

Docker Deep Dive



Feb 2018

Nigel Poulton

Docker Deep Dive

Zero to Docker in a single book!

Nigel Poulton

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

This version was published on 2018-02-11



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.

© 2016 - 2018 Nigel Poulton

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

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

Contents

0: About the book	1
What's this Docker Certified Associate stuff?	1
What about a print (paperback) version	2
Why should I read this book or care about Docker?	3
Isn't Docker just for developers?	3
Should I buy the book if I've already watched your video training courses?	3
How the book is organized	4
Versions of the book	5
Having problems getting the latest updates on your Kindle?	6

Part 1: The big picture stuff

1: Containers from 30,000 feet	8
The bad old days	8
Hello VMware!	8
VMwarts	9
Hello Containers!	9
Linux containers	10
Hello Docker!	10
Windows containers	11
Windows containers vs Linux containers	11
What about Mac containers?	11
What about Kubernetes	12
Chapter Summary	13

CONTENTS

2: Docker	14
Docker - The TLDR	14
Docker, Inc.	14
The Docker runtime and orchestration engine	16
The Docker open-source project (Moby)	17
The container ecosystem	18
The Open Container Initiative (OCI)	19
Chapter summary	21
3: Installing Docker	22
Docker for Windows (DfW)	22
Docker for Mac (DfM)	28
Installing Docker on Linux	32
Installing Docker on Windows Server 2016	35
Upgrading the Docker Engine	37
Docker and storage drivers	40
Chapter Summary	46
4: The big picture	47
The Ops Perspective	48
The Dev Perspective	56
Chapter Summary	61
 Part 2: The technical stuff	 62
5: The Docker Engine	63
Docker Engine - The TLDR	63
Docker Engine - The Deep Dive	64
Chapter summary	73
6: Images	74
Docker images - The TLDR	74
Docker images - The deep dive	75
Images - The commands	102
Chapter summary	103

CONTENTS

7: Containers	104
Docker containers - The TLDR	104
Docker containers - The deep dive	106
Containers - The commands	127
Chapter summary	128
8: Containerizing an app	130
Containerizing an app - The TLDR	130
Containerizing an app - The deep dive	131
Containerizing an app - The commands	153
Chapter summary	154
9: Deploying Apps with Docker Compose	155
Deploying apps with Compose - The TLDR	155
Deploying apps with Compose - The Deep Dive	156
Deploying apps with Compose - The commands	176
Chapter Summary	177
10: Docker Swarm	178
Docker Swarm - The TLDR	178
Docker Swarm - The Deep Dive	179
Docker Swarm - The Commands	204
Chapter summary	205
11: Docker Networking	206
Docker Networking - The TLDR	206
Docker Networking - The Deep Dive	207
Docker Networking - The Commands	234
Chapter Summary	235
12: Docker overlay networking	236
Docker overlay networking - The TLDR	236
Docker overlay networking - The deep dive	237
Docker overlay networking - The commands	252
Chapter Summary	253

CONTENTS

13: Volumes and persistent data	254
Volumes and persistent data - The TLDR	254
Volumes and persistent data - The Deep Dive	255
Volumes and persistent data - The Commands	265
Chapter Summary	266
14: Deploying apps with Docker Stacks	267
Deploying apps with Docker Stacks - The TLDR	267
Deploying apps with Docker Stacks - The Deep Dive	268
Deploying apps with Docker Stacks - The Commands	292
Chapter Summary	293
15: Security in Docker	294
Security in Docker - The TLDR	294
Security in Docker - The deep dive	296
Chapter Summary	321
16: Tools for the enterprise	322
Tools for the enterprise - The TLDR	322
Tools for the enterprise - The Deep Dive	323
Chapter Summary	353
17: Enterprise-grade features	355
Enterprise-grade features - The TLDR	355
Enterprise-grade features - The Deep Dive	355
Chapter Summary	384
Appendix A: Securing client and daemon communication	386
Lab setup	388
Create a CA (self-signed certs)	389
Configure Docker for TLS	394
Docker TLS Recap	399
Appendix B: The DCA Exam	401
Other resources to help with the exam	401
Mapping exam objectives to chapters	403

CONTENTS

Domain 1: Orchestration (25% of exam)	403
Domain 2: Image Creation, Management, and Registry (20% of exam) . .	404
Domain 3: Installation and Configuration (15% of exam)	405
Domain 4: Networking (15% of exam)	405
Domain 5: Security (15% of exam)	406
Domain 6: Storage and Volumes (10% of exam)	406
Appendix C: What next	408
Practice	408
Video training	408
Certifications	408
Community events	409
Feedback	409

0: About the book

This is a book about Docker. No prior knowledge required! The motto of the book is **Zero to Docker in a single book!**

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 if you get things wrong, this book is **not** for you.

What's this Docker Certified Associate stuff?

Docker released its first professional certification in the fall of 2017. It's called the **Docker Certified Associate (DCA)** and it's for people wanting to prove their mastery of Docker.



The exam objectives match a lot of real-world scenarios, so I decided to update the book so that it covered all objectives. In doing this, I worked extremely hard to keep the book interesting and applicable in the real world.

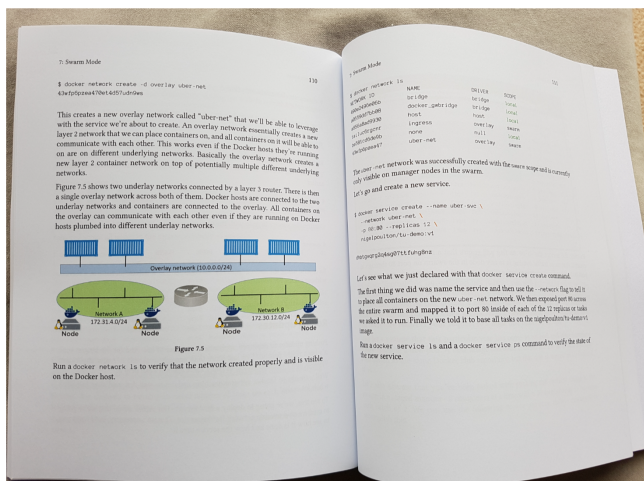
This is not an exam-cram book. Yes, **it covers all exam topics**, but this is a real-world book that is enjoyable to read.

At the time of publishing, **this is the only resource available that covers the entire set of DCA exam objectives!**

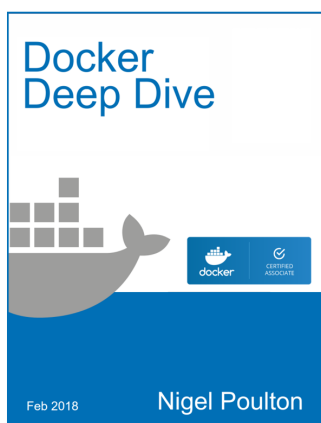
Good luck with your exam!

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. None of this black-and-white nonsense.



On the topic of Amazon... I'd love it if you could write a quick review on Amazon! You can even do this if you bought the book on Leanpub. Cheers!



★★★★★ 40+ customer reviews

Docker Deep Dive

by Nigel Poulton (Author)

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 *containerized* apps in our business-critical 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!

Containerized apps 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 production-grade 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, the book will skill you up!

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

Yes. The book is usually more up to date and covers additional material.

If you like my [video courses](#)¹ you'll probably like the book. If you don't like my video courses you probably won't like the book.

If you haven't watched my video courses, you should! They're fast-paced and fun and get *rave reviews*!

¹<https://app.pluralsight.com/library/search?q=nigel+poulton>

How the book is organized

I've divided the book into two sections:

- The big picture stuff
- The technical stuff

The **big picture stuff** section covers things like:

- Who is Docker, Inc.
- What is the Docker (Moby) project.
- What is the OCI.
- Why do we even have containers...

It's the kind of stuff that you need to know if you want a good rounded knowledge of Docker and containers.

The **technical stuff** section 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*. It even covers the stuff that enterprises love, like TLS, RBAC, AD integration, and backups. 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.

Most of the chapters in the *technical stuff* section are divided into three parts:

- The TLDR
- The Deep Dive
- The Commands

The *TLDR* gives you two or three paragraphs that you can use to explain the topic at the coffee machine. They're also a great way to remind yourself what something is about.

The Deep Dive is where we explain how everything works and go through the examples.

The Commands lists 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! In other words... *the older this book is, the less valuable it is*. So I keep this book up-to-date!

Welcome to the new normal!

We no-longer live in a world where a 1-year old book is valuable. That makes my life as an author really hard. But it's true!

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

If you buy the paperback copy from **Amazon.com**, you get the Kindle version for dirt-cheap as part of the Kindle MatchBook scheme! Kindle MatchBook is a new service that is only available on Amazon.com and is a bit buggy. If you cannot see how to get your Kindle version through MatchBook you need to contact Kindle support — I cannot help you with this :-)

The Kindle and Leanpub versions get all updates at no extra cost!

That's the best I can currently do!

Below is a list of versions:

- **Version 5.** This is the version of the book published on 6th February 2018. It includes ~200 new pages and covers all Docker Certified Associate exam topics. This version of the book got a new cover.
- **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.
- **Version 1.** Initial version.

Having problems getting the latest updates on your Kindle?

It's come to my attention that Kindle does not always download the latest version of the book. To fix this:

Go to <http://amzn.to/2l53jdg>

Under Quick Solutions (on the left) select Digital Purchases. Select Content and Devices for the Docker Deep Dive order. Your book should show up in the list with a button that says "Update Available". Click that button. Delete your old version in Kindle and download the new one.

If this doesn't work, contact Kindle support and they will resolve the issue for you. https://kdp.amazon.com/en_US/self-publishing/contact-us/

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 break. Sometimes they even go bust. 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 big. 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. Cue wild celebrations!

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 far from 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 the shortcomings of the VM model.

In the container model, the container is roughly analogous to the VM. The major difference is that every container does not require its own 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. Net result: 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 LLC 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**, **union filesystems**, 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. Thank you!

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 book 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. and the community.

The core Windows kernel 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 many of the lab exercises cited throughout the book.

Windows containers vs Linux containers

It's vital to understand that a running container shares the kernel of the host machine it is running on. This means that a containerized app 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, 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.

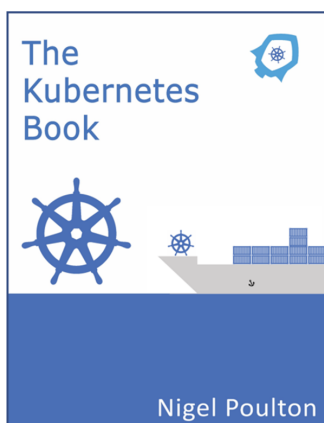
However, you can run Linux containers on your Mac using *Docker for Mac*. This works by seamlessly running your containers inside of a lightweight Linux VM on your Mac. It's extremely popular with developers, who can easily develop and test their Linux containers on their Mac.

What about Kubernetes

Kubernetes is an open-source project out of Google that has quickly emerged as the leading orchestrator of containerized apps. That's just a fancy way of saying *Kubernetes is an important piece of software that helps us deploy our containerized apps and keep them running*.

At the time of writing, Kubernetes uses Docker as its default container runtime — the piece of Kubernetes that starts and stops containers, as well as pulls images etc. However, Kubernetes has a pluggable container runtime interface called the CRI. This makes it easy to swap-out Docker for a different container runtime. In the future, Docker might be replaced by containerd as the default container runtime in Kubernetes. More on containerd later in the book.

The important thing to know about Kubernetes, at this stage, is that it's a higher-level platform than Docker, and it currently uses Docker for its low-level container-related operations.



★★★★★ ▾ 35 customer reviews

The Kubernetes Book

by Nigel Poulton ▾ (Author)

Check out my Kubernetes book and my **Getting Started with Kubernetes** [video training course](#)² for more info on Kubernetes.

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<https://app.pluralsight.com/library/courses/getting-started-kubernetes/>

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 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.

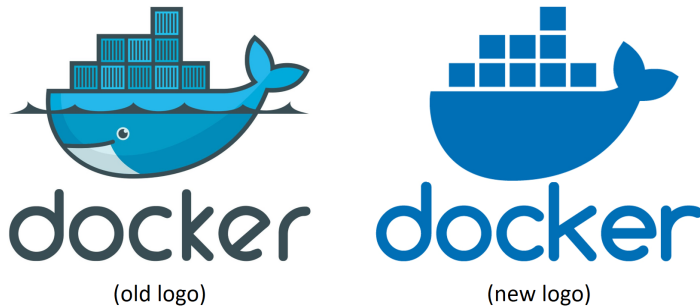


Figure 2.1 Docker, Inc. logo.

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 eventually 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 by some, to be in the region of \$1BN. At the time of writing, it has raised over \$240M via several 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 300-400 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 work__er__** — somebody who loads and unloads cargo from 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.

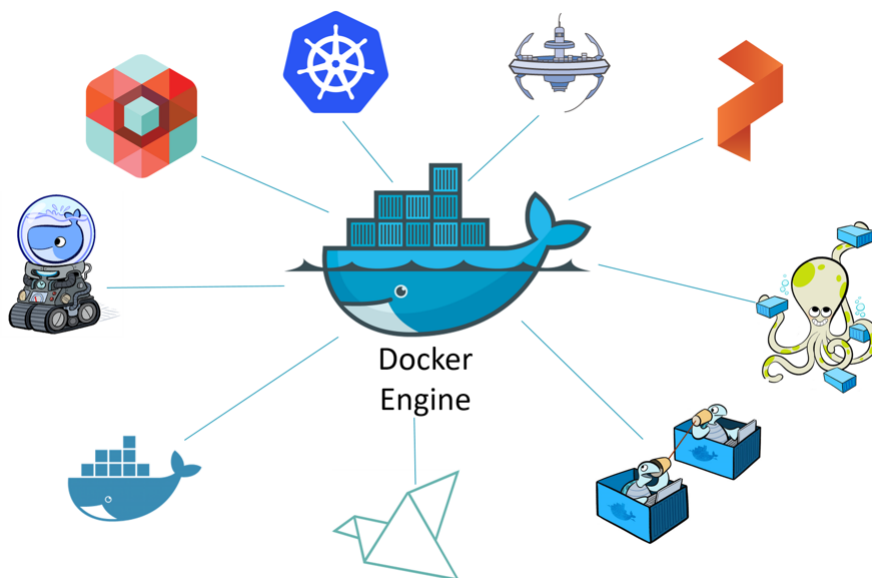


Figure 2.2

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 2018 will be 18.06.0-ce.

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

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.



The goal of the Moby project is to be the *upstream* for Docker, and 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 more parts of the Engine 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](#)³.

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 illusion, Docker is being used!

The container ecosystem

One of the core philosophies at Docker, Inc. is often referred to as *Batteries included but removable*.

³<https://github.com/docker/docker/blob/master/LICENSE>

This is a way of saying you can swap out a lot of the native Docker *stuff* and replace it with *stuff* from 3rd-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 has caused tension and raised levels competition within the ecosystem.

To cut a long story short, the native Docker batteries are still removable, there's just less and less of a **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](https://www.opencontainers.org)⁴.



The OCI is a 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).

⁴<https://www.opencontainers.org>

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 container 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 some parties would get frustrated. This is normal and healthy.

The TLDR of this *history according to Nigel* is that a company called [CoreOS](#)⁵ didn't like the way Docker did certain things. So they did something about it! They created a new open standard called [appc](#)⁶ 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](#)⁷
- The [runtime-spec](#)⁸

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!

⁵<https://coreos.com>

⁶<https://github.com/appc/spec/>

⁷<https://github.com/opencontainers/image-spec>

⁸<https://github.com/opencontainers/runtime-spec>

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, opinionated, and sometimes downright off the planet! 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 modern revolution. But a huge ecosystem of partners and competitors now exists.

The *Docker project* is open-source and the *upstream* 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, and more... On top of those, we've got 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
- Upgrading Docker
- Storage driver considerations

We'll also look at upgrading the Docker Engine and selecting an appropriate storage driver.

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 easy to download and has a slick installer. It 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. So it's not intended for production.

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.

Enough waffle. Let's see how to install *Docker for Windows*.

First up, pre-requisites. *Docker for Windows* requires:

- Windows 10 Pro | Enterprise | Education (1607 Anniversary Update, Build 14393 or newer)
- Must be 64-bit Windows 10
- 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 (a small link on the right).
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.

3: Installing Docker

24

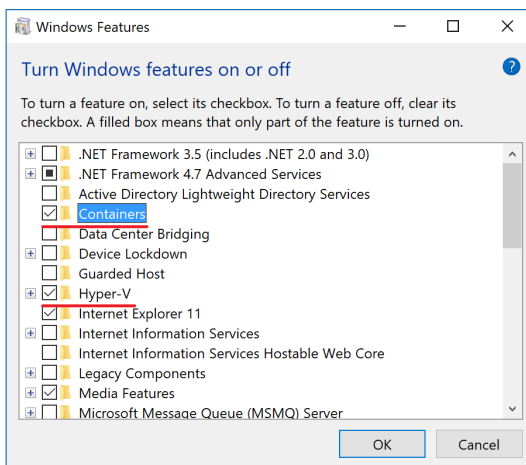


Figure 3.1

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 <https://www.docker.com/get-docker> and click the GET DOCKER COMMUNITY EDITION link.
2. Click the Download from Docker Store link beneath the DOCKER CE FOR WINDOWS section. This will take you to the Docker Store and you may need to login with your Docker ID.
3. Click one of the Get Docker download links.

Docker for Windows has a *stable* and *edge* channel. The edge channel contains newer features but may not be as stable.

An installer package called `Docker for Windows Installer.exe` will be downloaded to your default downloads directory.

4. Locate and launch the installer package downloaded in the previous step.

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*.

Open a command prompt or PowerShell terminal and try the following commands:

```
^^^
Client:
  Version:           18.01.0-ce
  API version:       1.35
  Go version:        go1.9.2
  Git commit:        03596f5
  Built: Wed Jan 10 20:05:55 2018
  OS/Arch:           windows/amd64
  Experimental:      false
  Orchestrator:      swarm

Server:
  Engine:
    Version:        18.01.0-ce
    API version:    1.35 (minimum version 1.12)
    Go version:     go1.9.2
    Git commit:     03596f5
    Built:          Wed Jan 10 20:13:12 2018
    OS/Arch:        linux/amd64
    Experimental:   false
  ^^^
```

Notice that the output is showing OS/Arch: `linux/amd64` for the **Server** component. This is because the default installation currently installs the Docker daemon inside of a lightweight Linux Hyper-V VM. In this 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 Switch to Windows containers.... You can achieve the same thing from the command line with the following command (located in the `\Program Files\Docker\Docker` directory):

3: Installing Docker

26

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

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

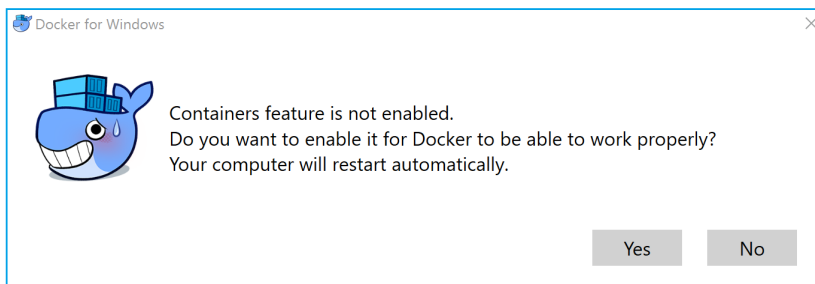


Figure 3.2

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:
  <Snip>

Server:
  Engine:
    Version:      18.01.0-ce
    API version:  1.35 (minimum version 1.24)
    Go version:   go1.9.2
    Git commit:   03596f5
    Built:        Wed Jan 10 20:20:36 2018
    OS/Arch:      windows/amd64
    Experimental: true
```

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

Also note that the system is now running the *experimental* version of Docker (Experimental: true). As previously mentioned, *Docker for Windows* has a stable and an edge channel. At the time of writing, Windows Containers is an experimental feature of the edge channel.

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

```
PS C:\Program Files\Docker\Docker> .\dockercli -Version
```

```
Docker for Windows
Version: 18.01.0-ce-win48 (15285)
Channel: edge
Sha1: ee2282129dec07b8c67890bd26865c8eccdea88e
OS Name: Windows 10 Pro
Windows Edition: Professional
Windows Build Number: 16299
```

The following listing shows that regular Docker commands work as normal.

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

> docker container ls
CONTAINER ID    IMAGE        COMMAND      CREATED      STATUS      PORTS      NAMES

> docker system info
Containers: 1
  Running: 0
  Paused: 0
  Stopped: 1
Images: 6
Server Version: 17.12.0-ce
Storage Driver: windowsfilter
<Snip>
```

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:

3: Installing Docker

28

```
C:\> docker --version
Docker version 18.01.0-ce, build 03596f5
```

```
C:\> docker-compose --version
docker-compose version 1.18.0, build 8dd22a96
```

```
C:\> docker-machine --version
docker-machine.exe version 0.13.0, build 9ba6da9
```

```
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. Installing DfM is ridiculously easy.

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 — 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, the Docker daemon is running inside a lightweight Linux VM. It then seamlessly exposes the daemon and API to your Mac environment. This means you can open a terminal on your Mac and use the regular Docker commands.

Although this works seamlessly on your Mac, don't forget that it's Docker on Linux under the hood — so it's only going to work with Linux-based Docker containers. This is good though, as it's where most of the container action is.

Figure 3.3 shows a high-level representation of the *Docker for Mac* architecture.

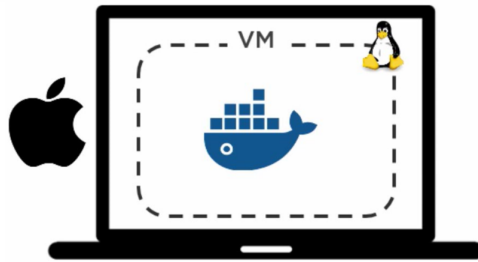


Figure 3.3

Note: For the curious reader, *Docker for Mac* leverages [HyperKit](https://github.com/docker/hyperkit)⁹ to implement an extremely lightweight hypervisor. HyperKit is based on the [xhyve](https://github.com/mist64/xhyve) hypervisor¹⁰. *Docker for Mac* also leverages features from [DataKit](https://github.com/docker/datakit)¹¹ and runs a highly tuned Linux distro called *Moby* that is based on [Alpine Linux](https://alpinelinux.org/)¹².

Let's get *Docker for Mac* installed.

1. Point your browser to <https://www.docker.com/get-docker> and click GET DOCKER COMMUNITY EDITION.
2. Click the Download from Docker Store option below DOCKER CE FOR MAC. This will take you to the Docker Store and you will need to provide your Docker ID and password.
3. Click one of the Get Docker CE download links.

Docker for Mac has a stable and edge channel. Edge has newer features, at the expense of stability.

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

⁹<https://github.com/docker/hyperkit>

¹⁰<https://github.com/mist64/xhyve>

¹¹<https://github.com/docker/datakit>

¹²<https://alpinelinux.org/> and <https://github.com/alpinelinux>

4. Launch the Docker .dmg file that you downloaded in the previous step. You will be asked to drag and drop the Moby Dock whale image into the **Applications** folder.
5. Open your **Applications** folder (it may open automatically) and double-click the Docker application icon to Start it. You may be asked to confirm the action because the application was downloaded from the internet.
6. Enter your password so that the installer can create the components that require elevated privileges.
7. The Docker daemon will now start.

An animated whale icon will appear in the status bar at the top of your screen while Docker starts. Once Docker has successfully started, the whale will stop being animated. You can click the whale icon to manage DfM.

Now that DfM is installed, you can open a terminal window and run some regular Docker commands. Try the following.

```
$ 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 that the OS/Arch: for the **Server** component is showing as linux/amd64. This is because 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* channel which comes with experimental features turned on.

Run some more Docker commands.

```
$ docker --version
```

```
Docker version 17.05.0-ce, build 89658be
```

```
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
------------	-----	----------	---------	------

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Docker for Mac installs the Docker Engine (client and daemon), Docker Compose, Docker machine, and the Notary command line. The following three commands show you how to verify that all of these components installed successfully, as well as 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 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'll be demonstrating. 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 this example, 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...

3. It is best practice to 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. The following command shows you 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 the installation.

3: Installing Docker

34

```
$ docker --version
Docker version 18.01.0-ce, build 03596f5

$ docker system info
Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
Images: 0
Server Version: 18.01.0-ce
Storage Driver: overlay2
  Backing Filesystem: extfs
<Snip>
```

If the process described above doesn't work for your Linux distro, you can go to the [Docker Docs](#)¹³ 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 package managers that 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 installation grunt-work for you — including configuring Docker to automatically start when the system boots.

Warning: If you install Docker from a source other than the official Docker repositories, you may end up with a forked version of Docker. In the past, some vendors and distros chose to fork the Docker project and develop their own slightly customized versions. You need to watch out for things like this, 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.

¹³<https://docs.docker.com/engine/installation/>

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 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 `Programs and Features` console.
2. Click `Turn Windows features on or off`. This will open the `Server Manager` app.
3. Make sure the `Dashboard` is selected and choose `Add Roles and Features`.
4. Click through the wizard until you get to the `Features` page.
5. Make sure that the `Containers` feature is checked, then complete the wizard. Your system 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 package management provider.

```
> Install-Module DockerProvider -Force
```

If prompted, accept the request to install the NuGet provider.

3. Install Docker.

```
> Install-Package Docker -ProviderName DockerProvider -Force
```

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

Name	Version	Source	Summary
-----	-----	-----	-----
Docker	17.06.2-ee-6	Docker	Docker for Windows Server 2016

Docker is now installed and configured to automatically start when the system boots.

4. You may want to restart your system to make sure that none of changes have introduced issues that cause your system not to boot. You can also check that Docker automatically starts after the reboot.

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
Docker version 17.06.2-ee-6, build e75fdb8
```

```
> docker system info
Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
Images: 0
Server Version: 17.06.2-ee-6
Storage Driver: windowsfilter
<Snip>
```

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

Upgrading the Docker Engine

Upgrading the Docker Engine is an important task in any Docker environment — especially production. This section of the chapter will give you the high-level process of upgrading the Docker engine, as well as some general tips and a couple of upgrade examples.

The high-level process of upgrading the Docker Engine is this:

Take care of any pre-requisites. These can include; making sure your containers have an appropriate restart policy, or draining nodes if you're using *Services* in Swarm mode. Once you've completed any potential pre-requisites you can follow the procedure below.

1. Stop the Docker daemon
2. Remove the old version
3. Install the new version
4. configure the new version to automatically start when the system boots
5. Ensure containers have restarted

That's the high-level process. Let's look at some examples.

Each version of Linux has its own slightly different commands for upgrading Docker. We'll show you Ubuntu 16.04. We'll also show you Windows Server 2016.

Upgrading Docker CE on Ubuntu 16.04

We're assuming you've completed all pre-requisites and your Docker host is ready for the upgrade. We're also assuming you're running commands as root. Running commands as root is obviously **not recommended**, but it does keep examples in the book simpler. If you're not running as root, well done! However, you will have to prepend the following commands with `sudo`.

1. Update your apt package list.

```
$ apt-get update
```

2. Uninstall existing versions of Docker.

```
$ apt-get remove docker docker-engine docker-ce docker.io -y
```

The Docker engine has had several different package names in the past. This command makes sure all older versions get removed.

3. Install the new version.

There are different versions of Docker and different ways to install each one. For example, Docker CE or Docker EE, both of which can be installed in more than one way. For example, Docker CE can be installed from apt or deb packages, or using a script on [docker.com](https://get.docker.com)

The following command will use a shell script at get.docker.com to install and configure the latest version of Docker CE.

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

4. Configure Docker to automatically start each time the system boots.

```
$ systemctl enable docker
```

```
Synchronizing state of docker.service...
```

```
Executing /lib/systemd/systemd-sysv-install enable docker
```

```
$ systemctl is-enabled docker
enabled
```

At this point you might want to restart the node. This will make sure that no issues have been introduced that prevent your system from booting in the future.

5. Make sure any containers and services have restarted.

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	\
NAMES					
97e599aca9f5	alpine	"sleep 1d"	14 minutes ago	Up 1 minute	


```
$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE
ibyo1t1t1ehjy	prod-equus1	replicated	1/1	alpine:latest

Remember, other methods of upgrading and installing Docker exist. We've just shown you one way, on Ubuntu Linux 16.04.

Upgrading Docker EE on Windows Server 2016

This section will walk you through the process of upgrading Docker on Windows from 1.12.2, to the latest version of Docker EE.

The process assumes you have completed any pre-flight tasks, such as configuring containers with appropriate restart policies and draining Swarm nodes if you're using Swarm services.

All commands should be ran from a PowerShell terminal.

1. Check the current version of Docker.

```
> docker version
Client:
Version:      1.12.2-cs2-ws-beta
<Snip>
Server:
Version:      1.12.2-cs2-ws-beta
```

2. Uninstall any potentially older modules provided by Microsoft, and install the module from Docker.

```
> Uninstall-Module DockerMsftProvider -Force
```

```
> Install-Module DockerProvider -Force
```

3. Update the docker package.

This command will force the update (no uninstall is required) and configure Docker to automatically start each time the system boots.

```
> Install-Package -Name docker -ProviderName DockerProvider -Update -Force
```

Name	Version	Source	Summary
----	-----	-----	-----
Docker	17.06.2-ee-6	Docker	Docker for Windows Server 2016

You might want to reboot your server at this point to make sure the changes have not introduced any issues that prevent it from restarting in the future.

4. Check that containers and services have restarted.

That's it. That's how to upgrade to the latest version of Docker EE on Windows Server 2016.

Docker and storage drivers

Every Docker container gets its own area of local storage where image layers are stacked and the container filesystem is mounted. By default, this is where all container read/write operations occur, making it integral to the performance and stability of every container.

Historically, this local storage area has been managed by the *storage driver*, which we sometimes call the *graph driver* or *graphdriver*. Although the high-level concepts of stacking image layers and using copy-on-write technologies are constant, Docker on Linux supports several different storage drivers, each of which implements layering and copy-on-write in its own way. While these *implementation differences* do not affect the way we *interact* with Docker, they can have a significant impact on *performance* and *stability*.

Some of the *storage drivers* available for Docker on Linux include:

- aufs (the original and oldest)
- overlay2 (probably the best choice for the future)
- devicemapper
- btrfs
- zfs

Docker on Windows only supports a single storage driver, the `windowsfilter` driver.

Selecting a storage driver is a *per node* decision. This means a single Docker host can only run a single storage driver — you cannot select the storage driver per-container. On Linux, you set the storage driver in `/etc/docker/daemon.json` and you need to restart Docker for any changes to take effect. The following snippet shows the storage driver set to `overlay2`.

```
{  
  "storage-driver": "overlay2"  
}
```

Note: If the configuration line is not the last line in the configuration file, you will need to add a comma to the end.

If you change the storage driver on an already-running Docker host, existing images and containers will not be available after Docker is restarted. This is because each storage driver has its own subdirectory on the host where it stores image layers (usually below `/var/lib/docker/<storage-driver>/...`). Changing the storage driver obviously changes where Docker looks for images and containers. Reverting the storage driver to the previous configuration will make the older images and containers available again.

If you need to change your storage driver, and you need your images and containers to be available after the change, you need to save them with `docker save`, push the saved images to a repo, change the storage driver, restart Docker, pull the images locally, and restart your containers.

You can check the current storage driver with the `docker system info` command:

```
$ docker system info
<Snip>
Storage Driver: overlay2
  Backing Filesystem: xfs
  Supports d_type: true
  Native Overlay Diff: true
<Snip>
```

Choosing which storage driver, and configuring it properly, is important in any Docker environment — especially production. The following list can be used as a **guide** to help you choose which storage driver to use. However, you should always consult the latest support documentation from Docker, as well as your Linux provider.

- **Red Hat Enterprise Linux** with a 4.x kernel or higher + Docker 17.06 and higher: `overlay2`
- **Red Hat Enterprise Linux** with an older kernel and older versions of Docker: `devicemapper`
- **Ubuntu Linux** with a 4.x kernel or higher: `overlay2`
- **Ubuntu Linux** with an earlier kernel: `aufs`
- **SUSE Linux Enterprise Server**: `btrfs`

Again, this list should only be used as a guide. Always check the latest support and compatibility matrixes in the Docker documentation, and with your Linux provider. This is especially important if you are using Docker Enterprise Edition (EE) with a support contract.

Devicemapper configuration

Most of the Linux storage drivers require little or no configuration. However, `devicemapper` needs configuring in order to perform well.

By default, `devicemapper` uses *loopback mounted sparse files* to underpin the storage it provides to Docker. This is fine for a smooth out-of-the box experience that *just*

works. But it's terrible for production. In fact, it's so bad that it's **not supported on production systems!**

To get the best performance out of `devicemapper`, as well as production support, you must configure it in `direct-lvm` mode. This significantly increases performance by leveraging an LVM `thinpool` backed by raw block devices.

Docker 17.06 and higher can configure `direct-lvm` for you. However, at the time of writing, it has some limitations. The main ones being; it will only configure a single block device, and it only works for fresh installations. This might change in the future, but a single block device will not give you the best in terms of performance and resiliency.

Letting Docker automatically configure `direct-lvm`

The following simple procedure will let Docker automatically configure `devicemapper` for `direct-lvm`.

1. Add the following storage driver configuration to `/etc/docker/daemon.json`

```
{
  "storage-driver": "devicemapper",
  "storage-opts": [
    "dm.directlvm_device=/dev/xdf",
    "dm.thinp_percent=95",
    "dm.thinp_metapercent=1",
    "dm.thinp_autoextend_threshold=80",
    "dm.thinp_autoextend_percent=20",
    "dm.directlvm_device_force=false"
  ]
}
```

Device Mapper and LVM are complex topics, and beyond the scope of a heterogeneous Docker book like this. However, let's quickly explain each option:

- `dm.directlvm_device` is where you specify the block device. For best performance and availability, this should be a dedicated high-performance device such as a local SSD, or RAID protected high performance LUN from an external storage array.

- `dm.thinp_percent=95` allows you to specify how much of the space you want Images and containers to be able to use. Default is 95%.
 - `dm.thinp_metapercent` sets the percentage of space to be used for metadata storage. Default is 1%.
 - `dm.thinp_autoextend_threshold` sets the threshold at which LVM should automatically extend the thinpool. The default value is currently 80%.
 - `dm.thinp_autoextend_percent` is the amount of space that should be added to the thin pool when an auto-extend operation is triggered.
 - `dm.directlvm_device_force` lets you specify whether or not to format the block device with a new filesystem.
2. Restart Docker.
 3. Verify that Docker is running and the devicemapper configuration is correctly loaded.

```
$ docker version
```

```
Client:
```

```
Version:      18.01.0-ce
```

```
<Snip>
```

```
Server:
```

```
Version:      18.01.0-ce
```

```
<Snip>
```

```
$ docker system info
```

```
<Snipped output only showing relevant data>
```

```
Storage Driver: devicemapper
```

```
Pool Name: docker-thinpool
```

```
Pool Blocksize: 524.3 kB
```

```
Base Device Size: 25 GB
```

```
Backing Filesystem: xfs
```

```
Data file:      << Would show a loop file if in loopback mode
```

```
Metadata file:  << Would show a loop file if in loopback mode
```

```
Data Space Used: 1.9 GB
```

```
Data Space Total: 23.75 GB
```

```
Data Space Available: 21.5 GB
```

```
Metadata Space Used: 180.5 kB
```

```
Metadata Space Total: 250 MB
```

```
Metadata Space Available: 250 MB
```

Although Docker will only configure `direct-lvm` mode with a single block device, it will still perform significantly better than `loopback` mode!

Manually configuring `devicemapper direct-lvm`

Walking you through the entire process of manually configuring device mapper `direct-lvm` is beyond the scope of this book. It is also something that can change and vary between OS versions. However, the following items are things you should know and consider when performing a configuration.

- **Block devices.** You need to have block devices available in order to configure `direct-lvm` mode. These should be high performance devices such as local SSD or high performance external LUNs. If your Docker environment is on-premises, external LUNs can be on FC, iSCSI, or other block-protocol storage arrays. If your Docker environment is in the public cloud, these can be any form of high performance block storage (usually SSD-based) supported by your cloud provider.
- **LVM config.** The `devicemapper` storage driver leverages LVM, the Linux Logical Volume Manager. This means you will need to configure the required physical devices (`pdev`), volume group (`vg`), logical volumes (`lv`), and thinpool (`tp`). You should use dedicated physical volumes and form them into a new volume group. You should not share the volume group with non-Docker workloads. You will also need to configure two logical volumes; one for data and the other for metadata. Create an LVM profile specifying the auto-extend threshold and auto-extend values, and configure monitoring so that auto-extend operations can happen.
- **Docker config.** Backup the current Docker config file (`/etc/docker/daemon.json`) and then update it as follows. The name of the `dm.thinpooldev` might be different in your environment and you should adjust as appropriate.

```
{
  "storage-driver": "devicemapper",
  "storage-opts": [
    "dm.thinpooldev=/dev/mapper/docker-thinpool",
    "dm.use_deferred_removal=true",
    "dm.use_deferred_deletion=true"
  ]
}
```

Once the configuration is saved you can start the Docker daemon.

For more detailed information, see the Docker documentation or talk to your Docker technical account manager.

Chapter Summary

Docker is available for Linux and Windows, and has a Community Edition (CE) and an Enterprise Edition (EE). In this chapter, we looked at some of the ways to install Docker on Windows 10, Mac OS X, Linux, and Windows Server 2016.

We looked at how to upgrade the Docker Engine on Ubuntu 16.04 and Windows Server 2016, as these are two of the most common configurations.

We also learned that selecting the right *storage driver* is essential when using Docker on Linux in production environments.

4: The big picture

The idea of this chapter is to paint 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

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

In the Dev Perspective section, we'll focus more on the app. We'll 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.** DevOps anyone?

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 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, is a single Docker host with an internet connection. This 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!

Another great way to get Docker, and get it fast, is Play With Docker (PWD). Play With Docker is a web-based Docker playground that you can use for free. Just point your web browser to <https://play-with-docker.com/> and you're ready to go (you may need a Docker Hub account to be able to login). It's my favourite way of spinning up temporary Docker environment!

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 Engine API](https://docs.docker.com/engine/api/v1.35/)¹⁴.

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 `npipes://./pipe/docker_engine`. You can use the `docker version` command to test that the client and daemon (server) are running and talking to each other.

```
> docker version
```

```
Client:
```

```
Version:      18.01.0-ce
API version:  1.35
Go version:   go1.9.2
Git commit:   03596f5
Built: Wed Jan 10 20:11:05 2018
OS/Arch:     linux/amd64
Experimental: false
Orchestrator: swarm
```

```
Server:
```

```
Engine:
```

```
Version:      18.01.0-ce
API version:  1.35 (minimum version 1.12)
Go version:   go1.9.2
Git commit:   03596f5
Built:       Wed Jan 10 20:09:37 2018
OS/Arch:     linux/amd64
Experimental: false
```

¹⁴<https://docs.docker.com/engine/api/v1.35/>

If you get a response back from the `Client` **and** `Server`, you're 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 the book with `sudo`.

Images

It's useful to think of a Docker image 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, or Play With Docker, you 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` image. If you are following along on Windows, pull the `microsoft/powershell:nanoserver` image.

```
latest: Pulling from library/ubuntu
50aff78429b1: Pull complete
f6d82e297bce: Pull complete
275abb2c8a6f: Pull complete
9f15a39356d6: Pull complete
fc0342a94c89: Pull complete
Digest: sha256:fbaf303...c0ea5d1212
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       00fd29ccc6f1  3 weeks ago   111MB
```

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 know 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 images, you can refer to them using either IDs or names. If you're working with image ID's, it's usually enough just to type the first few characters of the ID — as long as it's unique, Docker will know which image you mean.

Containers

Now that we have an image pulled locally, 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 pwsh.exe
```

```
Windows PowerShell
```

```
Copyright (C) 2016 Microsoft Corporation. All rights reserved.
```

```
PS C:\>
```

Look closely at the output from the previous commands. You should notice that the shell prompt has changed in each instance. This is because the `-it` flags switch your shell into the terminal of the 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 Docker to make the container interactive and to attach our current shell to the container's terminal (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 we're 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:

4: The big picture

52

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

The Linux container only has two processes:

- 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 can be a bit misleading, as it is a short-lived process that dies as soon as the `ps` command exits. This means the only long-running process inside of the container is the `/bin/bash` process.

The Windows container has a lot more going on. This is an artefact of the way the Windows Operating System works. However, even though 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 your shell back at the terminal 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-agent
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
```

4: The big picture

54

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 their respective containers. Windows containers run far fewer processes than Windows hosts, and Linux containers run far less than Linux hosts.

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	<code>"/bin/bash"</code>	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 the 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 the terminal of a running container with the `docker container exec` command. As the container from the previous steps is still running, let's make a new connection 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 pwsh.exe
```

```
Windows PowerShell
```

```
Copyright (C) 2016 Microsoft Corporation. All rights reserved.
```

```
PS C:\>
```

Notice that your shell prompt has changed again. You are logged in to the container again.

The format of the `docker container exec` command is: `docker container exec <options> <container-name or container-id> <command/app>`. 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 hex 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.

4: The big picture

56

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

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

Verify that the container was successfully deleted by running the `docker container ls` command with the `-a` flag. Adding `-a` tells Docker to list all containers, even those in the stopped state.

```
$ docker container ls -a
```

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 cloned from: <https://github.com/nigelpoulton/psweb.git>

The Windows app can be cloned 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 following repo 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 its contents.

```
$ 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 an app into a Docker image.

List the contents of the `Dockerfile`.

4: The big picture

58

```
$ 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 isn't 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 on how to build the app into a Docker image.

Use the `docker image build` command to create a new image using the instructions 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 f154cb3ddb4
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 image being pulled.

Once the build is complete, 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 Docker image with the app inside.

Run a container from the image and test the app.

Linux example:

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

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

If you are following along with Docker for Windows or Docker for Mac, you will be able to use `localhost:8080` or `127.0.0.1:8080`. If you're following along on Play with Docker, you will be able to click the `8080` hyperlink above the terminal screen.

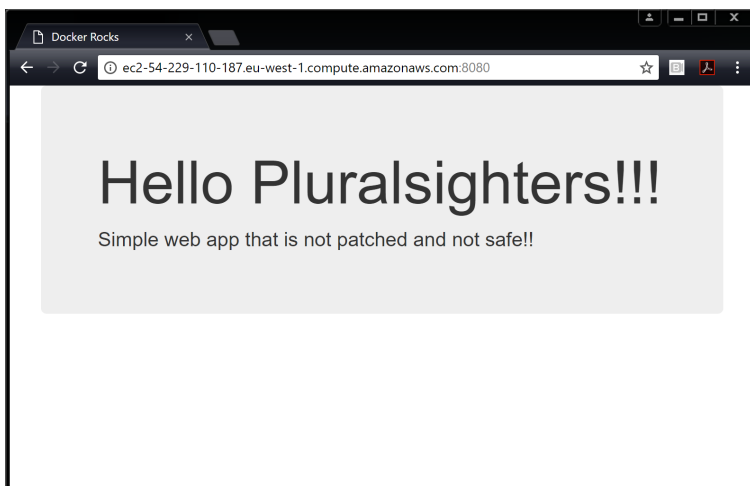


Figure 4.1

Windows example:

```
> docker container run -d \  
  --name web1 \  
  --publish 8080:8080 \  
  test:latest
```

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

The same rules apply if you're following along with Docker for Windows or Play with Docker.

