figure 1-1.

Eight years ago when I started using virtualization day-to-day this was an issue as my desktop machines rarely had more than four gigabytes of RAM and dual cores. Now I'm running a quad core box with twenty gigabytes of RAM and never notice the multiple machines inside of it. The cost of running virtual machines is still high but our machines are large enough to handle them without problems.

Chopping up a computer to allow multiple areas is not something new though, but it is something that is seeing the resurgence of this idea of Containers.

Containers are an alternative to virtual machines because instead of booting an entire machine inside of another one, containers section off processes to make

them think they are in their own little world.

There are many different ways that containers can be built, but the idea is pretty much the same. We want to separate processes from each other so that they don't interact and make it easier to deploy systems.

A Basic Container

As I said, this is nothing new. Containers have been around as an implementation detail in Unix-type systems since 1982 with the introduction of chroot. What chroot does is change the root directory for a process so that it does not see anything outside of the new root directory. Let's look at a sample directory structure:

Figure 1-1

```
1 /
2 |-> bin/
3 |   |-> which
4 |   |-> bash
5 |   |-> grep
6 |-> home/
7 |   |-> bob/
8 |       |-> bin/
9 |       |-> bash
10 |       |-> public_html/
11 |       |-> alice/
12 |-> usr/
13 |-> var/
```

Let us say that we have two users, Bob and Alice. Both are regular system users and can SSH into the computers. Alice is a normal user, so when she does `cd /`, she sees the folders `bin/`, `home/`, `usr/`, and `var/`. Depending on her privileges she can go into those folders, and call the programs `which`, `bash`, and `grep`. In fact, her shell is `/bin/bash`, so when she logs on that gets called. To her, she has full access to the system, and if you are using Linux, OSX, or any other system that is how it works for you.

Bob, however, is inside of a chroot. We've set his chroot to `/home/bob/` because he is a client of Alice's, and doesn't need full access to the machine. For Bob, when he SSH's in and runs `cd /`, he only sees `bin/` and `public_html/`. He can not see any higher in the directory tree than `/home/bob/` because the system has changed his root from `/` to `/home/bob/`. We've moved Bob off into his own little corner of the world.

This presents a few problems. If his shell is `/bin/bash`, we need to move a copy of that program into his world. So we now have two copies of `bash`, one in `/bin/` and another in `/home/bob/bin/`. We have to do this because Bob can't see anything higher than his `chroot`, and the regular system `bash` is outside of Bob's root.

We've put Bob inside his own container. He's free to do anything inside his container he wants without impacting the rest of the machine (barring running processes that use all of the system resources like I/O, RAM, or CPU usage). If he screws up and deletes everything by running `rm -rf /`, it's cool because he'll only destroy the files and folders in his `chroot`.

Beyond Basic Containers

The above situation is still used quite a bit today, and there are many different variations on the basic `chroot` setup. Basic `chroot` is OK for some things, but like mentioned above it doesn't really separate anything more than files. There was, and is, a clear need for something much more flexible and restrictive.

FreeBSD has a concept of BSD Jails which goes above and beyond by adding in things like disk, memory, and CPU quotas on FreeBSD systems. Solaris has Solaris Zones, introduced in 2004, which does pretty much the same thing as FreeBSD Jails but on Solaris systems. Many hosting companies run OpenVZ or Virtuozzo instead of full blown virtualization systems like Xen or KVM because they can cram many more containers onto a system than they can full virtual machines.

In 2008 Linux got LXC, or Linux Containers. LXC was a joint effort between groups like Parrallels (who run Virtuozzo), IBM, Google, and many other individual developers to bring containization to the Linux kernel.

All of these more powerful containers came about because of the lack of quotas and security in `chroot` (not that I think this is a fault of `chroot`, `chroot` was not designed to handle those concerns). Containers are useful not only because they help protect users from the rest of the system, but also because they are generally much less resource intensive on the host machine.

Much like Virtualization though, running containers was not something that was easily done. Many times it meant setting up networking or quotas manually, or installing extras onto a system that most system admins did not deal with. There

is a large amount of people that just do not know things like containers are even a thing.

Along Comes Docker

In 2013, dotCloud, a hosting company, released Docker, which was an internal project of dotCloud's that helped maintain their container technology. Instead of providing virtual machines, they used containers to scale and run their user's applications and needed a good way to maintain those containers. Docker was born from that.

Docker originally ran on top of Linux Containers, and did for LXC what Vagrant did for virtual machines. By installing Docker onto a computer, you could easily build your own containers, package them up, distribute them, and create and destroy them with very few commands. You no longer needed to know much more than you needed for something like Vagrant to start playing around with containers.

Docker consists of a client and server. The Docker client allows you to issue commands to a server, which will then start, stop, destroy, or do other things with containers or images (basic 'installations' of containers). The Docker client also allows you to build images yourself.

The Docker server does the heavy lifting of setting up networking, interacting with some sort of container technology to run the containers, and all the management cruft you no longer have to worry about as a sysadmin or a developer. New technologies for Docker also now allow you to provision machines from a command-line client through Docker Machine, or orchestrate complex multi-container setups through Docker Compose.

New OSes have popped up as well, such as CoreOS which is a minimal host operating system for running containers. We are seeing very small Linux distributions, like Alpine Linux, which are being designed to be run as bases for containers. Like Vagrant, an ecosystem is starting to sprout up around containers and related technologies that we as developers can start to use.

Why We Should Care as Developers

From everything I've described, this sounds like something that will be really helpful for our system administrators, and it is. I'm not going to downplay the great ability to quickly deploy a known system very quickly and repeatedly

great ability to quickly deploy a known system very quickly and repeatedly across pretty much any system.

What's in it for us though?

Much like virtualization has helped remove the “It works on my machine” plague that has been the bane of web developers for years, containers takes that to another level. Docker containers are identical once they are built, so as long as everyone is using the same container base image they are 100% the same.

This differs slightly from virtual machines handled by configuration management like Puppet, which are prone to ‘version creep’ over time. For example, a project I started was set up to install PHP and Apache. Puppet correctly downloaded Apache and PHP and set up mod_php. This worked just fine up until Ubuntu switched from using mod_php to using the PHP Filter module in favor of mod_php. Since Vagrant only provisions a box generally at the first boot now, I was left with mod_php while new people on the project were getting the PHP Filter module, which does have a few differences. Yes, this can be fixed by reprovisioning the box, but unless you know to do that (and I was only aware of the change because I had to actually work on servers where this was an issue) most developers aren't going to notice this.

The other advantage is that containers are generally small. They (generally) contain only the needed files for a single process and not entire operating systems so their footprint is small. This means they are small to download and small to store.

The major advantage I find is that it allows developers to swap out pieces of their application as needed. As a PHP developer I can test my application using PHP 5.4, 5.5, 5.6, and 7.0 by swapping out a single portion of my setup. Want to run unit tests against multiple PHP versions without running different ones on your system? Containers, and Docker, will allow you to do that.

This also leads us to the fact that containerizing processes allows us to keep host systems “pure.” You do not need to install PHP on your local system directly but can wrap it in an container. This keeps everything nicely packaged and you can quickly clean up systems without having all sorts of cruft in the host system. Couple that with the idea above and you have have multiple PHP versions running without conflict.

Throughout this book we'll explore setting up Docker and using it to our

Throughout this book we'll explore setting up Docker and using it to our advantage as developers.

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 disastrous 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 1.6. I'll go over some basic installation, but you can always refer to <https://docs.docker.com/installation/> 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's 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.

If you are running Windows or OSX I would also highly recommend setting up an Ubuntu virtual machine instead of using the installation instructions later in this chapter for those operating systems. The reason for this is because on non-Linux systems we will need to utilize a virtual machine to provide a way for our Docker containers, which rely on Linux subsystems, to function. OSX and Windows have a tool called Docker Toolbox which will set all of this up for you however. It is more seamless than the older boot2docker system, which was a third-party system that set up a VM and provided commands for working with the virtual machines. Docker Toolbox installation is detailed below.

Installing Docker

Ubuntu

Since Ubuntu ships with Long Term Release releases, I would recommend installing Ubuntu 14.04 and using that. The following instructions should work just fine for 14.10 or 15.04 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> and download the 14.04 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 14.04 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> and Desktop instructions can be found at <http://www.ubuntu.com/download/desktop/install-ubuntu-desktop>.

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 1.9.0, build 76d6bc9
```

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

```
1 $ docker run --rm hello-world
```

Ubuntu is all set up!

Windows 7/8/8.1

Microsoft is working on a native Docker implementation through HyperV, but as of the writing of this book it isn't finished. In the meantime we can use the excellent [Docker Toolbox](#) to set everything up for us. The Toolbox includes the Docker client, Docker Machine, Docker Compose, Kitematic, and VirtualBox. It does not come with Docker Swarm.

Start up the installer. When you get to Figure 2-1 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.

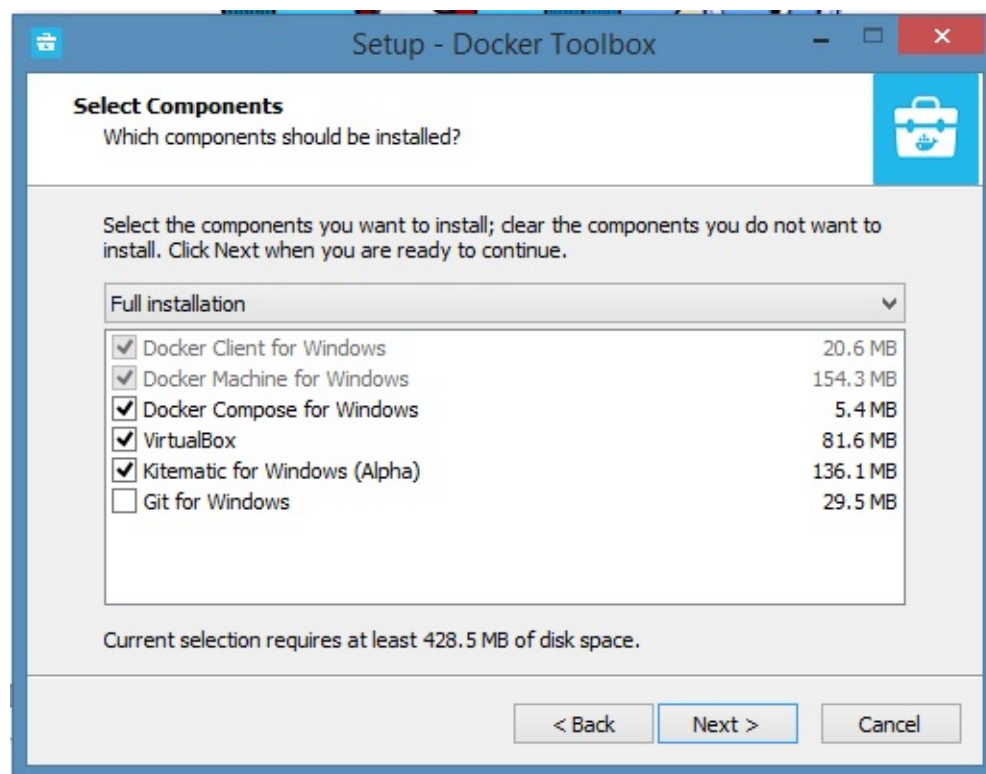


Figure 2-1

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.



Figure 2-2

```
1 $ docker -v
2 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 you 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.

```
1 $ docker-machine run --rm hello-world
```

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

```
MINGW64:/c/Users/Chris
Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com

For more examples and ideas, visit:
https://docs.docker.com/userguide/

Chris@sargonnas MINGW64 ~
$ |
```

Figure 2-3

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:

```
1 $ docker-machine rm default
2 $ 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 bug in the terminal. Try running the command by adding `winpty` to the beginning, like this:

```
1 $ 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, like Windows, does not have a native container or even virtualization system, so we are relegated to using a virtual machine. Like Windows, Docker has the [Docker Toolbox](#) 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`:



