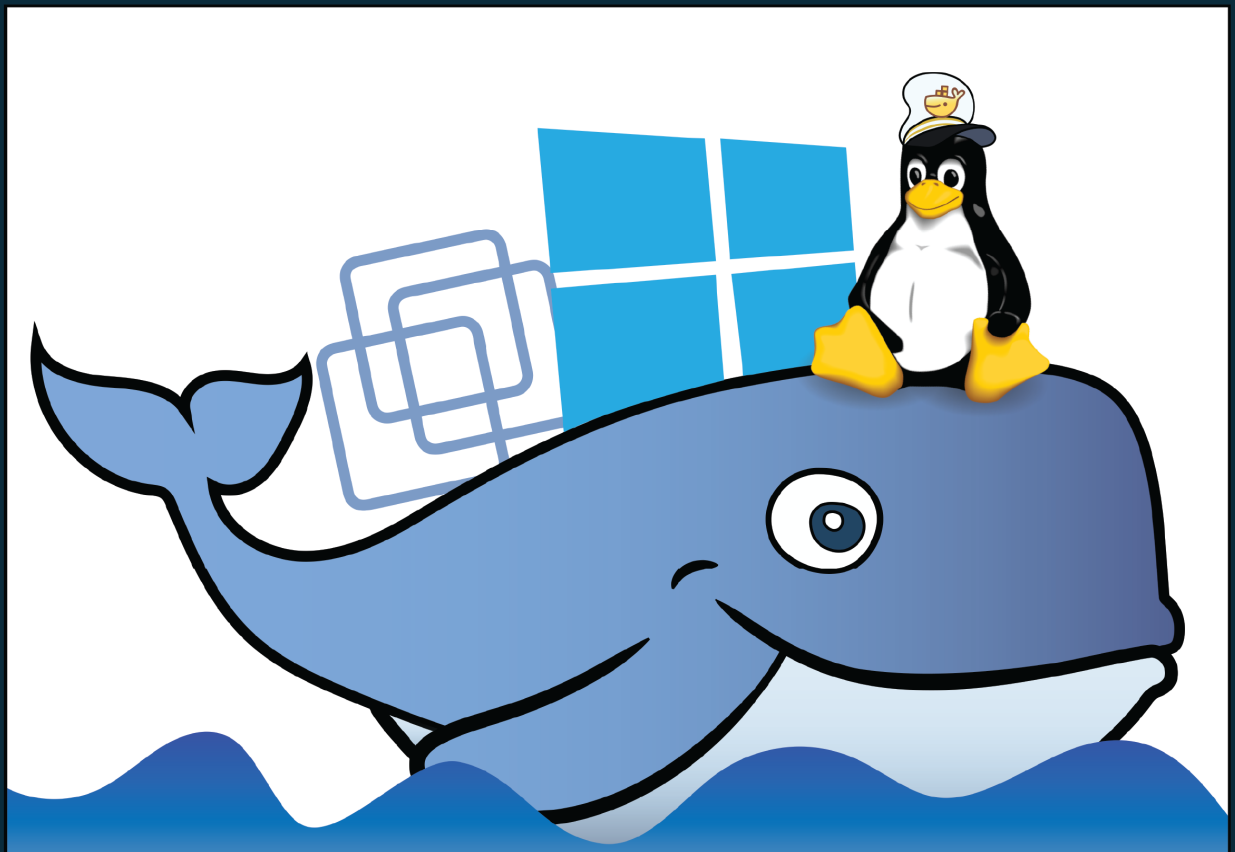
Docker
For Sysadmins
Linux Windows VMware



Hand-crafted for Infrastructure Ops

Nigel Poulton

Docker for Sysadmins: Linux Windows VMware

Getting started with Docker from the perspective of sysadmins and VM admins

Nigel Poulton

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

This version was published on 2016-09-23

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

Huge thanks to my wife and kids for putting up with a geek in the house who genuinely thinks he's a bunch of software running inside of a container on top of midrange biological hardware. It can't be easy living with me!

Massive thanks as well to everyone who watches my Pluralsight videos. I love connecting with you and really appreciate all the feedback I've gotten over the years. This was one of the major reasons I decided to write this book! I hope it'll be an amazing tool to help you drive your careers even further forward.

Contents

0: About the book

This is a book about Docker, **hand-crafted for system administrators**. No prior knowledge required!

But what about developers and DevOps?

If you're a developer with no interest in operations then this book is not for you. If you're into DevOps then I think you'll get a lot form the book.

To keep things short… the book is **not** about showing you how to develop microservice apps with Docker. The book **is** about how the core Docker plumbing works. You'll learn the *how* and the *why* - the *commands* and the *deep-dives*. I really want to set you on your way to being as good at Docker as you already are at Linux, Windows or VMware.

Why should I read this book or care about Docker?

Docker is coming and there's no hiding from it. Developers are all over it. In IT Ops, we need to get ready to support *Dockerized* apps in our business critical production environments.

Isn't Docker just for developers?

Hell no!!!

All of those *Dockerized* apps that developers are creating need a solid Docker infrastructure to run on. And that's where IT Ops comes into the picture… IT Ops will be asked to build and run high performance and highly available Docker infrastructures to support business applications. If we're not skilled-up on Docker, we're going to struggle.

Why this Docker book and not another one?

At the time I decided to write the first edition of this book, *so many* of the Docker books already out there were terrible! They were a shocking mix of *badly written*, full of *technical inaccuracies*, or massively *out of date*. And sometimes they were all three! It's honestly not my intention to offend people, but go and read some of the reviews on Amazon. ***Some*** of the Docker books out there are a shameful waste of trees and paper!

So I decided to write something that was *well written*, *technically accurate*, and *kept up to date*. I want you to love this book.

If you buy the book and think it's bad, call me out on Twitter[1], give the book bad reviews, do whatever you feel necessary. And I'll try and fix it. But I'm confident you won't need to do any of that.

Should I buy the book if I've already watched your video courses?

If you like my video courses[2] you'll probably like the book. If you don't like my video courses you probably won't like the book.

How the book is organized

I've divided the book into two sections:

- The general info stuff
- The technical stuff

The general info stuff covers things like - Who is Docker, Inc. What is the Docker project. What is the OCI. Why do we even have containers... Not the coolest part of

[1] https://twitter.com/nigelpoulton

[2] https://app.pluralsight.com/library/search?q=nigel+poulton

the book, but the kind of stuff that's important if you want a good rounded knowledge of Docker and containers. It's only a short section and you probably *should* read it.

The technical stuff is what the book is all about! This is where you'll find everything you need to start working with Docker. It gets into the detail of *images*, *containers* and the increasingly important topic of *orchestration*. You'll get the theory so that you know how it all fits together, and you'll get commands and examples to show you how it all works in practice.

Every chapter in the *technical stuff* section is divided into three parts:

- The TLDR
- The deep dive
- The commands

The TLDR will give you two or three paragraphs that you could use to explain the topic at the coffee machine.

> **TLDR** or TL;DR, is a modern acronym meaning "too long; didn't read". It's normally used to indicate something that was too long to bother reading. I'm using it here in the book to indicate a short section that you can read if you're in a hurry and haven't got time to read the longer *deep dive* that immediately follows it.

The deep dive is where we'll explain how everything works and go through the examples.

The Commands lists out all of the commands you've learned in an easy to read list with brief reminders of what each one does.

I think you'll love that format.

Other stuff about the book

Here are just a few other things I want you to know about the book.

Text wrapping

I've tried really hard to get the commands and outputs to fit on a single line without wrapping! So instead of getting this...

```
$ docker service ps uber-service
ID                         NAME                IMAGE                     NOD\
E              DESIRED STATE  CURRENT STATE               ERROR
7zi85ypj7t6kjdkevreswknys  uber-service.1      nigelpoulton/tu-demo:v2  ip-\
172-31-12-203  Running        Running about an hour ago
0v5a97xatho0dd4x5fwth87e5   \_ uber-service.1   nigelpoulton/tu-demo:v1  ip-\
172-31-12-207  Shutdown       Shutdown about an hour ago
31xx0df6je8aqmkjqn8w1q9cf  uber-service.2      nigelpoulton/tu-demo:v2  ip-\
172-31-12-203  Running        Running about an hour ago
```

... you *should* get this.

```
$ docker service ps web-fe
ID          NAME      IMAGE             NODE  DESIRED  CURRENT
817f...f6z  web-fe.1  nigelpoulton/...  mgr2  Running  Running 5 mins
a1dh...mzn  web-fe.2  nigelpoulton/...  wrk1  Running  Running 5 mins
cc0j...ar0  web-fe.3  nigelpoulton/...  wrk2  Running  Running 5 mins
```

> For best results you might want to flip your reading device onto its side.

In doing this I've had to trim some of the output from some commands, but I don't think you're missing anything important. However, despite all of this, if you're reading on a small enough device, you're still going to get some wrapping :-(

But you didn't include something I really hoped you would...

I know the book doesn't cover *everything* about Docker. But it's not supposed to! I've written the book to get you up to speed as quickly as possible while still spending

the time to learn how it all fits together. If the book was 1,000 printed pages it **would not** help you get up to speed quickly!

However, I will add sections to the book if I think they're important and fundamental enough. Please use the book's feedback pages and hit me up on Twitter[3] with ideas of what you think should be included in the next version of the book.

Right, that's enough waffling. Let's crack on!

[3] https://twitter.com/nigelpoulton

Part 1: The general info stuff

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 comments 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!

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.

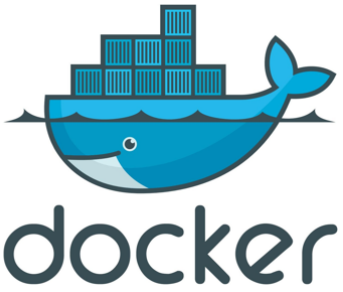


Figure 2.1 Docker, Inc. logo.

