Lesson 1: Containers from 30,000 feet

Containers are a new *thing* for a lot of people.

In this chapter we'll give some background and scratch the surface of topics like; why do we have containers, what do they do for us, and where can we use them.

The bad old days

Applications run businesses. If applications break, businesses suffer and sometimes die. These statements get truer every day!

For the most part, applications run on servers. And in the past we could only run a single application per server. The open-systems world of Windows and Linux just didn't have the technologies to safely and securely run multiple applications on the same server.

So the story usually went something like this… Every time the business needed a new application, IT would go out and buy a new server. And most of the time nobody knew the performance requirements of the new application! This meant IT had to make guesses when choosing the model and size of servers to buy.

As a result, IT did the only reasonable thing - it bought big fast servers with lots of resiliency. After all, the last thing anyone wanted - including the business - was under-powered servers. Under-powered servers might be unable to execute transactions, which might result in lost customers and lost revenue. So IT usually bought bigger servers than were actually needed. This resulted in huge numbers of servers operating as low as 5-10% of their potential capacity. A tragic waste of company capital and resources!

Hello VMware!

Amid all of this, VMware, Inc. gave the world the virtual machine (VM). And almost overnight the world changed into a much better place! Finally we had a technology that would let us run multiple business applications on a single server safely and securely.

This was a game changer! IT no longer needed to procure a brand new oversized server every time the business asked for a new application. More often than not they could run new apps on existing servers that were sitting around with spare capacity.

All of a sudden we could squeeze massive amounts of value out of existing corporate assets, such as servers, resulting in a lot more bang for the company's buck.

VMwarts

But… and there's always a *but!* As great as VMs are, they're not perfect!

The fact that every VM requires its own dedicated OS is a major flaw. Every OS consumes CPU, RAM and storage that could otherwise be used to power more applications. Every OS needs patching and monitoring. And in some cases every OS requires a license. All of this is a waste of op-ex and cap-ex.

The VM model has other challenges too. VMs are slow to boot and portability isn't great - migrating and moving VM workloads between hypervisors and cloud platforms is harder than it could be.

Hello Containers!

For a long time, the big web-scale players like Google have been using container technologies to address these shortcomings of the VM model.

In the container model the container is roughly analogous to the VM. The major difference through, is that every container does not require a full-blown OS. In fact all containers on a single system share a single OS. This frees up huge amounts of system resources such as CPU, RAM, and storage. It also reduces potential licensing costs and reduces the overhead of OS patching and other maintenance. This results in savings on the cap-ex and op-ex fronts.

Containers are also fast to start and ultra portable. Moving container workloads from your laptop, to the cloud, and then to VMs or bare metal in your data center is a breeze.

Linux containers

Modern containers started in the Linux world* and are the product of an immense amount of work from a wide variety of people over a long period of time. Just as one example, Google Inc. has contributed many container-related technologies to the Linux kernel. Without these, and other contributions, we wouldn't have modern containers today.

Some of the major technologies that enabled the massive growth of containers in recent years include kernel namespaces, control groups, and of course Docker. To re-emphasize what was said earlier - the modern container ecosystem is deeply indebted to the many individuals and organizations that laid the strong foundations that we now build on!

Despite all of this, containers remained outside of the reach of most organizations. It wasn't until Docker came along that containers were effectively democratized and accessible to the masses.

> * There are many operating system virtualization technologies similar to containers that pre-date Docker and modern containers. Some even date
> back to System/360 on the Mainframe. BSD Jails and Solaris Zones are some other well known examples of Unix-type container technologies. However, in this section we are restricting our conversation and com-ments to *modern containers* that have been made popular by Docker.

Hello Docker!

We'll talk about Docker in a bit more detail in the next chapter. But for now it's enough to say that Docker was the magic that made Linux containers usable for mere mortals. Put another way, Docker, Inc. gave a the world a set of technologies and tools that made creating and working with containers simple!

Windows containers

Although containers came to the masses via Linux, Microsoft Corp. has worked extremely hard to bring Docker and container technologies to the Windows platform.

At the time of writing, Windows containers are available on the Windows Server 2016 platform. In achieving this, Microsoft has worked closely with Docker, Inc.

The core Windows technologies required to implement containers are collectively referred to as *Windows Containers*. The user-space tooling to work with Windows Containers is Docker. This makes the Docker experience on Windows almost exactly the same as Docker on Linux. This way developers and sysadmins familiar with the Docker toolset form the Linux platform will feel right at home using Windows containers.

Windows containers vs Linux containers

It's vital to understand from the start that Windows containers will only run on Windows servers, and Linux containers will only run on Linux servers.

This is because all containers running on a system access the kernel of the OS running on that system. Therefore, containers running natively on a Windows system have to access the Windows kernel. Linux containers (which run Linux applications inside of them) obviously cannot use the Windows kernel, and vice versa.

Chapter Summary

We used to live in a world where every time the business wanted a new application we had to but a brand new server for it. Then VMware came along and enabled IT departments to drive more value out of new and existing company IT assets. But as good as VMware and the VM model is, it's not perfect. Following the success of VMware and hypervisors came a newer more efficient and lightweight virtualization technology called containers. But containers were initially hard to implement and were only found in the data centers of web giants that had Linux kernel engineers on staff. Then along came Docker Inc. and suddenly container virtualization technologies were available to the masses.

Speaking of Docker… let's go find who, what, and why Docker is!

Lesson 2: Docker

No book or conversation about containers is complete without talking about Docker. But when somebody says “Docker” they can be referring to any of at least three things:

1. Docker, Inc. the company
2. Docker the container runtime and orchestration technology
3. Docker the open source project

If you’re going to *make it* in the container world, you’ll need to know a bit about all three.

Docker - The TLDR

We’re about to get into a bit of detail on each, but before we do that here’s the TLDR: Docker is software that runs on Linux and Windows. It creates,

manages and orchestrates containers. The software is developed in the open as part of the Docker open-source project on GitHub. Docker, Inc. is a company based out of San Francisco and is the overall maintainer of the open-source project.

Ok that's the quick version. Now we'll explore each in a bit more detail. We'll also talk a bit about the container ecosystem, and we'll mention the Open Container Initiative (OCI).

Docker, Inc.

Docker, Inc. is the San Francisco based technology startup founded by French-born American developer and entrepreneur Solomon Hykes.

Figure 2.1 Docker, Inc. logo.

Interestingly, Docker, Inc. started its life as a platform as a service (PaaS) provider called *dotCloud*. Behind the scenes, the dotCloud platform leveraged Linux contain-ers. To help them create and manage these containers they built an internal tool that they nick-named "Docker". And that's how Docker was born!

In 2013 the dotCloud PaaS business was struggling and the company was in need of a new lease of life. To help with this they hired Ben Golub as new CEO, rebranded the company as "Docker, Inc.", got rid of the dotCloud PaaS platform, and started a new journey with a mission to bring to Docker and containers to the world.

Today Docker, Inc. is widely recognized as an innovative technology company with a market valuation said to be in the region of $1BN. At the time of writing, it has raised over $180M via 6 rounds of funding from

some of the biggest names in Silicon Valley venture capital. Almost all of this funding was raised after the company pivoted to become *Docker, Inc.*

Since becoming Docker, Inc. they've made several small acquisitions, for undisclosed fees, to help grow their portfolio of products and services.

At the time of writing, Docker, Inc. has somewhere in the region of 200-300 employees and holds an annual conference called Dockercon. The goal of Dockercon is to bring together the growing container ecosystem and drive the adoption of Docker and container technologies.
Throughout this book we'll use the term "Docker, Inc." when referring to Docker the company. All other uses of the term "Docker" will refer to the technology or the open-source project.
Note: The word "Docker" comes from a British colloquialism meaning dock worker - somebody who loads and unloads ships.

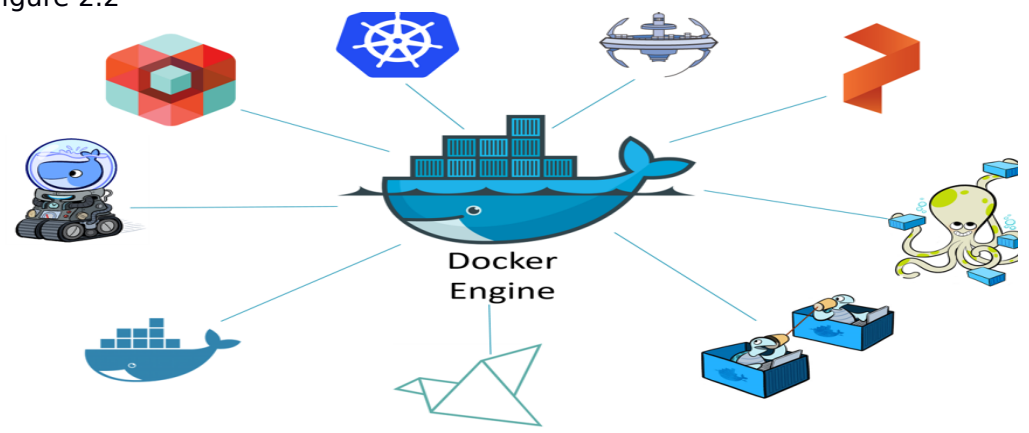
The Docker runtime and orchestration engine

When most *technologists* talk about Docker, they're referring to the *Docker Engine*.

The *Docker Engine* is the infrastructure plumbing software that runs and orchestrates containers. If you're a VMware admin, you can think of it as being similar to ESXi. In the same way that ESXi is the core hypervisor technology that runs virtual machines, the Docker Engine is the core container runtime that runs containers.

All other Docker, Inc. and 3rd party products plug into the Docker Engine and build around it. Figure 2.2 shows the Docker Engine at the center. All of the other products in the diagram build on top of the Engine and leverage it's core capabilities.

Figure 2.2



The Docker Engine can be downloaded from the Docker website or built from source from GitHub. It's available on Linux and Windows, with open-source and commer-cially supported offerings. At the time of writing there's a new major release of the Docker Engine approximately every three months (https://github.com/docker/docker/wiki).

The Docker open-source project

The term "Docker" can also refer to the open-source *Docker project*.

The Docker project is hosted on GitHub and you can see a list of the sub-projects and tools included in the Docker repository at https://github.com/docker. The core *Docker Engine* project is located at https://github.com/docker/docker.

As an open -source project, the source code is publically available and you are free to download it, contribute to it, tweak it, and use it, as long as you adhere to the terms of the Apache License 2.0[4].

If you take the time to look at the project's commit history you'll see the who's-who of infrastructure technology including; RedHat, Microsoft, IBM, Cisco, and HPE. You'll also see the names of individuals not associated with large corporations.

Most of the project and its tools are written in *Go* - the relatively new system-level programming language from Google also known as *Golang*.
If you code in Go you're in a great position to contribute to the project!

A nice side effect of Docker being an open-source project is the fact that so much of it is developed and designed in the open. This does away with a lot

of the *old ways* where code was proprietary and locked behind closed doors. It also means that release cycles are published and worked on in the open. No more uncertain release cycles that are kept a secret and then pre-announced months-in-advance to ridiculous pomp and ceremony. The Docker project doesn't work like that. Most things are done in the open for all to see and all to contribute to.

The Docker project is huge and gaining momentum. It has thousands of GitHub pull requests, tens of thousands of Dockerized projects, not to mention the billions of image pulls form Docker Hub. The project literally is taking the industry by storm!

Be under no illusions, Docker is being used!

The container ecosystem

One of the philosophies at Docker, Inc. is often referred to as *Batteries included but removable*.https://github.com/docker/docker/blob/master/LICENSE

This is a way of saying you can swap out a lot of the native Docker *stuff* and replace it with *stuff* from 3rd party ecosystem partners. A good example of this is the networking stack. The core Docker product ships with built-in networking. But the networking stack is pluggable meaning you can rip out the native Docker networking stack and replace it with something else form a 3rd party.

In the early days it was common for 3rd party plugins to be better than the native offerings that shipped with Docker. However, this presented some business model challenges for Docker, Inc. After all, Docker, Inc. has to turn a profit at some point to be a viable long-term business. As a result, the batteries that are included are getting better and better. This is something that is causing ripples across the wider ecosystem, which it seems may have expected Docker, Inc. to produce mediocre products and leave the door wide open for them to swoop in and plunder the spoils.

If that was once true, it's not any more. To cut a long story short, the native Docker batteries are still removable, there's just less and less reason to want to remove them.

Despite this, the container ecosystem is flourishing with a healthy balance of co-operation and competition. You'll often hear people use terms like *co-*

opetition (a balance of co-operation and competition) and *frenemy* (a mix of a friend and an enemy) when talking about the container ecosystem. This is great! Healthy competition is the mother of innovation!

The Open Container Initiative

No discussion of Docker and the container ecosystem is complete without mention-ing the Open Container Initiative - OCI[5].

The OCI is a relatively new governance council responsible for Standardizing the most fundamental components of container infrastructure such as *image format* and *container runtime* (don't worry if these terms are new to you, we'll cover them in the book).

It's also true that no discussion of the OCI is complete without mentioning a bit of history. And as with all accounts of history, the version you get depends on who's doing the talking.

[5] https://www.opencontainers.org

From day one, use of Docker has grown like crazy. More and more people used it in more and more ways for more and more things. So it was inevitable that somebody was going to get frustrated. This is normal and healthy. The TLDR of this *history* is that a company called CoreOS[6] didn't like the way Docker did certain things. So they did something about it! They created a new open standard called appc[7] that defined things like image format and container runtime. They also created an implementation of the spec called rkt (pronounced "rocket").

This put the container ecosystem in an awkward position with two competing standards. For want of better terms, the Docker stuff was the *de facto* standard and runtime, whereas the stuff from CoreOS was more like the *de jure* standard.Getting back to the story though, this all threatened to fracture the ecosystem and present users and customers with a dilemma. While competition is usually a good thing, *competing standards* is not. They cause confusion and slowdown adoption. Not good for anybody. With this in mind, everybody did their best to act like adults and came together to form the OCI - a lightweight agile council to govern container standards. At the time of writing, the OCI has published two specifications (standards) -

- An image spec[8]
- A runtime spec[9]

An analogy that's often used when referring to these two standards is rail tracks. These two standards are like agreeing on standard sizes and properties of rail tracks. Leaving everyone else free to build better trains, better carriages, better signaling systems, better stations… all safe in the knowledge that they'll work on the standardized tracks. Nobody wants two competing standards for rail track sizes!

It's fair to say that the two OCI specifications have had a major impact on the architecture and design of the core Docker Engine. As of Docker 1.11, the Docker Engine architecture conforms to the OCI runtime spec.

[6]https://coreos.com [7]https://github.com/appc/spec/
[8]https://github.com/opencontainers/image-spec
[9]https://github.com/opencontainers/runtime-spec

So far, the OCI has achieved good things and gone some way to bringing the ecosystem together. However, standards always slow innovation! Especially with new technologies that are developing at close to warp speed. This has resulted in some ~~raging arguments~~ passionate discussions in the container community. In the opinion of your author, this is a good thing! The container industry is changing the world and it's normal for the people at the vanguard to be passionate and opinionated. Expect more *passionate discussions* about standards and innovation!

The OCI is organized under the auspices of the Linux Foundation and both Docker, Inc. and CoreOS, Inc. are major contributors.

Lesson 3: Installing Docker

There are loads of ways and places to install Docker. There's Windows, there's Mac, and there's obviously Linux. But there's also in the cloud, on premises, on your laptop. Not to mention manual installs, scripted installs, wizard-based installs. There literally are loads of ways and places to install Docker!

But don't let that scare you! They're all pretty easy.

In this chapter we'll cover some of the most important installs:

- Desktop installs
  - Docker for Windows
  - Docker for Mac
- Server installs
  - Linux

We'll add a Windows Server 2016 installation method after Windows Server 2016 has gone G.A. At the time of writing, the installation method for Windows Server 2016 TP5 is in a state of flux and not stable enough to be included here.

Docker for Windows

The first thing to note is that *Docker for Windows* is a packaged product from Docker, Inc. It spins up a single-engine Docker environment on a 64-bit Windows 10 desktop or laptop.

This means we're not about to show you how to manually hack an installation of Docker onto a Windows desktop or laptop. No! In this Section we'll look at how to install the product from Docker, Inc. called "Docker for Windows". And it's insanely simple!

But a word of caution! *Docker for Windows* is only intended for test and dev work. You don't want to run your production estate on it! Remember, it's only going to install a single engine. That's another way of saying it's only going to install one copy of Docker. You might also find that some of the latest Docker features aren't always available straight away in *Docker for Windows*. This is because Docker, Inc. are taking a *stability first, features second* approach with the product. All of this adds up to a quick and easy setup, but one that is not for production workloads.

Enough waffle. Let's see how to install *Docker for Windows*.
First up, pre-requisites. *Docker for Windows* requires:

- Windows 10 Pro | Enterprise | Education
- Must be 64-bit

- The Hyper-V and Containers features must be enabled in Windows
- Hardware virtualization support must be enabled in your system's BIOS

Enabling hardware virtualization support in your BIOS varies between machine types. Most modern machines have the settings enabled by default, so we won't go into detail here. But the process is usually something like this: reboot your machine

> hold down the special BIOS key (this is usually something like Del, F12, or Insert)

> locate the hardware virtualization support settings in your BIOS (Intel VT-x or AMD-V) > enable the settings > Save & Exit.