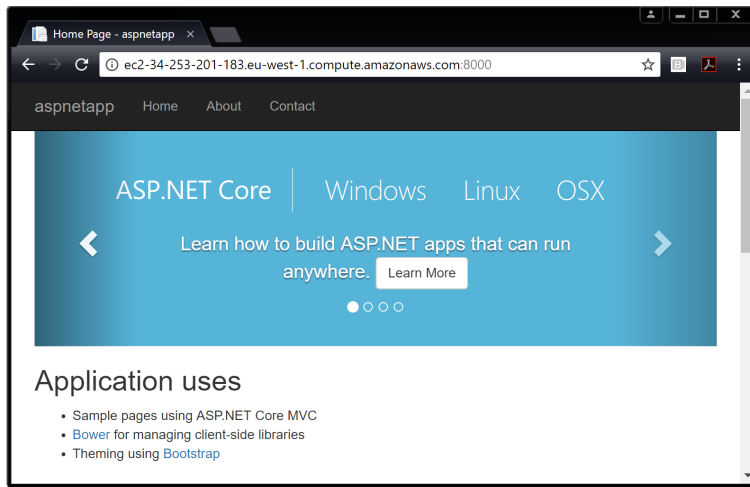


Figure 4.2

Well done. You’ve taken some application code from a remote Git repo and built it into a Docker image. You then ran a container from it. We call this “containerizing an app”.

Chapter Summary

In the Op section of the chapter you; downloaded a Docker image, launched a container from it, logged into the container, executed a command inside of it, and then stopped and deleted the container.

In the Dev section, you containerized a simple application by pulling some source code from GitHub and building it into an image using instructions in a Dockerfile. You then ran the containerized app.

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. So, to be a *real* Docker master, you need to know the stuff in this chapter.

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 and 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.

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 — 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.

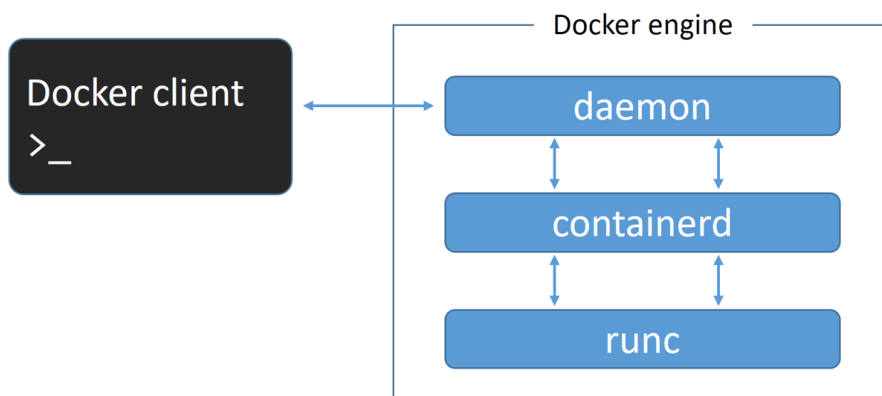


Figure 5.1

Throughout the book we'll refer to *runc* and *containerd* with lower-case “r” and “c”. This means sentences starting with either `__r__unc` `__c__ontainerd` will not start with a capital letter. This is intentional and not a mistake.

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 contained all of the code for the Docker client, the Docker API, the container runtime, image builds, and **much** more.

LXC provided the daemon with access to the fundamental building-blocks of containers that existed in the Linux kernel. Things like *namespaces* and *control groups* (*cgroups*).

Figure 5.2. shows how the daemon, LXC, and the OS, interacted in older versions of Docker.

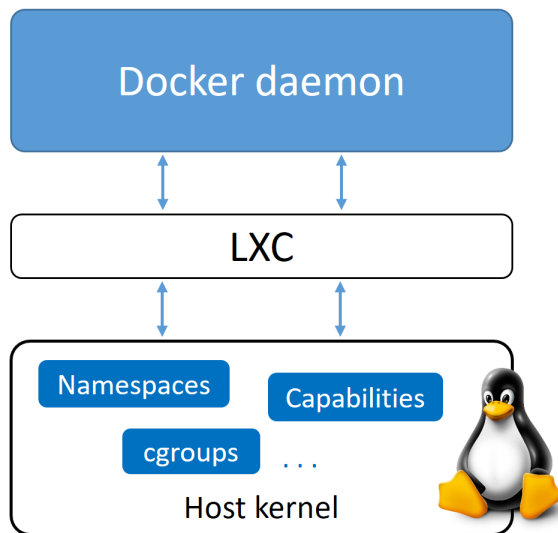


Figure 5.2 Previous Docker architecture

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 kernel.

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 was 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 re-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 current Docker engine architecture with brief descriptions.

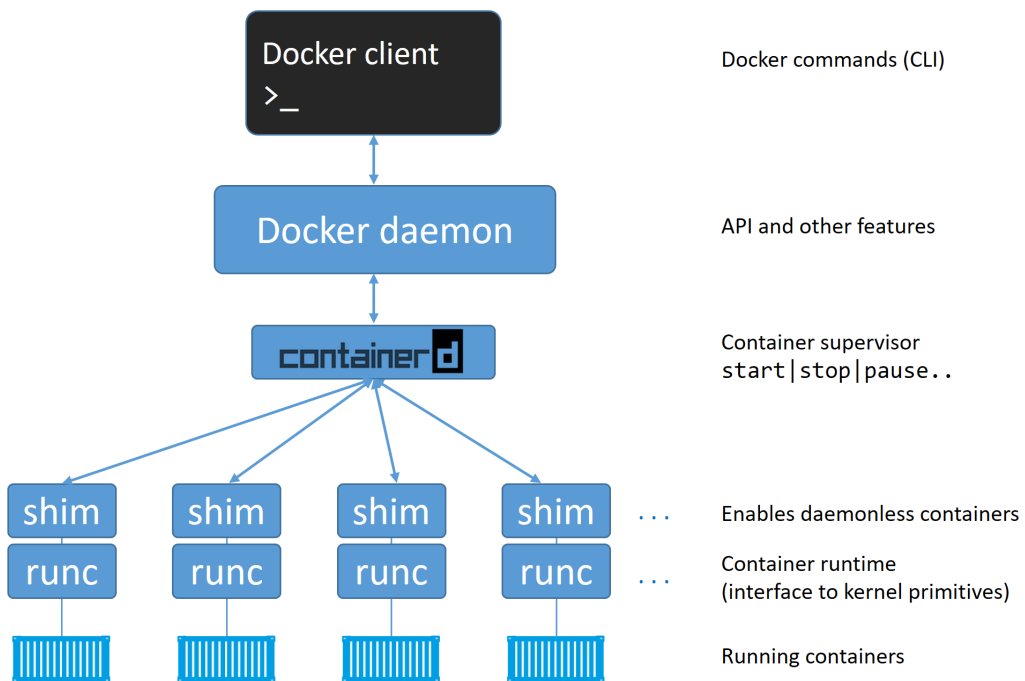


Figure 5.3

The influence of the Open Container Initiative (OCI)

While Docker, Inc. was breaking the daemon apart and refactoring code, the [OCI](https://www.opencontainers.org/)¹⁵ was in the process of defining two container-related specifications (a.k.a. standards):

1. [Image spec](https://github.com/opencontainers/image-spec)¹⁶
2. [Container runtime spec](https://github.com/opencontainers/runtime-spec/blob/master/RELEASES.md)¹⁷

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.

¹⁵<https://www.opencontainers.org/>

¹⁶<https://github.com/opencontainers/image-spec>

¹⁷<https://github.com/opencontainers/runtime-spec/blob/master/RELEASES.md>

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. This is the *runc* container runtime layer in Figure 5.3. A goal of the *runc* project be in-line with the OCI spec. However, now that the OCI spec's are both at 1.0, we shouldn't expect them to iterate too much — stability is the name of the game here.

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*.

If you strip everything else away, *runc* is a small, lightweight CLI wrapper for *libcontainer* (remember that *libcontainer* originally replaced *LXC* in the early Docker architecture).

runc has a single purpose in life — create containers. And it's damn good at it. And fast! But as it's a CLI wrapper, it's effectively a standalone container runtime tool. This means you can download and build the binary, and you'll have everything you need to build and play with *runc* (OCI) containers. But it's bare bones, you'll have none of the richness that you get with the full-blown Docker engine.

We sometimes call the layer that *runc* operates at, “the OCI layer”. See Figure 5.3.

You can see *runc* release information at:

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

containerd

As part of the effort to strip functionality out of the Docker daemon, all of the container execution logic was ripped out and refactored into a new tool called containerd (pronounced container-dee). Its sole purpose in life was to manage container lifecycle operations — `start` | `stop` | `pause` | `rm`...

containerd is available as a daemon for Linux and Windows, and Docker has been using it on Linux since the 1.11 release. In the Docker engine stack, containerd sits between the daemon and runc at the OCI layer. Kubernetes can also use containerd via cri-containerd.

As previously stated, containerd was originally intended to be small, lightweight, and designed for a single task in life — container lifecycle operations. However, over time it has branched out and taken on more functionality. Things like image management.

One of the reasons for this, is to make it easier to use in other projects. For example, containerd is a popular container runtime in Kubernetes. However, in projects like Kubernetes, it was beneficial for containerd to be able to do additional things like push and pull images. For these reasons, containerd now does a lot more than simple container lifecycle management. However, all the extra functionality is modular and optional, meaning you can pick and choose which bits you want. So it's possible to include containerd in projects such as Kubernetes, but only to take the pieces your project needs.

containerd was developed by Docker, Inc. and donated to the Cloud Native Computing Foundation (CNCF). It released version 1.0 in December 2017. You can see 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](https://grpc.io/)¹⁸.

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 (namespaces, cgroups etc.). 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.

The process is summarized in Figure 5.4.

¹⁸<https://grpc.io/>

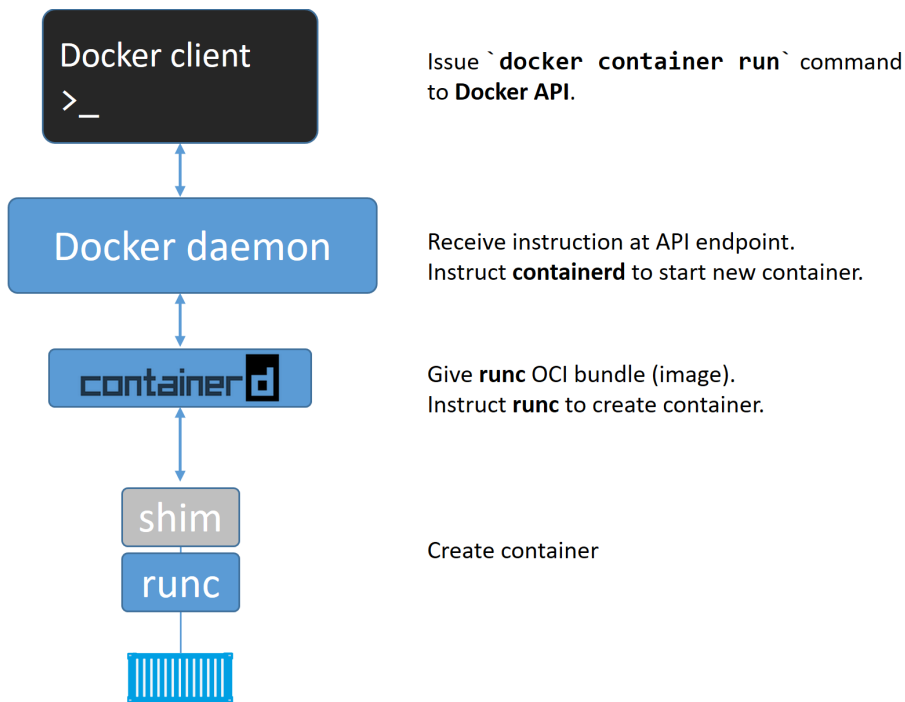


Figure 5.4

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 (what we just mentioned about decoupling running containers from the daemon for things like daemon upgrades).

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. 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.

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. For example, it's poised to become the default, and most common, container runtime in Kubernetes.

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 CLI tool to create containers. It's based on code from *libcontainer*, and can also be used by other projects and third-party tools.

There is still a lot of functionality implemented in 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, and volumes.

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 and 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](https://hub.docker.com)¹⁹, 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!

¹⁹<https://hub.docker.com>

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.

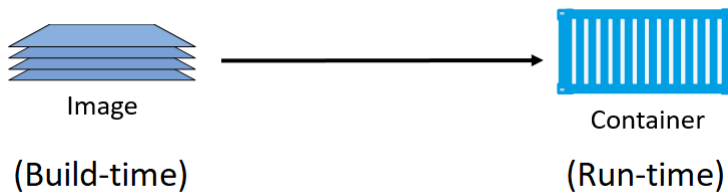


Figure 6.1

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 110MB. 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 1.7GB when pulled and uncompressed. The Windows Server 2016 Nano Server image (microsoft/nanoserver:latest) 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 following commands 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:

6: Images

78

```
> 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
```

```
ae0b4393afb: 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
```

```
cfdf178ff221e: 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 just pulled are now present in the Docker host's local repository. You can also see that the Windows images are a lot larger and comprise a lot more layers.

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.

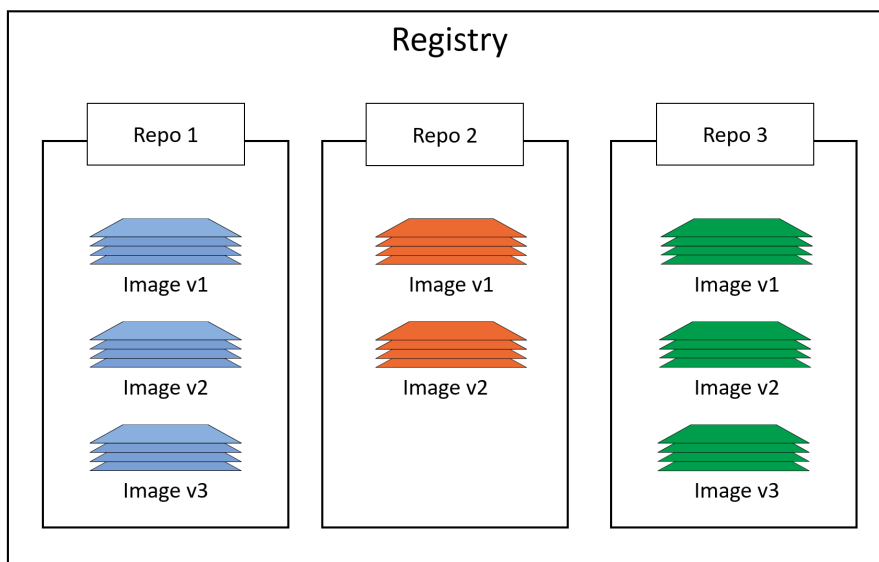


Figure 6.2

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 following list 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! Here 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 about those commands.

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 following example shows how to pull the v2 image from the tu-demo repository owned by a not-to-be-trusted 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 from 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	6ac21e..bead	1 yr ago	211.6 MB
nigelpoulton/tu-demo	latest	9b915a..1e29	1 yr ago	211.6 MB
nigelpoulton/tu-demo	v1	9b915a..1e29	1 yr ago	211.6 MB

A couple of things about what just happened:

First, the command pulled three images from the `nigelpoulton/tu-demo` 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 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 a perfect example of the warning 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!

Filtering the output of `docker image ls`

Docker provides the `--filter` flag to filter the list of images returned by `docker image ls`.

The following example will only return dangling images.

```
$ docker image ls --filter dangling=true
REPOSITORY    TAG        IMAGE ID      CREATED       SIZE
<none>        <none>     4fd34165afe0  7 days ago   14.5MB
```

A dangling image is an image that is no longer tagged, and appears in listings as `<none>:<none>`. A common way they occur is when building a new image and tagging it with an existing tag. When this happens, Docker will build the new image, notice that an existing image has a matching tag, remove the tag from the existing image, give the tag to the new image. For example, you build a new image based on `alpine:3.4` and tag it as `dodge:challenger`. Then you update the Dockerfile to replace `alpine:3.4` with `alpine:3.5` and run the exact same `docker image build` command. The build will create a new image tagged as `dodge:challenger` and remove the tags from the older image. The old image will become a dangling image.

You can delete all dangling images on a system with the `docker image prune` command. If you add the `-a` flag, Docker will also remove all unused images (those not in use by any containers).

Docker currently supports the following filters:

- `dangling`: Accepts `true` or `false`, and returns only dangling images (`true`), or non-dangling images (`false`).
- `before`: Requires an image name or ID as argument, and returns all images created before it.
- `since`: Same as above, but returns images created after the specified image.
- `label`: Filters images based on the presence of a label or label and value. The `docker image ls` command does not display labels in its output.

For all other filtering you can use reference.

Here's an example using reference to display only images tagged as "latest".

```
$ docker image ls --filter=reference="*:latest"
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
alpine        latest    3fd9065eaf02   8 days ago    4.15MB
test          latest    8426e7efb777   3 days ago    122MB
```

You can also use the `--format` flag to format output using Go templates. For example, the following command will only return the size property of images on a Docker host.

```
$ docker image ls --format "{{.Size}}"
99.3MB
111MB
82.6MB
88.8MB
4.15MB
108MB
```

Use the following command to return all images, but only display repo, tag and size.

```
$ docker image ls --format "{{.Repository}}: {{.Tag}}: {{.Size}}"
dodge: challenger: 99.3MB
ubuntu: latest: 111MB
python: 3.4-alpine: 82.6MB
python: 3.5-alpine: 88.8MB
alpine: latest: 4.15MB
nginx: latest: 108MB
```

If you need more powerful filtering, you can always use the tools provided by your OS and shell such as `grep` and `awk`.

Searching Docker Hub from the CLI

The `docker search` command lets you search Docker Hub from the CLI. You can pattern match against strings in the “NAME” field, and filter output based on any of the returned columns.

In its simplest form, it searches for all repos containing a certain string in the “NAME” field. For example, the following command searches for all repos with “nigelpoulton” in the “NAME” field.

```
$ docker search nigelpoulton
```

NAME	DESCRIPTION	STARS	AUTOMATED
nigelpoulton/pluralsight..	Web app used in...	8	[OK]
nigelpoulton/tu-demo		7	
nigelpoulton/k8sbook	Kubernetes Book web app	1	
nigelpoulton/web-fe1	Web front end example	0	
nigelpoulton/hello-cloud	Quick hello-world image	0	

The “NAME” field is the repository name, and includes the Docker ID, or organization name, for unofficial repositories. For example, the following command will list all repositories that include the string “alpine” in the name.

```
$ docker search alpine
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
alpine	A minimal Docker..	2988	[OK]	
mhart/alpine-node	Minimal Node.js..	332		
anapsix/alpine-java	Oracle Java 8...	270		[OK]

<Snip>

Notice how some of the repositories returned are official and some are unofficial. You can use `--filter "is-official=true"` so that only official repos are displayed.

```
$ docker search alpine --filter "is-official=true"
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
alpine	A minimal Docker..	2988	[OK]	

You can do the same again, but this time only show repos with automated builds.

```
$ docker search alpine --filter "is-automated=true"
```

NAME	DESCRIPTION	OFFICIAL	AUTOMATED
anapsix/alpine-java	Oracle Java 8 (and 7)...		[OK]
frolvlad/alpine-glibc	Alpine Docker image..		[OK]
kiasaki/alpine-postgres	PostgreSQL docker..		[OK]
zzrot/alpine-caddy	Caddy Server Docker..		[OK]

<Snip>

One last thing about `docker search`. By default, Docker will only display 25 lines of results. However, you can use the `--limit` flag to increase that to a maximum of 100.

Images and layers

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

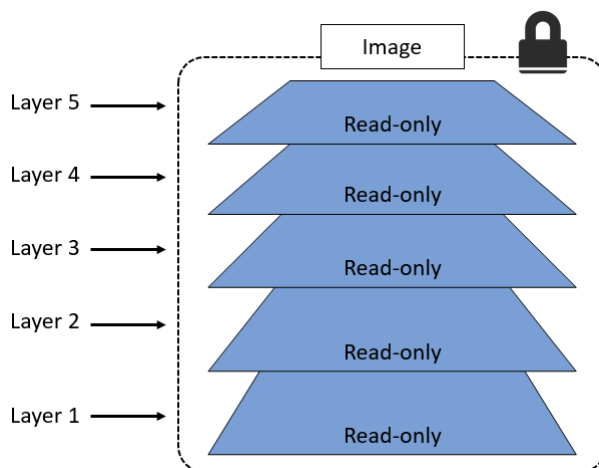


Figure 6.3

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 shows this in picture form, displaying layer IDs.

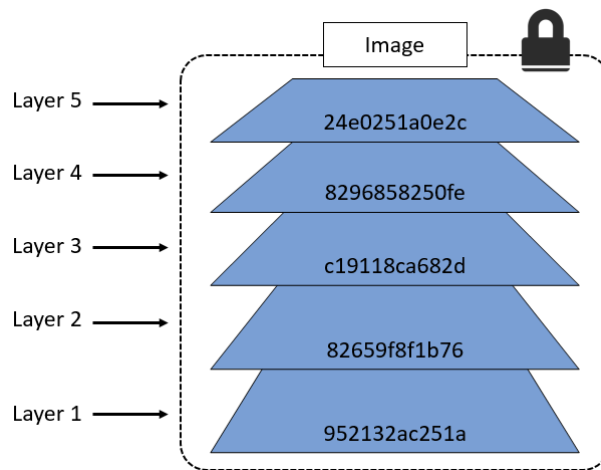


Figure 6.4

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

6: Images

90

```
$ 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; “ENV”, “EXPOSE”, “CMD”, and “ENTRY-POINT”. 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).

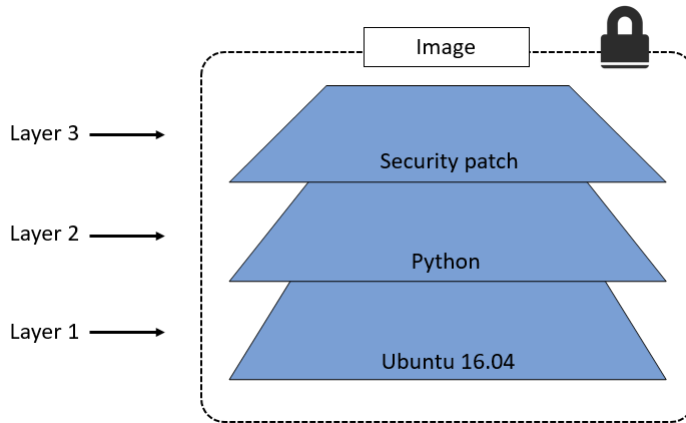


Figure 6.5

It's important to understand that as additional layers are added, the *image* is always 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.

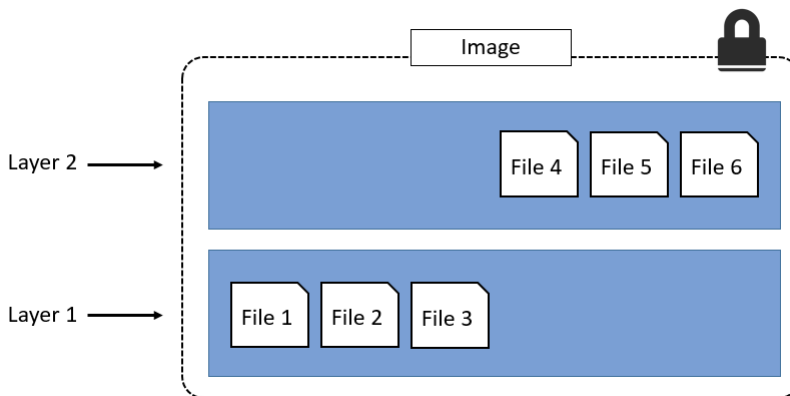


Figure 6.6

Note: We've shown the image layers in Figure 6.6 in a slightly different way to previous figures. This is just to make showing the 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.

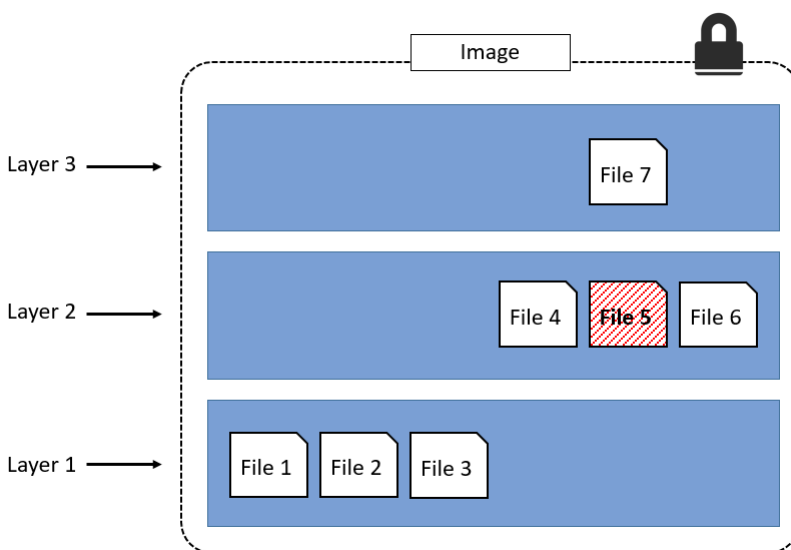


Figure 6.7

Docker employs a storage driver (snapshotter in newer versions) 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`, which implements layering and CoW on top of NTFS.

Figure 6.8 shows the same 3-layer image as it will appear to the system. I.e. all three layers stacked and merged, giving a single unified view.

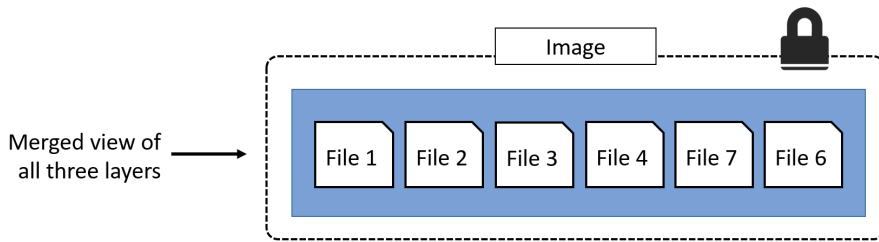


Figure 6.8

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 previously 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 pulled 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 storage drivers (snapshotters). Each is free to implement image layering, layer sharing, and copy-on-write (CoW) 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 the wrong 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 `golfttrack:1.5` and it has a known bug. You pull the image, apply a fix, and push the updated image back to its repository using the **same tag**.

Take a second to understand what just happened there... You have an image called `golfttrack:1.5` that has a bug. That image is being used in your production environment. You create a new version of the image that includes a fix. Then comes the mistake... you build and push the fixed image back to its repository with the **same**

tag as the vulnerable image!. This overwrites the original image and leaves without a great way of knowing which of your production containers are running from the vulnerable image and which are running from the fixed 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 following example 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 some metadata.

The *layers* are where the data lives (files etc.). Each one is fully independent, and has 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, and we can easily identify any changes made to either.

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 its 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, it runs a hash against the layer and checks to see if it matches the hash that was sent. 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 verify 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

One of the best things about Docker is how simple it is to use. For example, running an application is as simple as pulling the image and running a container. No need to worry about setup, dependencies, or config. It just works.

However, as Docker grew, things started getting complex — especially when new platforms and architectures, such as Windows, ARM, and s390x were added. All of a sudden we have to think about whether the image we're pulling is built for the architecture we're running on. This breaks the smooth experience.

Multi-architecture images to the rescue!

Docker (image and registry specs) now supports multi-architecture images. This means a single image (repository:tag) *can* have an image for Linux on x64, Linux on PowerPC, Windows x64, ARM etc. Let me be clear, we're talking about a single image tag supporting multiple platforms and architectures. We'll see it in action in a second.

To make this happen, the Registry API supports two important constructs:

- **manifest lists** (new)

- **manifests**

The **manifest list** is exactly what it sounds like: a list of architectures supported by a particular image tag. Each supported architecture then has its own **manifest* detailing the layers it's composed from.

Figure 6.9 uses the official `golang` image as an example. On the left is the **manifest list** with entries for each architecture the image supports. The arrows show that each entry in the **manifest list** points to a **manifest** containing image config and layer data.

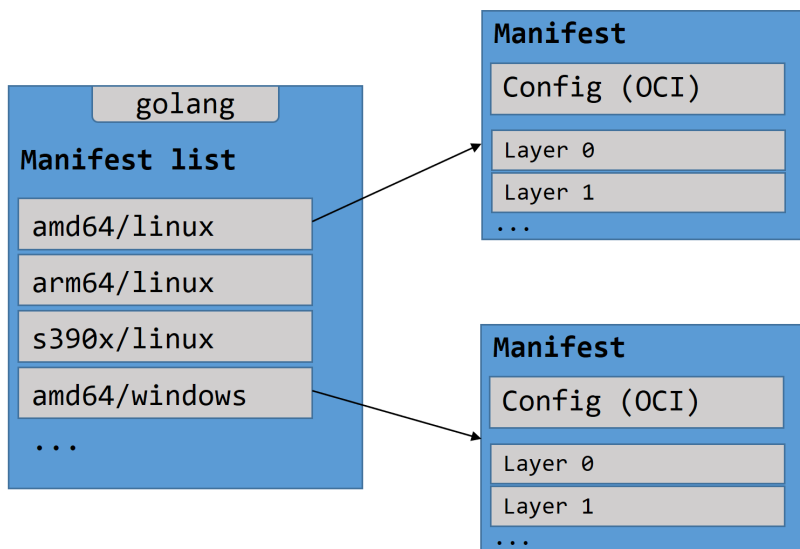


Figure 6.9

Let's look at the theory before seeing it in action.

Assume you are running Docker on a Raspberry Pi (Linux running on ARM architecture). When you pull an image, your Docker client makes the relevant calls to the Docker Registry API running on Docker Hub. If a **manifest list** exists for the image, it will be parsed to see if an entry exists for Linux on ARM. If an ARM entry exists, the **manifest** for that image is retrieved and parsed for the crypto ID's of the layers that make up the image. Each layer is then pulled from Docker Hub's blob store.

The following examples show how this works by pulling the official `golang` image (which supports multiple architectures) and running a simple command to show the version of Go along with the CPU architecture of the host. The thing to note, is that both examples use the exact same `docker container run` command. We do not have to tell Docker that we need the Linux x64 or Windows x64 versions of the image. We just run normal commands and let Docker take care of getting the right image for the platform and architecture we are running!

Linux on x64 example:

```
$ docker container run --rm golang go version
```

```
Unable to find image 'golang:latest' locally
latest: Pulling from library/golang
723254a2c089: Pull complete
<Snip>
39cd5f38ffb8: Pull complete
Digest: sha256:947826b5b6bc4...
Status: Downloaded newer image for golang:latest
go version go1.9.2 linux/amd64
```

Windows on x64 example:

```
PS> docker container run --rm golang go version
```

```
Using default tag: latest
latest: Pulling from library/golang
3889bb8d808b: Pull complete
8df8e568af76: Pull complete
9604659e3e8d: Pull complete
9f4a4a55f0a7: Pull complete
6d6da81fc3fd: Pull complete
72f53bd57f2f: Pull complete
6464e79d41fe: Pull complete
dca61726a3b4: Pull complete
9150276e2b90: Pull complete
cd47365a14fb: Pull complete
```

```
1783777af4bb: Pull complete
3b8d1834f1d7: Pull complete
7258d77b22dd: Pull complete
Digest: sha256:e2be086d86eeb789...e1b2195d6f40edc4
Status: Downloaded newer image for golang:latest
go version go1.9.2 windows/amd64
```

The previous operations pull the `golang` image from Docker Hub, start a container from it, execute the `go version` command, and output the version of Go and the OS/CPU architecture of the host system. The last line of each example shows the output of each `go version` command. See that both examples used exactly the same command, but the Linux example pulled the `linux/amd64` image, and the Windows example pulled the `windows/amd64` image.

At the time of writing, all *official images* have manifest lists. However, support for all architectures is an ongoing process.

Creating images that run on multiple architectures requires additional effort from the image publisher. Also, some software is not cross-platform. With this in mind, **manifest lists** are optional — if one doesn't exist for an image, the Registry will return the normal **manifest**.

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.

Deleting an image will remove the image and all of its layers from your Docker host. This means it will no longer show up in `docker image ls` commands, and all directories on the Docker host containing the layer data will be deleted. However, if an image layer is shared by more than one image, that layer will not be deleted until all images that reference it have been deleted.

Delete the images pulled in the previous steps with the `docker image rm` command. The following example 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 delete operation again.

A handy shortcut for **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 next.

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 next.

```
$ 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 an image — layer data and metadata.
- `docker image rm` is the command to delete images. This command shows how to delete the `alpine: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 looked at how Docker supports multi-architecture and multi-platform images, and 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, Docker has been hard at work implementing the image and container specs published by the Open Container Initiative (OCI) at <https://www.opencontainers.org>. This means a lot 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.

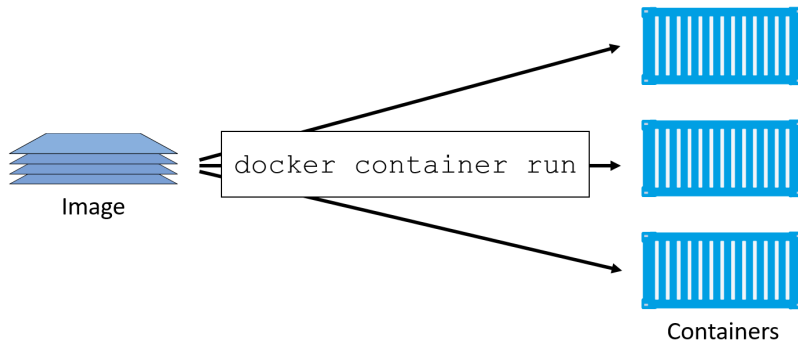


Figure 7.1

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 app to run: `docker container run <image> <app>`. This next command will start an Ubuntu Linux container running the Bash shell as its app: `docker container run -it ubuntu /bin/bash`. To start a Windows container running the PowerShell app, you could do `docker container run -it microsoft-/powershell:nanoserver pwsh.exe`.

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

Containers run until the app 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!

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.

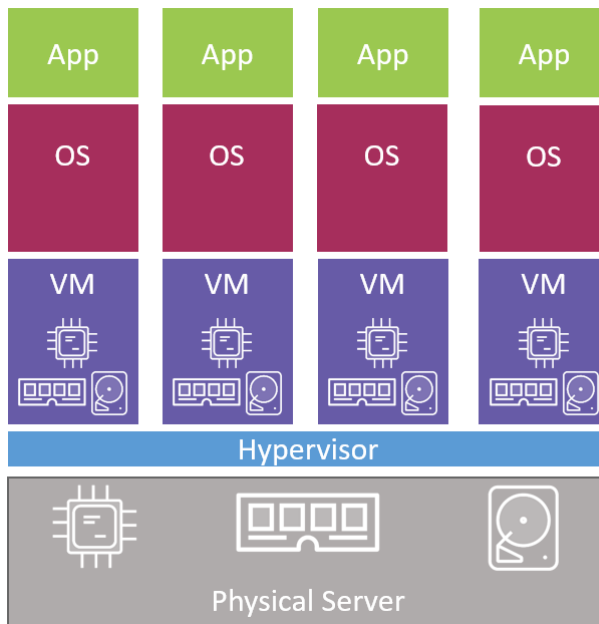


Figure 7.2

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. Similar to 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. This is shown in Figure 7.3.

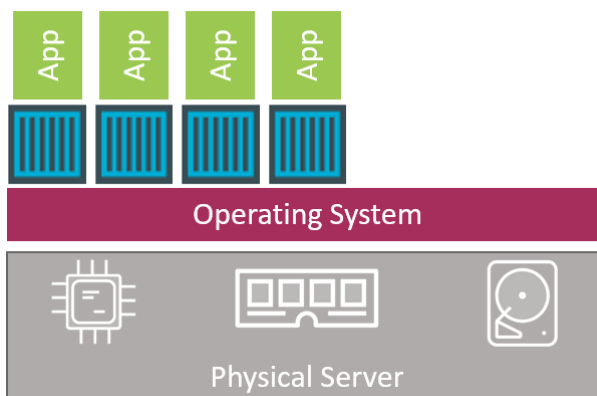


Figure 7.3

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 a single physical server and the 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, this can be game changing.

Another thing to consider is start times. 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.

7: Containers

110

```
$ 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, 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

If 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 following command 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 pwsh.exe
```

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

Let's break the command down.

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 app 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 the example cited, 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 local 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 `npip:////./pipe/docker_engine`. It's also possible to configure the Docker client and daemon to communicate 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 app 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 cited, 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 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 following example 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:/# 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 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. 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 kills 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:/#
```

The command to re-attach to the Windows Nano Server PowerShell container would be `docker container exec -it <container-name-or-ID> pwsh.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` in this shell will not terminate the container, because the original Bash or PowerShell process will continue running.

Type `exit` to leave the container and verify it's still running with a `docker container ps`. It will still be 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 containers 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 following examples 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# ls -l
total 0

root@9cb2d2fd1d65:/tmp# echo "DevOps FTW" > newfile

root@9cb2d2fd1d65:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 14 May 23 11:22 newfile

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
DevOps 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 next example 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 *serverless function* 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 its data will always be safe. It's not until you explicitly delete 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` app. 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 and shooting it in the back of the head. You're literally giving the container, and the app 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 PID 1 process 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 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!

Self-healing containers with restart policies

It's often a good idea to run containers with a restart policy. It's a form of self-healing that enables Docker to automatically restart them after certain events or failures have occurred.

Restart policies are applied per-container, and can be configured imperatively on the command line as part of `docker container run` commands, or declaratively in Compose files for use with Docker Compose and Docker Stacks.

At the time of writing, the following restart policies exist:

- `always`
- `unless-stopped`
- `on-failed`

The **always** policy is the simplest. It will always restart a stopped container unless it has been explicitly stopped, such as via a `docker container stop` command. An easy way to demonstrate this is to start a new interactive container with the `--restart always` policy, and tell it to run a shell process. When the container starts you will be attached to its shell. Typing `exit` from the shell will kill the container's PID 1 process and therefore kill the container. However, Docker will automatically

restart it because it was started with the `--restart always` policy. If you issue a `docker container ls` command, you will see that the container's uptime will be less than the time since it was created. We show this in the following example.

If you're following along with Windows, substitute the `docker container run` command in the example with this one: `docker container run --name neversaydie -it --restart always microsoft/powershell:nanoserver`.

```
$ docker container run --name neversaydie -it --restart always alpine sh
```

```
//Wait a few seconds before typing the exit command
```

```
/# exit
```

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
0901afb84439	alpine	<code>"sh"</code>	35 seconds ago	Up 1 second

Notice that the container was created 35 seconds ago, but has only been up for 1 second. This is because we killed it when we issued the `exit` command from within the container, and Docker has had to restart it.

An interesting feature of the `--restart always` policy is that a stopped container will be restarted when the Docker daemon starts. For example, you start a new container with the `--restart always` policy and then stop it with the `docker container stop` command. At this point the container is in the Stopped (Exited) state. However, if you restart the Docker daemon, the container will be automatically restarted when the daemon comes back up.

The main difference between the **always** and **unless-stopped** policies is that containers with the `--restart unless-stopped` policy will not be restarted when the daemon restarts if they were in the Stopped (Exited) state. That might be a confusing sentence, so let's walk through an example.

We'll create two new containers. One called "always" with the `--restart always` policy, and one called "unless-stopped" with the `--restart unless-stopped` policy. We'll stop them both with the `docker container stop` command and then restart Docker. The "always" container will restart, but the "unless-stopped" container will not.

1. Create the two new containers

```
$ docker container run -d --name always \
  --restart always \
  alpine sleep 1d

$ docker container run -d --name unless-stopped \
  --restart unless-stopped \
  alpine sleep 1d

$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  STATUS              NAMES
3142bd91ecc4   alpine    "sleep 1d"              Up 2 secs           unless-stopped
4f1b431ac729   alpine    "sleep 1d"              Up 17 secs          always
```

We now have two containers running. One called “always” and one called “unless-stopped”.

1. Stop both containers

```
$ docker container stop always unless-stopped

$ docker container ls -a
CONTAINER ID   IMAGE     STATUS              NAMES
3142bd91ecc4   alpine    Exited (137) 3 seconds ago  unless-stopped
4f1b431ac729   alpine    Exited (137) 3 seconds ago  always
```

2. Restart Docker.

The process for restarting Docker is different on different Operating Systems. This example shows how to stop Docker on Linux hosts running systemd. To restart Docker on Windows Server 2016 use `restart-service Docker`.

```
$ systemctl restart docker
```

1. Once Docker has restarted, you can check the status of the containers.

```
$ docker container ls -a
```

CONTAINER	CREATED	STATUS	NAMES
314..cc4	2 minutes ago	Exited (137) 2 minutes ago	unless-stopped
4f1..729	2 minutes ago	Up 9 seconds	always

Notice that the “always” container (started with the `--restart always` policy) has been restarted, but the “unless-stopped” container (started with the `--restart unless-stopped` policy) has not.

The **on-failure** policy will restart a container if it exits with a non-zero exit code. It will also restart containers when the Docker daemon restarts, even containers that were in the stopped state.

If you are working with Docker Compose or Docker Stacks, you can apply the restart policy to a service object as follows:

```
version: "3"
services:
  myservice:
    <Snip>
    restart_policy:
      condition: always | unless-stopped | on-failure
```

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 delete 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](https://www.pluralsight.com/search?q=nigel%20poulton%20docker&categories=all)²⁰. 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.