Interestingly, Docker, Inc. started its life as a platform as a service (PaaS) provider called *dotCloud*. Behind the scenes, the dotCloud platform leveraged Linux containers. 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.

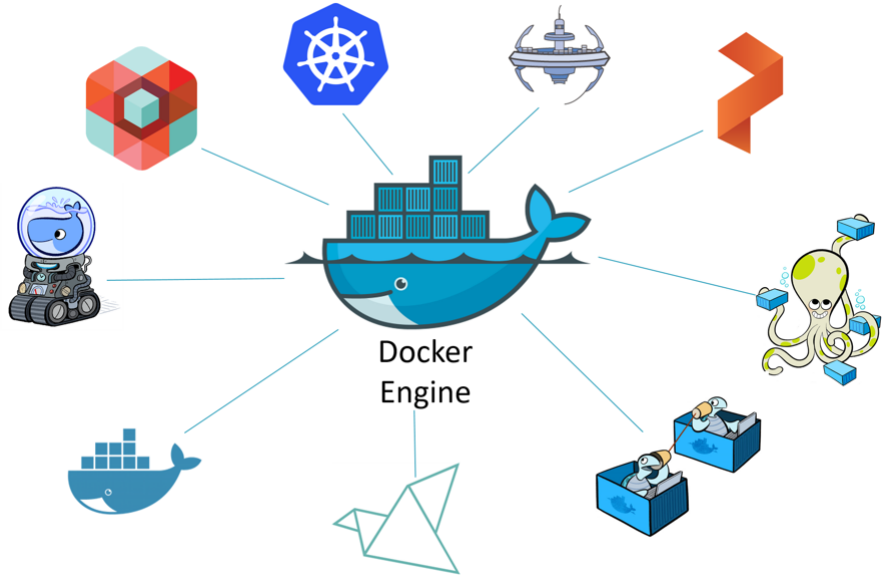


Figure 2.2

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 commercially 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*.

[4] 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 mentioning 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. So this is the version of history according to Nigel :-D

[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 according to Nigel* 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.

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.

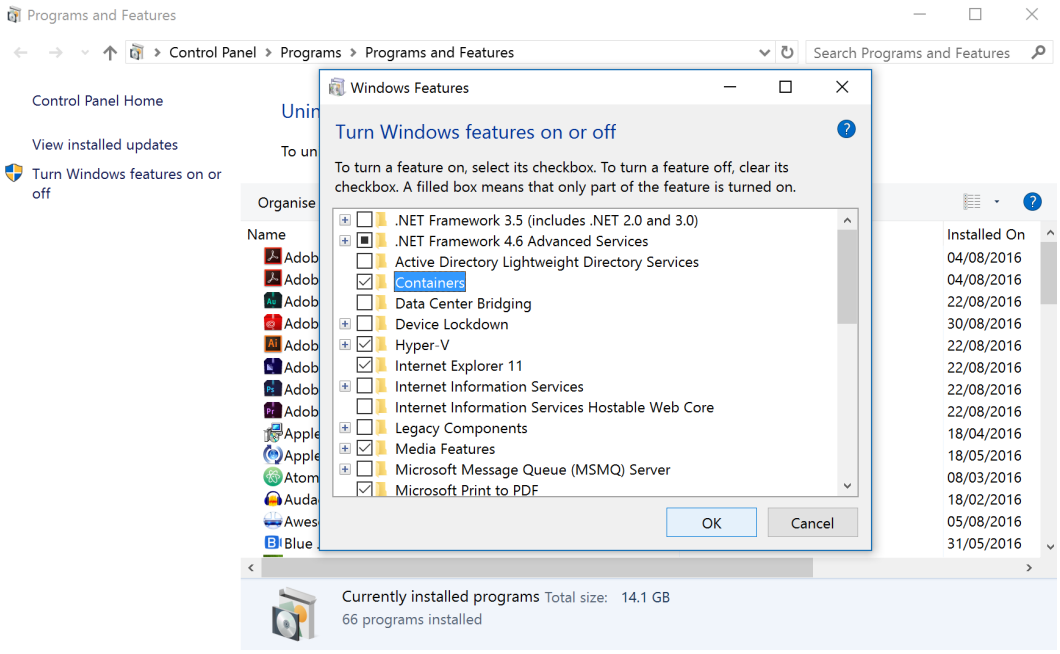


Figure 3.1

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\nigelpoulton> 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.

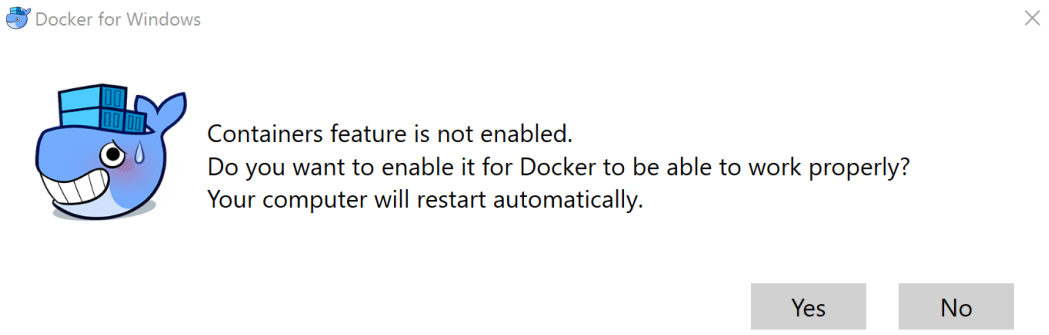


Figure 3.2

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\nigelpoulton> 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\nigelpoulton>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\nigelpoulton> docker --version
Docker version 1.12.1, build 23cf638, experimental
```

```
C:\Users\nigelpoulton> docker-compose --version
docker-compose version 1.8.0, build d988a55


C:\Users\nigelpoulton> 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 shows a high level representation of the *Docker for Mac* architecture.

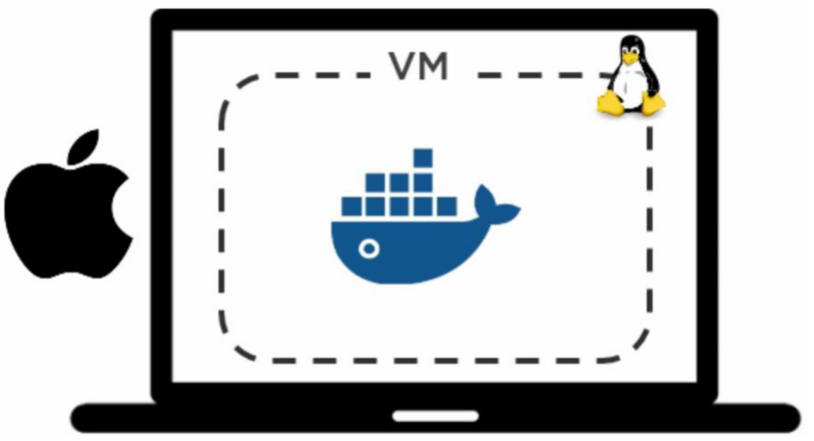


Figure 3.3

> **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/andhttps://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 npoulton
$
$ cat /etc/group | grep docker
docker:x:999:npoulton
```

   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:EJCI: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.

[14] 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.

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` 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...a2128ae95a60369c506dd6e6f6ab
Status: Downloaded newer image for ubuntu:latest
```

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

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

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.

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   NI ADDR SZ WCHAN  STIME TTY  TIME CMD
4 S root     1     0    0 -  4560 wait   13:38 ?    00:00:00 /bin/bash
0 R root     9     1    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.

Part 2: The technical stuff

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.

[16] https://hub.docker.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.

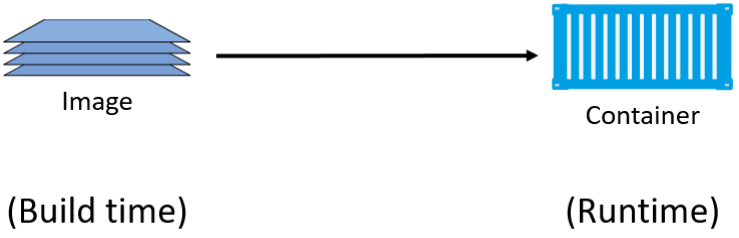


Figure 5.1

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
REPOSITORY  TAG      IMAGE ID       CREATED         SIZE
ubuntu      latest   bd3d4369aebc   12 days ago     126.6 MB
alpine      latest   4e38e38c8ce0   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.