> WARNING: Take great care when modifying settings in your BIOS. Making the wrong change can prevent your machine from booting.

The first thing to do in Windows 10 is make sure the Hyper-V and Containers features are installed and enabled.

1. Right-click the Windows Start button and choose Programs and Features.

2. Click Turn Windows features on or off.
3. Check the Hyper-V and Containers checkboxes and click OK.

This will install and enable the Hyper-V and Containers features. Your system may require a restart.

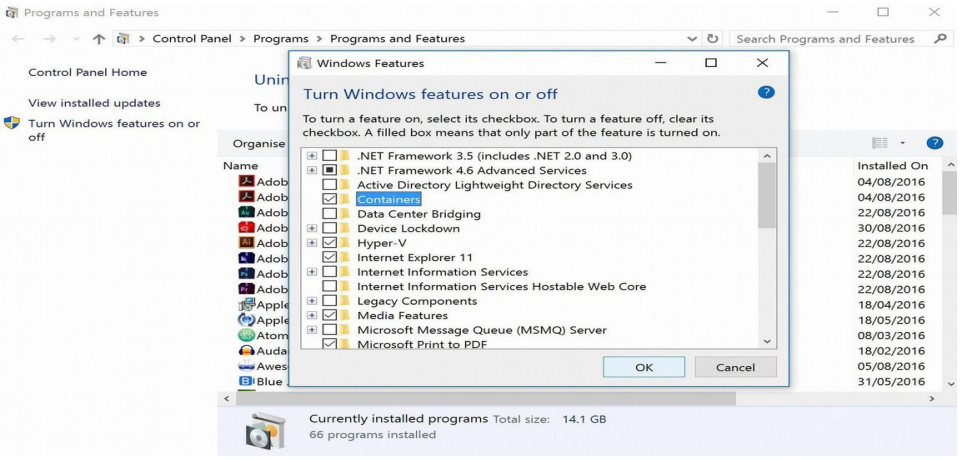


Figure 3.1

The *Containers* feature is only available if you are running the summer 2016 Windows 10 Anniversary Update (build 14393).

Once you've installed the Hyper-V and Containers features and restarted you machine, it's time to install *Docker for Windows*.

1. Head over to www.docker.com and click Get Docker from the top of the homepage.
2. Click the Learn More button under the WINDOWS section.
3. Click Download Docker for Windows to download the InstallDocker.msi package to your default downloads directory.
4. Locate and launch the InstallDocker.msi package that you just downloaded.

Step through the installation wizard and provide local administrator credentials to complete the installation. Docker will automatically start as a system service and a Moby Dock whale icon will appear in the Windows notifications tray.

Congratulations! You have installed *Docker for Windows*.

Now that *Docker for Windows* is installed you can open a command prompt or PowerShell window and run some Docker commands. Try the following commands:

```
C:\Users\visualpath> docker version
Client:
 Version:      1.12.1
 API version: 1.24
 Go version:  go1.6.3
 Git commit:  23cf638
 Built:       Thu Aug 18 17:32:24 2016
 OS/Arch:     windows/amd64
 Experimental: true
Server:
 Version:      1.12.1
 API version: 1.24
 Go version:  go1.6.3
 Git commit:  23cf638
 Built:       Thu Aug 18 17:32:24 2016
 OS/Arch:     linux/amd64
 Experimental: true
```

Notice that the OS/Arch: for the Server component is showing as linux/amd64 in the output above. This is because the default installation currently installs the Docker daemon inside of a lightweight Linux Hyper-V VM. In this default scenario you will only be able to run Linux containers on your *Docker for Windows* install.

If you want to run *native Windows containers* you can right click the Docker whale icon in the Windows notifications tray and select the option to Switch to Windows containers.... You may get the following alert if you have not enabled the Windows Containers feature.

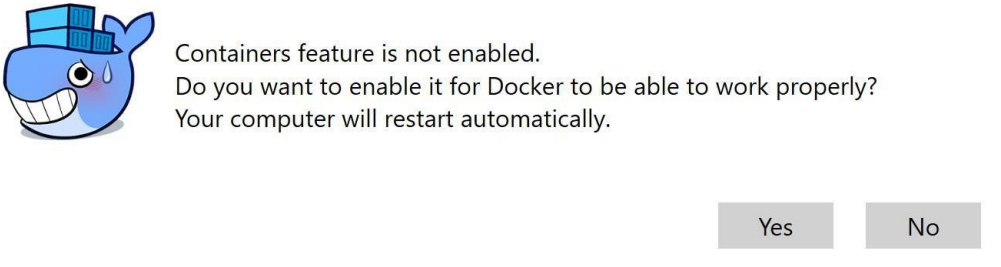


Figure 3.2

If you already have the Windows Containers feature enabled it will only take a few seconds to make the switch. Once the switch has been made the output to the docker version command will look like this.

```
C:\Users\visualpath> docker version
Client:
 Version:      1.12.1
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   23cf638
 Built:        Thu Aug 18 17:32:24 2016
 OS/Arch:      windows/amd64
 Experimental: true

Server:

 Version:      1.13.0-dev
 API version:  1.25
 Go version:   go1.7.1
 Git commit:   c2decbe
 Built:        Tue Sep 13 15:12:54 2016
 OS/Arch:      windows/amd64
```

Notice that the Server version is now also showing as windows/amd64. This means the daemon is now running natively on the Windows kernel and will therefore only run Windows containers.

Also note that the system above is running the *experimental* version of Docker (Experimental: true). *Docker for Windows* has stable and an

experimental channel. You can switch between the two, but you should check the Docker website for restrictions and implications before doing so.

As shown below, other regular Docker commands work as normal.

```
C:\Users\visualpath>docker
info Containers: 0 Running: 0
 Paused: 0
 Stopped:0
  Images:0
  Images:0
Server Version: 1.13.0-
dev Storage Driver:
windowsfilter Windows:
Logging Driver: json-file
Plugins:
 Volume: local
Network: nat null
overlay <Snip>
Registry: https://index.docker.io/v1/
Experimental: true
Insecure Registries:
 127.0.0.0/8
```

Docker for Windows includes the Docker Engine (client and daemon), Docker Compose, and Docker Machine. Use the following commands to verify that each was successfully installed and which versions of each you have:

C:\Users\visualpath> docker --version
Docker version 1.12.1, build 23cf638, experimental

C:\Users\visualpath> docker-compose

--version docker-compose version 1.8.0,

build d988a55

C:\Users\visualpath> docker-machine

--version docker-machine version 0.8.1,

build 41b3b25

Docker for Mac

The first thing to note about *Docker for Mac* is that it's a packaged product from Docker, Inc. So relax, you don't need to be a kernel engineer, and we're not about to walk through a complex hack for getting Docker onto your Mac. We'll walk you through the process of installing *Docker for Mac* on your Mac desktop or laptop, and it's ridiculously easy.

So what is *Docker for Mac?*

Docker for Mac is packaged product that allows you to easily get a small single-engine Docker environment up and running locally on your Mac. If you've heard of boot2docker then *Docker for Mac* is what you always wished *boot2docker* was - it's smooth, simple and stable. But *Docker for Mac* is only intended for test and dev work. You shouldn't think of it as a production platform to run your business from. No! *Docker for Mac* is all about getting a small working installation of Docker up and running on your Mac in the simplest way possible so that you can test and develop containerized applications on your Mac.

It's also worth noting that *Docker for Mac* will not give you the Docker Engine running natively on the Mac OS Darwin kernel. Behind the scenes it runs the Docker Engine inside of a lightweight Linux VM. It then seamlessly exposes that Docker Engine and API to your Mac environment. But it does it all in a way that the mystery and magic that pulls it all together is hidden away behind the scenes. All you need to know is that you can open a terminal on your Mac and use the regular Docker commands to hit the Docker API. Although this seamlessly works on your Mac.... Its obviously Docker on Linux under the hood, so it's only going work with Linux based Docker containers. This is good though, as this is where most of the container action is.

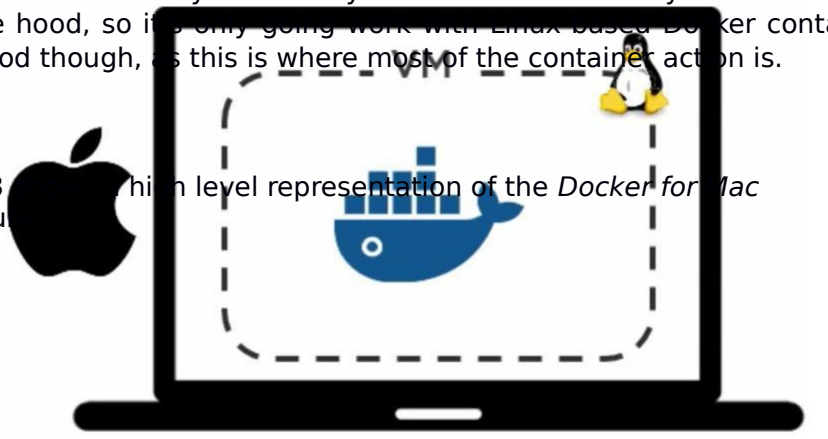


Figure 3.3 – A high level representation of the *Docker for Mac* architecture

Figure 3.3

Note: For the curious reader, *Docker for Mac* leverages HyperKit[10] to implement a super lightweight hypervisor. HyperKit in turn is based off the xhive hypervisor[11]. *Docker for Mac* also leverages features from DataKit[12] and runs a highly tuned Linux distro called Moby that is based off of Alpine Linux[13].

Let's get *Docker for Mac* installed.

1. Point your browser to www.docker.com
2. Click the Get Docker link near the top of the Docker homepage.
3. Click the Learn More button under the MAC section and then click Download Docker for Mac. This will download the Docker.dmg installation package to your default downloads directory.
4. Launch the Docker.dmg file that you downloaded in the previous step. You will be asked to drag and drop the Moby Dock whale image into the Applications folder.

[10] https://github.com/docker/hyperkit
[11] https://github.com/mist64/xhyve
[12] https://github.com/docker/datakit
[13] https://alpinelinux.org/
https://github.com/alpinelinux

5. Open your Applications folder (it may open automatically) and double-click the Docker application icon to Start it. You may be asked to confirm the action because the application was downloaded form the internet.

6. Enter your password so that the installer can create components, such as networking, that require elevated privileges.
7. The Docker daemon will now start.

   An animated whale icon will appear in the status bar at the top of your screen, and the animation will stop when the daemon has successfully started. Once the daemon has started you can click the whale icon and perform basic actions such as restarting the daemon, checking for updates, and opening the UI.

Now that *Docker for Mac* is installed you can open a terminal window and run some regular Docker commands. Try the commands listed below.

```
$ docker version
Client:
 Version:      1.12.0-rc3
 API version:  1.24
 Go version:   go1.6.2
 Git commit:   91e29e8
 Built:        Sat Jul 2 00:09:24 2016
 OS/Arch:      darwin/amd64
 Experimental: true
Server:

 Version:      1.12.0-rc3
 API version:  1.24
 Go version:   go1.6.2
 Git commit:   876f3a7
 Built:        Tue Jul 5 02:20:13 2016
 OS/Arch:      linux/amd64
 Experimental: true
```

Notice in the output above that the OS/Arch: for the Server component is showing as linux/amd64. This is because the server portion of the Docker Engine (a.k.a. the “daemon”) is running inside of the Linux VM we mentioned earlier. The Client component is a native Mac application and runs directly on the Mac OS Darwin kernel (OS/Arch: darwin/amd64).

Also note that the system is running the experimental version (Experimental: true) of Docker. *Docker for Mac* has stable and experimental channels. You can switch between channels, but you should check the Docker website for restrictions and implications before doing so.

Run some more Docker commands.
$ docker info

Containers:0

Running: 0

Paused: 0

Stopped: 0

Images: 0

Server Version:
1.12.0-rc3 <Snip>

Registry: https://index.docker.io/v1/

Experimental: true

Insecure Registries:

127.0.0.0/8

Docker for Mac installs the Docker Engine (client and daemon), Docker Compose, and Docker machine. The following three commands show you how to verify that all of these components installed successfully and find out which versions you have.

$ docker --version
Docker version 1.12.0-rc3, build 876f3a7, experimental

$ docker-compose --version
docker-compose version 1.8.0, build d988a55

$ docker-machine --version
docker-machine version 0.8.1, build 41b3b25

Installing Docker on Linux

Let’s look at how to install Docker on Linux.

This is the most common installation in production environments and is surprisingly easy. The most common difficulty is the slight variations between Linux distros such as Ubuntu vs CentOS. The example we’ll use in this section is based on Ubuntu Linux, but should work on upstream and downstream forks. It should also work on CentOS and it’s upstream and

downstream forks. It makes absolutely no difference if your Linux machine is a physical server in your own data center, on the other side of the planet in a public cloud, or a VM on your laptop. The only requirements are that the machine be running Linux and has access to https://get.docker.com.

The first thing you need to decide before you install Docker on Linux is which channel you wish to install. Docker currently has three channels:

- Stable (https://get.docker.com/)
- Experimental (https://experimental.docker.com/)
- Test (https://test.docker.com/)

In the examples below we'll use the wget command to call the script that installs the stable channel. If you want to install a different channel just replace
https://get.docker.com with the relevant channel from the list above.

1. Open a new shell on your Linux machine.
2. Use wget to retrieve the Docker install script from https://get.docker.com and pipe it through your shell.

   ```
   $ wget -qO- https://get.docker.com/ | sh


   modprobe: FATAL: Module aufs not found in directory
   /lib/modules/4.4.0-36-ge\ neric
   + sh -c 'sleep 3; yum -y -q install
   docker-engine' <Snip>
   If you would like to use Docker as a non-root user, you should now
   consider adding your user to the "docker" group with something
   like:


   sudo usermod -aG docker your-user


   Remember that you will have to log out and back in for this to take
   effect!
   ```

3. It's a good best practice to only use non-root users when working with the Docker Engine. To do this you need to add your non-root users to the local docker Unix group on your Linux machine. The

commands below show how to add the npoulton user to the docker group and verify that the operation succeeded.

$ sudo usermod -aG docker vagrant
$ cat /etc/group | grep docker docker:x:999:vagrant

If you are already logged in as the user that you just added to the docker group, you will need to log out and log back in for the group membership to take effect.

Congratulations! Docker is now installed on your Linux machine. Run the following commands to verify your installation.

$ docker --version
Docker version 1.12.1, build 23cf638

$ docker info

Containers:

0 Running: 0

Paused: 0

Stopped: 0

Images: 0

<Snip>

Kernel Version: 4.4.0-36-generic

Operating System: Ubuntu 16.04.1

LTS OSType: linux

Architecture:
x86_64 CPUs: 1

Total Memory: 990.7

MiB Name: ip-172-31-

41-77

ID:

QHFV:6HK7:VNLZ:RIKE:JWL6:BTIX:GC3V:RAVR:6AO5:RAMT:EJ

CI:PUA7 Docker Root Dir: /var/lib/docker

Debug Mode (client):

false Debug Mode

(server): false

Registry: https://index.docker.io/v1/

WARNING: No swap limit support

Insecure Registries:

127.0.0.0/8

If the process described above doesn't work for your Linux distro, you can go to the Docker Docs[14] website and click on the link relating to your distro. This will take you to the official
Docker installation instructions which are usually kept up to date. Be warned though, the instructions on the Docker website tend use the package manager and require a lot more steps than the procedure we used above. In fact, if you open a web browser to https://get.docker.com you will see that it's a shell script that does all of the hard work of installation for you.

> Warning: If you install Docker from a source other than the official Docker repositories, you may end up with a forked version of Docker.

https://docs.docker.com/engine/installation/linux/

> This is because some vendors and distros choose to fork the Docker project and develop their own slightly customized versions. You need to be aware of things like this if you are installing from custom repositories as you could unwittingly end up in a situation where you are running a fork that has diverged from the official Docker project. This isn't a problem as long as this is what you intend to do. If it is not what you intend, it can lead to situations where modifications and fixes your vendor makes do not make it back upstream in to the official Docker project. In these situations you will not be able to get commercial support for your installation from Docker, Inc. or it's authorized service partners.

Chapter Summary

In this chapter you saw how to install docker on Windows 10, Mac OS X, and Linux. Now that you know how to install Docker you are ready to start working with images and containers.

Lesson 4: The big picture

In the next few chapters we're going to get into the details of things like images, containers, and orchestration. But before we do that, I think it's a good idea to show you the big picture first.
In this chapter we'll download an image, start a new container, log in to the new container, run a command inside of it, and then destroy it. This will give you a good idea of what Docker is all about and how some of the major components fit together.

But don't worry if some of the stuff we do here is totally new to you. We're not trying to make you experts by the end of this chapter. All we're doing here is giving you a *feel* of things - setting you up so that when we get into the details in later chapters, you have an idea of how the pieces fit together.

All you need to follow along with the exercises in this chapter is a single Docker host. This can be any of the options we just installed in the previous chapter, though if you are using *Docker for Windows* you should be running it in "Linux Container" mode. It doesn't matter if this Docker host is a VM on your laptop, an instance in the public cloud, or bare metal server in your data center. All it needs, is to be running Docker with a connection to the internet.

