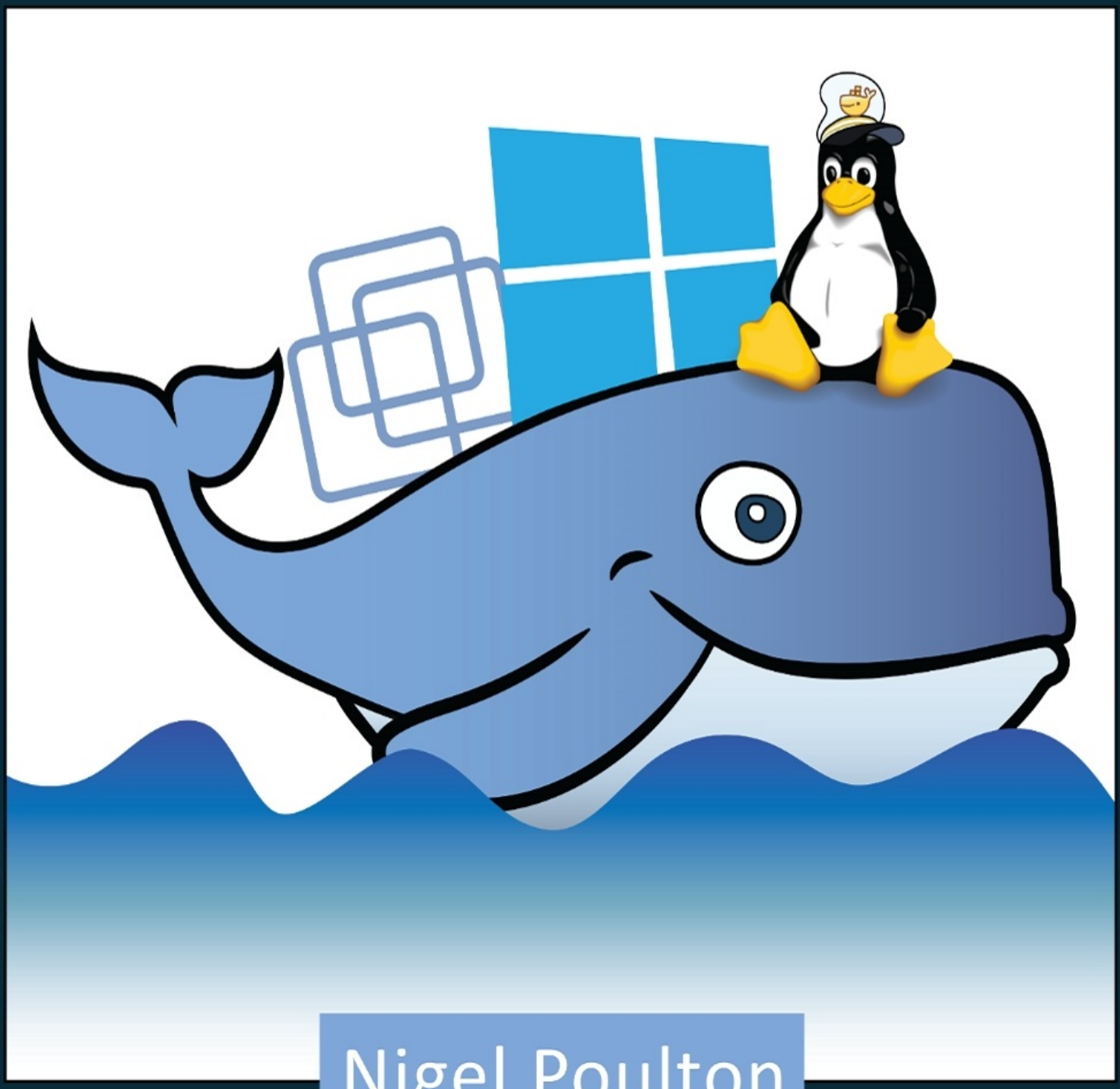
# Docker Deep Dive



Nigel Poulton

Version 2

# Docker Deep Dive

**Nigel Poulton**

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

This version was published on 2017-10-02


* * * * *

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.*

# Table of Contents

# 0: About the book

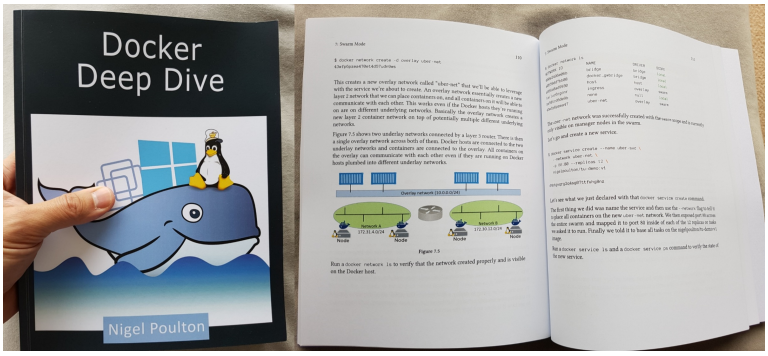
This is a book about Docker. No prior knowledge required!

If you're interested in Docker and *want to know how it works and how to do things properly* this book is dedicated to you!

If you just want to use Docker and you don't care about how it all works, this book is **not** for you. Bye.

## What about a print (paperback) version

No offense Leanpub and Amazon Kindle, but as good as modern e-books are, I'm still a fan of ink and paper! So…. this book is available as a high-quality, full-color, paperback edition via Amazon - and it's the business! None of this black-and-white nonsense.



On the topic of Amazon… I'd appreciate it if you'd give the book some stars and a quick review. Cheers!

## Why should I read this book or care about Docker?

Docker is here and there's no point hiding. Developers are all over it, and IT Ops need to be on their game! We damn well better know how to build and support production-quality *Dockerized* apps in our business-critical production environments. This book will help you.

## Isn't Docker just for developers?

If you think Docker is just for developers then prepare to have your world flipped on its head!

All the *Dockerized* apps that developers are churning out need somewhere to run and someone to manage them. If you think developers are going to do that, you're dreaming. Ops will need to build and run high performance and highly available Docker infrastructures. If you've got an Ops focus and you're not skilled-up on Docker, you're in for a world of pain. But don't stress, this book will skill you up!

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

If you like my video courses 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 big picture stuff
- The technical stuff

*The big picture stuff* covers things like - Who is Docker, Inc. What is the Docker (Moby) project. What is the OCI. Why do we even have containers… Not the coolest part of the book, but the kind of stuff that you need to know if you want a good rounded knowledge of Docker and containers. It's only a short section and you should probably 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 can use to explain the topic at the coffee machine.

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

*The Commands* lists out all the relevant commands in an easy to read list with brief reminders of what each one does.

I think you'll love that format.

# Versions of the book

Docker is developing at a warp speed! As a result, the value of a book like this is inversely proportional to how old it is! Put another way, the older this book is, the less valuable it is. So I keep it up-to-date!

If this seems strange… **welcome to the new normal!** We no-longer live in a world where a 5-year old book is valuable (believe me, it hurts for me to say that).

Don't worry though, your investment in this book is safe!

If you buy the paperback copy, you get the Kindle version for dirt-cheap! The Kindle and Leanpub versions are always kept up-to-date. That's the best we can currently do!

Below is a list of versions:

- **Version 4.** This is version 4 of the book, published on 3rd October 2017. This version added a new chapter titled "Containerizing an app". It also added content about *multi-architecture images* and *crypto ID's* to the **Images** chapter, and some additional content to **The Big Picture** chapter.
- **Version 3.** Added **The Docker Engine** chapter.
- **Version 2.** Added **Security in Docker** chapter.

# Part 1: The big picture stuff

# 1: Containers from 30,000 feet

Containers are definitely a *thing*.

In this chapter we'll get into things 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 go away. These statements get truer every day!

Most applications run on servers. And in the past, we could only run one 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 thing it could do - 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 a gift - the virtual machine (VM). And almost overnight the world changed into a much better place! We finally had a technology that would let us safely and securely run multiple business applications on a single server.

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 needs to 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 host 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 currently build on!

Despite all of this, containers remained complex and 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. made containers simple!

## Windows containers

Over the past few years, 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 10 and Windows Server 2016 platforms. 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 these *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 from the Linux platform will feel at home using Windows containers.

**This revision of the book includes Linux and Windows examples for almost every example cited.**

## Windows containers vs Linux containers

It's vital to understand that a running container uses the kernel of the host machine it is running on. This means that a container designed to run on a host with a Windows kernel will not run on a Linux host. This means that you can think of it like this at a high level - Windows containers require a

Windows Host, and Linux containers require a Linux host. However, it's not that simple…

At the time of writing this revision of the book, it is possible to run Linux containers on Windows machines. For example, *Docker for Windows* (a product offering from Docker, Inc. designed for Windows 10) can switch modes between *Windows containers* and *Linux containers*. This is an area that is developing fast and you should consult the Docker documentation for the latest.

## What about Mac containers?

There is currently no such thing as Mac containers.

However, you can run Linux containers on your Mac using the *Docker for Mac* product. This works by seamlessly running your containers inside of a lightweight Linux VM running on your Mac. It's extremely popular with developers who can easily develop and test their Linux containers on their Mac.

## Chapter Summary

We used to live in a world where every time the business wanted a new application we had to buy 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 (this is now called Moby)

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

## Docker - 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 *Moby* 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. Docker, Inc. also has offers commercial versions of Docker with support contracts etc.

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.

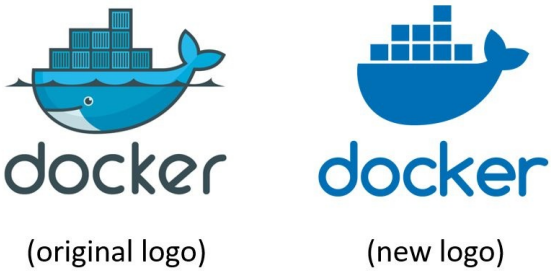



**Figure 2.1 Docker, Inc. logo.**

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 needed 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 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 7 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 its core capabilities.

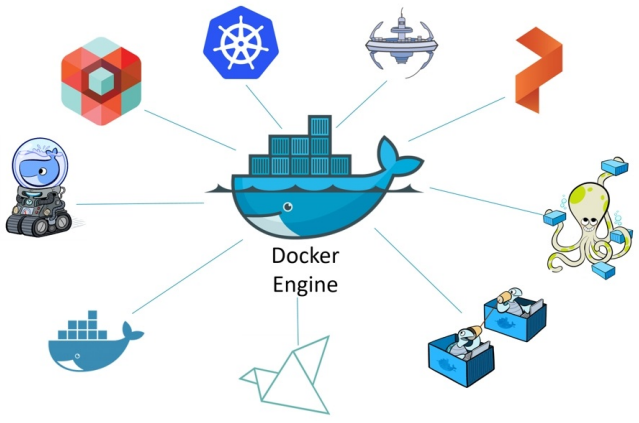



**Figure 2.2**

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 two main editions:

- Enterprise Edition (EE)
- Community Edition (CE)

The Enterprise Edition and the Community Edition both have a stable release channel with quarterly releases. Each Community Edition will be supported for 4 months and each Enterprise Edition will be supported for 12 months.

The Community Edition has an additional monthly release via an *edge* channel.

Starting from Q1 2017 Docker version numbers follow the YY.MM-xx versioning scheme similar to Ubuntu and other projects. For example, the first release of the Community Edition in June 2017 will be 17.06.0-ce.

> **Note:** Prior to Q1 2017, Docker version numbers followed the `major.minor` versioning scheme. The last version prior to the new version scheme as Docker 1.13.x.

## The Docker open-source project (Moby)

The term “Docker” is also used to refer to the open-source *Docker project*. This is the set of tools that get combined into things like the Docker daemon and client you can download and install from docker.com. However, the project was officially renamed as the *Moby* project at DockerCon 2017 in Austin, Tx. As part of this rename the GitHub repo was moved from docker/docker to moby/moby and the project got its own logo.


**Figure 2.3**

The goal of the Moby project is to break Docker down into more modular components, and to do this in the open. It’s hosted on GitHub and you can see a list of the current sub-projects and tools included in the Moby repository at https://github.com/moby. The core *Docker Engine* project is currently located at https://github.com/moby/moby but parts are being broken out and modularized all the time.

As an open-source project, the source code is publicly 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.

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

A nice side effect of Moby/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 Moby/Docker project doesn't work like that. Most things are done in the open for all to see and all to contribute to.

The Moby project, and the wider Docker movement 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 from 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 core philosophies at Docker, Inc. is often referred to as *Batteries included but removable*.

This is a way of saying you can swap out a lot of the native Docker *stuff* and replace it with *stuff* from 3rd parties. 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 and replace it with something else from a 3rd party. Plenty of people do that.

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 causing tension and raising levels competition within the ecosystem.

To cut a long story short, the native Docker batteries are still removable, there's just less and less reason to **need** 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 (OCI)

No discussion of Docker and the container ecosystem is complete without mentioning the Open Containers Initiative - OCI.


**Figure 2.4**

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

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 didn't like the way Docker did certain things. So they did something about it! They created a new open standard called **appc** 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.

Getting back to the story though, this threatened to fracture the ecosystem and present users and customers with a dilemma. While competition is usually a good thing, *competing standards* is usually not. They cause confusion and slowdown user 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) -

- The image-spec
- The runtime-spec

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 signalling 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 product. As of Docker 1.11, the Docker Engine architecture conforms to the OCI 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.

## Chapter summary

In this chapter we learned a bit about Docker, Inc. They're a startup tech company out of San Francisco with an ambition to change the way we do software. They were arguably the *first-movers* and instigators of the container revolution. But a huge ecosystem of partners and competitors now exists.

The *Docker project* is open-source and lives in the `moby/moby` repo on GitHub.

The Open Container Initiative (OCI) has been instrumental in standardizing the container runtime format and container image format.

# 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 easy.

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

- Desktop installs
  - Docker for Windows
  - Docker for Mac
- Server installs
  - Linux
  - Windows Server 2016

## Docker for Windows (DfW)

The first thing to note is that *Docker for Windows* is a packaged product from Docker, Inc. This means it's got a slick installer that spins up a single-engine Docker environment on a 64-bit Windows 10 desktop or laptop.

The second thing to note is that it is a Community Edition (CE) app. This means it is not intended for production workloads.

The third thing of note is that it might suffer some feature-lag. This is because Docker, Inc. are taking a *stability first, features second* approach with the product.

All three points add up to a quick and easy installation, but one that is **not** intended 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 (1511 November update, Build 10586 or later)
- 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

The following will assume that hardware virtualization support is already enabled in your system's BIOS. If it is not, you should carefully follow the procedure for your particular machine.

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 `Apps and Features`.
2. Click the `Programs and Features` link.
3. Click `Turn Windows features on or off`.
4. 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.

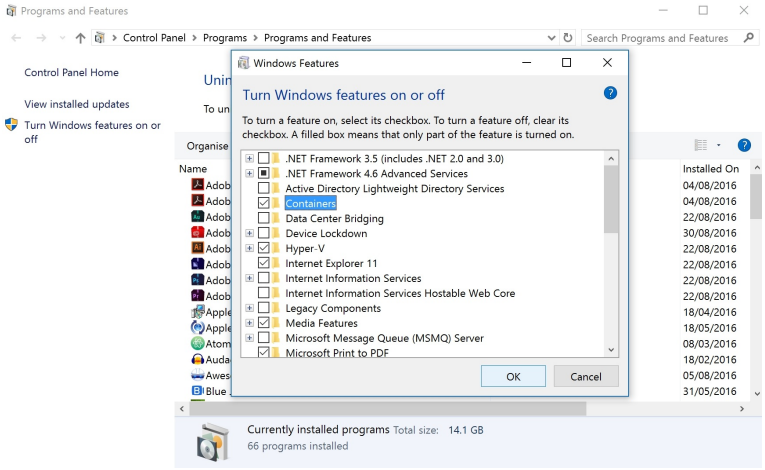



**Figure 3.1**

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

Once you've installed the Hyper-V and Containers features and restarted your 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 page. This will open a dropdown menu.
2. Choose `Windows` from under the `Desktop` section.
3. Click `Download from Docker Store`.
4. Click one of the `Get Docker` download links.

There are various stable and edge versions available. The edge version contains newer features but may not be as stable.

An installer package called `InstallDocker.msi` will be downloaded to your default downloads directory.

1. 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:\> docker version
Client:
  Version:      17.05.0-ce
  API version:  1.29
  Go version:   go1.7.5
  Git commit:   89658be
  Built:        Thu May  4 21:43:09 2017
  OS/Arch:      windows/amd64

Server:
  Version:      17.05.0-ce
  API version:  1.29 (minimum version 1.12)
  Go version:   go1.7.5
  Git commit:   89658be
  Built:        Thu May  4 21:43:09 2017
  OS/Arch:      linux/amd64
  Experimental: false
```

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 can achieve the same thing form the command line with the following command (located in the `\Program Files\Docker\Docker` directory):

```
C:\Program Files\Docker\Docker>dockercli -SwitchDaemon
```

You will get the following alert if you have not enabled the `Windows Containers` feature.

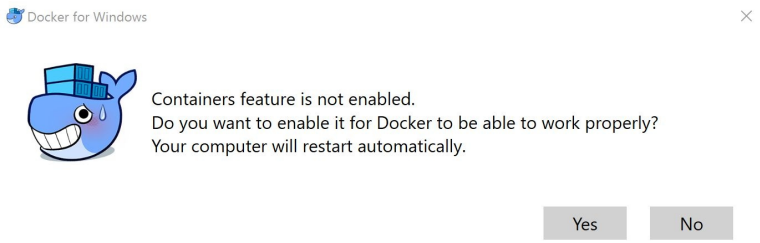



**Figure 3.2**

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:\> docker version
Client:
  Version:      17.05.0-ce
  API version:  1.29
  Go version:   go1.7.5
  Git commit:   89658be
```

```
  Built:        Thu May  4 21:43:09 2017
  OS/Arch:      windows/amd64

Server:
  Version:      17.05.0-ce
  API version:  1.29 (minimum version 1.12)
  Go version:   go1.7.5
  Git commit:   89658be
  Built:        Thu May  4 21:43:09 2017
  OS/Arch:      windows/amd64
  Experimental: true
```

Notice that the Server version is now 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 now running the *experimental* version of Docker (`Experimental: true`). As previously mentioned, *Docker for Windows* has a stable and an edge channel. The example above is from a Windows 10 machine running the *edge* channel. The Windows containers feature of the edge channel is currently an experimental feature.

You can check which channel you are running with the `dockercli -Version` command. The `dockercli` command is located in `C:\Program Files\Docker\Docker`.

```
C:\> dockercli -Version
Docker for Windows
Version: 17.05.0-ce-win11 (12053)
Channel: edge
Sha1: ffbc5f5871f44611dfb2bbf49e8312332531c112
OS Name: Windows 10 Pro
Windows Edition: Professional
Windows Build Number: 15063
```

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

```
   C:\>docker info
   Containers: 1
     Running: 0
     Paused: 0
     Stopped: 1
   Images: 1
   Server Version: 17.05.0-ce
   Storage Driver: windowsfilter
     Windows:
   Logging Driver: json-file
   Plugins:
     Volume: local
     Network: l2bridge l2tunnel nat null overlay transparent
   <SNIP>
   Experimental: true
   Insecure Registries:
    127.0.0.0/8
   Live Restore Enabled: false
```

Docker for Windows includes the Docker Engine (client and daemon), Docker Compose, Docker Machine, and the Docker Notary command line.

Use the following commands to verify that each was successfully installed and which versions of each you have:

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


C:\> docker-compose --version
docker-compose version 1.13.0, build 1719ceb8


C:\> docker-machine --version
docker-machine version 0.11.0, build 5b27455


C:\> notary version
notary
 Version:    0.4.3
 Git commit: 9211198
```

## Docker for Mac (DfM)

*Docker for Mac* is also 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?*

First up, *Docker for Mac* is a packaged product from Docker, Inc. that is based on the Community Edition of Docker. This means it's an easy way to install a single-engine version of Docker on you Mac. It also means that it's not intended for production use. If you've heard of **boot2docker** then *Docker for Mac* is what you always wished *boot2docker* was - it's smooth, simple and stable.

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 daemon inside of a lightweight Linux VM. It then seamlessly exposes the daemon 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, it's 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.

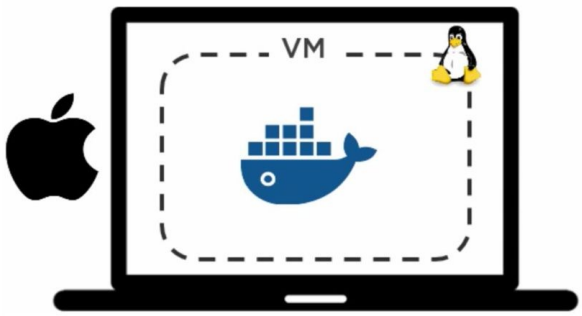



**Figure 3.3**

> **Note:** For the curious reader, *Docker for Mac* leverages HyperKit to implement an extremely lightweight hypervisor. HyperKit is based on the xhive hypervisor. *Docker for Mac* also leverages features from DataKit and runs a highly tuned Linux distro called *Moby* that is based off of Alpine Linux.

Let's get *Docker for Mac* installed.

1. Point your browser to www.docker.com and click `Get Docker` from the top of the page. This will open a dropdown menu.
2. Choose `Mac` from under the `Desktop` section.
3. Click `Download from Docker Store`.
4. Click one of the `Get Docker` download links.

There are various stable and edge versions available. The edge version contains newer features but may not be stable.

A **Docker.dmg** installation package will be downloaded.

1. 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.
2. 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 from the internet.
3. Enter your password so that the installer can create components, such as networking, that require elevated privileges.
4. The Docker daemon will now start.

   An animated whale icon will appear in the status bar at the top of your screen while the daemon starts. Once Docker has successfully started the whale will stop being animated. 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:      17.05.0-ce
 API version:  1.29
 Go version:   go1.7.5
 Git commit:   89658be
 Built:        Thu May 4 21:43:09 2017
 OS/Arch:      darwin/amd64

Server:
 Version:      17.05.0-ce
 API version:  1.29 (minimum version 1.12)
 Go version:   go1.7.5
 Git commit:   89658be
 Built:        Thu May 4 21:43:09 2017
 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. This is because the system is running the *edge* version which comes with experimental features turned on.

Run some more Docker commands.

```
$ docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 17.05.0-ce
<Snip>
Registry: https://index.docker.io/v1/
Experimental: true
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
```

Docker for Mac installs the Docker Engine (client and daemon), Docker Compose, and Docker machine and the Notary command line. 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 17.05.0-ce, build 89658be
```

```
$  docker-compose --version
docker-compose version 1.13.0, build 1719ceb


$  docker-machine --version
docker-machine version 0.11.0, build 5b27455


$ notary version
notary
  Version:    0.4.3
  Git commit: 9211198
```

## Installing Docker on Linux

Installing Docker on Linux is the most common installation type and it's 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 its 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 edition to install. There are currently two editions:

- Community Edition (CE)
- Enterprise Edition (EE)

Docker CE is free and is the version we will be demonstrating here. Docker EE is the same as CE but comes with commercial support and access to other Docker products such as Docker Trusted Registry and Universal Control Plane.

In the examples below we'll use the `wget` command to call a shell script that installs Docker CE. For information on other ways to install Docker on Linux go to https://www.docker.com and click on `Get Docker`.

> **Note:** You should ensure that your system is up-to-date with the latest packages and security patches before continuing.