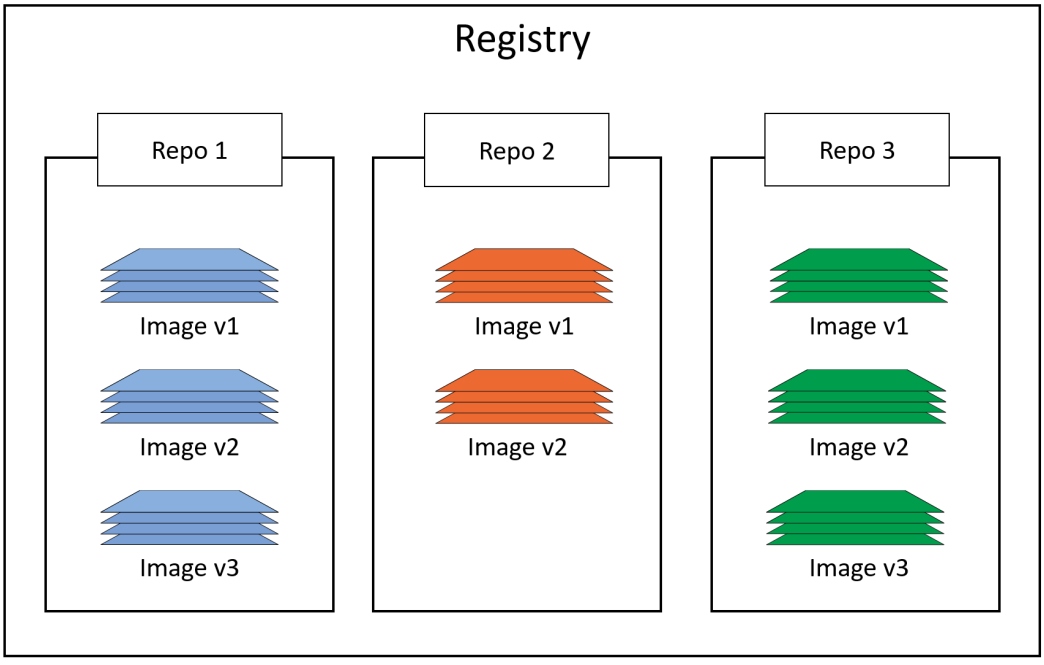


Figure 5.2

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:

- nigelpoulton/tu-demo - https://hub.docker.com/r/nigelpoulton/tu-demo/
- nigelpoulton/pluralsight-docker-ci - https://hub.docker.com/r/nigelpoulton/pluralsight-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 `nigelpoulton`.

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 `nigelpoulton`.

```
$ docker pull nigelpoulton/tu-demo:v2
//This will pull the image tagged 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 repository name with the DNS name of the registry. For example, if the image in the example above was in the Google Container Registry (GCR) you'd need to add `gcr.io` before the repository name as follows - `docker pull gcr.io/nigelpoulton/tu-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 nigelpoulton/tu-demo
latest: Pulling from nigelpoulton/tu-demo
237d5fcd25cf: Pull complete
a3ed95caeb02: Pull complete
<Snip>
Digest: sha256:42e34e546cee61adb1...3a0c5b53f324a9e1c1aae451e9
v1: Pulling from nigelpoulton/tu-demo
237d5fcd25cf: Already exists
a3ed95caeb02: Already exists
<Snip>
Digest: sha256:9ccc0c67e5c5eaae4b...624c1d5c80f2c9623cbcc9b59a
v2: Pulling from nigelpoulton/tu-demo
237d5fcd25cf: Already exists
a3ed95caeb02: Already exists
<Snip>
Digest: sha256:d3c0d8c9d5719d31b7...9fef58a7e038cf0ef2ba5eb74c
Status: Downloaded newer image for nigelpoulton/tu-demo
$
$ docker images
REPOSITORY            TAG       IMAGE ID      CREATED      SIZE
nigelpoulton/tu-demo   v2       6ac2...ad   4 months ago   211.6 MB
nigelpoulton/tu-demo   latest   9b91...29   4 months ago   211.6 MB
nigelpoulton/tu-demo   v1       9b91...29   4 months ago   211.6 MB
```

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.

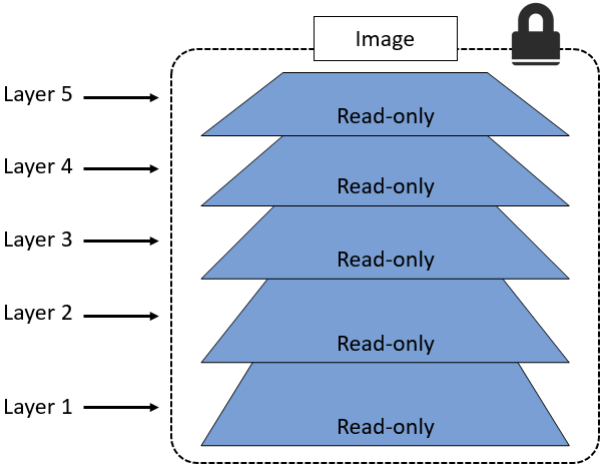


Figure 5.3

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.

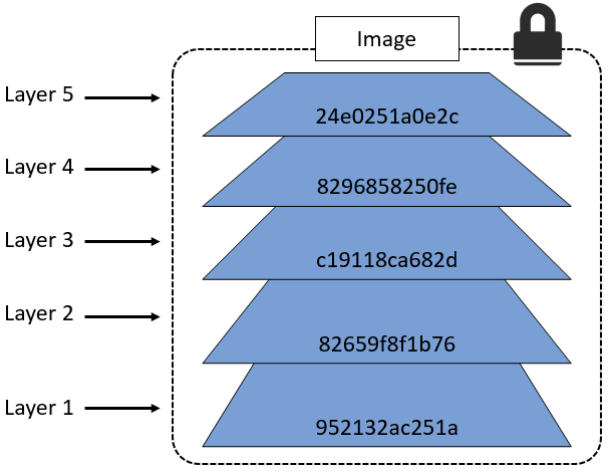


Figure 5.4

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:bd3d4369aebc4945be269859df0e15b1d32fefa2645f5699037d7d\
8c6b415a10",
        "RepoTags": [
            "ubuntu:latest"

        <Snip>

        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:c8a75145fc...894129005e461a43875a094b93412",
                "sha256:c6f2b330b6...7214ed6aac305dd03f70b95cdc610",
                "sha256:055757a193...3a9565d78962c7f368d5ac5984998",
                "sha256:4837348061...12695f548406ea77feb5074e195e3",
                "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.

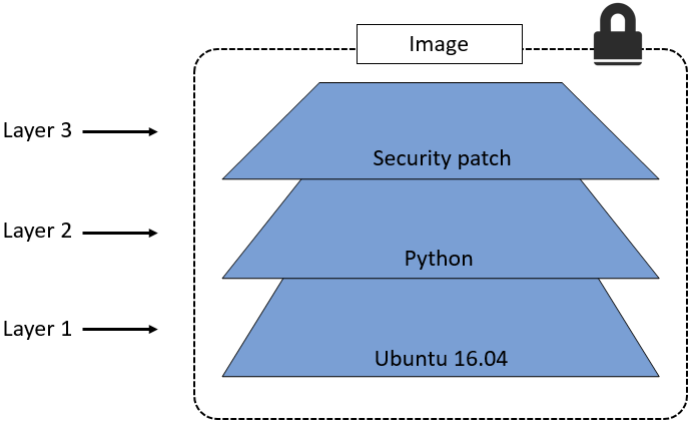


Figure 5.5

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.

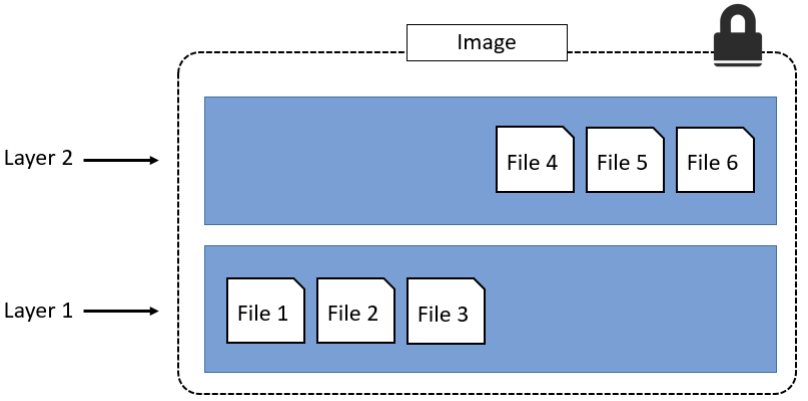


Figure 5.6

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.

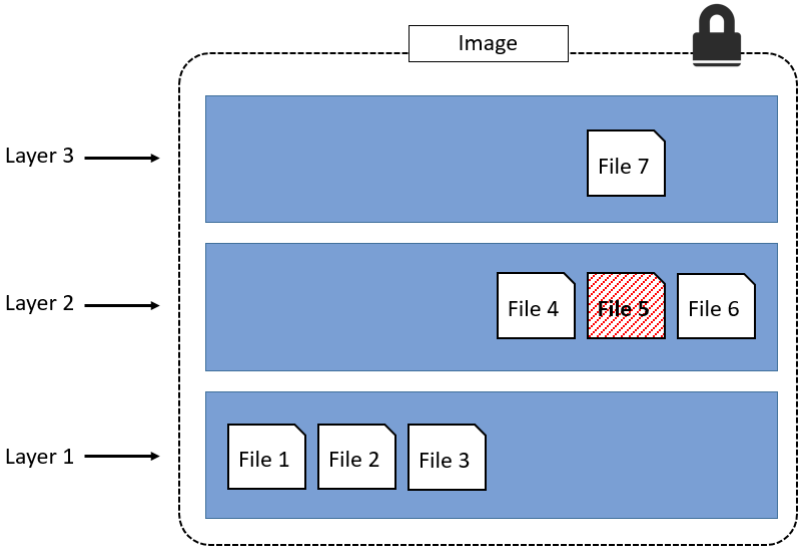


Figure 5.7

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 `nigelpoulton/tu-demo` repository.