Engine check

When you install Docker you get two major components:

- the Docker client
- the Docker daemon (sometimes called server)

The daemon implements the Docker Remote API[15]. In a default Linux installation the client talks to the daemon via a local IPC/Unix socket at /var/run/docker.sock. You can test that the client and daemon are operating and can talk to each other with the docker version command.

[15] https://docs.docker.com/engine/reference/api/docker_remote_api/

```
$ docker version


Client:
 Version:      1.12.1
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   23cf638
 Built:        Thu Aug 18 05:33:38 2016
 OS/Arch:      linux/amd64
Server:

 Version:      1.12.1
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   23cf638
 Built:        Thu Aug 18 05:33:38 2016
 OS/Arch:      linux/amd64
```

> As long as you get a response back from the Client and Server components you should be good to go. If you get an error response form the Server component, try the command again with sudo in front of it: sudo docker version.
> If it works with sudo chapter with sudo. you will need to prefix the remainder of the commands in this chapter with sudo.

Images

Now let’s look at *images*.

Right now, the best way to think of a Docker image is as an object that contains an operating system and an application. It’s not massively different from a virtual machine template. A virtual machine template is essentially a stopped virtual machine. In the Docker world, an image is effectively a stopped container.

Run the docker images command on your Docker host.

```
$ docker images
```

REPOSITORY TAG IMAGE ID CREATED SIZE
If you are working from a freshly installed Docker host it will have no images and will look like the output above.

Getting images onto your Docker host is called “pulling”. Pull the ubuntu:latest image to your Docker host with the command below.

```
$ docker pull ubuntu:latest
latest: Pulling from library/ubuntu

952132ac251a: Pull complete
82659f8f1b76: Pull
complete c19118ca682d:
Pull complete
8296858250fe: Pull complete
24e0251a0e2c: Pull complete
Digest:
sha256:f4691c96e6bbaa99d...a2128ae95a60369c506dd6e
6f6ab Status: Downloaded newer image for ubuntu:latest
```

Run the docker images command again to see the ubuntu:latest image you just pulled.

We’ll get into the details of where the image is stored and what’s inside of it in the next chapter. For now it’s enough to understand that it contains enough of an operating system (OS), as well as all the code to run whatever application it’s designed for. The ubuntu image that we’ve pulled has a stripped down version of the Ubuntu Linux OS including a few of the common Ubuntu utilities.

```
$ docker images
REPOSITORY  TAG     IMAGE ID      CREATED      SIZE
ubuntu      latest  bd3d4369aebc  11 days ago  126.6 MB
```

It’s worth noting as well that each image gets it’s own unique ID. When working with the image, as we will do in the next step, you can refer to it using either its ID or name.

Containers

Now that we have an image pulled locally on our Docker host, we can use the docker run command to launch a container from it.

```
$ docker run -it ubuntu:latest
/bin/bash root@6dc20d508db0:/#
```

Look closely at the output form the command above. You should notice that your shell prompt has changed. This is because your shell is now attached to the shell of the new container - you are literally inside of the new container!

Let's examine that docker run command. docker run tells the Docker daemon to start a new container. The -it flags tell the daemon to make the container interactive and to attach our current shell to the shell of the container (we'll get more specific about this in the chapter on containers). Next, the command tells Docker that we want the container to be based on the ubuntu:latest image, and we tell it to run the /bin/bash process inside the container.
Run the following ps command from inside of the container to list all running processes.

```
root@6dc20d508db0:/# ps -elf
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root         1     0  0  80   0 - 4560 wait   13:38 ?        00:00:00 /bin/bash
0 R root         9     1  0  80   0 - 8606 -      13:38 ?        00:00:00 ps -elf
```

As you can see from the output of the ps command, there are only two processes running inside of the container:

PID 1. This is the/bin/bash process that we told the container to run with the docker run command.
PID 9. This is the ps -elf process that we ran to list the running processes.

The presence of the ps -elf process in the output above could be a bit misleading as it is a short-lived process that dies as soon as the ps command exits. This means that the only long-running process inside of the container is the /bin/bash process.

Press Ctrl-PQ to exit the container. This will land you back in the shell of your Docker host. You can verify this by looking at your shell prompt.

Now that you are back at the shell prompt of you Docker host, run the ps -elf command again.

```
$ ps -elf
F S UID          PID    PPID  NI ADDR SZ WCHAN TIME CMD
4 S root           1       0   0    - 9407 -     00:00:03 /sbin/init
1 S root           2       0   0    -    0 -     00:00:00 [kthreadd]
1 S root           3       2   0    -    0 -     00:00:00 [ksoftirqd/0]
1 S root           5       2 -20    -    0 -     00:00:00 [kworker/0:0H]
1 S root           7       2   0    -    0 -     00:00:00 [rcu_sched]
<Snip>
0 R ubuntu     22783   22475   0    - 9021 -     00:00:00 ps -elf
```

Notice how many more processes are running on your Docker host compared to the single long-running process inside of the container.

In a previous step you pressed Ctrl-PQ to exit your shell from the container. Doing this from inside of a container will exit you form the container without killing it. You can see all of the running containers on your system using the docker ps command.

```
$ docker ps
CNTNR ID IMAGE          COMMAND    CREATED     STATUS      NAMES
0b3...41 ubuntu:latest  /bin/bash  7 mins ago  Up 7 mins   tiny_poincare
```

The output above shows a single running container. This is the container that you created earlier. The presence of your container in this output proves that it's still running. You can also see that it was created 7 minutes ago and has been running for 7 minutes.

Attaching to running containers

You can attach your shell to running containers with the docker exec command. As the container from the previous steps is still running let's connect back to it.

Note: The example below references a container called "tiny_poincare". The name of your container will be different, so remember to substitute "tiny_poincare" with the name or ID of the container running on your Docker host.

```
$ docker exec -it tiny_poincare bash
root@6dc20d508db0:/#
```

Notice that your shell prompt has changed again. You are back inside the container.

The format of the docker exec command is: docker exec -options <container-name or container-id> <command>. In our example we used the -it options to attach our shell to the container's shell. We referenced the container by name and told it to run the bash shell.

Exit the container again by pressing Ctrl-PQ.

Your shell prompt should be back to your Docker host.

Run the docker ps command again to verify that your container is still running.

```
$ docker ps
CNTNR ID IMAGE          COMMAND CREATED    STATUS     NAMES
0b3...41 ubuntu:latest  /bin/bash 9 mins ago Up 9 mins tiny_poincare
```

Stop the container and kill it using the docker stop and docker rm commands.

```
$ docker stop
tiny_poincare
tiny_poincare


$ docker rm
tiny_poincare
tiny_poincare
```

Verify that the container was successfully deleted by running another docker ps command.

```
$ docker ps
CONTAINER ID  IMAGE     COMMAND  CREATED   STATUS    PORTS     NAMES
```

Congratulations! You've downloaded a Docker image, launched a container from that image, executed a command inside of the container (ps -elf) and then stopped and deleted the container. This *big picture* view should help you with the up-coming chapters where we will dig deeper into images and containers.

Lesson 5: Images

In this chapter we'll dive a bit deeper into Docker images. The aim of the game here is to give you a solid working understanding of what Docker images are and how to work with them.

As this is our first chapter in the Technical section of the book, we're going to employ the three-tiered approach where we split the chapter into three sections:

- The TLDR: Two or three quick paragraphs that you can read while standing in line for a coffee)
- The deep dive: The really long bit where we get into the detail
- The commands: A quick list of the commands we learned

Docker images - The TLDR

Docker images are a lot like VM templates. A VM template is like a stopped VM, whereas a Docker image is like a stopped container.

You start out by pulling images from an image registry such as Docker Hub[16]. The *pull* operation downloads the image to your local Docker host where you can use it to start one or more Docker containers.

Images are made up of multiple layers that get stacked on top of each other and represented as a single object. Within the image is a cut-down operating system (OS) and all of the files required to run an application or service. Because containers are intended to be fast and lightweight, images tend to be quite small.

The most common commands used to work with Docker images are docker pull to *pull* images onto your local Docker host, docker images to view a list of the images already pulled to your Docker host, and docker rmi to delete images when you no longer need them.

http:dock.com
Congrats! You've now got half a clue what a Docker image is :-D Now it's time to dig a bit deeper.

Docker images - The deep dive

We've mentioned a couple of times already that container images are like stopped containers. In fact you can stop a container and create a new image from it. With this in mind, images are considered *build-time* constructs whereas containers are *run-time* constructs.

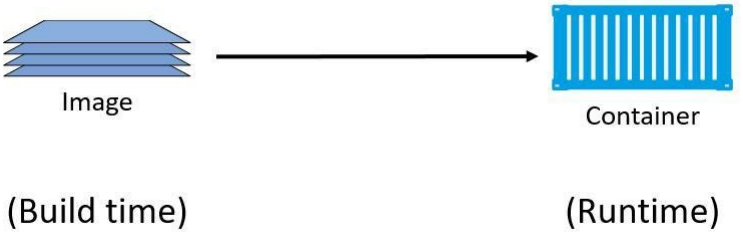


Figure 5.1

Images and containers

Figure 5.1 shows high level view of the relationship between images and containers. We use the docker run command to start one or more containers from a single image. However, once you've started a container form an image, the two constructs become dependent on each other and you cannot delete the image until the last container using it has been stopped and destroyed. Attempting to delete an image without stopping and destroying all containers using it will result in the following error:

```
$ docker rmi <image-name>
Error response from daemon: conflict: unable to remove repository reference
\ "<image-name>" (must force) - container <container-id> is using its
referenc\ ed image <image-id>
```

The whole purpose of a container is to run an application or service. This means that the image a container is created from must contain any OS and application

files required to run the container. However, containers are all about being fast and lightweight. This means that the images they're built from are usually small and stripped of all non-essential parts.

For example, Docker images tend not to ship with 6 different shells for you to choose from - they'll usually ship with a single minimalist shell. They also don't contain a kernel - all containers running on a Docker host share access to the Docker host's kernel. For these reasons we sometimes say images contain *just enough operating system*.

An extreme example of how small Docker images can be, might be the official Alpine Linux Docker image which is currently down at around 5MB. That's not a typo! It really is about 5 megabytes! However, a more typical example might be something like the official Ubuntu Docker image which is currently about 120-130MB.

Pulling images

A cleanly installed Docker host has no images in its local cache (/var/lib/docker/<storage-driver> on Linux hosts). You can verify this with the docker images command.

```
$ docker images
REPOSITORY TAG      IMAGE ID        CREATED         SIZE
```

The act of getting images onto a Docker host is called *pulling*. So if you want the latest Ubuntu image on your Docker host, you'd have to *pull* it. Use the commands below to *pull* the Alpine and Ubuntu images and then check their sizes.

If you haven't added your user account to the local docker Unix group, you may need to add sudo to the beginning of all of the following commands.

```
$ docker pull alpine:latest
latest: Pulling from library/alpine
e110a4a17941: Pull complete
Digest: sha256:3dcdb92d743...3626d99b889d0626de158f73a
Status: Downloaded newer image for alpine:latest
$
$ docker pull ubuntu:latest
latest: Pulling from
library/ubuntu 952132ac251a:
Pull complete 82659f8f1b76: Pull
complete c19118ca682d: Pull
complete
8296858250fe: Pull
complete 24e0251a0e2c:
Pull complete
Digest: sha256:f4691c96e6b...28ae95a60369c506dd6e6f6ab
Status: Downloaded newer image for ubuntu:latest
```

```
$ docker images
REPOSITOR
Y           TAG       IMAGE ID       CREATED        SIZE
                      bd3d4369aeb
ubuntu      latest    c              12 days ago    126.6 MB
                      4e38e38c8ce
alpine      latest    0              10 weeks ago   4.799 MB
```

As you can see, both images are now present on your Docker

host. Let’s look a bit closer at what we’ve just done.

We used the docker pull command to pull the images. As part of each command we had to specify which image to pull. So let’s take a minute to look at image naming. To do that we need a bit of background on how we store images.

Image registries

Docker images are stored in *image registries*. The most common image registry is Docker Hub. Other registries exist including 3rd party registries and secure on-premises registries, but Docker Hub is the default, and it's the one we'll use in this book.

Image registries contain multiple *image repositories*. Image repositories contain images. That might be a bit confusing, so Figure 5.2 shows a picture of an image registry containing 3 repositories, and each repository contains a few images.

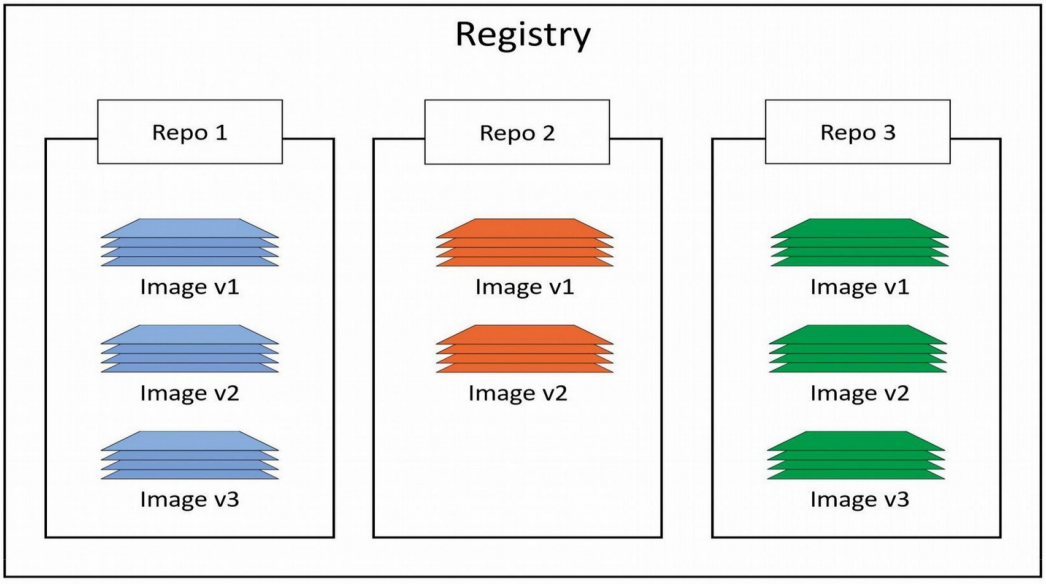


Figure 5.2

Docker Hub also has the concept of *official repositories* and *unofficial repositories*.

As the name suggests, *official repositories* contain images that have been vetted by Docker, Inc. This means they should contain up-to-date high quality secure code that is well documented and follows best practices.

Unofficial repositories are like the wild-west - they're controlled by none of the things on the previous list. That's not saying everything in *unofficial repositories* is bad! It's not! There's some great stuff in *unofficial repositories*. You just need to be very careful before trusting code from them. To be honest, you should always be careful when getting software from the internet - even images from *official repositories*.

Most of the popular operating systems and applications have their own *official repositories* on Docker Hub. They're easy to spot because they live at the top level of

the Docker Hub namespace. The list below contains a few of the *official repositories* and shows their URLs that exist at the top level of the Docker Hub namespace:

- nginx - https://hub.docker.com/_/nginx/
- busybox - https://hub.docker.com/_/busybox/
- redis - https://hub.docker.com/_/redis/
- mongo - https://hub.docker.com/_/mongo/

On the other hand, my own personal images live in the wild west of *unofficial repositories* and should not be trusted! Below are some examples of images in my repositories:

- visualpath/tu-demo - https://hub.docker.com/r/visualpath/tu-demo/
- visualpath/devops-docker-ci - https://hub.docker.com/r/visualpath/devops-docker-ci/

Not only are images in my repositories not vetted, not kept up-to-date, not secure, and not well documented… you should also notice that they don't live at the top level of the Docker Hub namespace. My repositories all live within a second level namespace called visualpath.

After all of that, we can finally look at how we address images on the Docker command line.

Image naming and tagging

Addressing images on the command line from *official repositories* is as simple as giving the repository name and tag separated by a colon ":". The format for docker pull when working with an image from an official repository is:

docker pull <repository>:<tag>

In our example form earlier we pulled an Alpine and an Ubuntu image with the following two commands:
docker pull alpine:latest and docker pull ubuntu:latest

These two commands pull the images tagged as “latest” from the “alpine” and “ubuntu” repositories.

The following examples show how to pull various different images from *official repositories*:
$ docker pull mongo:3.3.11
//This will pull the image tagged as `3.3.11` form the official `mongo` repo\ sitory.

$ docker pull redis:latest
//This will pull the image tagged as `latest` from the official `redis` repo\ sitory.

$ docker pull alpine
//This will pull the image tagged as `latest` from the official `alpine` rep\ ository.

A couple of points to note about the commands above. Firstly, if you do not specify an image tag after the repository name, Docker will assume you are referring to the image tagged as latest. Secondly, the latest tag doesn’t have any mystical powers! Just because an image is tagged as latest does not mean it is the most recent image in a repository! Moral of the story - take care when using the latest tag!

Pulling images from an *unofficial repository* is essentially the same - you just need to prepend the repository name with the Docker Hub username or organization name. The example below shows how to pull the v2 image from the tu-demo repository owned by a scary person whose Docker Hub account name is visualpath.