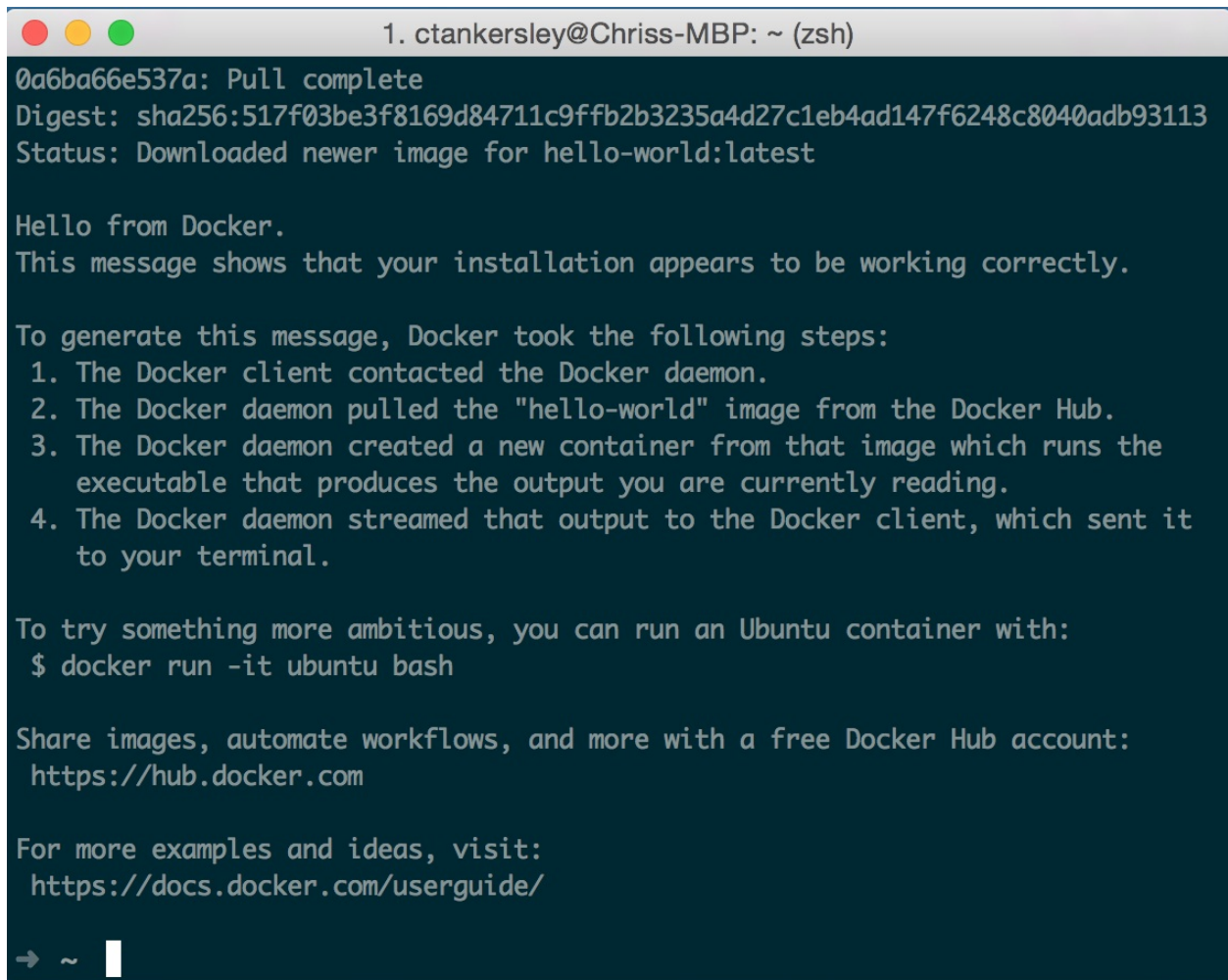
Figure 2-4

```
1 $ docker -v
2 Docker version 1.9.0, build 76d6bc9
```

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

```
1 $ docker run --rm hello-world
```

You should end up with output like in Figure 2-5.

A terminal window with a dark blue background and white text. The window title bar shows three colored circles (red, yellow, green) and the text '1. ctankersley@Chriss-MBP: ~ (zsh)'. The terminal output shows the Docker 'hello-world' message, including the pull status, digest, and a list of steps taken by Docker to generate the message. It also provides instructions on how to run an Ubuntu container and links to Docker Hub and documentation.

```
0a6ba66e537a: Pull complete
Digest: sha256:517f03be3f8169d84711c9ffb2b3235a4d27c1eb4ad147f6248c8040adb93113
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com

For more examples and ideas, visit:
https://docs.docker.com/userguide/

→ ~ █
```

Figure 2-5

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:

```
1 $ docker-machine rm default
2 $ docker-machine create --driver virtualbox default
3 $ 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:

```
1 $ docker run -ti --rm ubuntu bash
```

You'll see some output similar to Figure 2-6.

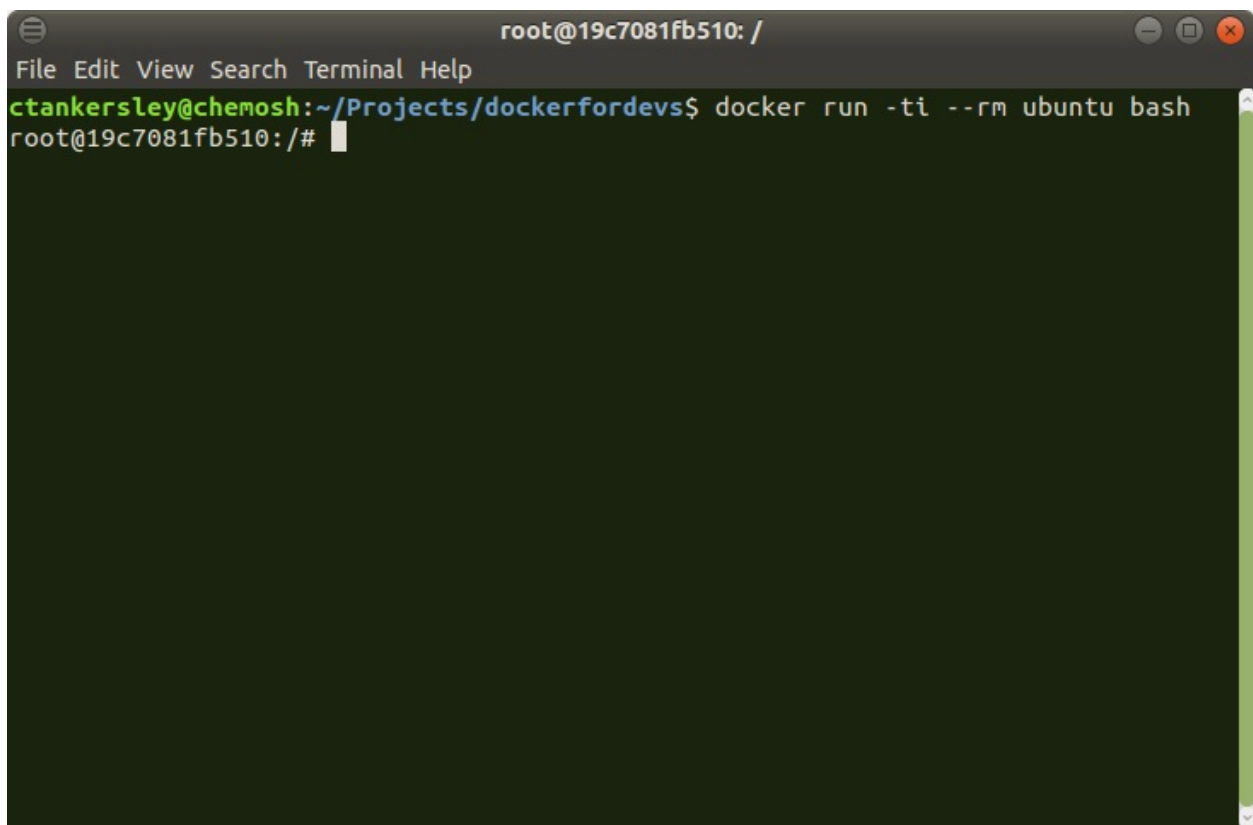


Figure 2-6

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 use 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 its 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 libraries. 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:

```
1 [root@cc10adc8847c /]# ps aux
2 USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
3 root         1  0.3  0.1  11752  2836 ?        Ss   00:51   0:00 /bin/bash
4 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

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 disruption.

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 technical 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 system 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).

Working With Containers

In Chapter 2 I talked a bit about how containers work in a very general sense. While we booted up a container and dug around a bit there is still a lot of things that docker brings that will make our lives easier as developers. Let's begin to start to dive deeper into Docker and see how some of these things work.

Images

The heart of Docker comes in the form of Images. Think of these as base boxes from Vagrant. Images are pre-compiled sets of files that are built for either being built upon, like the Ubuntu or CentOS images we looked at earlier, or already set to run your code, like a PHP or nginx image. We can build these images ourselves or download them from the Docker Registry, which is an online hub of published and freely available images for you to use.

Images are not containers, but are the building blocks of containers. We download an image from the internet (or build one ourselves), and then we use the image to create a container. This is like the definition of a class and an object - classes define how something works, whereas an object is the thing that we actually use. We can create many containers from a single image, and as we go along we will do that quite a bit.

For Docker, we can see what images are currently installed by running `docker images`:

```
1 dragonmantank@reorx:~$ docker images
2 REPOSITORY          TAG          IMAGE ID          CREATED
3 ubuntu              latest       07f8e8c5e660     2 weeks ago
4 centos              latest       fd44297e2ddb     3 weeks ago
```

If you have been following along with the book, your image list will look much like the above list. We have two images, ubuntu and centos, with specific IDs and file sizes (which I've omitted to space). There's also an Image ID, which designates a specific hash of an image. There is also a TAG header. We can have multiple versions of images. For example, we currently have the latest version of the Ubuntu image on our machine, which at the time of this writing is 14.04.1. If

we needed an older version, say 12.10, we can download that image using `docker pull` and supplying a tag name with our image name:

```
1 dragonmantank@reorx:~$ docker pull ubuntu:12.10
2 Pulling repository ubuntu
3 c5881f11ded9: Download complete
4 511136ea3c5a: Download complete
5 bac448df371d: Download complete
6 dfaad36d8984: Download complete
7 5796a7edb16b: Download complete
8 Status: Downloaded newer image for ubuntu:12.10
```

If you run `docker images` again you'll see two ubuntu entries, one set as 'latest' and the other as '12.10'.

The other thing to notice is that when we pull down an image, it will come in separate chunks. Each image is comprised of a set of layers that can be shared between images. The Ubuntu 12.10 image has five layers. The reason for this is twofold - it allows images to be shared and cached between other images and it allows for quicker building of images. If we publish a PHP image based on Ubuntu 12.10, if someone already has the 12.10 image downloaded and they pull our PHP image, they will only need to grab the new layer where we added PHP. This reduces the amount of storage space needed for multiple images.

Images can come from the Docker Registry, built from scratch using a Dockerfile, from a tarball of files, or from a private registry. We'll deal with the latter two options later, but I will generally see if there is an available image on the Docker Registry before building my own from scratch. Most of the time I will use a base image like Ubuntu and a Dockerfile and create my own with little work.

Controlling Containers

Docker helps add a bunch of nice wrappers around controlling and working with your containers. Docker allows us to run, start, stop, and destroy containers with a very easy syntax.