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

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

```
v2: Pulling from nigelpoulton/tu-demo
237d5fcd25cf: Already exists
a3ed95caeb02: Already exists
<Snip>
eab5aaac65de: Pull complete
Digest: sha256:d3c0d8c9d5719d31b79c...fef58a7e038cf0ef2ba5eb74c
Status: Downloaded newer image for nigelpoulton/tu-demo
$
$ docker images
REPOSITORY              TAG       IMAGE ID       CREATED         SIZE
nigelpoulton/tu-demo    v2        6ac...ead    4 months ago    211.6 MB
nigelpoulton/tu-demo    latest    9b9...e29    4 months ago    211.6 MB
nigelpoulton/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
REPOSITORY  TAG     DIGEST              IMAGE ID      CREATED       SIZE
alpine      latest  sha256:3dcd...f73a  4e38e38c8ce0  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 `latest` 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.

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 containers.

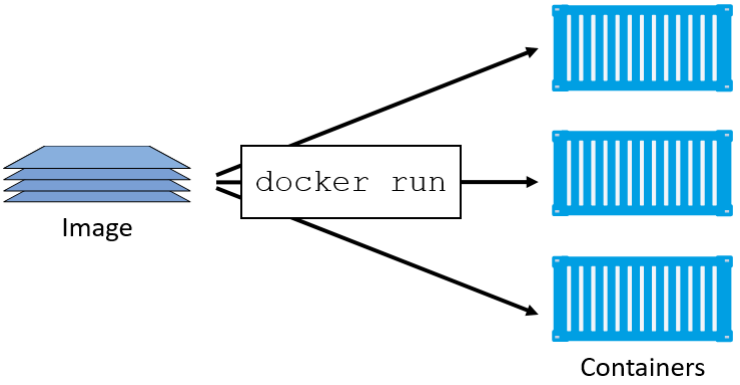


Figure 6.1

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.

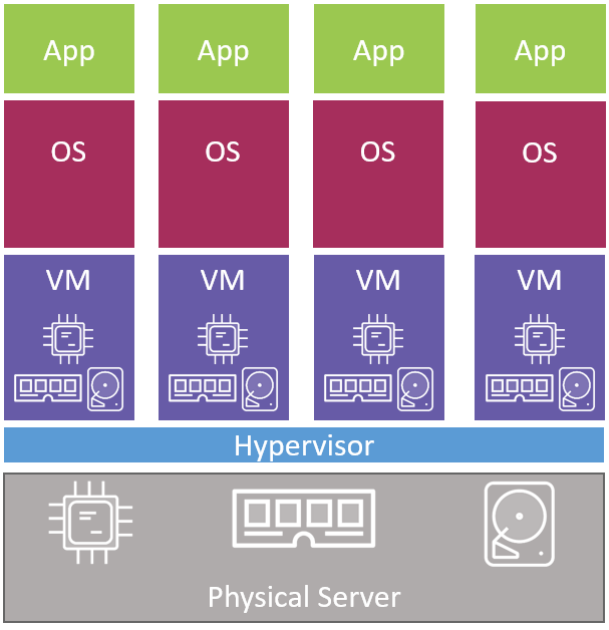


Figure 6.2

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.

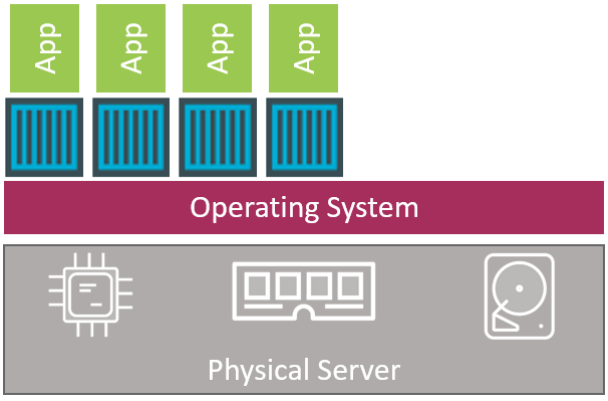


Figure 6.3

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      DisplayName
------    ----      -----------
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  NI ADDR SZ WCHAN  STIME TTY      TIME      CMD
4 S root    1     0   0 -  4558 wait   00:47 ?      00:00:00  /bin/bash
0 R root   11     1   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
CNTNR ID  IMAGE          COMMAND    CREATED  STATUS      NAMES
9cb...65  ubuntu:latest  /bin/bash  4 mins   Exited (0)  percy
```

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      CREATED  STATUS     NAMES
9cb2d2fd1d65  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 container vs 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 I use in a few of my Pluralsight video courses[17]. 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 \
  nigelpoulton/pluralsight-docker-ci