²⁰<https://www.pluralsight.com/search?q=nigel%20poulton%20docker&categories=all>

```
$ 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` stands for **daemon** mode, and tells the container to run in the background.

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 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.

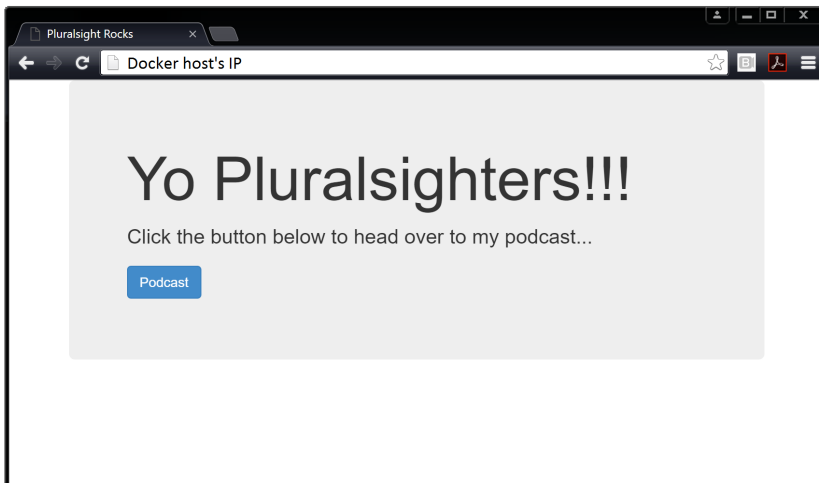


Figure 7.4

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 an app 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 an instruction that lists the default app you want containers using the image to run. If we run a `docker image inspect` against the image we used to run our container, we'll be able to see the app 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 it easier to find the information we're interested in.

The entries after `Cmd` show the command/app that the container will run unless you override with a different one when you launch the container with `docker container run`. If you remove all of the shell escapes in the example, you get the following command `/bin/sh -c "cd /src && node ./app.js"`. That's the default app 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 app it's supposed to run.

That's us done for the examples in this chapter. Let's see a quick way to tidy our system up.

Tidying up

Let's look at 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 application you want the container to run. This example will start an Ubuntu container in the foreground, and tell it to run 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 container 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 inherent in the VM model, and saw how the container model can bring huge advantages 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 PID 1 process 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 container metadata.

So far so good!

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 Linux web application. If you don’t have a Linux Docker environment to follow along with, you can use *Play With Docker* for free. Just point your web browser to <https://play-with-docker.com> and spin up some Linux Docker nodes. It’s my favourite way to spin up Docker and do testing!

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

- The TLDR
- The deep dive
- The commands

Let’s 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 in picture form.

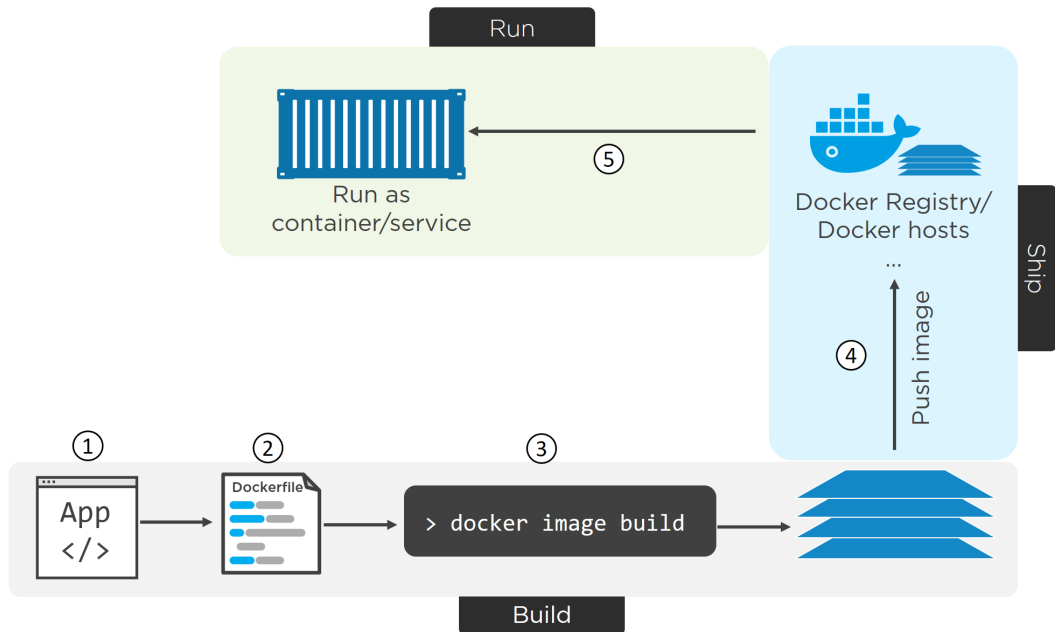


Figure 8.1 - Basic flow of containerizing an app

Containerizing an app - The deep dive

We'll break up this Deep Dive section of the chapter as follows:

- Containerize a single-container app
- Moving to Production with multi-stage builds
- A few best practices

Containerize a single-container app

The rest of this chapter will walk you through the process of containerizing a simple single-container 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**
- A few best practices

Although we'll be working with a single-container app in this chapter, we'll move on to a multi-container app in the next chapter on Docker Compose. After that, we'll move on to an even more complicated app in the chapter on Docker Stacks.

Getting the application code

The application used in this example can be cloned from 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 it into an image.

The directory containing the application 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 as a form of documentation! It has the ability to help bridge the gap between development and operations! 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. As such, it should be treated as code, and checked into a source control system.

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, document the app's 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's 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 .

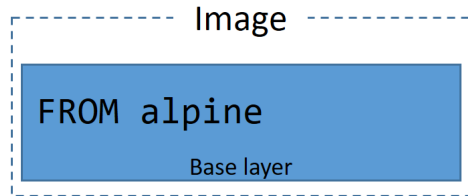


Figure 8.2

Next, the Dockerfile creates a LABEL that 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's 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.

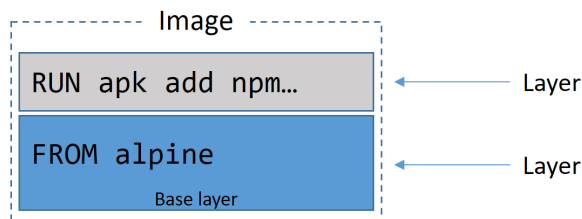


Figure 8.3

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

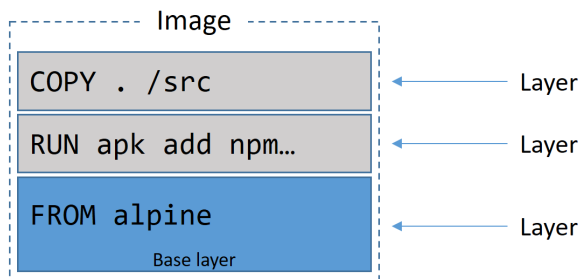


Figure 8.4

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 the `package.json` file in the build context. 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.

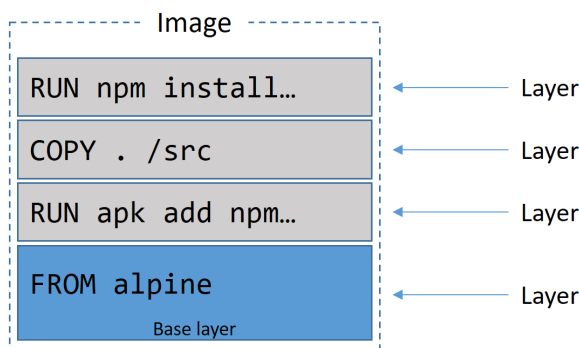


Figure 8.5

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 understand how it works, let's build it!

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 to 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 76.29kB
Step 1/8 : FROM alpine
latest: Pulling from library/alpine
ff3a5c916c92: Pull complete
Digest: sha256:7df6db5aa6...0bedab9b8df6b1c0
Status: Downloaded newer image for alpine:latest
--> 76da55c8019d
<Snip>
Step 8/8 : ENTRYPOINT node ./app.js
--> Running in 13977a4f3b21
--> fc69fdc4c18e
Removing intermediate container 13977a4f3b21
Successfully built fc69fdc4c18e
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	fc69fdc4c18e	10 seconds ago	64.4MB

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.

Pushing images

Once you’ve created an image, it’s a good idea to store it in an image registry to keep it safe and make it available to others. Docker Hub is the most common public image registry, and it’s the default push location for `docker image push` commands.

In order to push an image to Docker Hub, you need to login with your Docker ID. You also need tag the image appropriately.

Let’s log in to Docker Hub and push the newly created image.

In the following example’s you will need to substitute my Docker ID with your own. So any time you see “nigelpoulton”, swap it out for your Docker ID.

```
$ docker login
Login with **your** Docker ID to push and pull images from Docker Hub...
Username: nigelpoulton
Password:
Login Succeeded
```

Before you can push an image, you need to tag it in a special way. This is because Docker needs all of the following information when pushing an image:

- Registry
- Repository
- Tag

Docker is opinionated, so you don’t need to specify values for Registry and Tag. If you don’t specify values, Docker will assume `Registry=docker.io` and `Tag=latest`. However, Docker does not have a default value for the Repository value, it gets this from the “REPOSITORY” value of the image it is pushing. This might be confusing, so let’s take a closer look at the one from our example.

The previous `docker image ls` output shows our image with `web` as the repository name. This means a `docker image push` will try and push the image to `docker.io/web:latest`. However, I don't have access to the `web` repository, all of my images have to sit within the `nigelpoulton` second-level namespace. This means we need to re-tag the image to include my Docker ID.

```
$ docker image tag web:latest nigelpoulton/web:latest
```

The format of the command is `docker image tag <current-tag> <new-tag>` and it adds an additional tag, it does not overwrite the original.

Another image listing shows the image now has two tags, one of which includes my Docker ID.

```
$ docker image ls
```

REPO	TAG	IMAGE ID	CREATED	SIZE
web	latest	fc69fdc4c18e	10 secs ago	64.4MB
nigelpoulton/web	latest	fc69fdc4c18e	10 secs ago	64.4MB

Now we can push it to Docker Hub.

```
$ docker image push nigelpoulton/web:latest
The push refers to repository [docker.io/nigelpoulton/web]
2444b4ec39ad: Pushed
ed8142d2affb: Pushed
d77e2754766d: Pushed
cd7100a72410: Mounted from library/alpine
latest: digest: sha256:68c2dea730...f8cf7478 size: 1160
```

Figure 8.6 shows how Docker determined the push location.

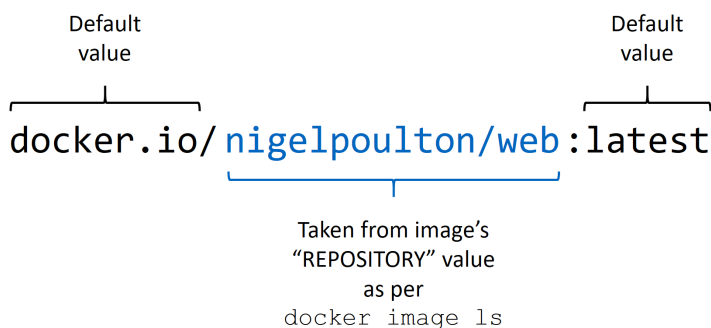


Figure 8.6

You will not be able to push images to repos in my Docker Hub namespace, you will have to use your own.

All of the examples in the rest of the chapter will use the shorter of the two tags (`web:latest`).

Run the app

The 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	STATUS	PORTS
49..	web:latest	"node ./app.js"	UP 6 secs	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 the app

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 .

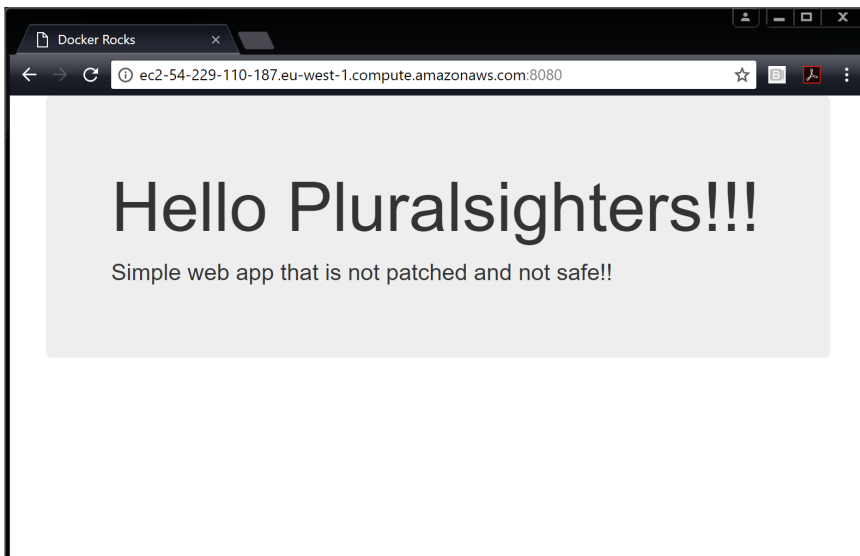


Figure 8.7

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 `EXPOSE`, `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
fc6..18e	/bin/sh -c #(nop) ENTRYPOINT ["node" "./a...	0B
334..bf0	/bin/sh -c #(nop) EXPOSE 8080/tcp	0B
b27..eae	/bin/sh -c npm install	14.1MB
932..749	/bin/sh -c #(nop) WORKDIR /src	0B
052..2dc	/bin/sh -c #(nop) COPY dir:2a6ed1703749e80...	22.5kB
c1d..81f	/bin/sh -c apk add --update nodejs nodejs-npm	46.1MB
336..b92	/bin/sh -c #(nop) LABEL maintainer=nigelp...	0B
3fd..f02	/bin/sh -c #(nop) CMD ["/bin/sh"]	0B
<missing>	/bin/sh -c #(nop) ADD file:093f0723fa46f6c...	4.15MB

Two things from the output above are worth noting.

First. Each line corresponds to an instruction in the Dockerfile (starting from the bottom and working up). The CREATED BY column even lists the exact Dockerfile instruction that was executed.

Second. Only 4 of the lines displayed in the output create new layers (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 might look like they create layers, they actually create metadata instead of layers. The reason that the `docker image history` output makes it look like all instructions create layers is an artefact of the way Docker builds and image layering 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:cd7100...1882bd56d263e02b6215",
    "sha256:b3f88e...cae0e290980576e24885",
    "sha256:3cfa21...cc819ef5e3246ec4fe16",
```

```

    "sha256:4408b4...d52c731ba0b205392567"
  ],
},

```

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 potential vulnerabilities.

You can view the output of the `docker image build` command to see the general process for building an image. As the following snippet 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 e690ddca785f    << Run inside of temp container
fetch http://dl-cdn...APKINDEX.tar.gz
fetch http://dl-cdn...APKINDEX.tar.gz
(1/10) Installing ca-certificates (20171114-r0)
<Snip>
OK: 61 MiB in 21 packages
---> c1d31d36b81f              << Create new layer
Removing intermediate container << Remove temp container
Step 4/8 : COPY . /src

```

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 more potential vulnerabilities and possibly a bigger 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 just the application stuff from 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 from the top starting at 0. However, we've also given each stage a friendly name.

- Stage 0 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 from 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.

An important thing to note, is that `COPY --from` instructions are used to **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 build` 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.

8: Containerizing an app

148

```
$ 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 build 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.

A few best practices

Let's list a few best practices before closing out the chapter. This list is not intended to be exhaustive.

Leverage the build cache

The build process used by Docker has the concept of a cache. The best way to see the impact of the cache is to build a new image on a clean Docker host, then

repeat the same build immediately after. The first build will pull images and take time building layers. The second build will complete almost instantaneously. This is because artefacts from the first build, such as layers, are cached and leveraged by later builds.

As we know, the `docker image build` process iterates through a Dockerfile one-line-at-a-time starting from the top. For each instruction, Docker looks to see if it already has an image layer for that instruction in its cache. If it does, this is a *cache hit* and it uses that layer. If it doesn't, this is a *cache miss* and it builds a new layer from the instruction. Getting *cache hits* can hugely speed up the build process.

Let's look a little closer.

We'll use this example Dockerfile to provide a quick walk-through:

```
FROM alpine
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

The first instruction tells Docker to use the `alpine:latest` image as its *base image*. If this image already exists on the host, the build will move on to the next instruction. If the image does not exist, it is pulled from Docker Hub ([docker.io](https://hub.docker.io)).

The next instruction (`RUN apk . . .`) runs a command against the image. At this point, Docker checks its build cache for a layer that was built from the same base image, as well as using the same instruction it is currently being asked to execute. In this case, it's looking for a layer that was built directly on top of `alpine:latest` by executing the `RUN apk add --update nodejs nodejs-npm` instruction.

If it finds a layer, it skips the instruction, links to that existing layer, and continues the build with the cache in tact. If it does **not** find a layer, it invalidates the cache and builds the layer. This operation of invalidating the cache invalidates it for the remainder of the build. This means all subsequent Dockerfile instructions are completed in full without attempting to reference the build cache.

Let's assume that Docker already had a layer for this instruction in the cache (a cache hit). And let's assume the ID of that layer was AAA.

The next instruction copies some code into the image (`COPY . /src`). Because the previous instruction resulted in a cache hit, Docker now checks to see if it has a cached layer that was built from the AAA layer with the `COPY . /src` command. If it does, it links to the layer and proceeds to the next instruction. If it does not, it builds the layer and invalidates the cache for the rest of the build.

Let's assume that Docker already has a layer for this instruction in the cache (a cache hit). And let's assume the ID of that layer is BBB.

This process continues for the rest of the Dockerfile.

It's important to understand a few things.

Firstly, as soon as any instruction results in a cache-miss (no layer was found for that instruction), the cache is no longer used for the rest of the entire build. This has an important impact on how you write your Dockerfiles. Try and build them in a way that places any instructions that are likely to change towards the end of the file. This means that a cache-miss will not occur until later stages of the build - allowing the build to benefit as much as possible from the cache.

You can force the build process to ignore the entire cache by passing the `--no-cache=true` flag to the `docker image build` command.

It is also important to understand that the `COPY` and `ADD` instructions include steps to ensure that the content being copied into the image has not changed since the last build. For example, it's possible that the `COPY . /src` instruction in the Dockerfile has not changed since the previous, **but...** the contents of the directory being copied into the image **have** changed!

To protect against this, Docker performs a checksum against each file being copied, and compares that to a checksum of the same file in the cached layer. If the checksums do not match, the cache is invalidated and a new layer is built.

Squash the image

Squashing an image isn't really a best practice as it has pros and cons.

At a high level, Docker follows the normal process to build an image, but then adds an additional step that squashes everything into a single layer.

Squashing can be good in situations where images are starting to have a lot of layers and this isn't ideal. An example might be when creating a new base image that you want to build other images from in the future - this is much better as a single-layer image.

On the negative side, squashed images do not share image layers. This can result in storage inefficiencies and larger push and pull operations.

Add the `--squash` flag to the `docker image build` command if you want to create a squashed image.

Figure 8.8 shows some of the inefficiencies that come with squashed images. Both images are exactly the same except for the fact that one is squashed and the other is not. The squashed image shares layers with other images on the host (saving disk space) but the squashed image does not. The squashed image will also need to send every byte to Docker Hub on a `docker image push` command, whereas the non-squashed image only needs to send unique layers.

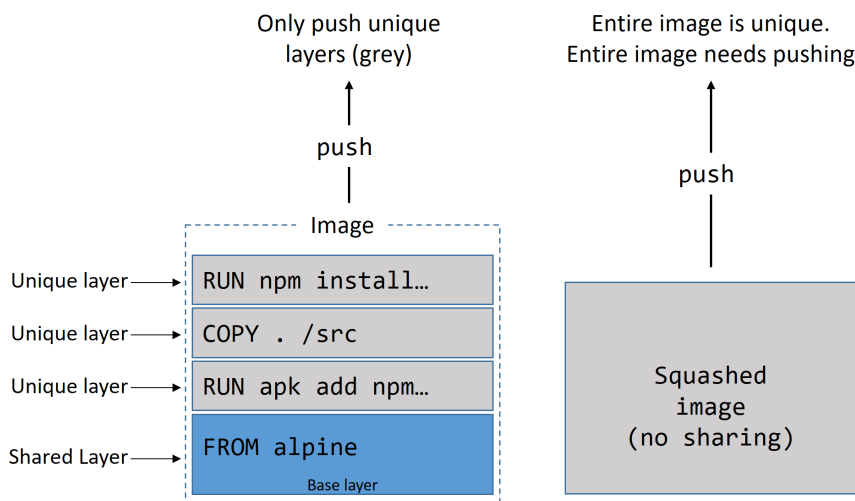


Figure 8.8 - Squashed images vs non-squashed images

Use no-install-recommends

If you are building Linux images, and using the `apt` package manager, you should use the `no-install-recommends` flag with the `apt-get install` command. This makes

sure that `apt` only installs main dependencies (packages in the `Depends` field) and not recommended or suggested packages. This can greatly reduce the number of unwanted packages that are downloaded into your images.

Do not install from MSI packages (Windows)

If you are building Windows images, you should try not to use the MSI package manager. It is not space efficient and results in substantially larger images than are required.

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, and 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 and in an arbitrary location. 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` 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` instruction in a Dockerfile allows you to run commands inside the image which create new layers. Each `RUN` instruction creates a single new layer.
- The `COPY` instruction in a Dockerfile 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` instruction in a Dockerfile documents the network port that the application uses.
- The `ENTRYPOINT` instruction in a Dockerfile 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 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 from 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! With this in mind, treat it like code and check it in and out of a source control system.

Although the example 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: Deploying Apps with Docker Compose

In this chapter, we'll look at how to deploy multi-container applications using Docker Compose.

Docker Compose and Docker Stacks are very similar. In this chapter we'll focus on Docker Compose, which deploys and manages multi-container applications on Docker nodes operating in **single-engine mode**. In a later chapter, we'll focus on Docker Stacks. Stacks deploy and manage multi-container apps on Docker nodes operating in **Swarm mode**.

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

- The TLDR
- The deep dive
- The commands

Deploying apps with Compose - The TLDR

Most modern apps are made of multiple smaller services that interact to form a useful app. We call this microservices. A simple example might be an app with the following four services:

- web front-end
- ordering
- catalog
- back-end database

Put all of these together, and you have a *useful application*.

Deploying and managing lots of services can be hard. This is where *Docker Compose* comes in to play.

Instead of gluing everything together with scripts and long docker commands, Docker Compose lets you describe an entire app in a single declarative configuration file. You then deploy it with a single command.

Once the app is *deployed*, you can *manage* its entire lifecycle with a simple set of commands. You can even store and manage the configuration file in a version control system! It's all very grown-up :-D

That's the basics. Let's dig deeper.

Deploying apps with Compose - The Deep Dive

We'll divide the Deep Dive section as follows:

- Compose background
- Installing Compose
- Compose files
- Deploying an app with Compose
- Managing an app with Compose

Compose background

In the beginning was *Fig*. Fig was a powerful tool, created by a company called *Orchard*, and it was the best way to manage multi-container Docker apps. It was a Python tool that sat on top of Docker, and allowed you to define entire multi-container apps in a single YAML file. You could then deploy the app with the `fig` command-line tool. Fig could even manage the entire life-cycle of the app.

Behind the scenes, Fig would read the YAML file and deploy and manage the app via the Docker API. It was a good thing!

In fact, it was so good, that in 2014, Docker, Inc. acquired Orchard and re-branded Fig as *Docker Compose*. The command-line tool was renamed from `fig` to `docker-compose`, and ever since the acquisition, it's been an external tool that gets bolted on top of the Docker Engine. Even though it's never been fully integrated into the Docker Engine, it's always been immensely popular and very widely used.

As things stand today, Compose is still an external Python binary that you have to install on a host running the Docker Engine. You define multi-container (multi-service) apps in a YAML file, pass the YAML file to the `docker-compose` binary, and Compose deploys it via the Docker Engine API.

Time to see it in action.

Installing Compose

Docker Compose is available on multiple platforms. In this section we'll demonstrate *some* of the ways to install it on Windows, Mac, and Linux. More installation methods exist, but the ones we show here will get you started.

Installing Compose on Windows 10

The recommended way to run Docker on Windows 10 is *Docker for Windows (DfW)*. Docker Compose is included as part of the standard DfW installation. So if you've got DfW, you've got Docker Compose.

Use the following command to check that Compose is installed. You can run this command from a PowerShell or CMD terminal.

```
> docker-compose --version
docker-compose version 1.18.0, build 8dd22a96
```

See **Chapter 3: Installing Docker** if you need more information on installing *Docker for Windows* on Windows 10.

Installing Compose on Mac

As with Windows 10, Docker Compose is installed as part of *Docker for Mac (DfM)*. So if you have DfM, you have Docker Compose.

Run the following command from a terminal window to verify you have Docker Compose.

```
$ docker-compose --version
docker-compose version 1.18.0, build 8dd22a96
```

See **Chapter 3: Installing Docker** if you need more information on installing *Docker for Mac*.

Installing Compose on Windows Server

Docker Compose is installed on Windows Server as a separate binary. To use it, you will need an up-to-date working installation of Docker on your Windows Server.

Type the following command into a PowerShell terminal to install Docker Compose.

For readability, the command uses backticks (‘) to escape carriage returns and wrap the command over multiple lines.

The following commands installs version 1.18.0 of Docker Compose. You can install any version listed here: <https://github.com/docker/compose/releases>. Just replace the 1.18.0 in the URL with the version you want to install.

```
> Invoke-WebRequest ` "https://github.com/docker/compose/releases/download/1\
.18.0/docker-compose-Windows-x86_64.exe" `
-UseBasicParsing `
-OutFile $Env:ProgramFiles\docker\docker-compose.exe
```

```
Writing web request
```

```
Writing request stream... (Number of bytes written: 5260755)
```

Use the `docker-compose --version` command to verify the installation.

```
> docker-compose --version
docker-compose version 1.18.0, build 8dd22a96
```

Compose is now installed. As long as your Windows Server machine has an up-to-date installation of the Docker Engine, you're ready to go.

Installing Compose on Linux

Installing Docker Compose on Linux is a two-step process. First, you download the binary using the `curl` command. Then you make it executable using `chmod`.

For Docker Compose to work on Linux, you'll need a working version of the Docker Engine.

The following command will download version 1.18.0 of Docker Compose and copy it to `/usr/bin/local`. You can check the releases page on [GitHub](https://github.com/docker/compose/releases)²¹ for the latest version and replace the 1.18.0 in the URL with the version you want to install.

The command may wrap over multiple lines in the book. If you run the command on a single line you will need to remove any backslashes (`\`).

```
$ curl -L \
  https://github.com/docker/compose/releases/download/1.18.0/docker-compose-`
uname -s`-`uname -m` \
  -o /usr/local/bin/docker-compose
```

% Total	% Received	Time Total	Time Spent	Time Left	Current Speed
100 617	0 617	0 --:--:--	--:--:--	--:--:--	1047
100 8280k	100 8280k	0 0:00:03	0:00:03	--:--:--	4069k

Now that you've downloaded the `docker-compose` binary, use the following `chmod` command to make it executable.

²¹<https://github.com/docker/compose/releases>

```
$ chmod +x /usr/local/bin/docker-compose
```

Verify the installation and check the version.

```
$ docker-compose --version
docker-compose version 1.18.0, build 8dd22a9
```

You're ready to use Docker Compose on Linux.

You can also use `pip` to install Compose from its Python package. But we don't want to waste pages showing every possible installation method. Enough is enough, time to move on!

Compose files

Compose uses YAML files to define multi-service applications. YAML is a subset of JSON, so you can also use JSON. However, all of the examples in this chapter will be YAML.

The default name for the Compose YAML file is `docker-compose.yml`. However, you can use the `-f` flag to specify custom filenames.

The following example shows a very simple Compose file that defines a small Flask app with two services (`web-fe` and `redis`). The app is a simple web server that counts the number of visits and stores the value in Redis. We'll call the app `counter-app` and use it as the example application for the rest of the chapter.

```
version: "3.5"
services:
  web-fe:
    build: .
    command: python app.py
    ports:
      - target: 5000
        published: 5000
    networks:
      - counter-net
```

```
volumes:
  - type: volume
    source: counter-vol
    target: /code
redis:
  image: "redis:alpine"
  networks:
    counter-net:

networks:
  counter-net:

volumes:
  counter-vol:
```

We'll skip through the basics of the file before taking a closer look.

The first thing to note is that the file has 4 top-level keys:

- `version`
- `services`
- `networks`
- `volumes`

Other top-level keys exist, such as `secrets` and `configs`, but we're not looking at those right now.

The `version` key is mandatory, and it's always the first line at the root of the file. This defines the version of the Compose file format (basically the API). You should normally use the latest version.

It's important to note that the `versions` key does not define the version of Docker Compose or the Docker Engine. For information regarding compatibility between versions of the Docker Engine, Docker Compose, and the Compose file format, google "Compose file versions and upgrading".

For the remainder of this chapter we'll be using version 3 or higher of the Compose file format.

The top-level `services` key is where we define the different application services. The example we're using defines two services; a web front-end called `web-fe`, and an in-memory database called `redis`. Compose will deploy each of these services as its own container.

The top-level `networks` key tells Docker to create new networks. By default, Compose will create bridge networks. These are single-host networks that can only connect containers on the same host. However, you can use the `driver` property to specify different network types.

The following code can be used in your Compose file to create a new *overlay* network called `over-net` that allows standalone containers to connect to it (*attachable*).

```
networks:
  over-net:
    driver: overlay
    attachable: true
```

The top-level `volumes` key is where we tell Docker to create new volumes.

Our specific Compose file

The example file we've listed uses the Compose v3.5 file format, defines two services, defines a network called `counter-net`, and defines a volume called `counter-vol`.

Most of the detail is in the `services` section, so let's take a closer look at that.

The `services` section of our Compose file has two second-level keys:

- `web-fe`
- `redis`

Each of these defines a service in the app. It's important to understand that Compose will deploy each of these as a container, and it will use the name of the keys as part of the container names. In our example, we've defined two keys; `web-fe` and `redis`. This means Compose will deploy two containers, one will have `web-fe` in its name and the other will have `redis`.

Within the definition of the `web-fe` service, we give Docker the following instructions:

- `build:` `.` This tells Docker to build a new image using the instructions in the `Dockerfile` in the current directory (`.`). The newly built image will be used to create the container for this service.
- `command:` `python app.py` This tells Docker to run a Python app called `app.py` as the main app in the container. The `app.py` file must exist in the image, and the image must contain Python. The `Dockerfile` takes care of both of these requirements.
- `ports:` Tells Docker to map port 5000 inside the container (`-target`) to port 5000 on the host (published). This means that traffic sent to the Docker host on port 5000 will be directed to port 5000 on the container. The app inside the container listens on port 5000.
- `networks:` Tells Docker which network to attach the service's container to. The network should already exist, or be defined in the `networks` top-level key. If it's an overlay network, it will need to have the `attachable` flag so that standalone containers can be attached to it (Compose deploys standalone containers instead of Docker Services).
- `volumes:` Tells Docker to mount the `counter-vol` volume (`source:`) to `/code` (`'target:'`) inside the container. The `counter-vol` volume needs to already exist, or be defined in the `volumes` top-level key at the bottom of the file.

In summary, Compose will instruct Docker to deploy a single standalone container for the `web-fe` service. It will be based on an image built from a `Dockerfile` in the same directory as the Compose file. This image will be started as a container and run `app.py` as its main app. It will expose itself on port 5000 on the host, attach to the `counter-net` network, and mount a volume to `/code`.

Note: Technically speaking, we don't need the `command: python app.py` option. This is because the application's `Dockerfile` already defines `python app.py` as the default app for the image. However, we're showing it here so you know how it works. You can also use it to override CMD instructions set in `Dockerfiles`.

The definition of the `redis` service is simpler:

- `image: redis:alpine` This tells Docker to start a standalone container called `redis` based on the `redis:alpine` image. This image will be pulled from Docker Hub.
- `networks:` The `redis` container will be attached to the `counter-net` network.

As both services will be deployed onto the same `counter-net` network, they will be able to resolve each other by name. This is important as the application is configured to communicate with the `redis` service by name.

Now that we understand how the Compose file works, let's deploy it!

Deploying an app with Compose

In this section, we'll deploy the app defined in the Compose file from the previous section. To do this, you'll need the following 4 files from <https://github.com/nigelpoulton/counter-app>:

- `Dockerfile`
- `app.py`
- `requirements.txt`
- `docker-compose.yml`

Clone the Git repo locally.

```
$ git clone https://github.com/nigelpoulton/counter-app.git
```

```
Cloning into 'counter-app' ...
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 9 (delta 1), reused 5 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.
Checking connectivity... done.
```

Cloning the repo will create a new sub-directory called `counter-app`. This will contain all of the required files and will be considered your *build context*. Compose will also use the name of the directory (`counter-app`) as your project name. We'll see this later, but Compose will pre-pend all resource names with `counter-app_`.

Change into the `counter-app` directory and check the files are present.

```
$ cd counter-app
$ ls
app.py  docker-compose.yml  Dockerfile  requirements.txt ...
```

Let's quickly describe each file:

- `app.py` is the application code (a Python Flask app)
- `docker-compose.yml` is the Docker Compose file that describes how Docker should deploy the app
- `Dockerfile` describes how to build the image for the web-fe service
- `requirements.txt` lists the Python packages required for the app

Feel free to inspect the contents of each file.

The `app.py` file is obviously the core of the application. But `docker-compose.yml` is the glue that sticks all the app components together.

Let's use Compose to bring the app up. You must run the all of the following commands from within the `counter-app` directory that you just cloned from GitHub.

```
$ docker-compose up &
```

```
[1] 1635
Creating network "counterapp_counter-net" with the default driver
Creating volume "counterapp_counter-vol" with default driver
Pulling redis (redis:alpine)...
alpine: Pulling from library/redis
1160f4abea84: Pull complete
a8c53d69ca3a: Pull complete
<Snip>
web-fe_1 | * Debugger PIN: 313-791-729
```

It'll take a few seconds for the app to come up, and the output can be quite verbose.

We'll step through what happened in a second, but first let's talk about the `docker-compose` command.

`docker-compose up` is the most common way to bring up a Compose app (we're calling a multi-container app defined in a Compose file a *Compose app*). It builds all required images, creates all required networks and volumes, and starts all required containers.

By default, `docker-compose up` expects the name of the Compose file to `docker-compose.yml` or `docker-compose.yaml`. If your Compose file has a different name, you need to specify it with the `-f` flag. The following example will deploy an application from a Compose file called `prod-equus-bass.yml`

```
$ docker-compose -f prod-equus-bass.yml up
```

It's also common to use the `-d` flag to bring the app up in the background. For example:

```
docker-compose up -d
```

--OR--

```
docker-compose -f prod-equus-bass.yml up -d
```

Our example brought the app up in the foreground (we didn't use the `-d` flag), but we used the `&` to give us the terminal window back. This is not normal, but it will output logs directly in our terminal window which we'll use later.

Now that the app is built and running, we can use normal `docker` commands to view the images, containers, networks, and volumes that Compose created.

```
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
counterapp_web-fe	latest	96..6ff9e	3 minutes ago	95.9MB
python	3.4-alpine	01..17a02	2 weeks ago	85.5MB
redis	alpine	ed..c83de	5 weeks ago	26.9MB

We can see that three images were either built or pulled as part of the deployment.

The `counterapp_web-fe:latest` image was created by the `build: .` instruction in the `docker-compose.yml` file. This instruction caused Docker to build a new image using the `Dockerfile` in the same directory. It contains the application code for the Python Flask web app, and was built from the `python:3.4-alpine` image. See the contents of the `Dockerfile` for more information.

```
FROM python:3.4-alpine          << Base image
ADD . /code                     << Copy app into image
WORKDIR /code                   << Set working directory
RUN pip install -r requirements.txt << install requirements
CMD ["python", "app.py"]        << Set the default app
```

I've added comments to the end of each line to help explain. They must be removed before deploying the app.

Notice how Compose has named the newly built image as a combination of the project name (`counter-app`), and the resource name as specified in the Compose file (`web-fe`). Compose has removed the dash (`-`) from the project name. All resources deployed by Compose will follow this naming convention.

The `redis:alpine` image was pulled from Docker Hub by the `image: "redis:alpine"` instruction in the `.Services.redis` section of the Compose file.

The following container listing shows two containers. The name of each is prefixed with the name of the project (name of the working directory). Also, each one has a numeric suffix that indicates the instance number — this is because Compose allows for scaling.

```
$ docker container ls
```

ID	COMMAND	STATUS	PORTS	NAMES
12..	"python app.py"	Up 2 min	0.0.0.0:5000->5000/tcp	counterapp_web-fe_1
57..	"docker-entry..."	Up 2 min	6379/tcp	counterapp_redis_1

The `counterapp_web-fe` container is running the application's web front end. This is running the `app.py` code and is mapped to port `5000` on all interfaces on the Docker host. We'll connect to this in just a second.

The following network and volume listings show the `counterapp_counter-net` and `counterapp_counter-vol` networks and volumes.

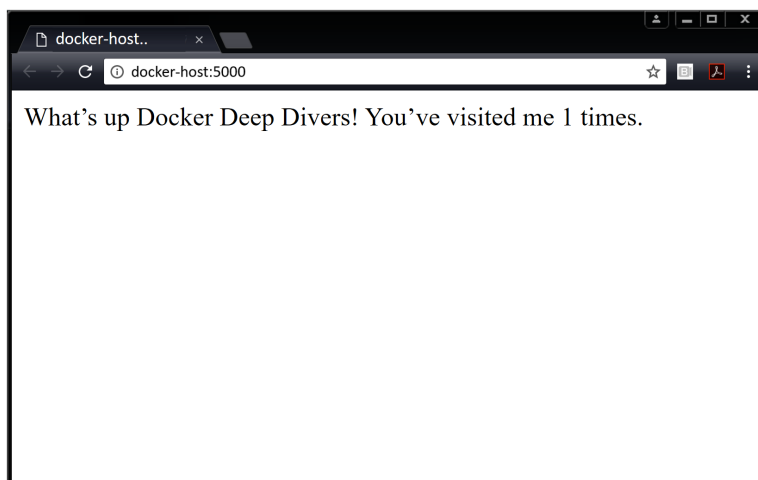
9: Deploying Apps with Docker Compose

168

```
$ docker network ls
NETWORK ID          NAME                                DRIVER  SCOPE
1bd949995471        bridge                             bridge  local
40df784e00fe        counterapp_counter-net            bridge  local
f2199f3cf275        host                               host    local
67c31a035a3c        none                               null    local

$ docker volume ls
DRIVER      VOLUME NAME
<Snip>
local      counterapp_counter-vol
```

With the application successfully deployed, you can point a web browser at your Docker host on port 5000 and see the application in all its glory.



Pretty impressive ;-)

Hitting your browser's refresh button will cause the counter to increment. Feel free to inspect the app (app.py) to see how the counter data is stored in the Redis back-end.

If you brought the application up using the &, you will be able to see the HTTP 200 response codes being logged in the terminal window. These indicate successful requests, and you'll see one for each time you load the web page.

```
web-fe_1 | 172.18.0.1 - - [09/Jan/2018 11:13:21] "GET / HTTP/1.1" 200 -
web-fe_1 | 172.18.0.1 - - [09/Jan/2018 11:13:33] "GET / HTTP/1.1" 200 -
```

Congratulations. You've successfully deployed a multi-container application using Docker Compose!

Managing an app with Compose

In this section, we'll see how to start, stop, delete, and get the status of applications being managed by Docker Compose. We'll also see how the volume we're using can be used to directly inject updates to the app's web front-end.

As the application is already up, let's see how to bring it down. To do this, replace the up sub-command with down.

```
$ docker-compose down
1. Stopping counterapp_redis_1 ...
2. Stopping counterapp_web-fe_1 ...
3. redis_1 | 1:signal-handler Received SIGTERM scheduling shutdown...
4. redis_1 | 1:M 09 Jan 11:16:00.456 # User requested shutdown...
5. redis_1 | 1:M 09 Jan 11:16:00.456 * Saving the final RDB snap...
6. redis_1 | 1:M 09 Jan 11:16:00.463 * DB saved on disk
7. Stopping counterapp_redis_1 ... done
8. counterapp_redis_1 exited with code 0
9. Stopping counterapp_web-fe_1 ... done
10. Removing counterapp_redis_1 ... done
11. Removing counterapp_web-fe_1 ... done
12. Removing network counterapp_counter-net
13. [1]+ Done docker-compose up
```

Because we started the app with the `&`, it's running in the foreground. This means we get a verbose output to the terminal, giving us an excellent insight into how things work. Let's step through what each line is telling us.

Lines 1 and 2 are stopping the two services. These are the web-fe and redis services defined in the Compose file.

Line 3 shows that the stop instruction sends a SIGTERM signal. This is sent to the PID 1 process in each container. Lines 4-6 show the Redis container gracefully handling the signal and shutting itself down. Lines 7 and 8 report the success of stop operation.

Line 9 shows the web-fe service successfully stopping.

Lines 10 and 11 show the stopped services being removed.

Line 12 shows the counter-net network being removed, and line 13 shows the docker-compose up process exiting.

It's important to note that the counter-vol volume was **not** deleted. This is because volumes are intended to be long-term persistent data stores. As such, their lifecycle is entirely decoupled from the containers they serve. Running a `docker volume ls` will show that the volume is still present on the system. If you'd written any data to the volume it would still exist.

Also, any images that were built or pulled as part of the docker-compose up operation will still remain on the system. This means future deployments of the app will be faster.

Let's look at a few other docker-compose sub-commands.

Use the following command to bring the app up again, but this time in the background.

```
$ docker-compose up -d
Creating network "counterapp_counter-net" with the default driver
Creating counterapp_redis_1 ... done
Creating counterapp_web-fe_1 ... done
```

See how the app started much faster this time — the counter-vol volume already exists, and no images needed building or pulling.

Show the current state of the app with the `docker-compose ps` command.

```
$ docker-compose ps
```

Name	Command	State	Ports
counterapp_redis_1	docker-entrypoint...	Up	6379/tcp
counterapp_web-fe_1	python app.py	Up	0.0.0.0:5000->5000/tcp

We can see both containers, the commands they are running, their current state, and the network ports they are listening on.

Use the `docker-compose top` command to list the processes running inside of each service (container).

```
$ docker-compose top
```

```
counterapp_redis_1
```

PID	USER	TIME	COMMAND
843	dockrema	0:00	redis-server

```
counterapp_web-fe_1
```

PID	USER	TIME	COMMAND
928	root	0:00	python app.py
1016	root	0:00	/usr/local/bin/python app.py

The PID numbers returned are the PID numbers as seen from the Docker host (not from within the containers).

Use the `docker-compose stop` command to stop the app without deleting its resources. Then show the status of the app with `docker-compose ps`.

9: Deploying Apps with Docker Compose

172

```
$ docker-compose stop
Stopping counterapp_web-fe_1 ... done
Stopping counterapp_redis_1 ... done
```

```
$ docker-compose ps
```

Name	Command	State
counterapp_redis_1	docker-entrypoint.sh redis	Exit 0
counterapp_web-fe_1	python app.py	Exit 0

As we can see, stopping a Compose app does not remove the application definition from the system. It just stops the app's containers. You can verify this with the `docker container ls -a` command.

You can delete a stopped Compose app with the `docker-compose rm` command. This will delete the containers and networks the app is using, but it will not delete volumes or images. Nor will it delete the application source code (`app.py`, `Dockerfile`, `requirements.txt`, and `docker-compose.yml`) in your project directory.

Restart the app with the `docker-compose restart` command.

```
$ docker-compose restart
Restarting counterapp_web-fe_1 ... done
Restarting counterapp_redis_1 ... done
```

Verify the operation.

```
$ docker-compose ps
```

Name	Command	State	Ports
counterapp_redis_1	docker-entrypoint...	Up	6379/tcp
counterapp_web-fe_1	python app.py	Up	0.0.0.0:5000->5000/tcp

Use the `docker-compose down` command to **stop and delete** the app with a single command.

```
$ docker-compose down
Stopping counterapp_web-fe_1 ... done
Stopping counterapp_redis_1 ... done
Removing counterapp_web-fe_1 ... done
Removing counterapp_redis_1 ... done
Removing network counterapp_counter-net
```

The app is now deleted. Only its images, volumes and source code remain.

Let's deploy the app one last time and see its volume in action.

```
$ docker compose up -d
Creating network "counterapp_counter-net" with the default driver
Creating counterapp_redis_1 ... done
Creating counterapp_web-fe_1 ... done
```

If you look in the Compose file, you'll see that we're defining a new volume called `counter-vol` and mounting it in to the `web-fe` service at `/code`.

```
services:
  web-fe:
    <Snip>
    volumes:
      - type: volume
        source: counter-vol
        target: /code
    <Snip>
volumes:
  counter-vol:
```

The first time we deployed the app, Compose checked to see if a volume already existed with this name. It did not, so it created it. You can see it with the `docker volume ls` command.