We've already seen how to initiate a container with `docker run`. Running a container kicks off the container by initializing the image, setting up the container, and starting the process inside of it. There are a couple of things we can do when we run the container.

```
1 $ docker run [options] IMAGE [command] [arguments]
```

[options] are options that will be used to configure the container. While there are a whole slew of them, we've been using three of them thus far. `-t` will start a TTY session inside the container that we can connect to, `-i` will run the container interactively meaning that it will be run like a normal process that we can control and interact with, and `--rm` tells the container to destroy itself as soon as it finishes. Putting this all together we end up with a `/bin/bash` process that we can interact with like a regular bash process, and it disappears when we type `exit`.

If we leave off `--rm`, the container will stop, but not delete itself, when we exit the container. If we have a process, for example a web server, we can leave that off so that our web server container doesn't disappear just because we stop it.

For containers that we do not need interactive access for, or want to run in the background, we can replace `-ti` with `-d`, which tells Docker to run the process in Daemon mode. This is like starting `nginx` or `Apache` using the `service` command, where the process kicks off and then gets shunted off to the background to run. It is there and available for us to interact with later, but it will not tie up our TTY session.

We have not used it yet, but one useful option we can pass is `--name`, which sets a specific name on a container. By default Docker will assign a random name to a container.

We'll look at more options as we go along.

`IMAGE` is the name of a Docker image we want to spawn. We've thus far looked at running an `Ubuntu 14.04.1`, `Ubuntu 12.10`, and `CentOS 7` image by virtue of the image names. You can search through the Docker Repository or build your own images and invoke them with the specified name.

[command] is the command we run inside the container itself. In our example so far, we've been using `/bin/bash`, so this runs `/bin/bash` inside the container. [arguments] are arguments for the [command] we are running, and are whatever arguments the [command] would normally take.

Now, once we have initiated a container with `docker run`, we can stop it with `docker stop [name]`.

If you want to start a stopped container back up, you can do that with `docker start [name]`. You'll notice this does not use the image name, but the name that Docker (or you via `--name`) set for the container. Many of the commands we will use going forward can use the name of a container.

If you need to re-connect to a container after you have stopped it and started it, you can use `docker attach [name]`. You'll be pushed back into the container and can interact with it like you just started it with a `run` command.

Over time you will probably forget which containers are stopped, or what their names are. You can see a list of all the registered containers that are running by using `docker ps`. This lists only the running containers, so if you want to see all the containers you can use `docker ps -a` to list everything. You'll see quite a few stats for the containers and how they were originally started.

Once you are all finished with a container, you can delete it manually if you did not set it to auto-delete with `--rm`. You can use `docker rm [name]` to remove a stopped container. If you want to stop and delete at the same time, you can use `docker rm -f [name]` to do a force delete.

Container Data

Up until this point, our containers have just been disposable blobs of binaries that we can run and then destroy. We are limited to whatever is inside the container when we pull it down (though, technically, we could download data into the container). Most of the stuff we will want to do, especially as developers, will require us to shove data inside containers, or be able to access the data inside of a container. Eventually we'll even want to be able to share data between containers.

Luckily, Docker has a concept called a Volume, which is a section of the image that is expected to have data that should be mounted into the image in some way. This is much like you plugging in a USB stick into your PC. The operating system knows there are entry points (be it in `/Volumes` if you are in OSX, or unused drive letters in Windows) where we will "plug in" or store data outside of the regular storage. We can tell Docker that a volume is either just a directory inside the image to persist or that the volume data will come from some external source such as another image or the host computer. Both have advantages and disadvantages, but we will discuss those later. For now, let's just start working with data.

For all of the Volume commands, we will start to use the `-v` flag to our docker run commands. You can even have multiple `-v` options to mount or create as many data volumes as we need. Some applications will need one, two, or many, and you may or may not mix host-mounted volumes with normal data volumes.

Mounting Host Directories

The quickest way to get files into a container is to just mount them into the container yourself. We can use `-v [hostpath]:[containerpath]` to mount a host directory or file into the container. Both `hostpath` and `containerpath` must be full paths. Relative ones will not work.

```
1 dragonmantank@reorx:~$ mkdir testdir
2 dragonmantank@reorx:~$ echo 'From the host' > testdir/text.txt
3 dragonmantank@reorx:~$ docker run \
4   -ti --rm -v `pwd`/testdir:/root \
5   ubuntu /bin/bash
6 root@eb10e0d46d2c:~# ls /root
7 text.txt
8 root@eb10e0d46d2c:~# cat /root/text.txt
9 From the host
10 root@eb10e0d46d2c:~# echo 'From container' > /root/container.txt
11 root@eb10e0d46d2c:~# exit
12 exit
13 dragonmantank@reorx:~$ cat testdir/container.txt
14 From container
```

The example is pretty straight-forward, but we created a directory with a file inside of it, ran a container that mounted our `testdir/` directory into `/root/` inside the container, and were able to read and write from it. Pretty simple!

The major downside to mounting host volumes in a production environment is that the container now has access to your host file system. Anything written to the shared volumes is now accessible by the host. It also means that the file paths must 100% match between production and development, which isn't always guaranteed. I did cheat a bit by using `pwd`, which ends up creating a full path, but it still must be taken into consideration.

I do not consider this a huge issue in development, but it is something to keep in mind when moving into production.

Windows Mounts

If you are a Windows user, the above will not work for you out of the box. This is due to Docker running inside of a virtual machine, which cannot see the host directories on your computer. There is a solution though!

Instead of using the Quickstart Terminal, open up Kitematic, which came with Docker Toolbox. At the bottom of the container list, in the lower left-hand corner, is a button for ‘Docker CLI’. Click that to get a custom Powershell to open up. This Powershell will be pre-configured for Docker, along with the ability to mount folders, much like the Quickstart Terminal.

The second caveat is that, because of the virtual machine, only certain folders are mountable by default. Anything that lives under C:\Users can be mounted inside Docker. We need to use a special path name to get it to work.

If I want to mount C:\Users\Chris\Projects\webapp, I need to use the following syntax:

```
1 $ docker run --ti --rm -v //c/Users/Chris/Projects/webapp:/opt/webapp ubuntu
```

The path will have all \ converted to /, the : will be removed, and a // will be added. This will allow you to mount directories properly on Windows.

This will not work using the Quickstart Terminal for some reason. The Quickstart Terminal will simply throw a “bad mode” error whenever you try to mount a folder.

Persisting Data with Data Volumes

This plays into more of a production environment, but normally you will need to persist data between containers, or even between rebuilds of containers. Normally when you stop a container, all of the data inside of it “reverts” back to the way it was when you started it up. This is intentional. Remember earlier when we pulled down the Ubuntu image and it grabbed all those layers? Docker runs based on those layers, and our files we create inside the container aren’t a part of those layers.

If we want data to persist data, we need to tell Docker about it. We can do that by passing -v [containerpath] with our create or run command, and Docker will section off that path as being persistent.

```
1 dragonmantank@reorx:~$ docker run \
2   -ti --name docker4devs -v /root \
3   ubuntu /bin/bash
4 root@5348b53817e3:/# cd /root
5 root@5348b53817e3:~# touch sample.txt
6 root@5348b53817e3:~# exit
7 exit
8 dragonmantank@reorx:~$ docker start docker4devs
```

```
9 docker4devs
10 dragonmantank@reorx:~$ docker attach docker4devs
11 root@5348b53817e3:/#
12 root@5348b53817e3:/# cd /root
13 root@5348b53817e3:~# ls
14 sample.txt
```

Volumes created in this way will stick around for as long as the container lives. If you destroy the container, you will destroy the underlying volume as well, so be careful.

This sort of setup works really, really well for containers that need to house storage, like database servers, or for keeping around logs. As I mentioned above, you can mix and match these persistent data volumes with host mounted volumes. Docker doesn't care as long as you are not trying to mount the same point inside the container twice.

Even More Persistence with Data Containers

We can take the idea of a Data Volume a step further though. Since Docker containers are just layers of a file system, and Data Volumes are persistent, we can share volumes between Docker containers. This is the first real exposure to some of the power of Docker, and containers in general. You can create entire containers that do nothing but store data, and they take up no more resources than just hard disk space.

I mentioned before that `docker run` will create and start a container for you. We can use `docker create` directly to generate a volume without running it, and specify data volumes to be generated at the same time.

```
1 $ docker create -v /root --name docker4devs_data busybox
```

This instructs Docker to create a container with a data volume located at `/root`. We aren't worried about running a command inside this container so we leave that part off, and since we are using this container for data we want a very small container. I like to use the `busybox` container for purposes like this, but if you are using a complicated application you might want to use an image that matches the data you are storing (for example, use the same `MySQL` image for both the app and the data volume).

This does not run the container, so you will not see it if you run `docker ps`. A `docker ps -a` will show it though, because it is technically just stopped. Again, that's fine, because we are just using it to hold data.

Docker allows us to link volumes with other containers by adding the `--volumes-from [name]` option. Since we have a data volume, let's take advantage of it.

```
1 dragonmantank@reorx:~$ docker run \
2   --rm -ti --volumes-from docker4devs_data \
3   ubuntu /bin/bash
4 root@85a3ba9c138b:/# cd /root
5 root@85a3ba9c138b:~# ls
6 root@85a3ba9c138b:~# touch file.txt
7 root@85a3ba9c138b:~# ls
8 file.txt
9 root@85a3ba9c138b:~# exit
10 exit
11 dragonmantank@reorx:~$ docker run \
12   --rm -ti --volumes-from docker4devs_data \
13   ubuntu /bin/bash
14 root@c231b69c8e9a:/# ls /root
15 file.txt
```

Here we generated a container that used the volume from `docker4devs_data`, and wrote a file to `/root`, which is actually the volume from our data container. When we exit the container it's destroyed, and when we create a new one we can see the file we stored in the data volume in `docker4devs_data`. We now have data persistence between containers that have life cycles!

We'll expand this further as we get further along in the book as we take advantage of sharing data between multiple running containers.

Networking

Part of working with containers is sectioning out concerns into different containers. Like with having to share data, we will have pieces of our application that will need to talk to each other. In a basic PHP application, we tend to have three layers to our application that need to talk together - the web server, the PHP process, and the database. While we could technically run them all inside of a container... we really should not. By splitting them into different containers we make it easier to modularize, but now we have to get them to talk together.

Docker automatically sets up and assigns IP addresses to each of the containers that run. For the commands that we have run thus far we haven't needed to care about this, but now we should start to dive into this aspect of Docker.

To show this off, we will create a few different containers. The first one we will create will be the PHP container, so that we can run a basic PHP script.

```

1 dragonmantank@reorx:~$ mkdir -p networking/html
2 dragonmantank@reorx:~$ echo '<?php phpinfo();' > networking/html/index.php
3 dragonmantank@reorx:~$ cd networking/
4 dragonmantank@reorx:~$ docker run \
5     -d --name app_php -v `pwd`/html:/var/www/html \
6     php:fpm

```

We generated a PHP container that is, for the moment, just going to mount a file from our host. We named it `app_php` just to make it a bit easier to work with in the following steps. If you run a `docker ps`, you'll also see that it is exposing port 9000 on it's IP address. This is not exposing it on the host IP, just the local address for the container. We can get this from the `docker inspect` command:

```

1 dragonmantank@reorx:~$ docker inspect \
2     -f "{{.NetworkSettings.IPAddress}}" app_php
3 172.17.0.16

```

`docker inspect` by default will return a bunch of information about the container, but in this case we are only looking for the IP address. In my case, it returns `172.17.0.16` as the address. If I stop and start this container again, it will get a new IP address.

Windows and OSX users will find that this IP will not work for them, due to the virtual machine Docker is running inside. You will need to use the IP of the virtual machine, which you can find with `docker-machine ip default`. You may also run into issues with exposed ports, which is further explained in the next section.