Unable to find image 'nigelpoulton/pluralsight-docker-ci:latest' locally
latest: Pulling from nigelpoulton/pluralsight-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 nigelpoulton/plur...docker-ci:latest
6efa1838cd51b92a4817e0e7483d103bf72a7ba7ffb5855080128d85043fef21
```

[17] https://app.pluralsight.com/library/search?q=nigel+poulton

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.

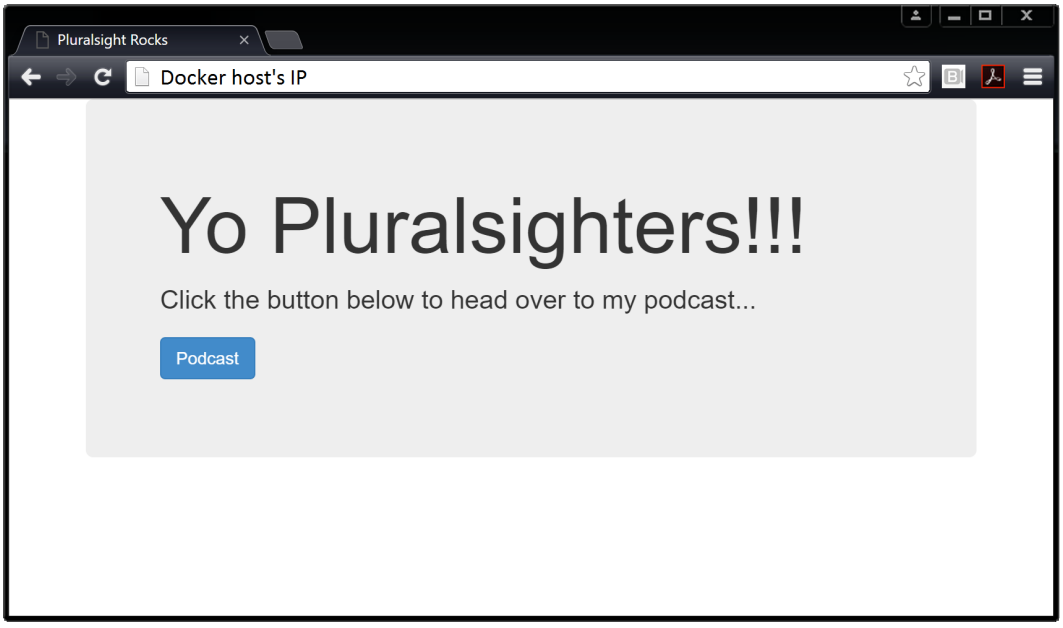


Figure 6.4

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 nigelpoulton/pluralsight-docker-ci

[
    {
        "Id": "sha256:07e574331ce3768f30305519...49214bf3020ee69bba1",
        "RepoTags": [
            "nigelpoulton/pluralsight-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 information 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.

7: Swarm mode

Now that we know how to install Docker, pull images, and work with containers, the next thing we need is a way to work with it all at scale. That's where orchestration and *swarm mode* comes into the picture.

As usual, we'll take a three-stage approach with a high level explanation at the top, followed by a longer section with all the detail and some examples, and we'll finish things up with a list of the main commands we learned.

Swarm mode - The TLDR

It's one thing to follow along with the simple examples in this book, but it's an entirely different thing running thousands of containers on tens or hundreds of Docker hosts! This is where orchestration comes to the rescue!

At a high level, orchestration is all about automating and simplifying the management of containerized applications at scale. Things like automatically rescheduling containers when nodes break, scaling things up when demand increases, and smoothly pushing updates and fixes into live production environments.

For the longest time orchestration like this was hard. Tools like *Docker Swarm* and *Kubernetes* were available, but they were complicated. Then along came Docker 1.12 and the new native *swarm mode* - and overnight things changed. *Swarm mode* made all this orchestration stuff a whole lot easier.

That's the quick explanation. Now let's get into the detail.

Swarm mode - The deep dive

First up, as the title of the chapter suggests, we're going to be focusing on *swarm mode* - the native clustering and orchestration technologies that first shipped as part Docker 1.12. Other orchestration solutions exist, but we're not covering those here.

Concepts and terminology

Swarm mode brought a load of changes and improvements to the way we manage containers at scale. At the heart of those changes is native clustering that's deeply integrated into the Docker Engine. We're not talking about something like Kubernetes that's a separate tool requiring a highly skilled specialist to configure it on top of existing Docker infrastructures. No! The clustering we're talking about here is a true first-class citizen in the Docker technology stack. And it's simple!

But the folks at Docker, Inc. don't really like using the term *cluster*. They're calling a cluster of orchestrated Docker engines a *swarm*, and the Docker engines participating in a *swarm* are said to operate in *swarm mode*. We'll try to be consistent and use these terms throughout the remainder of the book. We'll also start using the term *single-engine mode* to refer to Docker engines that are not running in *swarm mode*.

Figure 7.1 shows a 4-node *swarm* with nodes running in *swarm mode*, as well as two nodes not in the swarm operating in *single-engine mode*.

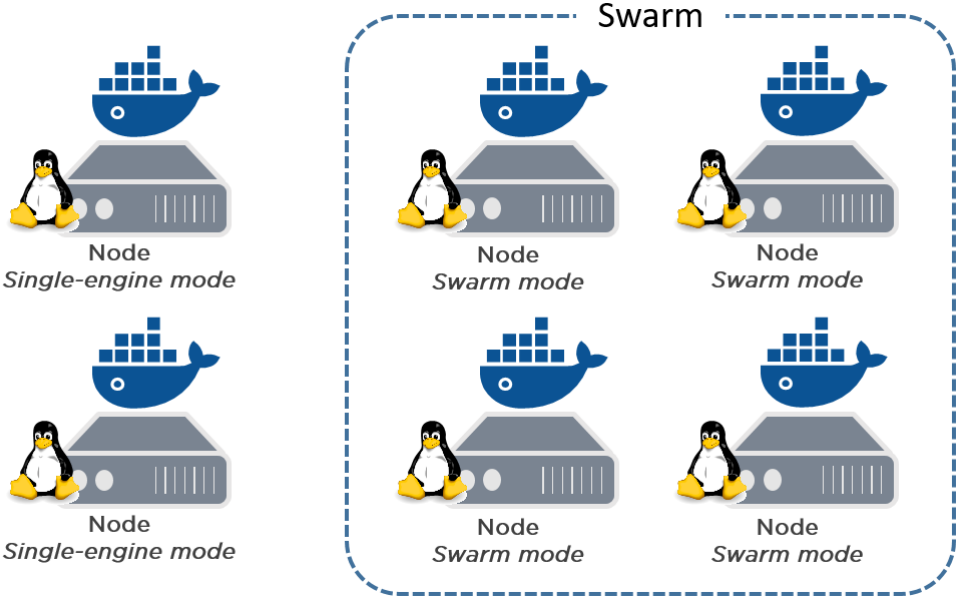


Figure 7.1

Backward compatibility

Introducing *swarm mode* was massively important for Docker, Inc. But so is maintaining backward compatibility! This led them to make **swarm mode entirely optional** in Docker 1.12. A standard installation of the Docker Engine would default to running in *single engine mode*, ensuring 100% backward compatibility with previous versions of Docker.

This is great news if you're a user or developer of 3rd party clustering tools and the likes. As long as you keep Docker 1.12 and later in *single-engine mode*, all of your existing tools and apps will work as normal! However, as soon as you take the plunge and put your Docker Engine into *swarm mode* you risk breaking those 3rd party tools and apps.

In short, putting a Docker Engine into *swarm mode* gives you all of the latest orchestration goodness, it just comes at the price of backward compatibility.

Swarm mode primer

Let's take a minute or two to explain the major components and constructs in a *swarm*.

A *swarm* consists of one or more *nodes*. These can be physical servers, VMs, or cloud instances. The only requirement is that all nodes in a *swarm* can communicate with each other over reliable networks.

Nodes are then configured as *managers* or *workers*. *Managers* look after the state of the cluster and are in charge of dispatching tasks to *workers*. *Workers* accept tasks from *managers* and execute them.

When talking about *tasks* in the context of a *swarm*, we mean containers. So when we say "managers dispatch *tasks* to workers", we're saying they dispatch container workloads. You might also hear them referred to as *replicas*. This might be confusing at this point, so try and remember that *tasks* and *replicas* are words that mean *containers*.

The next thing we need to know about is *services*. *Services* are the main construct we run on a *swarm*. And all a service is, is a declarative way of setting a *desired state* for a set of tasks (containers). But it's hugely powerful. The ability to set a *desired state* for things like the following is game changing:

- Set the number of tasks that make up a service
- Set the image the containers in the service will use
- Set the procedure for updating to newer versions of the image

The configuration and state of the *swarm* is held in a distributed *etcd* database located on all managers in the swarm. It's kept extremely up-to-date and is hosted in-memory on all *manager nodes* to make it fast. But the best thing about it is the fact that it requires zero configuration - it just takes care of itself.

Something else that's game changing about *swarm mode* is its approach to security. TLS is so tightly integrated that it's not possible to build a swarm without it. In today's security conscious world, things like this deserve all the props they get! Anyway, *swarm mode* uses TLS to encrypt communications, authenticate nodes, and authorize roles. Automatic key rotation is also thrown in as the icing on the cake! And it all happens so smoothly that you wouldn't even know it was there!

That's enough of a primer. Let's get our hands dirty with some examples.

Lab setup

For the remainder of this chapter we'll build the lab shown in Figure 7.2 with 6-nodes configured as 3 managers and 3 workers. Each node is running Linux with Docker 1.12 or higher. All nodes in the lab can communicate over the network.

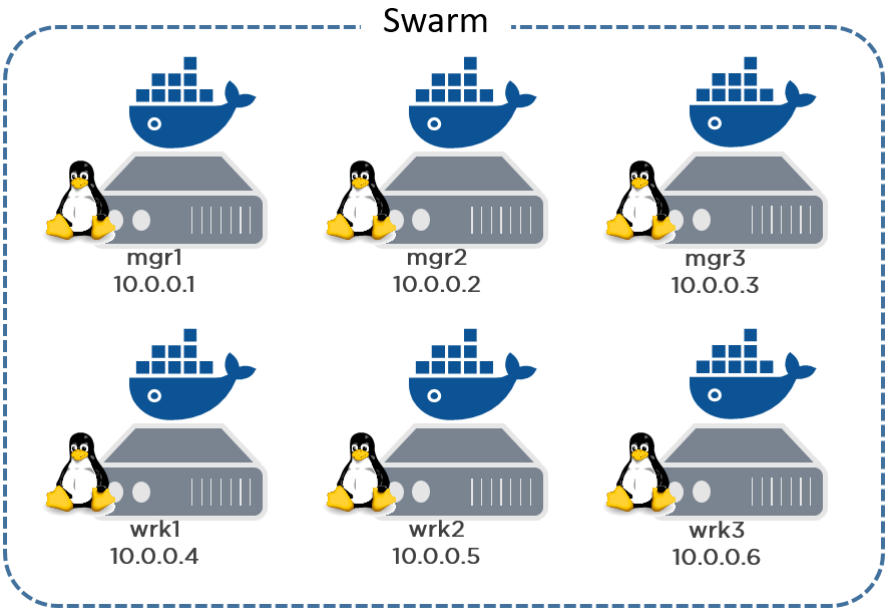


Figure 7.2

The names and IP addresses are not important and can be different in your lab. If you are following along with the examples, just remember to substitute them with your own.

Enabling swarm mode

Running `docker swarm init` on a Docker host operating in *single-engine mode* will switch that node into *swarm mode* and create a new *swarm*. It will also make the node a *manager*.

Additional nodes can then be joined to the swarm as workers and managers using the `docker swarm join` command. This also puts those nodes into *swarm mode* as part of the operation.

The following steps will put **mgr1** into *swarm mode* and initialize a new swarm. It will then join **wrk1**, **wrk2**, and **wrk3** as worker nodes - automatically putting them into *swarm mode*. Finally, it will add **mgr2** and **mgr3** as additional managers and switch them into *swarm mode*. At the end of the procedure all 6 nodes will be part of the same swarm and will all be operating in *swarm mode*.

1. Log on to **mgr1** and initialize a new swarm.

```
$ docker swarm init \
  --advertise-addr 10.0.0.1:2377 \
  --listen-addr 10.0.0.1:2377

Swarm initialized: current node (d21lyzn0v5qvgdyfzec79qzkx) is now a manager.
```

   The command can be broken down as follows:
   - `docker swarm init` tells Docker to start a new cluster and make this node the first manager. It also enables swarm mode on the node.
   - `--advertise-addr` is the IP and port that other nodes should use to connect to this manager. The flag is optional, but it gives you control over which IP gets used on nodes with multiple IPs. It also gives you the chance to specify an IP address that does not exist on the node, such as a load balancer IP address.
   - `--listen-addr` lets you specify which IP and port you want to listen on for swarm traffic. This will usually match the `--advertise-addr`, but is useful in situations where you want to restrict swarm to a particular IP on a system with multiple IPs. It's also required in situations where the `--advertise-addr` refers to a remote IP address like a load balancer.

   I recommend you always use both flags.

   The default port that swarm mode operates on is **2377**. This is entirely customizable, but Docker, Inc. are looking to register this with IANA as the official Docker swarm port.
2. List the nodes in the swarm

```
$ docker node ls
ID            HOSTNAME   STATUS  AVAILABILITY  MANAGER STATUS
d21...qzkx *  mgr1       Ready   Active        Leader
```

   Notice that **mgr1** is currently the only node in the swarm and is listed as the *Leader*. We'll come back to this in a second.
3. From **magr1** run the `docker swarm join-token` command to extract the commands and tokens required to add new workers and managers to the swarm.

```
$ docker swarm join-token worker
To add a manager to this swarm, run the following command:
   docker swarm join \
   --token SWMTKN-1-0uahebax...c87tu8dx2c \
   10.0.0.1:2377

$ docker swarm join-token manager
To add a manager to this swarm, run the following command:
   docker swarm join \
   --token SWMTKN-1-0uahebax...ue4hv6ps3p \
   10.0.0.1:2377
```

Notice that the commands to join a worker and a manager are identical apart from the join tokens (`SWMTKN`). This means that whether a node joins as a worker or a manager depends entirely on which token you use when joining it.

4. Log on to **wrk1** and join it to the swarm using the `docker swarm join` command with the token used for joining workers.

```
$ docker swarm join \
    --token SWMTKN-1-0uahebax...c87tu8dx2c \
    10.0.0.1:2377 \
    --advertise-addr 10.0.0.4:2377 \
    --listen-addr 10.0.0.4:2377

This node joined a swarm as a worker.
```

I've manually added the `--advertise-addr`, and `--listen-addr` flags as I consider it best practice to be as specific as possible when it comes to network configuration.

5. Repeat the previous step on **wrk2** and **wrk3** to join them to the swarm as workers. Make sure you use **wrk2** and **wrk3's** own IP addresses for the `--advertise-addr` and `--listen-addr` flags.
6. Log on to **mgr2** and join it to the swarm as a manager using the `docker swarm join` command with the token used for joining managers.

```
$ docker swarm join \
    --token SWMTKN-1-0uahebax...ue4hv6ps3p \
    10.0.0.1:2377 \
    --advertise-addr 10.0.0.2:2377 \
    --listen-addr 10.0.0.1:2377

This node joined a swarm as a manager.
```

7. Repeat the previous step on **mgr3** remembering to use **mgr3's** IP address for the `advertise-addr` and `--listen-addr` flags.
8. List the nodes in the swarm by running `docker node ls` from any of the manager nodes in the swarm.