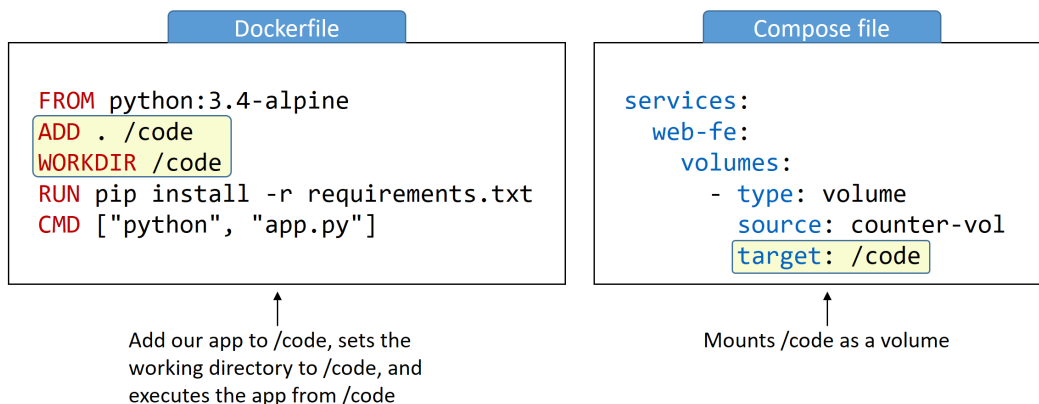
```
$ docker volume ls
RIVER          VOLUME NAME
local          counterapp_counter-vol
```

It's also worth knowing that Compose builds networks and volumes **before** deploying services. This makes sense, as they are lower-level infrastructure objects that are consumed by services (containers). The following snippet shows Compose creating the network and volume as its first two tasks (even before building and pulling images).

```
$ docker-compose up -d
```

```
Creating network "counterapp_counter-net" with the default driver
Creating volume "counterapp_counter-vol" with default driver
Pulling redis (redis:alpine)...
<Snip>
```

If we take another look at the service definition for `web-fe`, we'll see that it's mounting the counter-app volume into the service's container at `/code`. We can also see from the Dockerfile that `/code` is where the app is installed and executed from. Net result, our app code resides on a Docker volume.



This all means we can make changes to files in the volume, from the host side, and have them reflected immediately in the app. Let's see it.

The next few steps will walk through the following process. We'll edit the `app.py` file in the project's working directory so that the app will display different text in the web browser. We'll copy updated app to the volume on the Docker host. We'll refresh the app's web page to see the updated text. This will work because whatever you write to the location of the volume on the Docker host will immediately appear in the volume in the container.

Use your favourite text editor to edit the `app.py` file in the project's working directory. We'll use `vim` in the example.

```
$ vim ~/counter-app/app.py
```

Change text between the double quote marks ("") on line 22. The line starts with `return "What's up..."`. Enter any text you like, as long as it's within the double-quote marks, and save your changes.

Now that we've updated the app, we need to copy it into the volume on the Docker host. Each Docker volume is exposed at a location within the Docker host's filesystem, as well as a mount point in one or more containers. Use the following `docker volume inspect` command to find where the volume is exposed on the Docker host.

```
$ docker volume inspect counterapp_counter-vol | grep Mount
```

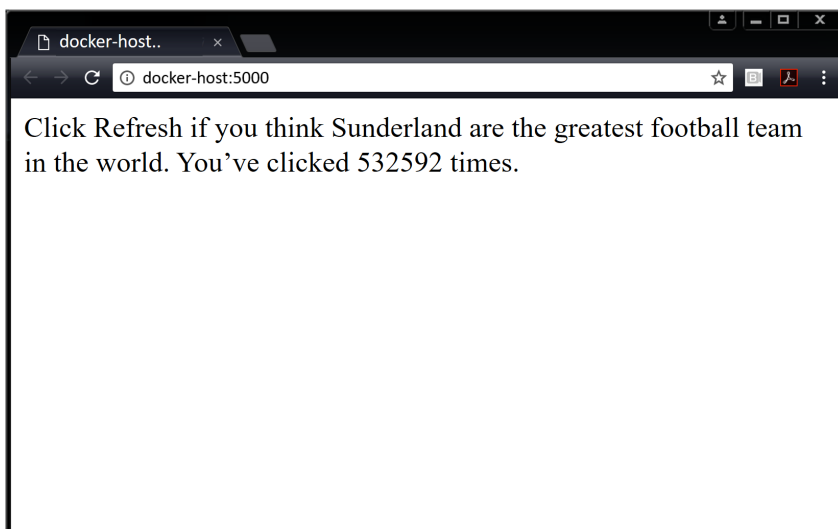
```
"Mountpoint": "/var/lib/docker/volumes/counterapp_counter-vol/_data",
```

Copy the updated app file to the volume's mount point on your Docker host. This will make it appear in the web-fe container at `/code`. The operation will overwrite the existing `/code/app.py` file in the container.

```
$ cp ~/counterapp/app.py \
  /var/lib/docker/volumes/counterapp_counter-vol/_data/app.py
```

The updated app file is now on the container. Connect to the app to see your change. You can do this by pointing your web browser to the IP of your Docker host on port 5000.

Figure 9.3 shows the updated app.



Obviously you wouldn't do this in production, but it's a real time-saver in development.

Congratulations! You've deployed and managed a simple multi-container app using Docker Compose.

Before reminding ourselves of the major commands we learned, it's important to understand that this was a very simple example. Docker Compose is capable of deploying and managing far more complex applications.

Deploying apps with Compose - The commands

- `docker-compose up` is the command we use to deploy a Compose app. It expects the Compose file to be called `docker-compose.yml` or `docker-compose.yaml`, but you can specify a custom filename with the `-f` flag. It's common to start the app in the background with the `-d` flag.
- `docker-compose stop` will stop all of the containers in a Compose app without

deleting them from the system. The app can be easily restarted with `docker-compose restart`.

- `docker-compose rm` will delete a stopped Compose app. It will delete containers and networks, but it will not delete volumes and images.
- `docker-compose restart` will restart a Compose app that has been stopped with `docker-compose stop`. If you have made changes to your Compose app since stopping it, these changes will **not** appear in the restarted app. You will need to re-deploy the app to get the changes.
- `docker-compose ps` will list each container in the Compose app. It shows current state, the command each one is running, and network ports.
- `docker-compose down` will stop and delete a running Compose app. It deletes containers and networks, but not volumes and images.

Chapter Summary

In this chapter, we learned how to deploy and manage a multi-container application using Docker Compose.

Docker Compose is a Python application that we install on top of the Docker Engine. It lets us define multi-container apps in a single declarative configuration file and deploy it with a single command.

Compose files can be YAML or JSON, and they define all of the containers, networks, volumes, and secrets that an application requires. We then feed the file to the `docker-compose` command line tool, and Compose instructs Docker to deploy it.

Once the app is deployed, we can manage its entire lifecycle using the many `docker-compose` sub-commands.

We also saw how volumes can be used to mount changes directly into containers.

Docker Compose is very popular with developers, and the Compose file is an excellent source of application documentation — it defines all the services that make up the app, the images they use, ports they expose, networks and volumes they use, and much more. As such, it can help bridge the gap between dev and ops. You should also treat your Compose files as if they were code. This means, among other things, storing them in source control repos.

10: Docker Swarm

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 things at scale. That's where Docker Swarm comes into the picture.

At a high level Swarm has two major components:

- A secure cluster
- An orchestration engine

As usual, we'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

The examples and outputs we'll use will be from a Linux-based swarm. However, most commands and features work with Docker on Windows.

Docker Swarm - The TLDR

Docker Swarm is two things: an enterprise-grade secure cluster of Docker hosts, and an engine for orchestrating microservices apps.

On the clustering front, it groups one or more Docker nodes and lets you manage them as a cluster. Out-of-the-box you get an encrypted distributed cluster store, encrypted networks, mutual TLS, secure cluster join tokens, and a PKI that makes managing and rotating certificates a breeze! And you can non-disruptively add and remove nodes. It's a beautiful thing!

On the orchestration front, swarm exposes a rich API that allows you to deploy and manage complicated microservices apps with ease. You can define your apps in declarative manifest files, and deploy them with native Docker commands. You can even perform rolling updates, rollbacks, and scaling operations. Again, all with simple commands.

In the past, Docker Swarm was a separate product that you layered on top of the Docker engine. Since Docker 1.12 it's fully integrated into the Docker engine and can be enabled with a single command. As of 2018, it has the ability to deploy and manage native swarm apps as well as Kubernetes apps. Though at the time of writing, support for Kubernetes apps is relatively new.

Docker Swarm - The Deep Dive

We'll split the deep dive part of this chapter as follows:

- Swarm primer
- Build a secure swarm cluster
- Deploy some swarm services
- Troubleshooting

The examples cited will be based on Linux, but they will also work on Windows. Where there are differences we'll be sure to point them out.

Swarm mode primer

On the clustering front, a *swarm* consists of one or more Docker *nodes*. These can be physical servers, VMs, Raspberry Pi's, or cloud instances. The only requirement is that all nodes can communicate over reliable networks.

Nodes are configured as *managers* or *workers*. *Managers* look after the control plane of the cluster, meaning things like the state of the cluster and dispatching tasks to *workers*. *Workers* accept tasks from *managers* and execute them.

The configuration and state of the *swarm* is held in a distributed *etcd* database located on all managers. It's kept in memory and is extremely up-to-date. 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 that's game changing on the clustering front is the approach to security. TLS is so tightly integrated that it's impossible to build a swarm without it. In today's security conscious world, things like this deserve all the props they get! Anyway, *swarm* 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!

On the application orchestration front, the atomic unit of scheduling on a swarm is the *service*. This is a new object in the API, introduced along with *swarm*, and is a higher level construct that wraps some advanced features around containers.

When a container is wrapped in a service we call it a *task* or a *replica*, and the service construct adds things like scaling, rolling updates, and simple rollbacks.

The high-level view is shown in Figure 10.1.

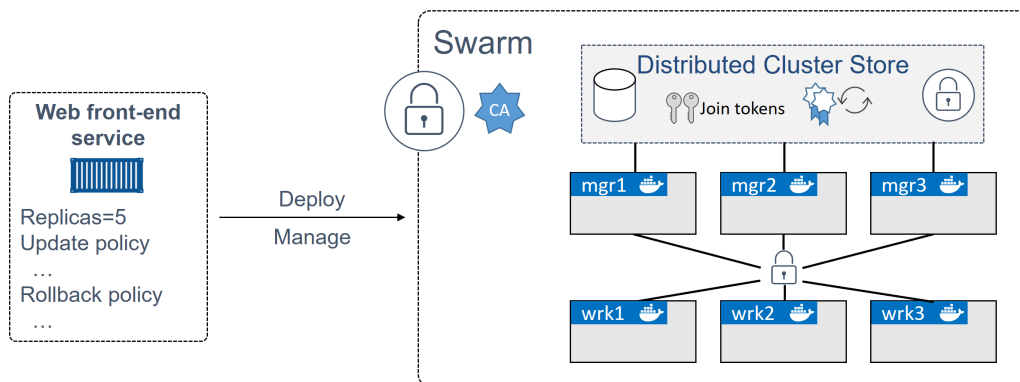


Figure 10.1 High-level swarm

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

Build a secure Swarm cluster

In this section we'll build a secure swarm cluster with three *manager nodes* and three *worker nodes*. You can use a different lab with different numbers of *managers* and

workers, and with different names and IPs, but the examples that follow will use the values in Figure 10.2.

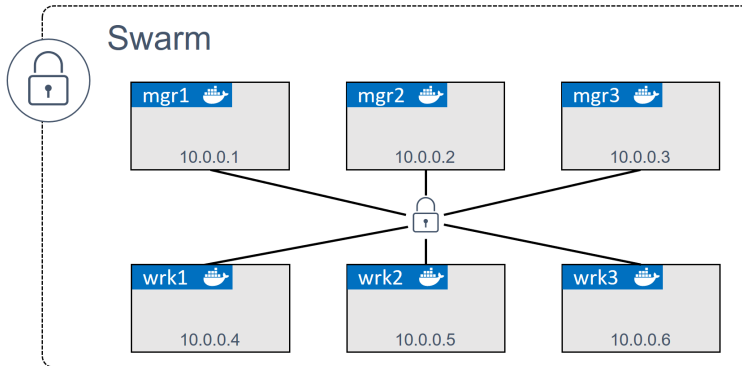


Figure 10.2

Each of the nodes needs Docker installed and needs to be able to communicate with the rest of the swarm. It's also beneficial if name resolution is configured — it makes it easier to identify nodes in command outputs and helps when troubleshooting.

On the networking front, you should have the following ports open on routers and firewalls:

- 2377/tcp: for secure client-to-swarm communication
- 7946/tcp and 7946/udp: for control plane gossip
- 4789/udp: for VXLAN-based overlay networks

Once you've satisfied the pre-requisites, you can go ahead and build a swarm.

The process of building a swarm is sometimes called *initializing a swarm*, and the high-level process is this: Initialize the first manager node > Join additional manager nodes > Join worker nodes > Done.

Initializing a brand new swarm

Docker nodes that are not part of a swarm are said to be in *single-engine mode*. Once they're added to a swarm they're switched into *swarm mode*.

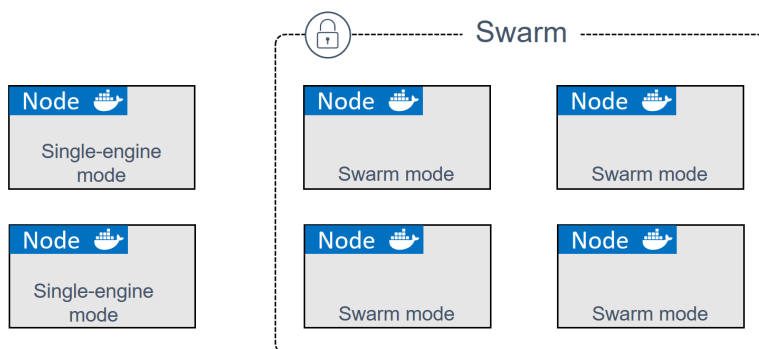


Figure 10.3 Swarm mode vs single-engine mode

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

Additional nodes can then be *joined* as workers and managers. This obviously switches them 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 in *swarm mode* and operating as part of the same swarm.

This example will use the IP addresses and DNS names of the nodes shown in Figure 10.2. Yours may be different.

1. Log on to **mgr1** and initialize a new swarm (don't forget to use backticks instead of backslashes if you're 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. It's an optional flag, 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.
- `--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.

The default port that swarm mode operates on is **2377**. This is customizable, but it's convention to use `2377/tcp` for secured (HTTPS) client-to-swarm connections.

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 **mgr1** 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 ensure that your join tokens are protected, as they 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 worker join token.

```
$ 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.

The `--advertise-addr`, and `--listen-addr` flags optional. I've added them 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** so that they join 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...15wt6	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 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 a much bigger pain if you get the network configuration of your swarm wrong. Also, 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 them.

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

Swarm manager high availability (HA)

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

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

Technically speaking, swarm implements a form of active-passive multi-manager HA. This means that although you might — and should — have multiple *managers*, only one of them is ever considered *active*. We call this active manager the “*leader*”, and the leader's the only one that will ever issue live commands against the *swarm*. So it's only ever the leader that changes the config, or issues tasks to workers. If a passive (non-active) manager receives commands for the swarm, it proxies them across to the leader.

This process is shown in Figure 10.4. 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 executing the command on the swarm.

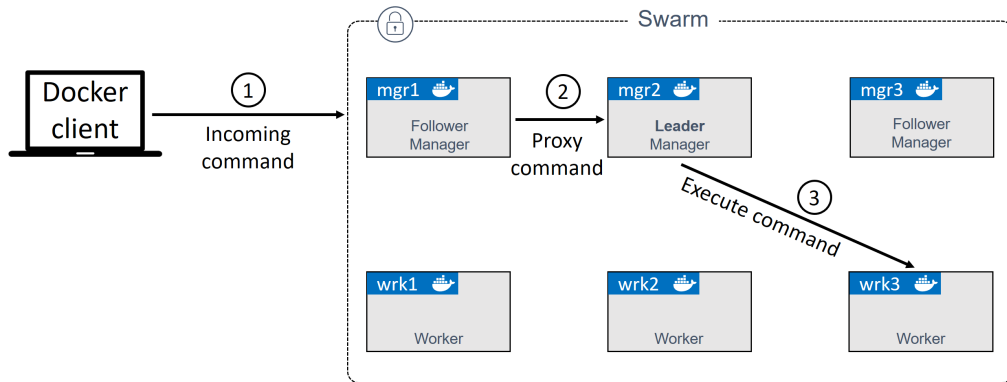


Figure 10.4

If you look closely at Figure 10.4 you'll notice that managers are either *leaders* or *followers*. This is Raft terminology, because swarm uses an implementation of the [Raft consensus algorithm](https://raft.github.io/)²² to power manager HA. And on the topic of HA, 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* reduces the chances of split-brain conditions. 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. But crucially, neither side has any way of knowing if the other two are still alive and whether it holds a majority (quorum). The cluster continues to operate during split-brain conditions, but you are no longer able to alter the configuration or add and manage application workloads.

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 partition. This means that one side achieve quorum and cluster management would remain available. The example on the right side of Figure 10.5 shows a partitioned cluster where the left side of the split knows it has a majority of managers.

²²<https://raft.github.io/>

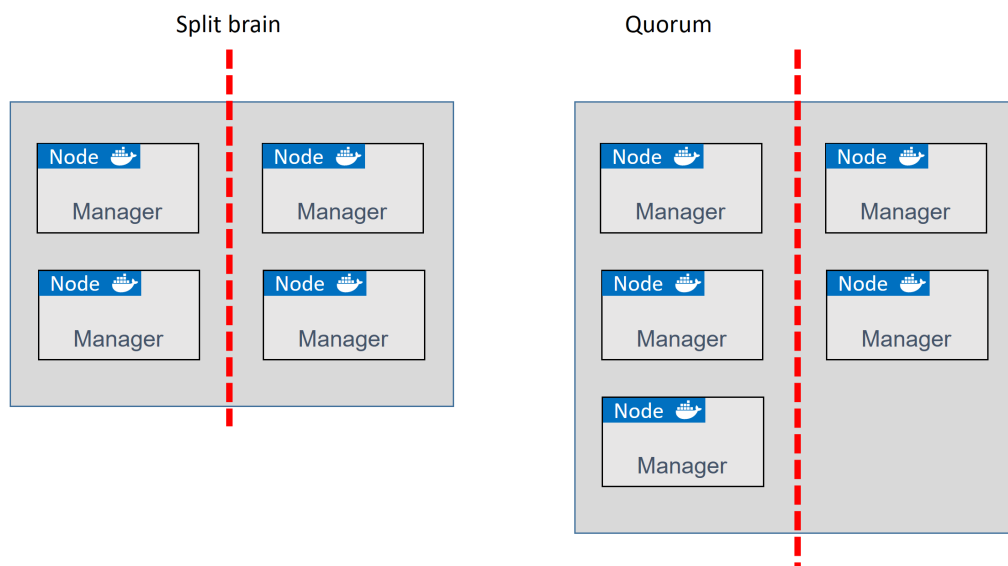


Figure 10.5

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 to decide with 3 people than it is with 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 reliable high-speed networks!

Built-in Swarm security

Swarm clusters have a ton of built-in security that's configured out-of-the-box with sensible defaults — CA settings, join tokens, mutual TLS, encrypted cluster store,

encrypted networks, cryptographic node ID's and more. See **Chapter 15: Security in Docker** for a detailed look at these.

Locking a Swarm

Despite all of this built-in native security, restarting an older manager or restoring an old backup has the potential to compromise the cluster. Old managers re-joining a swarm automatically decrypt and gain access to the Raft log time-series database — this can pose security concerns. Restoring old backups can wipe the current swarm configuration.

To prevent situations like these, Docker allows you to lock a swarm with the Autolock feature. This forces managers that have been restarted to present the cluster unlock key before being permitted back into the cluster.

It's possible to apply a lock directly to a new swarm you are creating by passing the `--autolock` flag to the `docker swarm init` command. However, we've already built a swarm, so we'll lock our existing swarm with the `docker swarm update` command.

Run the following command from a swarm manager.

```
$ docker swarm update --autolock=true
```

```
Swarm updated.
```

To unlock a swarm manager after it restarts, run the `docker swarm unlock` command and provide the following key:

```
SWMKEY-1-5+ICW2kRxPxZrVyBDWzBkzZdSd0Yc7C12o4Uuf9NPU4
```

Please remember to store this key in a password manager, since without it you will not be able to restart the manager.

Be sure to keep the unlock key in a secure place!

Restart one of your manager nodes to see if it automatically re-joins the cluster. You may need to prepend the command with `sudo`.

```
$ service docker restart
```

Try and list the nodes in the swarm.

```
$ docker node ls
```

```
Error response from daemon: Swarm is encrypted and needs to be unlocked before  
it can be used.
```

Although the Docker service has restarted on the manager, it has not been allowed to re-join the cluster. You can prove this even further by running the `docker node ls` command on another manager node. The restarted manager will show as down and unreachable.

Use the `docker swarm unlock` command to unlock the swarm for the restarted manager. You'll need to run this command on the restarted manager, and you'll need to provide the unlock key.

```
$ docker swarm unlock
```

```
Please enter unlock key: <enter your key>
```

The node will be allowed to re-join the swarm, and will show as ready and reachable if you run another `docker node ls`.

Locking your swarm and protecting the unlock key is recommended for production environments.

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

Swarm services

Everything we do in this section of the chapter gets improved on by Docker Stacks (Chapter 14). However, it's important that you learn the concepts here so that you're prepared for Chapter 14.

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

They let us specify most of the familiar container options, such as *name*, *port mappings*, *attaching to networks*, and *images*. But they add things, like letting us declare the *desired state* for an application service, feed that to Docker, and let Docker

take care of deploying it and managing it. For example, assume you've got an app with a web front-end. You have an image for it, and testing has shown that you'll need 5 instances to handle normal daily traffic. You would translate this requirement into a single *service* declaring the image the containers should use, and that the service should always have 5 running replicas.

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.

You create a new 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 this example 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 from a PowerShell terminal you will need to use the backtick (`) to indicate continuation on the next line.

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

```
z7ovearqmrwk0u2vc5o7q10p
```

Notice that many of the familiar `docker container run` arguments are the same. In the example, we specified `--name` and `-p` which work the same for standalone containers as well as services.

Let's review the 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 it **web-fe**. We told Docker to map port 8080 on every node in the swarm to 8080 inside of each service replica. Next, we used the `--replicas` flag to tell Docker that there should always be 5 replicas of this service. Finally, we told Docker which image to use for the replicas — it's important to understand that all service replicas use the same image and config!

After we hit Return, the manager acting as leader instantiated 5 replicas across the *swarm* — remember that swarm 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 stored on the cluster and 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 background *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, 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 a *worker* hosting one of the 5 **web-fe** replicas fails, the *actual state* for the **web-fe** service will drop from 5 replicas to 4. This will no longer match the *desired state* of 5, so Docker will start a new **web-fe** replica 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/tcp
```

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 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 often due to the time it takes to pull the image on each node.

You can use the `docker service ps` command to see a list of service replicas and the state of each.

```
$ 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 replica (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. I highly recommend you read through the output of `docker inspect` commands as they're a great source of information and a great way to learn what's going on under the hood.

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

Replicated vs global services

The default replication mode of a service is `replicated`. This will deploy a desired number of replicas and distribute them as evenly as possible across the cluster.

The other mode is `global`, which runs a single replica on every node in the swarm.

To deploy a *global service* you need to pass the `--mode global` flag to the `docker service create` command.

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 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
```

This command will scale the number of service 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 service replicas are balanced across all nodes in the swarm evenly.

```
$ 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...g6f	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
17k...jkv	web-fe.10	nigelpoulton/...	mgr2	Running	Running 46 seconds ago

Behind the scenes, swarm runs a scheduling algorithm that defaults to balancing replicas as evenly as possible across the nodes in the swarm. At the time of writing, this amounts to running an equal number of replicas 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 deployed earlier.

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

Confirm it's 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 service replicas without asking for confirmation.

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

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 doing it again.

Well... thanks to Docker *services*, pushing updates to well-designed apps just got a 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 to attach the service to 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 creates a new layer 2 network that we can place containers on, and all containers on it will be able to communicate. This works even if the Docker hosts the containers are 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 10.6 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 even if they are on Docker hosts plumbed into different underlay networks.

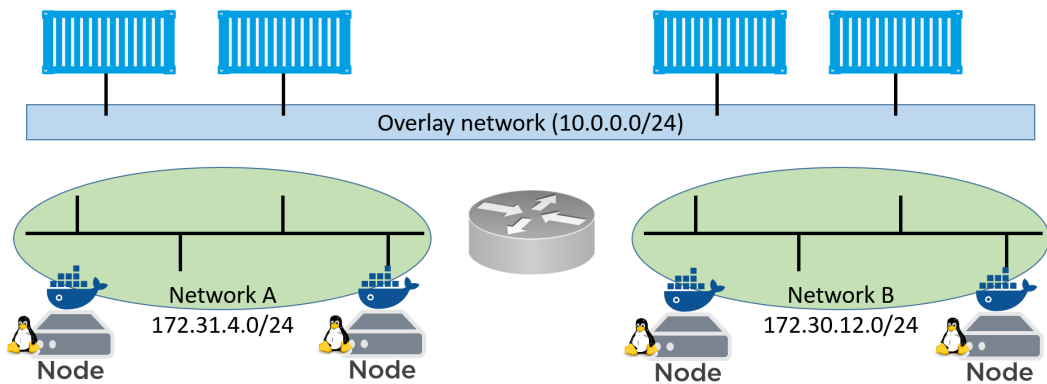


Figure 10.6

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
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 create a new service and attach it to the network.

```
$ 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 replicas 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 we asked it to run. Finally, we told it to base all replicas 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 all traffic, coming in to any node in the swarm on port 80, through to port 80 inside of any service replica.

This mode of publishing a port on every node in the swarm — even nodes not running service replicas — is called *ingress mode* and is the default. The alternative mode is *host mode* which only publishes the service on swarm nodes running replicas. Publishing a service in *host mode* requires the long-form syntax and looks like the following:

```
docker service create --name uber-svc \  
  --network uber-net \  
  --publish published=80,target=80,mode=host \  
  --replicas 12 \  
  nigelpoulton/tu-demo:v1
```

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 service running.

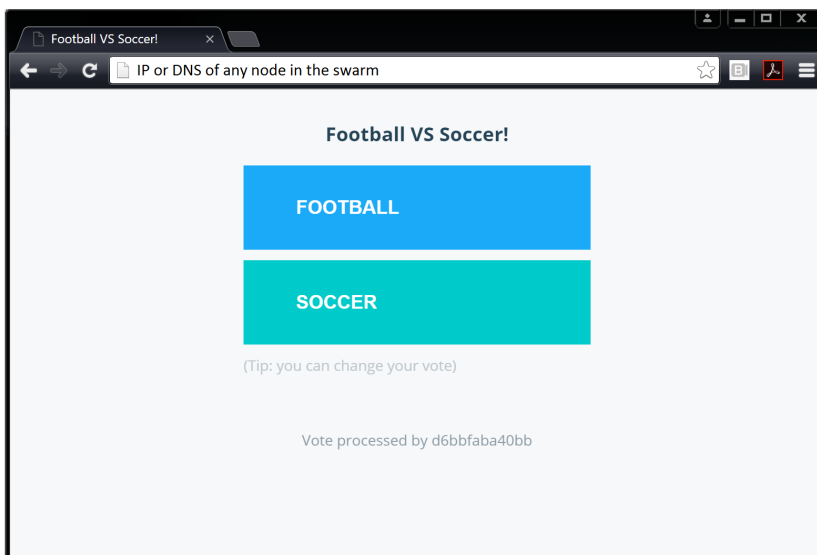


Figure 10.7

As you can see, it's 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'll be able to reach the web service from any node because the `-p 80:80` flag creates an *ingress mode* mapping on every swarm node. This is true even on

nodes that are not running a replica 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 wants to run 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 replicas at a time with a 20 second delay between each. 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
```

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 `--update-delay` flags to make sure that the new image was pushed to 2 replicas at a time with a 20 second cool-off period in between each. 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 replicas are at v2 while some are still at v1. If we give the operation enough time to complete (4 minutes) all replicas 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 replicas running the old version and some will be serviced by replicas running the new version. After enough time, all requests will be serviced by replicas running the updated version of the service.

Congratulations. You've just pushed a rolling update to a live containerized application. Remember, Docker Stacks take all of this to the next level in Chapter 14.

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

10: Docker Swarm

202

```
$ 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 replica 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 replica.

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

Troubleshooting

Swarm Service logs can be viewed with the `docker service logs` command. However, not all logging drivers support the command.

By default, Docker nodes configure services to use the `json-file` log driver, but other drivers exist, including:

- `journald` (only works on Linux hosts running `systemd`)
- `syslog`
- `splunk`
- `gelf`

`json-file` and `journald` are the easiest to configure, and both work with the `docker service logs` command. The format of the command is `docker service logs <service-name>`.

If you're using 3rd-party logging drivers you should view those logs using the logging platform's native tools.

The following snippet from a `daemon.json` configuration file shows a Docker host configured to use `syslog`.

```
{
  "log-driver": "syslog"
}
```

You can force individual services to use a different driver by passing the `--log-driver` and `--log-opts` flags to the `docker service create` command. These will override anything set in `daemon.json`.

Service logs work on the premise that your application is running as PID 1 in its container and sending logs to `STDOUT`, and errors to `STDERR`. The logging driver forwards these “logs” to the locations configured via the logging driver.

The following `docker service logs` command shows the logs for all replicas in the `svc1` service that experienced a couple of failures starting a replica.

```
$ docker service logs seastack_reverse_proxy
svc1.1.zhc3cjeti9d4@wrk-2 | [emerg] 1#1: host not found...
svc1.1.6m1nmbzmwh2d@wrk-2 | [emerg] 1#1: host not found...
svc1.1.6m1nmbzmwh2d@wrk-2 | nginx: [emerg] host not found..
svc1.1.zhc3cjeti9d4@wrk-2 | nginx: [emerg] host not found..
svc1.1.1tmya243m5um@mgr-1 | 10.255.0.2 "GET / HTTP/1.1" 302
```

The output is trimmed to fit the page, but you can see that logs from all three service replicas are shown (the two that failed and the one that's running). Each line starts with the name of the replica, which includes the service name, replica number, replica ID, and name of host that it's scheduled on. Following that is the log output.

It's hard to tell because it's trimmed to fit the book, but it looks like the first two replicas failed because they were trying to connect to another service that was still starting (a sort of race condition when dependent services are starting).

You can follow the logs (`--follow`), tail them (`--tail`), and get extra details (`--details`).

Docker Swarm - 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 and is switched to run in *swarm mode*.
- `docker swarm join-token` reveals the commands and tokens needed 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. 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 including which are managers and which is the leader.
- `docker service create` is the command to 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 replicas it's running.
- `docker service ps <service>` gives more detailed information about individual service replicas.

- `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 replicas in a service up and down.
- `docker service update` lets you update many of the properties of a running service.
- `docker service logs` lets you view the logs of a service.
- `docker service rm` is the command to delete a service from the swarm. Use it with caution as it deletes all service replicas without asking for confirmation.

Chapter summary

Docker swarm is key to the operation of Docker at scale.

At its core, swarm has a secure clustering component, and an orchestration component.

The secure clustering component is enterprise-grade and offers a wealth of security and HA features that are automatically configured and extremely simple to modify.

The orchestration component allows you to deploy and manage microservices applications in a simple declarative manner. Native Docker Swarm apps are supported, and so are Kubernetes apps.

We'll dig deeper into deploying microservices apps in a declarative manner in Chapter 14.

11: Docker Networking

It's always the network!

Any time there's a an infrastructure problem, we always blame the network. Part of the reason is that networks are at the center of everything — **no network, no app!**

In the early days of Docker, networking was hard — really hard! These days it's *almost* a pleasure ;-)

In this chapter, we'll look at the fundamentals of Docker networking. Things like the Container Network Model (CNM) and `libnetwork`. We'll also get our hands dirty building some networks.

As usual, we'll split the chapter into three parts:

- The TLDR
- The deep dive
- The commands

Docker Networking - The TLDR

Docker runs applications inside of containers, and these need to communicate over lots of different networks. This means Docker needs strong networking capabilities.

Fortunately, Docker has solutions for container-to-container networks, as well as connecting to existing networks and VLANs. The latter is important for containerized apps that need to communicate with functions and services on external systems such as VM's and physicals.

Docker networking is based on an open-source pluggable architecture called the Container Network Model (CNM). `libnetwork` is Docker's real-world implementation of the CNM, and it provides all of Docker's core networking capabilities. Drivers plug in to `libnetwork` to provide specific network topologies.

To create a smooth out-of-the-box experience, Docker ships with a set of native drivers that deal with the most common networking requirements. These include single-host bridge networks, multi-host overlays, and options for plugging into existing VLANs. Ecosystem partners extend things even further by providing their own drivers.

Last but not least, `libnetwork` provides a native service discovery and basic container load balancing solution.

That's this big picture. Let's get into the detail.

Docker Networking - The Deep Dive

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

- The theory
- Single-host bridge networks
- Multi-host overlay networks
- Connecting to existing networks
- Service Discovery
- Ingress networking

The theory

At the highest level, Docker networking comprises three major components:

- The Container Network Model (CNM)
- `libnetwork`
- Drivers

The CNM is the design specification. It outlines the fundamental building blocks of a Docker network.

`libnetwork` is a real-world implementation of the CNM, and is used by Docker. It's written in Go, and implements the core components outlined in the CNM.

Drivers extend the model by implementing specific network topologies such as VXLAN-based overlay networks.

Figure 11.1 shows how they fit together at a very high level.

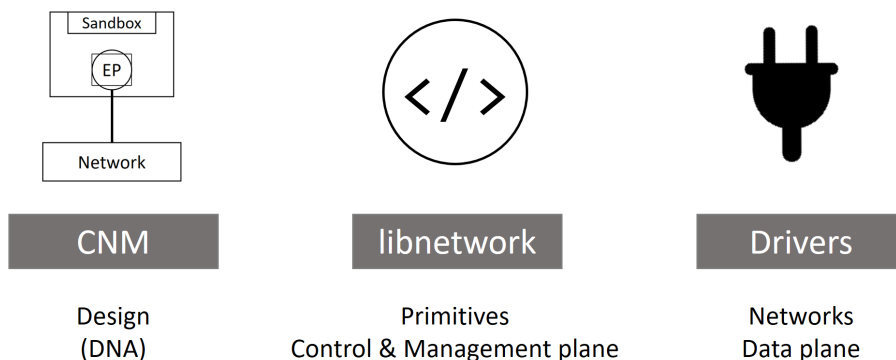


Figure 11.1

Let's look a bit closer at each.

The Container Network Model (CNM)

Everything starts with a design!

The design guide for Docker networking is the CNM. It outlines the fundamental building blocks of a Docker network, and you can read the full spec here: <https://github.com/docker/libnetwork/blob/master/docs/design.md>

I recommend reading the entire spec, but at a high level, it defines three building blocks:

- Sandboxes
- Endpoints
- Networks

A **sandbox** is an isolated network stack. It includes; Ethernet interfaces, ports, routing tables, and DNS config.

Endpoints are virtual network interfaces (E.g. `veth`). Like normal network interfaces, they're responsible for making connections. In the case of the CNM, it's the job of the *endpoint* to connect a *sandbox* to a *network*.

Networks are a software implementation of an 802.1d bridge (more commonly known as a switch). As such, they group together, and isolate, a collection of endpoints that need to communicate.

Figure 11.2 shows the three components and how they connect.

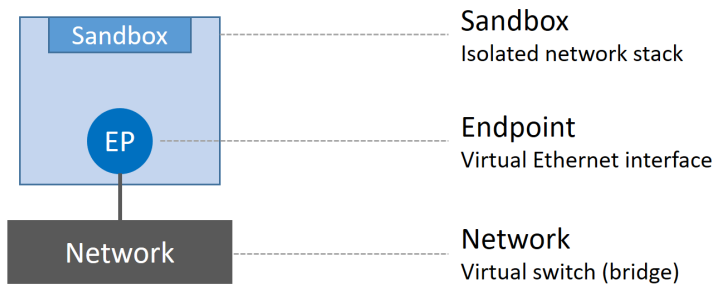


Figure 11.2 The Container Network Model (CNM)

The atomic unit of scheduling in a Docker environment is the container, and as the name suggests, the Container Network Model is all about providing networking to containers. Figure 11.3 shows how CNM components relate to containers — sandboxes are placed inside of containers to provide them with network connectivity.

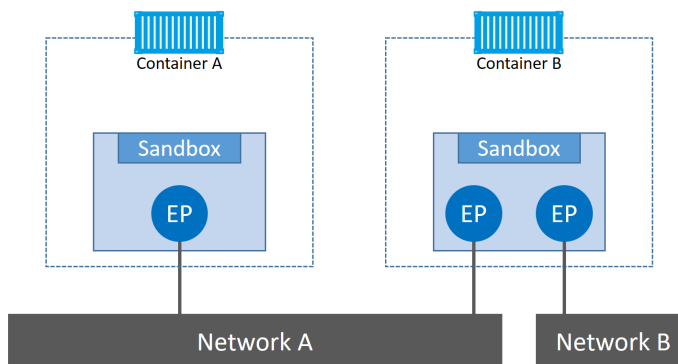


Figure 11.3

Container A has a single interface (endpoint) and is connected to Network A.

Container B has two interfaces (endpoints) and is connected to Network A **and** Network B. The containers will be able to communicate because they are both connected to Network A. However, the two *endpoints* in Container B cannot communicate with each other without the assistance of a layer 3 router.

It's also important to understand that *endpoints* behave like regular network adapters, meaning they can only be connected to a single network. Therefore, if a container needs connecting to multiple networks, it will need multiple endpoints.

Figure 11.4 extends the diagram again, this time adding a Docker host. Although Container A and Container B are running on the same host, their network stacks are completely isolated at the OS-level via the sandboxes.

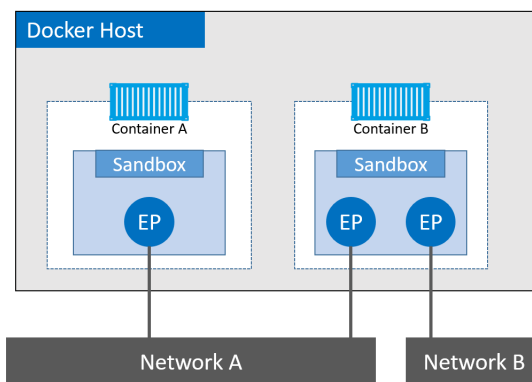


Figure 11.4

Libnetwork

The CNM is the design doc, and `libnetwork` is the canonical implementation. It's open-source, written in Go, cross-platform (Linux and Windows), and used by Docker.

In the early days of Docker, all the networking code existed inside the daemon. This was a nightmare — the daemon became bloated, and it didn't follow the Unix principle of building modular tools that can work on their own, but also be easily composed into other projects. As a result, it all got ripped out and refactored into an external library called `libnetwork`. Nowadays, all of the core Docker networking code lives in `libnetwork`.

As you’d expect, it implements all three of the components defined in the CNM. It also implements native *service discovery*, *ingress-based container load balancing*, and the network control plane and management plane functionality.

Drivers

If `libnetwork` implements the control plane and management plane functions, then drivers implement the data plane. For example, connectivity and isolation is all handled by drivers. So is the actual creation of network objects. The relationship is shown in Figure 11.5.

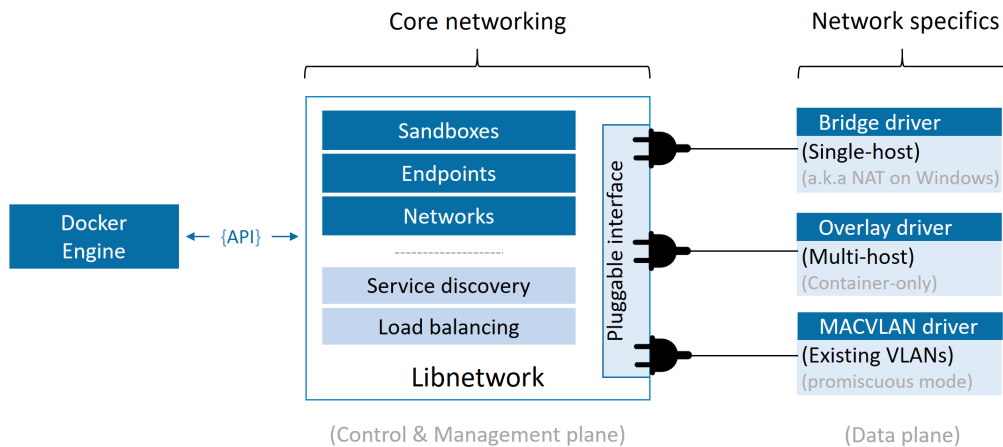


Figure 11.5

Docker ships with several built-in drivers, known as native drivers or *local drivers*. On Linux they include; bridge, overlay, and macvlan. On Windows they include; nat, overlay, transparent, and 12bridge. We’ll see how to use some of them later in the chapter.

3rd-parties can also write Docker network drivers. These are known as *remote drivers*, and examples include `calico`, `contiv`, `kuryr`, and `weave`.

Each driver is in charge of the actual creation and management of all resources on the networks it is responsible for. For example, an overlay network called “prod-fe-cuda” will be owned and managed by the `overlay` driver. This means the `overlay` driver will be invoked for the creation, management, and deletion of all resources on that network.

In order to meet the demands of complex highly-fluid environments, `libnetwork` allows multiple network drivers to be active at the same time. This means your Docker environment can sport a wide range of heterogeneous networks.

Single-host bridge networks

The simplest type of Docker network is the single-host bridge network.

The name tells us two things:

- **Single-host** tells us it only exists on a single Docker host and can only connect containers that are on the same host.
- **Bridge** tells us that it's an implementation of an 802.1d bridge (layer 2 switch).

Docker on Linux creates single-host bridge networks with the built-in `bridge` driver, whereas Docker on Windows creates them using the built-in `nat` driver. For all intents and purposes, they work the same.

Figure 11.6 shows two Docker hosts with identical local bridge networks called “mynet”. Even though the networks are identical, they are independent isolated networks. This means the containers in the picture cannot communicate directly because they are on different networks.

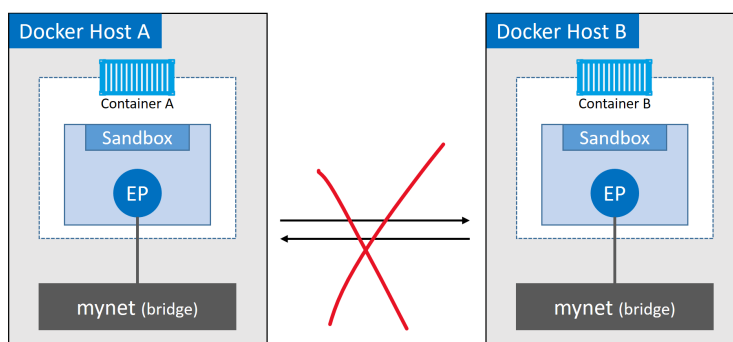


Figure 11.6

Every Docker host gets a default single-host bridge network. On Linux it's called “bridge”, and on Windows it's called “nat” (yes, those are the same names as the

drivers used to create them). By default, this is the network that all new containers will attach to unless you override it on the command line with the `--network` flag.

The following listing shows the output of a `docker network ls` command on newly installed Linux and Windows Docker hosts. The output is trimmed so that it only shows the default network on each host. Notice how the name of the network is the same as the driver that was used to create it — this is coincidence.

```
//Linux
```

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
333e184cd343	bridge	bridge	local

```
//Windows
```

```
> docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
095d4090fa32	nat	nat	local

The `docker network inspect` command is a treasure trove of great information! I highly recommended reading through its output if you're interested in low-level detail.