It's a pain to keep track of the IP address, but Docker has a built in way to somewhat discover and let containers know about the IP addresses of other containers. Let's spin up an nginx container and have it talk to PHP over port 9000. We'll need to modify the config for nginx to pass PHP requests to our PHP container. You can either use an nginx config you already have, or borrow the one from our sample app in [docker/nginx/nginx.conf](#) (copy only the server {} block from that file, do not use the entire file). Put the configuration in a new file called `default.conf`, and we will mount it inside the container.

```

1 dragonmantank@reorx:~$ mkdir -p config
2 dragonmantank@reorx:~$ vi config/default.conf
3 dragonmantank@reorx:~$ docker run \
4     -d --name app_nginx --volumes-from app_php \
5     -v `pwd`/config/default.conf:/etc/nginx/conf.d/default.conf \
6     --link app_php:phpserver \
7     nginx

```

Well, that run command is a doozy. We're going to create a container that runs in the background named `app_nginx`. This container will mount the volumes from `app_php` inside of itself, much like we did before with the data containers. We're also going to mount a config file inside of it, which contains our configuration so we can have nginx talk to the php-fpm process that lives inside `app_php`. That is all pretty standard from the stuff I have gone over so far.

What is new is the `--link [name]:[alias]` option. This tells Docker to link the named container to the container you are creating, with the alias you specify. In this case, we are linking the `app_php` container with the name of `phpserver`. Inside the `nginx` container, we can use `phpserver` as the domain name for our PHP container!

If I grab the IP address for my `app_nginx` container using `docker inspect`, I can visit that IP and be greeted with a basic PHP info screen, like in Figure 3-1.


<div> <div>PHP Version 5.6.9</div>  </div>	
System	Linux 21b09859fe07 3.19.0-16-generic #16-Ubuntu SMP Thu Apr 30 16:09:58 UTC 2015 x86_64
Build Date	May 26 2015 17:31:50
Configure Command	./configure '--with-config-file-path=/usr/local/etc/php' '--with-config-file-scan-dir=/usr/local/etc/php/conf.d' '--enable-fpm' '--with-fpm-user=www-data' '--with-fpm-group=www-data' '--disable-cgi' '--enable-mysqld' '--with-curl' '--with-openssl' '--with-pcre' '--with-readline' '--with-recode' '--with-zlib'
Server API	FPM/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/etc/php
Loaded Configuration File	(none)
Scan this dir for additional .ini files	/usr/local/etc/php/conf.d
Additional .ini files parsed	(none)
PHP API	20131106
PHP Extension	20131226
Zend Extension	220131226
Zend Extension Build	API20131226,NTS
PHP Extension Build	API20131226,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled

Figure 3-1

Linking containers together has a nice side effect in newer versions of Docker, at least since v1.9. Docker will now track the IP correctly when linked containers come up and down. For example, if you have the above `nginx` and `PHP` containers, and turn `PHP` on and off, `PHP` will more than likely get a new IP. This IP will be given to the `nginx` container automatically. Older versions of Docker set the IP statically when the container was built, so if `PHP` was at 10.10.0.2 when `nginx` came up, `nginx` would think `PHP` was at that IP until the

nginx container was restarted.

Exposing Ports

If we take a look at `docker ps`, we will see that the PHP container has an exposed port of `9000/tcp` and our nginx container has `80/tcp` and `443/tcp`. The network system in Docker will only route traffic to ports that are specifically exposed, which we can control at run/create time, or via config. In the case of our two containers, the original configuration for them told Docker to expose these three ports.

If you need to expose a port manually, you can do so with the `--expose=[ports]` option. We could do something like the following:

```
1 $ docker run -ti --rm --expose=80 ubuntu /bin/bash
```

These ports are also only exposed to the container IP, not the host IP. Many times though we need want to expose a container to the outside world, such as in the case of our nginx container. We can use the `-p [port]` option to expose a port to the outside world.

```
1 $ docker run -ti --rm -p 80 ubuntu /bin/bash
```

This command will assign port 80 to a random port on the host. We can get what that host port is by running `docker ps`, or by running `docker port [name] [port]`. It is a random port though, and every time you expose it you will get a different host port.

If you want to assign an exposed port to a specific host port, you can do so with `-p [host port]:[port]`, like with the following:

```
1 $ docker run -ti --rm -p 8080:80 ubuntu /bin/bash
```

This command will map port 8080 on the host to port 80 on the container. You can also specify the option as `-p [IP]:[host port]:[port]` if you need to bind the port to a specific IP address instead of `0.0.0.0`. You will also need to be aware that, like normal, ports below 1024 are generally reserved and will need root access to bind to. If you want your nginx container to listen on port 80 of the host, you may need to run the `docker create/run/start` command as root instead of the normal user you use for Docker.

For Windows and OSX users, you will run into a few issues with port mapping due to Docker running inside of a virtual machine. Since Docker is running inside of a virtual machine, the basic `-p [port]` will not work for you to access the containers. The Virtual Machine is acting as the 'host' for the container, so you will need to use `-p [host port]:[port]` for any examples going forward.

Containerizing Your Application

Now that we understand how containers work, and a bit about how we can work with the containers, let's start to take an application and convert it over to using Docker. At first we will do this all by hand, but then we will begin to look at other tools to make it slightly easier. We will start with 'stock' images and move to fine tuning everything.

We are going to take a simple PHP application, written in Zend Expressive, and chop it up into containers. Normally, an application will live inside of a single virtual machine that is managed by something like Vagrant. Having all of the parts of the application running in a single Docker container, while doable, presents some unique issues. Docker is a single-process-per-container ideal, so having our PHP, web server, and database all inside a single container is not the best. We will split up things based on spheres of influence.

First, go to <https://github.com/dragonmantank/dockerfordevs-app> and clone the repository to your local machine. For the purposes of demonstration, don't do anything else. We'll handle all of the setup and such in due time.

Getting PHP to Run

While we have a few moving parts to our application, the first thing we want to get up and running is a PHP container. Our demo application is a simple Zend Expressive PHP application with a docroot at `html/` instead of the normal `public/`. This is done to make it easier to work with our container we use, which assumes a docroot of `/var/www/html/`. If you followed the directions though, our application won't run. Modern PHP applications do not generally tote around their vendor/ directory and instead rely on Composer to do our dependency injection. Let's pull down the dependencies for the project.

```
1 $ docker run --rm -u $UID -v `pwd`: /app composer/composer install
```

This first initial run will download the image as you probably do not have this `composer/composer` image installed. This container will mount our project code, parse the `composer.lock`, and install our dependencies just like if we ran `composer` locally. The only difference is we wrapped the command in a `docker`

container which we know has PHP and all the correct libraries pre-installed to run composer.

Now that we have our dependencies, let's get a basic container running to hold our application. To start off with we will run a simple container using PHP 5.6 in running the development server:

```
1 $ docker run \  
2   -d -v `pwd`: /var/www --name testphp \  
3   php:5.6-fpm php -S 0.0.0.0:80 -t /var/www/html
```

If everything works as it should, you should get the initial page of our application. I chose the 5.6-fpm container only because we will use it later on, but the container happily will run our development server for the moment.

Now, our app isn't going to work because we don't have a database, but we are on the right track.

We are skirting the issue of having a web server running, but we're going to start to build to that. For the moment we are going to just run using the test server. We have taken the first steps however as now the application is running inside of a container. Since the code is mounted via a hosted volume you can edit the code with your favorite editor, and the changes will be reflected right away.

Not a great departure from using a VM, though if you are using Windows or OSX the performance might be slightly worse because of the shared folders. That is due to the way that Virtualbox mounts directories, not due to Docker.

Let's move on.

Getting a Database

PHP is actually going to be one of the easiest parts of the application to get running. That container doesn't need to keep any sort of real state since the files are mounted from our host machine, and if the container disappears it's not a big deal. Our database, however, needs to be persistent between machine restarts. That means we will need either make a data volume inside our database container, or make a data volume. For now, let's make a simple data volume inside the container.

```
1 $ docker run \  
2   -d --name mysqlserver \  
3   mysql:5.6 --innodb-buffer-pool-size=1024M
```

```

3     -v /var/lib/mysql \
4     -e MYSQL_DATABASE=dockerfordevs -e MYSQL_ROOT_PASSWORD=docker \
5     mysql

```

This will create a container named `mysqlserver` with a database named `dockerfordevs` and a root password of `docker`. We also specified that `/var/lib/mysql`, the default folder that databases are stored, will be persistent for this container. We can start and stop this container as much as we want and we will be fine from a data perspective.

We can connect to this MySQL server using network links. Let's run the MySQL command line client.

```

1 $ docker run \
2     -ti --link mysqlserver:mysqlserver --rm \
3     mysql mysql -h mysqlserver -u root -p

```

Enter 'docker' for a password, and you will get the familiar MySQL command line client interface. Run `show databases;` and you should get output like the following:

```

1 mysql> show databases;
2 +-----+
3 | Database |
4 +-----+
5 | information_schema |
6 | dockerfordevs      |
7 | mysql              |
8 | performance_schema |
9 | sys                |
10 +-----+
11 5 rows in set (0.00 sec)

```

Great! Exit out of her by typing 'exit', and this mysql-client docker container will disappear into the ether.

Linking PHP to the MySQL Server

We're closer than before. The only problem is now that our PHP container can't see the MySQL server. Granted, we can get the IP of the MySQL server and pass it in through the config of our script, but then the IP won't be persistent. The better way is to link the two containers together. We will remove our testphp container and rebuild it with the link.

```

1 $ docker rm -f testphp
2 $ docker run \
3     -d --link mysqlserver:mysqlserver -v `pwd`:var/www \

```



```
4     --name testphp \  
5     php:5.6-fpm php -S 0.0.0.0:80 -t /var/www/html
```

At this point we can change the database config of our application to point to 'mysqlserver' as a DB hostname, 'dockerfordevs' as a DB name, root' as a DB username, and 'docker' as a DB password.

That's actually about it from a development perspective. At this point you can work on your application like you would any other application. Mostly. Go ahead and play around with the test application and get a feel for editing it.

Getting a Web Server

Let's tear this down a bit. One of the main reasons that Docker is used is because it can mirror your production environment locally by virtue of the containers. The containers you deploy to production can be the exact same ones that you deploy locally. This goes a step further than configuration management sharing like Puppet or Ansible, because even if you share the same configuration management files with your VM, there will be slight differences. For example you may decide to install the php5-cli package on both production and your VM. Over time your VM and production will get different minor versions (5.6.1 vs 5.6.11) which might lead to differences.

Let's mirror production a bit more by putting an actual web server into the mix. For this example we'll set up nginx since that works well with php-fpm. Let's delete our PHP container and set it up more properly. We are also going to change the name to 'phpserver' while we are at it.

```
1 $ docker rm -f testphp  
2 $ docker run \  
3     -d --link mysqlserver:mysqlserver -v `pwd`:var/www \  
4     --name phpserver \  
5     php:5.6-fpm
```

Docker Hub has an nginx container already set up, so we can utilize that. It does not automatically set up PHP for us, so we are going to supply it with a custom nginx.conf file. If you are using the git repo for our project, there should be an nginx.conf file in the docker/ folder. We will use this to pass a more appropriate nginx configuration to the nginx container.

```
1 $ docker run \  
2     -d --name nginx --link phpserver:phpserver \  
3     -v `pwd`:var/www -v `pwd`/docker/nginx/nginx.conf:/etc/nginx/nginx.conf:ro \  
4     nginx
```

There isn't anything more complicated here than we showed in Chapter 3, or the above containers. We're creating a container named 'nginx', linking it to our new 'phpserver' container, mounting our files inside of it for our static content, and mounting a single file as our configuration.

Testing the application

Let's get the IP address of our server. We'll use that IP to test the parts of the application. Since we haven't exposed the port to the outside world we can't just go to localhost.

```
1 $ docker inspect -f "{{.NetworkSettings.IPAddress}}" nginx
```

In your browser, head over to <http://ip-of-container/api/v0/queue>. You'll probably see the following error in Figure 4-1, which is OK. We'll fix it in a second.