```
$ docker node ls
ID                HOSTNAME     STATUS  AVAILABILITY  MANAGER STATUS
0g4rl...babl8 *   mgr2         Ready   Active        Reachable
2xlti...l0nyp     mgr3         Ready   Active        Reachable
8yv0b...wmr67     wrk1         Ready   Active
9mzwf...e4m4n     wrk3         Ready   Active
d21ly...9qzkx     mgr1         Ready   Active        Leader
e62gf...l5wt6     wrk2         Ready   Active
```

Congratulations! You've just created a 6-node swarm with 3 managers and 3 workers. As part of the process you put the Docker Engine on each node into *swarm mode*. As an added bonus, the *swarm* is automatically secured with TLS.

If you look in the `MANAGER STATUS` column in the previous output you'll see that the three manager nodes are showing as either "Reachable" or "Leader". We'll learn more about leaders shortly. Nodes with nothing in the `MANAGER STATUS` column are *workers*. Also note the asterisk (*) after the ID on the line showing **mgr2**. This shows us which node we ran the `docker node ls` command from. In this instance the command was issued from **mgr2**.

> **Note:** It's a pain to specify the `--advertise-addr` and `--listen-addr` flags every time you join a node to the swarm. However, it can be even more of a pain if you get the network configuration of your swarm wrong. Manually adding nodes to a swarm is unlikely to be a daily task

> so I think it's worth the extra up-front effort to use the flags. It's your choice though. In lab environments or nodes with only a single IP you do not need to use the flags.

Now that we have a *swarm* up and running, let's take a look at manager high availability.

Swarm manager high availability (HA)

So far we've added three manager nodes to a swarm. Why did we add three and how do they work together? We'll answer all of this, plus more in this section.

Swarm *managers* have native support for high availability (HA). This means that one or more can fail and the survivors will keep the swarm running.

Technically speaking, swarm mode implements a form of active-passive multi-manager HA. This means that although you might - and should - have multiple *managers*, only one of them is ever considered *active*. We call this active manager the *leader*. And the leader's the only one that will ever issue live commands against the *swarm* such as changing the configuration of the swarm or issuing tasks to workers. If a non-active manager receives commands for the swarm it'll proxy them across to the leader.

This process is shown in Figure 7.3 where step `1` is the command coming in to a *manager* from a remote Docker client. Step `2` is the non-leader manager proxying the command to the leader. Step `3` is the leader pushing that command to the relevant node in the swarm.

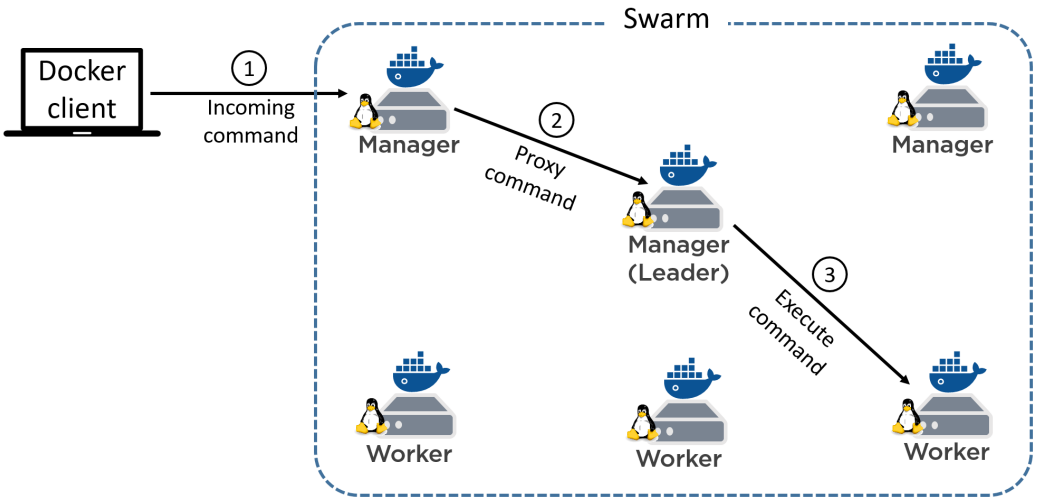


Figure 7.3

Swarm uses an implementation of the Raft consensus algorithm[18] to power manager HA, and the following two best practices apply:

1. Deploy an odd number of managers.
2. Don't deploy too many managers

Having an odd number of *managers* increases the chance of reaching quorum and avoiding a split-brain. For example, if you had 4 managers and the network partitioned, you could be left with two managers on each side of the partition. This is known as a split brain - each side knows there used to be 4 but can now only see 2. Neither side has a way of knowing if the two it can no longer see are still alive and which side holds the majority share (quorum). However, if you had 3 or 5 managers and the same network partition occurred, it would be impossible to have the same number of managers on both sides of the split. This means that one side would have a far better chance of knowing if it had more or less than the other side and achieving quorum.

[18] https://raft.github.io/

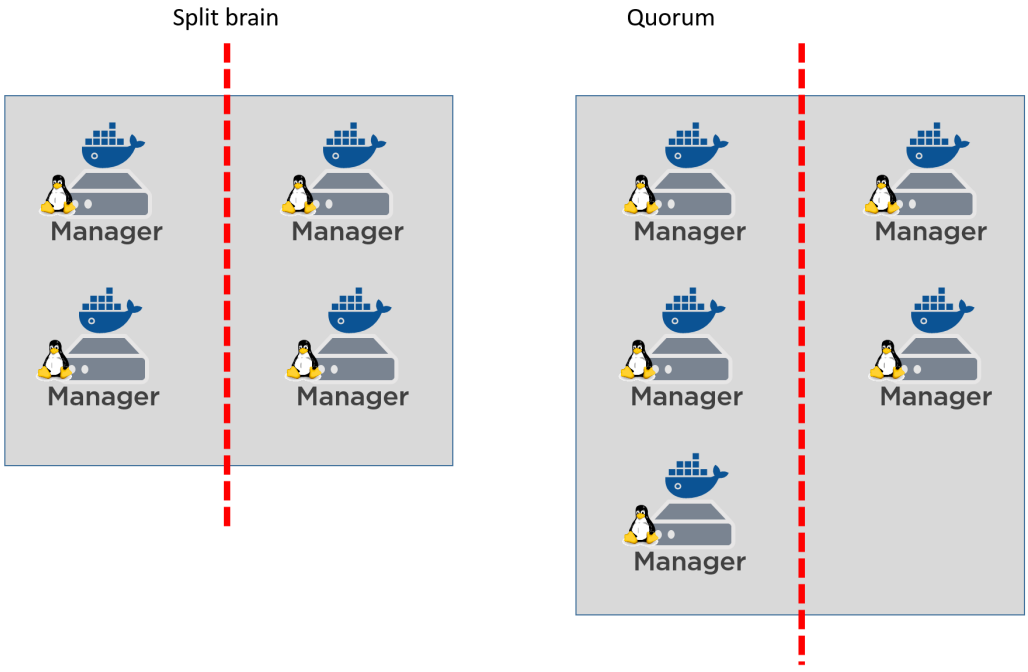


Figure 7.4

As with all consensus algorithms - more participants means more time required to achieve consensus. It's like deciding where to eat - it's always quicker and easier for 3 people to decide than it is for 33! With this in mind, it's a best practice to have either 3 or 5 managers for HA. 7 might work, but it's generally accepted that 3 or 5 is optimal. You definitely don't want more than 7 as the time taken to achieve consensus will be longer.

A final word of caution regarding manager HA. While it's obviously a good practice to spread your managers across availability zones within your network, you need to make sure that the networks connecting them are reliable! Network partitions can be a royal pain in the backside! This means, at the time of writing, the nirvana of hosting your active production applications and infrastructure across multiple cloud providers such as AWS and Azure is a bit of a daydream. Take time to make sure your managers are connected via high speed reliable networks!

Now that we've got our *swarm* built and understand the concepts of *leaders* and *manager HA*, let's move on to *services*.

Services

Like we said in the *Swarm primer*... services are a new construct introduced with Docker 1.12 that only exist in *swarm mode*.

They let us *declare* the *desired state* for a group of containers (tasks) and feed that to Docker. For example, assume you've got an app that has a web front-end. You have an image for the web server, and testing has shown that you will need 5 instances of the web service to handle normal daily traffic. You would translate this requirement into a *service* declaring the image the containers should use, and that the service should always have 5 running tasks.

We'll see some of the other things that can be declared as part of a service in a minute, but before we do that, let's see how to create the one we just described.

We create a service with the `docker service create` command.

```
$ docker service create --name web-fe \
   -p 8080:8080 \
   --replicas 5 \
   nigelpoulton/pluralsight-docker-ci

2kffzpz721nrjikmxqhj474qg
```

Let's review that command and output.

We used `docker service create` to tell Docker we are declaring a new service, and we used the `--name` flag to name the service **web-fe**. We told Docker to map port 8080 on every node in the swarm to 8080 inside of each task or container in the service. Next we used the `--replicas` flag to tell Docker that there should always be 5 tasks/containers in the service. Finally we told Docker which image to use for all tasks and containers - it's important to understand that all tasks in a service use the same image and config!

After we hit <Return>, the manager acting as leader instantiated 5 tasks across the *swarm* - remember that managers also act as workers. Each worker or manager then pulled the image and started a container from it running on port 8080. The swarm leader also ensured a copy the service's desired state was replicated to every manager in the swarm.

But this isn't the end. All *services* are constantly monitored by the swarm - the *swarm* runs a reconciliation loop that constantly compares the *actual state* of the service to the *desired state*. If the two states match, the world is a happy place and no further actions need taking. If they don't match, the swarm takes actions so that they do. Put another way, the swarm is constantly making sure that *actual state* matches *desired state*.

As an example, if one of the *workers* hosting one of the 5 **web-fe** container tasks fails, the *actual state* for the **web-fe** service will drop from 5 running tasks to 4. This will no longer match the *desired state* of 5, and Docker will start a new **web-fe** task to bring *actual state* back in line with *desired state*. This behavior is very powerful and allows the service to self-heal in the event of node failures and the likes.

Viewing and inspecting services

You can use the `docker service ls` command to see a list of all services running on a swarm.