1. Open a new shell on your Linux machine.
2. Use `wget` to retrieve and run 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 /lib/modules/4.4.0-36-generic
+ 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...
```

1. It is best practice to only use non-root users when working with Docker. 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. You will need to use a valid user account on your own system.

   ```
   $ 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 17.05.0-ce, build 89658be
$
$ docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
<Snip>
Kernel Version: 4.4.0-1013-aws
Operating System: Ubuntu 16.04.2 LTS
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 990.7MiB
Name: ip-172-31-45-57
ID: L3GX:LLFI:YABL:WVUS:DHLL:2ZQU:44E3:V6BB:LWUY:WIGX:Z6RJ:JBVL
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Experimental: false
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
```

If the process described above doesn't work for your Linux distro, you can go to the Docker Docs 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. 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 if 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 its authorized service partners.

## Installing Docker on Windows Server 2016

In this section we'll look at one of the ways to install Docker on Windows Server 2016. We'll complete the following high-level steps:

1. Install the Windows Containers feature
2. Install Docker
3. Verify the installation

Before proceeding you should ensure that your system is up-to-date with the latest package versions and security updates. You can do this quickly with the `sconfig` command and choosing option 6 to install updates. This may require a system restart.

We'll be demonstrating an installation on a version of Windows Server 2016 that does not already have the Containers feature or an older version of Docker already installed.

Ensure that the Containers feature is installed and enabled.

1. Right-click the Windows Start button and select `Programs and Features`. This will open the `Server Manager` app.
2. Select the `Dashboard` and click `Add Roles and Features`.
3. Click through the wizard until you get to the `Features` page.
4. Make sure that the `Containers` feature is checked and complete the wizard. This may require a system restart.

Now that the Windows Containers feature is installed you can install Docker. We'll use PowerShell to do this.

1. Open a new PowerShell Administrator terminal.
2. Use the following command to install the Docker-Microsoft package management provider.

   ```
   > Install-Module -Name DockerMsftProvider -Repository PSGallery -Force
   ```

   Accept the NuGet provider install if prompted.

3. Install Docker.

   ```
    > Install-Package -Name docker -ProviderName DockerMsftProvider
   ```

   Select `A` to confirm package installation and suppress any further prompts.

   Once the installation is complete you will get a summary as shown below.

   ```
   Name      Version       Source           Summary
   ----      -------       ------           -------
   Docker    17.03.1-ee    DockerDefault    Contains Docker EE...
   ```

4. Restart your system

Docker is now installed and you can start deploying containers. The following two commands are good ways to verify that the installation succeeded.

```
> docker version
Client:
 Version:      17.03.1-ee-3
 API version:  1.27
 Go version:   go1.7.5
 Git commit:   3fcee33
 Built:        Thu Mar 30 19:31:22 2017
 OS/Arch:      windows/amd64

Server:
 Version:      17.03.1-ee-3
 API version:  1.27 (minimum version 1.24)
 Go version:   go1.7.5
 Git commit:   3fcee33
 Built:        Thu Mar 30 19:31:22 2017
```

```
 OS/Arch:      windows/amd64
 Experimental: false


> docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 17.03.1-ee-3
Storage Driver: windowsfilter
 Windows:
Logging Driver: json-file
Plugins:
 Volume: local
 Network: l2bridge l2tunnel nat null overlay transparent
<SNIP>
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
```

Docker is now installed and you are ready to start using Windows containers.

## Chapter Summary

In this chapter you saw how to install Docker on Windows 10, Mac OS X, Linux, and Windows Server 2016. Now that you know how to install Docker you are ready to start working with images and containers.

# 4: The big picture

The idea of this chapter is to give you a quick big picture of what Docker is all about before we dive in deeper in later chapters.

We'll break this chapter into two:

- The Ops perspective
- The Dev perspective

The Ops Perspective section will download an image, start a new container, log in to the new container, run a command inside of it, and then destroy it.

The Dev Perspective section will pull some app-code from GitHub, inspect a Dockerfile, containerize the app, run it as a container.

These two sections will give you a good idea of what Docker is all about and how some of the major components fit together. **It is recommended that you read both sections to get the *dev* and the *ops* perspectives!**

Don't worry if some of the stuff we do here is totally new to you. We're not trying to make you an expert by the end of this chapter. This is all about 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 with an internet connection. Your Docker Host can be Linux or Windows, and it doesn't matter if it's a VM on your laptop, an instance in the public cloud, or a bare metal server in your data center. All it needs, is to be running Docker with a connection to the internet. We'll be showing examples using Linux and Windows!

## The Ops Perspective

When you install Docker, you get two major components:

- the Docker client
- the Docker daemon (sometimes called “server” or “engine”)

The daemon implements the Docker Remote API.

In a default Linux installation, the client talks to the daemon via a local IPC/Unix socket at `/var/run/docker.sock`. On Windows this happens via a named pipe at `npipe:////./pipe/docker_engine`. You can test that the client and daemon are running and can talk to each other with the `docker version` command.

```
$ docker version
Client:
 Version:      17.05.0-ce
 API version:  1.29
 Go version:   go1.7.5
 Git commit:   89658be
 Built:        Thu May  4 22:10:54 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.05.0-ce
 API version:  1.29 (minimum version 1.12)
 Go version:   go1.7.5
 Git commit:   89658be
 Built:        Thu May  4 22:10:54 2017
 OS/Arch:      linux/amd64
 Experimental: false
```

If you get a response back from the `Client` **and** `Server` components you should be good to go. If you are using Linux and get an error response from 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 add your user account to the local `docker` group, or prefix the remainder of the commands in this chapter with `sudo`.

## Images

A good way to think of a Docker image is as an object that contains an OS filesystem and an application. If you work in operations, it’s like 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. If you’re a developer, you can think of an image as a *class*.

Run the `docker image ls` command on your Docker host.

```
$ docker image ls
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”. If you are following along with Linux, pull the `ubuntu:latest`. If you are following along on Windows, pull the `microsoft/powershell:nanoserver` image.

```
$ docker image pull ubuntu:latest
latest: Pulling from library/ubuntu
b6f892c0043b: Pull complete
55010f332b04: Pull complete
2955fb827c94: Pull complete
3deef3fcbd30: Pull complete
cf9722e506aa: Pull complete
Digest: sha256:382452f82a8b....463c62a9848133ecb1aa8
Status: Downloaded newer image for ubuntu:latest
```

Run the `docker image ls` command again to see the 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 later chapters. For now, it’s enough to understand that an image contains enough of an operating system (OS), as well as all the code and dependencies to run whatever application it’s designed for. The `ubuntu` image that we’ve pulled has a stripped-down version of the Ubuntu Linux filesystem including a few of the common Ubuntu utilities. The `microsoft/powershell` image pulled in the Windows example contains a Windows Nano Server OS with PowerShell.

If you pull an application container such as `nginx` or `microsoft/iis`, you will get an image that contains some OS as well as the code to run either `nginx` or `IIS`.

It’s also worth noting that each image gets its own unique ID. When working with the images you can refer to them using either `IDs` or names.

## Containers

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

For Linux:

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

For Windows:

```
> docker container run -it microsoft/powershell:nanoserver PowerShell.exe
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\>
```

Look closely at the output from the commands above. You should notice that the shell prompt has changed in each instance. 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 container run` command. `docker container 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 terminal 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 (or the microsoft/powershell:nanoserver image if you're following along with Windows). Finally, we tell Docker which process we want to run inside of the container. For the Linux example we're running a Bash shell, for the Windows container were running PowerShell.

Run a `ps` command from inside of the container to list all running processes.

**Linux example:**

```
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
```

**Windows example:**

```
PS C:\> ps

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
      0       5      964       1292       0.00   4716   4 CExecSvc
      0       5      592        956       0.00   4524   4 csrss
      0       0        0          4                 0   0 Idle
      0      18     3984       8624       0.13    700   4 lsass
      0      52    26624      19400       1.64   2100   4 powershell
      0      38    28324      49616       1.69   4464   4 powershell
      0       8     1488       3032       0.06   2488   4 services
      0       2      288        504       0.00   4508   0 smss
      0       8     1600       3004       0.03    908   4 svchost
      0      12     1492       3504       0.06   4572   4 svchost
      0      15    20284      23428       5.64   4628   4 svchost
      0      15     3704       7536       0.09   4688   4 svchost
      0      28     5708       6588       0.45   4712   4 svchost
      0      10     2028       4736       0.03   4840   4 svchost
      0      11     5364       4824       0.08   4928   4 svchost
      0       0      128        136      37.02      4   0 System
      0       7      920       1832       0.02   3752   4 wininit
      0       8     5472      11124       0.77   5568   4 WmiPrvSE
```

Inside the Linux container there are only two processes running:

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

The presence of the `ps -elf` process in the Linux 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.

The Windows container has a lot more internal processes running. This is an artefact of the way the Windows Operating System works. Although the Windows container has a lot more processes than the Linux container, it is still a lot less than a regular Windows Server.

Press `Ctrl-PQ` to exit the container without terminating it. 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 your Docker host, run the `ps` command again.

**Linux example:**

```
$ 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
```

**Windows example:**

```
> ps
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    220      11     7396       7872       0.33   1732   0 amazon-ssm-agen
     84       5      908       2096       0.00   2428   3 CExecSvc
     87       5      936       1336       0.00   4716   4 CExecSvc
    203      13     3600      13132       2.53   3192   2 conhost
    210      13     3768      22948       0.08   5260   2 conhost
    257      11     1808        992       0.64    524   0 csrss
    116       8     1348        580       0.08    592   1 csrss
     85       5      532       1136       0.23   2440   3 csrss
    242      11     1848        952       0.42   2708   2 csrss
     95       5      592        980       0.00   4524   4 csrss
    137       9     7784       6776       0.05   5080   2 docker
    401      17    22744      14016      28.59   1748   0 dockerd
    307      18    13344       1628       0.17    936   1 dwm
    <SNIP>
   1888       0      128        136      37.17      4   0 System
```

```
    272      15     3372       2452       0.23   3340   2 TabTip
     72       7     1184          8       0.00   3400   2 TabTip32
    244      16     2676       3148       0.06   1880   2 taskhostw
    142       7     6172       6680       0.78   4952   3 WmiPrvSE
    148       8     5620      11028       0.77   5568   4 WmiPrvSE
```

Notice how many more processes are running on your Docker host compared to the containers we ran.

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

```
$ docker container ls
CONTAINER ID   IMAGE          COMMAND       CREATED  STATUS    NAMES
e2b69eeb55cb   ubuntu:latest  "/bin/bash"   7 mins   Up 7 min  vigilant_borg
```

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 container exec` command. As the container from the previous steps is still running, let's connect back to it.

**Linux example:**

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

```
$ docker container exec -it vigilant_borg bash
root@e2b69eeb55cb:/#
```

**Windows example:**

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

```
> docker container exec -it pensive_hamilton PowerShell.exe
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\>
```

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

The format of the `docker container exec` command is: `docker container 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 (PowerShell in the Windows example). We could easily have referenced the container by its ID.

Exit the container again by pressing `Ctrl-PQ`.

Your shell prompt should be back to your Docker host.

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

```
$ docker container ls
CONTAINER ID   IMAGE          COMMAND       CREATED  STATUS    NAMES
e2b69eeb55cb   ubuntu:latest  "/bin/bash"   9 mins   Up 9 min  vigilant_borg
```

Stop the container and kill it using the `docker container stop` and `docker container rm` commands. Remember to substitute the names/IDs of your own containers.

```
$ docker container stop vigilant_borg
vigilant_borg
$
$ docker container rm vigilant_borg
vigilant_borg
```

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

```
$ docker container ls
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

## The Dev Perspective

Containers are all about the apps!

In this section we'll clone an app from a Git repo, inspect its Dockerfile, containerize it, and run it as a container.

The Linux app can be located from: https://github.com/nigelpoulton/psweb.git

The Windows app can be located from: https://github.com/nigelpoulton/dotnet-docker-samples.git

The rest of this section will walk you through the Linux example. However, both examples are containerizing simple web apps so the process is the same. Where there are differences in the Windows example we will highlight them to help you follow along.

Run all of the following commands from a terminal on your Docker host.

Clone the repo locally. This will pull the application code to your local Docker host ready for you to containerize it.

Be sure to substitute the repo below with the Windows repo if you are following along with the Windows example.

```
$ git clone https://github.com/nigelpoulton/psweb.git
Cloning into 'psweb'...
remote: Counting objects: 15, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 15 (delta 2), reused 15 (delta 2), pack-reused 0
Unpacking objects: 100% (15/15), done.
Checking connectivity... done.
```

Change directory into the cloned repo's directory and list the contents of the directory.

```
$ cd psweb
$ ls -l
total 28
-rw-rw-r-- 1 ubuntu ubuntu  341 Sep 29 12:15 app.js
-rw-rw-r-- 1 ubuntu ubuntu  216 Sep 29 12:15 circle.yml
-rw-rw-r-- 1 ubuntu ubuntu  338 Sep 29 12:15 Dockerfile
-rw-rw-r-- 1 ubuntu ubuntu  421 Sep 29 12:15 package.json
-rw-rw-r-- 1 ubuntu ubuntu  370 Sep 29 12:15 README.md
drwxrwxr-x 2 ubuntu ubuntu 4096 Sep 29 12:15 test
drwxrwxr-x 2 ubuntu ubuntu 4096 Sep 29 12:15 views
```

For the Windows example you should `cd` into the `dotnet-docker-samples\aspnetapp` directory.

The Linux example is a simple nodejs web app. The Windows example is a simple ASP.NET Core web app.

Both Git repos contain a file called `Dockerfile`. A Dockerfile is a plain-text document describing how to build a Docker image.

List the contents of the Dockerfile.

```
$ cat Dockerfile

FROM alpine
LABEL maintainer="nigelpoulton@hotmail.com"
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN  npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

The contents of the Dockerfile in the Windows example are different. However, this is not important at this stage. We'll cover Dockerfiles in more detail later in the book. For now, it's enough to understand that each line represents an instruction that is used to build an image.

At this point we have pulled some application code from a remote Git repo. We also have a Dockerfile containing instructions that describe how to create a new Docker image with the application inside.

Use the `docker image build` command to create a new image using the instructions contained in the Dockerfile. This example creates a new Docker image called `test:latest`.

Be sure to perform this command from within the directory containing the app code and Dockerfile.

```
$ docker image build -t test:latest .

Sending build context to Docker daemon  74.75kB
Step 1/8 : FROM alpine
latest: Pulling from library/alpine
88286f41530e: Pull complete
Digest: sha256:f006ecbb824...0c103f4820a417d
Status: Downloaded newer image for alpine:latest
 ---> 76da55c8019d
<Snip>
Successfully built f154cb3ddbd4
Successfully tagged test:latest
```

> **Note:** It may take a long time for the build to finish in the Windows example. This is because of the size and complexity of the layers being pulled.

Check to make sure that the new `test:latest` image exists on your host.

```
$ docker image ls
REPO     TAG      IMAGE ID        CREATED         SIZE
test     latest   f154cb3ddbd4    1 minute ago    55.6MB
...
```

You now have a newly built image with the app inside.

Run a container from the image and test the app.

**Linux example:**

```
$ docker container run -d \
  --name web1 \
  -p 8080:8080 \
  test:latest
```

Open a web browser and navigate to the DNS name or IP address of the host that you are running the container from and point it to port 8080. You will see the following web page.

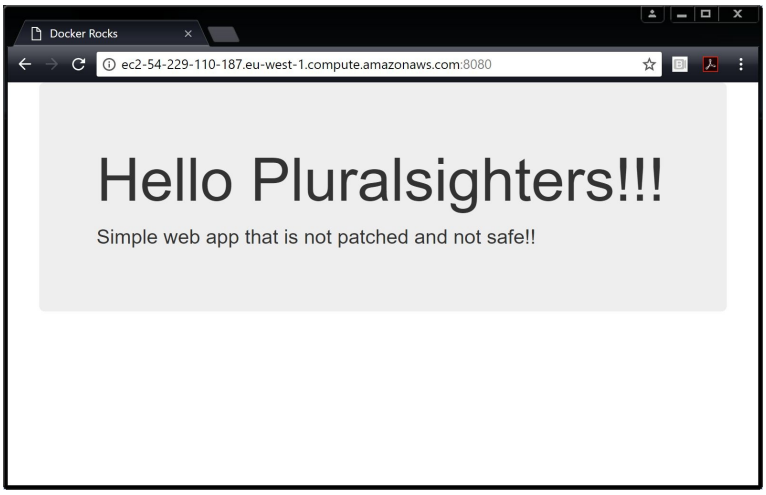



**Figure 4.1**

**Windows example:**

```
> docker container run -d \
  --name web1 \
  -p 8000:80 \
  test:latest
```

Open a web browser and navigate to the DNS name or IP address of the host that you are running the container from and point it to port 8080. You will see the following web page.

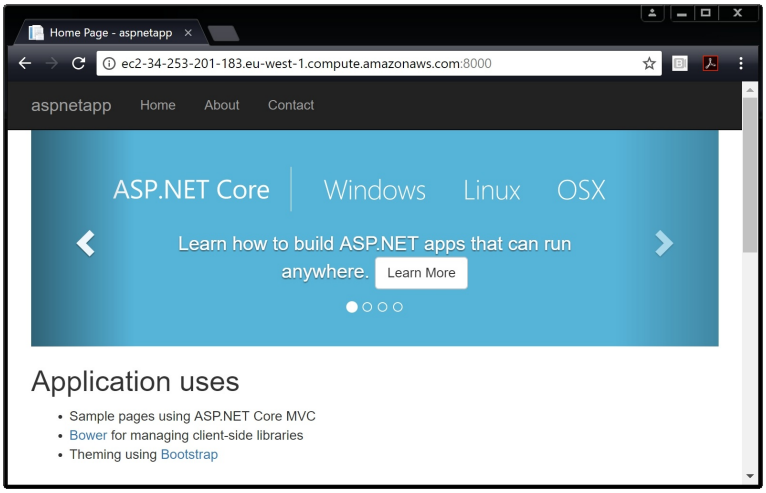



**Figure 4.2**

Well done. You've taken an application and containerized it (built a Docker image from it).

## Chapter Summary

In this chapter you performed the following operations-related tasks; downloaded a Docker image, launched a container from the image, executed a command inside of the container (`ps`), and then stopped and deleted the container. You also containerized a simple application by pulling some source code from GitHub and building it into an image using instructions in a Dockerfile.

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: The Docker Engine

In this chapter, we'll take a quick look under the hood of the Docker Engine.

You can use Docker without understanding any of the things we'll cover in this chapter. So, feel free to skip it. However, to be a real master of anything, you need to understand what's going on under the hood.

This will be a theory-based chapter with no hands-on exercises.

As this chapter is part of 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 recap of the commands we learned

Let's go learn about the Docker Engine!

## Docker Engine - The TLDR

The *Docker engine* is the core software that runs and manages containers. We often refer to it simply as *Docker*, or *the Docker platform*. If you know a thing or two about VMware, it might be useful to think of it as being like ESXi in the VMware world.

The Docker engine is modular in design with many swappable components. Where possible, these are based on open-standards outlined by the Open Container Initiative (OCI).

In many ways, the Docker Engine is like a car engine - both are modular and created by connecting many small specialized parts: - A car engine is made from many specialized parts that work together to make a car drive - intake manifolds, throttle body, cylinders, spark plugs, exhaust manifolds etc. - The Docker Engine is made from many specialized tools that work together to create and run containers - images, APIs, execution driver, runtime, shims etc.

At the time of writing, the major components that make up the Docker engine are: the *Docker client*, the *Docker daemon*, *containerd*, and *runc*. Together, these create and run containers.

Figure 5.1 shows a high-level view.

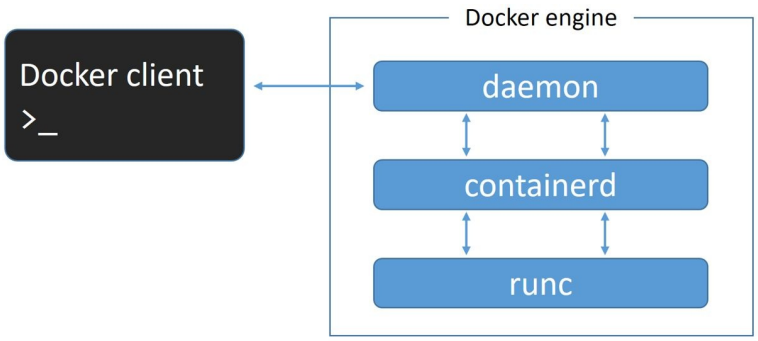



**Figure 5.1**

## Docker Engine - The Deep Dive

When Docker was first released, the Docker engine had two major components:

- The Docker daemon (hereafter referred to as just “the daemon”)
- LXC

The Docker daemon was a monolithic binary. It contains all of the code for the Docker client, the Docker API, the container runtime, image builds, and much more.

The LXC component provided the daemon with access to the fundamental building-blocks of containers such as *kernel namespaces* and *control groups (cgroups)*.

The interaction between the daemon, LXC and the OS is shown in Figure 5.2.

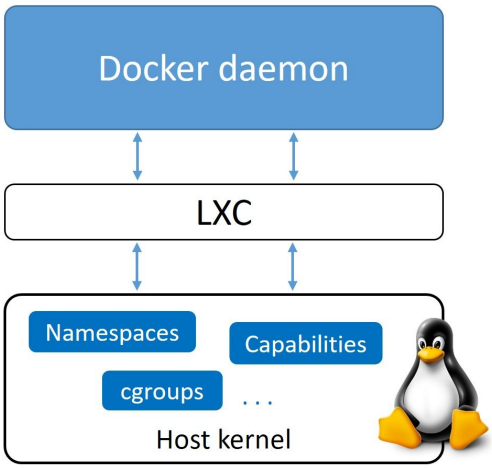



**Figure 5.2**

## Getting rid of LXC

The reliance on LXC was an issue from the start.

First up, LXC is Linux-specific. This was a problem for a project that had aspirations of being multi-platform.

Second up, being reliant on an external tool for something so core to the project was a huge risk that could hinder development.

As a result, Docker. Inc. developed their own tool called *libcontainer* as a replacement for LXC. The goal of *libcontainer* was to be a platform-agnostic tool that provided Docker with access to the fundamental container building-blocks that exist inside the OS.

Libcontainer replaced LXC as the default *execution driver* in Docker 0.9.

## Getting rid of the monolithic Docker daemon

Over time, the monolithic nature of the Docker daemon became more and more problematic:

1. It's hard to innovate on.
2. It got slower.
3. It wasn't what the ecosystem (or Docker, Inc.) wanted.

Docker, Inc. was aware of these challenges, and began a huge effort to break apart the monolithic daemon and modularize it. The aim of this work is to break out as much of the functionality as possible from the daemon, and re-implement it in smaller specialized tools. These specialized tools can be swapped out, as well as easily used by third parties to build other tools. This plan follows the tried-and-tested Unix philosophy of building small specialized tools that can be pieced together into larger tools.

This work of breaking apart and re-factoring the Docker engine is an ongoing process. However, it has already seen **all of the *container execution* and container *runtime* code entirely removed from the daemon and refactored into small, specialized tools**.

Figure 5.3 shows a high-level view of the Docker engine architecture with brief descriptions.

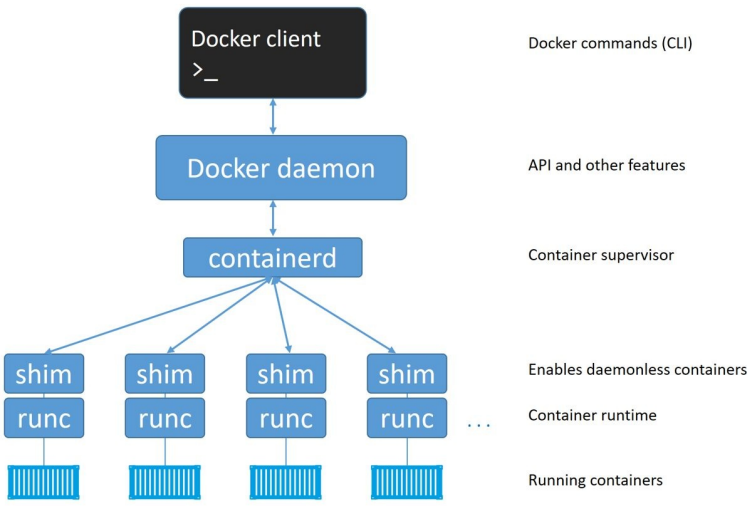



**Figure 5.3**

## The influence of the Open Container Initiative (OCI)

While Docker, Inc. was breaking the daemon apart and refactoring code, the OCI was in the process of defining two container-related standards:

1. Image spec
2. Container runtime spec

Both specifications were released as version 1.0 in July 2017.

Docker, Inc. was heavily involved in creating these specifications and contributed a lot of code to them.