$ docker pull visualpath/tu-demo:v2
//This will pull the image tagged docker pull as `v2` from the `tu-demo` repository with\ in the namespace of my personal Docker Hub account.

If you want to pull images from 3rd party registries, you need to prepend the repos-itory name with the DNS name of the registry. For example, if the image in the ex-ample above was in the Google Container Registry (GCR) you'd need to add gcr.io

before the repository name as follows -
demo:v2.

You may need to have an account on 3rd party registries and be logged in before you can pull images from them.

Images with multiple tags

One final word about image tags… a single image can have as many tags as you want. This is because tags are arbitrary alpha-numeric values that are stored as metadata alongside the image. Let’s look at an example.

Pull all of the images in the repository below using the docker pull command with the -a flag. Then run docker images to look at the images pulled.

```
$ docker pull -a visualpath/tu-demo
latest: Pulling from visualpath/tu-
demo 237d5fcd25cf: Pull complete
a3ed95caeb02: Pull
complete <Snip>
Digest:
sha256:42e34e546cee61adb1...3a0c5b53f324a9e1c1aae4
51e9 v1: Pulling from visualpath/tu-demo
237d5fcd25cf:          Already
exists          a3ed95caeb02:
Already exists <Snip>
Digest:
sha256:9ccc0c67e5c5eaae4b...624c1d5c80f2c9623cbcc9b
59a v2: Pulling from visualpath/tu-demo
237d5fcd25cf:          Already
exists          a3ed95caeb02:
Already exists <Snip>
Digest: sha256:d3c0d8c9d5719d31b7...9fef58a7e038cf0ef2ba5eb74c
Status: Downloaded newer image for visualpath/tu-demo
$
$ docker images
REPOSITORY              TAG       IMAGE ID       CREATED       SIZE
                                                                 M
visualpath/tu-demo      v2        6ac2...ad   4 months ago   211.6B
                                              4                  M
visualpath/tu-demo      latest    9b91...29   months    ago  211.6B
                                              4                  M
visualpath/tu-demo      v1        9b91...29   months    ago  211.6B
```

A few things to notice about what just happened

First. The command pulled three tagged images from the repository: latest, v1, and v2.
Second. Look closely at the IMAGE ID column in the output of the docker images command. You'll see that there are only two unique image IDs. This means that even though three tags were pulled, only two images were actually downloaded. This is because two of the tags refer to the same image. Or put another way, one of the images has two tags. If you look closely you'll see that the v1 and latest tags have the same IMAGE ID. This means they're two tags of the same image.

This is a perfect example of the warning we issued earlier about the latest tag. As we can see,
the latest tag in this example refers to the same image as the v1 tag, not the v2 tag. This means
it's pointing to the older of the two images - not the newest. latest is an arbitrary tag and
is not guaranteed to point to the newest image in a repository.

Images and layers

All Docker images are made up of one or more read-only *layers* as shown below.

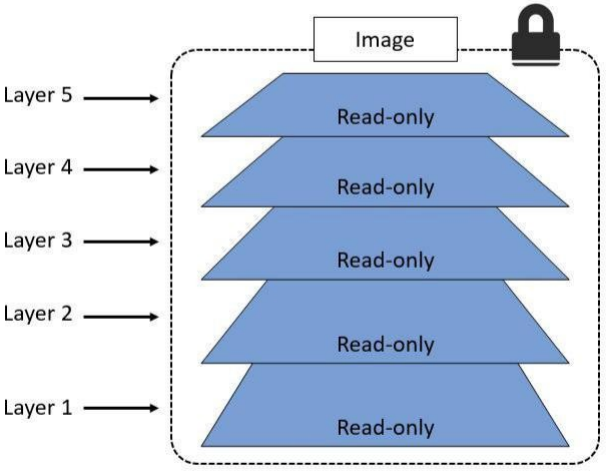


Figure 5.3

There are a few ways to see and inspect the layers that make up an image, and we've already seen one of them. Let's take a second look at the output of the docker

pull ubuntu:latest command from earlier:

```
$ docker pull ubuntu:latest
latest: Pulling from
library/ubuntu
952132ac251a: Pull complete
82659f8f1b76: Pull
complete c19118ca682d:
Pull complete
8296858250fe: Pull complete
24e0251a0e2c: Pull complete
Digest: sha256:f4691c96e6bbaa99d...28ae95a60369c506dd6e6f6ab
Status: Downloaded newer image for ubuntu:latest
```

Each line in the output above that ends with “Pull complete” represents a layer in the image that
was pulled. As we can see, this image has 5 layers. Figure 5.4 below shows this as a picture.

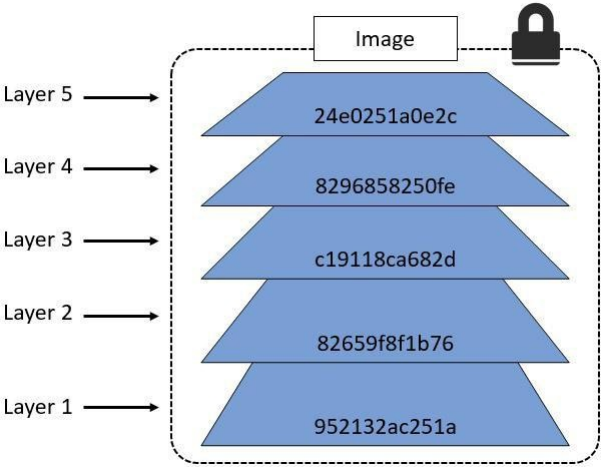


Figure 5.4

Another way to see the layers that make up an image is to inspect the

image with the docker inspect command. The example below inspects the same ubuntu:lates image.

```
$ docker inspect ubuntu:latest
[
    {
        "Id":
        "sha256:bd3d4369aebc4945be269859df0e15b1d32fefa2645f569903
        7d7d\
8c6b415a10",
        "RepoTags":
            [ "ubuntu:latest"

        <Snip>

        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:c8a75145fc...894129005e461a43875a0
                94b93412",
                "sha256:c6f2b330b6...7214ed6aac305dd03f70b9
                5cdc610",
                "sha256:055757a193...3a9565d78962c7f368d5a
                c5984998",
                "sha256:4837348061...12695f548406ea77feb50
                74e195e3",
                "sha256:0cad5e07ba...4bae4cfc66b376265e16c32a0aae9"
            ]
        }
    }
]
```

The trimmed output shows 5 layers again. Only this time they're shown using their SHA256 hashes. The point being, both commands show that the image has 5 layers.

> Note: The docker history command shows the build history of an image and is not a list of layers in the image. For example, some commands that appear in an image's build history do not result in image layers being created. Some of these commands (Dockerfile instructions) include "MAINTAINER", "ENV", "EXPOSE" and "CMD". Instead of these commands creating new image layers, their values are stored as part of the image's metadata.

Every layer in a Docker image gets it's own unique ID. This is a cryptographic hash
of the layer 's content. This means that the value of the crypto hash is determined by the contents of the image - changing the contents of the image changes its hash.
Using cryptographic content hashes improves security, avoids ID collisions that could occur if they were randomly generated, and gives us a way to guarantee data integrity after operations such as docker pull.

All Docker images start with a base layer, and as changes are made and new content is added, new layers are added on top. As an over-simplified example, you might create a brand new image based off of Ubuntu Linux 16.04. This would be your image's first layer. If you later add the Python package, this would be added as a second layer at the top of your image. If you then added a security patch, this would be added as a third layer at the top. Your image would now have three layers as shown in Figure 5.5 below.

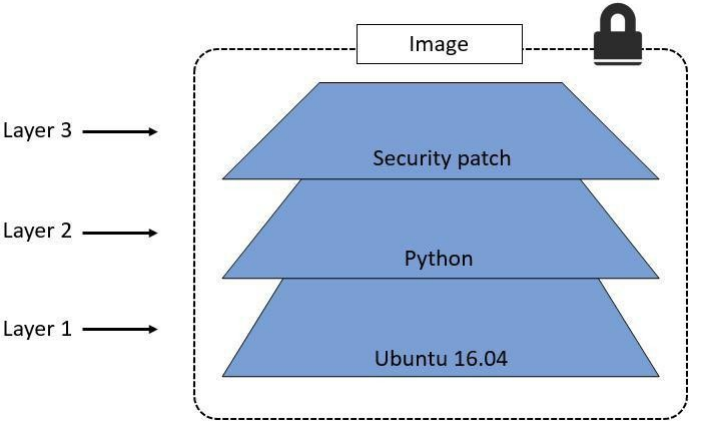


Figure 5.5

It's important to understand that as additional layers are added, the image becomes the combination of all of the layers. Take a simple example of two layers as shown in Figure 5.6. Each *layer* has 3 files, but the overall *image* has 6 files as it is the combination of both layers.

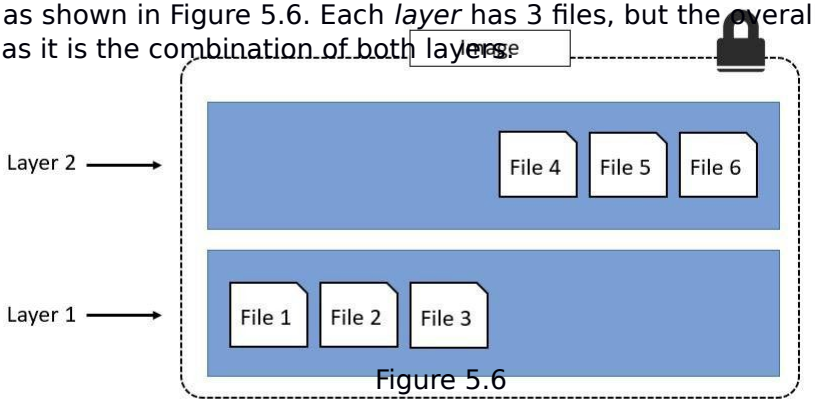


Figure 5.6

I've shown the image layers in Figure 5.6 in a slightly different way to previous figures. This is just to make showing files easier.

In the slightly more complex example of the three layered image in Figure 5.7, the overall image only ends up with 6 files. This is because file 7 in the top layer is an updated version of file 5 directly below. In this situation, the file in the higher layer obscures the file directly below it. This allows updated versions of files to be added as new layers to the image.

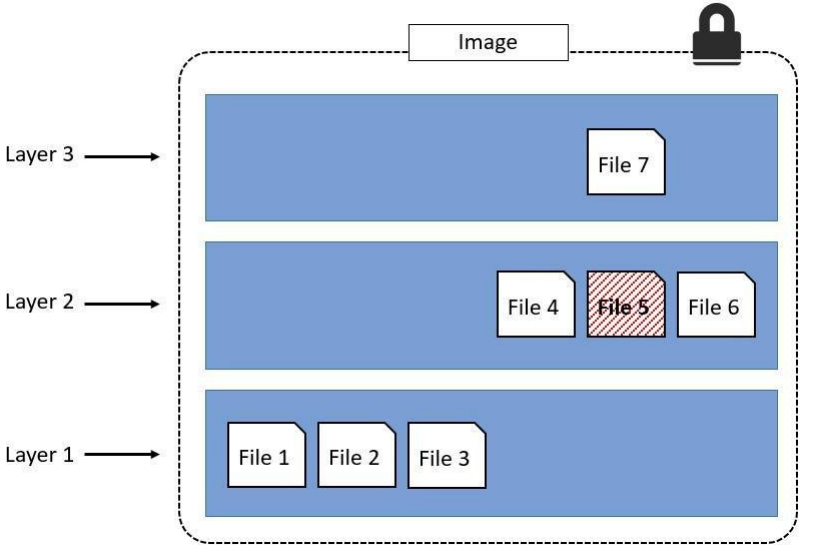


Figure 5.7

Sharing image layers

Multiple images can share layers. This leads to efficiencies in space and performance. Let's take a second look at the docker pull command with the -a flag that we ran a minute or two ago to pull all tagged images in the visualpath/tu-demo repository.

```
$ docker pull -a visualpath/tu-demo
latest: Pulling from visualpath/tu-
demo 237d5fcd25cf: Pull complete
a3ed95caeb02: Pull
complete <Snip>
Digest: sha256:42e34e546cee61adb100...a0c5b53f324a9e1c1aae451e9

v1: Pulling from visualpath/tu-
demo 237d5fcd25cf: Already exists
a3ed95caeb02: Already exists
<Snip>
Digest: sha256:9ccc0c67e5c5eaae4beb...24c1d5c80f2c9623cbcc9b59a

v2: Pulling from visualpath/tu-demo

237d5fcd25cf: Already exists
a3ed95caeb02: Already
exists <Snip>
eab5aaac65de: Pull complete
Digest: sha256:d3c0d8c9d5719d31b79c...fef58a7e038cf0ef2ba5eb74c
Status: Downloaded newer image for visualpath/tu-demo


$ docker images
REPOSITORY          TAG       IMAGE ID       CREATED         SIZE
visualpath/tu-demo  v2        6ac...ead      4 months ago    211.6 MB
visualpath/tu-demo  latest    9b9...e29      4 months ago    211.6 MB
visualpath/tu-demo  v1        9b9...e29      4 months ago    211.6 MB
```

Notice the lines ending in Already exists. This is because Docker is smart enough recognize when it's being asked to pull an image layer that it already has a copy of locally. In this example Docker pulled the image tagged as latest first. Then when it went to pull the v1 and v2 images it noticed that it already had some of the layers that make up those images. This happens because the three images in this repository are almost identical except for the top layer.

Docker on Linux supports many different filesystems and storage drivers. Each is free to implement image layering, copy-on-write behavior, and image layer sharing in its own way. However, the overall result and user experience is essentially the same.

Pulling images by digest

So far we've shown you how to pull images by tag, and this is by far the most common way. But it has a problem - tags are mutable! This means it's possible to accidentally tag an image with an incorrect tag. Sometimes it's even possible to tag an image with the same tag as an existing image. This is not good!

As an example, imagine that you've got an image called golftrack:1.5 and it has a known bug. You pull the image, apply a fix and push the updated image back to your repository
with the *same tag*. Take a second to understand what just happened there. You have an image called golftrack:1.5 that has a bug. That image is being used in your production environment. You pull the image and apply a fix. But then comes the mistake, you push the fixed image back to its repository with the same tag as the vulnerable image! How are you going to know which of your production systems are running the vulnerable image and which are running the patched image? They both have the same tag!

This is where *image digests* come to the rescue.

Docker 1.10 introduced a new content addressable storage model. As part of this new model all images now get cryptographic content hash. For the purposes of this discussion we'll refer to this hash as the *digest*. Because the digest is a hash of the contents of the image, it is not possible to change the contents of the image without the digest also changing. Put another way - digests are immutable. Clearly this avoids the problem we just talked about.

Every time you pull an image, the docker pull command will include the image's digest as part of the return code. You can also view the digests of images in your Docker host's local cache by adding the --digests flag to the docker images command. These are both shown in the following example.

```
$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
e110a4a17941: Pull complete
Digest:
sha256:3dcdb92d7432d56604d...6d99b889d0626de158f73a
Status: Downloaded newer image for alpine:latest
$
$ docker images --digests alpine
REPOSITOR
Y           TAG      DIGEST                IMAGE ID        CREATED        SIZE
                                           4e38e38c8ce
alpine      latest   sha256:3dcd...f73a    0               10 weeks ago   4.8 MB
```

The output above shows the digest for the alpine image as sha256:3dcdb92d7432...889d0626de

Now that we know the digest of the image, we can use it when pulling the image again. This will ensure that we get exactly the image we expect!

At the moment there is no native docker command or sub-command that will

retrieve the digest of an image from a remote registry such as Docker Hub.

This means the

only way to determine the digest of an image is to pull it by tag and then make a note of it's digest. This may change in the future.

The example below deletes the alpine:latest image from your Docker host and then shows how to pull it again using its digest instead of its tag.

```
$ docker rmi alpine:latest Untagged: alpine:latest

Untagged: alpine@sha256:3dcdb92d7432...313626d99b889d0626de158f73a
```

```
Deleted: sha256:4e38e38c8ce0b8d9...3b0bfe8cfa2321aec4bba

Deleted: sha256:4fe15f8d0ae69e16...b265cd2e328e15c6a869f


$ docker pull alpine@sha256:3dcdb92...b313626d99b889d0626de158f73a
sha256:3dcdb92d7432d...e158f73a: Pulling from library/alpine

e110a4a17941: Pull complete

Digest: sha256:3dcdb92d7432d56604...47b313626d99b889d0626de158f73a
Status: Downloaded newer image for
alpine@sha256:3dcd...b889d0626de158f73a
```

Deleting Images

When you no longer need an image you can delete it form your Docker host with the docker rmi command. rmi is short for remove image.

Delete the Alpine image pulled in the previous step with the docker rmi command. The example below addresses the image by its ID, this might be different on your system.

$ docker rmi 4e38e38c8ce0

Untagged: alpine:latest

Untagged: alpine@sha256:3dcdb92d7432d56..d99b889d0626de158f73a

Deleted: sha256:4e38e38c8ce0b8d90...3b0bfe8cfa2321aec4bba

Deleted: sha256:4fe15f8d0ae69e169...b265cd2e328e15c6a869f

If the image you are trying to delete is in use by a running container you will not be able to delete it. Stop and delete any containers before trying the remove operation again.