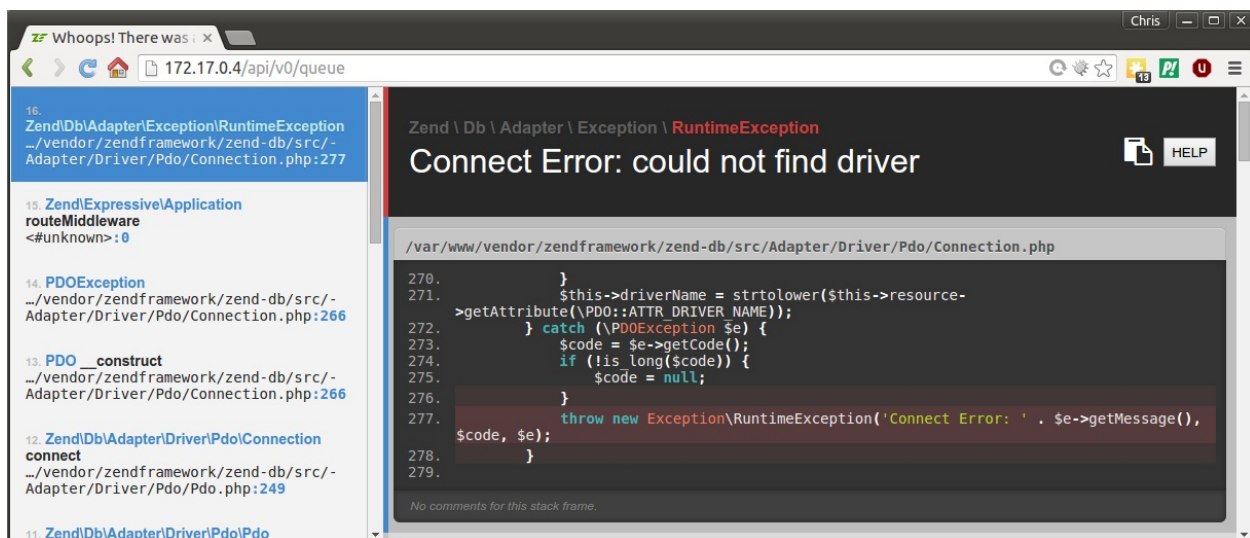


Figure 3-1

What we've run into is a failing of the PHP image we're using. For what I'm assuming is to conserve space, the PHP containers do not have very many PHP extensions installed. We will need to build a custom PHP image. In the next chapter I'll go over the nitty-gritty details of building our own custom container but I've also supplied a custom build script for this application.

In the code, run the following command from the root of the app:

```
1 $ docker build -t d4d_php docker/php
```

After a few moments we will have a custom image created from the PHP image we've been using thus far. Let's switch over to using it instead.

```
1 $ docker rm -f phpserver
2 $ docker rm -f nginx
3 $ docker run \
4   -d --link mysqlserver:mysqlserver -v `pwd`: /var/www \
5   --name phpserver
6   d4d_php
7 $ docker run \
8   -d --name nginx --link phpserver:phpserver \
9   -v `pwd`: /var/www -v `pwd`/docker/nginx/nginx.conf:/etc/nginx/nginx.conf:ro \
10  nginx
```

This removes the old PHP and nginx containers, and creates a new PHP container that uses the custom image we just created. We then re-create the nginx container, which is linked to the new PHP container, which has the MySQL and PDO extensions installed. Grab the IP of your new nginx container and try going to `http://ip-of-container/api/v0/queue`.

You'll see Figure 3-2.

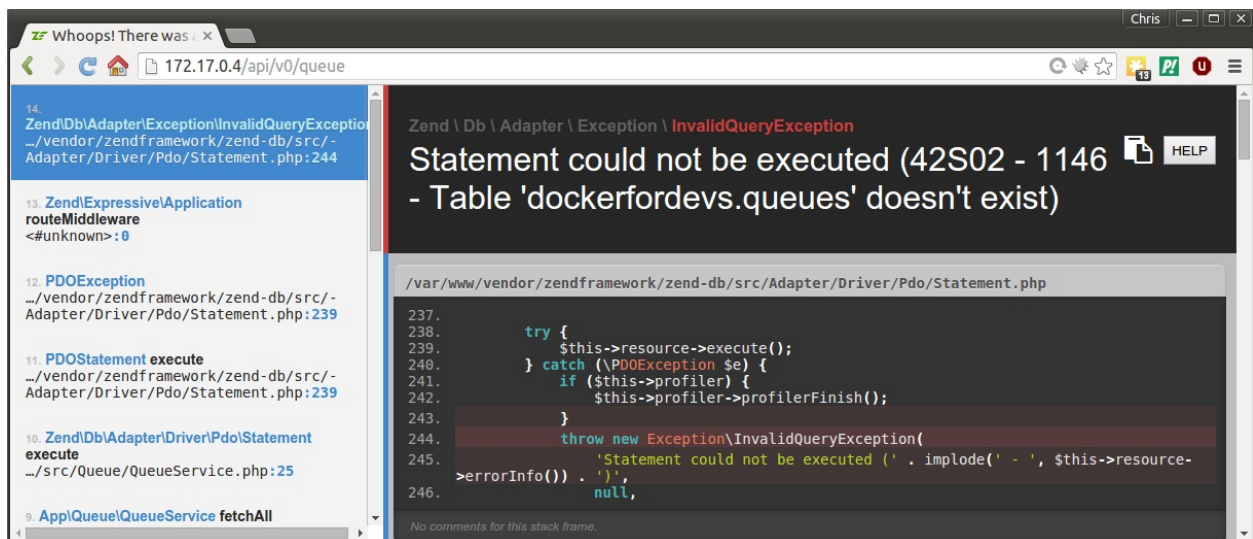


Figure 3-2

Ah, yes, we have no database to work with. In a real application you need to do things like database migrations or other things during deployment, and we haven't created any databases. We've potentially complicated things because there's a good chance your developer PC doesn't have PHP installed, since we're running all of this from containers, or connecting to the databases are a bit harder because the database server is in a container.

Since this is all in containers, we can just use a container to run our database migrations. The first thing we'll need to do is create a settings file for our migration system, [Phinx](#). Phinx uses a YAML file to connect to the database, so we need to run a command to generate it. The command is long, but much like what we've run before.

```
1 $ docker run \  
2   --rm -ti --link mysqlserver:mysqlserver \  
3   -v `pwd`: /var/www -u $UID -w /var/www \  
4   d4d_php \  
5   php vendor/bin/phinx init
```

The only major things we've added are `-u` and `-w`. `-u` allows us to change the UID of the user inside of the container. Ubuntu, at least, sets up a variable called `$UID` which holds your user's ID. By using it, the files that Docker writes will be matched to your user instead of root.

`-w` changes the working directory of the command. Normally Docker runs commands from the system root directory of `/`, but we want to use `/var/www` since that's where our code lives.

The `init` command creates a `phinx.yml` file in the root of the project. Since configuration is different from environment to environment I don't normally put this file into version control.

At the top of the file, change the migrations line to:

```
1 migrations: %%PHINX_CONFIG_DIR%%/data/migrations
```

Change the `development:` section of `phinx.yml` to:

```
1 development:  
2   adapter: mysql  
3   host: mysqlserver  
4   name: dockerfordevs  
5   user: root  
6   pass: docker  
7   port: 3306  
8   charset: utf8
```

Yaml doesn't allow us to pass environment variables into it, so we are statically setting these for right now.

Now we need to run the database migrations. I've already supplied migrations to set up the application, we just need to run them.

```
1 $ docker run \  
2   --rm -ti --link mysqlserver:mysqlserver \  
3   -v `pwd`: /var/www -u $UID -w /var/www \  
4   d4d_php \  
5   php vendor/bin/phinx migrate
```

The migrate command in Phinx will parse all of the migration scripts in data/migrations/ and apply them in the correct order. Now, if we visit our app we should see Figure 3-3.

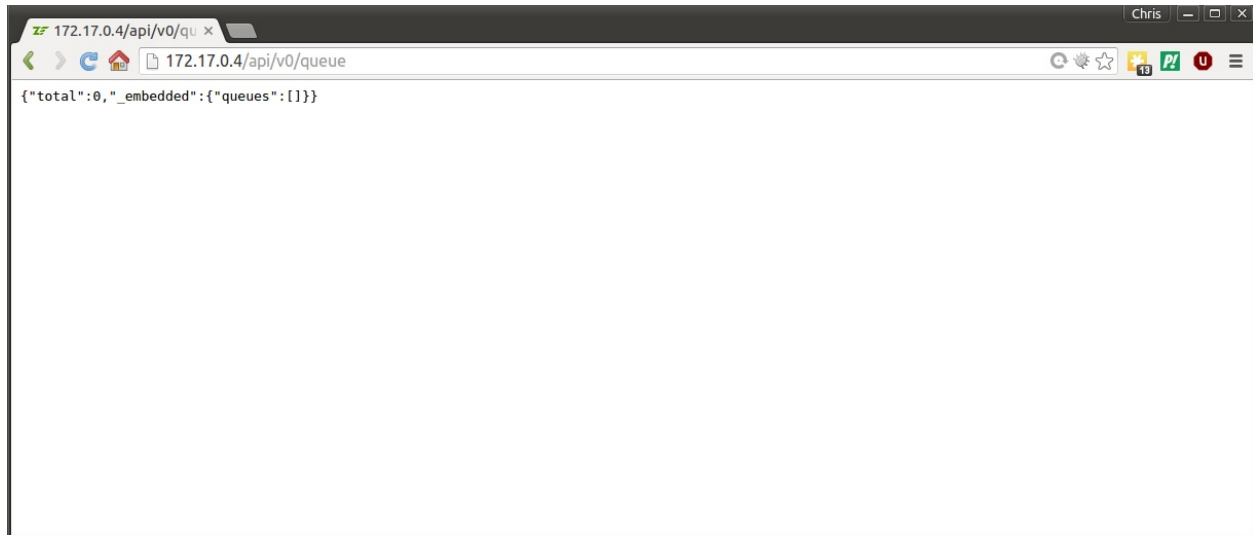


Figure 3-3

Success! Our application is now running inside Docker, all ready for data. That URL can also take a POST request to add a queue. Figure 3-4 shows the output of a POST request, and Figure 3-5 shows the API output when we have queues in the system.