As of Docker 1.11 (early 2016), the Docker engine implements the OCI specifications as closely as possible. For example, the Docker daemon no longer contains any container runtime code - all container runtime code is implemented in a separate OCI-compliant layer. By default, Docker uses a tool called *runc* for this. runc is the *reference implementation* of the OCI container-runtime-spec, and a goal of the runc project keep runc in lockstep with the OCI spec.

As well as this, the *containerd* component of the Docker Engine makes sure Docker images are presented to *runc* as valid OCI bundles.

> **Note:** The Docker engine implemented portions of the OCI specs before the specs were officially released as version 1.0.

## runc

As previously mentioned, *runc* is the reference implementation of the OCI container-runtime-spec. Docker, Inc. was heavily involved in defining the spec and developing runc.

runc is small. It's effectively a lightweight CLI that wraps around libcontainer. It has a single purpose in life - to create containers. And it's damn good at it. And fast!

We often refer to runc as a container *runtime*.

You can see runc release information at:

- https://github.com/opencontainers/runc/releases

## containerd

In order to use runc, the Docker engine needed something to act as a bridge between the daemon and runc. This is where containerd comes into the picture.

containerd implements the execution logic that was pulled out of the Docker daemon. This logic was obviously refactored and tuned when it was re-

written as containerd.

It's helpful to think of containerd as a container *supervisor* - the component that is responsible for container lifecycle operations such as; starting and stopping containers, pausing and un-pausing them, and destroying them.

Like runc, containerd is small, lightweight, and designed for a single task in life - containerd is only interested container lifecycle operations.

containerd was developed by Docker, Inc. and donated to the Cloud Native Computing Foundation (CNCF).

You can see containerd release information at:

- https://github.com/containerd/containerd/releases

## Starting a new container (example)

Now that we have a view of the big picture, and some of the history, let's walk through the process of creating a new container.

The most common way of starting containers is using the Docker CLI. The following `docker container run` command will start a simple new container based on the `alpine:latest` image.

```
$ docker container run --name ctr1 -it alpine:latest sh
```

When you type commands like this into the Docker CLI, the Docker client converts them into the appropriate API payload and POSTs them to the correct API endpoint.

The API is implemented in the daemon. It is the same rich, versioned, REST API that has become a hallmark of Docker and is accepted in the industry as the de facto container API.

Once the daemon receives the command to create a new container, it makes a call to containerd. Remember that the daemon no-longer contains any code to create containers!

The daemon communicates with containerd via a CRUD-style API over gRPC.

Despite its name, *containerd* cannot actually create containers. It uses *runc* to do that. It converts the required Docker image into an OCI bundle and tells runc to use this to create a new container.

runc interfaces with the OS kernel to pull together all of the constructs necessary to create a container (in Linux these include namespaces and

cgroups). The container process is started as a child-process of runc, and as soon as it is started runc will exit.

Voila! The container is now started.

Figure 5.4 summarises the process

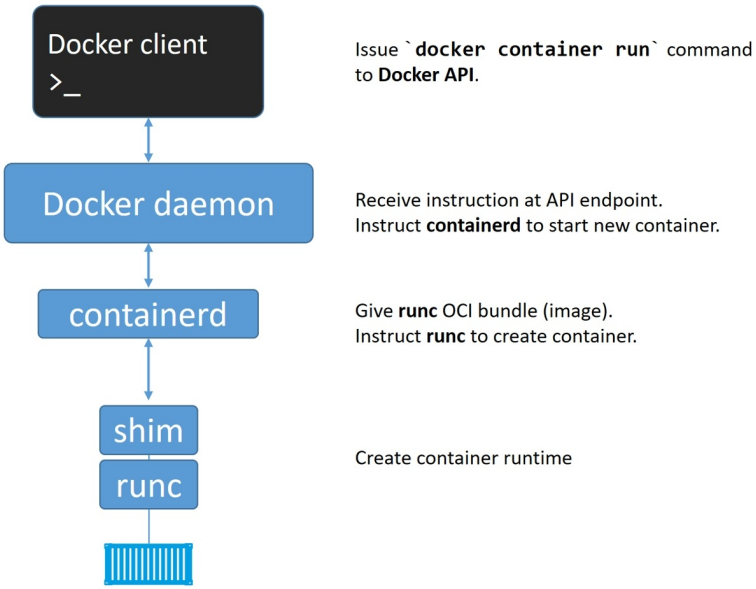



**Figure 5.4**

## One huge benefit of this model

Having all of the logic and code to start and manage containers removed from the daemon means that the entire container runtime is decoupled from the Docker daemon. We sometimes call this “daemonless containers”, and it makes it possible to perform maintenance and upgrades on the Docker daemon without impacting running containers!

In the old model, where all of container runtime logic was implemented in the daemon, starting and stopping the daemon would kill all running containers on the host. This was a huge problem in production environments - especially when you consider how frequently new versions of Docker are released! Every daemon upgrade would kill all containers on that host - not good!

Fortunately, this is no longer a problem.

## What’s this shim all about?

Some of the diagrams in the chapter have shown a shim component.

The shim is integral to the implementation of daemonless containers (the thing we just mentioned about decoupling running containers from the daemon for things like upgrading the daemon without killing containers).

We mentioned earlier that *containerd* uses runc to create new containers. In fact, it forks a new instance of runc for every container it creates. However, once each container is created, its parent runc process exits. This means we can run hundreds of containers without having to run hundreds of runc instances.

Once a container's parent runc process exits, the associated containerd-shim process becomes the container's parent process. Some of the responsibilities the shim performs as a container's parent include:

- Keeping any STDIN and STDOUT streams open so that when the daemon is restarted, the container doesn't terminate due to pipes being closed etc.
- Reports the container's exit status back to the daemon.

## How it's implemented on Linux

On a Linux system, the components we've discussed are implemented as separate binaries as follows: - `dockerd` (the Docker daemon) - `docker-containerd` (containerd) - `docker-containerd-shim` (shim) - `docker-runc` (runc)

You can see all of these on a Linux system by running a `ps` command on the Docker host. Obviously, some of them will only be present when the system has running containers.

## So what's the point of the daemon

With all of the execution and runtime code stripped out of the daemon you might be asking the question: "what is left in the daemon?".

Obviously, the answer to this question will change over time as more and more functionality is stripped out and modularized. However, at the time of writing, some of the major functionality that still exists in the daemon includes; image management, image builds, the REST API, authentication, security, core networking, and orchestration.

## Chapter summary

The Docker engine is modular in design and based heavily on open-standards from the OCI.

The *Docker daemon* implements the Docker API which is currently a rich, versioned, HTTP API that has developed alongside the rest of the Docker project. This Docker API is accepted as the industry-standard container API.

Container execution is handled by *containerd*. containerd was written by Docker, Inc. and contributed to the CNCF. You can think of it as a container supervisor that handles container lifecycle operations. It is small and lightweight and can be used by other projects and third-party tools.

containerd needs to talk to an OCI-compliant container runtime to actually create containers. By default, Docker uses *runc* as its default container runtime. runc is the de facto implementation of the OCI container-runtime-spec and expects to start containers from OCI-compliant bundles. containerd talks to runc and ensures Docker images are presented to runc as OCI-compliant bundles.

runc can be used as a standalone tool to create containers. It can also be used by other projects and third-party tools.

There is still a lot of functionality implemented within the Docker daemon. More of this may be broken out over time. Functionality currently still inside of the Docker daemon include, but is not limited to: the API, image management, authentication, security features, core networking.

The work of modularizing the Docker engine is ongoing.

# 6: Images

In this chapter we'll dive into Docker images. The aim of the game is to give you a **solid understanding** of what Docker images are and how to perform basic operations. In a later chapter we'll see how to build new images with our own applications inside of them (containerizing an app).

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Let's go learn about images!

## Docker images - The TLDR

If you're a former VM admin you can think of Docker images as being like VM templates. A VM template is like a stopped VM - a Docker image is like a stopped container. If you're a developer you can think of them as being similar to *classes*.

You start by *pulling* images from an image registry. The most popular registry is Docker Hub, but others do exist. 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. Inside of the image is a cut-down operating system (OS) and all of the files and dependencies required to run an application. Because containers are intended to be fast and lightweight, images tend to be small.

Congrats! You've now got half a clue what a Docker image is :-D Now it's time to blow your mind!

## Docker images - The deep dive

We've mentioned a couple of times already that **images** are like stopped containers (or **classes** if you're a developer). 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.

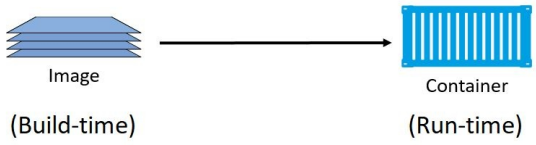



**Figure 6.1**

## Images and containers

Figure 6.1 shows high-level view of the relationship between images and containers. We use the `docker container run` and `docker service create` commands to start one or more containers from a single image. However, once you've started a container from 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 image rm <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>
```

## Images are usually small

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 all OS and application files required to run the app/service. 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 do not ship with 6 different shells for you to choose from - they usually ship with a single minimalist shell, or no shell at all. They also don't contain a kernel - all containers running on a Docker host share access to the host's kernel. For these reasons, we sometimes say images contain *just enough operating system* (usually just OS-related files and filesystem objects).

> **Note:** Hyper-V containers run inside of a dedicated lightweight VM and leverage the kernel of the OS running inside the VM.

The official *Alpine Linux* Docker image is about 4MB in size and is an extreme example of how small Docker images can be. That's not a typo! It really is about 4 megabytes! However, a more typical example might be something like the official Ubuntu Docker image which is currently about 120MB. These are clearly stripped of most non-essential parts!

Windows-based images tend to be bigger than Linux-based images because of the way that the Windows OS works. For example, the latest Microsoft .NET image (`microsoft/dotnet:latest`) is over 2GB when pulled an uncompressed. The Windows Server 2016 Nano Server image is slightly over 1GB when pulled and uncompressed.

## Pulling images

A cleanly installed Docker host has no images in its local repository.

The local image repository on a Linux-based Docker host is usually located at `/var/lib/docker/<storage-driver>`. On Windows-based Docker hosts this is `C:\ ProgramData\docker\windowsfilter`.

You can check if your Docker host has any images in its local repository with the following command.

```
$ docker image ls
REPOSITORY  TAG      IMAGE ID       CREATED         SIZE
```

The process 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* some images and then check their sizes.

> If you are following along on Linux and haven't added your user account to the local `docker` Unix group, you may need to add `sudo` to the beginning of all the following commands.

Linux example:

```
$ docker image pull ubuntu:latest

latest: Pulling from library/ubuntu
b6f892c0043b: Pull complete
55010f332b04: Pull complete
2955fb827c94: Pull complete
3deef3fcbd30: Pull complete
cf9722e506aa: Pull complete
Digest: sha256:38245....44463c62a9848133ecb1aa8
```

```
Status: Downloaded newer image for ubuntu:latest
$
$ docker image pull alpine:latest

latest: Pulling from library/alpine
cfc728c1c558: Pull complete
Digest: sha256:c0537...497c0a7726c88e2bb7584dc96
Status: Downloaded newer image for alpine:latest
$
$ docker image ls

REPOSITORY   TAG     IMAGE ID        CREATED       SIZE
ubuntu       latest  ebcd9d4fca80    3 days ago    118MB
alpine       latest  02674b9cb179    8 days ago    3.99MB
```

Windows example:

```
> docker image pull microsoft/powershell:nanoserver

nanoserver: Pulling from microsoft/powershell
bce2fbc256ea: Pull complete
58f68fa0ceda: Pull complete
04083aac0446: Pull complete
e42e2e34b3c8: Pull complete
0c10d79c24d4: Pull complete
715cb214dca4: Pull complete
a4837c9c9af3: Pull complete
2c79a32d92ed: Pull complete
11a9edd5694f: Pull complete
d223b37dbed9: Pull complete
aee0b4393afb: Pull complete
0288d4577536: Pull complete
8055826c4f25: Pull complete
Digest: sha256:090fe875...fdd9a8779592ea50c9d4524842
Status: Downloaded newer image for microsoft/powershell:nanoserver
>
> docker image pull microsoft/dotnet:latest

latest: Pulling from microsoft/dotnet
bce2fbc256ea: Already exists
4a8c367fd46d: Pull complete
9f49060f1112: Pull complete
0334ad7e5880: Pull complete
ea8546db77c6: Pull complete
710880d5cbd5: Pull complete
d665d26d9a25: Pull complete
caa8d44fb0b1: Pull complete
cfd178ff221e: Pull complete
Digest: sha256:530343cd483dc3e1...6f0378e24310bd67d2a
Status: Downloaded newer image for microsoft/dotnet:latest
>
> docker image ls
REPOSITORY            TAG         IMAGE ID    CREATED     SIZE
microsoft/dotnet      latest      831..686d   7 hrs ago   1.65 GB
microsoft/powershell  nanoserver  d06..5427   8 days ago  1.21 GB
```

As you can see, the images you just pulled are now present in your Docker host's local repository.

## Image naming

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 registry is Docker Hub (https://hub.docker.com). Other registries exist, including 3rd party registries and secure on-premises registries. However, the Docker client is opinionated and defaults to using Docker Hub. We'll be using Docker Hub for the rest of the book.

Image registries contain multiple *image repositories*. In turn, image repositories can contain multiple images. That might be a bit confusing, so Figure 6.2 shows a picture of an image registry containing 3 repositories, and each repository contains one or more images.

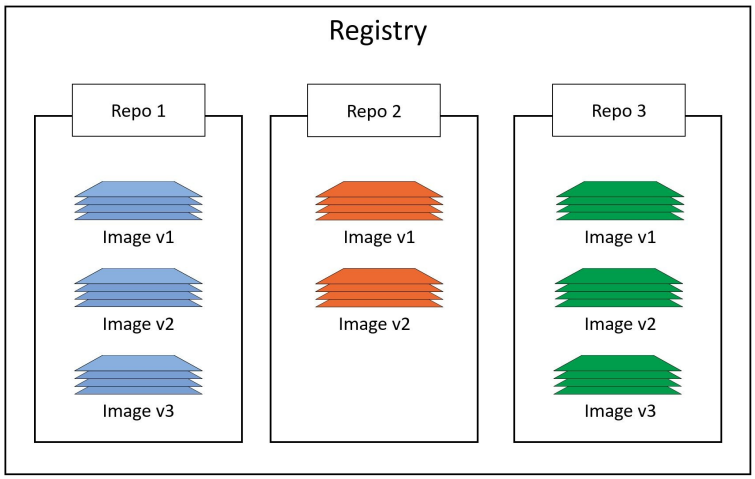



**Figure 6.2**

#### Official and unofficial repositories

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 code, that is secure, well-documented, and in-line with best practices (please can I have an award for using five hyphens in a single sentence).

*Unofficial repositories* can be like the wild-west - you should not *expect* them to be safe, well-documented or built according to best practices. That's not saying everything in *unofficial repositories* is bad! There's some **brilliant** 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`.

You'll probably notice that the Microsoft images we've used do not exist at the top-level of the Docker Hub namespace. At the time of writing, they exist under the `microsoft` second-level namespace.

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

## Image naming and tagging

Addressing images from official repositories is as simple as giving the repository name and tag separated by a colon (`:`). The format for `docker image pull` when working with an image from an official repository is:

```
docker image pull <repository>:<tag>
```

In the Linux examples from earlier, we pulled an Alpine and an Ubuntu images with the following two commands:

`docker image pull alpine:latest` and `docker image 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 image pull mongo:3.3.11
//This will pull the image tagged as `3.3.11`
//from the official `mongo` repository.

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

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

A couple of points to note about the commands above.

First, if you **do not** specify an image tag after the repository name, Docker will assume you are referring to the image tagged as `latest`.

Second, the `latest` tag doesn’t have any magical powers! Just because an image is tagged as `latest` does not guarantee it is the most recent image in a repository! For example, the most recent image in the `alpine` repository is usually tagged as `edge`. 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 a Docker Hub username or organization name. The example below shows how to pull the `v2` image from the `tu-demo` repository owned by a dodgy person whose Docker Hub account name is `nigelpoulton`.

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

In our earlier Windows examples, we pulled a PowerShell and a .NET image with the following two commands:

```
> docker image pull microsoft/powershell:nanoserver

> docker image pull microsoft/dotnet:latest
```

The first command pulls the image tagged as `nanoserver` from the `microsoft/powershell` repository. The second command pulls the image tagged as `latest` from the `microsoft/dotnet` repository.

If you want to pull images from 3rd party registries (not Docker Hub), 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` (no such repository and image exists).

You may need to have an account on 3rd party registries and be logged into them 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 a repository by adding the `-a` flag to them `docker image pull` command. Then run `docker image ls` to look at the images pulled. If you are following along with Windows you can pull from the `microsoft/nanoserver` repository instead of `nigelpoulton/tu-demo`.

> **Note:** If the repository you are pulling form contains images for multiple architectures and platforms, such as Linux **and** Windows, the command is likely to fail.

```
$ docker image 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 image ls
REPOSITORY            TAG       IMAGE ID      CREATED      SIZE
nigelpoulton/tu-demo   v2       6ac2...ad  12 months ago   211.6 MB
nigelpoulton/tu-demo   latest   9b91...29  12 months ago   211.6 MB
nigelpoulton/tu-demo   v1       9b91...29  12 months ago   211.6 MB
```

A couple of things about what just happened in the example cited above:

First. The command pulled three images from the repository: `latest`, `v1`, and `v2`.

Second. Look closely at the `IMAGE ID` column in the output of the `docker image ls` command. You'll see that there are only two unique image IDs. This is because only two images were actually downloaded. This in turn, is because two of the tags refer to the same image. 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 also a perfect example of the warning we issued earlier about the `latest` tag. In this example, the `latest` tag refers to the same image as the `v1` 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

A Docker image is just a bunch of loosely-connected read-only layers. This is shown in Figure 6.3.

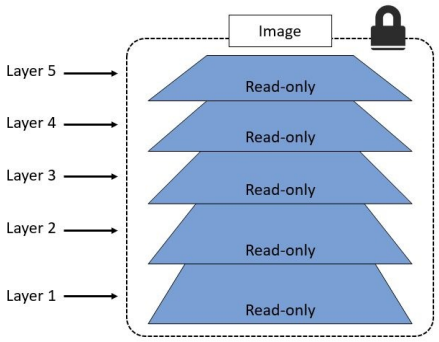



**Figure 6.3**

Docker takes care of stacking these layers and representing them as a single unified object.

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 image pull ubuntu:latest` command from earlier:

```
$ docker image 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 6.4 below shows this in picture form.

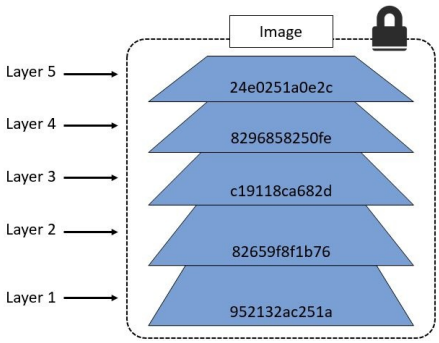



**Figure 6.4**

Another way to see the layers of an image is to inspect the image with the `docker image inspect` command. The example below inspects the same `ubuntu:latest` image.

```
$ docker image inspect ubuntu:latest
[
    {
        "Id": "sha256:bd3d4369ae.......fa2645f5699037d7d8c6b415a10",
        "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. However, 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 strict list of layers in the image. For example, some Dockerfile instructions used to build an image do not result in layers being created. These include; "MAINTAINER", "ENV", "EXPOSE" and "ENTRYPOINT". Instead of these creating new layers, they add metadata to the image.

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 new image based off 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 on top of the base layer. 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 6.5 (remember this is an over-simplified example for demonstration purposes).

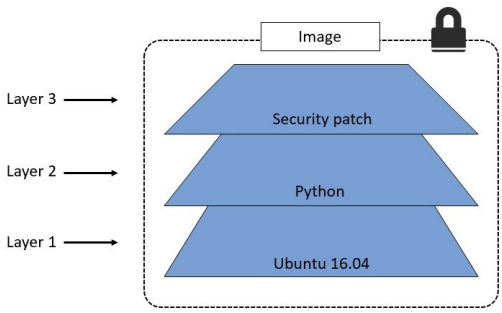



**Figure 6.5**

It's important to understand that as additional layers are added, the image becomes the combination of all layers. Take a simple example of two layers as shown in Figure 6.6. Each *layer* has 3 files, but the overall *image* has 6 files as it is the combination of both layers.

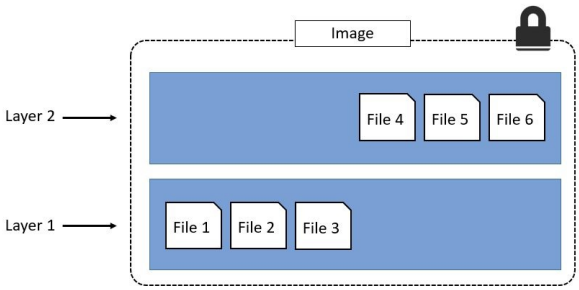



**Figure 6.6**

> We've shown the image layers in Figure 6.6 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 6.7, the overall image only presents 6 files in the unified view. This is because file 7 in the top layer is an updated version of file 5 directly below (inline). 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.

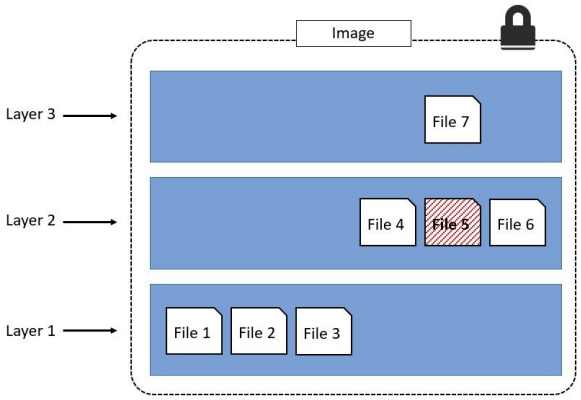



**Figure 6.7**

Docker employs a storage driver that is responsible for stacking layers and presenting them as a single unified filesystem. Examples of storage drivers on Linux include `AUFS`, `overlay2`, `devicemapper`, `btrfs` and `zfs`. As their names suggest, each one is based on a Linux filesystem or block-device technology, and each has its own unique performance characteristics. The only driver supported by Docker on Windows is `windowsfilter` and implements layering and CoW on top of NTFS.

## Sharing image layers

Multiple images can, and do, share layers. This leads to efficiencies in space and performance.

Let's take a second look at the `docker image 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 image 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 image ls
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`.

These lines tell us that Docker is smart enough recognize when it's being asked to pull an image layer that it already has a copy of. 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, and therefore share many layers.

As mentioned previously, Docker on Linux supports many different filesystems and storage drivers. Each is free to implement image layering, layer sharing, and copy-on-write behaviour in its own way. However, the overall result and user experience is essentially the same. Although Windows only supports a single storage driver, that driver provides the same experience as Linux.

## 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, but different, image. This can cause problems!

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 its 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. 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? Both images 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 a 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. This means digests are immutable. This helps avoid the problem we just talked about.

Every time you pull an image, the `docker image 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 repository by adding the `--digests` flag to the `docker image ls` command. These are both shown in the following example.

```
$ docker image 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 image ls --digests alpine
REPOSITORY  TAG     DIGEST              IMAGE ID      CREATED       SIZE
alpine      latest  sha256:3dcd...f73a  4e38e38c8ce0  10 weeks ago  4.8 MB
```

The snipped output above shows the digest for the `alpine` image as -

`sha256:3dcdb92d7432d56604d...6d99b889d0626de158f73a`

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 time of writing, there is no native Docker 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 its digest. This will no doubt 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 image rm alpine:latest
Untagged: alpine:latest
Untagged: alpine@sha256:c0537...7c0a7726c88e2bb7584dc96
Deleted: sha256:02674b9cb179d...abff0c2bf5ceca5bad72cd9
Deleted: sha256:e154057080f40...3823bab1be5b86926c6f860
$
$ docker image pull alpine@sha256:c0537...7c0a7726c88e2bb7584dc96
sha256:c0537...7726c88e2bb7584dc96: Pulling from library/alpine
cfc728c1c558: Pull complete
Digest: sha256:c0537ff6a5218...7c0a7726c88e2bb7584dc96
Status: Downloaded newer image for alpine@sha256:c0537...bb7584dc96
```

## A little bit more about image hashes (digests)

Since Docker version 1.10, an image is a very loose collection of independent layers.