```
docker network inspect bridge
```

```
[
  {
    "Name": "bridge",      << Will be nat on Windows
    "Id": "333e184...d9e55",
    "Created": "2018-01-15T20:43:02.566345779Z",
    "Scope": "local",
    "Driver": "bridge",   << Will be nat on Windows
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "172.17.0.0/16"
```

```

        }
    ]
},
"Internal": false,
"Attachable": false,
"Ingress": false,
"ConfigFrom": {
    "Network": ""
},
<Snip>
}
]

```

Docker networks built with the bridge driver on Linux hosts are based on the battle-hardened *linux bridge* technology that has existed in the Linux kernel for over 15 years. This means they're high performance and extremely stable! It also means you can inspect them using standard Linux utilities. For example.

```

$ ip link show docker0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc...
    link/ether 02:42:af:f9:eb:4f brd ff:ff:ff:ff:ff:ff

```

The default “bridge” network, on all Linux-based Docker hosts, maps to an underlying *Linux bridge* in the kernel called “**docker0**”. We can see this from the output of `docker network inspect`.

```

$ docker network inspect bridge | grep bridge.name
"com.docker.network.bridge.name": "docker0",

```

The relationship between Docker’s default “bridge” network and the “docker0” bridge in the Linux kernel is shown in Figure 11.7.

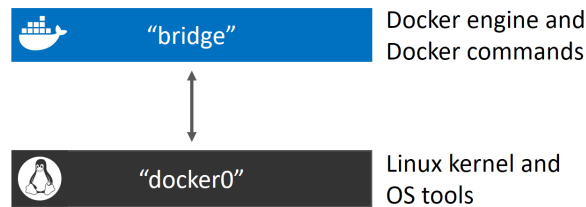


Figure 11.7

Figure 11.8 extends the diagram by adding containers at the top that plug into the “bridge” network. The “bridge” network maps to the “docker0” Linux bridge in the host’s kernel, which can be mapped back to an Ethernet interface on the host via port mappings.

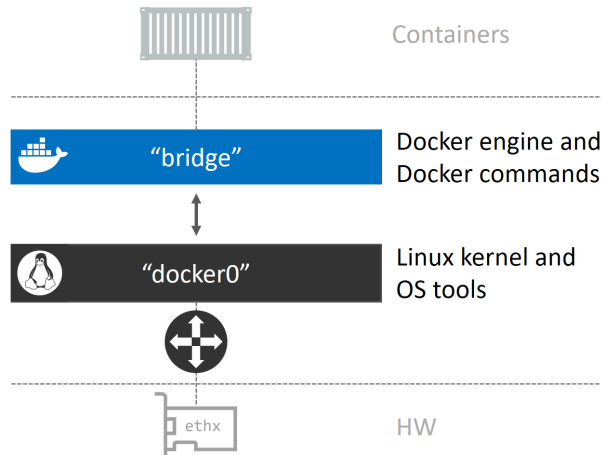


Figure 11.8

Let’s use the `docker network create` command to create a new single-host bridge network called “localnet”.

```
//Linux
$ docker network create -d bridge localnet
```

```
//Windows
> docker network create -d nat localnet
```

The new network is created, and will appear in the output of any future `docker`

network ls commands. If you are using Linux, you will also have a new *Linux bridge* created in the kernel.

Let's use the Linux `brctl` tool to look at the Linux bridges currently on the system. You may have to manually install the `brctl` binary using `apt-get install bridge-utils`, or the equivalent for your Linux distro.

```
$ brctl show
```

bridge name	bridge id	STP enabled	interfaces
docker0	8000.0242aff9eb4f	no	
br-20c2e8ae4bbb	8000.02429636237c	no	

The output shows two bridges. The first line is the “docker0” bridge that we already know about. This relates to the default “bridge” network in Docker. The second bridge (br-20c2e8ae4bbb) relates to the new localnet Docker bridge network. Neither of them have spanning tree enabled, and neither have any devices connected (interfaces column).

At this point, the bridge configuration on the host looks like Figure 11.9.

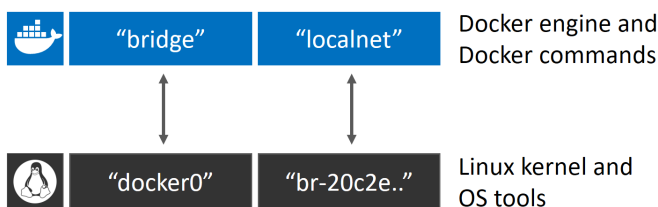


Figure 11.9

Let's create a new container and attach it to the new localnet bridge network. If you're following along on Windows, you should substitute “alpine sleep 1d” with “microsoft/powershell:nanoserver pwsh.exe -Command Start-Sleep 86400”.

```
$ docker container run -d --name c1 \
  --network localnet \
  alpine sleep 1d
```

This container will now be on the localnet network. You can confirm this with a `docker network inspect`.

```
$ docker network inspect localnet --format '{{json .Containers}}'
{
  "4edcbd...842c3aa": {
    "Name": "c1",
    "EndpointID": "43a13b...3219b8c13",
    "MacAddress": "02:42:ac:14:00:02",
    "IPv4Address": "172.20.0.2/16",
    "IPv6Address": ""
  }
},
```

The output shows that the new “c1” container is on the `localnet` bridge/nat network. If we run the Linux `brctl show` command again, we’ll see c1’s interface attached to the `br-20c2e8ae4bbb` bridge.

```
$ brctl show
bridge name      bridge id        STP enabled      interfaces
br-20c2e8ae4bbb  8000.02429636237c  no               vethe792ac0
docker0          8000.0242aff9eb4f  no
```

This is shown in Figure 11.10.

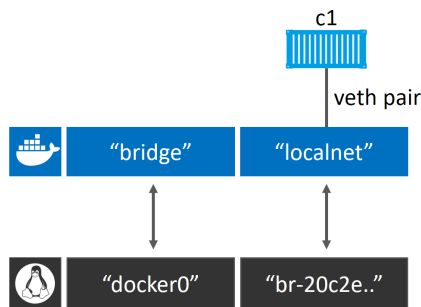


Figure 11.10

If we add another new container to the same network, it should be able to ping the “c1” container by name. This is because all new containers are registered with the embedded Docker DNS service so can resolve the names of all other containers on the same network.

Beware: The default bridge network on Linux does not support name resolution via the Docker DNS service. All other *user-defined* bridge networks do!

Let's test it.

1. Create a new interactive container called “c2” and put it on the same localnet network as “c1”.

```
//Linux
$ docker container run -it --name c2 \
  --network localnet \
  alpine sh
```

```
//Windows
> docker container run -it --name c2 `
  --network localnet `
  microsoft/powershell:nanoserver
```

Your terminal will switch into the “c2” container.

2. From within the “c2” container, ping the “c1” container by name.

```
> ping c1
Pinging c1 [172.26.137.130] with 32 bytes of data:
Reply from 172.26.137.130: bytes=32 time=1ms TTL=128
Reply from 172.26.137.130: bytes=32 time=1ms TTL=128
Control-C
```

It works! This is because the c2 container is running a local DNS resolver that forwards requests to an internal Docker DNS server. This DNS server maintains mappings for all containers started with the `--name` or `--net-alias` flag.

Try running some network-related commands while you're still logged on to the container. It's a great way of learning more about how Docker container networking works. The following snippet shows the `ipconfig` command ran from inside the “c2” Windows container previously created. You can match this IP address to the one shown in the `docker network inspect nat` output.

```
> ipconfig
Windows IP Configuration

Ethernet adapter Ethernet:

    Connection-specific DNS Suffix  . : 
    Link-local IPv6 Address . . . . . : fe80::14d1:10c8:f3dc:2eb3%4
    IPv4 Address. . . . . : 172.26.135.0
    Subnet Mask . . . . . : 255.255.240.0
    Default Gateway . . . . . : 172.26.128.1
```

So far, we've said that containers on bridge networks can only communicate with other containers on the same network. However, you can get around this using *port mappings*.

Port mappings let you map a container port to a port on the Docker host. Any traffic hitting the Docker host on the configured port will be directed to the container. The high-level flow is shown in Figure 1.11

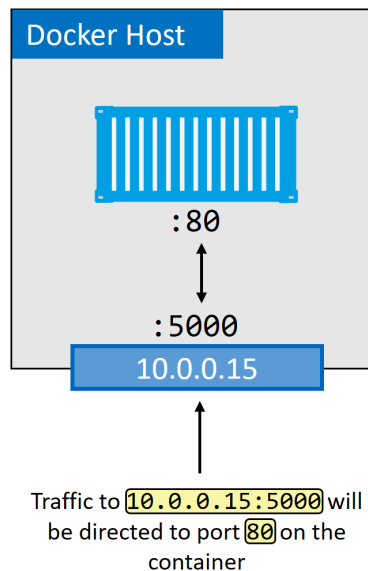


Figure 11.11

In the diagram, the application running in the container is operating on port 80. This is mapped to port 5000 on the host's 10.0.0.15 interface. The end result is all traffic hitting the host on 10.0.0.15:5000 being redirected to the container on port 80.

Let's walk through an example of mapping port 80 on a container running a web server, to port 5000 on the Docker host. The example will use NGINX on Linux. If you're following along on Windows, you'll need to substitute `nginx` with a Windows-based web server image.

1. Run a new web server container and map port 80 on the container to port 5000 on the Docker host.

```
$ docker container run -d --name web \
  --network localnet \
  --publish 5000:80 \
  nginx
```

2. Verify the port mapping.

```
$ docker port web
80/tcp -> 0.0.0.0:5000
```

This shows that port 80 in the container is mapped to port 5000 on all interfaces on the Docker host.

3. Test the configuration by pointing a web browser to port 5000 on the Docker host. To complete this step, you will need to know the IP or DNS name of your Docker host. If you're using Docker for Windows or Docker for Mac, you'll be able to use `localhost` or `127.0.0.1`.

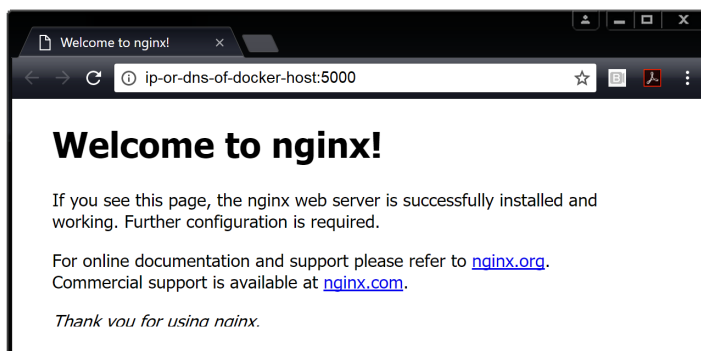


Figure 11.12

External systems, can now access the NGINX container running on the `localnet` bridge network via a port mapping to TCP port 5000 on the Docker host.

Mapping ports like this works, but it's clunky and doesn't scale. For example, only a single container can bind to any port on the host. This means no other containers will be able to use port 5000 on the host we're running the NGINX container on. This is one of the reason's that single-host bridge networks are only useful for local development and very small applications.

Multi-host overlay networks

We've got an entire chapter dedicated to multi-host overlay networks. So we'll keep this section short.

Overlay networks are multi-host. They allow a single network to span multiple hosts so that containers on different hosts can communicate at layer 2. They're ideal for container-to-container communication, including container-only applications, and they scale well.

Docker provides a native driver for overlay networks. This makes creating them as simple as adding the `--d overlay` flag to the `docker network create` command.

Connecting to existing networks

The ability to connect containerized apps to external systems and physical networks is vital. A common example is a partially containerized app — the containerized parts will need a way to communicate with the non-containerized parts still running on existing physical networks and VLANs.

The built-in `MACVLAN` driver (transparent on Windows) was created with this in mind. It makes containers first-class citizens on the existing physical networks by giving each one its own MAC and IP addresses. We show this in Figure 11.13.

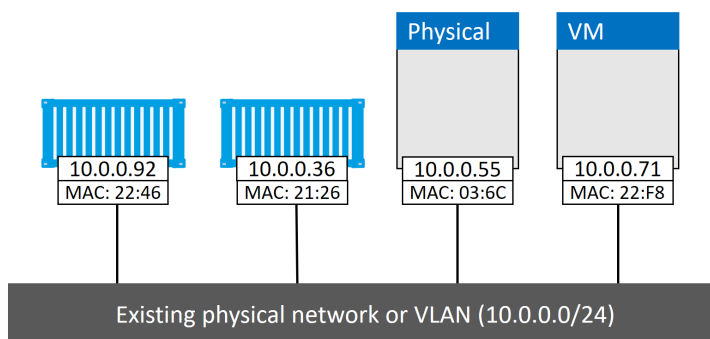


Figure 11.13

On the positive side, MACVLAN performance is good as it doesn't require port mappings or additional bridges — you connect the container interface through to the hosts interface (or a sub-interface). However, on the negative side, it requires the host NIC to be in **promiscuous mode**, which isn't allowed on most public cloud platforms. So MACVLAN is great for your corporate data center networks (assuming your network team can accommodate promiscuous mode), but it won't work in the public cloud.

Let's dig a bit deeper with the help of some pictures and a hypothetical example.

Assume we have an existing physical network with two VLANs:

- VLAN 100: 10.0.0.0/24
- VLAN 200: 192.168.3.0/24

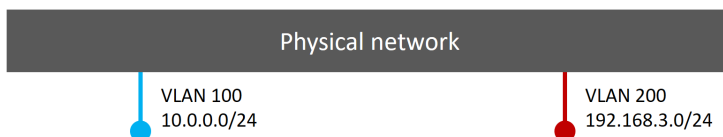


Figure 11.14

Next, we add a Docker host and connect it to the network.

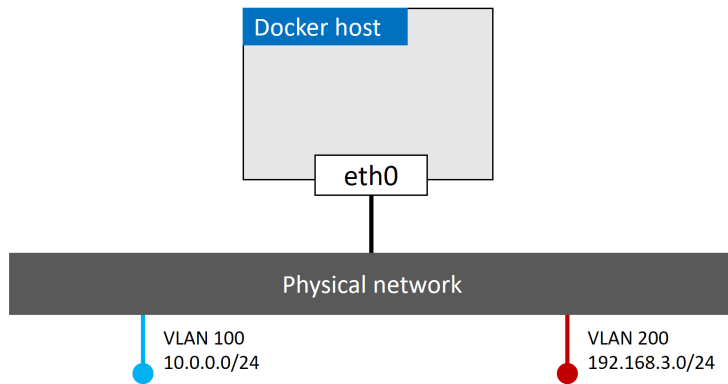


Figure 11.15

We then have a requirement for a container (app service) to be plumbed into VLAN 100. To do this, we create a new Docker network with the `macvlan` driver. However, the `macvlan` driver needs us to tell it a few things about the network we’re going to associate it with. Things like:

- Subnet info
- Gateway
- Range of IP’s it can assign to containers
- Which interface or sub-interface on the host to use

The following command will create a new MACVLAN network called “`macvlan100`” that will connect containers to VLAN 100.

```
$ docker network create -d macvlan \
  --subnet=10.0.0.0/24 \
  --ip-range=10.0.0.0/25 \
  --gateway=10.0.0.1 \
  -o parent=eth0.100 \
  macvlan100
```

This will create the “`macvlan100`” network and the `eth0.100` sub-interface. The config now looks like this.

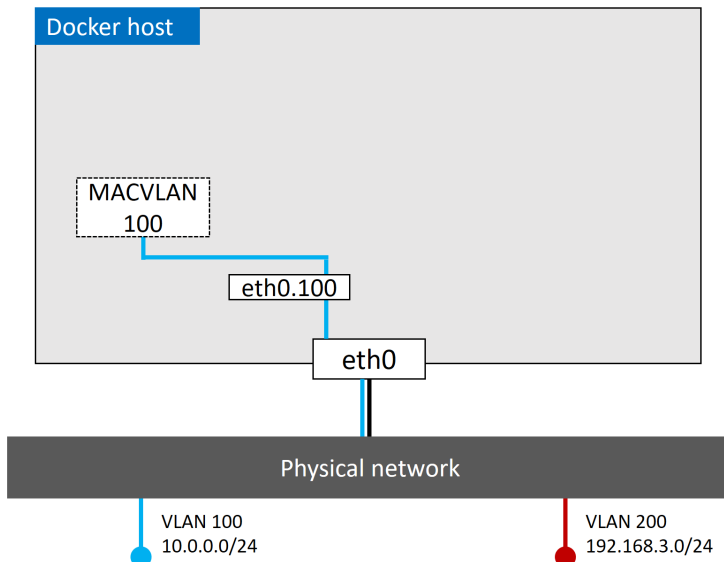


Figure 11.16

MACVLAN uses standard Linux sub-interfaces, and you have to tag them with the ID of the VLAN they will connect to. In this example we're connecting to VLAN 100, so we tag the sub-interface with `.100` (`eth0.100`).

We also used the `--ip-range` flag to tell the MACVLAN network which sub-set of IP addresses it can assign to containers. It's vital that this range of addresses be reserved for Docker and not in use by other nodes or DHCP servers, as there is no management plane feature to check for overlapping IP ranges.

The `macvlan100` network is ready for containers, so let's deploy one with the following command.

```
$ docker container run -d --name mactainer1 \  
  --network macvlan100 \  
  alpine sleep 1d
```

The config now looks like Figure 11.17. But remember, the underlying network (VLAN 100) does not see any of the MACVLAN magic, it only sees the container

with its MAC and IP addresses. And with that in mind, the “mactainer1” container will be able to ping and communicate with any other systems on VLAN 100. Pretty sweet!

Note: If you can’t get this to work, it might be because the host NIC is not in promiscuous mode. Remember that public cloud platforms do not allow promiscuous mode.

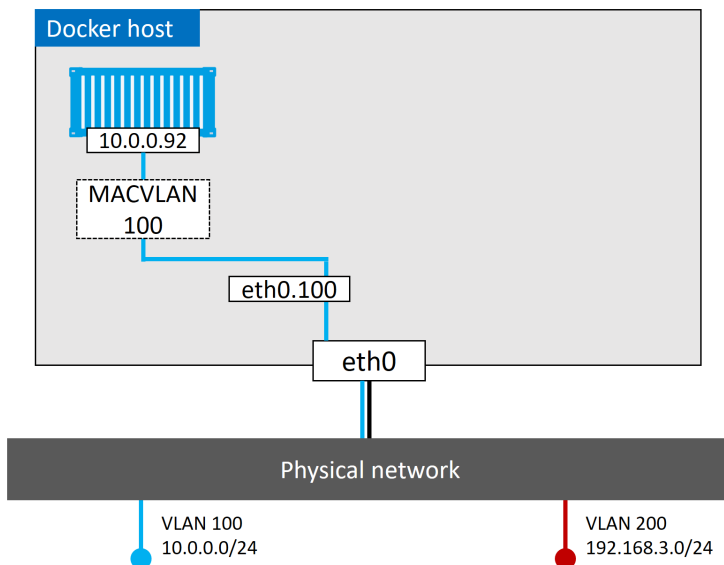


Figure 11.17

At this point, we’ve got a MACVLAN network and used it to connect a new container to an existing VLAN. However, it doesn’t stop there. The Docker MACVLAN driver is built on top of the tried-and-tested Linux kernel driver with the same name. As such, it supports VLAN trunking. This means we can create multiple MACVLAN networks and connect containers on the same Docker host to them as shown in Figure 11.18.

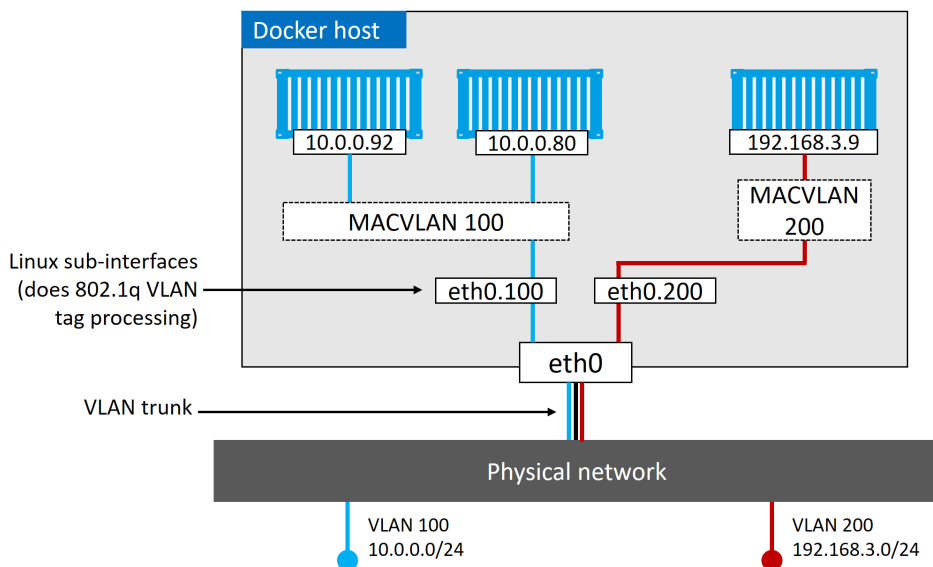


Figure 11.18

That pretty much covers MACVLAN. Windows offers a similar solution with the transparent driver.

Container and Service logs for troubleshooting

A quick note on troubleshooting connectivity issues before moving on to Service Discovery.

If you think you're experiencing connectivity issues between containers, it's worth checking the daemon logs and container logs (app logs).

On Windows systems, the daemon logs are stored under `~AppData\Local\Docker`, and you can view them in the Windows Event Viewer. On Linux, it depends what init system you're using. If you're running a `systemd`, the logs will go to `journald` and you can view them with the `journalctl -u docker.service` command. If you're not running `systemd` you should look under the following locations:

- Ubuntu systems running `upstart`: `/var/log/upstart/docker.log`

- RHEL-based systems: `/var/log/messages`
- Debian: `/var/log/daemon.log`
- Docker for Mac: `~/Library/Containers/com.docker.docker/Data/com.docker.driver/linux/console-ring`

You can also tell Docker how verbose you want daemon logging to be. To do this, you edit the daemon config file (`daemon.json`) so that “debug” is set to “true” and “log-level” is set to one of the following:

- debug The most verbose option
- info The default value and second-most verbose option
- warn Third most verbose option
- error Fourth most verbose option
- fatal Least verbose option

The following snippet from a `daemon.json` enables debugging and sets the level to debug. It will work on all Docker platforms.

```
{  
  <Snip>  
  "debug":true,  
  "log-level":"debug",  
  <Snip>  
}
```

Be sure to restart Docker after making changes to the file.

That was the daemon logs. What about container logs?

Logs from standalone containers can be viewed with the `docker container logs` command, and Swarm Service logs can be viewed with the `docker service logs` command. However, Docker supports lots of logging drivers, and they don’t all work with the `docker logs` command.

As well as a driver and configuration for engine logs, every Docker host has a default logging driver and configuration for containers. Some of the drivers include:

- json-file (default)
- journald (only works on Linux hosts running systemd)
- syslog
- splunk
- gelf

json-file and journald are probably the easiest to configure, and they both work with the `docker logs` and `docker service logs` commands. The format of the commands is `docker logs <container-name>` and `docker service logs <service-name>`.

If you're using other logging drivers you can view logs using the 3-rd party platform's native tools.

The following snippet from a `daemon.json` shows a Docker host configured to use `syslog`.

```
{
  "log-driver": "syslog"
}
```

You can configure an individual container, or service, to start with a particular logging driver with the `--log-driver` and `--log-opts` flags. These will override anything set in `daemon.json`.

Container logs work on the premise that your application is running as PID 1 in its container, and sending logs to `STDOUT`, and errors to `STDERR`. The logging driver then forwards these “logs” to the locations configured via the logging driver.

If your application logs to a file, it's possible to use a symlink to redirect log-file writes to `STDOUT` and `STDERR`.

The following is an example of running the `docker logs` command against a container called “vantage-db” configured to use the `json-file` logging driver.

```
$ docker logs vantage-db
1:C 2 Feb 09:53:22.903 # o000o000o000o Redis is starting o000o000o000o
1:C 2 Feb 09:53:22.904 # Redis version=4.0.6, bits=64, commit=00000000, modi\
fied=0, pid=1
1:C 2 Feb 09:53:22.904 # Warning: no config file specified, using the default\
t config.
1:M 2 Feb 09:53:22.906 * Running mode=standalone, port=6379.
1:M 2 Feb 09:53:22.906 # WARNING: The TCP backlog setting of 511 cannot be e\
nforced because...
1:M 2 Feb 09:53:22.906 # Server initialized
1:M 2 Feb 09:53:22.906 # WARNING overcommit_memory is set to 0!
```

There's a good chance you'll find network connectivity errors reported in the daemon logs or container logs.

Service discovery

As well as core networking, `libnetwork` also provides some important network services.

Service discovery allows all containers and Swarm services to locate each other by name. The only requirement is that they be on the same network.

Under the hood, this leverages Docker's embedded DNS server, as well as a DNS resolver in each container. Figure 11.19 shows container “c1” pinging container “c2” by name. The same principle applies to Swarm Services.

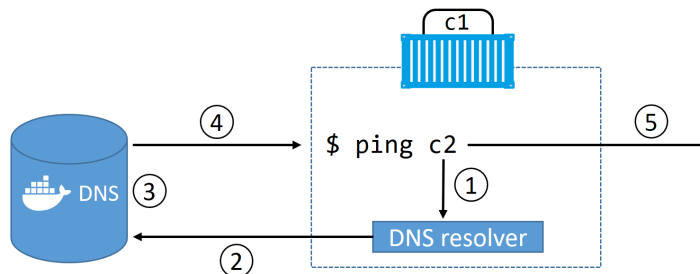


Figure 11.19

Let's step through the process.

- **step 1:** The `ping c2` command invokes the local DNS resolver to resolve the name “c2” to an IP address. All Docker containers have a local DNS resolver.
- **Step 2:** If the local resolver does not have an IP address for “c2” in its local cache, it initiates a recursive query to the Docker DNS server. The local resolver is pre-configured to know the details of the embedded Docker DNS server.
- **Step 3:** The Docker DNS server holds name-to-IP mappings for all containers created with the `--name` or `--net-alias` flags. This means it knows the IP address of container “c2”.
- **Step 4:** The DNS server returns the IP address of “c2” to the local resolver in “c1”. It does this because the two containers are on the same network — if they were on different networks this would not work.
- **Step 5:** The `ping` command is sent to the IP address of “c2”.

Every Swarm Service and standalone container started with the `--name` flag will register its name and IP with the Docker DNS service. This means all containers and service replicas can use the Docker DNS service to find each other.

However, service discovery is *network-scoped*. This means that name resolution only works for containers and Services on the same network. If two containers are on different networks, they will not be able to resolve each other.

One last point on service discovery and name resolution...

It's possible to configure Swarm Services and standalone containers with customized DNS options. For example, the `--dns` flag lets you specify a list of custom DNS servers to use in case the embedded Docker DNS server cannot resolve a query. You can also use the `--dns-search` flag to add custom search domains for queries against unqualified names (i.e. when the query is not a fully qualified domain name).

On Linux, these all work by adding entries to the `/etc/resolv.conf` file inside the container.

The following example will start a new standalone container and add the infamous 8.8.8.8 Google DNS server, as well as `dockercerts.com` as search domain to append to unqualified queries.

```
$ docker container run -it --name c1 \  
  --dns=8.8.8.8 \  
  --dns-search=dockercerts.com \  
  alpine sh
```

Ingress load balancing

Swarm supports two publishing modes that make Services accessible from outside of the cluster:

- Ingress mode (default)
- Host mode

Services published via *ingress mode* can be accessed from any node in the Swarm — even nodes **not** running a service replica. Services published via *host mode* can only be accessed via nodes running service replicas. Figure 11.20 shows the difference between the two modes.

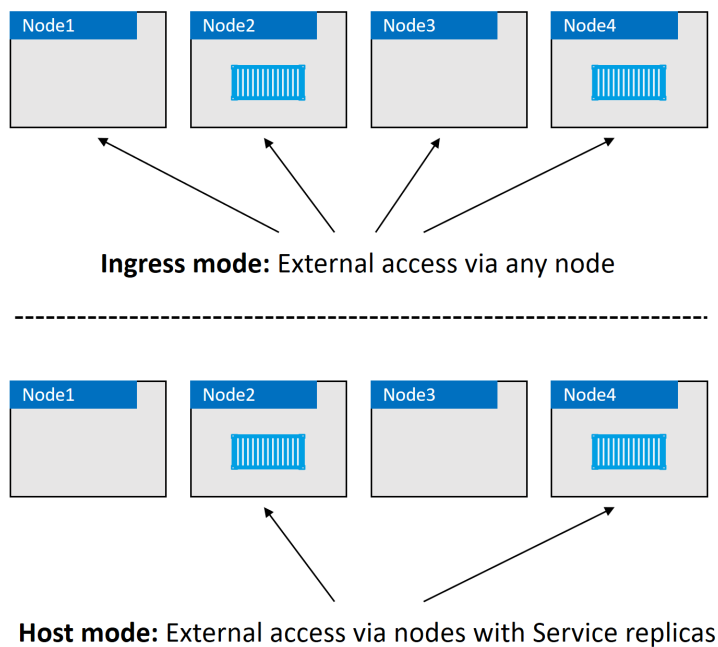


Figure 11.20

Ingress mode is the default. This means that any time you publish a service with `-p` or `--publish` it will default to *ingress mode*. To publish a service in *host mode* you need to use the long format of the `--publish` flag **and** add `mode=host`. Let's see an example using *host mode*.

```
$ docker service create -d --name svc1 \
  --publish published=5000,target=80,mode=host \
  nginx
```

A few notes about the command. `docker service create` lets you publish a service using either a *long form syntax* or *short form syntax*. The short form looks like this: `-p 5000:80` and we've seen it a few times already. However, you cannot publish a service in *host mode* using short form.

The long form looks like this: `--publish published=5000,target=80,mode=host`. It's a comma-separated list with no whitespace after each comma. The options work as follows:

- `published=5000` makes the service available externally via port 5000
- `target=80` makes sure that external requests to the published port get mapped back to port 80 on the service replicas
- `mode=host` makes sure that external requests will only reach the service if they come in via nodes running a service replica.

Ingress mode is what you'll normally use.

Behind the scenes, *ingress mode* uses a layer 4 routing mesh called the **Service Mesh** or the **Swarm Mode Service Mesh**. Figure 11.21 shows the basic traffic flow of an external request to a service exposed in ingress mode.

```
$ docker service create -d --name svc1 --network overnet \
  --publish published=5000,target=80 nginx
```

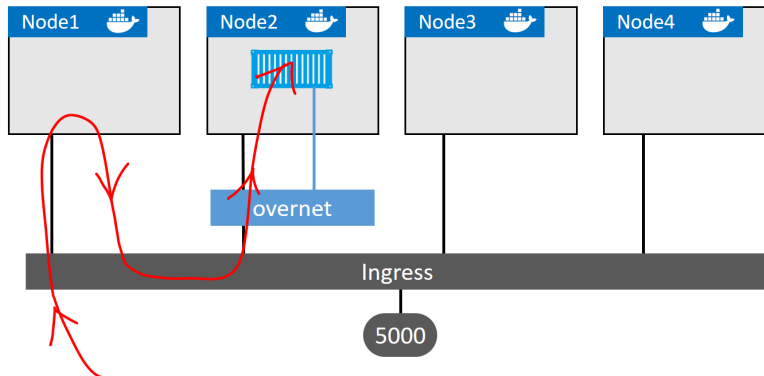


Figure 11.21

Let's quickly walk through the diagram.

1. The command at the top is deploying a new Swarm service called “svc1”. It's attaching the service to the overnet network and publishing it on port 5000.
2. Publishing a Swarm service like this (`--publish published=5000,target=80`) will publish it on port 5000 on the ingress network. As all nodes in a Swarm are attached to the ingress network, this means the port is published *swarm-wide*.
3. Logic is implemented on the cluster ensuring that any traffic hitting the ingress network, via **any node**, on port 5000 will be routed to the “svc1” service on port 80.
4. At this point, a single replica for the “svc1” service is deployed, and the cluster has a mapping rule that says “*all traffic hitting the ingress network on port 5000 needs routing to a node running a replica for the “svc1” service*”.
5. The red line shows traffic hitting node1 on port 5000 and being routed to the service replica running on node2 via the ingress network.

It's vital to know that the incoming traffic could have hit any of the four Swarm nodes on port 5000 and we would get the same result. This is because the service is published *swarm-wide* via the ingress network.

It's also vital to know that if there were multiple replicas running, as shown in Figure 11.22, the traffic would be balanced across all replicas.

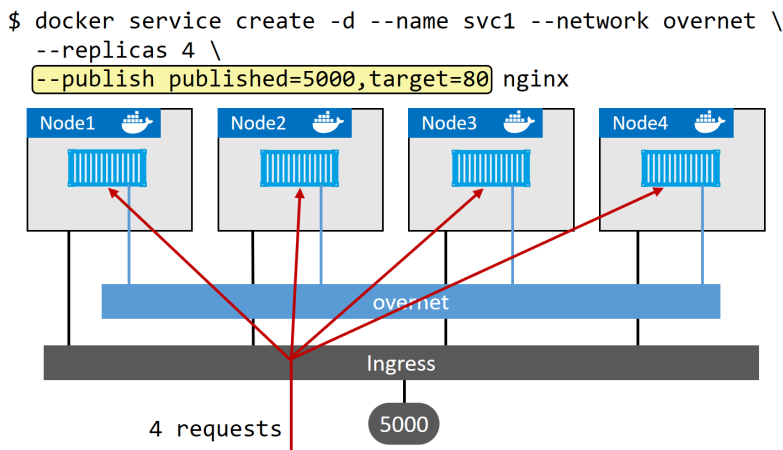


Figure 11.22

Docker Networking - The Commands

Docker networking has its own `docker network` sub-command. The main commands include:

- `docker network ls` Lists all networks on the local Docker host.
- `docker network create` Creates new Docker networks. By default, it creates them with the `nat` driver on Windows, and the `bridge` driver on Linux. You can specify the driver (type of network) with the `-d` flag. `docker network create -d overlay overnet` will create a new overlay network called `overnet` with the native Docker overlay driver.
- `docker network inspect` Provides detailed configuration information about a Docker network.
- `docker network prune` Deletes all unused networks on a Docker host.
- `docker network rm` Deletes specific networks on a Docker host.

Chapter Summary

The Container Network Model (CNM) is the master design document for Docker networking and defines the three major constructs that are used to build Docker networks — *sandboxes*, *endpoints*, and *networks*.

`libnetwork` is the open-source library, written in Go, that implements the CNM. It's used by Docker and is where all of the core Docker networking code lives. It also provides Docker's network control plane and management plane.

Drivers extend the Docker network stack (`libnetwork`) by adding code to implement specific network types, such as bridge networks and overlay networks. Docker ships with several built-in drivers, but you can also use 3rd-party drivers.

Single-host bridge networks are the most basic type of Docker network and are suitable for local development and very small applications. They do not scale, and they require port mappings if you want to publish your services outside of the network. Docker on Linux implements bridge networks using the built-in `bridge` driver, whereas Docker on Windows implements them using the built-in `nat` driver.

Overlay networks are all the rage and are excellent container-only multi-host networks. We'll talk about them in-depth in the next chapter.

The `macvlan` driver (transparent on Windows) allows you to connect containers to existing physical networks and VLANs. They make containers first-class citizens by giving them their own MAC and IP addresses. Unfortunately, they require promiscuous on the host NIC, meaning they won't work in the public cloud.

Docker also uses `libnetwork` to implement basic service discovery, as well as a service mesh for container-based load balancing of ingress traffic.

12: Docker overlay networking

Overlay networks are at the beating heart of most things we do with container-related networking. In this chapter we'll cover the fundamentals of native Docker overlay networking, as implemented in a Docker Swarm cluster.

Docker overlay networking on Windows has feature parity with Linux. This means the examples we'll use in this chapter will all work on Linux and Windows.

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

- The TLDR
- The deep dive
- The commands

Let's 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 that are 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. Containers connect to this and can communicate directly.

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

Behind the scenes, it's built on top of `libnetwork` and `drivers`.

- `libnetwork`
- `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 a container networking startup called *Socket Plane*. Two of the reasons behind the acquisition were to bring *real networking* to Docker, and to make container networking simple enough that even developers could do it :-P

They've made immense progress on both fronts.

However, 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. See Figure 12.1, and note the different networks that each node is on.

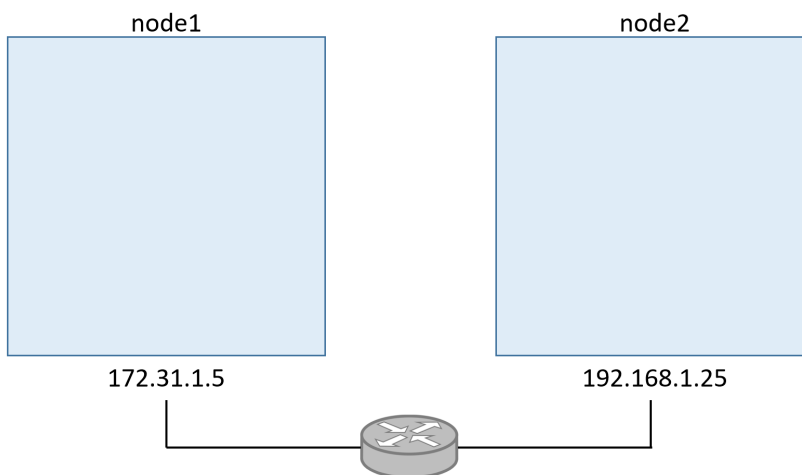


Figure 12.1

You can follow along with either Linux or Windows Docker hosts. Linux should have at least a 4.4 Linux kernel (newer is always better) and Windows should be Windows Server 2016 with the latest hotfixes installed.

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.

Run the next command on **node2**. For this to work on Windows Server, you may need to modify your Windows 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 with **node1** as a manager and **node2** as a worker.

Create a new overlay network

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

Run the following command from **node1** (manager). For this to work on Windows you may need to add a rule for port 4789/udp on your Windows Docker nodes.

```
$ 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 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
c740ydi11m89        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 initialized the Swarm.

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 them. This lazy approach improves network scalability by reducing the amount of network gossip.

Attach a service to the overlay network

Now that we have an overlay network, let's create a new *Docker service* and attach it to it. 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 uses the backtick character to split parameters over multiple lines to make the command more readable. The backtick is how PowerShell escapes line feeds.

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

Because we're running two replicas (containers), and the Swarm has two nodes, one replica will be scheduled 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. You've also attached two containers to it. How simple was that!

Test the overlay network

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

As shown in Figure 12.2, we've got two Docker hosts on separate networks, with a single overlay plumbed into both. We've got one container connected to the overlay network on each node. Let's see if they can ping each other.

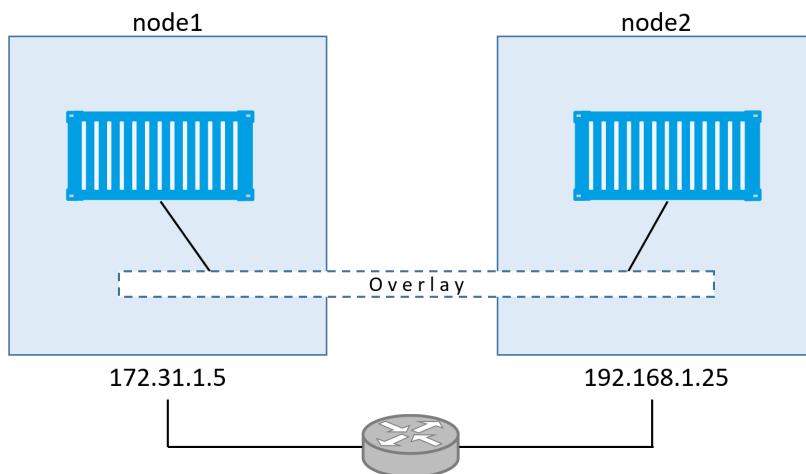


Figure 12.2

To perform the test, we'll need the IP address of each container (for the purposes of this test, we're ignoring the fact that containers on the same overlay can ping each other by name).

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**. These will get the container's ID's and IP addresses. Be sure to use the container ID's 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 12.3 shows the configuration so far. Subnet and IP addresses may be different in your lab.

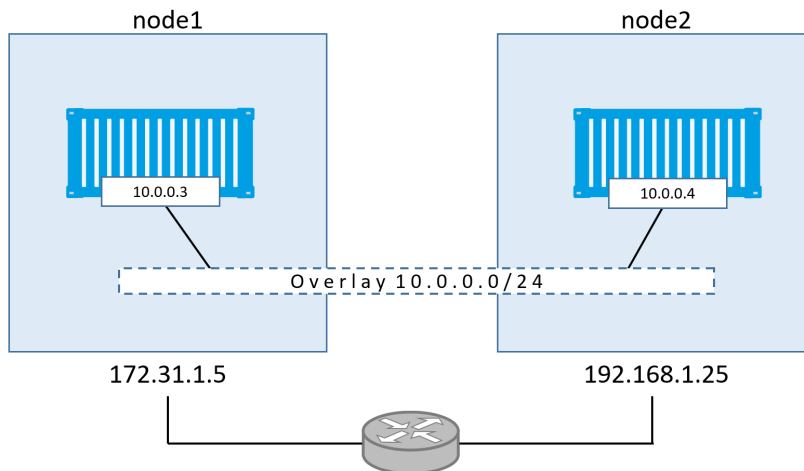


Figure 12.3

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 different 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 will be different in your environment.

Linux example:

```
$ docker container exec -it 396c8b142a85 bash
```

```
root@396c8b142a85:/# apt-get update
```

```
<Snip>
```

```
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:/# 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 pwsh.exe
```

```
Windows PowerShell
```

```
Copyright (C) 2016 Microsoft Corporation. All rights reserved.
```

```
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:\>
```

Congratulations. 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 a single hop, proving that the containers are communicating directly over the overlay network — blissfully unaware of any underlay networks that are being traversed.

Note: 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 it. The containers were scheduled on two hosts that were on two different Layer 2 underlay 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.

Some of the detail in this section 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 12.4.

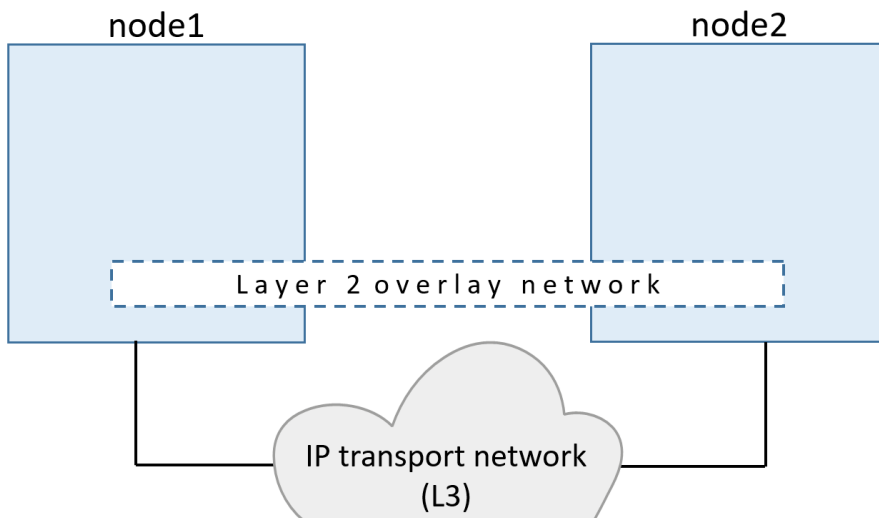


Figure 12.4

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 12.5.