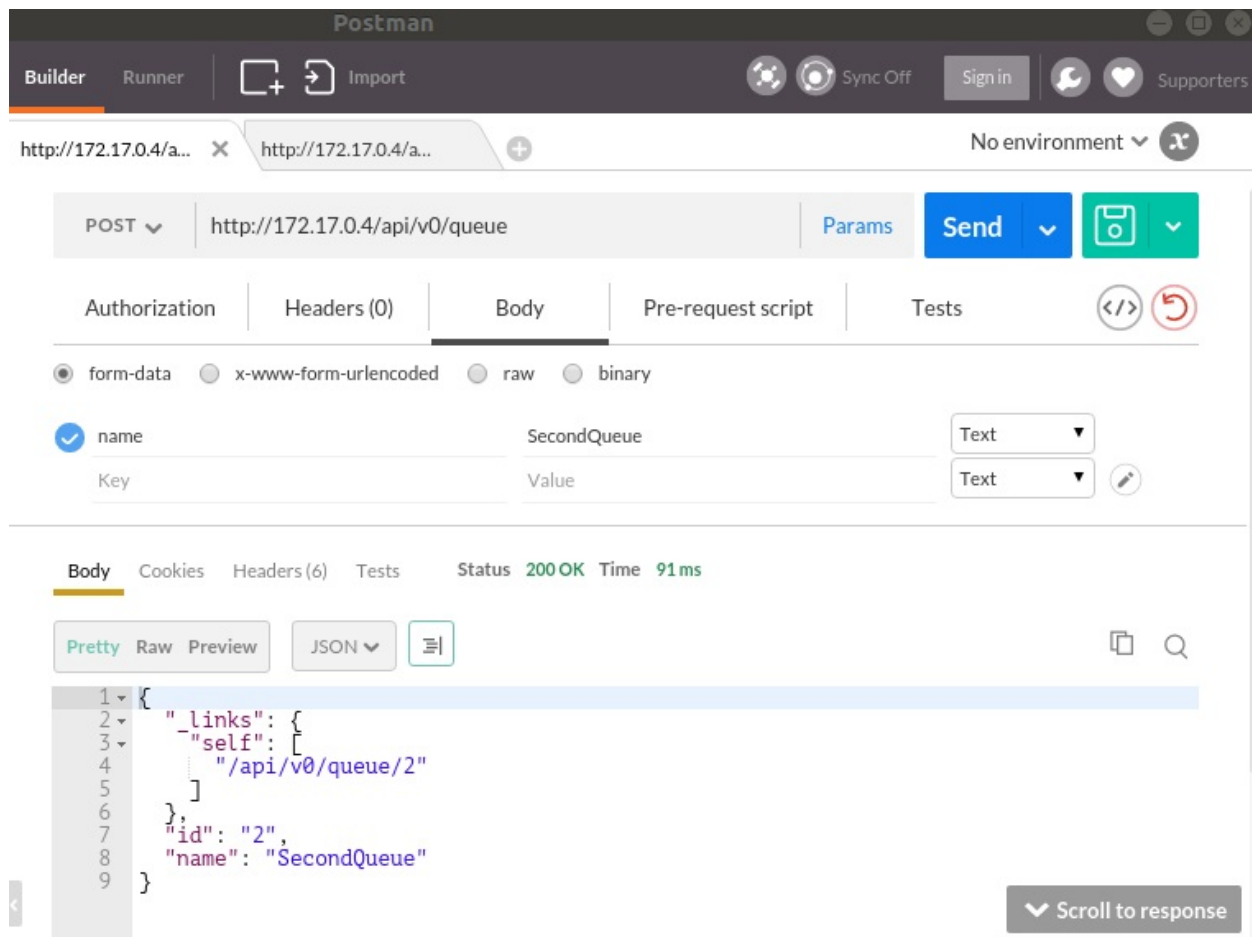


Figure 3-4

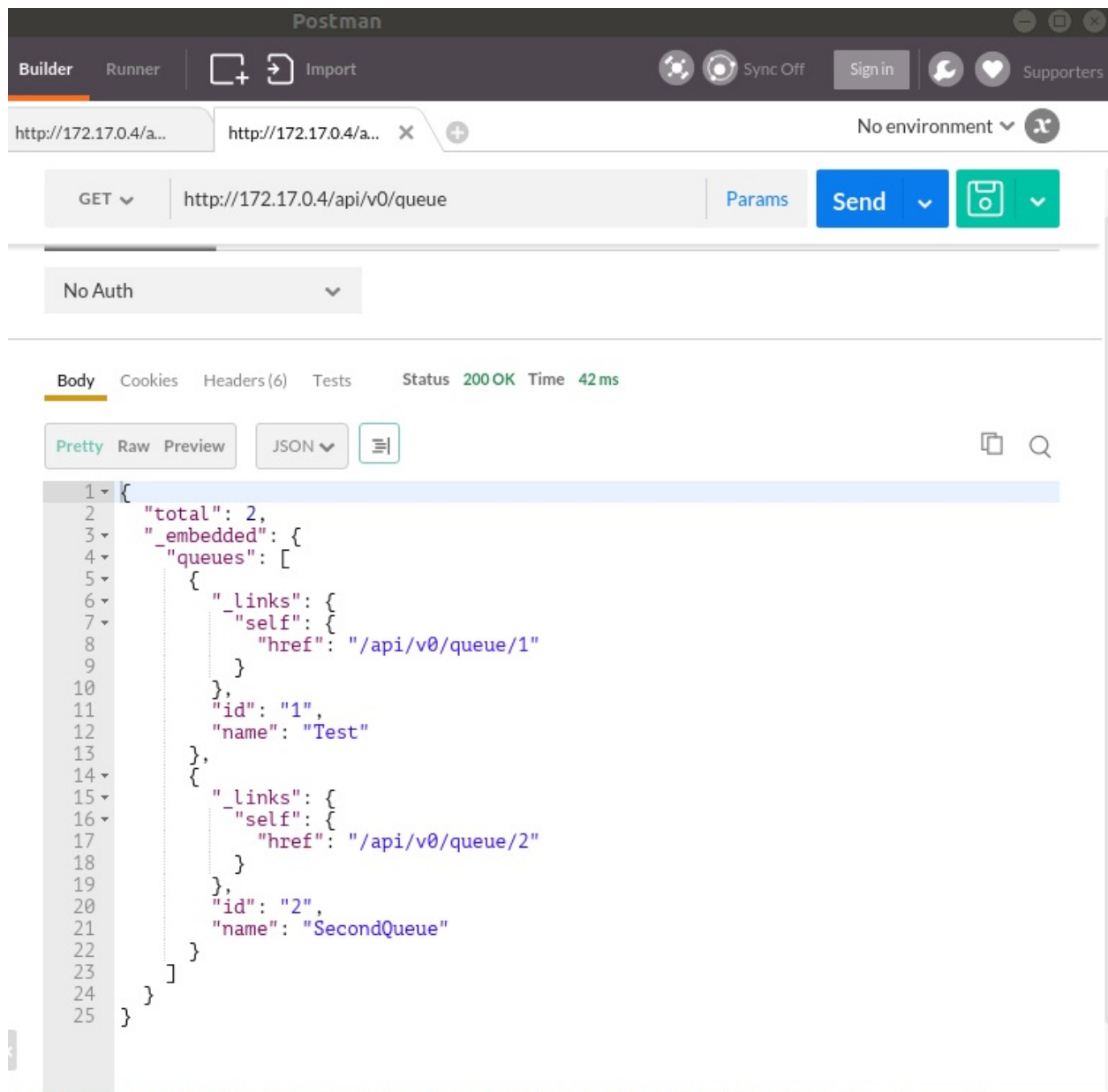


Figure 3-5

Thinking About Architecture

I will be honest, we didn't do much here but put into place many of the commands we talked about in Chapter 3. What I wanted to show is that when we start to look at using Docker, we need to start thinking about the architecture of our applications. Many of us are not building small little single-script command line scripts, though we did use Composer inside of a Docker container to show that you can do that. Our applications are multi-layered and are more than just PHP scripts. They are web servers, script interpreters, database servers, queue engines, third party APIs, data stores... we can break down many of our

applications into smaller chunks.

None of the above should have required any code changes at all. If you take a look at your application, the only thing you might have needed to change is having a small blip of code to help use the PHP development server run a bit more sanely. All we did we take each chunk of this app - the web server, PHP scripts, and database - and section them off. By sectioning things off we can quickly swap them out.

Want to make sure that our application works on multiple versions of PHP?

```
1 $ docker run \
2   --rm -v `pwd`:/var/www \
3   php:5.4-cli /bin/sh -c 'cd /var/www && php vendor/bin/phpunit'
4 $ docker run \
5   --rm -v `pwd`:/var/www \
6   php:5.5-cli /bin/sh -c 'cd /var/www && php vendor/bin/phpunit'
7 $ docker run \
8   --rm -v `pwd`:/var/www \
9   php:5.6-cli /bin/sh -c 'cd /var/www && php vendor/bin/phpunit'
10 $ docker run \
11   --rm -v `pwd`:/var/www \
12   php:7.0-cli /bin/sh -c 'cd /var/www && php vendor/bin/phpunit'
```

It's as simple as that. From an application perspective if we want to run our application on PHP 5.5 because that's what a client's server is running, we simply build our PHP section of the application with `php:5.5-fpm` instead. At worst you have to bounce the other containers, but being able to quickly swap around containers is an incredibly powerful tool.

There are many containers out there, and even many tutorials, that say it's not a bad thing to run multiple processes in a single container. Many containers for PHP container nginx or Apache inside of them. In fact, the PHP container we're using has an Apache option. The downside to running multiple processes inside one container is flexibility. If I wanted to use multiple versions of PHP, or swap between database engines, or test against nginx and Apache, I would have to build custom containers for all of those situations. By breaking it out I can quickly swap them with little work.

That's not to say that custom containers are bad. The nginx container we used in this chapter requires special configuration. There also is not a container that handles every situation. Next chapter we'll look at building our own custom containers.

Creating Custom Containers

Thus far we have been using containers that have been created by either Docker or the community, through the [Docker Hub](#). These containers were pre-built to handle a specific problem, such as running PHP, a web server, a database server, composer, etc. There is nothing inherently wrong with these containers and, as far as general containers go, many will be good enough for what we need.

There will come a time where you either do not like how a container is set up, like the nginx or the MySQL container, or you need specialized setups. A good example is the MySQL container. An older version of it was nearly 2.5 gigabytes of space because the maintainer did not remove all the source code (the new version clocks in at a much more manageable 350 megabytes). It was possible to create a much smaller one, so I would build a MySQL container myself.

You may also not trust a container, or be able to vet a container, and want to build one yourself. Some situations and industries require a more stringent auditing of software, so building your own containers might be quicker than trying to explain to auditors that some third party container is safe.

The final reason for building your own container is for deployment. As I've mentioned before you can use host mounting for volumes, but that isn't scalable. Data volumes are not necessarily transferable between hosts. This leaves us with deploying our code inside of a container, which means we have to build a custom container.

Whatever the reason is, building a custom container is fairly easy. We'll define a list of steps that need to be applied, much like a bash script, and in the end we'll have an image.

Dockerfiles

Docker has a build system that uses a settings file called a 'Dockerfile.' The Dockerfile contains all the instructions needed for building an image. There are a few new commands we will use as well as commands for doing things like exposing ports and creating volumes

exposing ports and creating volumes.

For a quick reminder, we deploy containers (actual things that run) from images (copies of a system). So we will build a new Image, and then deploy that as a running Container.

The format of a Docker file is very simple. We have two basic things:

```
1 # Comments
2 INSTRUCTION arguments
```

Comments are denoted as a line that starts with a '#'. Comments help explain or remind you what you did six months ago. There's a good reason you needed that weird sounding package.

Instructions are generally a single word, like RUN, ENV, or WORKDIR. We then supply an argument to the instruction. Instructions apply a change to the image that is being built. Remember how when we pull down an image there are multiple layers? Each instruction, or change, to a base image is a new layer. Docker supports the following instructions:

- ADD
- ARG
- CMD
- COPY
- ENTRYPOINT
- ENV
- EXPOSE
- FROM
- LABEL
- MAINTAINER
- STOPSIGNAL
- ONBUILD
- USER
- VOLUME
- WORKDIR

I will cover most of the common ones, further documentation can be found at <http://docs.docker.com/engine/reference/builder/>.

Let's take a look at a very simple Dockerfile, in Figure 5-1.

Figure 5-1

```
1 FROM ubuntu:14.04
2 MAINTAINER Chris Tankersley <chris@ctankersley.com>
3
4 RUN apt-get update && apt-get install -y \
5     nginx \
6     && apt-get clean \
7     && rm -rf /var/lib/apt/lists/*
8
9 COPY nginx.conf /etc/nginx/nginx.conf
10
11 EXPOSE 80 443
12
13 CMD ["nginx", "-g", "daemon off;"]
```

The Dockerfile is pretty easy to read, even if you have never looked at one before. If you are using the git repository for this book, you will find a copy of the docker file in `docker/Dockerfile` as well. I'll break it down line by line:

1. Build this container from the `ubuntu:14.04` tag
2. Set a `MAINTAINER` flag, useful for when sharing on Docker Hub
3. Update the apt cache, and
4. and install the nginx package
5. and clean the apt cache
6. and remove and leftover lists to conserve space in the image
7. Copy the `nginx.conf` file from the local system into `/etc/nginx/nginx.conf` in the container
8. Expose two ports, 80 and 443
9. Start nginx using `nginx -g daemon off;`

Most Dockerfiles will contain `FROM`, `MAINTAINER`, `RUN`, `ADD/COPY`, `CMD`, `EXPOSE`, and `VOLUME`. If these instructions do not seem to fit the bill for what you want to do, check out the full Dockerfile documentation that I linked to above.