The *image* itself is really just a configuration object that lists the layers and as well as some metadata.

The *layers* that make up an image are fully independent and have no concept of being part of a collective image.

Each image is identified by a crypto ID that is a hash of the config object. Each layer is identified by a crypto ID that is a hash of the content it contains.

This means that changing the contents of the image, or any of its layers, will cause the associated crypto hashes to change. As a result, images and layers are immutable.

We call these hashes **content hashes**.

So far, things are pretty simple. But they're about to get a bit more complicated.

When we push and pull images, we compress their layers to save bandwidth as well as space in the Registry's blob store.

Cool, but compressing a layer changes its content! This means that it's content hash will no longer match after the push or pull operation! This is obviously a problem.

For example, when you push an image layer to Docker Hub, Docker Hub will attempt to verify that the image arrived without being tampered with en-route. To do this, is runs a hash against the layer and checks to see if it matches the hash that was sent with the layer. Because the layer was compressed (changed) the hash verification will fail.

To get around this, each layer also gets something called a distribution hash. This is a hash of the compressed version of the layer. When a layer is pushed and pulled from the registry, its distribution hash is included, and this is what is used to verify that the layer arrived without being tampered with.

This content-addressable storage model vastly improves security by giving us a way to very image and layer data after push and pull operations. It also avoids ID collisions that could occur if image and layer IDs were randomly generated.

## Multi-architecture images

Docker now includes support for multi-platform and multi-architecture images. This means a single image repository and tag to have an image for Linux on x64 and Linux on PowerPC etc. Other examples exist.

To enable this, the Registry API supports a **fat manifest** as well as an **image manifest**. Fat manifests list the architectures supported by a particular image, whereas image manifests list the layers that make up a particular image.

Let’s look at a quick example.

Assume you are running Docker on Linux x64. When you pull an image from Docker hub, your Docker client makes the relevant API requests to the Docker Registry API running on Docker Hub. If a fat manifest exists for that image, it will be parsed to see if an entry exists for Linux on x64. If it exists, the image manifest for that image is retrieved and parsed for the actual layers that make up the image. The layers are identified by their crypto IDs and are pulled from the Registry’s blob store.

## Deleting Images

When you no longer need an image, you can delete it from your Docker host with the `docker image rm` command. `rm` is short for remove.

Delete the images pulled in the previous steps with the `docker image rm` command. The example below deletes an image by its ID, this might be different on your system.

```
$ docker image rm 02674b9cb179
Untagged: alpine@sha256:c0537ff6a5218...c0a7726c88e2bb7584dc96
Deleted: sha256:02674b9cb179d57...31ba0abff0c2bf5ceca5bad72cd9
Deleted: sha256:e154057080f4063...2a0d13823bab1be5b86926c6f860
```

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 image rm` command and pass it a list of all image IDs on the system by calling `docker image ls` with the `-q` flag. This is shown below.

If you are performing the following command on a Windows system, it will only work in a PowerShell terminal. It will not work on a CMD prompt.

```
$ docker image rm $(docker image ls -q) -f
```

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

```
$ docker image 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 image 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 image ls -q
bd3d4369aebc
4e38e38c8ce0
```

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

```
$ docker image rm $(docker image ls -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 image ls
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 image 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 image pull alpine:latest`.
- `docker image ls` 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 image inspect` is a thing of beauty! It gives you all of the glorious details of image - layer data and metadata.
- `docker image rm` is the command to delete images. This command shows how to delete the `apline:latest` image - `docker image rm 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 they are like virtual machine templates and are used to start containers. Under the hood they are made up one or more read-only layers that when stacked together make up the overall image.

We used the `docker image pull` command to pull some images into our Docker host's local registry.

We covered image naming, official and unofficial repos, layering, sharing, and crypto IDs.

We finished off by looking at some of the most common commands used to work with images.

In the next chapter we'll take a similar tour of containers - the runtime cousin of images.

# 7: Containers

Now that we know a bit about images, it's time to get into containers. As this is a book about Docker, we'll be talking specifically about Docker containers. However, the Docker project has been hard at work implementing the image and container specs published by the Open Containers 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.

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

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 a 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 share the OS/kernel with the host they're running on.

Figure 7.1 shows a single Docker image being used to start multiple Docker containers.

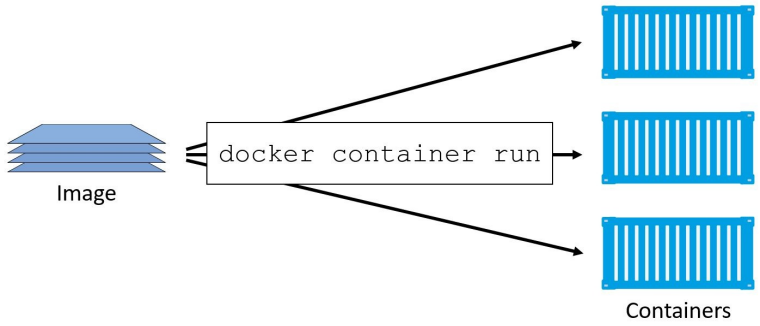



**Figure 7.1**

The simplest way to start a container is with the `docker container run` command. The command can take a lot of arguments, but in its most basic form you tell it an image to use and a command to run: `docker container run <image> <command>`. This next command will start an Ubuntu Linux container running the Bash shell: `docker container run -it ubuntu /bin/bash`. To start a Windows container running PowerShell you could do `docker container run -it microsoft/powershell:nanoserver PowerShell.exe`.

The `-it` flags used in the commands above will connect your current terminal window to the container's shell.

Containers run until the program they are executing exits. In the two examples above, the Linux container will exit when the Bash shell exits, and the Windows container will exit when the PowerShell process terminates.

A really simple way to demonstrate this is to start a new container and tell it to run the sleep command for 10 seconds. The container will start, run for 10 seconds and exit. If you run the following command from a Linux host (or Windows host running in Linux containers mode) your shell will attach to the container's shell for 10 seconds and then exit: `docker container run alpine:latest sleep 10`. You can do the same with a Windows container

with the following command `docker container run microsoft/powershell:nanoserver Start-Sleep -s 10`.

You can manually stop a container with the `docker container stop` command, and then restart it with `docker container start`. To get rid of a container forever you have to explicitly delete it using `docker container 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 it's 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. But 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 7.2.

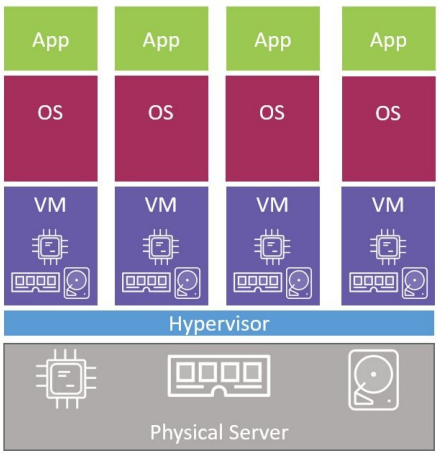



**Figure 7.2**

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 a modern 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 7.3.

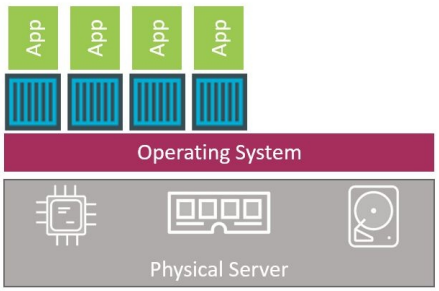



**Figure 7.3**

At a high level we can say that hypervisors perform **hardware virtualization** - they carve up physical hardware resources into virtual versions. On the other hand, 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 (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 its own OS to claim, initialize and manage all of those virtual resources. And sadly, every OS comes with its 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 has a single kernel running in the host OS. 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 consuming 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. What’s not to like about that!

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.

## 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:      17.05.0-ce
 API version:  1.29
 Go version:   go1.7.5
 Git commit:   89658be
 Built:        Thu May  4 22:10:54 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.05.0-ce
 API version:  1.29 (minimum version 1.12)
 Go version:   go1.7.5
 Git commit:   89658be
 Built:        Thu May  4 22:10:54 2017
 OS/Arch:      linux/amd64
 Experimental: false
```

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 you're running Linux and 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
```

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 container run` command.

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

```
$ docker container 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:/#
```

A Windows example could be

```
docker container run -it microsoft/powershell:nanoserver PowerShell.exe
```

The format of the command is essentially `docker container run -<options> <image>:<tag> <command>`.

Let's break the command down a bit.

We started with `docker container 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` or `microsoft/powershell:nanoserver` image. Finally we told it to run the Bash shell in the Linux example, and the PowerShell.exe program in the Windows example..

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 requested image. In this example it didn't, so it went to Docker Hub to see if it could find it there. It could, so it *pulled* it locally and stored it in its cache.

> **Note:** In a standard out-of-the-box Linux installation, the Docker daemon implements the Docker Remote API on a local IPC/Unix socket at `/var/run/docker.sock`. On Windows it listens on a named pipe at

`npipe:////./pipe/docker_engine`. It's also possible to configure the Docker client and daemon to operate over the network. The default non-TLS network port for Docker is 2375, the default TLS port is 2376.

Once the image was pulled, the daemon created the container and executed the specified command 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 container's unique ID.

Try executing some basic commands from inside of the container. You might notice that some commands do not work. This is because the images we used, like almost all container images, are highly optimized for containers. This means they don'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:/#
```

As shown in the output above, the `ping` utility is not included as part of the official Ubuntu image.

## Container processes

When we started the Ubuntu 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 aren't. 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 also exit (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. This is also true of Windows containers - killing the main process in the container will also kill the container.

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 container ls` command to view the list of running containers on your system.

```
$ docker container ls
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 container exec` command.

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

> **Note:** You can address a container by its name or ID. The command to re-attach to the Windows Nano Server PowerShell container would be `docker container exec -it <container-name-or-ID> PowerShell.exe`.

As you can see, the shell prompt has changed back to the container. If you run the `ps` command again you will now see **two** Bash or PowerShell processes. This is because the `docker container exec` command created a new Bash or PowerShell process and attached to that. This means that typing `exit` from this shell will not terminate the container because the original Bash or PowerShell process will continue running. The same works for the Windows PowerShell container.

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

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 container stop 3027eb64487
3027eb64487

$ docker container rm 3027eb64487
3027eb64487
```

The container(s) started in the previous examples will no longer be present on your system.

## 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 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 wonder if we're seeing something similar with the move to containerization - are there people out there trying to protect their empires of persistent workloads from what they perceive as the threat of containers?

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 container run` command. Let's start another one so we can walk it through its entire lifecycle. The examples below will be from a Linux Docker host running an Ubuntu container. However, all of the examples will work with the Windows PowerShell container we've used in previous examples - though you'll have to substitute Linux commands with their equivalent Windows commands.

```
$ docker container 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 "DevOps FTW" > newfile
root@9cb2d2fd1d65:/tmp#
root@9cb2d2fd1d65:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 14 May 23 11:22 newfile
root@9cb2d2fd1d65:/tmp#
root@9cb2d2fd1d65:/tmp# cat newfile
DevOps FTW
```

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

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

```
$ docker container stop percy
percy
```

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

Now run a `docker container ls` command to list all running containers.

```
$ docker container ls
CONTAINER ID   IMAGE   COMMAND   CREATED  STATUS  PORTS   NAMES
```

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

```
$ docker container ls -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 container start` command to bring it back from vacation.

```
$ docker container start percy
percy
$
$ docker container ls
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 container exec` command.

```
$ docker container 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.

While this example illustrates the persistent nature of containers, I should point out that *volumes* are the preferred way to store persistent 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 container 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 container stop percy
percy
$
$ docker container rm percy
percy
$
$ docker container ls -a
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 unikernel 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 container 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 in the Linux world will run a single process. In the Windows world they run a few processes, but the following rules still apply.

In our previous example the container was running the `/bin/bash` program. When you kill a running container with `docker container 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 its affairs before being killed.

However, the `docker container stop` command is far more polite (like pointing a gun to the containers head and saying "you've got 10 seconds to say any final words"). 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 container rm`.

The magic behind the scenes here can be explained with Linux/POSIX *signals*. `docker container stop` sends a **SIGTERM** signal to the 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 container 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 the 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 Linux 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. The image runs an insanely simple web server on port 8080.

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

```
$ docker container 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
```

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 container run` command, so let's take a quick look at them.

We know `docker container 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 **d**aemon mode, and `-d` and `-it` are mutually exclusive. This means you can't use both on the same container.

After that, we name the container and then give it `-p 80:8080`. The `-p` flag maps ports on the Docker host to ports inside 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: `nigelpoulton/pluralsight-docker-ci`. This image is not kept up-to-date and will contain security vulnerabilities!

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

> **Note:** We've removed some of the columns from 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 7.4 shows the web page that is being served up by the container.

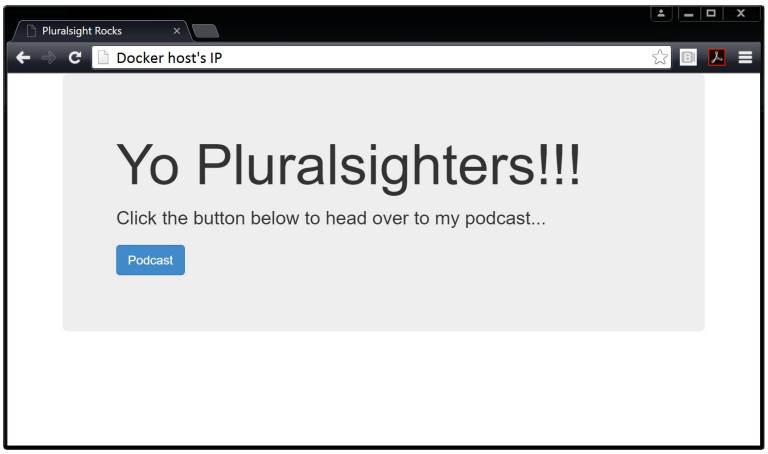



**Figure 7.4**

The same `docker container stop`, `docker container pause`, `docker container start`, and `docker container rm` commands can be used on the container. Also, 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 program for the container when we issued the `docker container run` command. 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 program you want containers using the image to run. If we run a `docker image inspect` command against the image we used to run our container, we'll be able to see the command/program that the container will run when it starts.

```
$ docker image 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 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 container run`. If you remove all of the shell escapes in the example above, you get the following command `/bin/sh -c "cd /src && node ./app.js"`. That's the default command a container based on this image will run.

It's common to build images with default commands like this as it makes starting containers easier. It also forces a default behavior and is a form of self documentation for the image - i.e. we can *inspect* the image and know what it's intended to do.

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 of **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 container rm $(docker container ls -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 image rm $(docker image ls -q)` command we used in the previous chapter to delete all images on a single Docker host. We already know the `docker container rm` command deletes containers. Passing it `$(docker container ls -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.

The above command will work in a PowerShell terminal on a Windows Docker host.

## Containers - The commands

- `docker container 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 container 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 container ls` 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 container 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 container exec -it <container-name or container-id> bash`. For this to work, the image used to create your container must contain the Bash shell.
- `docker container 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 container stop` accepts container IDs and container names as arguments.
- `docker container start` will restart a stopped `(Exited)` container. You can give `docker container start` the name or ID of a container.
- `docker container 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 container stop` command before deleting it with `docker rm`.
- `docker container inspect` will show you detailed configuration and runtime information about a container. It accepts container names and container IDs as its main argument.

## 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 container 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 container 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.

# 8: Containerizing an App

Docker is all about taking applications and running them in containers.

The process of taking an application and configuring it to run as a container is called “containerizing”. Sometimes we call it “Dockerizing”.

In this chapter we’ll walk through the process of containerizing a simple web application.

We’ll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Let’s go containerize an app!

## Containerizing an App - The TLDR

Containers are all about apps! In particular, they're about making apps simple to **build**, **ship**, and **run**.

The process of containerizing an app looks like this:

1. Start with your application code.
2. Create a *Dockerfile* that describes your app, its dependencies, and how to run it.
3. Feed this *Dockerfile* into the `docker image build` command.
4. Sit back while Docker builds your application into a Docker image.

Once your app is containerized (made into a Docker image), you're ready to ship it and run it as a container.

Figure 8.1 shows the process of building, shipping, and running an app (apologies for the colours on the diagram).

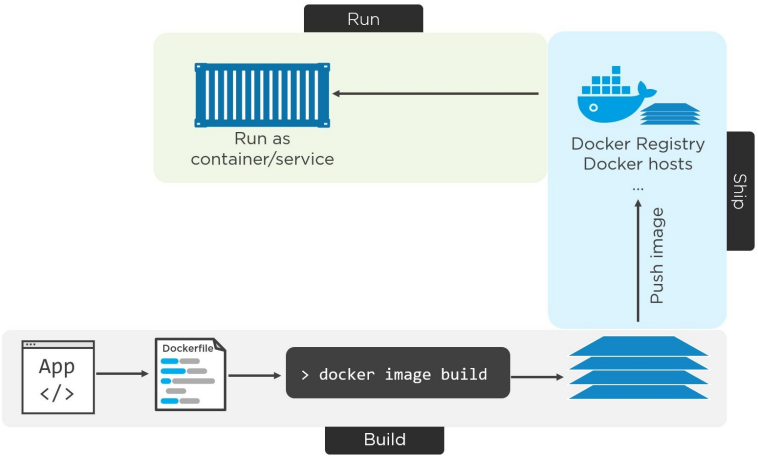



**Figure 8.1**

## Containerizing an App - The deep dive

The rest of this chapter will walk you through the process of containerizing a simple Linux-based Node.js web app. The process is the same for Windows, and future editions of the book will include a Windows example.

We'll complete the following high-level steps: - Get the app code - Inspect the Dockerfile - Containerize the app - Run the app - Test the app - Look a bit closer - Move to production with **Multi-stage Builds**

## Getting the application code

The application used in this example can be cloned form GitHub:

https://github.com/nigelpoulton/psweb.git

Clone the sample app from GitHub.

```
$ git clone https://github.com/nigelpoulton/psweb.git

Cloning into 'psweb'...
remote: Counting objects: 15, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 15 (delta 2), reused 15 (delta 2), pack-reused 0
Unpacking objects: 100% (15/15), done.
Checking connectivity... done.
```

The clone operation creates a new directory called `psweb`. Change directory into `psweb` and list its contents.

```
$ cd psweb
$
$ ls -l
total 28
-rw-r--r-- 1 root root  341 Sep 29 16:26 app.js
-rw-r--r-- 1 root root  216 Sep 29 16:26 circle.yml
-rw-r--r-- 1 root root  338 Sep 29 16:26 Dockerfile
-rw-r--r-- 1 root root  421 Sep 29 16:26 package.json
-rw-r--r-- 1 root root  370 Sep 29 16:26 README.md
drwxr-xr-x 2 root root 4096 Sep 29 16:26 test
drwxr-xr-x 2 root root 4096 Sep 29 16:26 views
```

This directory contains all of the application source code, as well as subdirectories for views and unit tests. Feel free to look at the files - the app is extremely simple. We won't be using the unit tests in this chapter.

Now that we have the app code, let's look at its Dockerfile.

## Inspecting the Dockerfile

Notice that the repo has a file called **Dockerfile**. This is the file that describes the application and tells Docker how to build an image from it.

The directory that contains your application code is referred to as the *build context*. It's a common practice to keep your Dockerfile in the root directory of the *build context*. It's also important that **Dockerfile** starts with a capital "**D**" and is all one word. "dockerfile" and "Docker file" are not valid.

Let's look at the contents of the Dockerfile.

```
$ cat Dockerfile

FROM alpine
LABEL maintainer="nigelpoulton@hotmail.com"
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

The Dockerfile has two main purposes:

1. To describe the application
2. To tell Docker how to containerize the application (create an image with the app inside)

Do not underestimate the impact of the Dockerfile from a documentation perspective. It has the ability to bridge the gap between dev and ops! It also has the power to speed up on-boarding of new developers etc. This is because the file accurately describes the application and its dependencies in an easy-to-read format.

At a high-level, the example Dockerfile says: Start with the `alpine` image, add "nigelpoulton@hotmail.com" as the maintainer, install Node.js and NPM, copy in the application code, set the working directory, install dependencies, expose a network port, and set `app.js` as the default application to run.

Let's look at it in a bit more detail.

All Dockerfiles start with the `FROM` instruction. This will be the base layer of the image, and the rest of the app will be added on top as additional layers. This particular application is a Linux app, so it is important that the FROM instruction refers to a Linux-based image. If you are containerizing a Windows application, you will need to specify the appropriate Windows base image - such as `microsoft/aspnetcore-build`.

At this point, the image looks like Figure 8.2 .

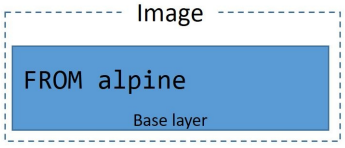


**Figure 8.2**

Next, the Dockerfile creates a LABEL and specifies "nigelpoulton@hotmail.com" as the maintainer of the image. Labels are simple key-value pairs and are an excellent way of adding custom metadata to an image. It is considered a best practice to list a maintainer of an image so that other potential users have a point of contact when working with it.

> **Note:** I will not be maintaining this image. I'm including the label to show you how to use labels as well as showing you a best practice.

The `RUN apk add --update nodejs nodejs-npm` instruction uses the Alpine `apk` package manager to install `nodejs` and `nodejs-npm` into the image. The RUN instruction installs these packages as a new image layer on top of the `alpine` base image created by the `FROM alpine` instruction. The image now looks like Figure 8.3.

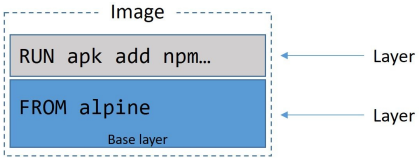



**Figure 8.3**

The `COPY . /src` instruction copies in the app files from the *build context*. The RUN instruction copies these files into the image as a new layer. The image now has three layers as shown in Figure 8.4.

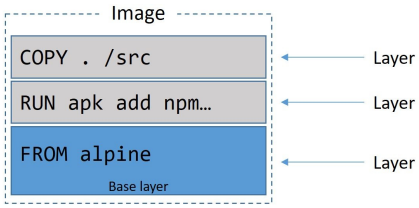



**Figure 8.4**

Next, the Dockerfile uses the `WORKDIR` instruction to set the working directory for the rest of the instructions in the file. This directory is relative to the image, and the info is added as metadata to the image config and not as a new layer.

Then the `RUN npm install` instruction uses `npm` to install application dependencies listed in `package.json`. It runs within the context of the `WORKDIR` set in the previous instruction, and installs the dependencies as a new layer in the image. The image now has four layers as shown in Figure 8.5.

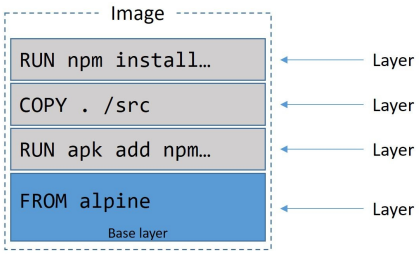



**Figure 8.5**

The application exposes a web service on TCP port 8080, so the Dockerfile documents this with the `EXPOSE 8080` instruction. This is added as image metadata and not an image layer.

Finally, the `ENTRYPOINT` instruction is used to set the main application that the image (container) should run. This is also added as metadata and not an image layer.

## Containerize the app/build the image

Now that we have the application code and the Dockerfile, let's build the image!

The following command will build a new image called `web:latest`. The period (`.`) at the end of the command tells Docker to use the shell's current working directory as the *build context*.

Be sure include the period (.) at the end of the command, and be sure to run the command from the `psweb` directory that contains the Dockerfile and application code.

```
$ docker image build -t web:latest .