```
$ docker service ls
ID            NAME    REPLICAS  IMAGE                          COMMAND
2kffzpz721nr  web-fe  5/5       nigelpoulton/plur...cker-ci
```

The output above shows a single running service as well as some basic information about state. Among other things, we can see the name of the service and that 5 out of the 5 desired tasks/replicas are in the running state. If you run this command soon after deploying the service it might not show all tasks/replicas as running. This is probably because of the time it takes to pull the image on each node.

You can use the `docker service ps` command to see a list of tasks in a service and their state.

```
$ docker service ps web-fe
ID         NAME      IMAGE              NODE  DESIRED  CURRENT
817...f6z  web-fe.1  nigelpoulton/...   mgr2  Running  Running 2 mins
a1d...mzn  web-fe.2  nigelpoulton/...   wrk1  Running  Running 2 mins
cc0...ar0  web-fe.3  nigelpoulton/...   wrk2  Running  Running 2 mins
6f0...azu  web-fe.4  nigelpoulton/...   mgr3  Running  Running 2 mins
dyl...p3e  web-fe.5  nigelpoulton/...   mgr1  Running  Running 2 mins
```

The format of the command is `docker service ps <service-name or service-id>`. The output displays each task on its own line, shows which node in the swarm it's executing on, and shows desired state and actual state.

For detailed information about a service, use the `docker service inspect` command.

```
$ docker service inspect --pretty web-fe
ID:             2kffzpz721nrjikmxqhj474qg
Name:           web-fe
Mode:           Replicated
 Replicas:      5
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
ContainerSpec:
 Image:         nigelpoulton/pluralsight-docker-ci
Resources:
Ports:
 Protocol = tcp
 TargetPort = 8080
 PublishedPort = 8080
```

The example above uses the `--pretty` flag to limit the output to the most interesting items printed in an easy-to-read format. Leaving off the `--pretty` flag will give a more verbose output.

We'll come back to some of these outputs later.

Let's go and see how to scale a service.

Scaling a service

Another powerful feature of *services* is the ability to easily scale them up and down.

Let's assume business is booming and we're seeing double the amount of anticipated traffic hitting the web front-end. Fortunately scaling the **web-fe** service is as simple as running the `docker service scale` command.

```
$ docker service scale web-fe=10
web-fe scaled to 10
```

The above command will scale the number of tasks/replicas from 5 to 10. In the background it's updating the service's *desired state* from 5 to 10. Run another `docker service ls` command to verify the operation was successful.

```
$ docker service ls
2kffzpz721nr  web-fe  10/10     nigelpoulton/pluralsight-docker-ci
```

Running a `docker service ps` command will show that the tasks in the service are balanced across all nodes in the swarm as evenly as possible.

```
$ docker service ps web-fe
ID          NAME      IMAGE              NODE  DESIRED  CURRENT
817...f6z  web-fe.1  nigelpoulton/...  mgr2  Running  Running 5 mins
a1d...mzn  web-fe.2  nigelpoulton/...  wrk1  Running  Running 5 mins
cc0...ar0  web-fe.3  nigelpoulton/...  wrk2  Running  Running 5 mins
6f0...azu  web-fe.4  nigelpoulton/...  mgr3  Running  Running 5 mins
dyl...p3e  web-fe.5  nigelpoulton/...  mgr1  Running  Running 5 mins
912...vtb  web-fe.6  nigelpoulton/...  mgr1  Running  Running 1 min
3wu...o7y  web-fe.7  nigelpoulton/...  wrk3  Running  Running 1 min
aso...6hh  web-fe.8  nigelpoulton/...  wrk3  Running  Running 1 min
97u...4bn  web-fe.9  nigelpoulton/...  wrk1  Running  Running 1 min
a1u...4jj  web-fe.10 nigelpoulton/...  mgr2  Running  Running 1 min
```

Behind the scenes, swarm-mode runs a scheduling algorithm that defaults to trying to balance tasks as evenly as possible across the nodes in the swarm. At the time of

writing, this amounts to running an equal number of tasks on each node without taking into consideration things like CPU load etc.

Run another `docker service scale` command to bring the number back down from 10 to 5.

```
$ docker service scale web-fe=5
web-fe scaled to 5
```

Now that we know how to scale a service, let's see how we remove one.

Removing a service

Removing a service is simple - may be too simple.

The following `docker service rm` command will delete the service we deployed earlier.

```
$ docker service rm web-fe
web-fe
```

Confirm the service is gone with the `docker service ls` command.

```
$ docker service ls
ID  NAME  REPLICAS  IMAGE  COMMAND
```

Be careful using the `docker service rm` command as it deletes all tasks in a service without asking for confirmation.

Now that the service is deleted from the system, let's go and look at how to push rolling updates to a service.

Rolling updates

Pushing updates to deployed applications is a fact of life. And for the longest time it's been really painful. I've lost more than enough weekends to major application updates, and I've no intention of going there again if I can help it.

Well… thanks to Docker *services*, pushing updates to well designed apps just got a whole lot easier!

To see this, we're going to deploy a new service. But before we do that we're going to create a new overlay network for the service. This isn't necessary, but I wanted you to see how it was done and how the service uses it.

```
$ docker network create -d overlay uber-net
43wfp6pzea470et4d57udn9ws
```

This creates a new overlay network called "uber-net" that we'll be able to leverage with the service we're about to create. An overlay network essentially creates a new layer 2 network that we can place containers on, and all containers on it will be able to communicate with each other. This works even if the Docker hosts they're running on are on different underlying networks. Basically the overlay network creates a new layer 2 container network on top of potentially multiple different underlying networks.

Figure 7.5 shows two underlay networks connected by a layer 3 router. There is then a single overlay network across both of them. Docker hosts are connected to the two underlay networks and containers are connected to the overlay. All containers on the overlay can communicate with each other even if they are running on Docker hosts plumbed into different underlay networks.

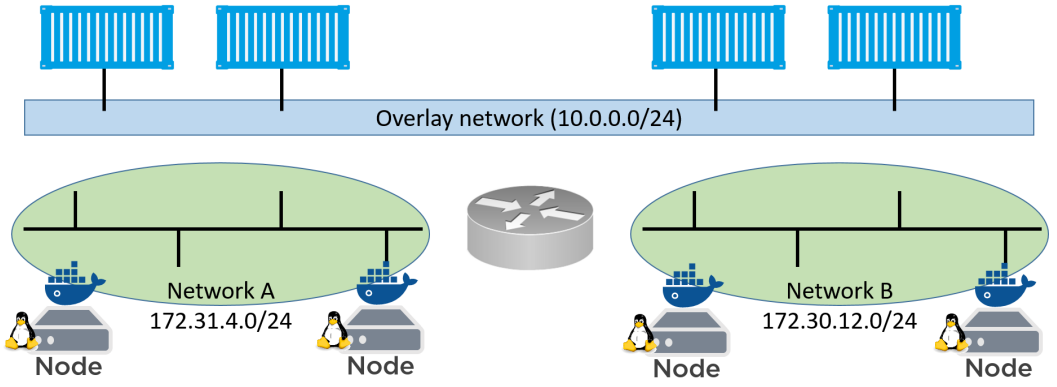


Figure 7.5

Run a `docker network ls` to verify that the network created properly and is visible on the Docker host.

```
$ docker network ls
NETWORK ID          NAME                DRIVER      SCOPE
490e2496e06b        bridge              bridge      local
a0559dd7bb08        docker_gwbridge     bridge      local
a856a8ad9930        host                host        local
1ailuc6rgcnr        ingress             overlay     swarm
be581cd6de9b        none                null        local
43wfp6pzea47        uber-net            overlay     swarm
```

The `uber-net` network was successfully created with the `swarm` scope and is currently only visible on manager nodes in the swarm.

Let's go and create a new service.

```
$ docker service create --name uber-svc \
   --network uber-net \
   -p 80:80 --replicas 12 \
   nigelpoulton/tu-demo:v1

dhbtgvqrg2q4sg07ttfuhg8nz
```

Let's see what we just declared with that `docker service create` command.

The first thing we did was name the service and then use the `--network` flag to tell it to place all containers on the new `uber-net` network. We then exposed port 80 across the entire swarm and mapped it to port 80 inside of each of the 12 replicas or tasks we asked it to run. Finally we told it to base all tasks on the nigelpoulton/tu-demo:v1 image.

Run a `docker service ls` and a `docker service ps` command to verify the state of the new service.