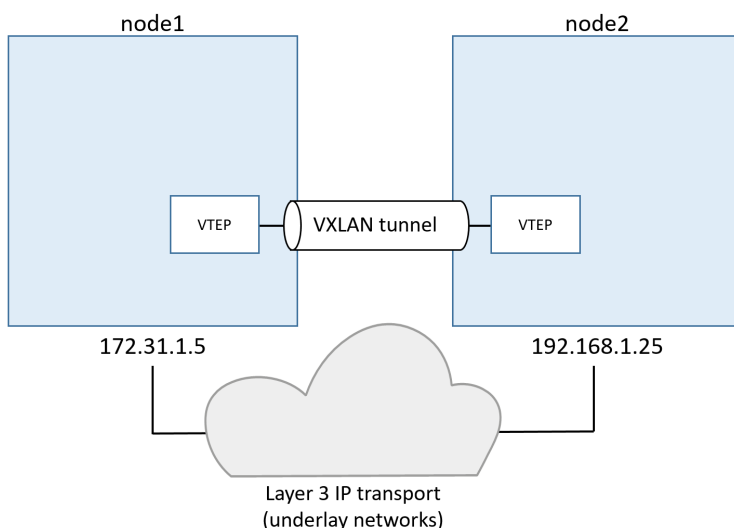


Figure 12.5

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 *sandbox* (network namespace) was created on each host. As mentioned in the previous chapter, a *sandbox* is like a container, but instead of running an application, it runs an isolated network stack — one that's sandboxed from the network stack of the host itself.

A virtual switch (a.k.a. virtual bridge) called **Br0** is created inside the sandbox. A VTEP is also created with one end plumbed into the **Br0** virtual switch, and the

other end plumbed into the host network stack (VTEP). 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 12.6.

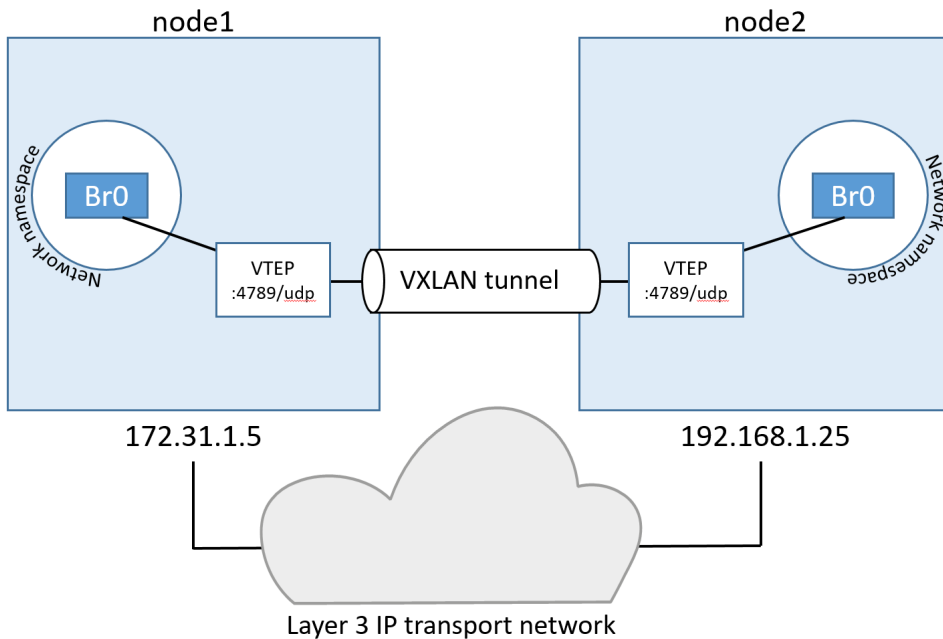


Figure 12.6

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 12.7, 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.

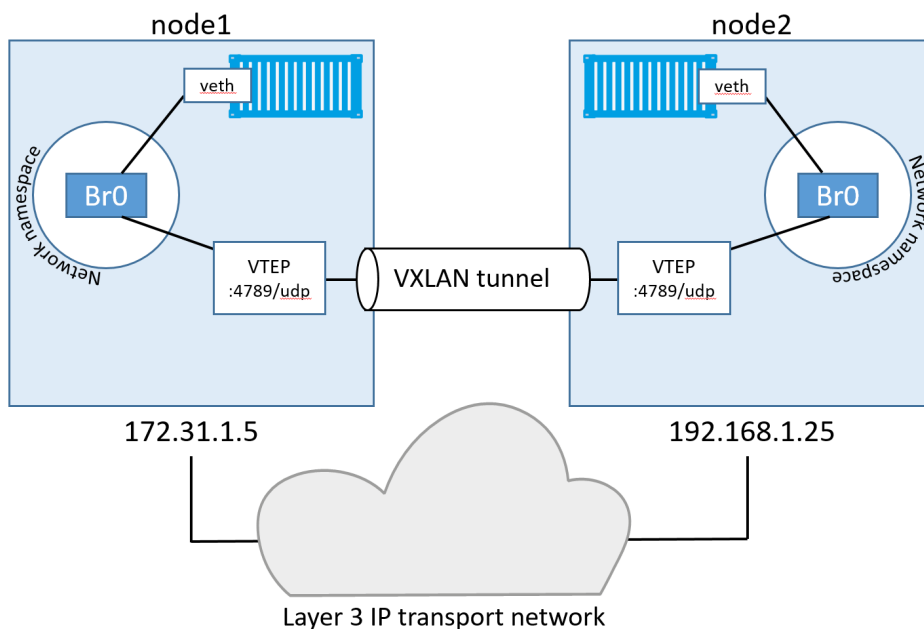


Figure 12.7

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.

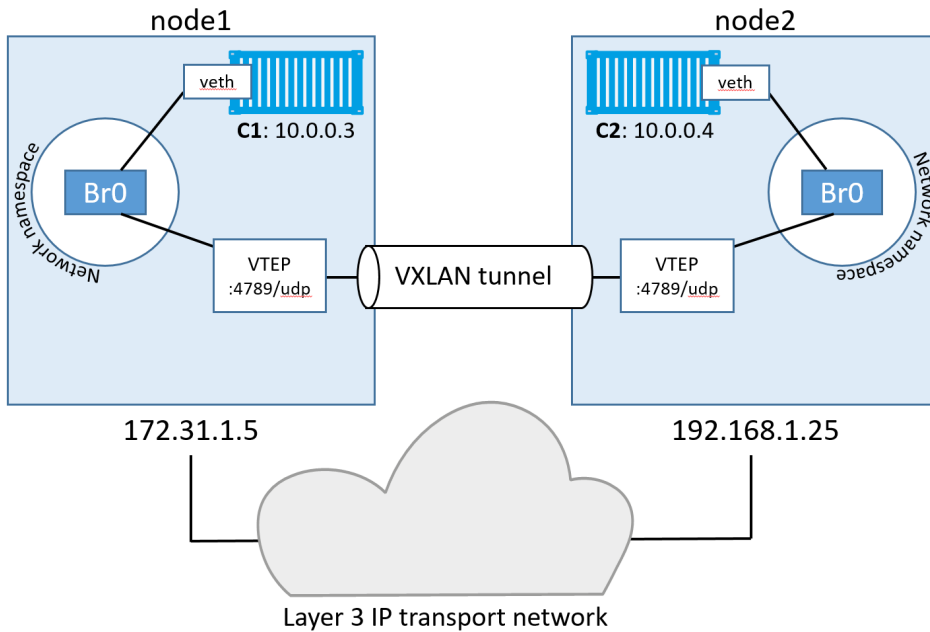


Figure 12.8

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. So it updates its ARP table, mapping 10.0.0.4 to the MAC address of the local 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 the packet can be de-encapsulated on the receiving end and forwarded to the correct VLAN. This obviously maintains network isolation. The encapsulation also wraps the frame in a UDP packet with the IP address of the remote 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 this 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 networking.

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. 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 *sandbox*, 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. You can also specify *remote* drivers from 3rd parties. For overlay networks, the control plane is encrypted by default. Just add the `-o encrypted` flag to encrypt the data plane (performance overheads may be incurred).

- `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 keeps network-related gossip to a minimum.
- `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 how easy it is to create new Docker overlay networks with the `docker network create` command. We then learned how they are put together behind the scenes using VXLAN technology.

We've only scratched the surface of what can be done with Docker overlay networking.

13: Volumes and persistent data

It's time to look at how Docker manages data. We'll look at persistent and non-persistent data. However, the main focus of the chapter will be on persistent data.

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

- The TLDR
- The deep dive
- The commands

Volumes and persistent data - The TLDR

There are two main categories of data. Persistent and non-persistent.

Persistent is the stuff you need to *keep*. Things like; customer records, financials, bookings, audit logs, and even some types of application *log* data. Non-persistent is the stuff you don't need to keep.

Both are important, and Docker has options for both.

Every Docker container gets its own non-persistent storage. It's automatically created, alongside the container, and it's tied to the lifecycle of the container. That means deleting the container will delete this storage and any data on it.

If you want your container's data to stick around (persist), you need to put it on a *volume*. Volumes are decoupled from containers, meaning you create and manage them separately, and they're not tied to the lifecycle of any container. Net result, you can delete a container with a volume, and the volume will not be deleted.

That's the TLDR. Let's take a closer look.

Volumes and persistent data - The Deep Dive

Containers are excellent for microservices design patterns. And we often associate microservices with words like *ephemeral* and *stateless*. So.... microservices are all about stateless and ephemeral workloads, and containers are great microservices. Therefore, we often jump to the conclusion that containers must be just for ephemeral stuff.

But that's wrong. Just wrong, wrong, wrong!

Containers and non-persistent data

It's true that containers are great at stateless and non-persistent stuff.

Every container automatically gets a bunch of local storage. By default, this is where all of the container's files and filesystems go. You'll hear this referred to by names like; *local storage*, *graphdriver storage*, and *snapshotter storage*. Either way, it's an integral part of the container, and is tied to the container's lifecycle — it gets created when the container gets created, and it gets deleted when the container gets deleted. Simple.

On Linux systems, it exists somewhere under `/var/lib/docker/<storage-driver>/` as part of the container. On Windows it goes under `C:\ProgramData\Docker\windowsfilter\`.

If you're running Docker in production on Linux, you'll want to make sure you match the right storage driver (graphdriver) with the version of Linux on your Docker host. Use the following list as a *guide*:

- **Red Hat Enterprise Linux:** Use the `overlay2` driver with modern versions of RHEL running Docker 17.06 or higher. Use the `devicemapper` driver with older versions. This applies to Oracle Linux and other Red Hat related upstream and downstream distros.
- **Ubuntu:** Use the `overlay2` or `aufs` drivers. If you're using a Linux 4.x kernel or higher you should go with `overlay2`.
- **SUSE Linux Enterprise Server:** Use the `btrfs` storage driver.
- **Windows** Windows only has one driver and it is configured by default.

The above list should only be used as a guide. As things progress, the `overlay2` driver is increasing in popularity and may become the recommended storage driver on more platforms. If you are using Docker Enterprise Edition (EE), and have a support contract, you should consult the latest compatibility support matrix.

Let's get back on track.

By default, all storage within a container uses this *local storage*. So every directory in a container uses this storage by default.

If your containers don't create persistent data, *local storage* will be fine and you're good to go. But if your containers **do** need to persist data, you need to read the next section.

Containers and persistent data

The recommended way to persist data in containers is with *volumes*.

At a high-level, you create a volume, then you create a container, and you mount the volume into it. The volume gets mounted to a directory in the container's filesystem, and anything written to that directory is written to the volume. If you then delete the container, the volume and its data will still exist.

Figure 13.1 shows a Docker volume mounted into a container at `/code`. Any data written to the `/code` directory will be stored on the volume and will exist after the container is deleted.

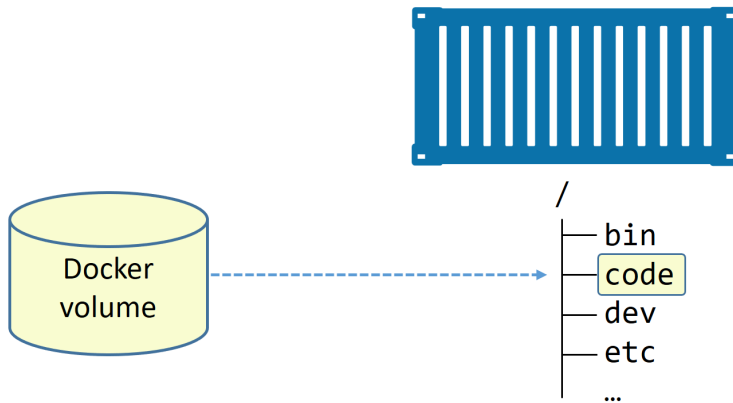


Figure 13.1 High-level view of volumes and containers

In Figure 13.1, the `/code` directory is a Docker volume. All other directories use the containers ephemeral local storage. The arrow from the volume to the `/code` directory is a dashed line to represent the decoupled relationship between volumes and containers.

Creating and managing Docker volumes

Volumes are first-class citizens in Docker. Among other things, this means they are their own object in the API, and they have their own `docker volume` sub-command.

Use the following command to create a new volume called `myvol`.

```
$ docker volume create myvol
```

By default, Docker creates new volumes with the built-in `local` driver. As the name suggests, local volumes are only available to containers on the node they're created on. Use the `-d` flag to specify a different driver.

Third-party drivers are available as plugins. These can provide advanced storage features, and integrate external storage systems with Docker. Figure 13.2 shows an external storage system (e.g. SAN or NAS) being used to provide the backend storage for the volume. The driver integrates the external storage system, with its advanced features, into the Docker environment.

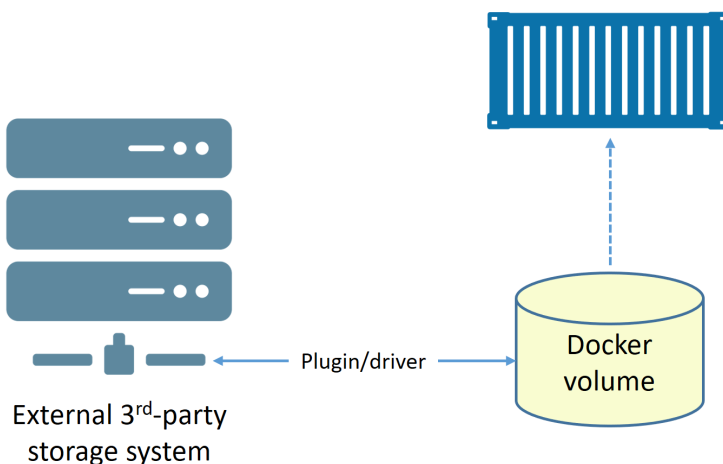


Figure 13.2 Plugging external storage into Docker

At the time of writing, there are over 25 volume plugins. These cover block storage, file storage, object storage, and more:

- **Block storage** tends to be high performance and good for small-block random access workloads. Examples of block storage systems with Docker volume plugins include HPE 3PAR, Amazon EBS, and the OpenStack Block Storage service (cinder).
- **File storage** includes systems that use the NFS and SMB protocols, and is also good for high performance workloads. Examples of file storage systems that have Docker volume plugins include NetApp FAS, Azure Files storage, and Amazon EFS.
- **Object storage** is good for long term storage of large data blobs that do not change frequently. It is often content addressable, and is usually low performance. Examples with Docker volume drivers include; Amazon S3, Ceph, and Minio.

Now that the volume is created, you can see it with the `docker volume ls` command, and inspect it with the `docker volume inspect` command.

```
$ docker volume ls
DRIVER          VOLUME NAME
local           myvol

$ docker volume inspect myvol
[
  {
    "CreatedAt": "2018-01-12T12:12:10Z",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/myvol/_data",
    "Name": "myvol",
    "Options": {},
    "Scope": "local"
  }
]
```

Some interesting points from the output of the inspect command. The driver and scope are both local. This means the volume was created with the default local driver, and is only available to containers on this Docker host. The mountpoint property tells us where on the host the volume is surfaced. In this example the volume is surfaced on the Docker host at `/var/lib/docker/volumes/myvol/_data`. On a Windows Docker host it will report as `Mountpoint": "C:\\ProgramData\\Docker\\volumes\\myvol\\_data`.

All volumes created with the local driver get their own directory under `/var/lib/docker/volumes` on Linux, and `C:\\ProgramData\\Docker\\volumes` on Windows. This means you can see them in your Docker host's filesystem, and even read and write data to them from your Docker host. We saw an example of this in the chapter on Docker Compose — where we copied a file into a volume's directory on the Docker host, and the file immediately appeared in the volume inside the container.

You can now use the `myvol` volume with Docker services and containers. For example, you can mount it into a new container using `docker container run` with the `--mount` flag. We'll see some examples in a minute.

There are two ways to delete a Docker volume:

- `docker volume prune`

- `docker volume rm`

`docker volume prune` will delete **all volumes** that are not mounted into a container or service replica, so **use with caution!** `docker volume rm` lets you specify exactly which volumes you want to delete. Neither command will delete a volume that is in use by a container or service replica.

As the `myvol` volume is not in use, delete it with the `prune` command.

```
$ docker volume prune
```

```
WARNING! This will remove all volumes not used by at least one container.  
Are you sure you want to continue? [y/N] y
```

```
Deleted Volumes:
```

```
myvol
```

```
Total reclaimed space: 0B
```

Congratulations, you've created, inspected, and deleted a Docker volume. And you did it all without interacting with a container. This demonstrates the independent nature of volumes.

At this point, you know all of the commands to create, list, inspect, and delete Docker volumes. However, it's also possible to deploy volumes via Dockerfiles using the `VOLUME` instruction. The format is `VOLUME <container-mount-point>`. However, it's not possible to specify the host directory portion in a Dockerfile. This is because *host* directories are, by nature, *host*-dependent, meaning they can change between hosts and potentially break builds. If specifying via a Dockerfile, you have to specify host directories at deploy-time.

Demonstrating volumes with containers and services

Now that we know the basic volume-related Docker commands, let's see how we use them with containers and services.

We'll be working from a system with no volumes, and everything we demonstrate applies to both Linux and Windows.

Use the following command to create a new standalone container and mount a volume called `bizvol`.

Linux example:

```
$ docker container run -dit --name voltainer \
  --mount source=bizvol,target=/vol \
  alpine
```

Windows example:

Use PowerShell for all Windows examples, and note the use of backticks (`) to split commands across multiple lines.

```
> docker container run -dit --name voltainer `
  --mount source=bizvol,target=c:\vol `
  microsoft/powershell:nanoserver
```

The command should run successfully, even though there is no volume on the system called `bizvol`. This raises an interesting point:

- If you specify an existing volume, Docker will use the existing volume
- If you specify a volume that does not exist, Docker will create it for you

In this case, `bizvol` did not exist, so Docker created it and mounted it into the new container. This means you'll be able to see it with `docker volume ls`.

```
$ docker volume ls
DRIVER          VOLUME NAME
local           bizvol
```

Although containers and volumes have separate lifecycle's, you cannot delete a volume that is in use by a container. Try it.

```
$ docker volume rm bizvol
```

```
Error response from daemon: unable to remove volume: volume is in use - [b44\
d3f82...dd2029ca]
```

The volume is currently empty. Let's exec onto the container and write some data to it. The example cited is Linux, if you're following along on Windows, you should replace sh with pwsh.exe at the end of the docker container exec command. All other commands will work on Linux and Windows.

```
$ docker container exec -it voltainer sh
```

```
/# echo "I promise to write a review of the book on Amazon" > /vol/file1
```

```
/# ls -l /vol
```

```
total 4
```

```
-rw-r--r-- 1 root  root   50 Jan 12 13:49 file1
```

```
/# cat /vol/file1
```

```
I promise to write a review of the book on Amazon
```

Type exit to return to the shell of your Docker host, and then delete the container with the following command.

```
$ docker container rm voltainer -f
voltainer
```

Even though the container is deleted, the volume still exists:

```
$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
--------------	-------	---------	---------	--------

```
$ docker volume ls
```

DRIVER	VOLUME NAME
--------	-------------

local	bizvol
-------	--------

Because the volume still exists, you can look at its mount point on the host to check if the data you wrote is still there.

Run the following commands from the terminal of your Docker host. The first one will show that the file still exists, the second will show the contents of the file.

Be sure to use the `C:\ProgramData\Docker\volumes\bizvol\_data` directory if you're following along on Windows.

```
$ ls -l /var/lib/docker/volumes/bizvol/_data/
total 4
-rw-r--r-- 1 root root 50 Jan 12 14:25 file1
```

```
$ cat /var/lib/docker/volumes/bizvol/_data/file1
I promise to write a review of the book on Amazon
```

Great, the volume and data still exists.

It's even possible to mount the `bizvol` volume into a new service or container. The following command creates a new Docker service, called `hellcat`, and mounts `bizvol` into the service replica at `/vol`.

```
$ docker service create \
  --name hellcat \
  --mount source=bizvol,target=/vol \
  alpine sleep 1d

overall progress: 1 out of 1 tasks
1/1: running  [======>]
verify: Service converged
```

We didn't specify the `--replicas` flag, so only a single service replica will be deployed. Find which node in the Swarm it's running on.

```
$ docker service ps hellcat
```

ID	NAME	NODE	DESIRED STATE	CURRENT STATE
13nh...	hellcat.1	node1	Running	Running 19 seconds ago

In this example, the replica is running on node1. Log on to node1 and get the ID of the service replica container.

```
node1$ docker container ls
```

CTR ID	IMAGE	COMMAND	STATUS	NAMES
df6..a7b	alpine:latest	"sleep 1d"	Up 25 secs	hellcat.1.13nh...

Notice that the container name is combination of service-name, replica-number, and replica-ID separated by periods.

Exec onto the container and check that the data is present in /vol. We'll use the service replica's container ID in the exec example. If you're following along on Windows, remember to replace sh with pwsh.exe.

```
node1$ docker container exec -it df6 sh
```

```
/# cat /vol/file1
```

I promise to write a review of the book on Amazon

I guess it's time to jump over to Amazon and write that book review :-D

Excellent, the volume has preserved the original data and made it available to a new container.

Sharing storage across cluster nodes

Integrating Docker with external storage systems makes it easy to share the external storage between cluster nodes. For example, a single storage LUN or NFS share can be presented to multiple Docker hosts, and therefore made available to containers and service replicas no-matter which host they're running on. Figure 13.3 shows a single external shared volume being shared presented to two Docker nodes. These Docker nodes then make the shared volume available to a couple of containers.

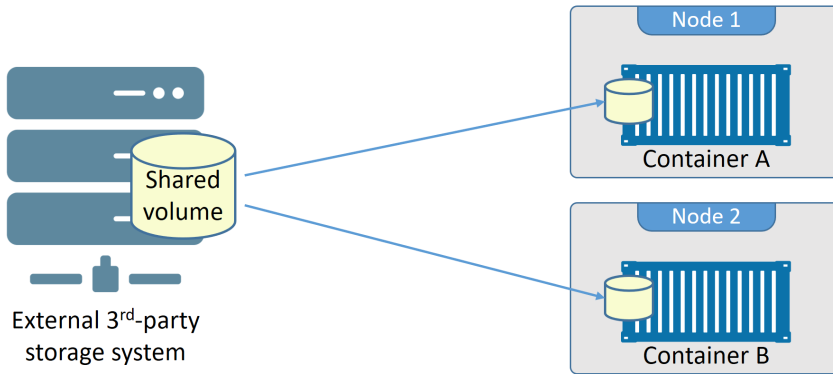


Figure 13.3

Building a setup like this requires knowledge of the external storage system, as well as how your applications read and write data to the shared storage.

A major concern with a configuration like this is **data corruption**.

Assume the following example based on Figure 13.3: Container A on node1 updates some data in the shared volume. But instead of writing the update directly to the volume, it holds it in its local buffer for faster recall. At this point, Container A thinks the data has been updated. However, before container A on node 1 flushes its buffers and commits the data to the volume, container B on node 2 updates the same data with a different value and commits it straight to the volume. At this point, both containers *think* they've updated the data in the volume, but in reality only container B has. At a later date, container A on node 1 flushes its buffers, overwriting the previous changes container B on node 2 made. But container B and node 2 may not be made aware of this. This is how corruption happens.

To prevent this, you need to write your applications in a way to avoid this.

Volumes and persistent data - The Commands

- `docker volume create` is the command we use to create new volumes. By default, volumes are created with the native `local` driver, but you can use the `-d` flag to specify a different driver.
- `docker volume ls` will list all volumes on the local Docker host.

- `docker volume inspect` shows detailed volume information. Use this command to find out where a volume exists in the Docker host's filesystem.
- `docker volume prune` will delete **all** volumes that are not in use by a container or service replica. **Use with caution!**
- `docker volume rm` deletes specific volumes that are not in use.

Chapter Summary

There are two main types of data: persistent and non-persistent data. Persistent data is data that you need to keep, non-persistent is data that you don't need to keep. By default, all containers get non-persistent storage that lives and dies with the container — we call this *local storage* and it's ideal for non-persistent data. However, if your containers create data that you need to keep, you should store the data in a Docker volume.

Docker volumes are first-class citizens in the Docker API, and are managed independently of containers with their own `docker volume` sub-command. This means that deleting a container will not delete the volumes it was using.

Volumes are the recommended way to work with persistent data in a Docker environment.

14: Deploying apps with Docker Stacks

Deploying and managing multi-service apps at scale is hard.

Fortunately, Docker Stacks are here to help! They simplify application management by providing; *desired state, rolling updates, simple, scaling operations, health checks*, and more! All wrapped in a nice declarative model. Love it!

Now then, if these buzzwords are new to you or sound complicated, don't worry! You'll understand them all by the end of the chapter!

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

- The TLDR
- The deep dive
- The commands

Deploying apps with Docker Stacks - The TLDR

Testing and deploying simple apps on your laptop is easy. But that's for amateurs. Deploying and managing multi-service apps, in real-world production environments... That's for pro's!

Fortunately, stacks are here to help!. They let you define complex multi-service apps in a single declarative file. They also provide a simple way deploy the app and manage its entire lifecycle — initial deployment > health checks > scaling > updates > rollbacks and more!

The process is simple. Define your app in a *Compose file*, then deploy and manage it with the `docker stack deploy` command. That's it!

The Compose file includes the entire stack of services that make up the app. It also includes all of the volumes, networks, secrets, and other infrastructure the app needs.

You then use the `docker stack deploy` command to deploy the app from the file. Simple.

To accomplish all of this, stacks build on top of Docker Swarm, meaning you get all of the security and advanced features that come with Swarm.

In a nutshell, Docker is great for development and testing. Docker Stacks are great for scale and production!

Deploying apps with Docker Stacks - The Deep Dive

If you know Docker Compose, you'll find Docker Stacks really easy. In fact, in many ways, stacks are what we always wished Compose was — fully integrated into Docker, and able to manage the entire lifecycle of applications.

Architecturally speaking, stacks are at the top of the Docker application hierarchy. They build on top of *services*, which in turn build on top of containers. See Figure 14.1.

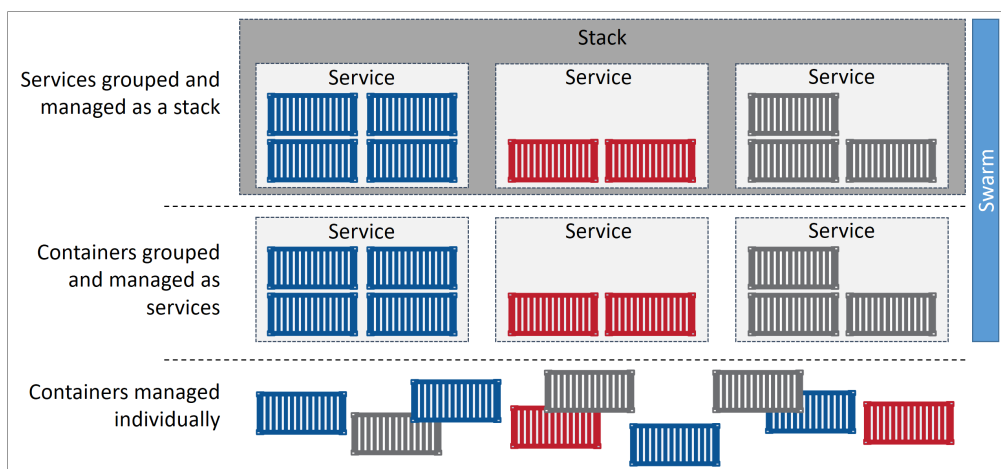


Figure 14.1 AtSea Shop high level architecture

We'll divide this section of the chapter as follows:

- Overview of the sample app
- Looking closer at the stack file
- Deploying the app
- Managing the app

Overview of the sample app

For the rest of the chapter, we'll be using the popular **AtSea Shop** demo app. It lives on [GitHub](https://github.com/dockersamples/atsea-sample-shop-app)²³ and is open-sourced under the [Apache 2.0 license](https://www.apache.org/licenses/LICENSE-2.0)²⁴.

We're using this app because it's moderately complicated without being too big to list and describe in a book. Beneath the covers, it's a multi-technology microservices app that leverages certificates and secrets. The high-level application architecture is shown in Figure 14.2.

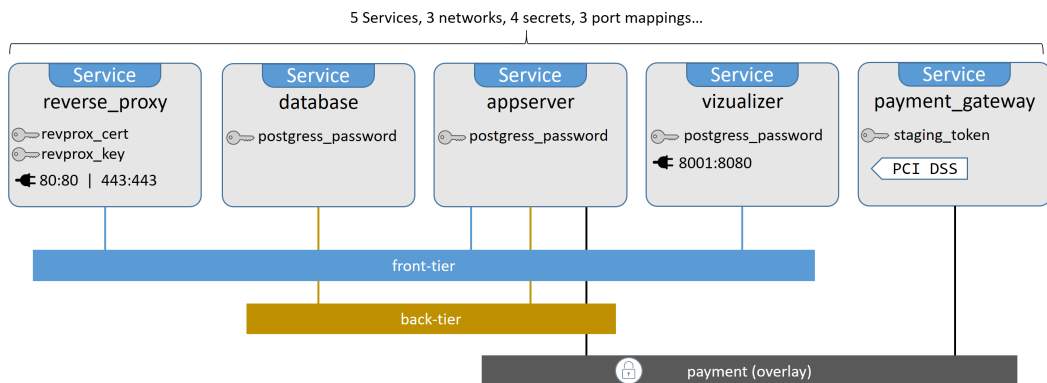


Figure 14.2 AtSea Shop high level architecture

As we can see, it comprises 5 *Services*, 3 networks, 4 secrets, and 3 port mappings. We'll see each of these in detail when we inspect the stack file.

Note: When referring to *services* in this chapter, we're talking about Docker Services (a collection of containers managed as a single object and the service object that exists in the Docker API).

²³<https://github.com/dockersamples/atsea-sample-shop-app>

²⁴<https://github.com/dockersamples/atsea-sample-shop-app/blob/master/LICENSE>

Clone the application's GitHub repo so that you have all of the application source files on your local machine.

```
$ git clone https://github.com/docker-samples/atsea-sample-shop-app.git
Cloning into 'atsea-sample-shop-app'...
remote: Counting objects: 636, done.
remote: Total 636 (delta 0), reused 0 (delta 0), pack-reused 636
Receiving objects: 100% (636/636), 7.23 MiB | 28.25 MiB/s, done.
Resolving deltas: 100% (197/197), done.
```

The application consists of several directories and source files. Feel free to explore them all. However, we're going to focus on the `docker-stack.yml` file. We'll refer to this as the *stack file*, as this defines the app and its requirements.

At the highest level, it defines 4 top-level keys.

```
version:
services:
networks:
secrets:
```

Version indicates the version of the Compose file format. This has to be 3.0 or higher to work with stacks. **Services** is where we define the stack of services that make up the app. **Networks** lists the required networks, and **secrets** defines the secrets the app uses.

If we expand each top-level key, we'll see how things map to Figure 14.1. The stack file has five services called “reverse_proxy”, “database”, “appserver”, “visualizer”, and “payment_gateway”. So does Figure 14.1. The stack file has three networks called “front-tier”, “back-tier”, and “payment”. So does Figure 14.1. Finally, the stack file has four secrets called “postgres_password”, “staging_token”, “revprox_key”, and “revprox_cert”. So does Figure 14.1.

```
version: "3.2"
services:
  reverse_proxy:
  database:
  appserver:
  visualizer:
  payment_gateway:
networks:
  front-tier:
  back-tier:
  payment:
secrets:
  postgres_password:
  staging_token:
  revprox_key:
  revprox_cert:
```

It's important to understand that the stack file captures and defines many of the requirements of the entire application. As such, it's a form of application self-documentation and a great tool for bridging the gap between dev and ops.

Let's take a closer look at each section of the stack file.

Looking closer at the stack file

The stack file is a Docker Compose file. The only requirement is that the `version:` key specify a value of "3.0" or higher. See the [the Docker docs](https://docs.docker.com/compose/compose-file/)²⁵ for the latest information on Compose file versions.

One of the first things Docker does when deploying an app from a stack file, is check for, and create the networks listed under the `networks:` key. If the networks do not already exist, Docker will create them.

Let's see the networks defined in the stack file.

Networks

²⁵<https://docs.docker.com/compose/compose-file/>

```
networks:
  front-tier:
  back-tier:
  payment:
    driver: overlay
    driver_opts:
      encrypted: 'yes'
```

Three networks are defined; `front-tier`, `back-tier`, and `payment`. By default, they'll all be created as overlay networks by the `overlay` driver. But the `payment` network is special — it requires an encrypted data plane.

By default, the control plane of all overlay networks is encrypted. To encrypt the data plane, you have two choices:

- Pass the `-o encrypted` flag to the `docker network create` command.
- Specify `encrypted: 'yes'` under `driver_opts` in the stack file.

The overhead incurred by encrypting the data plane depends on various factors such as traffic type and traffic flow. However, expect it to be in the region of 10%.

As previously mentioned, all three networks will be created before the secrets and services.

Now let's look at the secrets.

Secrets

Secrets are defined as top-level objects, and the stack file we're using defines four:

```
secrets:
  postgres_password:
    external: true
  staging_token:
    external: true
  revprox_key:
    external: true
  revprox_cert:
    external: true
```

Notice that all four are defined as `external`. This means that they must already exist before the stack can be deployed.

It's possible for secrets to be created on-demand when the application is deployed — just replace `external: true` with `file: <filename>`. However, for this to work, a plaintext file containing the unencrypted value of the secret must already exist on the host's filesystem. This has obvious security implications.

We'll see how to create these secrets when we come to deploy the app. For now, it's enough to know that the application defines four secrets that need pre-creating.

Let's look at each of the services.

Services

Services are where most of the action happens.

Each service is a JSON collection (dictionary) that contains a bunch of keys. We'll step through each one and explain what each of the options does.

The reverse_proxy service

As we can see, the `reverse_proxy` service defines an image, ports, secrets, and networks.

```
reverse_proxy:
  image: dockersamples/atseasampleshopapp_reverse_proxy
  ports:
    - "80:80"
    - "443:443"
  secrets:
    - source: revprox_cert
      target: revprox_cert
    - source: revprox_key
      target: revprox_key
  networks:
    - front-tier
```

The `image` key is the only mandatory key in the service object. As the name suggests, it defines the Docker image that will be used to build the replicas for the service.

Docker is opinionated, so unless you specify otherwise, the **image** will be pulled from Docker Hub. You can specify images from 3rd-party registries by prepending the image name with the DNS name of the registry's API endpoint such as `gcr.io` for Google's container registry.

One difference between Docker Stacks and Docker Compose, is that stacks do not support **builds**. This means all images have to be built prior to deploying the stack.

The **ports** key defines two mappings:

- `80:80` maps port 80 on the Swarm to port 80 on each service replica.
- `443:443` maps port 443 on the Swarm to port 443 on each service replica.

By default, all ports are mapped using *ingress mode*. This means they'll be mapped and accessible from every node in the Swarm — even nodes not running a replica. The alternative is *host mode*, where ports are only mapped on Swarm nodes running replicas for the service. However, *host mode* requires you to use the long-form syntax. For example, mapping port 80 in *host mode* using the long-form syntax would be like this:

```
ports:
  - target: 80
    published: 80
    mode: host
```

The long-form syntax is recommended, as it's easier to read and more powerful (it supports ingress mode **and** host mode). However, it requires at least version 3.2 of the Compose file format.

The **secrets** key defines two secrets — `revprox_cert` and `revprox_key`. These must be defined in the top-level `secrets` key, and must exist on the system.

Secrets get mounted into service replicas as a regular file. The name of the file will be whatever you specify as the `target` value in the stack file, and the file will appear in the replica under `/run/secrets` on Linux, and `C:\ProgramData\Docker\secrets` on Windows. Linux mounts `/run/secrets` as an in-memory filesystem, but Windows does not.

The secrets defined in this service will be mounted in each service replica as `/run/secrets/revprox_cert` and `/run/secrets/revprox_key`. To mount one of them as `/run/secrets/uber_secret` you would define it in the stack file as follows:

```
secrets:
  - source: revprox_cert
    target: uber_secret
```

The **networks** key ensures that all replicas for the service will be attached to the front-tier network. The network specified here must be defined in the `networks` top-level key, and if it doesn't already exist, Docker will create it as an overlay.

The database service

The database service also defines; an image, a network, and a secret. As well as those, it introduces environment variables and placement constraints.

```
database:
  image: dockersamples/atsea_db
  environment:
    POSTGRES_USER: gordonuser
    POSTGRES_DB_PASSWORD_FILE: /run/secrets/postgres_password
    POSTGRES_DB: atsea
  networks:
    - back-tier
  secrets:
    - postgres_password
  deploy:
    placement:
      constraints:
        - 'node.role == worker'
```

The **environment** key lets you inject environment variables into services replica. This service uses three environment variables to define a database user, the location of the database password (a secret mounted into every service replica), and the name of the database.

```
environment:
  POSTGRES_USER: gordonuser
  POSTGRES_DB_PASSWORD_FILE: /run/secrets/postgres_password
  POSTGRES_DB: atsea
```

Note: It would be more secure to pass all three values in as secrets, as this would avoid documenting the database name and database user in plaintext variables.

The service also defines a *placement constraint* under the `deploy` key. This ensures that replicas for this service will always run on Swarm *worker* nodes.

```
deploy:
  placement:
    constraints:
      - 'node.role == worker'
```

Placement constraints are a form of topology-aware scheduling, and can be a great way of influencing scheduling decisions. Swarm currently lets you schedule against all of the following:

- Node ID. `node.id == o2p4kw2uuw2a`
- Node name. `node.hostname == wrk-12`
- Role. `node.role != manager`
- Engine labels. `engine.labels.operatingsystem==ubuntu 16.04`
- Custom node labels. `node.labels.zone == prod1`

Notice that `==` and `!=` are both supported.

The appserver service

The `appserver` service uses an image, attaches to three networks, and mounts a secret. It also introduces several additional features under the `deploy` key.

```
appserver:
  image: dockersamples/atsea_app
  networks:
    - front-tier
    - back-tier
    - payment
  deploy:
    replicas: 2
    update_config:
      parallelism: 2
      failure_action: rollback
    placement:
      constraints:
```

```
    - 'node.role == worker'
  restart_policy:
    condition: on-failure
    delay: 5s
    max_attempts: 3
    window: 120s
  secrets:
    - postgres_password
```

Let's take a closer look at the new stuff under the `deploy` key.

First up, `services.appserver.deploy.replicas = 2` will set the desired number of replicas for the service to 2. If omitted, the default value is 1. If the service is running, and you need to change the number of replicas, you should do so declaratively. This means updating `services.appserver.deploy.replicas` in the stack file with the new value, and then redeploying the stack. We'll see this later, but re-deploying a stack does not affect services that you haven't made a change to.

`services.appserver.deploy.update_config` tells Docker how to act when rolling-out updates to the service. For this service, Docker will update two replicas at-a-time (parallelism) and will perform a 'rollback' if it detects the update is failing. Rolling back will start new replicas based on the previous definition of the service. The default value for `failure_action` is `pause`, which will stop further replicas being updated. The other option is `continue`.

```
update_config:
  parallelism: 2
  failure_action: rollback
```

The `services.appserver.deploy.restart-policy` object tells Swarm how to restart replicas (containers) if and when they fail. The policy for this service will restart a replica if it stops with a non-zero exit code (`condition: on-failure`). It will try to restart the failed replica 3 times, and wait up to 120 seconds to decide if the restart worked. It will wait 5 seconds between each of the three restart attempts.

```
restart_policy:
  condition: on-failure
  delay: 5s
  max_attempts: 3
  window: 120s
```

visualizer

The visualizer service references an image, maps a port, defines an update config, and defines a placement constraint. It also and mounts a volume and defines a custom grace period for container stop operations.

```
visualizer:
  image: dockersamples/visualizer:stable
  ports:
    - "8001:8080"
  stop_grace_period: 1m30s
  volumes:
    - "/var/run/docker.sock:/var/run/docker.sock"
  deploy:
    update_config:
      failure_action: rollback
    placement:
      constraints:
        - 'node.role == manager'
```

When Docker stops a container, it issues a SIGTERM to the process with PID 1 inside the container. The container (its PID 1 process) then has a 10-second grace period to perform any clean-up operations. If it doesn't handle the signal, it will be forcibly terminated after 10 seconds with a SIGKILL. The `stop_grace_period` property overrides this 10 second grace period.

The `volumes` key is used to mount pre-created volumes and host directories into a service replica. In this case, it's mounting `/var/run/docker.sock` from the Docker host, into `/var/run/docker.sock` inside of each service replica. This means any reads and writes to `/var/run/docker.sock` in the replica will be passed through to the same directory in the host.

`/var/run/docker.sock` happens to be the IPC socket that the Docker daemon exposes all of its API endpoints on. This means giving a container access to it allows the container to consume all API endpoints — essentially giving the container the ability to query and manage the Docker daemon. In most situations this is a huge “No!”. However, this is a demo app in a lab environment.

The reason this service requires access to the Docker socket is because it provides a graphical representation of services on the Swarm. To do this, it needs to be able to query the Docker daemon on a manager node. To accomplish this, a placement constraint forces all service replicas onto manager nodes, and the Docker socket is bind-mounted into each service replica. The *bind mount* is shown in Figure 14.3.

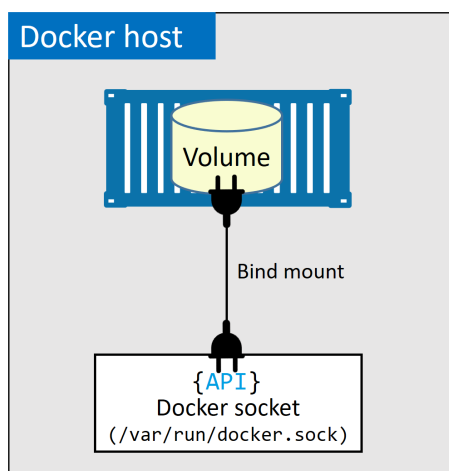


Figure 14.3

payment_gateway

The `payment_gateway` service specifies an image, mounts a secret, attaches to a network, defines a partial deployment strategy, and then imposes a couple of placement constraints.