Sending build context to Docker daemon  74.75kB
Step 1/8 : FROM alpine
latest: Pulling from library/alpine
88286f41530e: Pull complete
Digest: sha256:f006ecbb8...d935c0c103f4820a417d
Status: Downloaded newer image for alpine:latest
 ---> 76da55c8019d
<Snip>
Step 8/8 : ENTRYPOINT node ./app.js
 ---> Running in c576be4427a7
 ---> e33cdd8266d0
Removing intermediate container c576be4427a7
Successfully built e33cdd8266d0
Successfully tagged web:latest
```

Check that the image exists in your Docker host's local repository.

```
$ docker image ls
REPO    TAG       IMAGE ID          CREATED              SIZE
web     latest    e33cdd8266d0      About a minute ago   55.6MB
```

Congratulations, the app is containerized!

You can use the `docker image inspect web:latest` command to verify the configuration of the image. It will list all of the settings that were configured from the Dockerfile.

## Run the app

The example application that we've containerized is a simple web server that listens on TCP port 8080. You can verify this in the `app.js` file.

The following command will start a new container called `c1` based on the `web:latest` image we just created. It maps port `80` on the Docker host, to port `8080` inside the container. This means that you will be able to point a web browser at the DNS name or IP address of the Docker host and access the app.

> **Note:** If your host is already running a service on port 80, you can specify a different port as part of the `docker container run` command. For example, to map the app to port 5000 on the Docker host, use the `-p 5000:8080` flag.

```
$ docker container run -d --name c1 \
  -p 80:8080 \
  web:latest
```

The `-d` flag runs the container in the background, and the `-p 80:8080` flag maps port 80 on the host to port 8080 inside the running container.

Check that the container is running and verify the port mapping.

```
$ docker container ls

ID       IMAGE       COMMAND           <Snip>  PORTS
82...88  web:latest  "node ./app.js"   ...     0.0.0.0:80->8080/tcp
```

The output above is snipped for readability, but shows that the app container is running. Note that port 80 is mapped on all host interfaces (`0.0.0.0:80`) to port 8080 in the container.

## Test connectivity

Open a web browser and point it to the DNS name or IP address of the host that the container is running on. You will see the web page shown in Figure .

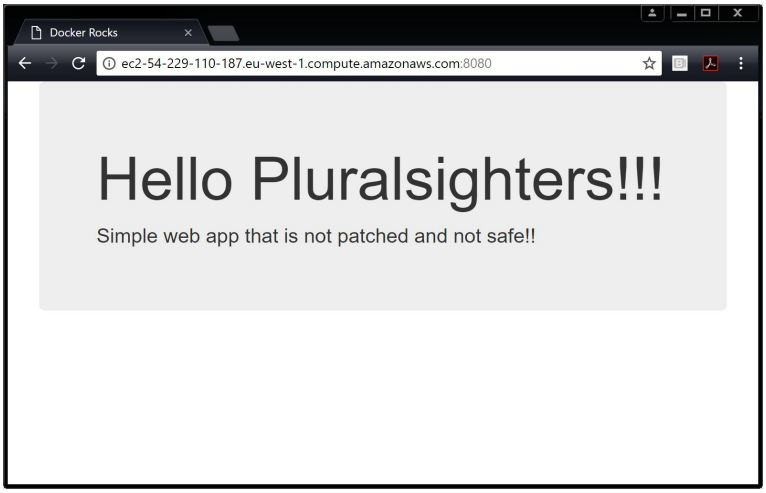



**Figure 8.6**

If the test does not work, try the following:

1. Make sure that the container is up and running with the `docker container ls` command. The container name is `c1` and you should see the port mapping as `0.0.0.0:80->8080/tcp`.
2. Check that the firewall and other network security settings are not blocking traffic to port 80 on the Docker host.

Congratulations, the application is containerized and running!

## Looking a bit closer

Now that the application is containerized, let's take a closer look at how some of the machinery works.

Comment lines in a Dockerfile start with the `#` character.

All non-comment lines are **Instructions**. Instructions take the format `INSTRUCTION argument`. Instruction names are not case sensitive, but it is normal practice to write them in UPPERCASE. This makes reading the Dockerfile easier.

The `docker image build` command parses the Dockerfile one-line-at-a-time starting from the top.

Some Instructions create new layers, whereas others just add metadata to the image.

Examples of instructions that create new layers are `FROM`, `RUN`, and `COPY`. Examples of instructions that create metadata include `EXPSOE`, `WORKDIR`, `ENV`, and `ENTRYPOINT`. The basic premise is this - if an instruction is adding *content* such as files and programs to the image, it will create a new layer. If it is

adding instructions on how to build the image and run the application, it will create metadata.

You can view the instructions that were used to build the image with the `docker image history command`.

```
$ docker image history web:latest

IMAGE     CREATED BY                                      SIZE
e33..6d0  /bin/sh -c #(nop)  ENTRYPOINT ["node" "./a...   0B
d38..20c  /bin/sh -c #(nop)  EXPOSE 8080/tcp              0B
e2a..0b6  /bin/sh -c npm install                          18.7MB
a8e..50e  /bin/sh -c #(nop) WORKDIR /src                  0B
23b..b58  /bin/sh -c #(nop) COPY dir:03b6808e26dacac...   22kB
fda..b35  /bin/sh -c apk add --update nodejs nodejs-npm   32.9MB
8d3..501  /bin/sh -c #(nop)  LABEL maintainer=nigelp...   0B
76d..19d  /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing> /bin/sh -c #(nop) ADD file:4583e12bf5caec4...   3.97MB
```

Two things from the output above are worth noting.

First, each line in the output corresponds to an instruction in the Dockerfile. The `CREATED BY` column even lists the exact instruction that was executed.

Second, only 4 of the image layers displayed in the output contain any data (the ones with non-zero values in the `SIZE` column). These correspond to the `FROM`, `RUN`, and `COPY` instructions in the Dockerfile. Although the other instructions look like they create layers, they actually create metadata instead of layers. The reason that the `docker image history` output makes it looks like all instructions create layers is an artefact of the way Docker builds used to work.

Use the `docker image inspect` command to confirm that only 4 layers were created.

```
$ docker image inspect web:latest

<Snip>
},
"RootFS": {
    "Type": "layers",
    "Layers": [
        "sha256:5bef08...00324f75e56f589aedb0",
        "sha256:03f8d2...f7061341ab09fab9d2d5",
        "sha256:7bb5e2...5718961a7a706c5d0085",
        "sha256:110b48...541f301505b0da017b34"
    ]
},
```

It is considered a good practice to use images from official repositories with the `FROM` instruction. This is because they tend to follow best practices and be relatively free from known vulnerabilities. It is also a good idea to start from (`FROM`) small images as this reduces the potential attack surface.

You can view the output of the `docker image build` command to see the general process for building an image. As the snippet below shows, the basic process is: `spin up a temporary container` > `run the Dockerfile instruction inside of that container` > `save the results as a new image layer` > `remove the temporary container`.

```
Step 3/8 : RUN apk add --update nodejs nodejs-npm
 ---> Running in 6f3..06d    <---- run inside temp container
fetch ...86_64/APKINDEX.tar.gz
fetch ...86_64/APKINDEX.tar.gz
(1/9) Installing...
<Snip>
 ---> fdaa341c6b35    <---- create layer
Removing intermediate container <---- remove temp container
Step 4/8...
```

## Moving to production with Multi-stage Builds

When it comes to Docker images, big is bad!

Big means slow. Big means hard to work with. And big means a large attack surface!

For these reasons, Docker images should be small. The aim of the game is to only ship production images containing the stuff **needed** to run your app in production.

The problem is… keeping images small *was* hard work.

For example, the way you write your Dockerfiles has a huge impact on the size of your images. A common example is that every `RUN` instruction adds a new layer. As a result, it's usually considered a best practice to include multiple commands as part of a single RUN instruction - all glued together with double-ampersands (&&) and backslash () line-breaks. While this isn't rocket science, it requires time and discipline.

Another issue is that we don't clean up after ourselves. We'll RUN a command against an image that pulls some build-time tools, and we'll leave all those tools in the image when we ship it to production. Not ideal!

There were ways around this - most notably the *builder pattern*. But most of these required discipline and added complexity.

The builder pattern required you to have at least two Dockerfiles - one for development and one for production. You'd write your Dockerfile.dev to start from a large base image, pull in any additional build tools required, and build your app. You'd then build an image from the Dockerfile.dev and create a container from it. You'd then use your Dockerfile.prod to build a new image from a smaller base image, and copy over the application form the container

you just created from the build image. And everything needed to be glued together with a script.

This approach was doable, but at the expense of complexity.

Multi-stage builds to the rescue!

Multi-stage builds are all about optimizing builds without adding complexity. And they deliver on the promise!

Here's the high-level…

With multi-stage builds, we have a single Dockerfile containing multiple FROM instructions. Each FROM instruction is a new **build stage** that can easily COPY artefacts from previous **stages**.

Let's look at an example!

This example app is available at https://github.com/nigelpoulton/atsea-sample-shop-app.git and the Dockerfile is in the `app` directory. It's a Linux-based application so will only work on a Linux Docker host.

The repo is a fork of `dockersamples/atsea-sample-shop-app` and I've forked it in case the upstream repo is removed or deleted.

The Dockerfile is shown below:

```
FROM node:latest AS storefront
WORKDIR /usr/src/atsea/app/react-app
COPY react-app .
RUN npm install
RUN npm run build

FROM maven:latest AS appserver
WORKDIR /usr/src/atsea
COPY pom.xml .
RUN mvn -B -f pom.xml -s /usr/share/maven/ref/settings-docker.xml dependency\
:resolve
COPY . .
RUN mvn -B -s /usr/share/maven/ref/settings-docker.xml package -DskipTests

FROM java:8-jdk-alpine AS production
RUN adduser -Dh /home/gordon gordon
WORKDIR /static
COPY --from=storefront /usr/src/atsea/app/react-app/build/ .
WORKDIR /app
COPY --from=appserver /usr/src/atsea/target/AtSea-0.0.1-SNAPSHOT.jar .
ENTRYPOINT ["java", "-jar", "/app/AtSea-0.0.1-SNAPSHOT.jar"]
CMD ["--spring.profiles.active=postgres"]
```

The first thing to note is that the Dockerfile has three `FROM` instructions. Each of these constitutes a distinct **build stage**. Internally they're numbered form the top starting at 0. However, we've also given each stage a friendly name.

Stage o is called `storefront` Stage 1 is called `appserver` Stage 2 is called `production`

The `storefront` stage pulls the `node:latest` image which is over 600MB in size. It sets the working directory, copies in some app code, and uses two RUN instructions to perform some `npm` magic. This adds three layers and considerable size. The result is an even bigger image containing lots of build stuff and not very much app code.

The `appserver` stage pulls the `maven:latest` image which is over 700MB in size. It adds four layers of content via two COPY instructions and two RUN instructions. This produces another very large image with lots of build tools and very little actual production code.

The `production` stage starts by pulling the `java:8-jdk-alpine` image. This image is approximately 150MB - considerably smaller than the node and maven images used by the previous build stages. It adds a user, sets the working directory and copies in some app code from the image produced by the `storefront` stage. After that, it sets a different working directory and copies in the application code form the image produced by the `appserver` stage. Finally, it sets the main application for the image to run when it's started as a container.

The important things to note are that the `COPY --from` instructions **only copy production-related application code** from the images built by the previous stages. They do not copy across build artefacts that are not needed for production.

It's also important to note that we only need a single Dockerfile, and no extra arguments are needed for the `docker image run` command!

Speaking of which… let's build it.

Clone the repo.

```
$ git clone https://github.com/nigelpoulton/atsea-sample-shop-app.git

Cloning into 'atsea-sample-shop-app'...
remote: Counting objects: 632, done.
remote: Total 632 (delta 0), reused 0 (delta 0), pack-reused 632
Receiving objects: 100% (632/632), 7.23 MiB | 1.88 MiB/s, done.
Resolving deltas: 100% (195/195), done.
Checking connectivity... done.
```

Change directory into the `app` folder of the cloned repo and verify that the Dockerfile exists.

```
$ cd atsea-sample-shop-app/app
$
$ ls -l
total 24
-rw-r--r-- 1 root root  682 Oct  1 22:03 Dockerfile
-rw-r--r-- 1 root root 4365 Oct  1 22:03 pom.xml
drwxr-xr-x 4 root root 4096 Oct  1 22:03 react-app
drwxr-xr-x 4 root root 4096 Oct  1 22:03 src
```

Perform the build (this may take several minutes to complete).

```
$ docker image build -t multi:stage .

Sending build context to Docker daemon  3.658MB
Step 1/19 : FROM node:latest AS storefront
latest: Pulling from library/node
aa18ad1a0d33: Pull complete
15a33158a136: Pull complete
<Snip>
Step 19/19 : CMD --spring.profiles.active=postgres
 ---> Running in b4df9850f7ed
 ---> 3dc0d5e6223e
Removing intermediate container b4df9850f7ed
Successfully built 3dc0d5e6223e
Successfully tagged multi:stage
```

> **Note:** The `multi:stage` tag used in the example above is arbitrary. You can tag your images according to your own requirements and standards - there is no requirement to tag multi-stage builds the way we did in this example.

Run a `docker image ls` to see the list of images pulled and created by the build operation.

```
$ docker image ls

REPO    TAG             IMAGE ID        CREATED         SIZE
node    latest          9ea1c3e33a0b    4 days ago      673MB
<none>  <none>          6598db3cefaf    3 mins ago      816MB
maven   latest          cbf114925530    2 weeks ago     750MB
<none>  <none>          d5b619b83d9e    1 min ago       891MB
java    8-jdk-alpine    3fd9dd82815c    7 months ago    145MB
multi   stage           3dc0d5e6223e    1 min ago       210MB
```

The top line in the output above shows the `node:latest` image pulled by the `storefront` stage. The image below is the image produced by that stage (created by adding the code and running the npm install and build operations). Both are very large images with lots of build tools included.

The 3rd and 4th lines are the images pulled and produced by the `appserver` stage. These are both large and contain lots of builds tools.

The last line is the `multi:stage` image built by the final build stage in the Dockerfile (stage2/production). You can see that this is significantly smaller than the images pulled and produced by the previous stages. This is because it’s based off the much smaller `java:8-jdk-alpine` image and has only added the production-related app files from the previous stages.

The net result is a small production image created by a single Dockerfile, a normal `docker image build` command, and zero additional scripting!

Multi-stage builds were new with Docker 17.05 and are an excellent feature for building small production-worthy images.

## Containerizing an app - The commands

- `docker image build` is the command that reads a Dockerfile and containerizes an application. The `-t` flag tags the image, the `-f` flag lets you specify the name and location of the Dockerfile. With the `-f` flag it is possible to use a Dockerfile with an arbitrary name. The *build context* is where your application files exist, and this can be a directory on your local Docker host or a remote Git repo.
- The `FROM` Dockerfile instruction in a Dockerfile specifies the base image for the new image you will build. It is usually the first instruction in a Dockerfile.
- The `RUN` Dockerfile instruction allows you to run commands inside the image which create new layers. Each `RUN` instruction creates a single new layer.
- The `COPY` Dockerfile instruction adds files into the image as a new layer. It is common to use the `COPY` instruction to copy your application code into an image.
- The `EXPOSE` Dockerfile instruction documents the network port that the application uses.
- The `ENTRYPOINT` Dockerfile instruction sets the default application to run when the image is started as a container.
- Other Dockerfile instructions include `LABEL`, `ENV`, `ONBUILD`, `HEALTHCHECK`, `CMD` and more…

## Chapter summary

In this chapter we learned how to containerize (Dockerize) an application.

We pulled some application code from a remote Git repo. The repo included the application code, as well as a Dockerfile containing instructions on how to build the application into an image. We learned the basics of the how Dockerfiles work, and fed one into a `docker image build` command to create a new image.

Once the image was created, we started a container form it and tested it worked with a web browser.

After that, we saw how multi-stage builds give us a simple way to build and ship smaller images to our production environments.

We also learned that the Dockerfile is a great tool for documenting an app. As such, it can speed-up the on-boarding of new developers and bridge the divide between developers and operations staff!

Although the examples cited was a Linux-based example, the process for containerizing Windows apps is the same: Start with your app code, create a Dockerfile describing the app, build the image with `docker image build`. Job done!

# 9: 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.

The examples and outputs in this chapter will be from a Linux-based Swarm. However, all commands and features also work with Docker on Windows.

> **Note:** If you are following along with Windows in a PowerShell terminal and listing command options over multiple lines, you will need to indicate continuation on the next line with backticks (`).

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Let's go build a swarm!

## 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 into play!

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. All this orchestration stuff got 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, most notably Kubernetes, 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 of Docker hosts that's deeply integrated into the Docker platform. 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 hosts a *swarm*, and the Docker hosts 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 hosts that are not running in *swarm mode*.

Figure 9.1 shows a 4-node *swarm* with nodes running in *swarm mode*. It also shows two nodes not in the swarm operating in *single-engine mode*.

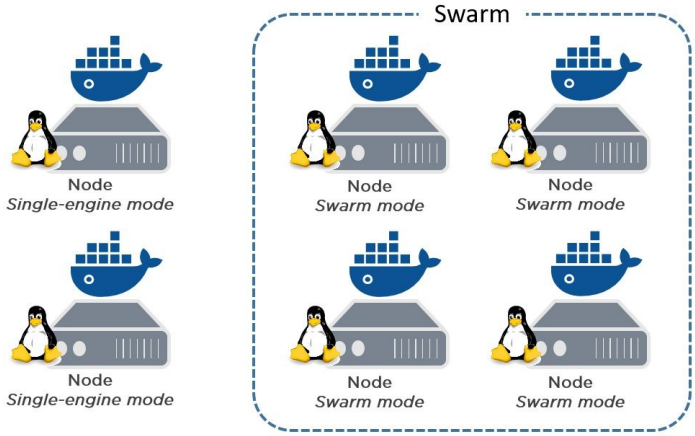



**Figure 9.1**

## Backward compatibility

Introducing *swarm mode* was extremely important for Docker, Inc. But so is maintaining backward compatibility! As a result, **swarm mode is entirely**

**optional** (this may or may not change in the future). A standard installation of Docker will 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 etc. 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 hosts 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 some 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*. At the highest level, *services* are the way to run tasks on a Swarm. To run a task (container) on Swarm we wrap it in a service and deploy the service. Beneath the hood, services are a declarative way of setting the *desired state* on the cluster. For example:

- Set the number of tasks (containers) in the 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's installed as part of the Swarm and 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 9.2 with 6-nodes configured as 3 managers and 3 workers. In the examples we'll use, each node is running Linux with Docker 17.05 or higher. You can build the same swarm on Windows, the only difference would be that you'd need to deploy Windows containers to it. All nodes in the lab can communicate over the network.

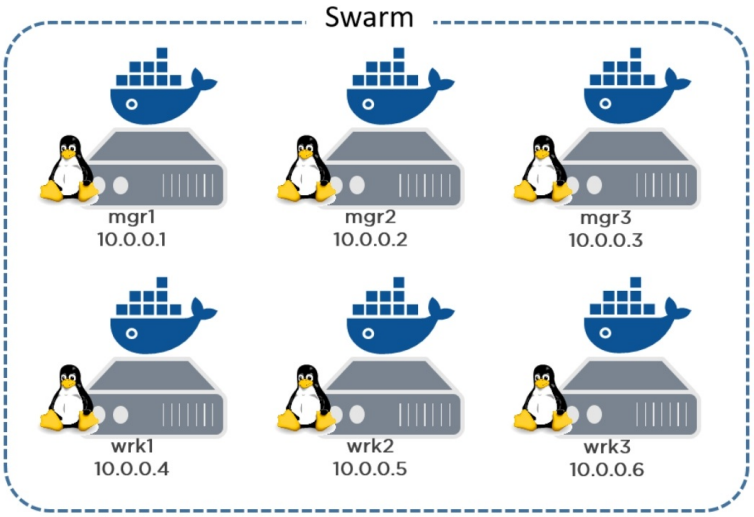



**Figure 9.2**

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 in *single-engine mode* will switch that node into *swarm mode* and create a new *swarm*. It will also make the node the first *manager* of the Swarm.

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*.

This example will use the IP addresses and DNS names of the nodes shown in Figure 9.2. Yours may be different.

1. Log on to **mgr1** and initialize a new swarm (don't forget to use backticks instead of backslashes if following along with Windows in a PowerShell terminal).

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

Swarm initialized: current node (d21lyz...c79qzkx) is now a manager.
```

   The command can be broken down as follows:

   - `docker swarm init` tells Docker to initialize a new Swarm 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 be specific and always use both flags. All of the Windows Swarm work I've done has required these two 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. You should also protect your join tokens as these are all that is required to join a node to a Swarm.

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 a 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 probably don't need to use the flags.

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

## Swarm manager high availability (H/A)

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 (H/A). 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 H/A. 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 9.3 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.

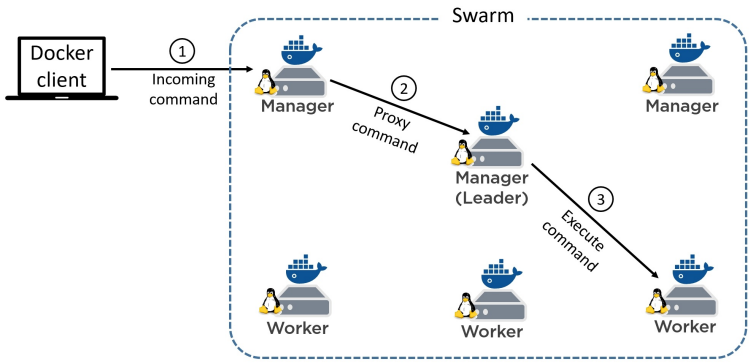



**Figure 9.3**

Swarm uses an implementation of the [Raft consensus algorithm] 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 (3 or 5 is recommended)

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 any 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.

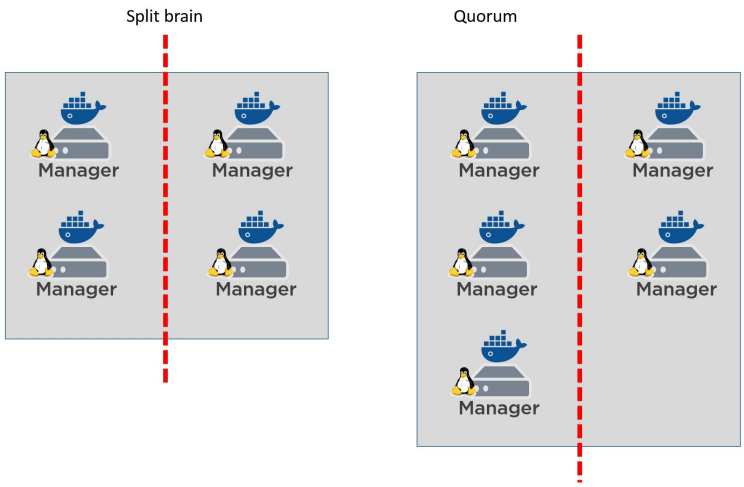



**Figure 9.4**

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 an application service 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 service, 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 what we just described.

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

> **Note:** The command to create a new service is the same on Windows. However, the image used in the example below is a Linux image and will not work on Windows. You can substitute the image for a Windows web server image and the command will work. Remember, if you are typing Windows commands form a PowerShell terminal you will need to use the backtick (`) character to indicate continuation on the next line.