A handy shortcut for cleaning up a system and deleting all images on a Docker host is to run the docker rmi command and pass it a list of all image IDs on the system by calling docker images with the -q flag as shown below.$ docker rmi **$(**docker images -q**)** -f

To understand how this works, download a couple of images and then run docker images -q.

```
$ docker pull alpine Using default tag: latest

latest: Pulling from library/alpine e110a4a17941: Pull complete
Digest: sha256:3dcdb92d7432d5...3626d99b889d0626de158f73a

Status: Downloaded newer image for alpine:latest
```

```
$ docker pull ubuntu Using default tag: latest

latest: Pulling from library/ubuntu

952132ac251a: Pull complete

82659f8f1b76: Pull complete c19118ca682d: Pull complete 8296858250fe:
Pull complete 24e0251a0e2c: Pull complete
Digest: sha256:f4691c96e6bba...128ae95a60369c506dd6e6f6ab Status:
Downloaded newer image for ubuntu:latest


$ docker images -q bd3d4369aebc 4e38e38c8ce0
```

See how docker images -q returns a list containing just the image IDs of all images pulled locally on the system. Returning this list to docker rmi will therefore delete all images on the system as shown below.

```
$ docker rmi $(docker images -q) -f Untagged: ubuntu:latest

Untagged:   ubuntu@sha256:f4691c9...2128ae95a60369c506dd6e6f6ab

Deleted: sha256:bd3d4369aebc494...fa2645f5699037d7d8c6b415a10
Deleted: sha256:cd10a3b73e247dd...c3a71fcf5b6c2bb28d4f2e5360b
Deleted:    sha256:4d4de39110cd250...28bfe816393d0f2e0dae82c363a

Deleted:    sha256:6a89826eba8d895...cb0d7dba1ef62409f037c6e608b

Deleted: sha256:33efada9158c32d...195aa12859239d35e7fe9566056
Deleted: sha256:c8a75145fcc4e1a...4129005e461a43875a094b93412
Untagged: alpine:latest

Untagged: alpine@sha256:3dcdb92...313626d99b889d0626de158f73a
Deleted: sha256:4e38e38c8ce0b8d...6225e13b0bfe8cfa2321aec4bba
Deleted: sha256:4fe15f8d0ae69e1...eeeeebb265cd2e328e15c6a869f


$ docker images

REPOSITORY  TAG   IMAGE ID    CREATED     SIZE
```

Let’s remind ourselves of the major commands we use to work with Docker images.

Images - The commands

- docker pull is the command to download images. We pull images from repositories inside of remote registries. By default images will be pulled

from repositories on Docker Hub. This command will pull the image tagged as lat-est from the alpine repository on Docker Hub docker pull alpine:latest.

- docker images lists all of the images stored in your Docker host's local cache. To see the SHA256 digests of images add the --digests flag.
- docker rmi is the command to delete images. This command shows how to delete the apline:latest image docker rmi alpine:latest. You cannot delete an image that is associated with a container in the running (UP) or stopped (Exited) states.

Chapter summary

In this chapter we learned about Docker images. We learned that images are made up one or more read-only layers that when stacked together make up the overall image.

We used the docker pull command to pull them into our Docker host's local cache and we covered image naming conventions. Then we learned about image layers and how they can be shared among multiple images. We then covered the most common commands used for working with images.In the next chapter we'll take a similar tour of containers - the runtime cousin of images.

Lesson 6: Containers

Now that we know a bit about images, the next logical step is get into containers. As this is a book about Docker, we'll be talking specifically about Docker containers. However, the Docker project has recently been hard at work implementing the image and container specs published by the Open Container Initiative (OCI) at https://www.opencontainers.org. This means some of what you learn here will apply to other container runtimes that are OCI compliant.

Let’s go and learn about containers!

Docker containers - The TLDR

A container is the runtime instance of an image. In the same way that we can start a virtual machine (VM) from virtual machine template, we start one or more containers from a single image. The big difference between a VM and a container is that containers are faster and more lightweight - instead of running a full-blown OS like a VM, containers run no kernel and just enough OS to get the essentials done.
Figure 6.1 shows a single Docker image being used to start multiple Docker contain-ers.

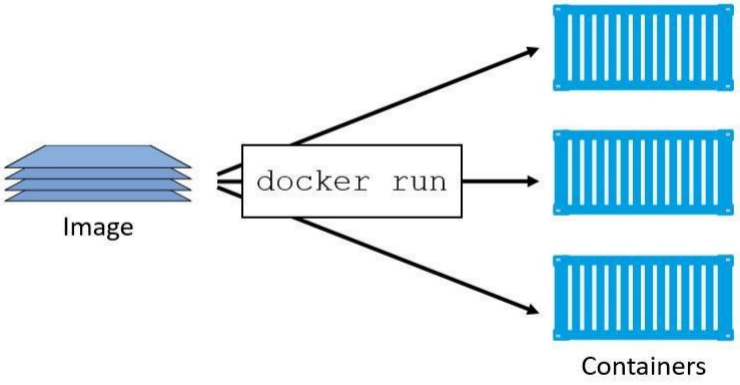


Figure 6.1

The most basic way to start a container is with the docker run command. The command can take a lot of arguments, but in it’s most basic form you tell it an image to use and a command to run: docker run <image> <command>. This next command will start an Ubuntu Linux container running the Bash shell: docker run ubuntu
/bin/bash.

Containers run until the command they are executing exits. You can manually stop a container with the docker stop command, and then restart it with docker start. To get rid of a container forever you have to explicitly delete it using docker rm.

That’s the elevator pitch! Now let’s get into the detail…

Docker containers - The deep dive

The first things we'll cover here are the fundamental differences between a container and a VM. It's mainly theory at this stage, but important stuff. Along the way We'll point out where the container model has potential advantages over the VM model.

> Heads-up: As the author I'm going to say this before we go any further. A lot of us get passionate about the things we do and the skills we have. I remember *big Unix* people resisting the rise of Linux. You might remember the same. You might also remember people attempting to
> resist VMware and the VM juggernaut. In both cases "resistance was futile". In this section I'm going to highlight what I consider some of
> the advantages the container model has over the VM model. But I'm guessing a lot of you will be VM experts with a lot invested in the VM ecosystem. And I'm guessing that one or two of you might want to fight me over some of the things I say. So let me be clear… I'm a big guy and I'd beat you down in hand-to-hand combat :-D Just kidding. What I meant to say was that I'm not trying to destroy your empire or call your baby ugly! I'm trying to help. The whole reason for me writing this book is to help you get started with Docker and containers!

Anyway, here we go.

Containers vs VMs

Containers and VMs both need a host to run on. This can be anything from your laptop, a bare metal server in your data center, all the way up to an

instance the public cloud. In this example we'll assume a single physical server that we need to run 4 business applications on.

In the VM model, the physical server is powered on and the hypervisor boots (we're skipping the BIOS and bootloader code etc.). Once the hypervisor boots it lays claim to all physical resources on the system such as CPU, RAM, storage, and NICs. The hypervisor then carves these hardware resources into virtual versions that look smell and feel exactly like the real thing. It then packages them into a software construct called a virtual machine (VM). We then take those VMs and install an operating system and application on each one. We said we had a single physical server and
needed to run 4 applications, so we'd create 4 VMs, install 4 operating systems, and then install the 4 applications. When it's all done it looks a bit like Figure 6.2.

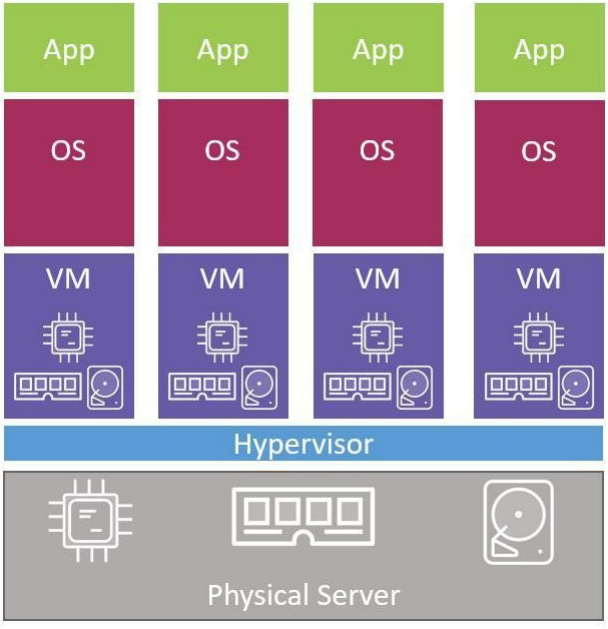


Figure 6.2

Things are a bit different in the container model.

When the server is powered on, your chosen OS boots. In the Docker world this can be Linux, or any version of Windows that has support for the container primitives in its kernel. As per the VM model, the OS claims all hardware resources. On top of the OS we install a container engine such as

Docker. The container engine then takes OS resources such as the *process tree*, the *filesystem*, and the *network stack*, and carves them up into secure isolated constructs called *containers*. Each container looks smells and feels just like a real OS. Inside of each *container* we can run an application. Like before, we're
assuming a single physical server with 4 applications. Therefore we'd carve out 4 containers and run a single application inside of each as shown in Figure 6.3.

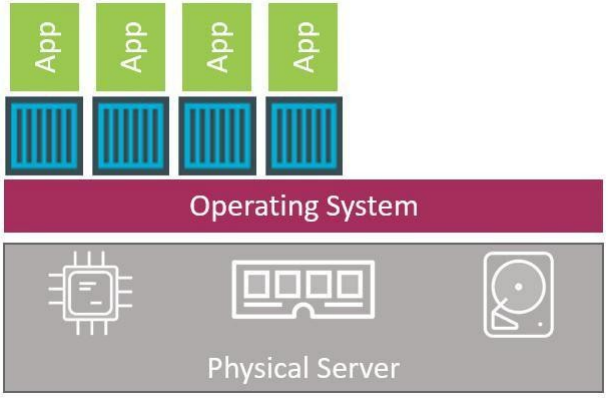


Figure 6.3

At a high level we can say that hypervisors perform hardware virtualization - they carve up physical hardware resources into virtual versions. Whereas containers perform OS virtualization - they carve up OS resources into virtual versions.

The VM tax

Let's build on what we just covered and drill into one of the main problems with the hypervisor model.

We started out with the same physical server and requirement to run 4 business applications. In both models we installed either an OS or a hypervisor (obviously a hypervisor is a type of OS that is highly tuned for VMs). So far the models are almost identical. But this is where the similarities stop.

The VM model then carves low-level hardware resources into VMs. Each VM is a software construct containing virtual CPU, virtual RAM, virtual disk etc. As such, every VM needs it's own OS to claim, initialize and manage all of those

virtual resources. And sadly, every OS comes with it's own set of baggage and overheads. For example, every OS consumes a slice of CPU, a slice of RAM, a slice of storage etc. Most need their own licenses as well as people and infrastructure to patch and upgrade them. Each OS also presents a sizable attack surface. We often refer to all of this as the *OS tax*, or *VM tax* - every OS you install consumes resources!

The container model only has a single kernel down at the host OS layer. It's possible to run tens or hundreds of containers on a single host with every container sharing

that single OS kernel. That means a single OS that consumes CPU, RAM, and storage. A single OS that needs licensing. A single OS that needs upgrading and patching. And a single OS kernel presenting an attack surface. All in all, a single OS tax bill!

That might not seem a lot in our example of a single server needing to run 4 business applications. But when we're talking about hundreds or thousands of apps (VM or containers) this can be game changing.

Another thing to consider is that because a container isn't a full-blown OS, it starts much faster than a VM. Remember, there's no kernel inside of a container that needs locating, decompressing, and initializing - not to mention all of the hardware enumerating and initializing associated with a normal kernel bootstrap. None of that is needed when starting a container! The single shared kernel down at the OS level is already started! Net result, containers can start in less than a second. The only thing that has an impact on container start time is the time it takes to start the application it's running.

This all amounts to the container model being leaner and more efficient than the VM model. We can pack more applications onto less resources, start them faster, and pay less in licensing and admin costs, as well as present less of an attack surface to the dark side. All of which is better for the business!

With that theory out of the way, let's have a play around with some containers.

Running containers

To follow along with these examples you'll need a working Docker host. For most of the commands it won't make a difference if it's Linux or Windows. However, when writing the book I used a Docker host running Ubuntu 16.04 for all examples.

Checking the Docker daemon

The first thing I always do when I log on to a Docker host is check that Docker is running.


```
$ docker version

Client:
 Version:      1.12.1
 API version:  1.24
```

```
 Go version:     go1.6.3
 Git commit:     23cf638
 Built:          Thu Aug 18 05:33:38 2016
 OS/Arch:        linux/amd64
Server:

 Version:        1.12.1
 API version:    1.24
 Go version:     go1.6.3
 Git commit:     23cf638
 Built:          Thu Aug 18 05:33:38 2016
 OS/Arch:        linux/amd64
```

As long as you get a response back in the Client and Server sections you should be good to go. If you get an error code in the Server section there's a good chance that the docker daemon (server) isn't running, or that your user account doesn't have permission to access it.

If your user account doesn't have permission to access the daemon, you need to make sure it's a member of the local docker Unix group. If it isn't, you can add it with usermod -aG docker <user> and then you'll have to logout and log back in to your shell for the changes to take effect.

If your user account is already a member of the local docker group then the problem might be that the Docker daemon isn't running. To check the status of the Docker daemon run one of the following commands depending on your Docker host's operating system.

```
//Run this command on Linux systems not using
Systemd $ service docker status
docker start/running, process 29393

//Run this command on Linux systems that are using
Systemd $ systemctl is-active docker
active

//Run this command on Windows Server 2016 systems from a
PowerShell window > Get-Service docker

Status     Name       DisplayNam
                      e
------     ----       -----------
```

Running docker Docker Engine
Assuming the Docker daemon is running you're fine to continue.

Starting a simple container

The simplest way to start a container is with the docker run command.

The command below starts a simple container that will run a containerized version of Ubuntu Linux.

```
$ docker run -it ubuntu:latest /bin/bash

Unable to find image 'ubuntu:latest' locally latest: Pulling from
library/ubuntu

952132ac251a: Pull complete 82659f8f1b76: Pull complete c19118ca682d:
Pull complete

8296858250fe: Pull   complete

24e0251a0e2c: Pull complete

Digest: sha256:f4691c96e6bbaa99d9...e95a60369c506dd6e6f6ab Status:
Downloaded newer image for ubuntu:latest root@3027eb644874:/#
The format of the command is essentially docker run -<options>
<image>:<tag> <command>.
```

Let's break the command down a bit.

We started with docker run, this is the standard command to start a new container. We then used the -it flags to make the container interactive and attach it to our terminal. Next we told it to use the ubuntu:latest image. Finally we told it to run the Bash shell as its application.

When we hit <Return> the Docker client made the appropriate API calls to the Docker daemon. The Docker daemon accepted the command and searched the Docker host's local cache to see if it already had a copy of the image. In this example it didn't, so it went to Docker Hub to see if it could find it there. I could, so it pulled it locally and stored it in its cache.

> Note: In a standard out-of-the-box installation, the Docker daemon implements the Docker Remote API on a local IPC/Unix socket at /var/run/docker.sock.

Once the image was pulled, the daemon created the container and executed the Bash shell inside of it.

If you look closely you'll see that your shell prompt has changed and you're now inside of the container. In the example above the shell prompt has changed to root@3027eb644874:/#. The long number after the @ is the first 12 characters of the containers unique ID.

Try executing some basic Linux commands from inside of the container. You might notice that some commands do not work. This is because the ubuntu:latest image, like almost all container images, is highly optimized for containers. This means it
doesn't have all of the normal commands and packages installed. The example below shows a couple of commands - one succeeds and the other one fails.

```
root@3027eb644874:/# ls -l

total 64
drwxr-xr-x   2   root root 4096 Aug 19 00:50 bin
drwxr-xr-x   2   root root 4096 Apr 12 20:14 boot
drwxr-xr-x   5 root root   380 Sep 13 00:47 dev
drwxr-xr-x  45   root root 4096 Sep 13 00:47 etc
drwxr-xr-x   2   root root 4096 Apr 12 20:14 home
drwxr-xr-x   8   root root 4096 Sep 13  2015 lib
drwxr-xr-x   2   root root 4096 Aug 19 00:50 lib64
drwxr-xr-x   2   root root 4096 Aug 19 00:50 media
drwxr-xr-x   2   root root 4096 Aug 19 00:50 mnt
drwxr-xr-x   2   root root 4096 Aug 19 00:50 opt
dr-xr-xr-x 129     root root    0 Sep 13 00:47 proc
drwx------   2 root root 4096 Aug 19 00:50 root
drwxr-xr-x   6   root root 4096 Aug 26 18:50 run
drwxr-xr-x   2   root root 4096 Aug 26 18:50 sbin
drwxr-xr-x   2   root root 4096 Aug 19 00:50 srv
dr-xr-xr-x  13 root root    0 Sep 13 00:47 sys
drwxrwxrwt   2 root root 4096 Aug 19 00:50 tmp
drwxr-xr-x  11   root root 4096 Aug 26 18:50 usr
drwxr-xr-x 13 root root    4096 Aug 26 18:50 var
root@3027eb644874:/#
root@3027eb644874:/#
root@3027eb644874:/# ping
www.docker.com bash: ping:
```

command not found
root@3027eb644874:/#

‘

Container processes

When we started the container in the previous section we told it to run the Bash shell
(/bin/bash). This makes the Bash shell the one and only process running inside of the container. You can see this by running ps -elf from inside the container.

```
root@3027eb644874:/# ps -elf
F S UID        PID  PPID  C NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root         1     0  0  0 -  4558 wait   00:47 ?        00:00:00 /bin/bash
0 R root        11     1  0  0 -  8604 -      00:52 ?        00:00:00 ps -elf
```

Although it might look like there are two processes running in the output above, there re not. The first process in the list, with PID 1, is the Bash shell we told the container to run. The second process in the list is the ps -elf command we ran to produce the list. This is a short-lived process that has already exited by the time the output is displayed on the terminal. Long story short, this container is running a single process - /bin/bash.