```
$ docker service ls
ID            NAME      REPLICAS  IMAGE
dhbtgvqrg2q4  uber-svc  12/12     nigelpoulton/tu-demo:v1
$
$ docker service ps uber-svc
ID        NAME          IMAGE                NODE  DESIRED   CURRENT STATE
0v...7e5  uber-svc.1    nigelpoulton/...:v1  wrk3  Running   Running 1 min
bh...wa0  uber-svc.2    nigelpoulton/...:v1  wrk2  Running   Running 1 min
23...u97  uber-svc.3    nigelpoulton/...:v1  wrk2  Running   Running 1 min
82...5y1  uber-svc.4    nigelpoulton/...:v1  mgr2  Running   Running 1 min
c3...gny  uber-svc.5    nigelpoulton/...:v1  wrk3  Running   Running 1 min
e6...3u0  uber-svc.6    nigelpoulton/...:v1  wrk1  Running   Running 1 min
78...r7z  uber-svc.7    nigelpoulton/...:v1  wrk1  Running   Running 1 min
2m...kdz  uber-svc.8    nigelpoulton/...:v1  mgr3  Running   Running 1 min
b9...k7w  uber-svc.9    nigelpoulton/...:v1  mgr3  Running   Running 1 min
ag...v16  uber-svc.10   nigelpoulton/...:v1  mgr2  Running   Running 1 min
e6...dfk  uber-svc.11   nigelpoulton/...:v1  mgr1  Running   Running 1 min
e2...k1j  uber-svc.12   nigelpoulton/...:v1  mgr1  Running   Running 1 min
```

Passing the service the `-p 80:80` flag will ensure that a swarm-wide mapping is created that maps traffic coming in to any node in the swarm on port 80 through to port 80 inside of any container in the service.

Open a web browser and point it to the IP address of any of the nodes in the swarm on port 80 to see the app running in the service.

Figure 7.6

As you can see, the application is a simple voting application that will register votes for either "football" or "soccer". Feel free to point you web browser to other nodes in the swarm. You will be able to reach the web server from any node in the swam because the `-p 80:80` creates a mapping on every host. This is true even on nodes that might be running a task for the service - **every node gets a mapping and can therefore redirect your request to a node that runs the service**.

Now let's assume that this particular vote has come to an end and your company is now running a new poll. A new image has been created for the new poll and has been added to the same Docker Hub repository, but this one is tagged as `v2` instead of `v1`.

Let's also assume that you've been tasked with pushing the updated image to the swam in a staged manner - 2 containers at a time with a 20 second delay in between each batch of 2. We can use the following `docker service update` command to accomplish this.

```
$ docker service update \
   --image nigelpoulton/tu-demo:v2 \
   --update-parallelism 2 \
   --update-delay 20s uber-svc

uber-svc
```

Let's review the command. `docker service update` lets us make updates to running services by updating the service's desired state. This time we gave it a new image tag `v2` instead of `v1`. And we used the `--update-parallelism` and the `--upate-dealy` flags to make sure that the new image was pushed to 2 tasks at a time with a 20 second cool-off period in between each pair. Finally we told Docker to make these changes to the `uber-svc` service.

If we run a `docker service ps` against the service we'll see that some of the tasks in the service are at `v2` while some are at `v1`. If we give the operation enough time to complete (4 minutes) all tasks will eventually reach the new desired state of using the `v2` image.

```
$ docker service ps uber-svc
ID        NAME              IMAGE        NODE   DESIRED   CURRENT STATE
7z...nys  uber-svc.1        nigel...v2   mgr2   Running   Running 13 secs
0v...7e5  \_uber-svc.1      nigel...v1   wrk3   Shutdown  Shutdown 13 secs
bh...wa0  uber-svc.2        nigel...v1   wrk2   Running   Running 1 min
e3...gr2  uber-svc.3        nigel...v2   wrk2   Running   Running 13 secs
23...u97  \_uber-svc.3      nigel...v1   wrk2   Shutdown  Shutdown 13 secs
82...5y1  uber-svc.4        nigel...v1   mgr2   Running   Running 1 min
c3...gny  uber-svc.5        nigel...v1   wrk3   Running   Running 1 min
e6...3u0  uber-svc.6        nigel...v1   wrk1   Running   Running 1 min
78...r7z  uber-svc.7        nigel...v1   wrk1   Running   Running 1 min
2m...kdz  uber-svc.8        nigel...v1   mgr3   Running   Running 1 min
b9...k7w  uber-svc.9        nigel...v1   mgr3   Running   Running 1 min
ag...v16  uber-svc.10       nigel...v1   mgr2   Running   Running 1 min
e6...dfk  uber-svc.11       nigel...v1   mgr1   Running   Running 1 min
e2...k1j  uber-svc.12       nigel...v1   mgr1   Running   Running 1 min
```

You can witness the update happening in real-time by opening a web browser to any node in the swarm and hitting refresh several times. Some of the requests will

be serviced by containers running the old version and some will be serviced by containers running the new version. After enough time all requests will be service by containers running the updated copy of the service.

Congratulations. You've just pushed a rolling update to a live containerized application.

If you run a `docker inspect --pretty` command against the service you'll see the update parallelism and update delay settings you just used are now part of the service definition. This means future updates that you push will automatically use these settings unless you override them as part of the `docker service update` command.

```
$ docker service inspect --pretty uber-svc
ID:             dhbtgvqrg2q4sg07ttfuhg8nz
Name:           uber-svc
Mode:           Replicated
 Replicas:      12
Update status:
 State:         completed
 Started:       11 minutes ago
 Completed:     8 minutes ago
 Message:       update completed
Placement:
UpdateConfig:
 Parallelism:   2
 Delay:         20s
 On failure:    pause
ContainerSpec:
 Image:         nigelpoulton/tu-demo:v2
Resources:
Networks: 43wfp6pzea470et4d57udn9ws
Ports:
 Protocol = tcp
 TargetPort = 80
 PublishedPort = 80
```

You should also note a couple of things about the service's network config. All nodes in the swarm that are running a task for the service will have the `uber-net` overlay

network that we created earlier. We can verify this by running `docker network ls` on any node running a task.

You should also note the `Networks` portion of the `docker inspect` output above. This shows the `43wfp6pzea470et4d57udn9ws uber-net` network as well as the swarm-wide `80:80` port mapping.

The future of services

Services are still relatively new to Docker, but they're massively strategic! This means we should expect to see significant development around them.

A couple of those developments will more than likely be; the ability to define a service in a manifest file, and to run more than just container workloads as part of services.

On the topic of manifest files. In this chapter we've shown you how to declare a service using the `docker service create` command and passing it a lot of flags and options. In the future we should expect to be able to pass the command a JSON or YAML file that holds the entire service declaration. This will allow us to keep a repository of service definition files, version control them, and easily pass them to Docker to instantiate new services. Expect this very soon.

In the more distant future we may even see non-container workloads running under the auspices of services. We said earlier in the chapter that service *tasks* = containers. However, the executor component of the swarm architecture, which currently executes container workloads, is pluggable. This means you might be able to swap it out in the future for executors that can run things like unikernel workloads. However, this is very forward thinking and probably more of a long-term vision than a short-term goal.

A quick word on the maturity of *swarm mode*

Swarm mode is based on the battle-hardened and production-tested code from the Docker Swarm project. At a high level, all of the good stuff from Docker Swarm was extracted and dumped into a re-usable toolkit called *SwarmKit*. This was then implemented natively into the Docker Engine, and *swarm mode* was born.

But the point to note is that although *swarm mode* was new in Docker 1.12, it's not like the project recklessly dropped in thousands of lines of brand new code that had

never seen the light of day. The underlying code has been around for a while and was being actively deployed in production environments.

That all said, you should still perform your normal testing before deciding to run your business critical apps on it!

Clean-up

Let's clean-up our service.

```
$ docker service rm uber-svc
uber-svc
```

Verify the uber-svc is no longer running with the `docker service ls` command.

```
$ docker service ls
ID  NAME  REPLICAS  IMAGE  COMMAND
```

Swarm mode - The commands

- `docker swarm init` is the command to create a new swarm. The node that you run the command on becomes the first manager in the new swarm and is switched to run in *swarm mode*.
- `docker swarm join-token` reveals the commands and tokens required to join workers and managers to existing swarms. To expose the command to join a new manager use the `docker swarm join-token manager` command, and to get the command to join a worker use the `docker swarm join-token worker` command.
- `docker node ls` lists all nodes in the swarm and lists which are managers and which is the leader.
- `docker service create` is the command to declaratively create a new service.
- `docker service ls` lists running services in the swarm and gives basic info on the state of the service and any tasks it's running.

- `docker service ps` gives more detailed information about individual tasks running in a service.
- `docker service inspect` gives very detailed information on a service. It accepts the `--pretty` flag to limit the information returned to the most important information.
- `docker service scale` lets you scale the number of tasks in a service up and down.
- `docker service update` lets you update many of the properties of a running service.
- `docker service rm` is the command to delete a service from the swarm. Use it with caution as it deletes all tasks in a service without asking for confirmation.

Chapter summary

In this chapter we learned about swarm mode and how to build a swarm.

We used the `docker swarm init` command to create a new swarm and make the node we ran the command on the first manager of that swarm. We then joined managers and workers. We learned that managers operate in an HA formation and the recommended number of managers is either 3 or 5.

We learned how to declare services and run them on a swarm. We saw how network ports are exposed across the entire swarm allowing us to hit any node in the swarm and reach the service endpoint - even if the node we hit wasn't running a task for the service.

We wrapped the chapter up by scaling a service up then down, and pushing an update to a live service using a rolling update.

8: What next

Hopefully you're now comfortable talking about Docker and working with it.

Taking your journey to the next step is simple in today's world. It's insanely easy to spin up infrastructure and workloads in the cloud where you can build and test Docker until you're a world authority!

You can also head over to my video training courses at Pluralsight[19]. If you're not a member of Pluralsight then become one! Yes it costs money, but its definitely a service where you get value for your money! And if you're unsure... they always have a free trail period where you can get access to my courses for free for a limited period.

I'd also recommend you hit events like Dockercon[20] and your local Docker meetups[21].

Feedback

A massive thanks for reading my book. I really hope it was useful for you!

On that point, I'd love your feedback - good and bad. If you think the book was amazing I'd love you to tell me and others! But I also want to know what you didn't like about it and how I can make the next version better!!! Please leave comments on the book's feedback pages and feel free to hit me on Twitter[22] with your thoughts!

Thanks again for reading my book and good luck driving your career forward!!

[19] http://app.pluralsight.com/author/nigel-poulton
[20] https://www.dockercon.com
[21] https://www.docker.com/community/meetup-groups
[22] https://twitter.com/nigelpoulton