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

z7ovearqmruwk0u2vc5o7ql0p
```

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 container (task) 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 of 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 action is needed. 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, so 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     MODE        REPLICAS   IMAGE               PORTS
z7o...uw  web-fe   replicated  5/5        nigel...ci:latest   *:8080->8080/t\
cp
```

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 (container) 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:             z7ovearqmruwk0u2vc5o7ql0p
Name:           web-fe
Service Mode:   Replicated
 Replicas:      5
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
```

```
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:   nigelpoulton/pluralsight-docker-ci:latest@sha256:7a6b01...d8d3d
Resources:
Endpoint Mode:  vip
Ports:
 PublishedPort = 8080
  Protocol = tcp
  TargetPort = 8080
  PublishMode = ingress
```

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
ID        NAME     MODE        REPLICAS   IMAGE               PORTS
z7o...uw  web-fe   replicated  10/10      nigel...ci:latest   *:8080->8080/t\
cp
```

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
nwf...tpn  web-fe.1  nigelpoulton/...  mgr1  Running  Running 7 mins
yb0...e3e  web-fe.2  nigelpoulton/...  wrk3  Running  Running 7 mins
mos...gf6  web-fe.3  nigelpoulton/...  wrk2  Running  Running 7 mins
utn...6ak  web-fe.4  nigelpoulton/...  wrk3  Running  Running 7 mins
2ge...fyy  web-fe.5  nigelpoulton/...  mgr3  Running  Running 7 mins
64y...m49  web-fe.6  igelpoulton/...   wrk3  Running  Running about a min
ild...51s  web-fe.7  nigelpoulton/...  mgr1  Running  Running about a min
vah...rjf  web-fe.8  nigelpoulton/...  wrk2  Running  Running about a mins
xe7...fvu  web-fe.9  nigelpoulton/...  mgr2  Running  Running 45 seconds ago
l7k...jkv  web-fe.10 nigelpoulton/...  mgr2  Running  Running 46 seconds ago
```

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    MODE   REPLICAS    IMAGE      PORTS
```

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 want you to see how it is 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 9.5 shows two underlay networks connected by a layer 3 router. There is then a single overlay network across both. 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.

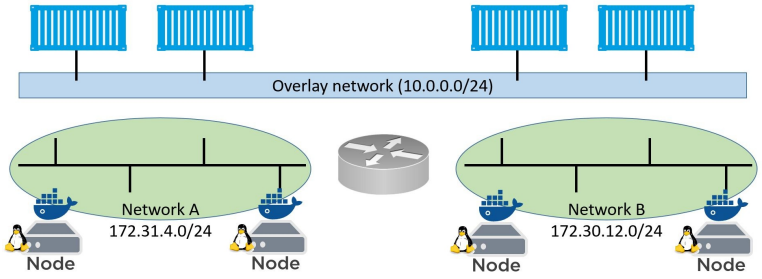



**Figure 9.5**

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.

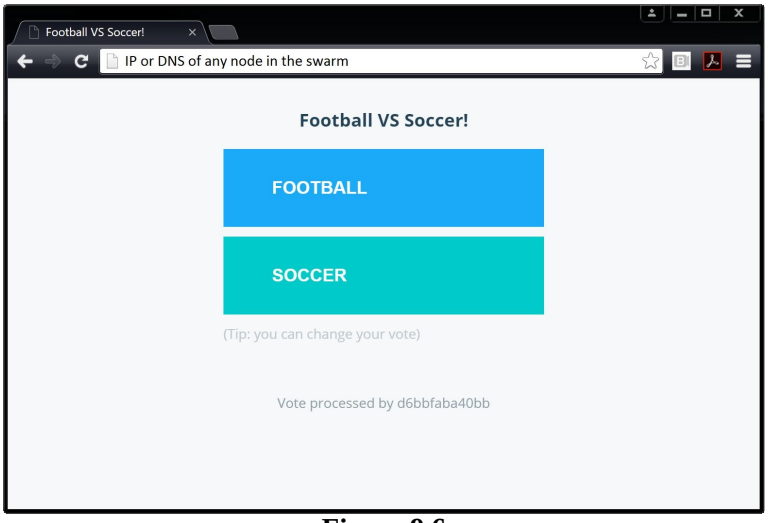



**Figure 9.6**

As you can see, the application is a simple voting application that will register votes for either “football” or “soccer”. Feel free to point your 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` flag creates a mapping on every host.

This is true even on nodes that are not 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 swarm 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-delay` 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 serviced 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:             mub0dgtc8szm80ez5bs8wlt19
Name:           uber-svc
Service Mode:   Replicated
 Replicas:      12
UpdateStatus:
 State:         updating
 Started:       About a minute
 Message:       update in progress
Placement:
UpdateConfig:
 Parallelism:   2
 Delay:         20s
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:    nigelpoulton/tu-demo:v2@sha256:d3c0d8c9...cf0ef2ba5eb74c
Resources:
Networks: uber-net
Endpoint Mode:  vip
Ports:
 PublishedPort = 80
  Protocol = tcp
  TargetPort = 80
  PublishMode = ingress
```

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 `uber-net` network as well as the swarm-wide `80:80` port mapping.

## The future of services

Services are the preferred way to deploy and manager Dockerized applications. We should expect to see significant development around them.

NEED TO UPDATE THIS. 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.

## 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 platform, 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
```

Remove the uber-net network with `docker network rm uber-net`.

## 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 <service>` 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.

# 10: Docker overlay networking

Container networking is increasingly important. Especially in production environments.

In this chapter we'll cover the fundamentals of native Docker overlay networking as implemented in a Docker swarm cluster.

At the time of writing this revision of the book, Docker networking on Windows has come a long way. The examples we'll use in this chapter will all work on Windows as well as Linux.

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Let's go do some networking magic!

## Docker overlay networking - The TLDR

In the real world it's vital that containers can communicate with each other reliably and securely, even when they're on different hosts on different networks. This is where overlay networking comes in to play. It allows you to create a flat secure layer 2 network spanning multiple hosts that containers can connect to. Containers on this network can then communicate directly.

Docker offers native overlay networking that is simple to configure and secure by default.

Behind the scenes Docker networking is comprised of `libnetwork` and `drivers`. Libnetwork is the canonical implementation of the Container Network Model (CNM) and drivers are pluggable components that implement different networking technologies and topologies. Docker offers native drivers such as the `overlay` driver, and third parties also offer drivers.

## Docker overlay networking - The deep dive

In March 2015 Docker, Inc. acquired container networking startup *Socket Plane*. Two of the reasons behind the acquisition were to bring *real networking* to Docker, and to make container networking so simple that even developers could do it :-P

They're making great progress on both accounts.

But hiding behind the simple networking commands are a lot of moving parts. The kind of stuff you need understand before doing production deployments and attempting to troubleshoot issues!

The rest of this chapter will be broken into two parts:

- Part 1: we'll build and test a Docker overlay network in swarm mode
- Part 2: We'll explain the theory behind how it works.

## Build and test a Docker overlay network in swarm mode

For the following examples we'll use two Docker hosts on two separate Layer 2 networks connected by a router as shown in Figure 10.1.

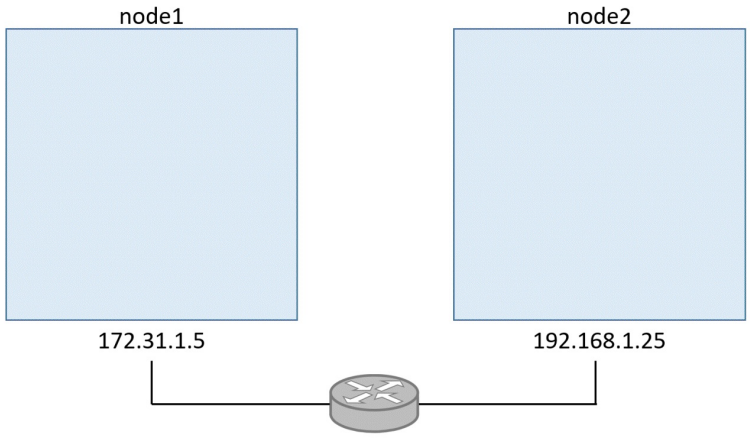



**Figure 10.1**

You can follow along with either Linux or Windows Docker hosts. Linux should have a 4.4 Linux kernel (newer is always better) and Windows should be Windows Server 2016 with the latest hotfixes installed. The examples in the book have been tested using Docker 17.05 on Linux and 17.03 on Windows.

#### Build a swarm

The first thing we'll do is configure the two hosts into a two-node Swarm. We'll run the `docker swarm init` command on **node1** to make it a *manager*,

and then we'll run the `docker swarm join` command on **node2** to make it a *worker*.

> Warning: If you are following along in your own lab you'll need to swap the IP addresses, container IDs, tokens etc. with the correct values for your environment.

Run the following command on **node1**.

```
$ docker swarm init \
  --advertise-addr=172.31.1.5 \
  --listen-addr=172.31.1.5:2377

Swarm initialized: current node (1ex3...o3px) is now a manager.
```

To add a worker to this swarm, run the following command:

```
    docker swarm join \
    --token SWMTKN-1-0hz2ec...2vye \
    172.31.1.5:2377
```

Run the next command on **node2**. For this command to work on Windows Server you may need to modify your firewall rules to allow ports `2377/tcp`, `7946/tcp` and `7946/udp`.

```
$ docker swarm join \
  --token SWMTKN-1-0hz2ec...2vye \
  172.31.1.5:2377
This node joined a swarm as a worker.
```

We now have a two-node Swarm where **node1** is a manager and **node2** is a worker.

#### Create a new overlay network

Now let's create a new *overlay network* called **uber-net**.

Run the following command from **node1** that was just created as a *manager*. For this to work on Windows you may need to add a rule for port `4789/udp` on your Windows Docker hosts.

```
$ docker network create -d overlay uber-net
c740ydi1lm89khn5kd52skrd9
```

That's it! You've just created a brand-new overlay network that is available to all hosts in the swarm and has its control plane encrypted with TLS! If you want to encrypt the data plane you can just add the `-o encrypted` flag to the command.

You can list all networks on each node with the `docker network ls` command.

```
$ docker network ls
NETWORK ID      NAME              DRIVER     SCOPE
ddac4ff813b7    bridge            bridge     local
389a7e7e8607    docker_gwbridge   bridge     local
a09f7e6b2ac6    host              host       local
ehw16ycy980s    ingress           overlay    swarm
2b26c11d3469    none              null       local
c740ydi1lm89    uber-net          overlay    swarm
```

The output will look more like this on a Windows server:

```
NETWORK ID      NAME              DRIVER      SCOPE
8iltzv6sbtgc    ingress           overlay     swarm
6545b2a61b6f    nat               nat         local
96d0d737c2ee    none              null        local
nil5ouh44qco    uber-net          overlay     swarm
```

The network we created is at the bottom of the list called **uber-net**. The other networks were automatically created when Docker was installed and when we created the swarm. We're only interested in the **uber-net** overlay network.

If you run the `docker network ls` command on **node2** you'll notice that it can't see the **uber-net** network. This is because new overlay networks are only made available to worker nodes that are running containers attached to the overlay. This reduces the scope of the network gossip protocol and helps with scalability.

#### Attach a service to the overlay network

Let's create a new *Docker service* and attach it to the **uber-net** overlay network. We'll create the service with two replicas (containers) so that one runs on **node1** and the other runs on **node2**. This will automatically extend the **uber-net** overlay to **node2**

Run the following commands from **node1**.

Linux example:

```
$ docker service create --name test \
   --network uber-net \
   --replicas 2 \
   ubuntu sleep infinity
```

Windows example:

```
> docker service create --name test `
  --network uber-net `
  --replicas 2 `
  microsoft\powershell:nanoserver Start-Sleep 3600
```

> **Note:** The Windows example above uses the backtick character to split parameters over multiple lines to make the command more readable.

The command creates a new service called **test**, attaches it to the **uber-net** overlay network, and creates two containers (replicas) based on the image provided. In both examples we issued a sleep command to the container to keep them running and stop them from exiting.

Because we're running two containers (replicas) and the Swarm has two nodes, one container will run on each node.

Verify the operation with a `docker service ps` command.

```
$ docker service ps test
ID          NAME    IMAGE   NODE    DESIRED STATE  CURRENT STATE
77q...rkx   test.1  ubuntu  node1   Running        Running
97v...pa5   test.2  ubuntu  node2   Running        Running
```

When Swarm starts a container on an overlay network it automatically extends that network to the node the container is running on. This means that the **uber-net** network is now visible on **node2**.

Congratulations! You've created a new overlay network spanning two nodes on separate physical underlay networks, and you've scheduled two containers on the network. How simple was that!

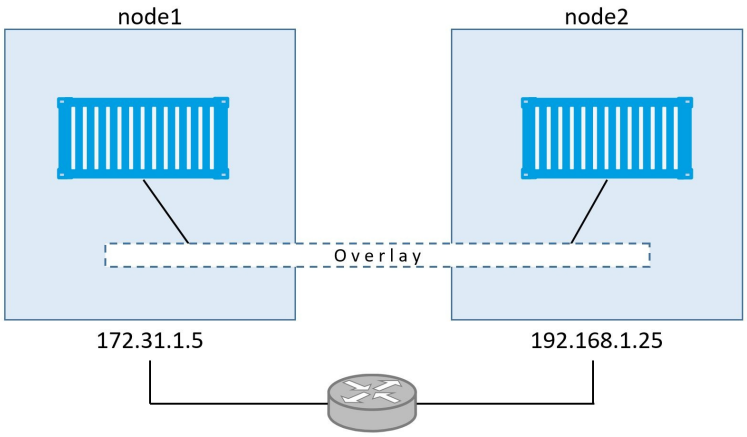



**Figure 10.2**

## Test the overlay network

Now let's test the overlay network with the ping command.

To do this, we need to do a bit of digging around to get each container's IP address.

Run a `docker network inspect` to see the **Subnet** assigned to the overlay.

```
$ docker network inspect uber-net
[
    {
        "Name": "uber-net",
        "Id": "c740ydi1lm89khn5kd52skrd9",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.0.0.0/24",
                    "Gateway": "10.0.0.1"
                }
<Snip>
```

The output above shows that **uber-net**'s subnet is `10.0.0.0/24`. Note that this does not match either of the physical underlay networks (`172.31.1.0/24` and `192.168.1.0/24`).

Run the following two commands on **node1** and **node2** to get the container ID's and their IP addresses. Be sure to use the CONTAINER ID from your own lab in the second command.

```
$ docker container ls
CONTAINER ID   IMAGE           COMMAND            CREATED       STATUS
396c8b142a85   ubuntu:latest   "sleep infinity"   2 hours ago   Up 2 hrs
$
$ docker container inspect \
  --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' 396c8b\
142a85
10.0.0.3
```

Make sure you run these commands on both nodes to get the IP addresses of both containers.

Figure 10.3 shows the configuration so far.

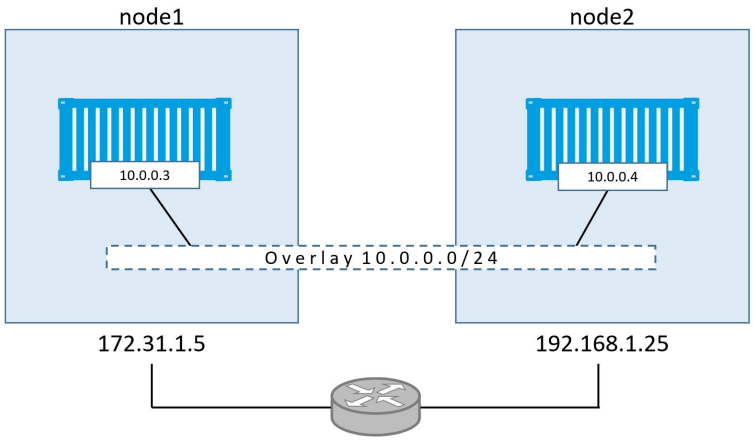



**Figure 10.3**

As we can see, there is a Layer 2 overlay network spanning both hosts, and each container has an IP address on this overlay network. This means that the

container on **node1** will be able to ping the container on **node2** using its `10.0.0.4` address from the overlay network. This works despite the fact that both nodes are on separate Layer 2 underlay networks. Let's prove it.

Log on to the container on **node1** and ping the remote container.

To do this on the Linux Ubuntu container you will need to install the `ping` utility. If you're following along with the Windows PowerShell example the `ping` utility is already installed.

Remember that the container IDs used below will be different in your environment.

Linux example:

```
$ docker container exec -it 396c8b142a85 bash
root@396c8b142a85:/#
root@396c8b142a85:/#
root@396c8b142a85:/# apt-get update
<Snip>
root@396c8b142a85:/#
root@396c8b142a85:/#
root@396c8b142a85:/# apt-get install iputils-ping
Reading package lists... Done
Building dependency tree
Reading state information... Done
<Snip>
Setting up iputils-ping (3:20121221-5ubuntu2) ...
Processing triggers for libc-bin (2.23-0ubuntu3) ...
root@396c8b142a85:/#
root@396c8b142a85:/#
root@396c8b142a85:/# ping 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=1.06 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=1.07 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=1.03 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=1.26 ms
^C
root@396c8b142a85:/#
```

Windows example:

```
> docker container exec -it 1a4f29e5a4b6 PowerShell.exe
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\>
PS C:\> ping 10.0.0.4

Pinging 10.0.0.4 with 32 bytes of data:
Reply from 10.0.0.4: bytes=32 time=1ms TTL=128
Reply from 10.0.0.4: bytes=32 time<1ms TTL=128
Reply from 10.0.0.4: bytes=32 time=2ms TTL=128
Reply from 10.0.0.4: bytes=32 time=2ms TTL=12
PS C:\>
```

As shown above, the container on **node1** can ping the container on **node2** using the overlay network.

You can also trace the route of the ping command from within the container. This will report only a single hop, proving that the containers are communicating directly over the overlay network - blissfully unaware of any underlay networks that are being traversed.

> For the traceroute to work on the Linux example you will need to install the traceroute package.

Linux example:

```
$ root@396c8b142a85:/# traceroute 10.0.0.4
traceroute to 10.0.0.4 (10.0.0.4), 30 hops max, 60 byte packets
 1  test-svc.2.97v...a5.uber-net (10.0.0.4)  1.110ms  1.034ms  1.073ms
```

Windows example:

```
PS C:\> tracert 10.0.0.3

Tracing route to test.2.ttcpiv3p...7o4.uber-net [10.0.0.4]
over a maximum of 30 hops:

  1  <1 ms  <1 ms  <1 ms  test.2.ttcpiv3p...7o4.uber-net [10.0.0.4]

Trace complete.
```

So far, we've created an overlay network with a single command. We then added containers to the overlay network on two hosts on two different Layer 2 networks. Once we worked out the container's IP addresses, we proved that they could talk directly over the overlay network.

## The theory of how it all works

Now that we've seen how to build and use a container overlay network, let's find out how it's all put together behind the scenes.

> **Note:** In this section some of the detail will be specific to Linux. However, the same overall principles apply to Windows.

#### VXLAN primer

First and foremost, Docker overlay networking uses VXLAN tunnels to create virtual Layer 2 overlay networks. So, before we go any further, let's do a quick VXLAN primer.

At the highest level, VXLANs let you create a virtual Layer 2 network on top of an existing Layer 3 infrastructure. The example we used earlier created a new 10.0.0.0/24 Layer 2 network on top of a Layer 3 IP network comprising

two Layer 2 networks - 172.31.1.0/24 and 192.168.1.0/24. This is shown in Figure 10.4 below.

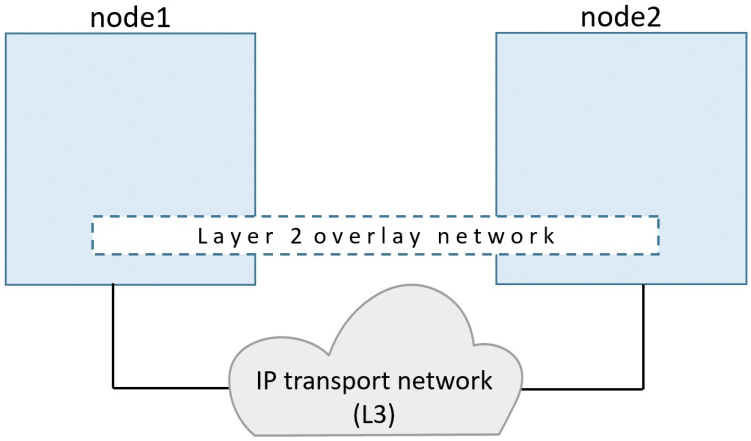



**Figure 10.4**

The beauty of VXLAN is that it's an encapsulation technology that existing routers and network infrastructure just see as regular IP/UDP packets and handle without issue.

To create the virtual Layer 2 overlay network a VXLAN *tunnel* is created through the underlying Layer 3 IP infrastructure. You might hear the term *underlay network* used to refer to the underlying Layer 3 infrastructure.

Each end of the VXLAN tunnel is terminated by a VXLAN Tunnel Endpoint (VTEP). It's this VTEP that performs the encapsulation/de-encapsulation and other magic required to make all of this work. See Figure 10.5.

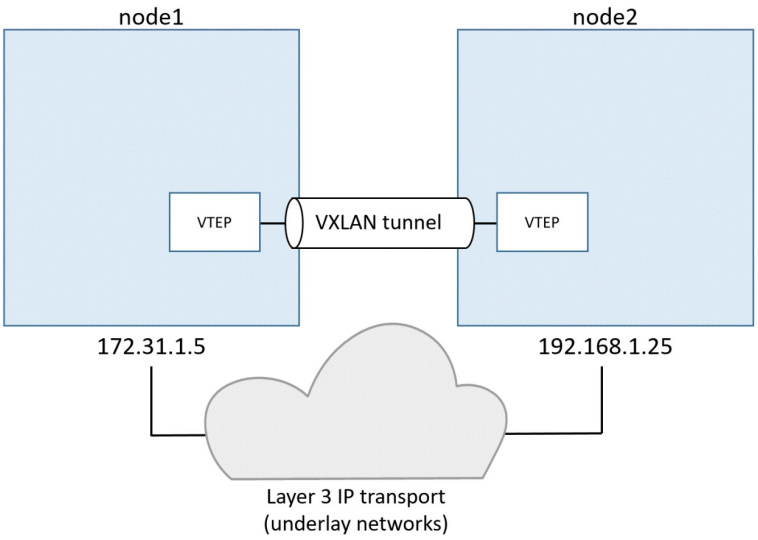



**Figure 10.5**

#### Walk through our two-container example

In the example we built earlier, we had two hosts connected via an IP network. Each host ran a single container, and we created a single VXLAN

overlay network for the containers to connect to.

To accomplish this, a new *network namespace* was created on each host. A *network namespace* is like a container, but instead of running an application it runs an isolated network stack - one that's sandboxed from the network stack on the host itself.

A virtual switch (a.k.a. virtual bridge) called **Br0** is created inside the network namespace. A VTEP is also created with one end plumbed into the **Br0** virtual switch, and the other end plumbed into the host network stack. The end in the host network stack gets an IP address on the underlay network the host is connected to and is bound to a UDP socket on port 4789. The two VTEPs on each host create the overlay via a VXLAN tunnel as seen in Figure 10.6.

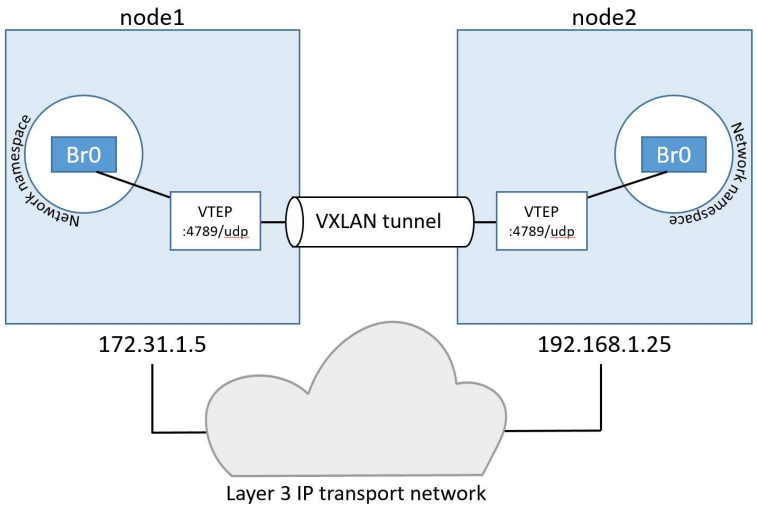



**Figure 10.6**

This is essentially the VXLAN overlay network created and ready for use.

Each container then gets its own virtual Ethernet (veth) adapter that is also plumbed into the local **Br0** virtual switch. The topology now looks like Figure 10.7, and it should be getting easier to see how the two containers can communicate over the VXLAN overlay network despite their hosts being on two separate networks.