FROM

```
1 FROM <image>[:tag]
```

While we can create containers from an existing filesystem already on our computers, Docker has a mechanism for using an existing image as a base. In our example Dockerfile, we declared `FROM ubuntu:14.04`, meaning our custom image will use the `ubuntu:14.04` image as a base, and we will run all the commands from there.

Any Docker image can be used as a base image, and it is recommended to use

one of the official base images to build your images. Docker recommends using the Debian image they supply as it is small, tightly controlled, and is a well-known distribution.

You must supply at least an image name (for example, ‘ubuntu’, ‘php’, ‘debian’). If you do not, the latest tagged version will be used. If you need a specific tag, you can supply the full ‘image:tag’ format.

MAINTAINER

```
1 MAINTAINER <name> <email>
```

Sets the author field on generated images.

RUN

```
1 RUN <command>
2 RUN ["executable", "param1", "param2", ... ]
```

RUN runs a specific command inside the image that will be persisted. In our sample Dockerfile, we use RUN to update our apt cache and install nginx. Each RUN command in a Dockerfile will generate a new layer in your image, so it is recommended to try and combine as many like commands together. In our sample Dockerfile we put together four commands to install nginx and clean up apt, otherwise there would be four separate layers.

There are two forms for the RUN command, the first being the ‘shell’ form and the second being the ‘exec’ form. The ‘shell’ format will run the commands inside the shell that is running inside the container (for example, /bin/sh). If your container doesn’t have a shell, or you need a command to be run exactly without variable substitution, use the ‘exec’ format.

ADD and COPY

```
1 COPY <local/file/path> </image/file/path>
2 ADD <http://domain.com/file> </image/file/path>
3 ADD <file.tar.gz> </image/path/>
```

ADD and COPY will move files into the image, but each one works slightly differently. Most of the time you are going to want to use COPY to move a file from outside of the image to inside.

COPY only works to move a file or folder into the image. It does not work on

remote files.

ADD will move a file or folder into an image as well, but it supports remote URLs as well as auto-extraction of tar files. If you have a local files packaged into a .tar or .tar.gz you can use ADD to extract them inside of the container. The same works for remote tar files as well.

If you do not need the files after extraction (for example, you are downloading installer files and you no longer need them after install), then you are actually better off downloading the files using curl or wget and then deleting them when you are finished through the RUN instruction. ADDing an archive results in the files being committed to a layer, where downloading, running, and removing the files via RUN will not commit the layer until the entire RUN instruction is complete.

CMD

```
1 CMD ["executable", "param1", "param2", ... ]
2 CMD ["param1", "param2", ... ]
3 CMD <command> [arguments]
```

CMD is generally the command that will be run when you create and start a container if the user doesn't pass another one. In our sample Dockerfile, the CMD will start up nginx for us as a foreground application. If the “executable” portion of the CMD is left off you can use it in conjunction with an ENTRYPOINT instruction to provide default parameters.

The first form of CMD will work outside of a shell inside the container, while the third form will run inside /bin/sh -c. Most of the time you will probably want the first form.

In chapter 4 where we pass php -S to the php-fpm container, we are overriding the default CMD. If you always want a command to run, use ENTRYPOINT.

ENTRYPOINT

```
1 ENTRYPOINT ["executable", "param1", "param2", ... ]
2 ENTRYPOINT <command> [arguments]
```

ENTRYPOINT configures a container to run as an executable. In Chapter 4 we used the ‘composer/composer’ container to run composer for us. This container isn't designed to execute anything except composer.

Using ENTRYPOINT versus CMD depends on what you want to do. For example, we used the following command to start up the development server in Chapter 4:

```
1 $ docker run \  
2   -d -v `pwd`:/var/www --name testphp \  
3   php:5.6-fpm php -S 0.0.0.0:80 -t var/www/html
```

We can simplify our command down a bit by using the following in a Dockerfile:

```
1 FROM php:5.6  
2 EXPOSE 80  
3 ENTRYPOINT ["php", "-S", "0.0.0.0:80"]  
4 CMD ["-t", "/var/www/html"]
```

This sets the command that will always execute to `php -S 0.0.0.0:80`, and we can override the `-t /var/www/html` if we want to. If we build this Dockerfile as ‘phpdevserver’, we can run it like the following:

```
1 $ docker run -d -v `pwd`:/var/www phpdevserver  
2 // Or to override the path we are watching  
3 $ docker run -d -v `pwd`:/opt/app phpdevserver -t /opt/app/html
```

This does lock our container down to always running a specific command, but still allow us to modify the command slightly as we need.

EXPOSE

```
1 EXPOSE <port1> [port2 port3 ...]
```

This exposes a port that the container will listen on. This does *not* bind the port to the host machine, just simply exposes it through the containers IP address. If you need to bind an EXPOSEd port to the outside world via the host, you will need to use the `-p` parameter on `docker run`.

You simply specify a list of ports, separated by a space, and Docker will open them on the container.

VOLUME

```
1 VOLUME ["/path/inside/image"]  
2 VOLUME /path/inside/image
```

This creates a placeholder for a volume inside of an image. This is useful for things like automatically creating a volume to hold log files, as containers that are started will automatically set these volumes up without you having to specify them via the `-v` flag on `docker run`.

You can use a combination of the `VOLUME` instruction with `-v` as well, so you are not limited to just the volumes that the Dockerfile creates.

Building a Custom Image

The Dockerfile is just a recipe though, it doesn't do anything special by itself. We use it to build an image using the `docker build` command. We can build this with the following command:

```
1 $ docker build -t customnginx .
```

We are adding a customer tag with `-t customnginx` so that it is easier to find and re-use this image. We then supply it a directory where the Dockerfile exists. By using `.` we are telling Docker to build in the current directory. You should see some output like below:

```
1 $ docker build -t customnginx .
2 Sending build context to Docker daemon 3.584 kB
3 Step 1 : FROM ubuntu:14.04
4 14.04: Pulling from library/ubuntu
5
6 Digest: sha256:d4b37c2fd31b0693e9e446e56a175d142b098b5056be2083fb4afe5f97c78fe5
7 Status: Downloaded newer image for ubuntu:14.04
8 ---> 1d073211c498
9 Step 2 : MAINTAINER Chris Tankersley <chris@ctankersley.com>
10 ---> Running in 60f06fa69695
11 ---> 1d1bbaca5635
12 Removing intermediate container 60f06fa69695
13 Step 3 : RUN apt-get update && apt-get upgrade
```

There will be a lot more as it runs through the commands. Each Step is a layer in the total image. It may take a few minutes to build the image, but once it is finished you should be able to run `docker images` and see our custom image in there.

```
1 $ docker images
2 REPOSITORY          TAG          IMAGE ID          CREATED
3 customnginx         latest      2af5f1cfc29e     6 seconds ago
```

We can now use this image instead of the generic `nginx` container we were using before, and no longer have to mount our custom `nginx.conf` file through the command line.

```
-- -- --  
1 $ docker rm -f nginx  
2 $ docker run \  
3   --name nginx --link phpserver:phpserver -v `pwd`:var/www -d \  
4   customnginx
```

Last chapter I talked about how important it was to be able to separate out the components of your application. Here we started with a generic nginx container by using the nginx image, and we've swapped that out using our custom customnginx image without having to touch the PHP or the MySQL layer.

Docker Tools

I've purposely held off talking about using some of the more nice tools that are in Docker so that you get a good base for what goes on when the tools themselves are being run. Since I've started this book, Docker has managed to bring in a few different types of applications that make running and managing Docker a much nicer experience. By making the following applications official we also skirt the issue of what happens if a tool stops being developed.

Docker Machine

Docker, like just about any software, can be installed from a package. If you followed along with this book thus far you've either installed Docker through a package repository or by using an software download from docker.com. That's all pretty standard. Doing it this way doesn't scale well though. Docker has a solution for that in the form of Docker Machine, an additional bit of software that makes provisioning machines for Docker much easier.

If you have installed Docker Toolbox, you are all set to use it. If not, head over to <https://github.com/docker/machine/releases/> and follow the instructions for setting it up. More than likely you will just download a zip file and extract it to the appropriate places.

Once installed we can use to to create a brand new, pre-configured machine that we can use directly or in conjunction with another tool, Docker Swarm.

To create a new machine we will run the `docker-machine create` command and specify the appropriate options for the driver. Docker Machine supports many different providers for creating hosts, such as:

- Amazon EC2
- Digital Ocean
- Microsoft Azure
- VMWare
- VirtualBox
- HyperV

and many, many others. Each service requires their own additional configuration options, but for the purposes of demonstration I'll set up a new machine on [Digital Ocean](#).

We will need to get an API access token from Digital Ocean. If you do not have an account head over to <https://digitalocean.com> and sign up for a new account. Once you are in your account click on 'API', and then generate a new token. We'll need this to create a new machine. Once we have an API token we can create the machine using the following command:

```
1 $ docker-machine create \  
2   --driver digitalocean \  
3   --digitalocean-access-token [token] \  
4   dockerfordevs  
5  
6 $ docker-machine ls  
7 NAME                ACTIVE    DRIVER        STATE     URL  
8 dockerfordevs      -         digitalocean  Running   tcp://XXX.XXX.XXX.XXX:2376
```

This doesn't get us to use the machine however. We'll need to set that up by getting the config options, which you can see by running `docker-machine env dockerfordevs`:

```
1 $ docker-machine env dockerfordevs  
2 export DOCKER_TLS_VERIFY="1"  
3 export DOCKER_HOST="tcp://XXX.XXX.XXX.XXX:2376"  
4 export DOCKER_CERT_PATH="/home/user/.docker/machine/machines/dockerfordevs"  
5 export DOCKER_MACHINE_NAME="dockerfordevs"  
6 # Run this command to configure your shell:  
7 # eval "$(docker-machine env dockerfordevs)"
```

The last line of the output tells you how to switch over to using the new host. However, Windows users will need to manually run each line, replacing `export` with `SET`, as Docker Machine does not fix the output for Windows machines yet.

Once you have run the `eval` command you will be pointing at the new machine host, and any commands you run will be run against the remote host.

You can stop, start, and restart the machine remotely using `docker-machine`, and even SSH into it. There is even a built-in command to copy files to the remote host which can be used to help mount host volumes, but as I've mentioned before that's not the best way to scale up an application. When you are finished with a machine, you can `rm` it and the machine will be destroyed.

Docker Swarm

Docker Machine has an additional capability that I did not cover in the previous section, since it pertains to a separate tool. Docker, as we have been using it, has all been running on a single machine. One of the big selling points of Docker is that you can make it much easier to deploy your code, and in this day and age we are not always deploying everything to one machine. There are also the issues of appropriately using resources on hardware, and as hardware gets bigger and bigger we don't want to waste CPU or memory that is just idle.

Docker Swarm allows us to link together Docker installations into one giant mega-installation, across many different computers, be it physical or virtual. We can start to balance our deployments automatically, or do things like putting together scaling across multiple machines, using our Docker containers. Swarm handles all of the bookkeeping and distribution of containers to the various Docker hosts.

Under Docker Swarm, we run containers against the swarm instead of a specified machine. Docker Swarm will handle finding an appropriate host, starting the image, and any other things it needs to do to get the image running. From there you can query the swarm, or the individual container, to get information such as networking or which host it physically resides on.

The easiest way to get started using Swarm is in conjunction with Docker Machine. In the previous section, some of the options we can send to the docker machine command is a series of `--swarm*` options that we can use to configure a machine to be part of a Docker Swarm.