```
payment_gateway:
  image: dockersamples/atseasampleshopapp_payment_gateway
  secrets:
    - source: staging_token
      target: payment_token
  networks:
    - payment
  deploy:
    update_config:
      failure_action: rollback
    placement:
      constraints:
        - 'node.role == worker'
        - 'node.labels.pcidss == yes'
```

We've seen all of these options before, except for the `node.label` in the placement constraint. Node labels are custom-defined labels added to Swarm nodes with the `docker node update` command. As such, they're only applicable within the context of the nodes role in the Swarm (you can't leverage them on standalone containers or outside of the Swarm).

In this example, the `payment_gateway` service performs operations that require it to run on a Swarm node that has been hardened to PCI DSS standards. To enable this, you can apply a custom *node label* to any Swarm node meeting these requirements. We'll do this when we build the lab to deploy the app.

As this service defines two placement constraints, replicas will only be deployed to nodes that match both. I.e. a **worker** node with the `pcidss=yes` node label.

Now that we're finished examining the stack file, we should have a good understanding of the application's requirements. As mentioned previously, the stack file is a great piece of application documentation. We know that the application has 5 services, 3 networks, and 4 secrets. We know which services attach to which networks, which ports need publishing, which images are required, and we even know that some services need to run on specific nodes.

Let's deploy it.

Deploying the app

There's a few pre-requisites that need taking care of before we can deploy the app:

- **Swarm mode:** We'll deploy the app as a Docker Stack, and stacks require Swarm mode.
- **Labels:** One of the Swarm worker nodes needs a custom node label.
- **Secrets:** The app uses secrets which need pre-creating before we can deploy it.

Building a lab for the sample app

In this section we'll build a three-node Linux-based Swarm cluster that satisfies all of the application's pre-req's. Once we're done, the lab will look like this.

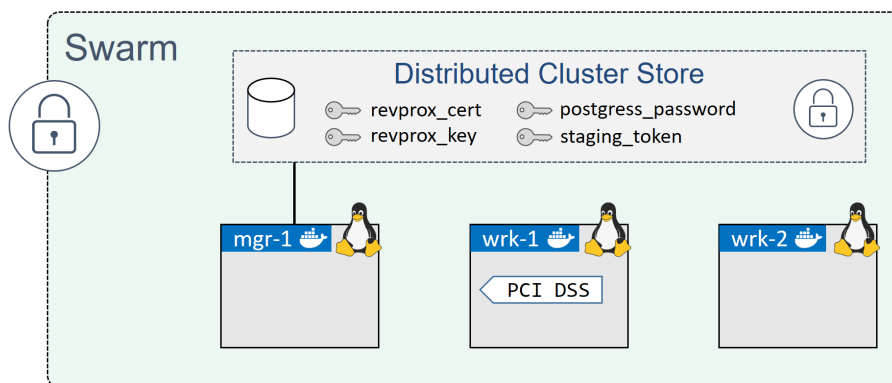


Figure 14.4 Sample lab

We'll complete the following three steps:

- Create a new Swarm
- Add a node label
- Create the secrets

Let's create a new three-node Swarm cluster.

1. Initialize a new Swarm.

Run the following command on the node that you want to be your Swarm manager.

```
$ docker swarm init
Swarm initialized: current node (lhma...w4nn) is now a manager.
<Snip>
```

2. Add worker nodes.

Copy the `docker swarm join` command that displayed in the output of the previous command. Paste it into the two nodes you want to join as workers.

```
//Worker 1 (wrk-1)
wrk-1$ docker swarm join --token SWMTKN-1-2h16...-...3lqg 172.31.40.192:2377
This node joined a swarm as a worker.

//Worker 2 (wrk-2)
wrk-2$ docker swarm join --token SWMTKN-1-2h16...-...3lqg 172.31.40.192:2377
This node joined a swarm as a worker.
```

3. Verify that the Swarm is configured with one manager and two workers.

Run this command from the manager node.

```
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
lh...4nn *	mgr-1	Ready	Active	Leader
b74...gz3	wrk-1	Ready	Active	
o9x...um8	wrk-2	Ready	Active	

The Swarm is now ready.

The `payment_gateway` service has set of placement constraints forcing it to only run on **worker nodes** with the `pcidss=yes` node label. In this step we'll add that node label to `wrk-1`.

In the real world you would harden at least one of your Docker nodes to PCI standards before labelling it. However, this is just a lab, so we'll skip the hardening step and just add the label to `wrk-1`.

Run the following commands from the Swarm manager.

1. Add the node label to wrk-1.

```
$ docker node update --label-add pcidss=yes wrk-1
```

Node labels only apply within the Swarm.

2. Verify the node label.

```
$ docker node inspect wrk-1
[
  {
    "ID": "b74rzajmrimfv7hood6l4lgz3",
    "Version": {
      "Index": 27
    },
    "CreatedAt": "2018-01-25T10:35:18.146831621Z",
    "UpdatedAt": "2018-01-25T10:47:57.189021202Z",
    "Spec": {
      "Labels": {
        "pcidss": "yes"
      }
    },
    <Snip>
```

The wrk-1 worker node is now configured so that it can run replicas for the payment_gateway service.

The application defines four secrets, all of which need creating before the app can be deployed:

- postgres_password
- staging_token
- revprox_cert
- revprox_key

Run the following commands from the manager node to create them.

1. Create a new key pair.

Three of the secrets will be populated with cryptographic keys. We'll create the keys in this step and then place them inside of Docker secrets in the next steps.

```
$ openssl req -newkey rsa:4096 -nodes -sha256 \
  -keyout domain.key -x509 -days 365 -out domain.crt
```

You'll have two new files in your current directory. We'll use them in the next step.

2. Create the `revprox_cert`, `revprox_key`, and `postgres_password` secrets.

```
$ docker secret create revprox_cert domain.crt
cqblzfpyv5cxb5wbvtrbpvrrj
```

```
$ docker secret create revprox_key domain.key
jqd1ramk2x7g0s2e9ynhdyl4p
```

```
$ docker secret create postgres_password domain.key
njpdklhjcg8noy64aileyod6l
```

3. Create the `staging_token` secret.

```
$ echo staging | docker secret create staging_token -
sqy21qep9w17h04k3600o6qsj
```

4. List the secrets.

```
$ docker secret ls
```

ID	NAME	CREATED	UPDATED
njp...d6l	postgres_password	47 seconds ago	47 seconds ago
cqb...rrj	revprox_cert	About a minute ago	About a minute ago
jqd...l4p	revprox_key	About a minute ago	About a minute ago
sqy...qsj	staging_token	23 seconds ago	23 seconds ago

That's all of the pre-requisites taken care of. Time to deploy the app!

Deploying the sample app

If you haven't already done so, clone the app's GitHub repo to your Swarm manager.

```
$ git clone https://github.com/docker-samples/atsea-sample-shop-app.git
Cloning into 'atsea-sample-shop-app'...
remote: Counting objects: 636, done.
Receiving objects: 100% (636/636), 7.23 MiB | 3.30 MiB/s, done.
remote: Total 636 (delta 0), reused 0 (delta 0), pack-reused 636
Resolving deltas: 100% (197/197), done.
Checking connectivity... done.

$ cd atsea-sample-shop-app
```

Now that you have the code, you are ready to deploy the app.

Stacks are deployed using the `docker stack deploy` command. In its basic form, it accepts two arguments:

- name of the stack file
- name of the stack

The application's GitHub repository contains a stack file called `docker-stack.yml`, so we'll use this as stack file. We'll call the stack `seastack`, though you can choose a different name if you don't like that.

Run the following commands from within the `atsea-sample-shop-app` directory on the Swarm manager.

Deploy the stack (app).

```
$ docker stack deploy -c docker-stack.yml seastack
Creating network seastack_default
Creating network seastack_back-tier
Creating network seastack_front-tier
Creating network seastack_payment
Creating service seastack_database
Creating service seastack_appserver
Creating service seastack_visualizer
Creating service seastack_payment_gateway
Creating service seastack_reverse_proxy
```

You can run `docker network ls` and `docker service ls` commands to see the networks and services that were deployed as part of the app.

A few things to note from the output of the command.

The networks were created before the services. This is because the services attach to the networks, so need the networks to be created before they can start.

Docker prepends the name of the stack to every resource it creates. In our example, the stack is called `seastack`, so all resources are named `seastack_<resource>`. For example, the payment network is called `seastack_payment`. Resources that were created prior to the deployment, such as secrets, do not get renamed.

Another thing to note is the presence of a network called `seastack_default`. This isn't defined in the stack file, so why was it created? Every service needs to attach to a network, but the `visualizer` service didn't specify one. Therefore, Docker created one called `seastack_default` and attached it to that.

You can verify the status of a stack with a couple of commands. `docker stack ls` lists all stacks on the system, including how many services they have. `docker stack ps <stack-name>` gives more detailed information about a particular stack, such as *desired state* and *current state*. Let's see them both.

```
$ docker stack ls
```

NAME	SERVICES
seastack	5

```
$ docker stack ps seastack
```

NAME	NODE	DESIRED STATE	CURRENT STATE
seastack_reverse_proxy.1	wrk-2	Running	Running 7 minutes ago
seastack_payment_gateway.1	wrk-1	Running	Running 7 minutes ago
seastack_visualizer.1	mgr-1	Running	Running 7 minutes ago
seastack_appserver.1	wrk-2	Running	Running 7 minutes ago
seastack_database.1	wrk-2	Running	Running 7 minutes ago
seastack_appserver.2	wrk-1	Running	Running 7 minutes ago

The `docker stack ps` command is a good place to start when troubleshooting services that fail to start. It gives an overview of every service in the stack, including which node each replica is scheduled on, current state, desired state, and error

message. The following output shows two failed attempts to start a replica for the `reverse_proxy` service on the `wrk-2` node.

```
$ docker stack ps seastack
```

NAME	NODE	DESIRED STATE	CURRENT STATE	ERROR
<code>reverse_proxy.1</code>	<code>wrk-2</code>	Shutdown	Failed	<code>"task: non-zero exit (1)"</code>
<code>_reverse_proxy.1</code>	<code>wrk-2</code>	Shutdown	Failed	<code>"task: non-zero exit (1)"</code>

For more detailed logs of a particular service you can use the `docker service logs` command. You pass it either the service name/ID, or replica ID. If you pass it the service name or ID, you'll get the logs for all service replicas. If you pass it a particular replica ID, you'll only get the logs for that replica.

The following `docker service logs` command shows the logs for all replicas in the `seastack_reverse_proxy` service that had the two failed replicas in the previous output.

```
$ docker service logs seastack_reverse_proxy
seastack_reverse_proxy.1.zhc3cj9d4@wrk-2 | [emerg] 1#1: host not found...
seastack_reverse_proxy.1.6m1nmbzmwh2d@wrk-2 | [emerg] 1#1: host not found...
seastack_reverse_proxy.1.6m1nmbzmwh2d@wrk-2 | nginx: [emerg] host not found..
seastack_reverse_proxy.1.zhc3cj9d4@wrk-2 | nginx: [emerg] host not found..
seastack_reverse_proxy.1.1tmya243m5um@mgr-1 | 10.255.0.2 "GET / HTTP/1.1" 302
```

The output is trimmed to fit the page, but you can see that logs from all three service replicas are shown (the two that failed and the one that's running). Each line starts with the name of the replica, which includes the service name, replica number, replica ID, and name of host that it's scheduled on. Following that is the log output.

Note: You might have noticed that all of the replicas in the previous output showed as replica number 1. This is because Docker created one at a time and only started a new one when the previous one had failed.

It's hard to tell because the output is trimmed to fit the book, but it looks like the first two replicas failed because they were relying on something in another service that was still starting (a sort of race condition when dependent services are starting).

You can follow the logs (`--follow`), tail them (`--tail`), and get extra details (`--details`).

Now that the stack is up and running, let's see how to manage it.

Managing the app

We know that a *stack* is set of related services and infrastructure that gets deployed and managed as a unit. And while that's a fancy sentence full of buzzwords, it reminds us that the stack is built from normal Docker resources — networks, volumes, secrets, services etc. This means we can inspect and reconfigure these with their normal docker commands: `docker network`, `docker volume`, `docker secret`, `docker service`...

With this in mind, it's possible to use the `docker service` command to manage services that are part of the stack. A simple example would be using the `docker service scale` command to increase the number of replicas in the `appserver` service. However, **this is not the recommended method!**

The recommended method is the declarative method, which uses the stack file as the ultimate source of truth. As such, all changes to the stack should be made to the stack file, and the updated stack file used to redeploy the app.

Here's a quick example of why the imperative method (making changes via the CLI) is bad:

Imagine that we have a stack deployed from the `docker-stack.yml` file that we cloned from GitHub earlier in the chapter. This means we have two replicas of the `appserver` service. If we use the `docker service scale` command to change that to 4 replicas, the current state of the cluster will be 4 replicas, but the stack file will still define 2. Admittedly, that doesn't sound like the end of the world. However, imagine we then make a different change to the stack, this time via the stack file, and we roll it out with the `docker stack deploy` command. As part of this rollout, the number of `appserver` replicas in the cluster will be rolled back to 2, because this is what the stack file defines. For this kind of reason, it is recommended to make all changes to the application via the stack file, and to manage the file in a proper version control system.

Let's walk through the process of making a couple of declarative changes to the stack.

We'll make the following changes:

- Increase the number of appserver replicas from 2 to 10
- Increase the stop grace period for the visualizer service to 2 minutes

Edit the `docker-stack.yml` file and update the following two values:

- `.services.appserver.deploy.replicas=10`
- `.services.visualizer.stop_grace_period=2m`

The relevant sections of the stack file will now look like this:

```
<Snip>
appserver:
  image: dockersamples/atsea_app
  networks:
    - front-tier
    - back-tier
    - payment
  deploy:
    replicas: 2          <<Updated value
<Snip>
visualizer:
  image: dockersamples/visualizer:stable
  ports:
    - "8001:8080"
  stop_grace_period: 2m  <<Updated value
<Snip>
```

Save the file and redeploy the app.

```
$ docker stack deploy -c docker-stack.yml seastack
Updating service seastack_reverse_proxy (id: z4crmmrz7zi83o0721heohsku)
Updating service seastack_database (id: 3vvpkgunetxaatbvqxfic115)
Updating service seastack_appserver (id: ljht639w33dhv0dmht1q6mueh)
Updating service seastack_visualizer (id: rbwoyuciglre01hsm5fviabjf)
Updating service seastack_payment_gateway (id: w4gsdxfnb5gofwtvmdiooqvxs)
```

Re-deploying the app like this will only update the changed components.

Run a `docker stack ps` to see the number of appserver replicas increasing.

```
$ docker stack ps seastack
```

NAME	NODE	DESIRED STATE	CURRENT STATE
seastack_visualizer.1	mgr-1	Running	Running 1 second ago
seastack_visualizer.1	mgr-1	Shutdown	Shutdown 3 seconds ago
seastack_appserver.1	wrk-2	Running	Running 24 minutes ago
seastack_appserver.2	wrk-1	Running	Running 24 minutes ago
seastack_appserver.3	wrk-2	Running	Running 1 second ago
seastack_appserver.4	wrk-1	Running	Running 1 second ago
seastack_appserver.5	wrk-2	Running	Running 1 second ago
seastack_appserver.6	wrk-1	Running	Starting 7 seconds ago
seastack_appserver.7	wrk-2	Running	Running 1 second ago
seastack_appserver.8	wrk-1	Running	Starting 7 seconds ago
seastack_appserver.9	wrk-2	Running	Running 1 second ago
seastack_appserver.10	wrk-1	Running	Starting 7 seconds ago

The output has been trimmed so that it fits on the page, and so that only the affected services are shown.

Notice that there are two lines for the `visualizer` service. One line shows a replica that was shutdown 3 seconds ago, and the other line shows a replica that has been running for 1 second. This is because we pushed a change to the `visualizer` service, so Swarm terminated the existing replica and started a new one with the new `stop_grace_period` value.

Also note that we now have 10 replicas for the `appserver` service, and that they are in various states in the “CURRENT STATE” column — some are *running* whereas others are still *starting*.

After enough time, the cluster will converge so that *desired state* and *current state* match. At that point, what is deployed and observed on the cluster will exactly match what is defined in the stack file. This is a happy place to be :-D

This update pattern should be used for all updates to the app/stack. I.e. **all changes should be made declaratively via the stack file, and rolled out using `docker stack deploy`.**

The correct way to delete a stack is with the `docker stack rm` command. Be warned though! It deletes the stack without asking for confirmation.

```
$ docker stack rm seastack
Removing service seastack_appserver
Removing service seastack_database
Removing service seastack_payment_gateway
Removing service seastack_reverse_proxy
Removing service seastack_visualizer
Removing network seastack_front-tier
Removing network seastack_payment
Removing network seastack_default
Removing network seastack_back-tier
```

Notice that the networks and services were deleted, but the secrets were not. This is because the secrets were pre-created and existed before the stack was deployed. If your stack defines volumes at the top-level, these will not be deleted by `docker stack rm` either. This is because volumes are intended as long-term persistent data stores and exist independent of the lifecycle of containers, services, and stacks.

Congratulations! You know how to deploy and manage a multi-service app using Docker Stacks.

Deploying apps with Docker Stacks - The Commands

- `docker stack deploy` is the command we use to deploy **and** update stacks of services defined in a stack file (usually `docker-stack.yml`).

- `docker stack ls` will list all stacks on the Swarm, including how many services they have.
- `docker stack ps` gives detailed information about a deployed stack. It accepts the name of the stack as its main argument, lists which node each replica is running on, and shows *desired state* and *current state*.
- `docker stack rm` is the command to delete a stack from the Swarm. It does not ask for confirmation before deleting the stack.

Chapter Summary

Stacks are the native Docker solution for deploying and managing multi-service applications. They're baked into the Docker engine, and offer a simple declarative interface for deploying and managing the entire lifecycle of an application.

We start with application code and a set of infrastructure requirements — things like networks, ports, volumes and secrets. We containerize the application and group together all of the app services and infrastructure requirements into a single declarative stack file. We set the number of replicas, as well as rolling update and restart policies. Then we take the file and deploy the application from it using the `docker stack deploy` command.

Future updates to the deployed app should be done declaratively by checking the stack file out of source control, updating it, re-deploying the app, and checking the stack file back in to source control.

Because the stack file defines things like number of service replicas, you should maintain separate stack files for each of your environments, such as dev, test and prod.

15: 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 plenty 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

Large parts of the chapter will be 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! Generally speaking, the more security layers you have, the more secure you are. Well... Docker offers a lot of security layers. Figure 15.1 shows some of the security technologies that we'll cover in the chapter.

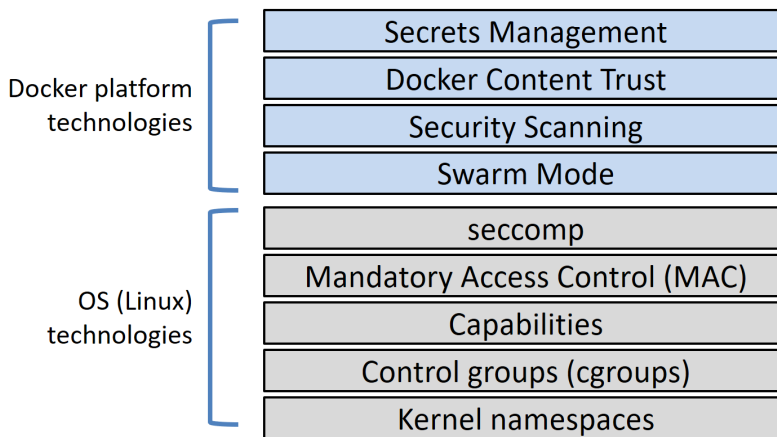


Figure 15.1

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 detailed reports.

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, stored in in-memory filesystems when in use, and operate a least-privilege model.

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 platform's own 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 main 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 on 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 may have to map each one to a separate port on the Docker host, but each can run without being re-written or reconfigured to use a different port.
- 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 15.2 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.

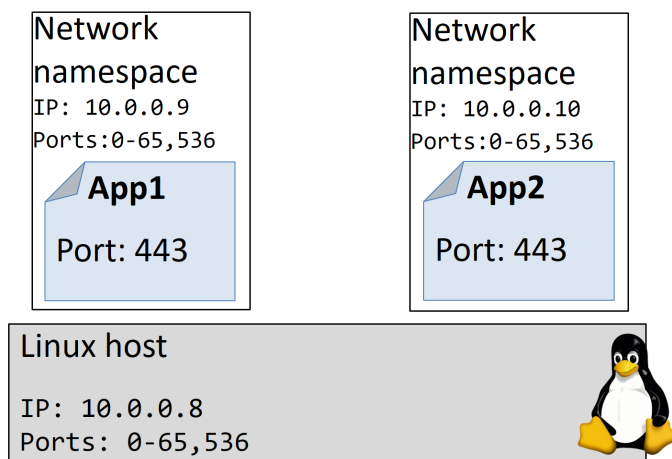


Figure 15.2

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 15.3 shows a single Linux host running two containers.

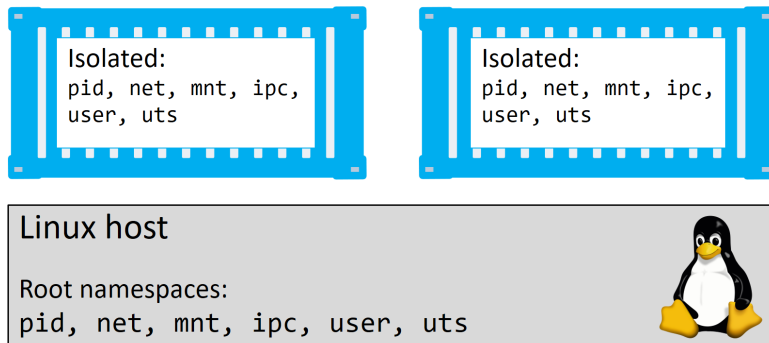


Figure 15.3

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 the 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 different users on the Linux host. A common example is mapping the `root` user of a container to a non-root user on the Linux host. User namespaces are quite new to Docker and currently optional. This may change in the future.

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

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

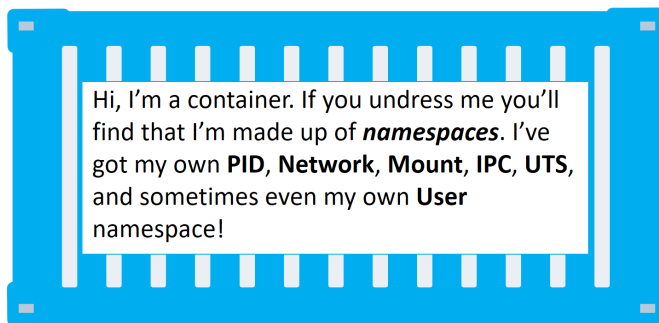


Figure 15.4

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 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 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.

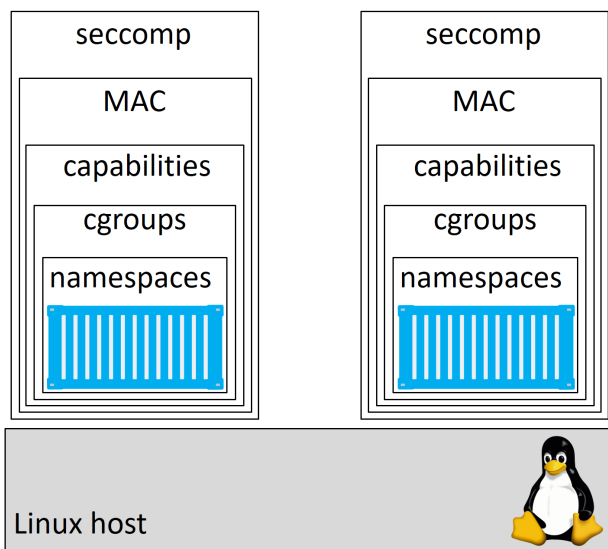


Figure 15.5

Some of these technologies can be complicated to customize, as they require deep knowledge of how they work, as well as 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 higher. The examples cited use three Docker hosts called “mgr1”, “mgr2”, and “wrk1”. Each one is running Docker 18.01.0-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 15.6.

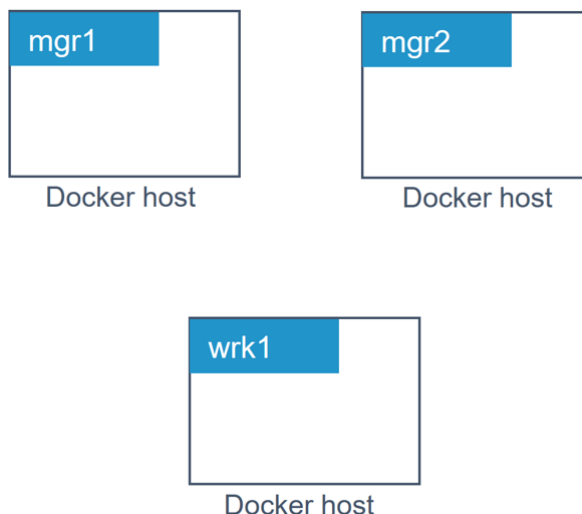


Figure 15.6

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 from “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!

“mgr1” is configured as the first manager of the Swarm and also as the root CA. The Swarm has been given a cryptographic Swarm 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 15.7 shows how the lab looks now.

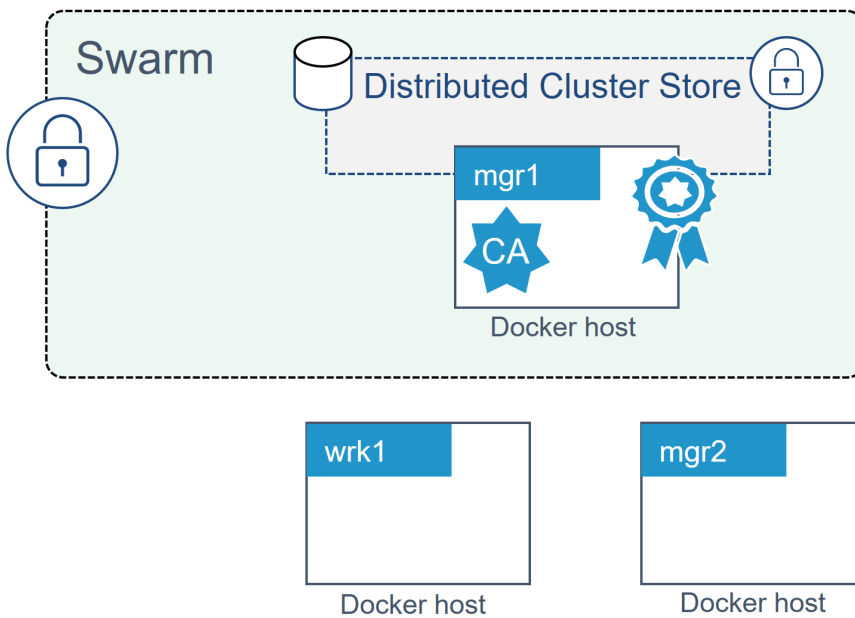


Figure 15.7

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, 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" to 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 gives you the exact command you need to run on nodes that you want 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”:

```
$ 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>:<swarm-port>`.

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

```
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
7xamk...ge662z	mgr1	Ready	Active	Leader
i0ue4...zcyj7f *	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 15.8.

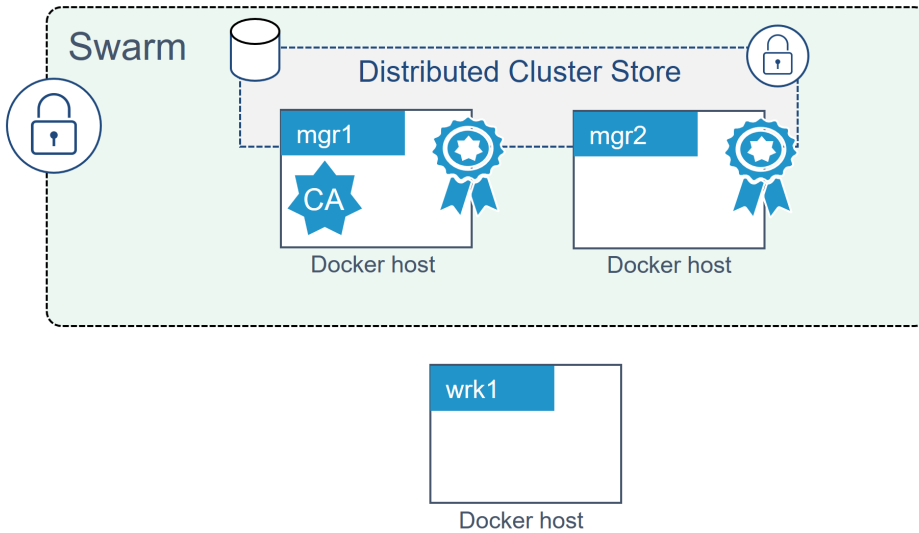


Figure 15.8

Two managers is possibly the worst number you can have. However, we’re just messing about in a demo lab, not building a business critical production environment ;-)

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
```

Again, you get the exact command you need to run on nodes that you want to join as workers. The join token and IP address will be different in your lab.

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

15: Security in Docker

308

```
$ 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...zcyj7f	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 15.9.

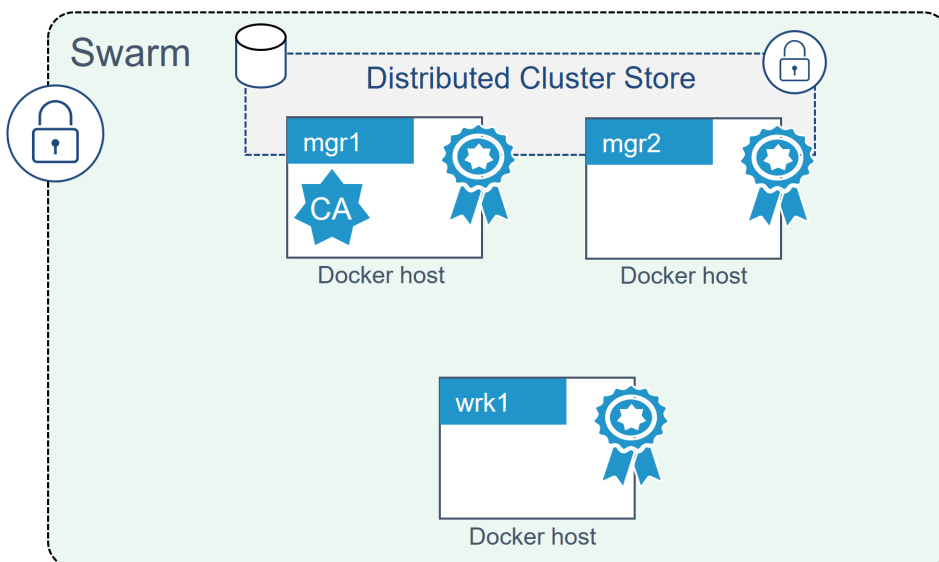


Figure 15.9

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 repos!

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", enabling you to pattern match against it and prevent people from accidentally posting it publicly. 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 it can be used to join nodes as managers or workers.

As the following shows, 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-1i7txlh6k3hb921z3yjtjcjrc7 \
  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=mfbkgjm2t1ametbnfqt2zid8x, 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 node's role in the Swarm
- The Canonical Name CN field stores the node's crypto ID.

This is shown in Figure 15.10.

```

Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      3:8f:7f:9f:f3:90:21:29:a8...
    Signature Algorithm: ecdsa-w
    Issuer: CN=swarm-ca
    Validity
      Not Before: Jul 19 07:56:06 2017 GMT
      Not After: Oct 17 08:56:06 2017 GMT
    Subject: O=mfbk...zid8x, OU=swarm-manager, CN=7xam...662z
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
  
```

Figure 15.10

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

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      # Relates to the CN field
Is Manager: true                        # Relates to the OU field
ClusterID: mfbkgjm2tlametbnfqt2zid8x   # Relates to the O field
...
<SNIP>
...
CA Configuration:
  Expiry Duration: 3 months              # Relates to Validity field
  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 following example changes the certificate rotation period to 30 days.

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

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:
```

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

```
rated
```

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 in 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 simple.

Note: At the time of writing, Docker Security Scanning is available for images in private repositories on Docker Hub. It is also available as part of the Docker Trusted Registry on-premises registry solution. Finally, 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` repo. Figure 15.11 shows the `Tags` tab of the official Alpine repo.

Scanned Images ?	
<code>edge</code> Compressed size: 2 MB Scanned 15 hours ago	✓ This image has no known vulnerabilities
<code>latest</code> Compressed size: 2 MB Scanned 15 hours ago	✓ This image has no known vulnerabilities
<code>3.6</code> Compressed size: 2 MB Scanned 15 hours ago	✓ This image has no known vulnerabilities
<code>3.5</code> Compressed size: 2 MB Scanned 15 hours ago	ⓘ This image has vulnerabilities

Figure 15.11

The Alpine repo is an official repo. This means it automatically gets scanned and scan reports are made 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 15.12.

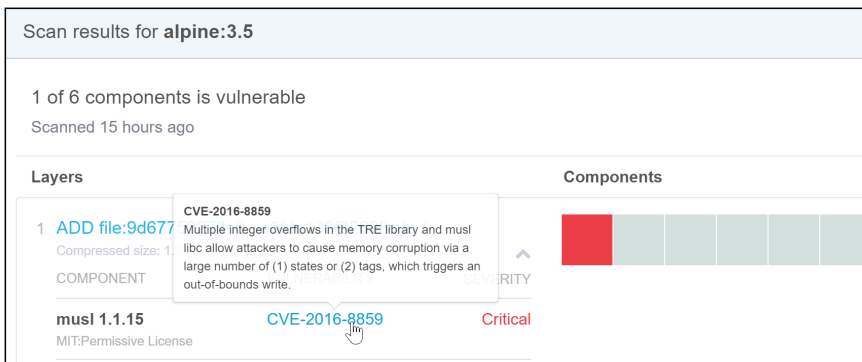


Figure 15.12

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.

That's Docker Security Scanning — a great way to deeply inspect your Docker images for known vulnerabilities. Beware though, with great knowledge comes great responsibility — once you know about vulnerabilities, it's your responsibility to deal with them.

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 15.13

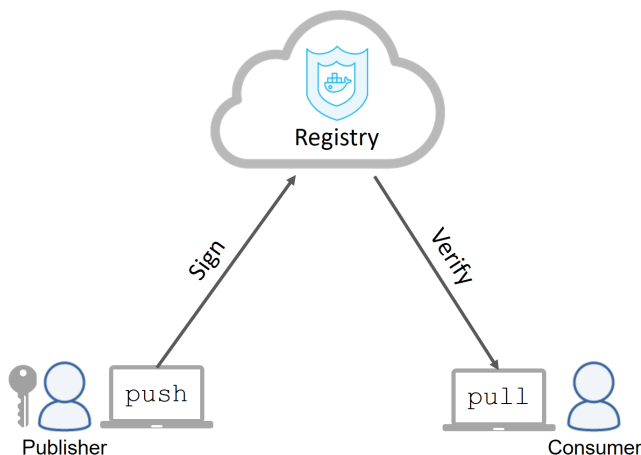


Figure 15.13

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.

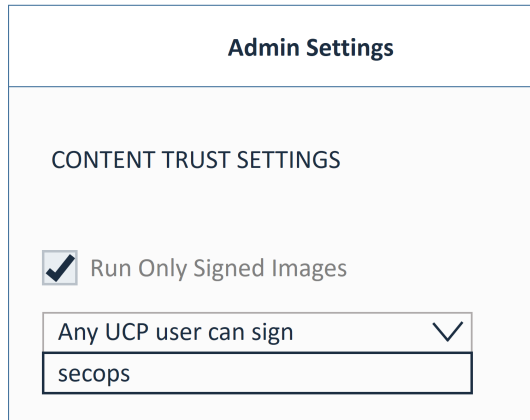
At the time of writing, the *context* offerings of DCT are in their infancy and quite complex to configure.

All you need to do, to enable DCT on a Docker host, is export an 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 15.14. This will force all nodes in the UCP cluster to only work with signed images.



The screenshot shows the 'Admin Settings' page of the Docker Universal Control Plane. Under the 'CONTENT TRUST SETTINGS' section, the 'Run Only Signed Images' checkbox is checked. Below this, there is a dropdown menu labeled 'Any UCP user can sign' with a downward arrow, and a text input field containing the value 'secops'.

Figure 15.14

You can see from Figure 15.14 that Universal Control Plane takes DCT one step further by giving the option to list security principals that are required to sign an image. 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 15.15 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”)

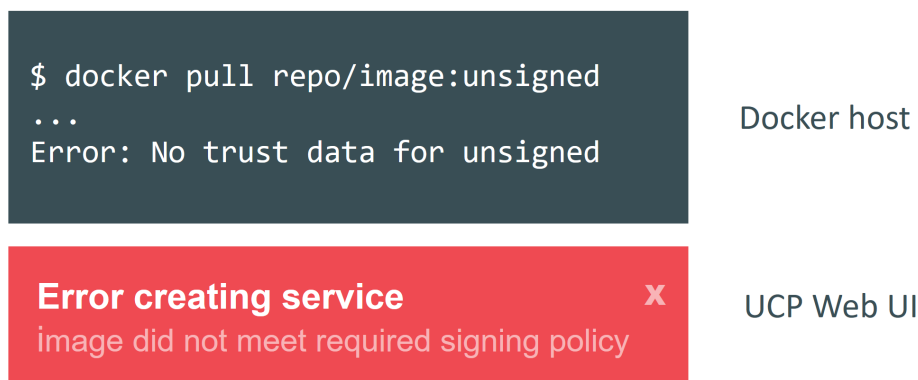


Figure 15.15

Figure 15.16 shows how DCT prevents a Docker client from pulling an image that has been tampered with. Figure 15.17 shows DCT preventing a client pulling an image that is stale.

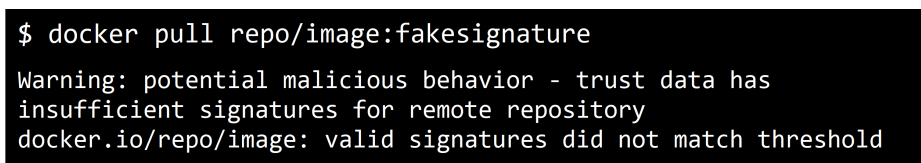


Figure 15.16 Pulling an image that has been tampered with

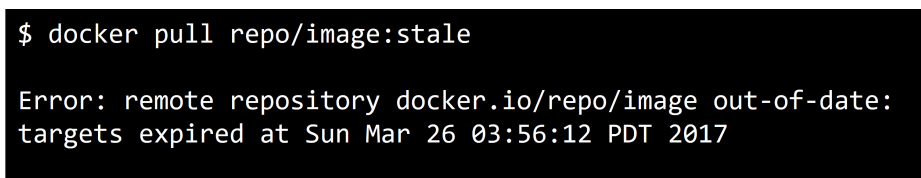


Figure 15.17 Pulling a stale image

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 operate under a least-privilege model where they are only made available to services that have been explicitly granted access to them. It's quite a comprehensive end-to-end solution.

Figure 15.18 shows a high-level workflow:

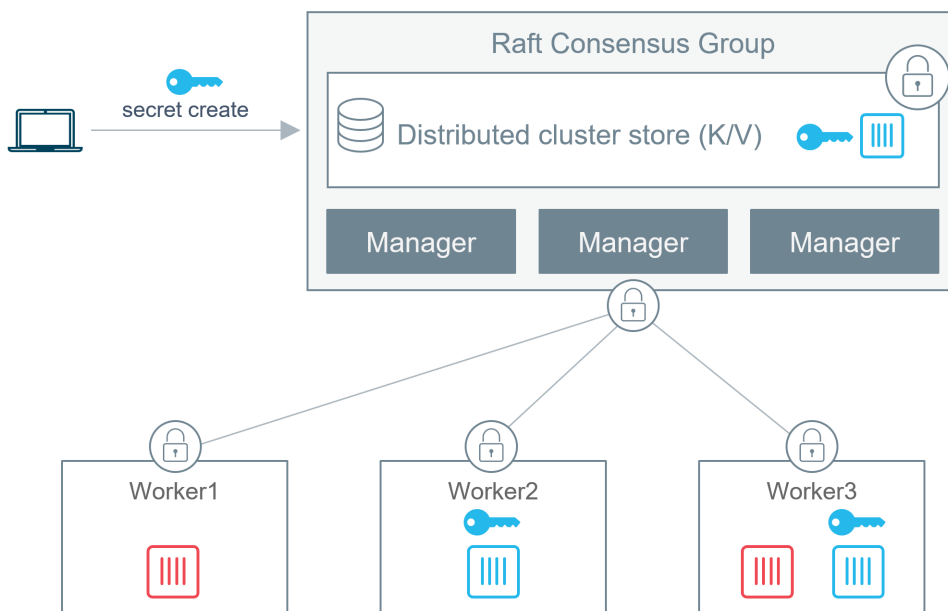


Figure 15.18

The following steps walk through the high-level workflow shown in Figure 15.18.

1. The blue 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 tasks (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 and the secret flushed from the node.
7. The red containers in the red service cannot access the secret.

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. It ships with sensible defaults for all of these, but you can customize them and even disable them.

Over and above the general Linux security technologies, the Docker platform 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 scans 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.

16: Tools for the enterprise

In this chapter, we'll look at some of the enterprise-grade tools provided by Docker, Inc. We'll see how to install them, configure them, back them up, and restore.

This will be quite a long chapter with a lot of step-by-step technical detail. I'll try and keep it interesting, but it'll be hard :-D

Other tools exist, but we'll be concentrating on the ones from Docker, Inc.

Let's dive straight in.

Tools for the enterprise - The TLDR

Docker and containers have taken the application development world by storm — building, shipping, and running applications has never been easier. So it's no surprise that enterprises want in on the action. But enterprises have a much stricter set of requirements than the typical bleeding-edge developer.

Enterprises need Docker packaged in a way they can consume. This usually means an on-premises solution that they own and manage themselves. It also means roles and security features that allow it to fit their internal structure and be *blessed* by the security department. And it means having everything backed by a meaningful support contract.

This is where Docker Enterprise Edition (EE) comes into play!

Docker EE is *Docker for the enterprise*. It's a suit of products that includes a hardened engine, an Operations UI, and a secure private registry. You can deploy it on-premises and it's wrapped in a support contract.

The high-level stack is shown in Figure 16.1.

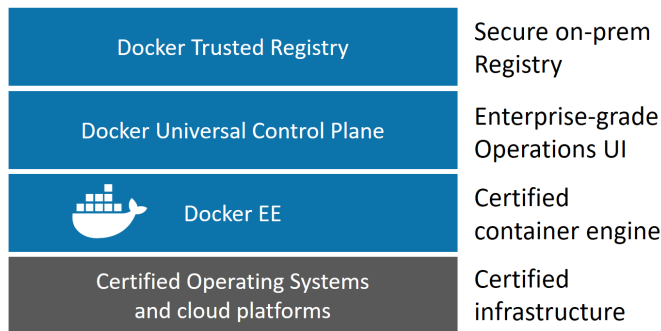


Figure 16.1 Docker EE

Tools for the enterprise - The Deep Dive

We'll divide the rest of the chapter as follows:

- Docker EE Engine
- Docker Universal Control Plane (UCP)
- Docker Trusted Registry (DTR)

We'll see how to install each, and where applicable, configure HA, and perform backup and restore jobs.

Docker EE engine

The *Docker engine* provides all of the core Docker features. Things like image and container management, networks, volumes, clustering, secrets, and more. At the time of writing, there are two versions:

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

The two biggest differences, as far as we're concerned, are the release cycles and support.

Docker EE is on a quarterly release cycle, and uses a time-based versioning scheme. For example, the June 2018 release of Docker EE will be 18.06.x-ee. Docker, Inc. guarantees 1 year of support and patches for each version.

Installing Docker EE

Installing Docker EE is simple. However, there are slight differences between platforms. We'll show you how to do it on Ubuntu 16.04, but doing it on other platforms is just as easy.

Docker EE is a subscription-based service, so you'll need a Docker ID and an active subscription. This gives you access to a unique personalized Docker EE repository that we'll configure and use in the next steps. [Trial licenses](#)²⁶ are usually available.

Note: Docker on Windows Server always installs Docker EE. See Chapter 3 for information on how to install Docker EE on Windows Server 2016.

You may need to prefix the following commands with `sudo`.

1. Make sure you've got the latest package lists.

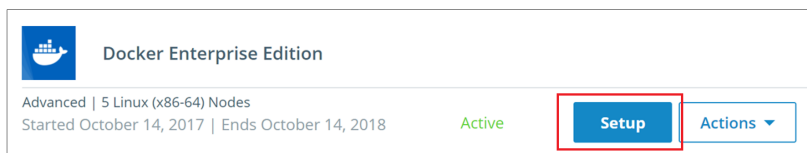
```
$ apt-get update
```

2. Install the packages needed to access the Docker EE repo over HTTPS.

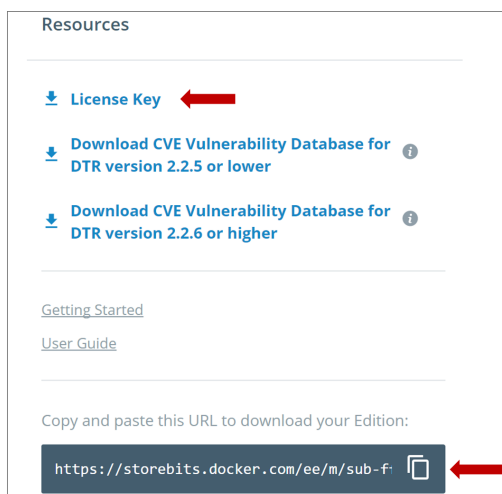
```
$ apt-get install -y \
    apt-transport-https \
    curl \
    software-properties-common
```

²⁶<https://store.docker.com/editions/enterprise/docker-ee-trial>

3. Log in to the [Docker Store](https://store.docker.com/)²⁷ and copy your unique Docker EE repository URL. Point your web browser to store.docker.com. Click your username at the top right and select My Content. Select Setup under one of your active Docker EE subscriptions.



Copy your repository URL from under the Resources pane.
Download your license as well.



We're demonstrating how to setup the repo for Ubuntu. However, this Docker Store page contains instructions on how to do the setup for other flavors of Linux.

4. Add you unique Docker EE repository URL to an environment variable.