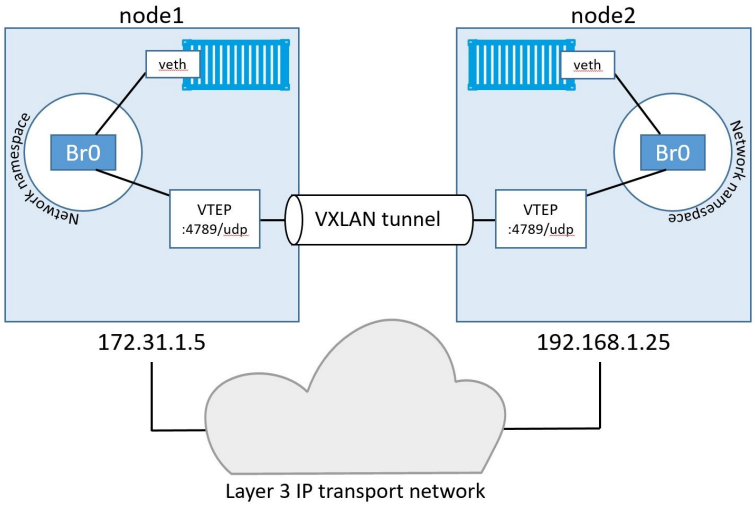



**Figure 10.7**

#### Communication example

Now that we've seen the main plumbing elements let's see how the two containers communicate.

For this example, we'll call the container on node1 "**C1**" and the container on node2 "**C2**". And let's assume **C1** wants to ping **C2** like we did in the practical example earlier in the chapter.

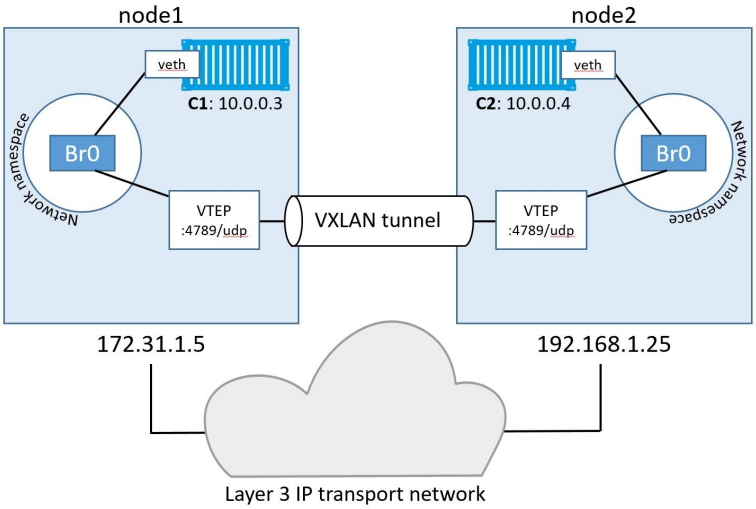



**Figure 10.8**

Container **C1** creates the ping requests and sets the destination IP address to be the `10.0.0.4` address of **C2**. It sends the traffic over its `veth` interface which is connected to the **Br0** virtual switch. The virtual switch doesn't know where to send the packet as it doesn't have an entry in its MAC address table (ARP table) that corresponds to the destination IP address. As a result, it floods the packet to all ports. The VTEP interface connected to **Br0** knows how to forward the frame so responds with its own MAC address. This is a *proxy ARP* reply and results in the **Br0** switch *learning* how to forward the

packet and it updates its ARP table mapping 10.0.0.4 to the MAC address of the VTEP.

Now that the **Br0** switch has *learned* how to forward traffic to **C2** all future packets for **C2** will be transmitted directly to the VTEP interface. The VTEP interface knows about **C2** because all newly started containers have their network details propagated to other nodes in the swarm using the network's built-in gossip protocol.

The switch then sends the packet to the VTEP interface which encapsulates the frames so they can be sent over the underlay transport infrastructure. At a fairly high level this encapsulation includes adding a VXLAN header to the Ethernet frame. The VXLAN header contains the VXLAN network ID (VNID) which is used to map frames from VLANs to VXLANs and vice versa. Each VLAN gets mapped to VNID so that on the receiving end the packet can be de-encapsulated and forwarded on to the correct VLAN. This obviously maintains network isolation. The encapsulation also wraps the frame in a IP/UDP packet with the IP address of the VTEP on node2 in the *destination IP field* and the UDP port 4789 socket information. This encapsulation allows the data to be sent across the underlying networks without the underlying networks having to know anything about VXLAN.

When the packet arrives at node2, the kernel sees that it's addressed to UDP port 4789. The kernel also knows that it has a VTEP interface bound to that socket. As a result, it sends the packet to the VTEP which reads the VNID, de-encapsulates the packet and sends it on to its own local **Br0** switch on the VLAN that corresponds the VNID. From there it is delivered to container C2.

That's the basics of how VXLAN technology is leveraged by native Docker overlay networks.

We're only scratching the surface here, but it should be enough for you to be able to start the ball rolling with any potential production Docker deployments. It should also give you the knowledge required to talk to your networking team about the networking aspects of your Docker infrastructure.

One final thing to mention about Docker overlay networks is that Docker also supports Layer 3 routing within the same overlay network. For example, you can create an overlay network with two subnets, and Docker will take care of routing between them. The command to create a network like this could be `docker network create --subnet=10.1.1.0/24 --subnet=11.1.1.0/24 -d overlay prod-net`. This would result in two virtual switches **Br0** and **Br1** being created inside the *network namespace* and routing happens by default.

## Docker overlay networking - The commands

- `docker network create` is the command that we use to create a new container network. The `-d` flag lets you specify the driver to use, and the most common driver is the `overlay` driver. However, you can also specify so-called *remote* drivers from 3rd parties. The control plane is encrypted by default. To encrypt the data plane just add the `-o encrypted` flag.
- `docker network ls` lists all of the container networks visible to a Docker host. Docker hosts running in *swarm mode* only see overlay networks if they are hosting containers running on that particular network. This helps reduce the amount network-related gossiping between nodes.
- `docker network inspect` shows you detailed information about a particular container network. This includes *scope*, *driver*, *IPv6*, *subnet configuration*, *VXLAN network ID*, and *encryption state*.
- `docker network rm` deletes a network

## Chapter Summary

In this chapter we saw the simplicity of creating new Docker overlay networks using the `docker network create` command. We then learned how they are created behind the scenes using VXLAN technology. This only scratches the surface of what you can do with native Docker overlay networking.

# 11: Security in Docker

Good security is all about layers, and Docker has lots of layers. It supports all the major Linux security technologies, as well as having a lot of its own - and most of them are simple and easy to configure.

In this chapter, we'll look at some of the technologies that make running containers on Docker very secure.

When we get to the deep dive part of the chapter we'll divide things up into two categories:

- Linux security technologies
- Docker platform security technologies

> **Note:** Large parts of this chapter are Linux specific. However, the **Docker platform security technologies** section is platform agnostic and applies equally to Linux and Windows.

## Security in Docker - The TLDR

Security is all about layers! The more security layers you have, the more secure you are. Well… Docker offers a lot of security layers. Figure 11.1 shows some of the security technologies that we'll cover in the chapter.

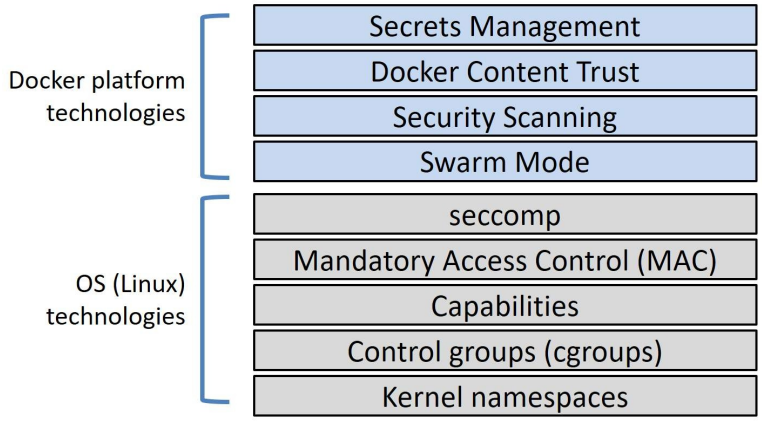



**Figure 11.1**

Docker on Linux leverages most of the common Linux security technologies. These include *namespaces*, *control groups (cgroups)*, *capabilities*, *mandatory access control (MAC) systems*, and *seccomp*. For each one, Docker implements sensible defaults for a seamless and *moderately secure* out-of-the-box experience. However, it also allows you to customize each one to your own specific requirements.

The Docker platform itself offers some excellent native security technologies. And one of the best things about these is that they're **amazingly simple to use!**

**Docker Swarm Mode** is secure by default. You get all of the following with zero configuration required; cryptographic node IDs, mutual authentication, automatic CA configuration, automatic certificate rotation, encrypted cluster store, encrypted networks, and more.

**Docker Content Trust (DCT)** lets you sign your images and verify the integrity and publisher of images you pull.

**Docker Security Scanning** analyses Docker images, detects known vulnerabilities, and provides you with a detailed report.

**Docker secrets** makes secrets first-class citizens in the Docker ecosystem. They get stored in the encrypted cluster store, encrypted in-flight when delivered to containers, and stored in in-memory filesystems when in use.

The important thing to know is that Docker works with the major Linux security technologies as well as providing its own extensive and growing set of security technologies. While the Linux security technologies can be a bit complicated to configure, the Docker platforms security technologies are very simple.

## Security in Docker - The deep dive

We all know that security is important. We also know that security can be complicated and boring!

When Docker decided to bake security into its platform, it decided to make it simple and easy. They knew that if security was hard to configure people wouldn't use it. As a result, most of the security technologies offered by the Docker platform are simple to use. They also ship with sensible defaults - this means that you get a *fairly secure* platform at zero effort. Of course, the defaults are not perfect, but they're usually enough to give you a safe start. If they don't suit your needs you can always customize them.

We'll organize the rest of this chapter as follows:

- Linux security technologies
  - Namespaces
  - Control Groups
  - Capabilities
  - Mandatory Access Control
  - seccomp
- Docker platform security technologies
  - Swarm Mode
  - Docker Security Scanning
  - Docker Content Trust
  - Docker secrets

## Linux security technologies

All *good* container platforms should use *namespaces* and *cgroups* to build containers. The *best* container platforms will also integrate with other Linux security technologies such as *capabilities*, *Mandatory Access Control systems* like SELinux and AppArmor, and *seccomp*. As expected, Docker integrates with them all!

In this section of the chapter we'll take a *brief* look at some of the major Linux security technologies used by Docker. We won't go into detail as I want the focus of the chapter to be on the Docker platform technologies.

#### Namespaces

Kernel namespaces are at the very heart of containers! They let us slice up an operating system (OS) so that it looks and feels like multiple isolated operating systems. This lets us do really cool things like run multiple web

servers on the same OS without having port conflicts. It also lets us run multiple apps on the same OS without them fighting over shared config files and shared libraries.

A couple of quick examples:

- You can run multiple web servers, each requiring port 443, on a single OS. To do this you just run each web server app inside of its own *network namespace*. This works because each *network namespace* gets its own IP address and full range of ports.
- You can run multiple applications, each requiring their own particular version of a shared library or configuration file. To do this you run each application inside of its own *mount namespace*. This works because each *mount namespace* can have its own isolated copy of any directory on the system (e.g. /etc, /var, /dev etc.)

Figure 11.2 shows a high-level example of two web server applications running on a single host and both using port 443. Each web server app is running inside of its own network namespace.

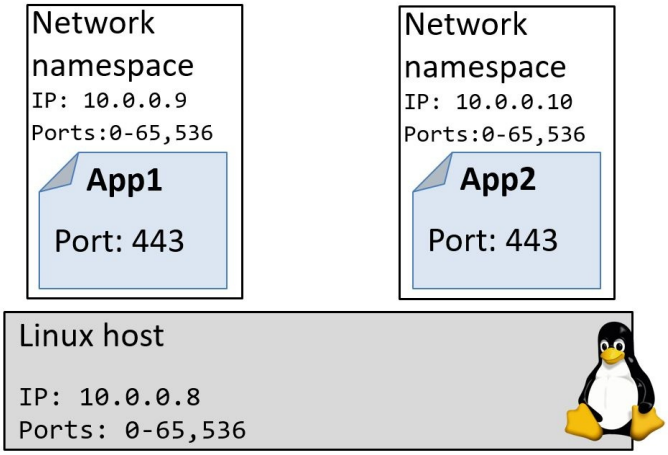



**Figure 11.2**

Docker on Linux currently utilizes the following kernel namespaces:

- Process ID (pid)
- Network (net)
- Filesystem/mount (mnt)
- Inter-process Communication (ipc)
- User (user)
- UTS (uts)

We'll briefly explain what each one does in a moment. But the most important thing to understand is that **Docker containers are an organized collection of**

**namespaces**. Let me repeat that… ***A Docker container is an organized collection of namespaces***.

For example, every container is made up of its own `pid`, `net`, `mnt`, `ipc`, `uts`, and potentially `user` namespace. The organized collection of these namespaces is what we call a container. Figure 11.3 shows a single Linux host running two containers.

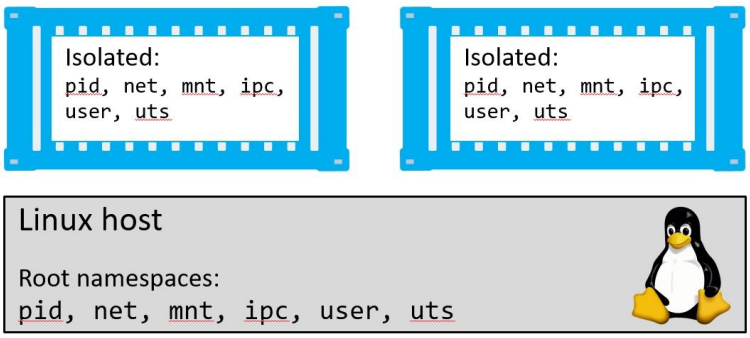



**Figure 11.3**

Let's briefly look at how Docker uses each namespace:

- Process ID namespace: Docker uses the `pid` namespace to provide isolated process trees for each container. Every container gets its own process tree meaning that every container can have its own PID 1. PID namespaces also mean that a container cannot see or access to the process tree of other containers or host it's running on.
- Network namespace: Docker uses the `net` namespace to provide each container its own isolated network stack. This stack includes; interfaces, IP addresses, port ranges, and routing tables. For example, every container gets its own `eth0` interface with its own unique IP and range of ports.
- Mount namespace: Every container gets its own unique isolated root `/` filesystem. This means that every container can have its own `/etc`, `/var`, `/dev` etc. Processes inside of a container cannot access the mount namespace of the Linux host or other containers - they can only see and access their own isolated mount namespace.
- Inter-process Communication namespace: Docker uses the `ipc` namespace for shared memory access within a container. It also isolates the container from shared memory outside of the container.
- User namespace: Docker lets you use `user` namespaces to map users inside of a container to a different user on the Linux host. A common example would be mapping the `root` user of a container to a non-root user on the Linux host. User namespaces are quite new to Docker and are currently optional.

- UTS namespace: Docker uses the `uts` namespace to provide each container with its own hostname.

Remember… a container is an organized collection of namespaces!!!

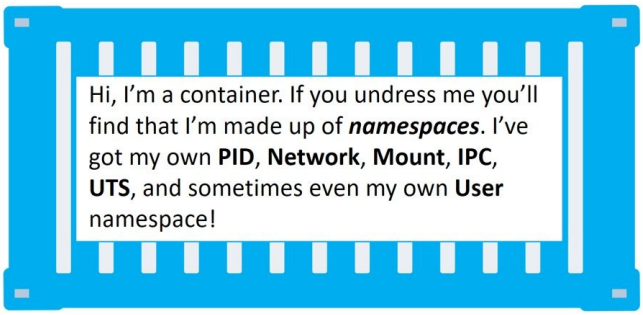



**Figure 11.4**

#### Control Groups

If namespaces are about isolation, *control groups (cgroups)* are about setting limits.

Think of containers as similar to rooms in a hotel. Yes, each room is isolated, but each room also shares a common set of resources - things like water supply, electricity supply, shared swimming pool, shared gym, shared breakfast bar etc. Cgroups let us set limits on containers so that (sticking with the hotel analogy) no single container can use all of the water or eat everything at the breakfast bar.

In the real world (not the silly hotel analogy) containers are isolated from each other but all share a common set of OS resources - things like CPU, RAM and disk I/O. Cgroups let us set limits on each of these so that a single container cannot use all of the CPU, RAM, or storage I/O of the Linux host.

#### Capabilities

It's a bad idea to run containers as root - root is all-powerful and therefore very dangerous. But it's a pain in the backside running containers as non-root - non-root is so powerless it's practically useless. What we need is a technology that lets us pick and choose which root powers our containers need in order to run. Enter *capabilities!*

Under the hood, the Linux root account is made up of a long list of capabilities. Some of these include:

- `CAP_CHOWN` lets you change file ownership
- `CAP_NET_BIND_SERVICE` lets you bind a socket to low numbered network ports
- `CAP_SETUID` lets you elevate the privilege level of a process

- `CAP_SYS_BOOT` lets you reboot the system.

The list goes on.

Docker works with *capabilities* so that you can run containers as root, but strip out the root capabilities that you don't need. For example, if the only root privilege your container needs is the ability to bind to low numbered network ports, you should start a container and drop all root capabilities, then add back the CAP_NET_BIND_SERVICE capability.

Docker also imposes restrictions so that containers cannot re-add the removed capabilities.

#### Mandatory Access Control systems

Docker works with major Linux MAC technologies such as AppArmor and SELinux.

Depending on your Linux distribution, Docker applies a default AppArmor profile to all new containers. According to the Docker documentation, this default profile is "moderately protective while providing wide application compatibility".

Docker also lets you start containers without a policy applied, as well as giving you the ability to customize policies to meet your specific requirements.

#### seccomp

Docker uses seccomp, in filter mode, to limit the syscalls a container can make to the host's kernel.

As per the Docker security philosophy, all new containers get a default seccomp profile configured with sensible defaults. This is intended to provide moderate security without impacting application compatibility.

As always, you can customize seccomp profiles and you can pass a flag to Docker so that containers can be started without a seccomp profile.

#### Final thoughts on the Linux security technologies

Docker supports most of the important Linux security technologies and ships with sensible defaults that add security but aren't too restrictive.

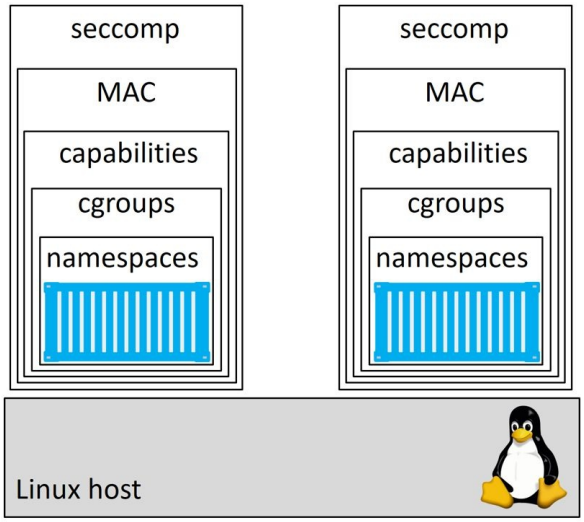



**Figure 11.5**

Some of these technologies can be complicated to customize as they can require deep knowledge of how they work and how the Linux kernel works. Hopefully they will get simpler to configure in the future, but for now, the default configurations that ship with Docker are a good place to start.

## Docker platform security technologies

In this section of the chapter we'll take a look at some of the major security technologies offered by the Docker platform.

#### Security in Swarm Mode

Swarm Mode is the future of Docker. It lets you cluster multiple Docker hosts and deploy your applications in a declarative way. Every Swarm is comprised of *managers* and *workers* that can be Linux or Windows. Managers make up the control plane of the cluster and are responsible for configuring the cluster and dispatching work to it. Workers are the nodes that run your application code as containers.

As expected, Swarm Mode includes many security features that are enabled out-of-the-box with sensible defaults. These include:

- Cryptographic node IDs
- Mutual authentication via TLS
- Secure join tokens
- CA configuration with automatic certificate rotation
- Encrypted cluster store (config DB)
- Encrypted networks

Let's walk through the process of building a secure Swarm and configuring some of the security aspects.

To follow along you will need at least three Docker hosts running Docker 1.13 or later. The examples cited here use three Docker hosts called “mgr1”, “mgr2”, and “wrk1”. Each one is running Docker 17.06.0-ce on Ubuntu 16.04. There is network connectivity between all three hosts and all three can ping each other by name. The setup is shown in Figure 11.6.

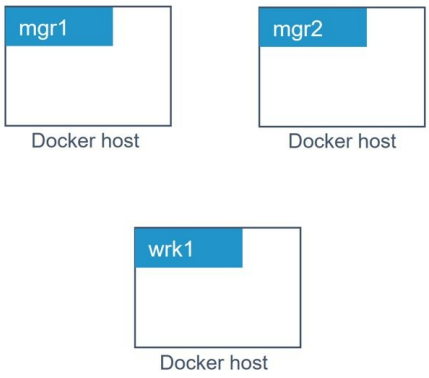



**Figure 11.6**

#### Configure a secure Swarm

Run the following command from the node you want to be the first manager in the new Swarm. In the example, we will run it form “mgr1”.

```
$ docker swarm init
Swarm initialized: current node (7xam...662z) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token \
     SWMTKN-1-1dmtwu...r17stb-ehp8g...hw738q 172.31.5.251:2377

To add a manager to this swarm, run 'docker swarm join-token manager'
and follow the instructions.
```

That’s it! That is literally all you need to do to configure a secure Swarm!

You now have a single-node secure Swarm. “mgr1” is configured as the first manager of the Swarm and also as the root CA. The Swarm has been given a cryptographic ID and “mgr1” has issued itself with a client certificate that identifies it as a manager in the Swarm. Certificate rotation has been configured with the default value of 90 days and a cluster config database has been configured and encrypted. A set of secure tokens have also been created so that new managers and new workers can be joined to the Swarm. And all of this with a **single command!**

Figure 11.7 shows how the lab looks now.

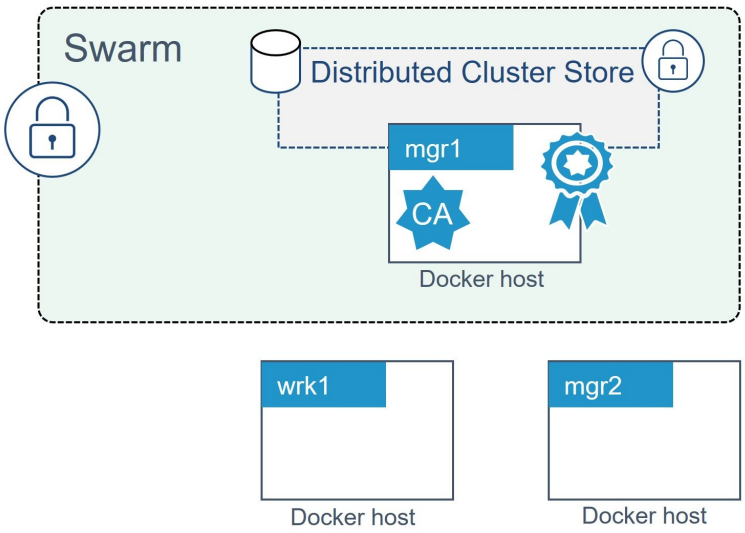



**Figure 11.7**

Now let's join "mgr2" as an additional manager.

Joining new managers to a Swarm is a two-step process. In the first step you'll extract the token required to join new managers to the Swarm. In the second step you'll run a `docker swarm join` command on "mgr2". As long as you include the manager join token as part of the `docker swarm join` command, "mgr2" will join the Swarm as a manager.

Run the following command from "mgr1" extract the manager join token.

```
$ docker swarm join-token manager
To add a manager to this swarm, run the following command:

    docker swarm join --token \
    SWMTKN-1-1dmtwu...r17stb-2axi5...8p7glz \
    172.31.5.251:2377
```

The output of the command above has been edited so that it doesn't wrap over multiple lines. The output gives you the exact command you need to run on nodes that you wish to join the Swarm as managers. The join token and IP address will be different in your lab.