Creating a Swarm

A Swarm consists of a Swarm Master, which keeps track of everything, and any number of Swarm hosts, which is where the containers will actually run, of which the master is also a part. Before we begin we need to get a new Token, which is how all of our swarm machines will know they are part of the same swarm. Luckily there is a swarm image we can use to help us generate a new Token.

```
1 $ docker run --rm swarm create
2 // ...
3 40122bb69c98825b4ac7094c87a07e21
```

If this is the first time you have used the swarm image, it will download image. The very last line of the output will be a valid token that we can use. If you run it again in the future, you will get just the token

again in the future, you will get just the token.

For this tutorial, 40122bb69c98825b4ac7094c87a07e21 will be the token we will use. Whatever token you generate, keep track of it. You will need it to join other nodes into the swarm.

Now that we have a token, let's create a basic Swarm master:

```
1 $ docker-machine create -d virtualbox \
2   --swarm \
3   --swarm-master \
4   --swarm-discovery token://40122bb69c98825b4ac7094c87a07e21 \
5   swarm-master
```

In addition to the normal docker-machine output that you would see, you will also see a line saying `Configuring swarm...`, which will be setting up the Swarm information for this container, which will be the stuff we need to run the swarm master. In addition to running the swarm, it will act as a normal node in the swarm.

Speaking of which, let's create two nodes now:

```
1 $ docker-machine create -d virtualbox \
2   --swarm \
3   --swarm-discovery token://40122bb69c98825b4ac7094c87a07e21 \
4   swarm-node-1
5 // Output from machine creation
6
7 $ docker-machine create -d virtualbox \
8   --swarm \
9   --swarm-discovery token://40122bb69c98825b4ac7094c87a07e21 \
10  swarm-node-2
11 // Output from machine creation
```

If we run `docker-machine ls` we can see that they exist. I removed the DRIVER column to save space:

```
1 $ docker-machine ls
2 NAME          ACTIVE  STATE    URL                                SWARM
3 swarm-master  -       Running  tcp://192.168.99.100:2376         swarm-master (master)
4
5 swarm-node-1  -       Running  tcp://192.168.99.101:2376         swarm-master
6 swarm-node-2  -       Running  tcp://192.168.99.102:2376         swarm-master
```

We can see that each one is up and running, which swarm they are connected to (swarm-master, the name of the master), and which one is actually the master. Let's connect to the master, and get some information from it to see how our nodes are:

```

1 $ eval $(docker-machine env --swarm swarm-master)
2 $ docker info
3 Containers: 4
4 Images: 3
5 Role: primary
6 Strategy: spread
7 Filters: health, port, dependency, affinity, constraint
8 Nodes: 3
9  swarm-master: 192.168.99.100:2376
10   L Containers: 2
11   L Reserved CPUs: 0 / 1
12   L Reserved Memory: 0 B / 1.021 GiB
13  swarm-node-1: 192.168.99.101:2376
14   L Containers: 1
15   L Reserved CPUs: 0 / 1
16   L Reserved Memory: 0 B / 1.021 GiB
17  swarm-node-2: 192.168.99.102:2376
18   L Containers: 1
19   L Reserved CPUs: 0 / 1
20   L Reserved Memory: 0 B / 1.021 GiB
21 CPUs: 3
22 Total Memory: 3.064 GiB
23 Name: 1d3f17ec089e

```

We can see the number of containers we have running (one on each node running the docker-swarm client, and two on the master, one for the master software and one for the client software), that we have three CPUs allocated, and three gigabytes of memory. Your output will also have a Label section, showing that these machines are running Boot2Docker, which is a lightweight ISO designed for running inside of a virtual machine. It is also the name of a suite of software similar to docker-machine, which has mostly fallen out of favor since the introduction of docker-machine. docker-machine still uses the ISO however.

Let's create some containers and see where they land.

```

1 $ docker run -d --name nginx1 nginx
2 $ docker run -d --name nginx2 nginx
3 $ docker run -d --name nginx3 nginx
4 $ docker ps
5 CONTAINER ID        IMAGE               PORTS              NAMES
6 1b88eda4bcf0        nginx              80/tcp, 443/tcp    swarm-master/nginx3
7 1813990f820b        nginx              80/tcp, 443/tcp    swarm-node-1/nginx2
8 99cffaa613b0        nginx              80/tcp, 443/tcp    swarm-node-1/nginx1

```

Under the NAMES column we can see which node the containers are running on. Since each container gets a unique name across the entire system, we can use 'nginx1', 'nginx2', or 'nginx3' to work with the containers without having to know which node they are on.

What you will find next is something that Windows and OSX users have already discovered. You can't access any of these containers, even if you use docker

inspect to get their IP. The containers are unreachable. We did not do anything wrong however.

Each node in the cluster is running inside of a virtual machine. We can't access those containers right now from our host machine that Swarm is running on. We will need to specify that the ports should be mapped to external IP address, and then use the IP address of the node to connect to it. In fact this is true of any host that we provision our swarm on: without any external ports being mapped, we can't access any of the containers. This isn't something that's special to Virtualbox.

Let's fix this for now:

```
1 $ docker rm -f nginx1 nginx2 nginx3
2 $ docker run -d -p 80:80 --name nginx1 nginx
3 $ docker run -d -p 80:80 --name nginx2 nginx
4 $ docker run -d -p 80:80 --name nginx3 nginx
```

This starts the containers and binds them to the external port 80 of their host, so now we can go to the various IPs of the nodes in our system. In mine case, it's 192.168.99.100, 192.168.99.101, and 192.168.99.102.

What happens when we try to add a fourth box?

```
1 $ docker run -d -p 80:80 --name nginx4 nginx
2 Error response from daemon: unable to find a node with port 80 available
```

Swarm is smart enough to check each of the nodes to find the appropriate node to run our container. Since we only have three nodes, and all three nodes are already bound to the host port 80, we can't run any more containers that bind to the host's port 80. Swarm will take into consideration RAM, CPU, and disk space when trying to decide where to allocate a new container.

So why go through setting up Swarm when you can just manually provision boxes and launch instances on them? Docker Swarm allows us to point to a single instance (the master) and run docker commands against it. The master will then find an available machine and start the container on one of them, without you as a developer caring where they need to go. We are starting to get a rudimentary system in place for scaling up our systems and load balancing them.

Docker Compose

Docker Compose is based on an older orchestration tool called ‘[Fig](#).’ The idea of Fig was to make it easy to define full-stack applications in a single configuration file yet still help maintain the “one process per application” design of Docker. Docker Compose extends this idea into an official tool that will allow you to define how applications are structured into Docker containers while bundling everything up into an easy-to-deploy configuration.

Compose will read a file and build the necessary `docker run` commands that we need, in the proper order, to deploy our application. Since you should understand how all of this works from a manual process, Compose will just make the work you have done thus far much, much easier.

Compose starts with a `docker-compose.yml` file which contains all of the information on each container inside of an application, and how it should be configured.

```
1 phpserver:
2   build: ../docker/php
3   volumes:
4     - /home/ctankersley/Projects/dockerfordevs:/var/www/
5   links:
6     - mysqlserver
7
8 mysqlserver:
9   image: mysql
10  environment:
11    MYSQL_DATABASE: dockerfordevs
12    MYSQL_ROOT_PASSWORD: docker
13  volumes:
14    - /var/lib/mysql
15
16 nginx:
17   build: ../docker/nginx
18   ports:
19     - "80:80"
20     - "443:443"
21   links:
22     - phpserver
```

By knowing how to use `docker run` to start up our application containers, the `docker-compose.yml` file is fairly straight-forward. We give each container a specific name, such as ‘mysqlserver’, ‘nginx’, or ‘phpserver’. Each container can use a Dockerfile build to build from, like we do with our ‘nginx’ container, or just point to an image that we can modify through parameters, like our ‘mysql’ and ‘phpserver’ containers. Since each container has a unique name we can link them together using the `links:` key. We can specify the volumes that we want to both preserve, in the case of the ‘mysqlserver’ container, as well as the directories that we want to mount, in the case of the ‘phpserver’ container.

Any of the things that we can configure through `docker run` can be configured through the `docker-compose.yml` file.

To boot up a Compose configuration, go into the directory containing the `docker-compose.yml` file, which in our case is in the root directory of our application. We can boot the entire configuration using `docker-compose -d up`, and we should see something similar to the following:

```
1 $ docker-compose up -d
2 Creating dockerfordevs_mysqlserver_1...
3 Creating dockerfordevs_phpserver_1...
4 Creating dockerfordevs_nginx_1...
5
6 $ docker-compose ps
7
8      Name                                Command                                State      Ports
9 -----
9 dockerfordevs_mysqlserver_1    /entrypoint.sh mysqld                Up         3306/tcp
10 dockerfordevs_nginx_1         nginx -g daemon off;                 Up         443/tcp, 80/tcp
11 dockerfordevs_phpserver_1     php-fpm                              Up         9000/tcp
```

Each container in the configuration gets appended with the name of the project, which is picked up from the folder that the configuration is in or specified with `-p`, the name of the actual container, and a number. The number is important because we can actually scale up the number of each container as needed. For example, we can scale the PHP servers up to four total containers:

```
1 $ docker-compose scale phpserver=4
2 Creating dockerfordevs_phpserver_2...
3 Creating dockerfordevs_phpserver_3...
4 Creating dockerfordevs_phpserver_4...
5 Starting dockerfordevs_phpserver_2...
6 Starting dockerfordevs_phpserver_3...
7 Starting dockerfordevs_phpserver_4...
```

By default this will not link new instances, just boot them up. If we do another `docker-compose up -d` Compose will rebuild the containers, using the existing data volumes, and rebuild the `/etc/hosts` file inside the appropriate containers (in this case, our nginx container). Whenever you do a scale up or down, you will want to run up again. Keep in mind that this will not affect any configuration in your application, so unless your nginx config already understands how to find and use the additional PHP servers, nginx will still only use what is in the configuration.

You can stop all of the containers at once with `docker-compose stop` or, if they are stuck, `docker-compose kill`. If you are all finished, `docker-compose rm` will destroy all of the containers.

A full listing of all the options can be found at <http://docs.docker.com/compose/compose-file/>.

Compose and Swarm

Right now, using a native setup of Compose and Swarm, they work together but not in a load-balanced sort of way. Docker's networking and volume support do not traverse hosts, so when you do a `docker-compose up`, it will attempt to build each container on the same host. Depending on your application and your Swarm nodes this may or may not work for your application. Docker is currently working on network-aware linking and volume support, but it is not there now. You may need to look into a different orchestration system if you need cross-host volumes and linking.

Command Cheatsheets

Images

- `docker images` - List out all of the images on the host machine
- `docker rmi [image]` - Removes a Docker image if no containers are using it
- `docker pull [image][:tag]` - Downloads a tag from a registry

Containers

- `docker ps [-a]` - Lists containers currently on the system
- `docker run [options] IMAGE [command] [arguments]` - Creates a container and starts it
- `docker start [name]` - Starts a running container
- `docker attach [name]` - Re-connects to a container that is running in the background
- `docker stop [name]` - Stops a running container
- `docker rm [-f] [name]` - Removes a container
- `docker create [options] IMAGE [command] [arguments]` - Creates a container

docker-machine

- `docker-machine create --driver [driver] [machinename]` - Creates a new Docker Machine
- `docker-machine ls` - Lists all of the available machines
- `docker-machine env [machinename]` - Returns the information about a machine
- `docker-machine start [machinename]` - Starts a stopped machine
- `docker-machine stop [machinename]` - Stops a running machine
- `docker-machine restart [machinename]` - Restarts a machine
- `docker-machine ip [machinename]` - Returns the IP of a machine
- `docker-machine rm [machinename]` - Destroys a machine
- `docker-machine ssh [machinename]` - SSH's into a machine

docker-compose

- `docker-compose up -d` - Builds a project
- `docker-compose ps` - Shows container information for project
- `docker-compose rm` - Destroys a project
- `docker-compose stop` - Stops a project
- `docker-compose scale [container]=[size]` - Adds or removes containers for scaling