> Note: Windows containers are slightly different and tend to run quite a few processes.

This means that if you type exit to exit the Bash shell, the container will terminate. The reason for this is that a container cannot exist without a running process - killing the Bash shell would kill the container’s only process, resulting in the container also being killed.

Press Ctrl-PQ to exit the container without terminating it. Doing this will place you back in the shell of your Docker host and leave the container running in the background. You can use the docker ps command to view the list of running containers on your system.

$ docker ps

CNTNR ID	IMAGE	COMMAND	CREATED	STATUS	NAMES
302...74	ubuntu:latest	/bin/bash	6 mins	Up 6mins	sick_montalcini

It's important to understand that this container is still running and you can re-attach your terminal to it with the docker exec command.

$ docker exec -it 3027eb644874
bash root@3027eb644874:/#

Note: You can address a container by its name or ID
As you can see, the shell prompt has changed back to the container. If you run the ps -elf command again you will now see two Bash processes. This is because the docker exec command created a new Bash process and attached to that. This means that typing exit from this Bash prompt will not terminate the container because the original Bash process with PID 1 will continue running.

Type exit to leave the container and verify it's still running with a docker ps.

If you are following along with the examples on your own Docker host you should stop and delete the container with the following two commands (you will need to substitute the ID of your container).

```
$ docker stop

3027eb64487 3027eb64487

$ docker rm 3027eb64487

3027eb64487
```

Container lifecycle

It's a common myth that containers can't persist data. They can!

A big part of the reason people think containers aren't good for persistent workloads or persisting data is because they're so freaking good at non-persistent stuff. But being good at one thing doesn't mean you can't do other things. A lot of VM admins out there will remember companies like Microsoft and Oracle telling you that you couldn't run their applications inside of VMs - or at least they wouldn't support you if you did. I personally wonder if there's a little bit of something similar with the move to containerization - are there

people out there trying to protect their empires of persistent data and workloads from what they perceive as the threat of containers?

Anyway, in this section we'll look at the lifecycle of a container - from birth, through work and vacations, to eventual death.
We've already seen how to start containers with the docker run command.

Let's start another one so we can walk it through its entire lifecycle.

```
$ docker run --name percy -it ubuntu:latest /bin/bash
root@9cb2d2fd1d65:/#
```

That's our container created and we named it "percy" for

persistent :-S Now let's put it to work by writing some data to it.

From within the shell of your new container follow the procedure below to write some data to a new file in the tmp directory and verify that the write operation succeeded.

```
root@9cb2d2fd1d65:/# cd
tmp
root@9cb2d2fd1d65:/tmp#
root@9cb2d2fd1d65:/tmp#
ls -l total 0
root@9cb2d2fd1d65:/tmp#
root@9cb2d2fd1d65:/tmp# echo "sysadmins FTW" > newfile
root@9cb2d2fd1d65:/tmp#
root@9cb2d2fd1d65:/tmp
# ls -l total 4
-rw-r--r-- 1 root root 14 Sep 13 04:22
newfile root@9cb2d2fd1d65:/tmp#
root@9cb2d2fd1d65:/tmp# cat newfile
sysadmins FTW
```

Press Ctrl-PQ to exit the container without killing it.

Now use the docker stop command to stop the container and put in on vacation.

$ docker stop percy percy
You can use the container's name or ID with the docker stop command. The format is docker stop <container-id or container-name>.

Now run a docker ps.

```
$ docker ps
CONTAINER ID IMAGE  COMMAND                  CREATED STATUS
PORTS                NAMES
```

The container is not listed in the output above because it's in the stopped state. Run the same command again, only this time add the -a flag to show all containers including those that are stopped.

```
$ docker ps -a
```

Now we can see the container showing as Exited (0) . Stopping a container is like stopping a virtual machine. Although it's not currently running, its entire configuration and contents still exist on the filesystem of the Docker host and it can be restarted at any time.

Let's use the docker start command to bring it back from vacation.

```
$ docker start percy
percy
$ docker ps
CONTAINER
ID            IMAGE          COMMAND       CREATE
D        STATUS       NAME
S
9cb2d2fd1d6
5             ubuntu:latest  "/bin/bash"   4 mins   Up 3 secs    percy
```

The stopped container is now restarted. Time to verify that the file we created earlier still exists. Connect to the restarted container with the docker exec command.

```
$ docker exec -it percy bash
root@9cb2d2fd1d65:/#
```

Your shell prompt will change to show that you are now operating within the namespace of the container.
Verify that the file you created earlier is still there and contains the data you wrote to it.

```
root@9cb2d2fd1d65:/#
cd tmp
root@9cb2d2fd1d65:/# ls
-l
-rw-r--r-- 1 root root 14 Sep 13 04:22
newfile root@9cb2d2fd1d65:/#
root@9cb2d2fd1d65:/# cat
newfile sysadmins FTW
```

As if by magic the file you created is still there and the data it contains is exactly how you left it! This proves that stopping a container does not destroy the container or the data inside of it.

Now I should point out that there are better and more recommended ways to store data in containers. But at this stage of our journey I think this is an effective example of the persistent nature of containers.
So far I think you'd be hard pressed to draw a major difference in the behavior of a containers a VM.

Now let's kill the container and delete it from our system.

It is possible to delete a *running* container with a single command by passing the -f flag to docker rm. However, it's considered a best practice to take the two-step approach of stopping the container first and then deleting it. This gives the application/process that the container is running a fighting chance of stopping cleanly. More on this in a second.

The example below will stop the percy container, delete it, and verify the operation. If your terminal is still attached to the percy container you will need to get back to your Docker host's terminal by pressing Ctrl-PQ.

```
$ docker stop percy
percy
$
$ docker rm percy
percy
$
$ docker ps
CONTAINER ID  IMAGE       COMMAND CREATED STATUS     PORTS       NAMES
```

The container is now deleted - literally wiped off the face of the planet. If it was a good container, it becomes a VM in the afterlife. If it was a naughty container it becomes a dumb terminal :-D

To summarize the lifecycle of a container. You can stop, start, pause, and restart a container as many times as you want. And it'll all happen really fast. But the container and it's data will always be safe. It's not until you explicitly kill a container that you run any chance of losing its data. And even then, if you're storing data in a *volume*, that data's going to persist even after the container has gone.

Let's quickly mention why we recommended a two-stage approach of stopping the container before deleting it.

Stopping containers gracefully

Most containers will run a single process. In our previous example that process was /bin/bash. When you kill a running container with docker rm <container> -f the container will be killed without warning. The procedure is quite violent - a bit like sneaking up behind the container it and shooting it in the back of the head. You're literally giving the container and the process it's running no chance to straighten it's affairs before being killed.

However, the docker stop command is far more polite. It gives the process inside of the container a heads-up that it's about to be stopped, giving it a chance to get things in order before the end comes. Once the docker stop command returns, you can then delete the container with docker rm.

The magic behind the scenes here has to do with *signals*. docker stop sends a SIGTERM signal to process with PID 1 inside of the container. As we just said, this

gives the process a chance to clean things up and gracefully shut itself down. If it doesn't exit within 10 seconds it will receive a SIGKILL. This is effectively the bullet to the head. But hey, it got 10 seconds to sort itself out first.

docker rm <container> -f doesn't bother asking nicely with a SIGTERM, it just goes straight to the SIGKILL. Like we said a second ago, this is like creeping up from behind and smashing it over the head. I'm not a violent person by then way!

Web server example

So far we've seen how to start a simple container and interact with it. We've also seen how to stop, restart and destroy containers. Now let's take a look at a web server example.

In this example we'll start a new container from an image.The image runs an insanely simple web server on port 8080.

Use the docker stop and docker rm commands to clean up any existing containers on your system. Then run the following docker run command.

```
$ docker run -d --name webserver -p
  80:8080 \ visualpath/devops-docker-ci

Unable to find image 'visualpath/devops-docker-ci:latest' locally
latest: Pulling from visualpath/devops-docker-ci
a3ed95caeb02: Pull
complete 3b231ed5aa2f:
Pull complete
7e4f9cd54d46:      Pull      complete
929432235e51:      Pull      complete
6899ef41c594:      Pull      complete
0b38fccd0dab: Pull complete
Digest: sha256:7a6b0125fe7893e70dc63b2...9b12a28e2c38bd8d3d
Status: Downloaded newer image for visualpath/plur...docker-ci:latest
6efa1838cd51b92a4817e0e7483d103bf72a7ba7ffb5855080128d85043fef21
```

Notice that your shell prompt hasn't changed. This is because we started this container in the background with the -d flag. Starting a container in the background does not attach it to your terminal.

This example threw a few more arguments at the docker run command, so let's take a quick look at them.

We know docker run starts a new container. But this time we give it the -d flag instead of -it. -d tells the container to run in the background rather than attaching to your terminal in the foreground. The "d" stands for daemon mode, and -d and -it are mutually exclusive. This means you can't use both on the same container - for obvious reasons you cannot start a container in the background *and* in the foreground at the same time.

After that, we name the container and then give it -p 80:8080. The -p flag maps ports on the Docker host to ports in the container. This time we're mapping port 80 on the Docker host to port 8080 inside the container. This means that traffic hitting the Docker host on port 80 will be directed to port 8080 inside of the container. It just so happens that the image we're using for this container defines a web service that listens on port 8080. This means our container will come up running a web server listening on port 8080.

Finally we tell it which image to use.

Running a docker ps command will show the container as running and show the ports that are mapped. It's important to know that port mappings are expressed as
host-port:container-port.

```
$ docker ps
CONTAINER ID COMMAND      STATUS         PORTS                    NAMES
6efa1838cd51 /bin/sh -c...  Up 2 mins 0.0.0.0:80->8080/tcp  webserver
```

We've removed some of the columns form the output above to help with readability.
Now that the container is running and ports are mapped, we can connect to the container by pointing a web browser at the IP address or DNS name of the Docker host on port 80. Figure 6.4 shows the web page that is being

served up by the container

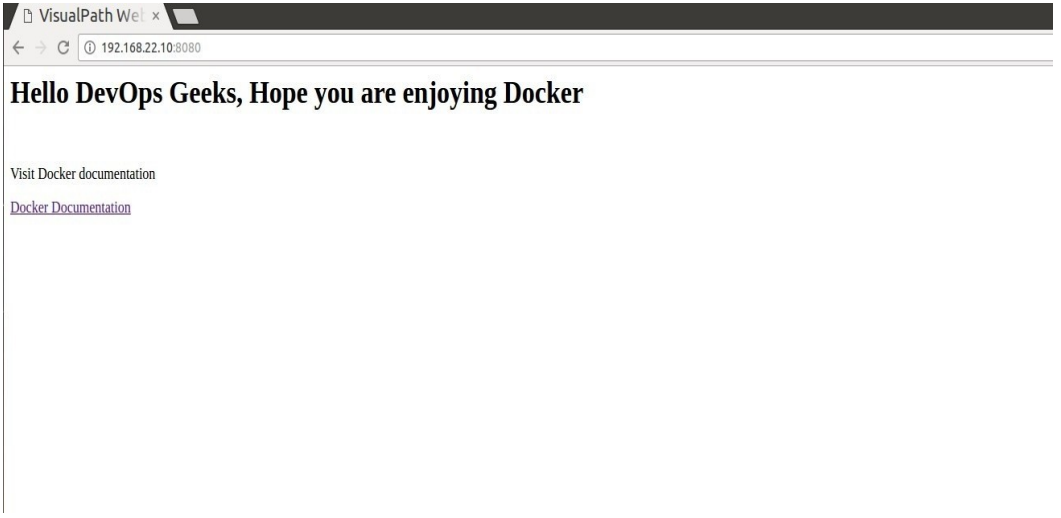


Figure 6.4

The same docker stop, docker pause, docker start, and docker rm commands can be used on the container, and the same rules of persistence apply - stopping or pausing the container does not destroy the container or any data stored in it.

Inspecting containers

In the previous example you might have noticed that we didn't specify a command for the container when we issued the docker run. Yet the container ran a simple web service. How did this happen?
When building a Docker image it's possible to embed a default command or process you want containers using the image to run. If we run a docker inspect command against the image we used to run our container, we'll be able to see the command/process that the container will run when it starts.

```
$ docker inspect visualpath/devops-docker-ci

[
    {
        "Id":
        "sha256:07e574331ce3768f30305519...49214bf3020ee69bba1",
        "RepoTags": [
            "visualpath/devops-docker-ci:latest"

            <Snip>

            ],
            "Cmd"
            : [
                "/bin/sh",
                "-c",
                "#(nop) CMD [\"/bin/sh\" \"-c\" \"cd /src \u0026\u0026 node \
./app.js\"]" ],

<Snip>
```

We've snipped out the output to make is easier to find the information we're interested in.

The entries after "Cmd" show the command(s) that the container will run unless you override the with a different command as part of docker run. If you remove all of the shell escapes in the example above, you get the following command /bin/sh -c
"cd /src \u0026\u0026 node ./app.js".

It's common to build images with default commands like this as it makes starting containers easier, forces a default behavior, and is a form of self documentation for the image.
That's us done for the examples in this chapter. Let's see a quick way to tidy our system up.

Tidying up

Here we're going to show you the simplest and quickest way to get rid if every running container on your Docker host. Be warned though, the procedure will

forcible destroy all containers without giving them a chance to clean up. This should never be performed on production systems or systems running important containers.

Run the following command from the shell of your Docker host to delete all containers.

```
$ docker rm $(docker ps -aq) -f
6efa1838cd51
```

In this example we only had a single container running, so only one was deleted (6efa1838cd51). However, the command works the same way as the docker rmi $(docker images -q) command we used in the previous chapter to delete all images on a single Docker host. We already know the docker rm command deletes containers. Passing it $(docker ps -aq) as an argument effectively passes it the ID of every container on the system. The -f flag forces the operation so that running containers will also be destroyed. Net result... all containers, running or stopped, will be destroyed and removed from the system.

Containers - The commands

- docker run is the command used to start new containers. In its simplest form it accepts an image and a command as arguments. The image is used to create the container and the command is the process or application you want the container to run. This example will start an Ubuntu container in the foreground and running the Bash shell: docker run -it ubuntu /bin/bash.
- Ctrl-PQ will detach your shell from the terminal of a container and leave the container running (UP) in the background.
- docker ps lists all containers in the running (UP) state. If you add the -a flag you will also see containers in the stopped (Exited) state.
- docker exec lets you run a new process inside of a running container. It's
  useful for attaching the shell of your Docker host to a terminal inside of a running container. This command will start a new Bash shell inside of a running container and connect to it: docker exec -it <container-name or
  container-id> bash.

- docker stop will stop a running container and put it in the (Exited (0)) state. It does this by issuing a SIGTERM to the process with PID 1 inside of the container. If the process has not cleaned up and stopped within 10 seconds, a SIGKILL will be issued to forcibly stop the container. docker stop accepts container IDs and container names as arguments.
- docker start will restart a stopped (Exited) container. You can give docker start the name or ID of a container.
- docker rm will delete a stopped container. You can specify containers by name or ID. It is recommended that you stop a container with the docker stop command before deleting it with docker rm.
- docker inspect will show you detailed configuration and runtime informa-tion about a container. It accepts container names and container IDs as its main argument. You can also use docker inspect with Docker images.

Chapter summary

In this chapter we compared and contrasted the container and VM models. We looked at the *OS tax* problem of the VM model and saw how the container model can bring huge efficiencies in much the same way as the VM model brought huge advantages over the physical model.

We saw how to use the docker run command to start a couple of simple containers, and we saw the difference between interactive containers in the foreground versus containers running in the background.
We know that killing the process with PID 1 inside of a container will kill the container. And we've seen how to start, stop, and delete containers.

We finished the chapter using the docker inspect command to view detailed configuration metadata.

So far so good!

In the next chapter we'll see how to orchestrate containerized applications across multiple Docker hosts with some game changing technologies introduced in Docker 1.12.

Lesson 7:Using Dockerfiles toAutomate Building of Images

Introduction

Docker containers are created by using [base] images. An image can be basic, with nothing but the operating-system fundamentals, or it can consist of a sophisticated pre-built application stack ready for launch.
When building your images with docker, each action taken (i.e. a command executed such as apt-get install) forms a new layer on top of the previous one. These base images then can be used to create new containers.