Copy the command and run it on "mgr2" as shown below:

```
$ docker swarm join --token SWMTKN-1-1dmtwu...r17stb-2axi5...8p7glz \
> 172.31.5.251:2377

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

"mgr2" has now joined the Swarm as an additional manager.

> The format of the join command is `docker swarm join --token <manager-join-token> <ip-of-existing-manager>:<default-swarm-port>`.

You can verify the operation by running a `docker node ls` command on either of the two managers.

```
$ docker node ls
ID                  HOSTNAME   STATUS    AVAILABILITY    MANAGER STATUS
7xamk...ge662z      mgr1       Ready     Active          Leader
i0ue4...zcjm7f *    mgr2       Ready     Active          Reachable
```

The output above shows that “mgr1” and “mgr2” are both part of the Swarm and are both Swarm managers. The updated configuration is shown in Figure 11.8.

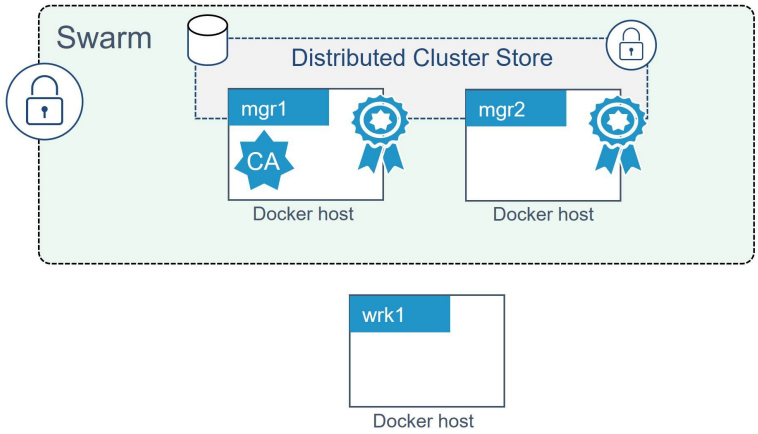



**Figure 11.8**

Adding a Swarm worker is a similar two-step process. Step 1 is to extract the join token for new workers, and step 2 is to run a `docker swarm join` command on the node you want to join as a worker.

Run the following command on either of the managers to expose the worker join token.

```
$ docker swarm join-token worker

To add a worker to this swarm, run the following command:

    docker swarm join --token \
    SWMTKN-1-1dmtw...17stb-ehp8g...w738q \
    172.31.5.251:2377
```

The output of the command above has been edited so that it doesn’t wrap over multiple lines. It gives you the exact command you need to run on nodes that you wish to join the Swarm as workers. The join token and IP address will be different in your lab.

Copy the command and run it on “wrk1” as shown below:

```
$ docker swarm join --token SWMTKN-1-1dmtw...17stb-ehp8g...w738q \
> 172.31.5.251:2377

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

Run another `docker node ls` command from either of the Swarm managers.

```
$ docker node ls
ID                  HOSTNAME      STATUS      AVAILABILITY   MANAGER STATUS
7xamk...ge662z *    mgr1          Ready       Active         Leader
ailrd...ofzv1u      wrk1          Ready       Active
i0ue4...zcjm7f      mgr2          Ready       Active         Reachable
```

You now have a Swarm with two managers and one worker. The managers are configured for high availability (HA) and the cluster store is replicated to them both. This updated configuration is shown in Figure 11.9.

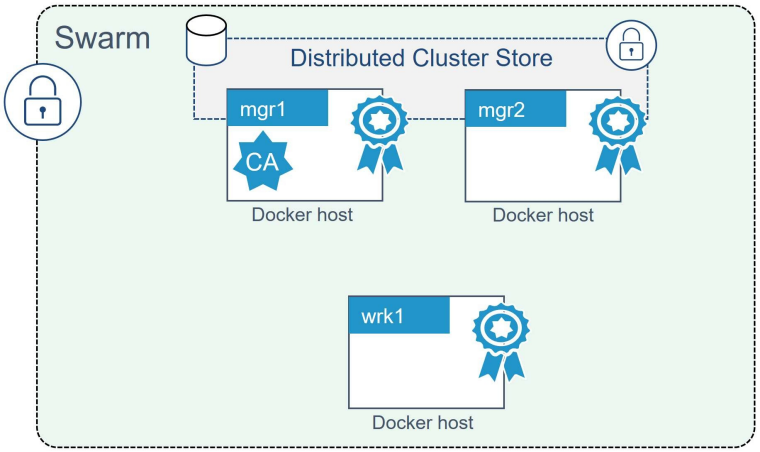



**Figure 11.9**

#### Looking behind the scenes at Swarm security

Now that we've built a secure Swarm let's take a minute to look behind the scenes at some of the security technologies involved.

#### Swarm join tokens

The only thing that is needed to join managers and workers to an existing Swarm is the relevant join token. For this reason, it is vital that you keep your join-tokens safe! No posting them on public GitHub pages!

Every Swarm maintains two distinct join tokens: - One for joining new managers - One for joining new workers

It's worth understanding the format of the Swarm join token. Every join token is comprised of 4 distinct fields separated by dashes (-):

PREFIX - VERSION - SWARM ID - TOKEN

The prefix is always "SWMTKN". The version field indicates the version of the Swarm. The Swarm ID field is a hash of the Swarm's certificate. The token portion is the part that determines if the token can be used to join the node as a manager or worker.

As you can see below, the manager and worker join tokens for a given Swarm are identical except for the final TOKEN field.

- MANAGER: SWMTKN-1-1dmtwusdc…r17stb-**2axi53zjbs45lqxykaw8p7glz**
- WORKER: SWMTKN-1-1dmtwusdc…r17stb-**ehp8gltji64jbl45zl6hw738q**

If you suspect that either of your join tokens has been compromised you can revoke them and issue new ones with a single command. The following example revokes the existing *manager* join token and issues a new one.

```
$ docker swarm join-token --rotate manager

Successfully rotated manager join token.

To add a manager to this swarm, run the following command:

    docker swarm join --token \
     SWMTKN-1-1dmtwu...r17stb-1i7txlh6k3hb921z3yjtcjrc7 \
     172.31.5.251:2377
```

Notice that the only difference between the old and new join tokens is the last field. The Swarm ID remains the same.

Join tokens are stored in the cluster config database which is encrypted by default.

#### TLS and mutual authentication

Every manager and worker that joins a Swarm is issued a client certificate. This certificate is used for mutual authentication. It identifies the node, which Swarm the node is a member of, and role the node performs in the Swarm (manager or worker).

On a Linux host you can inspect a node's client certificate with the following command.

```
$ sudo openssl x509 \
  -in /var/lib/docker/swarm/certificates/swarm-node.crt \
  -text

  Certificate:
      Data:
          Version: 3 (0x2)
          Serial Number:
              80:2c:a7:b1:28...a8:af:89:a1:2a:51:89
      Signature Algorithm: ecdsa-with-SHA256
          Issuer: CN=swarm-ca
          Validity
              Not Before: Jul 19 07:56:00 2017 GMT
              Not After : Oct 17 08:56:00 2017 GMT
          Subject: O=mfbkgjm2tlametbnfqt2zid8x, OU=swarm-manager,
          CN=7xamk8w3hz9q5kgr7xyge662z
          Subject Public Key Info:
<SNIP>
```

The `Subject` data in the output above uses the standard `O`, `OU`, and `CN` fields to specify the Swarm ID, the node's role, and the node ID.

- The organization `O` field stores the Swarm ID
- The organizational unit `OU` field stores the nodes role in the Swarm
- The canonical name `CN` field stores the nodes crypto ID.

This is shown in Figure 11.10.

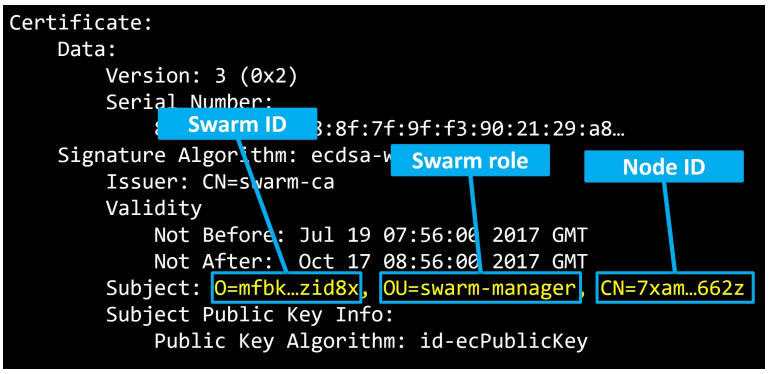



**Figure 11.10**

We can also see the certificate rotation period in the `Validity` section.

We can match these values to the corresponding values shown in the output of a `docker system info` command.

```
$ docker system info
<SNIP>
Swarm: active
 NodeID: 7xamk8w3hz9q5kgr7xyge662z
 Is Manager: true
 ClusterID: mfbkgjm2tlametbnfqt2zid8x
 ...
 <SNIP>
 ...
 CA Configuration:
  Expiry Duration: 3 months
  Force Rotate: 0
 Root Rotation In Progress: false
 <SNIP>
```

#### Configuring some CA settings

You can configure the certificate rotation period for the Swarm with the `docker swarm update` command. The example below changes the certificate rotation period to 30 days.

```
$ docker swarm update --cert-expiry 720h

Swarm updated.
```

Swarm allows nodes to renew certificates early (slightly before they expire) so that not all nodes in the Swarm try and update their certificates at the same

time.

You can configure an external CA when creating a Swarm by passing the `--external-ca` flag to the `docker swarm init` command.

The new `docker swarm ca` sub-command can be used to manage CA related configuration. Run the command with the `--help` flag to see a list of things it can do.

```
$ docker swarm ca --help

Usage:  docker swarm ca [OPTIONS]

Manage root CA

Options:
      --ca-cert pem-file          Path to the PEM-formatted root CA
      certificate to use for the new cluster
      --ca-key pem-file           Path to the PEM-formatted root CA
      key to use for the new cluster
      --cert-expiry duration      Validity period for node certificates
      (ns|us|ms|s|m|h) (default 2160h0m0s)
  -d, --detach                    Exit immediately instead of waiting for
      the root rotation to converge
      --external-ca external-ca   Specifications of one or more certificate
      signing endpoints
      --help                      Print usage
  -q, --quiet                     Suppress progress output
      --rotate                    Rotate the swarm CA - if no certificate
      or key are provided, new ones will be generated
```

#### The cluster store

The cluster store is the brains of a Swarm and is the place where cluster config and state are stored.

The store is currently based on an implementation of `etcd` and is automatically configured to replicate itself to all managers in the Swarm. It is also encrypted by default.

The cluster store is becoming a critical component of many Docker platform technologies. For example, Docker networking and Docker secrets both leverage the cluster store. This is one of the reasons that Swarm Mode is so important to the future of Docker - many parts of the Docker platform already leverage the cluster store and more will leverage it in the future. The moral of the story… if you're not running in Swarm Mode you'll be limited as to what other Docker features you can use.

The day-to-day maintenance of the cluster store is taken care of automatically by Docker. However, in production environments you should have strong backup and recovery solutions in place for it.

That's enough for now about Swarm Mode security.

#### Detecting vulnerabilities with Docker Security Scanning

The ability to quickly identify code vulnerabilities is vital. Docker Security Scanning makes detecting known vulnerabilities in Docker images really simple.

> **Note:** At the time of writing, Docker Security Scanning is available for private repositories on Docker Hub. It is also available as part of the Docker Enterprise Edition on premises solution. All official Docker images are scanned and scan reports are available in their repos.

Docker Security Scanning performs binary-level scans of Docker images and checks the software in them against databases of known vulnerabilities (CVE databases). After the scan is performed a detailed report is made available.

Open a web browser to https://hub.docker.com and search for the `Alpine` image. Figure 11.11 shows the `Tags` tab of the Alpine image repo.

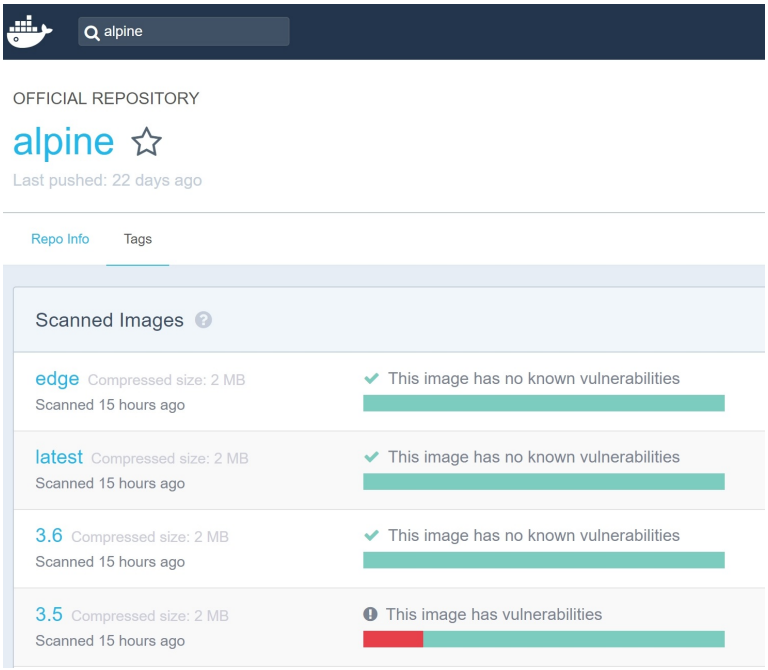



**Figure 11.11**

Because the Alpine image is an official image it gets scanned and scan reports are available. As you can see, the images tagged as `edge`, `latest`, and `3.6` are free from known vulnerabilities. However, the `alpine:3.5` image has known vulnerabilities (red).

If you drill into the `alpine:3.5` image you get a more detailed report as shown in Figure 11.12.

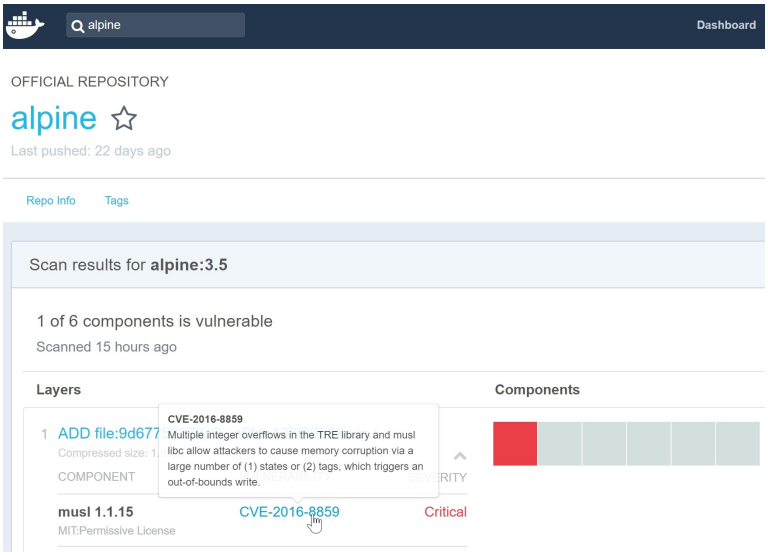



**Figure 11.12**

This is a simple and easy way to get detailed information about known vulnerabilities in your software.

Docker Trusted Registry (DTR), which is an on-premises Docker registry included as part of Docker Enterprise Edition, provides the same capabilities and gives you control over how and when image scans are performed. For example, DTR lets you decide if images should be automatically scanned as soon as they are pushed, or if scans should only be triggered manually. It also allows you to manually upload CVE database updates - this is ideal for situations where your DTR infrastructure is air-gapped from the internet and cannot automatically sync updates.

### Signing and verifying images with Docker Content Trust

Docker Content Trust (DCT) makes it simple and easy to verify the integrity and the publisher of images that you download. This is especially important when pulling images over untrusted networks such as the internet.

At a high level, DCT allows developers to sign their images when they are pushed to Docker Hub or Docker Trusted Registry. It will also automatically verify images when they are pulled. This high-level process is shown in Figure 11.13

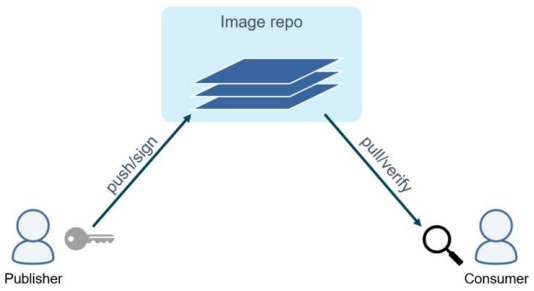



**Figure 11.13**

DCT can also provide important *context*. This includes things like; whether or not an image has been signed for use in a production environment, or whether an image has been superseded by a newer version and is therefore stale.

At the time of writing, the *context* offerings of DTC are in their infancy and quite complex to configure. As and when it matures and becomes more stable it will be included in an update to the book.

All you need to do to enable DCT on a Docker host is export and environment variable called `DOCKER_CONTENT_TRUST` with a value of `1`.

```
$ export DOCKER_CONTENT_TRUST=1
```

In the real world you may want to make this a more permanent feature of your system.

If you are using Docker Universal Control Plane (part of Docker Enterprise Edition) you need to set the `Only run signed images` checkbox as shown in Figure 11.14. This will force all nodes in the UCP cluster to only work with signed images.

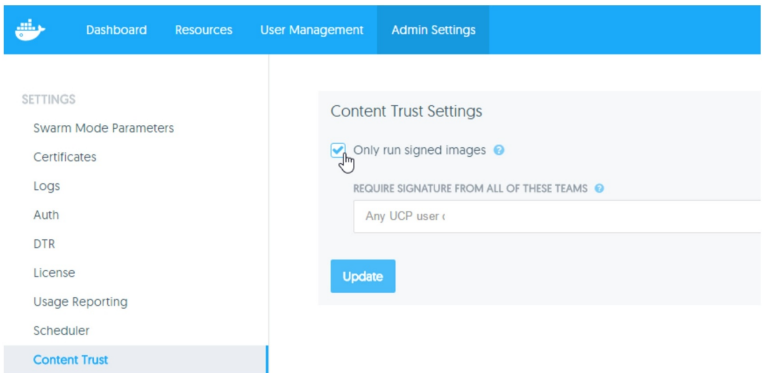



**Figure 11.14**

You can see from Figure 11.14 that Universal Control Plane takes DCT one step further by giving the option to list security principals that are required to sign an image before it can be used. For example, you might have a corporate policy that all images used in production need to be signed by the `secops` team.

Once DCT has been enabled you will no longer be able to pull and work with unsigned images. Figure 11.15 shows the errors you will get if you attempt to pull an unsigned image using the Docker CLI and the Universal Control Plane web UI (both examples are attempting to pull an image tagged as “unsigned”)

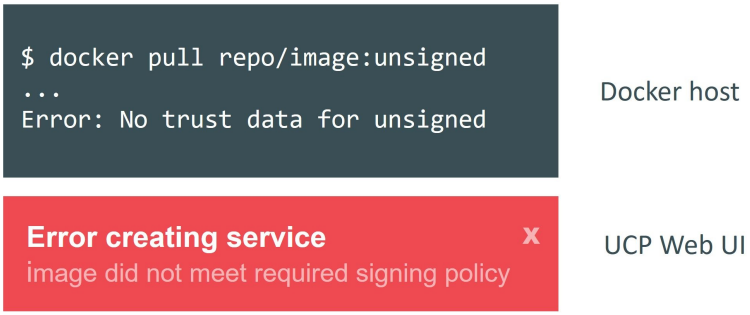



**Figure 11.15**

Figure 11.16 shows how DCT prevents a Docker client from pulling an image that has been tampered with. Figure 11.17 shows DCT preventing a client pulling an image that is stale.

```
$ docker pull repo/image:fakesignature

Warning: potential malicious behavior - trust data has
insufficient signatures for remote repository
docker.io/repo/image: valid signatures did not meet threshold
```

**Figure 11.16 - Pulling an image that has been tampered with**

```
$ docker pull repo/image:stale

Error: remote repository docker.io/repo/image out-of-date:
targets expired at Sun Mar 26 03:56:12 PDT 2017
```

**Figure 11.17 - Pulling a stale image**

Docker Content Trust is an important technology for helping you verify the images you are pulling from Docker registries. It's simple to configure in its basic form, but more advanced features such as *context* are currently more complex to configure.

## Docker secrets

Many applications need secrets. Things like passwords, TLS certificates, SSH keys and more.

Prior to Docker 1.13 there was no standard way of making secrets available to apps in a secure way. It was common for developers to insert secrets into apps via plain text environment variables (we've all done it). This was far from ideal.

Docker 1.13 introduced *Docker Secrets*, effectively making secrets first-class citizens in the Docker ecosystem. For example, there is a whole new `docker secret` sub-command dedicated to managing secrets. There's also a page for creating and managing secrets in the Docker Universal Control Plane UI. Behind the scenes secrets are encrypted at rest, encrypted in-flight, mounted in in-memory filesystems, and only available to services/containers that have

been explicitly granted access to them. It's quite a comprehensive end-to-end solution.

Figure 11.18 shows a high-level workflow:

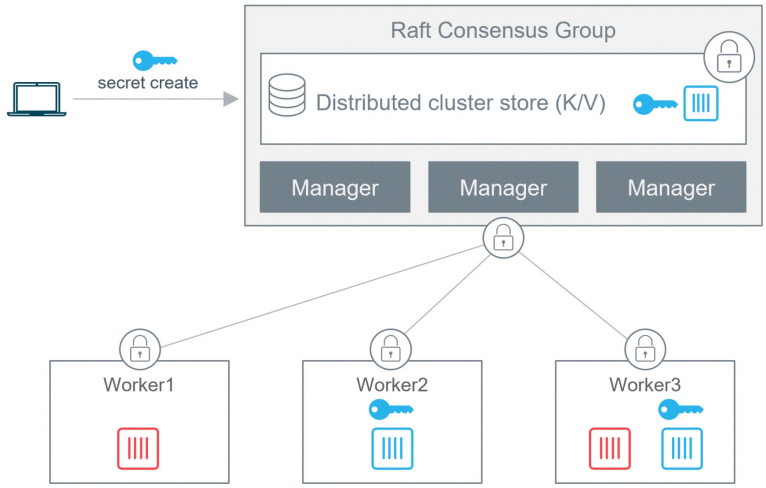



**Figure 11.18**

The following steps walk through the high-level workflow shown in Figure 11.18.

1. The secret is created and posted to the Swarm
2. It gets stored in the encrypted cluster store (all managers have access to the cluster store)
3. The blue service is created and the secret is attached to it
4. The secret is encrypted in-flight while it is delivered to the containers in the blue service
5. The secret is mounted into the containers of the blue service as an unencrypted file at `/run/secrets/`. This is an in-memory tmpfs filesystem (this step is different on Windows Docker hosts as they do not have the notion of an in-memory filesystem like tmpfs)
6. Once the container (service task) completes the in-memory filesystem is torn down.
7. The red containers/service cannot access the secret.

From the command line you can create and manage secrets with the `docker secret` sub-command, and you can attach them to services by specifying the `--secret` flag to the `docker service create` command.

## Chapter Summary

Docker can be configured to be extremely secure. It supports all of the major Linux security technologies including; kernel namespaces, cgroups, capabilities, MAC, and seccomp. For all of these it ships with sensible defaults, but you can customize them and even disable them.

Over and above the general Linux security technologies, the Docker platform also includes an extensive set of its own security technologies. Swarm Mode is built on TLS and is insanely simple to configure and customize. Security Scanning performs binary-level inspections of Docker images and provides detailed reports of known vulnerabilities. Docker Content Trust lets you sign and verify content, and secrets are now first-class citizens in Docker.

The net result is that your Docker environment can be configured to be as secure or insecure as you desire - it all depends on how you configure it.

# 12: 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. If you're not a member of Pluralsight then become one! Yes, it costs money, but it's definitely a service where you get value for your money! And if you're unsure… they always have a free trial period where you can get access to my courses for free for a limited period.

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

## Feedback

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

Now let me ask a favor…

It takes a *lot* of effort to write a book! My hope in writing this book, is that it inspires you and opens new opportunities. If you've enjoyed it, show it some love with a few stars and a review on Amazon!


To quote William Shakespeare "*They do not love, that do not show their love.*" So, if you love the book, show it with some stars!"

Feel free to hit me on Twitter as well, but stars and cars are what I dream about at night ;-)


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