In this article, we will see about automating this process as much as possible, as well as demonstrate the best practices and methods to make most of docker and containers via Dockerfiles: scripts to build containers, step-by-step, layer-by-layer, automatically from a source (base) image.

Glossary

1. **Docker in Brief**

2. **Dockerfiles**
3. **Dockerfile Syntax**

   - What is Syntax?
   - Dockerfile Syntax Example

4. **Dockerfile Commands**

- ADD
- CMD
- ENTRYPOINT
- ENV
- EXPOSE
- FROM
- MAINTAINER
- RUN
- USER
- VOLUME
- WORKDIR

5. **How To Use Dockerfiles**
6. **Dockerfile Example: Creating an Image to Install MongoDB**
   - Creating the Empty Dockerfile
   - Defining Our File and Its Purpose
   - Setting The Base Image to Use
   - Defining The Maintainer (Author)
   - Updating The Application Repository List
   - Setting Arguments and Commands for Downloading MongoDB
   - Setting The Default Port For MongoDB
   - Saving The Dockerfile
   - Building Our First Image
   - Running A MongoDB Instance

Docker in Brief

The **docker project** offers higher-level tools which work together, built on top of some Linux kernel features. The goal is to help developers and system administrators port applications. - with all of their dependencies conjointly - and get them running across systems and machines headache free.

> Docker achieves this by creating safe, LXC (i.e. Linux Containers) based environments for applications called “docker containers”. These containers are created using docker images, which can be built either by executing commands manually or automatically through Dockerfiles

Note: To learn more about docker and its parts (e.g. docker daemon, CLI, images etc.), check out our introductory article to the project: Docker Explained: Getting Started .

Dockerfiles

Each Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image in order to create (or form) a new one. They are used for organizing things and greatly help with deployments by simplifying the process start-to-finish.

Dockerfiles begin with defining an image FROM which the build process starts. Followed by various other methods, commands and arguments (or conditions), in return, provide a new image which is to be used for creating docker containers.

They can be used by providing a Dockerfile's content - in various ways - to the **docker daemon** to build an image (as explained in the "How To Use" section).

Dockerfile Syntax

Before we begin talking about Dockerfile, let's quickly go over its syntax and what that actually means.

What is Syntax?

Very simply, syntax in programming means a structure to order commands, arguments, and everything else that is required to program an application to perform a procedure (i.e. a function / collection of instructions).

These structures are based on rules, clearly and explicitly defined, and they are to be followed by the programmer to interface with whichever computer application (e.g. interpreters, daemons etc.) uses or expects them. If a script (i.e. a file containing series of tasks to be performed) is not correctly structured (i.e. wrong syntax), the computer program will not be able to parse it. Parsing roughly can be understood as going over an input with the end goal of understanding what is meant.

Dockerfiles use simple, clean, and clear syntax which makes them strikingly easy to create and use. They are designed to be self explanatory, especially because they allow commenting just like a good and properly written application source-code.

Dockerfile Syntax Example

Dockerfile syntax consists of two kind of main line blocks: comments

and commands + arguments.

Line blocks used for commenting

command argument argument ..

A Simple Example:

```
# Print "Hello docker!" RUN
echo "Hello docker!"
```

Dockerfile Commands (Instructions)

Currently there are about a dozen different set of commands which Dockerfiles can contain to have docker build an image. In this section, we will go over all of them, individually, before working on a Dockerfile example.

Note: As explained in the previous section (Dockerfile Syntax), all these commands are to be listed (i.e. written) successively, inside a single plain text file (i.e. Dockerfile), in the order you would like them performed (i.e. executed) by the docker daemon to build an image. However, some of these commands (e.g. MAINTAINER) can be placed anywhere you seem fit (but always after FROM command), as they do not constitute of any execution but rather value of a definition (i.e. just some additional information).

ADD

The ADD command gets two arguments: a source and a destination. It basically copies the files from the source on the host into the container's own filesystem at the set destination. If, however, the source is a URL (e.g. http://github.com/user/file/), then the contents of the URL are downloaded and placed at the destination.

Example:

```
# Usage: ADD [source directory or URL] [destination directory] ADD
/my_app_folder /my_app_folder
```

CMD

The command CMD, similarly to RUN, can be used for executing a specific command. However, unlike RUN it is not executed during build, but when a container is instantiated using the image being built. Therefore, it should be considered as an initial,default command that gets executed (i.e. run) with the creation of containers based on the image.

To clarify: an example for CMD would be running an application upon creation of a container which is already installed using RUN (e.g. RUN apt-get install ...) inside the image. This default application execution command that is set with CMD becomes the default and replaces any command which is passed during the creation.

Example:

```
# Usage 1: CMD application "argument", "argument", .. CMD
"echo" "Hello docker!"
```

ENTRYPOINT

ENTRYPOINT argument sets the concrete default application that is used every time a container is created using the image. For example, if you have installed a specific application inside an image and you will use this image to only run that application, you can state it with ENTRYPOINT and whenever a container is created from that image, your application will be the target.

If you couple ENTRYPOINT with CMD, you can remove "application" from CMD and just leave "arguments" which will be passed to the ENTRYPOINT.

Example:

```
#  Usage: ENTRYPOINT application "argument", "argument", ..
#  Remember: arguments are optional. They can be provided by
```

```
CMD
#       or during the creation of a container. ENTRYPOINT echo

#  Usage example with CMD:
#  Arguments set with CMD can be overridden during *run* CMD
"Hello docker!"
ENTRYPOINT echo
```

ENV

The ENV command is used to set the environment variables (one or more). These variables consist of “key = value” pairs which can be accessed within the container by scripts and applications alike. This functionality of docker offers an enormous amount of flexibility for running programs.

Example:

```
# Usage: ENV key value
ENV SERVER_WORKS 4
```

EXPOSE

The EXPOSE command is used to associate a specified port to enable networking between the running process inside the container and the outside world (i.e. the host).

Example:

```
# Usage: EXPOSE [port]
EXPOSE 8080
```

To learn about ports, check out this document on port redirection .

FROM

FROM directive is probably the most crucial amongst all others for Dockerfiles. It defines the base image to use to start the build process. It can be any image, including the ones you have created previously. If a FROM image is not found on the host, docker will try to find it (and download) from the **docker image index**. It needs to be the first command declared inside a Dockerfile.

Example:

```
# Usage: FROM [image name]
FROM ubuntu
```

MAINTAINER

One of the commands that can be set anywhere in the file - although it would be better if it was declared on top - is MAINTAINER. This non-executing command declares the author, hence setting the author field of the images. It should come nonetheless after FROM.

Example:

```
# Usage: MAINTAINER [name]
MAINTAINER authors_name
```

RUN

The RUN command is the central executing directive for Dockerfiles. It takes a command as its argument and runs it to form the image. Unlike CMD, it actually **is** used to build the image (forming another layer on top of the previous one which is committed).

Example:

Usage: RUN [command]
RUN aptitude install -y riak

USER

The USER directive is used to set the UID (or username) which is to run the container based on the image being built.

Example:

Usage: USER [UID]

USER 751

VOLUME

The VOLUME command is used to enable access from your container to a

directory on the host machine (i.e. mounting it).

Example:
Usage: VOLUME ["/dir_1", "/dir_2" ..] VOLUME ["/my_files"]

WORKDIR

The WORKDIR directive is used to set where the command defined with CMD is to be executed.

Example:

Usage: WORKDIR /path WORKDIR ~/

How to Use Dockerfiles

Using the Dockerfiles is as simple as having the docker daemon run one. The output after executing the script will be the ID of the new docker image.

Usage:

```
# Build an image using the Dockerfile at current location
# Example: sudo docker build -t [name] .

$ sudo docker build -t my_mongodb .
```

Dockerfile Example: Creating an Image to Install MongoDB

In this final section for Dockerfiles, we will create a Dockerfile document and populate it step-by-step with the end result of having a Dockerfile, which can be used to create a docker image to run MongoDB containers.

Note: After starting to edit the Dockerfile, all the content and arguments from the sections below are to be written (appended) inside of it successively, following our example and explanations from the **Docker Syntax** section. You can see what the end result will look like at the latest section of this walkthrough.

Creating the Empty Dockerfile

Using the nano text editor, let's start editing our Dockerfile.
$ sudo vi Dockerfile

Defining Our File and Its Purpose

Albeit optional, it is always a good practice to let yourself and everybody figure out (when necessary) what this file is and what it is intended to do. For this, we will begin our Dockerfile with fancy comments (i#) to describe it - and have it like cool kids.

```
###############################################################
# Dockerfile to build MongoDB container images
# Based on Ubuntu
###############################################################
```

Setting The Base Image to Use

```
# Set the base image to Ubuntu
FROM ubuntu
```

Defining The Maintainer (Author)

```
# File Author / Maintainer
MAINTAINER Example McAuthor
```

Updating The Application Repository List

Note: This step is not necessary, given that we are not using the repository right afterwards. However, it can be considered good practice.

```
# Update the repository sources list
```

```
RUN apt-get update
```

Setting Arguments and Commands for Downloading MongoDB

```
################### BEGIN INSTALLATION ######################

# Install MongoDB Following the Instructions at MongoDB Docs
# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-
ubuntu/

# Add the package verification key
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
7F0CEB10

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

# Update the repository sources list once more
RUN apt-get update

# Install MongoDB package (.deb)
RUN apt-get install -y mongodb-10gen

# Create the default data directory
RUN mkdir -p /data/db

###################### INSTALLATION END ######################
```

Setting The Default Port For MongoDB

```
# Expose the default port
EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]

# Set default container command
ENTRYPOINT usr/bin/mongod
```

Saving The Dockerfile

After you have appended everything to the file, it is time to save and exit. Press CTRL+X and then Y to confirm and save the Dockerfile.

This is what the final file should look like:

```
###################################################################
# Dockerfile to build MongoDB container images
# Based on Ubuntu
###################################################################

# Set the base image to Ubuntu
FROM ubuntu

# File Author / Maintainer
MAINTAINER Example McAuthor

# Update the repository sources list
RUN apt-get update

################## BEGIN INSTALLATION ######################
# Install MongoDB Following the Instructions at MongoDB Docs
# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-
ubuntu/

# Add the package verification key
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
7F0CEB10

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

# Update the repository sources list once more
RUN apt-get update

# Install MongoDB package (.deb)
RUN apt-get install -y mongodb-10gen

# Create the default data directory
RUN mkdir -p /data/db

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

# Expose the default port
EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]

# Set default container command
ENTRYPOINT usr/bin/mongod
```

Building Our First Image

Using the explanations from before, we are ready to create our first MongoDB image with docker!

```
$ sudo docker build -t my_mongodb .
```

Note: The **- t [name]** flag here is used to tag the image. To learn more about what else you can do during build, run sudo docker build --help.

Running A MongoDB Instance

Using the image we have build, we can now proceed to the final step: creating a container running a MongoDB instance inside, using a name of our choice (if desired with **-name [name]**).

```
$ sudo docker run -name my_first_mdb_instance -i -t my_mongodb
```

Note: If a name is not set, we will need to deal with complex, alphanumeric IDs which can be obtained by listing all the containers using sudo docker ps -l.
Note: To detach yourself from the container, use the escape sequence CTRL+P by followed by CTRL+Q

Lesson 8: Container Networking Basics

Objectives

We will now run network services (accepting requests) in containers. At the end of this lesson, you will be able to:

- Run a network service in a container.
- Manipulate container networking basics.
- Find a container's IP address.
- We will also explain the different network models used by Docker.

A simple, static web server

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

```
$ docker run -d -P nginx
66b1ce719198711292c8f34f84a7b68c3876cf9f67015e752b9 4e189d35a204e
```

Docker will download the image from the Docker Hub.

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

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

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

Finding our web server port

We will use docker ps:

```
$ docker ps
```

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

Connecting to our web server (GUI)

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

Connecting to our web server (CLI)

You can also use curl directly from the Docker host.

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

```
$ curl localhost:32769
<!DOCTYPE html> <html>
<head>
<title>Welcome to nginx!</title>
```

Why are we mapping ports?

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

Finding the web server port in a script

Parsing the output of docker ps would be painful.There is a command to help us:

```
$ docker port <containerID>
 80 32769
```

Manual allocation of port numbers

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

```
$ docker run -d -p 80:80 nginx
$ docker run -d -p 8000:80 nginx
$ docker run -d -p 8080:80 -p 8888:80 nginx
```

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

Plumbing containers into your infrastructure

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

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

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

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

Finding the container's IP address

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

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

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

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

Pinging our container

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

```
$ ping <ipAddress>

64 bytes from <ipAddress>: icmp_req=1 ttl=64 time=0.085 ms 64 bytes from
    <ipAddress>:     icmp_req=2     ttl=64
    time=0.085   ms  64     bytes   from
    <ipAddress>:     icmp_req=3     ttl=64
    time=0.085 ms
```

The different network drivers

A container can use one of the following drivers:

bridge (default)

none

host

container

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

The default bridge

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

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

Addresses are allocated on a private, internal subnet.
(Docker uses 172.17.0.0/16 by default; configurable with --bip.)
Outbound traffic goes through an iptables MASQUERADE rule.Inbound traffic goes through an iptables DNAT rule.The container can have its own routes, iptables rules, etc.

Section summary

We've learned how to:

Expose a network port.

Manipulate container networking basics.

Find a container's IP address.

In the next chapter, we will see how to connect containers together without exposing their ports.

Lesson 9: The Container Network Model

Objectives

We will learn about the CNM (Container Network Model). At the end of this lesson, you will be able to:

- Create a private network for a group of containers.
- Use container naming to connect services together.
- Dynamically connect and disconnect containers to networks.
- Set the IP address of a container.

We will also explain the principle of overlay networks and network plugins.

The Container Network Model

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

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

```
$ docker network ls
NETWORK ID          NAME                DRIVER
6bde79dfcf70        bridge              bridge
8d9c78725538        none                null
eb0eeab782f4        host                host
4c1ff84d6d3f        blog-dev            overlay
228a4355d548        blog-prod           overlay
```

What's in a network?

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

Creating a network

Let's create a network called dev.

```
$ docker network create dev
4c1ff84d6d3f1733d3e233ee039cac276f425a9d5228a4355d54878293a889ba
```

The network is now visible with the network ls command:

```
$ docker network ls
NETWORK ID          NAME                DRIVER
6bde79dfcf70        bridge              bridge
8d9c78725538        none                null
eb0eeab782f4        host                host
4c1ff84d6d3f        dev                 bridge
```

Placing containers on a network

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

```
$ docker run -d --name search --net dev elasticsearch
8abb80e229ce8926c7223beb69699f5f34d6f1d438bfc5682db893e798046863
```

Communication between containers

Now, create another container on this network.

```
$ docker run -ti --net dev alpine sh root@0ecccdfa45ef:/#
```

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

```
/ # ping search
PING search (172.18.0.2) 56(84) bytes of data.
64 bytes from search.dev (172.18.0.2): icmp_seq=1 ttl=64 time=0.221 ms 64 bytes
from search.dev (172.18.0.2): icmp_seq=2
ttl=64 time=0.114 ms 64 bytes from search.dev (172.18.0.2): icmp_seq=3 ttl=64
time=0.114 ms ^C
--- search ping statistics ---

3 packets transmitted, 3 received, 0% packet loss, time 2000ms rtt min/avg/max/mdev =
0.114/0.149/0.221/0.052 ms
root@0ecccdfa45ef:/#
```

Resolving container addresses

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

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

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

Connecting multiple containers together

- Let's try to run an application that requires two containers.
- The first container is a web server.

- The other one is a redis data store.
- We will place them both on the dev network created before.

Running the web server

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

Start the container, exposing all its ports:

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

Check the port that has been allocated to it:

```
$ docker ps -l
```

Test the web server

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

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

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

Start the data store

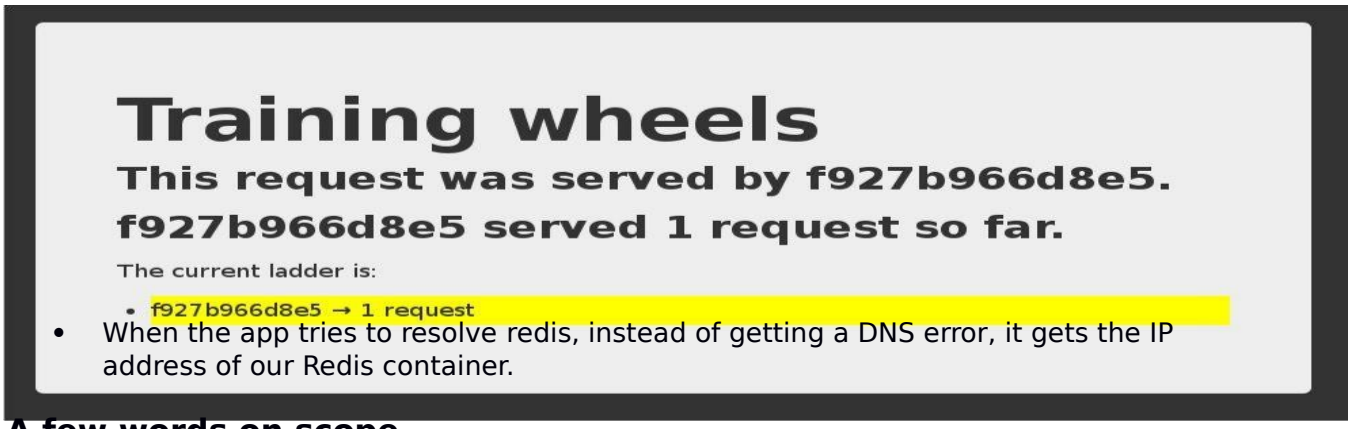
- We need to start a Redis container.
- That container must be on the same network as the web server.
- It must have the right name (redis) so the application can find it.

Start the container:

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

Test the web server again

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



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

A few words on scope

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

Let's remove the redis container:

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

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

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

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

Names are local to each network

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

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

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

Network aliases

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

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

Creating containers on another network

Create the prod network.

```
$ docker create network prod
5a41562fecf2d8f115bedc16865f7336232a04268bdf2bd816aecca01b68d5 0c
```

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

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

38079d21caf0c5533a391700d9e9e920724e89200083df73211081c8a356d771 $ docker run -d
--name prod-es-2 --net-alias search --net prod elasticsearch
1820087a9c600f43159688050dcc164c298183e1d2e62d5694fd46b10ac3bc3d
```

Resolving network aliases

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

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

Address 1: 172.23.0.3 prod-es-2.prod Address 2: 172.23.0.2 prod-es-1.prod
```

(You can ignore the can't resolve '(null)' errors.)

Connecting to aliased containers

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

Try the following command a few times:

```
$ docker run --rm --net dev centos curl -s search:9200
{
"name" : "Tarot",
...
}
```

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

```
$ docker run --rm --net prod centos curl -s search:9200
{
"name" : "The Symbiote",
...
}
```

Good to know...

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

A few words about round robin DNS

Don't rely exclusively on round robin DNS to achieve load balancing. Many factors can affect DNS resolution, and you might see:

- all traffic going to a single instance;
- traffic being split (unevenly) between some instances;
- different behavior depending on your application language;
- different behavior depending on your base distro;
- different behavior depending on other factors (sic).

It's OK to use DNS to discover available endpoints, but remember that you have to re-resolve every now and then to discover new endpoints.

Custom networks

- When creating a network, extra options can be provided.
- --internal disables outbound traffic (the network won't have a default gateway).
- --gateway indicates which address to use for the gateway (when utbound traffic is allowed).
- --subnet (in CIDR notation) indicates the subnet to use.
- --ip-range (in CIDR notation) indicates the subnet to allocate from.
- --aux-address allows to specify a list of reserved addresses (which won't be allocated to containers).

Setting containers' IP address

It is possible to set a container's address with --ip. The IP address has to be within the subnet used for the container.

A full example would look like this.

```
$ docker network create --subnet 10.66.0.0/16 pubnet
42fb16ec412383db6289a3e39c3c0224f395d7f85bcb1859b279e7a564d4e135 $

docker run --net pubnet --ip 10.66.66.66 -d nginx
b2887adeb5578a01fd9c55c435cad56bbbe802350711d2743691f95743680b09
```

Note: don't hard code container IP addresses in your code!

I repeat: don't hard code container IP addresses in your code!

Overlay networks

The features we've seen so far only work when all containers are on a single host.If containers span multiple hosts, we need an overlay network to connect them together. Docker ships with a default network plugin, overlay, implementing an overlay network leveraging VXLAN. Other plugins (Weave, Calico...) can provide overlay networks as well.Once you have an overlay network, all the features that we've used in this chapter work identically.

Multi-host networking (overlay)

Out of the scope for this intro-level workshop! Very short instructions: enable Swarm Mode (docker swarm init then docker swarm join on other nodes) docker network create mynet --driver overlay docker service create --network mynet my image.
See http://jpetazzo.github.io/orchestration-workshop for all the deets about clustering!

Multi-host networking (plugins)

Out of the scope for this intro-level workshop!

General idea:

install the plugin (they often ship within containers)

run the plugin (if it's in a container, it will often require extra parameters; don't just docker run it blindly!)

some plugins require configuration or activation (creating a special file that tells Docker "use the plugin whose control socket is at the following location")

you can then docker network create --driver pluginname

Section summary

We've learned how to:

- Create private networks for groups of containers.
- Assign IP addresses to containers.
- Use container naming to implement service discovery.

Lesson 10: Local Development Workflow with Docker

Objectives

At the end of this lesson, you will be able to:

- Share code between container and host.
- Use a simple local development workflow.

Using a Docker container for local development

Never again:

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

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

Our "namer" application

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

Let's run it with:

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

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

Let's look at the code

Let's download our application's source code.

```
$ git clone https://github.com/jpetazzo/namer
$ cd namer $ ls -1 company_name_generator.rbconfig.ru docker-compose.yml
Dockerfile Gemfile
```

Where's my code?

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

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

COPY . /src
WORKDIR /src
RUN bundler install

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

```
EXPOSE 9292
```
We want to make changes *inside the container* without rebuilding it each time.
For that, we will use a volume.

Our first volume

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

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

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

Mounting volumes inside containers

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

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

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

There will be a full chapter about volumes!

Testing the development container

Now let us see if our new container is running.

```
$ docker ps
```

Viewing our application

Now let's browse to our web application on:

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

We can see our company naming application.

Making a change to our application

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

```
$ vi company_name_generator.rb
```

And change

```
color: royalblue;

To:  color: red;
```

Refreshing our application

Now let's refresh our browser:

`http://<yourHostIP>:80`
We can see the updated color of our company naming application.

Improving the workflow with Compose

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

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

```
www: build: . volumes:
.:/src
ports:
80:9292
```

Why Compose?

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

Workflow explained

We can see a simple workflow:

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

Debugging inside the container

In 1.3, Docker introduced a feature called docker exec.

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

docker exec example

```
$ # You can run ruby commands in the area the app is running and more! $ docker exec
-it <yourContainerId> bash root@5ca27cf74c2e:/opt/namer# irb
irb(main):001:0> [0, 1, 2, 3, 4].map {|x| x ** 2}.compact => [0, 1, 4, 9,
16]
irb(main):002:0> exit
```

Stopping the container

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

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

And remove it
```
$ docker rm ourContainerID>
```

Section summary

We've learned how to:

- Share code between container and host.
- Set our working directory.
- Use a simple local development workflow.

Lesson 11: Using Docker Compose for Development Stacks

Objectives

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

Compose For Development Stacks

What is Docker Compose?

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

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

Compose overview

This is how you work with Compose:

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

Checking if Compose is installed

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

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

Installing Compose

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

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

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

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

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

Launching Our First Stack with Compose

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

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

Second step: start your app.

```
$ docker-compose up
```

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

Launching Our First Stack with Compose

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

Stopping the app

When you hit ^C, Compose tries to gracefully terminate all of the containers.After ten seconds (or if you press ^C again) it will forcibly kill them.

The docker-compose.yml file

Here is the file used in the demo:

```
version: "2"

services:
  www:
    build: www ports:
      - 8000:5000 user:
    nobody environment:
      DEBUG: 1
    command: python counter.py
    volumes:
      - ./www:/src

  redis:
    image: redis
```

Compose file versions

Version 1 directly has the various containers (www, redis...) at the top level of the file.

Version 2 has multiple sections:

- version is mandatory and should be "2".
- services is mandatory and corresponds to the content of the version 1 format.
- networks is optional and can define multiple networks on which containers can be placed.
- volumes is optional and can define volumes to be used (and potentially shared) by the containers.

Containers in docker-compose.yml

Each service in the YAML file must contain either build, or image.

- build indicates a path containing a Dockerfile.
- image indicates an image name (local, or on a registry).

The other parameters are optional.
They encode the parameters that you would typically add to docker run. Sometimes they have several minor improvements.

Container parameters

- command indicates what to run (like CMD in a Dockerfile).
- ports translates to one (or multiple) -p options to map ports. You can specify local ports (i.e. x:y to expose public port x).
- volumes translates to one (or multiple) -v options.

You can use relative paths here.

For the full list, check http://docs.docker.com/compose/yml/.

Compose commands

We already saw docker-compose up, but another one is docker-compose build. It will execute docker build for all containers mentioning a build path.It is common to execute the build and run steps in sequence:

```
docker-compose build && docker-compose up
```

Another common option is to start containers in the background:

```
docker-compose up –d
```

Check container status

It can be tedious to check the status of your containers with docker ps, especially when running multiple apps at the same time.
Compose makes it easier; with docker-compose ps you will see only the status of the containers of the current stack:

```
$ docker-compose ps       Command               State     Ports
Name
-------------------------------------------------------------------------------
trainingwheels_redis_1    /entrypoint.sh red    Up        6379/tcp
trainingwheels_www_1      python counter.py     Up        0.0.0.0:8000->5000/tcp
```

Cleaning up

If you have started your application in the background with Compose and want to stop it easily, you can use the kill command:

```
$ docker-compose kill
```

Likewise, docker-compose rm will let you remove containers (after confirmation):

```
$ docker-compose rm
Going to remove trainingwheels_redis_1, trainingwheels_www_1
Are you sure? [yN] y
Removing trainingwheels_redis_1...
Removing trainingwheels_www_1...
```

Alternatively, docker-compose down will stop and remove containers.

```
$ docker-compose down
Stopping
trainingwheels_www_1 ... done
Stopping trainingwheels_redis_1
... done Removing
trainingwheels_www_1 ... done
Removing trainingwheels_redis_1
... done
```

Special handling of volumes

Compose is smart. If your container uses volumes, when you restart your application,

Compose will create a new container, but carefully re-use the volumes it was using previously.
This makes it easy to upgrade a stateful service, by pulling its new image and just restarting your stack with Compose.

Project:

Prereqs

1. Create directories on your local ubuntu machine
   # mkdir -p ~/Docker-Projects/host-projects

2. Place the vagrant file in ~/Docker-Projects/Vagrantfile info
   Box => ubuntu/trusty64
   Network => Host only(Assign a private IP)
   Directory sync, host dir ~/Docker-Projects/host-projects to vm dir ~/Projects
   Memory => 2048 MB
3. Bring up the vagrant vm and login into it.

Getting setup - Installing Docker on vagrant vm

Download Docker for Ubuntu trusty64
$ wget https://apt.dockerproject.org/repo/pool/main/d/docker-engine/docker-engine_1.12.3-0~trusty_amd64.deb

First install Docker's system dependencies
$ sudo apt-get install libapparmor1 aufs-tools ca-certificates

Install Docker itself
$ sudo dpkg -i docker-engine_1.12.3-0~trusty_amd64.deb

Add your user to the docker group so you can run Docker without root
$ sudo usermod -aG docker $(whoami)

Getting setup - Installing additional Docker tools

curl because it's not installed by default
$ sudo apt-get install curl

Download Docker Compose for ubuntu (Linux)
$ curl -L https://github.com/docker/compose/releases/download/1.8.1/docker-compose-Linux-x86_64 > /tmp/docker-compose

Make docker-compose executable
$ chmod +x /tmp/docker-compose

Put docker-compose on your system path
$ sudo mv /tmp/docker-compose /usr/local/bin

Verify that you can access it
$ docker-compose –version

Getting setup - Installing Sublime Text

Download Sublime Text (get the one for your desktop Ubuntu)
http://www.sublimetext.com/3

Sublime text should be installed on your desktop ubuntu OS and not in the vagrant vm.
We will put all our code in the synced folder ~/Docker/host-projects
which will be accessible inside the vagrant vm's ~/Projects folder.

We can then use sublime text and edit our code from out desktop ubuntu OS.

Building a Dockerized web app - Setting up a project directory

Login to the vagrant vm

Create the MobyDock folder inside of a new Projects folder in your home directory
$ mkdir -p ~/Projects/MobyDock

Flask - Project scaffolding

Create the mobydock web application folder
$ mkdir -p ~/Projects/MobyDock/mobydock

Move into the mobydock folder
$ cd ~/Projects/MobyDock/mobydock

Create a few necessary files
$ touch requirements.txt .gitignore Dockerfile docker-compose.yml
.dockerignore

Create a few necessary folders
$ mkdir mobydock config instance

Create settings files
$ touch config/__init__.py config/settings.py instance/__init__.py
instance/settings.py_production_example

Move into the mobydock python module
$ cd mobydock

^ You should now be in
~/Projects/MobyDock/mobydock/mobydock

Create application files
$ touch app.py __init__.py

$ Create application folders
mkdir templates static

$ Create templates and assets
touch templates/layout.html static/main.css

```
$ Download the Docker whale image to your static folder
curl -L
http://doc.rultor.com/images/docker-logo.png  >
static/docker-logo.png
```

Flask - The Dockerfile

```
Move into the mobydock folder
$ cd ~/Projects/MobyDock/mobydock

Write the docker file to create image which will host our flask app.

$ vi Dockerfile
FROM python:2.7-slim
MAINTAINER Nick Janetakis <nick.janetakis@gmail.com>

RUN apt-get update && apt-get install -qq -y build-essential libpq-dev postgresql-
client-9.4 --fix-missing --no-install-recommends

ENV INSTALL_PATH /mobydock
RUN mkdir -p $INSTALL_PATH

WORKDIR $INSTALL_PATH

COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt

COPY . .
VOLUME ["static"]
CMD gunicorn -b 0.0.0.0:8000 "mobydock.app:create_app()"
```

Docker Compose

We will use docker-compose to automate our docker runtime enviornment.
To create a functional project we need three containers.

- Flask app container to host our flask project
- Postgres container for the DB
- Redis container for the caching

Write a docker-compose file to run all the three containers.

```
Move into the mobydock folder
$ cd ~/Projects/MobyDock/mobydock
$  vi docker-compose.yml
postgres:
  image: postgres:9.4.5
  environment:
    POSTGRES_USER: mobydock
    POSTGRES_PASSWORD: yourpassword
  ports:
    - '5432:5432'
  volumes:
```

```
    - ~/.docker-volumes/mobydock/postgresql/data:/var/lib/postgresql/data
  redis:
  image: redis:2.8.22
  ports:
    - '6379:6379'
  volumes:
    - ~/.docker-volumes/mobydock/redis/data:/var/lib/redis/data

mobydock:
  build: .
  command: gunicorn -b 0.0.0.0:8000 --reload --access-logfile -
"mobydock.app:create_app()"
  environment:
    PYTHONUNBUFFERED: true
  links:
    - postgres
    - redis
  volumes:
    - .:/mobydock
  ports:
- '8000:8000'
```

Flask - Building the app

Get the the resource for the flask app and copy it in the synced folder of your desktop OS ~/Docker-Projects/host-projects
Name of the file is Dockerized-app-Flask.zip

Login to the vagrant vm and move to ~/Projects directory, you will see Dockerized-app-Flask.zip file. Unzip the file so it copies all the project files inside your MobyDock directory.

```
$ cd ~/Projects
$ unzip Dockerized-app-Flask.zip
$ cd ~/Projects/MobyDock/
$ tree (if tree is not installed, install it with apt-get install tree command)
.
└── mobydock
    ├── config
    │   ├── __init__.py
    │   ├── __init__.pyc
    │   ├── settings.py
    │   └── settings.pyc
    ├── docker-compose.yml
    ├── Dockerfile
    ├── instance
    │   ├── __init__.py
    │   └── settings.py.production_example
    ├── mobydock
    │   ├── app.py
    │   ├── app.pyc
    │   ├── __init__.py
    │   ├── __init__.pyc
    │   ├── static
    │   │   ├── docker-logo.png
```

```
│   │   └── main.css
│   └── templates
│       └── layout.html
└── requirements.txt
```

6 directories, 16 files

Flask - Running Docker Compose

Make sure you're in the same directory as the Dockerfile
$ cd ~/Projects/MobyDock/mobydock

Investigate the Docker Compose help menu
$ docker-compose --help

Run Docker Compose by launching everything in one go
$ docker-compose up

[open a 2nd terminal and login to vagrant vm]

Confirm the newly downloaded images
$ docker images

Confirm the newly running containers with Docker
$ docker ps

Confirm the newly running containers with Docker Compose
$ docker-compose ps

Create the PostgreSQL database
$ docker exec mobydock_postgres_1 createdb -U postgres mobydock

^ If you see that it already exists, that's ok.

Create the database user and grant permissions onto the database
docker exec mobydock_postgres_1 psql -U postgres -c "CREATE USER mobydock WITH PASSWORD 'yourpassword'; GRANT ALL PRIVILEGES ON DATABASE mobydock to mobydock;"

^ If you see that it already exists, that's ok.

^ If you changed the database name, username or password in the config/settings.py file you will need to adjust the files in both this command as well as the docker-compose.yml file.

Visit the /seed route in the xubuntu's web browser
http://localhost:8000/seed

Feed MobyDock until you get bored
<Keep reloading your browser>

[Go back to the 1st terminal tab]

Hit CTRL+C to exit Docker Compose and see if everything shut down
docker ps

^ If you see anything, this is an edge case bug with Docker Compose.

Ensure the containers get stopped
docker-compose stop

Install git
sudo apt-get install git

Configure git
git config --global user.email "
you@example.com
"
git config --global user.name "
Your Name
"

^ Customize the text in blue with whatever you want.

Create a git repo out of the project
git init && git add -A && git commit -m "Initial commit"