```
$ DOCKER_EE_REPO=<paste-in-your-unique-ee-url>
```

5. Add the official Docker GPG key to all your keyrings.

²⁷<https://store.docker.com/>

```
$ curl -fsSL "${DOCKER_EE_REPO}/ubuntu/gpg" | sudo apt-key add -
```

6. Setup the latest stable repository. You may have to substitute the value on the last line with the latest stable release.

```
$ add-apt-repository \  
  "deb [arch=amd64] $DOCKER_EE_REPO/ubuntu \  
  $(lsb_release -cs) \  
  stable-17.06"
```

7. Run another apt-get update to pull the latest package lists from your newly added Docker EE repo.

```
$ apt-get update
```

8. Uninstall previous versions of Docker.

```
$ apt-get remove docker docker-engine docker-ce docker.io
```

9. Install Docker EE

```
$ apt-get install docker-ee -y
```

10. Check that the installation worked.

```
$ docker --version  
Docker version 17.06.2-ee-6, build e75fdb8
```

That's it, you've installed the Docker EE engine.

Now you can install Universal Control Plane.

Docker Universal Control Plane (UCP)

We'll be referring to Docker Universal Control Plane as **UCP** for the rest of the chapter.

UCP is an enterprise-grade container-as-a-service platform with an Operations UI. It takes the Docker Engine, and adds all of the features enterprises love and require. Things like; *RBAC*, *policies*, *trust*, *a highly-available control plane*, and a *simple UI*. Under-the-covers, it's a containerized microservices app that you download and run as a bunch of containers.

Architecturally, UCP builds on top of Docker EE in *Swarm mode*. As shown in Figure 16.4, the UCP control plane runs on Swarm managers, and apps are deployed on Swarm workers.

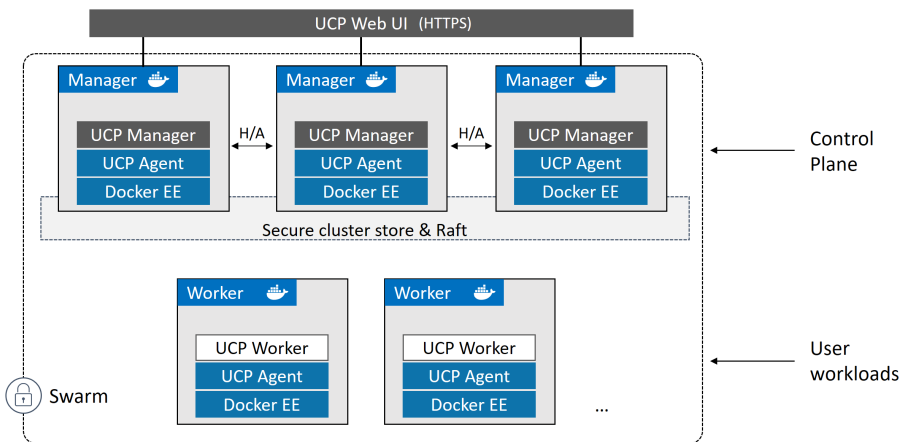


Figure 16.4 High level UCP architecture

At the time of writing, UCP managers have to be Linux. Workers can be a mix of Windows and Linux.

Planning a UCP installation

When planning a UCP installation, it's important to size and spec your cluster appropriately. We'll look at some of things you should consider.

All nodes in the cluster should have their clocks in sync (e.g. NTP). If they don't, problems can occur that are a pain to troubleshoot.

All nodes should have a static IP address and a stable DNS name.

By default, UCP managers don't run user workloads. This is a recommended best practice, and you should enforce it for production environments — it allows managers to focus solely on control plane duties. It also makes troubleshooting easier.

You should always have an odd number of managers. This helps avoid split-brain conditions where managers fail or become partitioned from the rest of the cluster. The ideal number is 3, 5, or 7, with 3 or 5 usually being the best. Having more than 7 can cause issues with the back-end Raft and cluster reconciliation. If you don't have enough nodes for 3 managers, 1 is better than 2!

If you're implementing a backup schedule (which you should) and taking regular backups, you might want to deploy 5 managers. This is because Swarm and UCP backup operations require stopping Docker and UCP services. Having 5 managers can help maintain cluster resiliency during such operations.

Manager nodes should be spread across data center availability zones. The last thing you want, is a single availability zone failing and taking all of the UCP managers with it. However, it's important to connect your managers via high-speed reliable networks. So if your data center availability zones are not connected by good networks, you might be better-off keeping all managers in a single availability zone. As a general rule, if you're deploying on public cloud infrastructure, you should deploy your managers in availability zones within a single *region*. Spanning *regions* usually involves less-reliable high-latency networks.

You can have as many *worker nodes* as you want — they don't participate in cluster Raft operations, so won't impact control plane operations.

Planning the number and size of worker nodes requires an understanding of the apps you plan on running on the cluster. For example, knowing this will help you determine things like how many Windows vs Linux nodes you require. You will also need to know if any of your apps have special requirements and need specialised worker nodes — may be PCI workloads.

Also, although the Docker engine is lightweight and small, the containerized applications you run on your nodes might not be. With this in mind, it's important to size nodes according to the CPU, RAM, network, and disk I/O requirements of your applications.

Making server sizing requirements isn't something I like to do, as it's entirely dependant on *your* workloads. However, the Docker website is currently suggesting the following **minimum** requirements for Docker UCP 2.2.4 on Linux:

- UCP Manager nodes running DTR: 8GB of RAM with 3GB of disk space
- UCP Worker nodes: 4GB of RAM with 3GB of free disk space

Recommended requirements are:

- UCP Manager nodes running DTR: 8GB RAM, 4 vCPUs, and 100GB disk space
- UCP Worker nodes: 4GB RAM 25-100GB of free disk space

Take this with a pinch of salt, and be sure to do your own sizing exercise.

One thing's for sure — Windows images are **a lot bigger** than Linux images. So be sure to factor this into your sizing.

One final word on sizing requirements. Docker Swarm and Docker UCP make it extremely easy to add and remove managers and workers. New managers are automagically added to the HA control plane, and new workers are immediately available for workload scheduling. Similarly, removing managers and workers is simple. As long as you have multiple managers, you can remove a manager without impacting cluster operations. With worker nodes, you can drain them and remove them from a running cluster. This all makes UCP very forgiving when it comes to changing your managers and workers.

With these considerations in mind, we're ready to install UCP.

Installing Docker UCP

In this section, we'll walk through the process of installing Docker UCP on the first manager node in a new cluster.

1. Run the following command from a Linux-based Docker EE node that you want to be the first manager in your UCP cluster.

A few things to note about the command. The example installs UCP using the `docker/ucp:2.2.5` image, you will want to substitute your desired version. The `--host-address` is the address you will use to access the web UI. For example, if you're installing in AWS and plan on accessing from your corporate network via the internet, you would enter the AWS public IP.

The installation is interactive, so you'll be prompted for further input to complete it.

```
$ docker container run --rm -it --name ucp \  
-v /var/run/docker.sock:/var/run/docker.sock \  
docker/ucp:2.2.5 install \  
--host-address <node-ip-address> \  
--interactive
```

2. Configure credentials.

You'll be prompted to create a username and password for the UCP Admin account. This is a local account, and you should follow your corporate guidelines for choosing the username and password. Be sure you don't forget it :-D

3. Subject alternative names (SANs).

The installer gives you the option to enter a list of alternative IP addresses and names that might be used to access UCP. These can be public and private IP addresses and DNS names, and will be added to the certificates.

A few things to note about the install.

UCP leverages Docker Swarm. This means UCP managers have to run on Swarm managers. If you install UCP on a node in *single-engine mode*, it will automatically be switched into *Swarm mode*.

The installer pulls all of the images for the various UCP services, and starts containers from them. The following listing shows some of them being pulled by the installer.

```
INFO[0008] Pulling required images... (this may take a while)
INFO[0008] Pulling docker/ucp-auth-store:2.2.5
INFO[0013] Pulling docker/ucp-hrm:2.2.5
INFO[0015] Pulling docker/ucp-metrics:2.2.5
INFO[0020] Pulling docker/ucp-swarm:2.2.5
INFO[0023] Pulling docker/ucp-auth:2.2.5
INFO[0026] Pulling docker/ucp-etcd:2.2.5
INFO[0028] Pulling docker/ucp-agent:2.2.5
INFO[0030] Pulling docker/ucp-cfssl:2.2.5
INFO[0032] Pulling docker/ucp-dsinfo:2.2.5
INFO[0080] Pulling docker/ucp-controller:2.2.5
INFO[0084] Pulling docker/ucp-proxy:2.2.5
```

Some of the interesting ones include:

- `ucp-agent` This is the main UCP agent. It gets deployed to all nodes in the cluster and is in charge of making sure the required UCP containers are up and running.
- `ucp-etcd` The cluster's persistent key-value store.
- `ucp-auth` Shared authentication service (also used by DTR for single-sign-on).
- `ucp-proxy` Controls access to the local Docker socket so that unauthenticated clients cannot make changes to the cluster.
- `ucp-swarm` Provides compatibility with the underlying Swarm.

Finally, the installation creates a couple of root CA's: one for internal cluster communications, and one for external access. They issue self-signed certs, which are fine for labs and testing, but not production.

To install UCP with certificates from a trusted CA, you will need a certificate bundle with the following three files:

- `ca.pem` Certificate of the trusted CA (usually one of your internal corporate CA's).
- `cert.pem` UCP's public certificate. This needs to contain all IP addresses and DNS names that the cluster will be accessed by — including any load-balancers that are fronting it.

- `key.pem` UCP's private key.

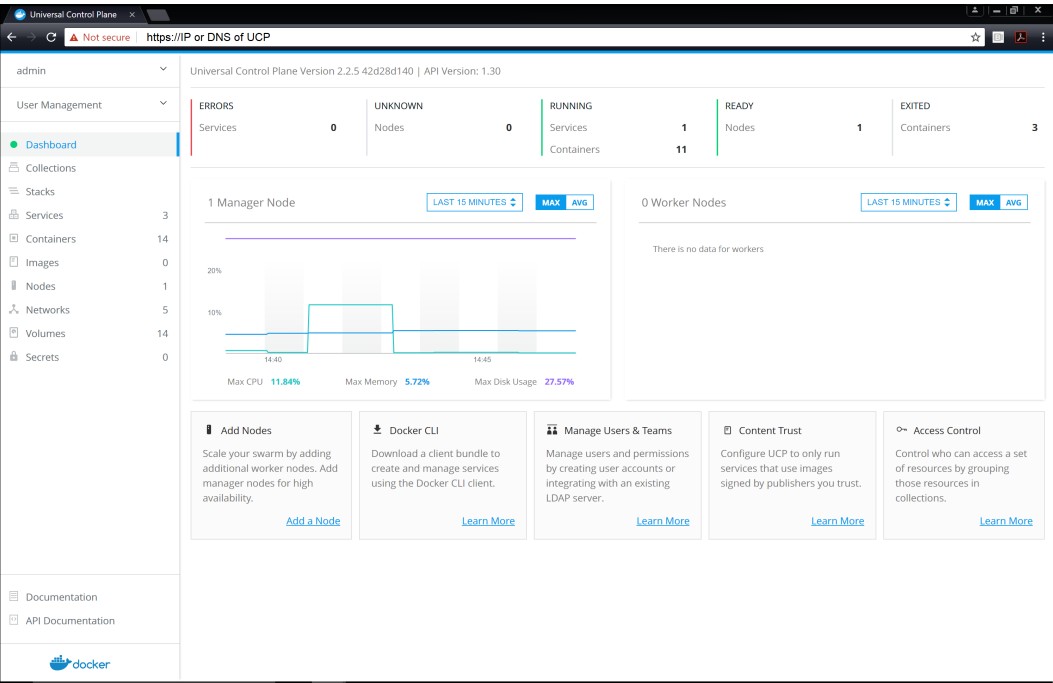
If you have these files, you need to mount them into a Docker volume called `ucp-controller-server-certs`, and use the `--external-ca` flag to specify the volume. You can also change the certificates from the Admin Settings page of the web UI after the installation.

The last thing the UCP installer outputs is the URL that you can access it from.

```
<Snip>
INFO[0049] Login to UCP at https://<IP or DNS>:443
```

Point a web browser to that address and login. If you're using self-signed certificates you'll need to accept the browser warnings. You'll also need to specify your license file, which can be downloaded from the My Content section of the Docker Store.

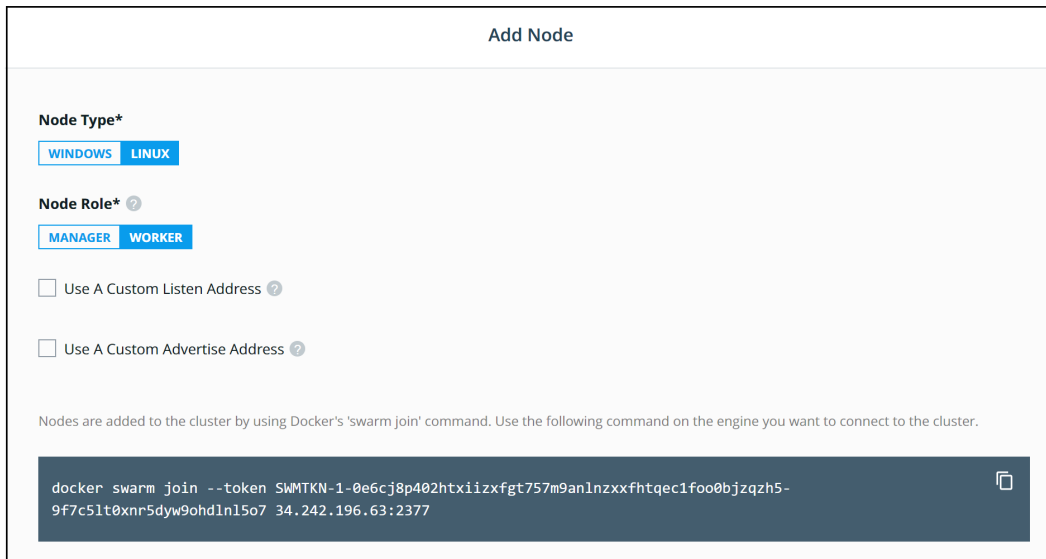
Once you're logged in, you'll be landed at the UCP Dashboard.



At this point, you have a single-node UCP cluster.

You can add more manager and worker nodes from the Add Nodes link at the bottom of the Dashboard.

Figure 16.6 shows the Add Nodes screen. You can choose to add managers or workers, and it gives you the appropriate command to run on the nodes you want to add. The example shows the command to add a Linux worker node. Notice that the command is a `docker swarm` command.



Add Node

Node Type*

WINDOWS LINUX

Node Role* ?

MANAGER WORKER

☐ Use A Custom Listen Address ?

☐ Use A Custom Advertise Address ?

Nodes are added to the cluster by using Docker's 'swarm join' command. Use the following command on the engine you want to connect to the cluster.

```
docker swarm join --token SWMTKN-1-0e6cj8p402htxiizxfgt757m9anlnzxsfhtqec1foo0bjzqzh5-9f7c51t0xnr5dyw9ohd1n15o7 34.242.196.63:2377
```

Adding a node will join it to the Swarm and configure the required UCP services on it. If you're adding managers, it's recommended to wait between each new addition. This gives Docker a chance to download and run the required UCP containers, as well as allow the cluster to register the new manager and achieve quorum.

Newly added managers are automatically configured into the highly-available (HA) Raft consensus group and granted access to the cluster store. Also, although external load-balancers aren't generally considered core parts of UCP HA, they provide a stable DNS hostname that masks what's going on behind the scenes — such as node failures.

You should configure external load-balancers for *TCP pass-through* on port 443, with a custom HTTPS health check for each UCP manager node at `https://<ucp_man-`

ager>/_ping.

Now that you have a working UCP, you should look at the options that can be configured from the Admin Settings page.

Admin Settings	
Swarm	
Certificates	
Routing Mesh	
Cluster Configuration	Cluster Configuration
Authentication & Authorization	Controller Port ⓘ
Logs	443
License	Swarm Port ⓘ
Docker Trusted Registry	2376
Docker Content Trust	Container Scheduling Strategy ⓘ
Usage	Spread
Scheduler	External Service Load Balancer ⓘ
Upgrade	
	KV Store Timeout ⓘ
	5000
	KV Store Snapshot Count ⓘ
	20000

Figure 16.7 UCP Admin Settings

The settings on this page make up the bulk of the configuration data that is backed as part of the UCP backup operation.

Controlling access to UCP

All access to UCP is controlled via the identity management sub-system. This means you need to authenticate with a valid UCP username and password before you can perform any actions on the cluster. This includes cluster administration, as well as deploying and managing services.

We've seen this already with UI — we had to log in with a username and password. But the same applies to the Docker CLI — you cannot run unauthenticated commands against UCP from the command line! This is because the local Docker socket on UCP

cluster nodes is protected by the ucp-proxy service that will not accept unauthorized commands.

Let's see it.

Client bundles

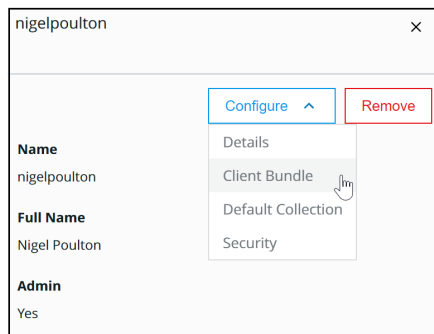
Any node running the Docker CLI is capable of deploying and managing workloads on a UCP cluster, **so long as it presents a valid certificate for a UCP user!**

In this section we'll create a new UCP user, create and download a certificate bundle for that user, and configure a Docker client to use the certificates. Once we're done, we'll explain how it works.

1. If you aren't already, login to UCP as admin.
2. Click User Management > Users and then create a new user.

As we haven't discussed roles and grants yet, make the user a Docker EE Admin.

3. With the new user still selected, click the Configure drop-down box and choose Client Bundle.



4. Click the New Client Bundle + link to generate and download a client bundle for the user.

At this point, it's important to note that client bundles are user-specific. The certificates downloaded will enable any properly configured Docker client to execute commands on the UCP cluster under the identity of the user that the bundle belongs to.

5. Copy the bundle to the Docker client that you want to configure to manage UCP.
6. Logon to the client node and perform all of the following commands from that node.
7. Unzip the contents of the bundle.

This example uses the Linux `unzip` package to unzip the contents of the bundle to the current directory. Substitute the name of the bundle to match the one in your environment.

```
$ unzip ucp-bundle-nigelpoulton.zip
Archive:  ucp-bundle-nigelpoulton.zip
extracting: ca.pem
extracting: cert.pem
extracting: key.pem
extracting: cert.pub
extracting: env.sh
extracting: env.ps1
extracting: env.cmd
```

As the output shows, the bundle contains the required `ca.pem`, `cert.pem`, and `key.pem` files. It also includes scripts that will configure the Docker client to use the certificates.

8. Use the appropriate script to configure the Docker client. `env.sh` works on Linux and Mac, `env.ps1` and `env.cmd` work on Windows.

You'll probably need administrator/root privileges to run the scripts.

The example works on Linux and Mac.

```
$ eval "$(env.sh)"
```

At this point, the client node is fully configured.

9. Test access.

```
$ docker version
```

<Snip>

Server:

```
Version:      ucp/2.2.5
API version:  1.30 (minimum version 1.20)
Go version:   go1.8.3
Git commit:   42d28d140
Built:        Wed Jan 17 04:44:14 UTC 2018
OS/Arch:      linux/amd64
Experimental: false
```

Notice that the server portion of the output shows the version as `ucp/2.2.5`. This proves the Docker client is successfully talking to the daemon on a UCP node!

Under-the-hood, the script configures three environment variables:

- `DOCKER_HOST`
- `DOCKER_TLS_VERIFY`
- `DOCKER_CERT_PATH`

`DOCKER_HOST` points the client to the remote Docker daemon on the UCP controller. An example might look like this `DOCKER_HOST=tcp://34.242.196.63:443`. As we can see, access via port 443.

`DOCKER_TLS_VERIFY` is set to 1, telling the client to use TLS verification in *client mode*.

`DOCKER_CERT_PATH` tells the Docker client where to find the certificate bundle.

The net result is all docker commands from the client will be signed by the user's certificate and sent across the network to the remote UCP manager. This is shown in Figure 16.9.

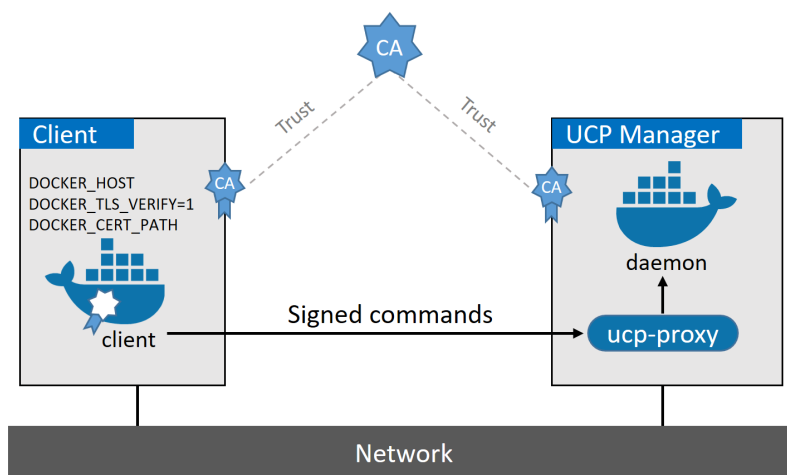


Figure16.9

Let's switch tack and see how we backup and recover UCP.

Backing up UCP

First and foremost, high availability (HA) is not the same as a backup!

Consider the following example. You have a highly available UCP cluster with 5 managers nodes. All manager nodes are healthy and the control plane is replicating. A dissatisfied employee corrupts the cluster (or deletes all user accounts). This *corruption* is automatically replicated to all 5 manager nodes, rendering the cluster broken. There is no way that HA can help you in this situation. What you need, is a backup!

A UCP cluster is made from three major components that need backing up separately:

- Swarm
- UCP
- Docker Trusted Registry (DTR)

We'll walk you through the process of backing up Swarm and UCP, and we'll show you how to back up DTR later in the chapter.

Although UCP sits on top of Swarm, they are separate components. Swarm holds all of the node membership, networks, volumes, and service definitions. UCP sits on top and maintains its own databases and volumes that hold things such as users, groups, grants, bundles, license files, certificates, and more.

Let's see how to **backup Swarm**.

Swarm configuration and state is stored in `/var/lib/docker/swarm`. This includes Raft log keys, and it's replicated to every manager node. A Swarm backup is a copy of all the files in this directory.

Because it's replicated to every manager, you can perform the backup from any manager.

You need to stop Docker on the node that you want to perform the backup on. This means it's probably not a good idea to perform the backup on the leader manager, as a leader election will be instigated. You should also perform the backup at a quiet time for the business — even though stopping Docker on a single manager node isn't a problem in a multi-manager Swarm, it can increase the risk of the cluster losing quorum if another manager fails during the backup.

Before proceeding, you might want to create a couple of Swarm objects so that you can prove the backup and restore operation work. The example Swarm we'll be backing up in these examples has an overlay network called `vantage-net` and a Swarm service called `vantage-svc`.

1. Stop Docker on the Swarm manager node you are performing the backup from.

This will stop all UCP containers on the node. If UCP is configured for HA, the other managers will make sure the control plane remains available.

```
$ service docker stop
```

2. Backup the Swarm config.

The example uses the Linux `tar` utility to perform the file copy. Feel free to use a different tool.

```
$ tar -czvf swarm.bkp /var/lib/docker/swarm/  
tar: Removing leading `/' from member names  
/var/lib/docker/swarm/  
/var/lib/docker/swarm/docker-state.json  
/var/lib/docker/swarm/state.json  
<Snip>
```

3. Verify that the backup file exists.

```
$ ls -l  
-rw-r--r-- 1 root  root  450727 Jan 29 14:06 swarm.bkp
```

You should rotate, and store the backup file off-site according to your corporate backup policies.

4. Restart Docker.

```
$ service docker restart
```

Now that Swarm is backed up, it's time to **backup UCP**.

A few notes on backing up UCP before we start.

The UCP backup job runs as a container, so Docker needs to be running for the backup to work.

You can run the backup from any UCP manager node in the cluster, and you only need to run the operation on one node (UCP replicates its configuration to all manager nodes, so backing up from multiple nodes is not required).

Backing up UCP will stop all UCP containers on the manager that you're executing the operation on. With this in mind, you should be running a highly available UCP cluster, and you should run the operation at a quiet time for the business.

Finally, user workloads running on the manager node will not be stopped. However, it is not recommended to run user workloads on UCP managers.

Let's backup UCP.

Perform the following command on a UCP manager node. Docker will need to be running on the node.

```
$ docker container run --log-driver none --rm -i --name ucp \  
-v /var/run/docker.sock:/var/run/docker.sock \  
docker/ucp:2.2.5 backup --interactive \  
--passphrase "Password123" > ucp.bkp
```

It's a long command, so let's step through it.

The first line is a standard `docker container run` command that tells Docker to run a container with no log driver, to remove it when the operation is complete, and to call it `ucp`. The second line mounts the *Docker socket* into the container so that the container has access to the Docker API to stop containers etc. The third line tells Docker to run a `backup --interactive` command inside of a container based on the `docker/ucp:2.2.5` image. The final line creates an encrypted file called `ucp.bkp` and protects it with a password.

A few points worth noting.

It's a good idea to be specific about the version (tag) of the UCP image to use. This example specifies `docker/ucp:2.2.5`. One of the reasons for being specific, is that it's recommended to run backup and restore operations with the same version of the image. If you don't explicitly state which image to use, Docker will use the one tagged as `latest`, which might be different between the time you run the backup command and the time you run the restore.

You should always use the `--passphrase` flag to protect the contents of the backup, and you should definitely use a better password than the one in the example :-D

You should catalogue and make off-site copies of the backup file according to your corporate backup policies. You should also configure a backup schedule and job verification.

Now that Swarm and UCP are backed up, you can safely recover them in the event of disaster. Speaking of which....

Recovering UCP

We need to be clear about one thing before we get into the weeds of recovering UCP: Restoring from backup is a last resort, and should only be used when the cluster has been corrupted or all manager nodes have been lost!

You do not need to recover from a backup if you've lost a single manager in an HA cluster. In that case, you can easily add a new manager and it'll join the cluster.

We'll show how to recover Swarm from a backup, and then UCP.

Perform the following tasks from the Swarm/UCP manager node that you wish to recover.

1. Stop Docker.

```
$ service docker stop
```

2. Delete any existing Swarm configuration.

```
$ rm -r /var/lib/docker/swarm
```

3. Restore the Swarm configuration from backup.

In this example, we'll restore from a zipped tar file called `swarm.bkp`. Restoring to the root directory is required with this command as it will include the full path to the original files as part of the extract operation. This may be different in your environment.

```
$ tar -zxvf swarm.bkp -C /
```

4. Initialize a new Swarm cluster.

Remember, you are not recovering a manager and adding it back to a working cluster. This operation is to recover a failed Swarm that has no surviving managers. The `--force-new-cluster` flag tells Docker to create a new cluster using the configuration stored in `/var/lib/docker/swarm` on the current node.

```
$ docker swarm init --force-new-cluster  
Swarm initialized: current node (jhsg...3l9h) is now a manager.
```

5. Check that the network and service were recovered as part of the operation.

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
snkqjy0chtd5	vantage-net	overlay	swarm


```
$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE
w9dimu8jfrze	vantage-svc	replicated	5/5	alpine:latest

Congratulations. The Swarm is recovered.

6. Add new manager and worker nodes to the Swarm, and take a fresh backup.

With Swarm recovered, you can now **recover UCP**.

In this example, UCP was backed up to a file called `ucp.bkp` in the current directory. Despite the name of the backup file, it is a Linux tarball.

Run the following commands from the node that you want to recover UCP on. This can be the node that you just recovered Swarm on.

1. Remove any existing, and potentially corrupted, UCP installations.

```
$ docker container run --rm -it --name ucp \
-v /var/run/docker.sock:/var/run/docker.sock \
docker/ucp:2.2.5 uninstall-ucp --interactive
```

```
INFO[0000] Your engine version 17.06.2-ee-6, build e75fdb8 is compatible
INFO[0000] We're about to uninstall from this swarm cluster.
Do you want to proceed with the uninstall? (y/n): y
INFO[0000] Uninstalling UCP on each node...
INFO[0009] UCP has been removed from this cluster successfully.
INFO[0011] Removing UCP Services
```

2. Restore UCP from the backup.

```
$ docker container run --rm -i --name ucp \  
-v /var/run/docker.sock:/var/run/docker.sock \  
docker/ucp:2.2.5 restore --passphrase "Password123" < ucp.bkp  
  
INFO[0000] Your engine version 17.06.2-ee-6, build e75fdb8 is compatible  
<Snip>  
time="2018-01-30T10:16:29Z" level=info msg="Parsing backup file"  
time="2018-01-30T10:16:38Z" level=info msg="Deploying UCP Agent Service"  
time="2018-01-30T10:17:18Z" level=info msg="Cluster successfully restored."
```

3. Log on to the UCP web UI and ensure that the user created earlier is still present (or any other UCP objects that previously existed in your environment).

Congrats. You now know how to backup and recover Docker Swarm and Docker UCP.

Let's shift our attention to Docker Trusted Registry.

Docker Trusted Registry (DTR)

Docker Trusted Registry, which we're going to refer to as DTR, is a secure, highly available on-premises Docker registry. If you know Docker Hub, think of DTR as a private Docker Hub that you can install on-premises and manage yourself.

In this section, we'll show how to install it in an HA configuration, and how to back it up and perform recovery operations. We'll show how DTR implements advanced features in the next chapter.

Let's mention a few important things before getting your hands dirty with the installation.

If possible, you should run your DTR instances on dedicated nodes. You definitely shouldn't run user workloads on your production DTR nodes.

As with UCP, you should run an odd number of DTR instances. 3 or 5 is best for fault tolerance. A recommended configuration for a production environment might be:

- 3 dedicated UCP managers

- 3 dedicated DTR instances
- However many worker nodes your application requirements demand

Let's install and configure a single DTR instance.

Install DTR

The next few steps will walk through the process of configuring the first DTR instance in a UCP cluster.

To follow along, you'll need a UCP node that you will install DTR on, and a load balancer configured to listen on port 443 in TCP passthrough mode with a health check configured for /health on port 443. Figure 16.10 shows a high-level diagram of what we'll build.

Configuring a load balancer is beyond the scope of this book, but the diagram shows the important DTR-related configuration requirements.

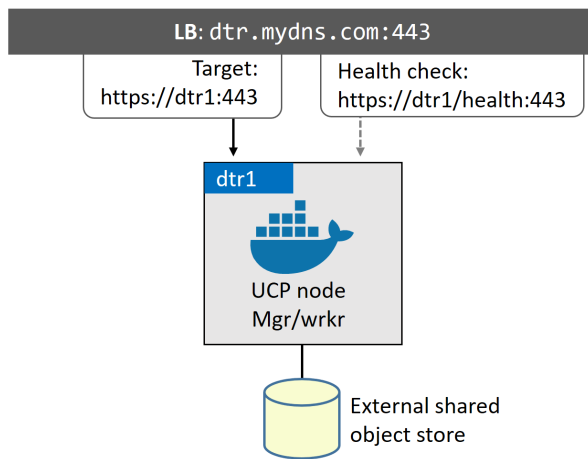


Figure 16.10 High level single-instance DTR config.

1. Log on to the UCP web UI and click Admin > Admin Settings > Docker Trusted Registry.
2. Fill out the DTR configuration form.
 - DTR EXTERNAL URL: Set this to the URL of your external load balancer.

- UCP NODE: Select the name of the node you wish to install DTR on.
 - Disable TLS Verification For UCP: Check this box if you're using self-signed certificates.
3. Copy the long command at the bottom of the form.
 4. Paste the command into any UCP manager node.

The command includes the `--ucp-node` flag telling UCP which node to perform the install on.

The following is an example DTR install command that matches the configuration in Figure 16.10. It assumes that you already have a load balancer configured at `dtr.mydns.com`

```
$ docker run -it --rm docker/dtr install \  
--dtr-external-url dtr.mydns.com \  
--ucp-node dtr1 \  
--ucp-url https://34.252.195.122 \  
--ucp-username admin --ucp-insecure-tls
```

You will need to provide the UCP admin password to complete the installation.

5. Once the installation is complete, point your web browser to your load balancer. You will be automatically logged in to DTR.

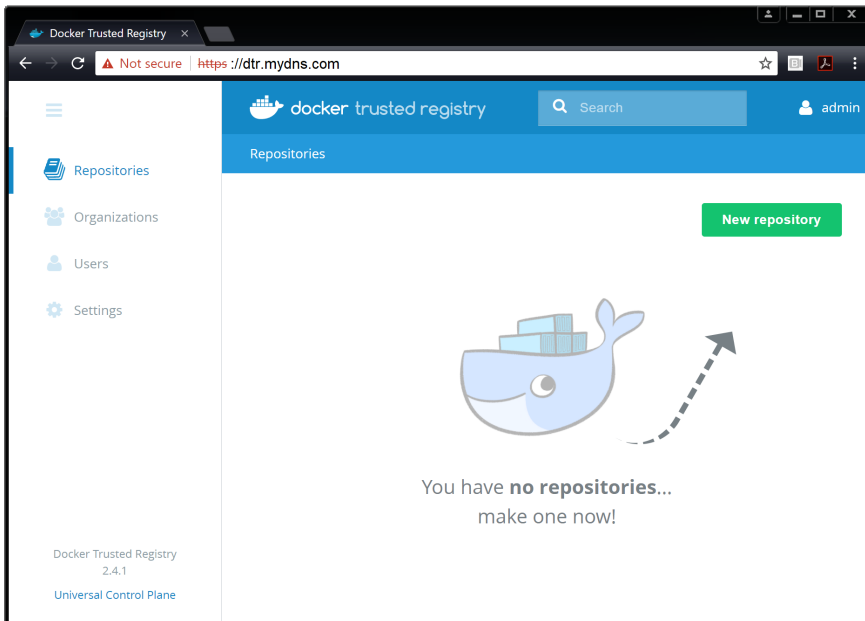


Figure 16.11 DTR home page

DTR is ready to use. But it's not configured for HA.

Configure DTR for high availability

Configuring DTR with multiple replicas for HA requires a shared storage backend. This can be NFS or object storage, and can be on-premises or in the public cloud. We'll walk through the process of configuring DTR for HA using an Amazon S3 bucket as the shared backend.

1. Log on to the DTR console and navigate to Settings.
2. Select the Storage tab and configure the shared storage backend.

Figure 16.12 shows DTR configured to use an AWS S3 bucket called `deep-dive-dtr` in the `eu-west-1` AWS availability zone. You will not be able to use this example.

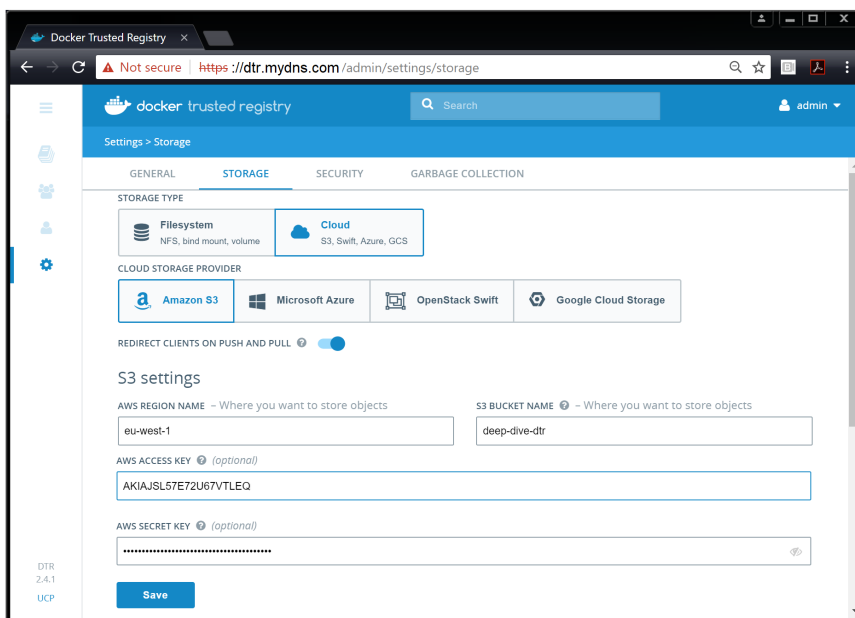


Figure 16.12 DTR Shared Storage configuration for AWS

DTR is now configured with a shared storage backend and ready to have additional replicas.

1. Run the following command from a manager node in the UCP cluster.

```
$ docker run -it --rm \
  docker/dtr:2.4.1 join \
  --ucp-node dtr2 \
  --existing-replica-id 47f20fb864cf \
  --ucp-insecure-tls
```

The `--ucp-node` flag tells the command which node to add the new DTR replica on. The `--insecure-tls` flag is required if you're using self-signed certificates.

You will need to substitute the version of the image and the replica ID. The replica ID was displayed as part of the output when you installed the initial replica.

2. Enter the UCP URL and port, as well as admin credentials when prompted.

When the join is complete, you will see some messages like the following.

```
INFO[0166] Join is complete
INFO[0166] Replica ID is set to: a6a628053157
INFO[0166] There are currently 2 replicas in your DTR cluster
INFO[0166] You have an even number of replicas which can impact availability
INFO[0166] It is recommended that you have 3, 5 or 7 replicas in your cluster
```

Be sure to follow the advice and install additional replicas so that you operate an odd number.

You may need to update your load balancer configuration so that it balances traffic across the new replicas.

DTR is now configured for HA. This means you can lose a replica without impacting the availability of the service. Figure 16.13 shows an HA DTR configuration.

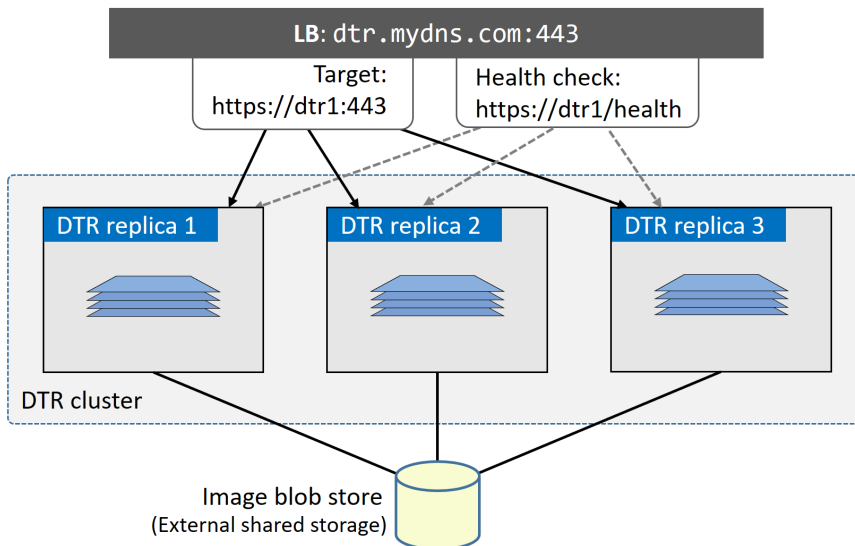


Figure 16.13 DTR HA

Notice that the external load balancer is sending traffic to all three DTR replicas, as well as performing health checks on all three. All three DTR replicas are also sharing the same external shared storage backend.

In the diagram, the load balancer and the shared storage backend are 3rd party products and depicted as singletons (not HA). In order to keep the entire environment as highly available as possible, you should ensure they have native HA, and that you back up their contents and configurations as well (e.g. make sure the load balancer and storage systems are natively HA, and perform backups of them).

Backup DTR

As with UCP, DTR has a native backup command that is part of the Docker image that was used to install the DTR. This native backup command will backup the DTR configuration that is stored in a set of named volumes, and includes:

- DTR configuration
- Repository metadata
- Notary data
- Certificates

Images are not backed up as part of a native DTR backup. It is expected that images are stored in a highly available storage backend that has its own independent backup schedule using non-Docker tools.

Run the following command from a UCP manager node to perform a DTR backup.

```
$ read -sp 'ucp password: ' UCP_PASSWORD; \  
  docker run --log-driver none -i --rm \  
  --env UCP_PASSWORD=$UCP_PASSWORD \  
  docker/dtr:2.4.1 backup \  
  --ucp-insecure-tls \  
  --ucp-username admin \  
> ucp.bkp
```

Let's explain what the command is doing.

The `read` command will prompt you to enter the password for the UCP admin account, and will store it in a variable called `UCP_PASSWORD`. The second line tells Docker to start a new temporary container for the operation. The third line makes the UCP password available inside the container as an environment variable. The fourth line issues the backup command. The fifth line makes it work with self-signed certificates. The sixth line sets the UCP username to "admin". The last line directs the backup to a file in the current directory called `ucp.bkp`.

You will be prompted to enter the UCP URL as well as a replica ID. You can specify these as part of the backup command, I just didn't want to explain a single command that was 9 lines long!

When the backup is finished, you will have a file called `ucp.bkp` in your working directory. This should be picked up by your corporate backup tool and managed in-line with your existing corporate backup policies.

Recover DTR from backups

Restoring DTR from backups should be a last resort, and only attempted when the majority of replicas are down and the cluster cannot be recovered any other way. If you have lost a single replica and the majority are still up, you should add a new replica using the `dtr join` command.

If you are sure you have to restore from backup, the workflow is like this:

1. Stop and delete DTR on the node (might already be stopped)
2. Restore images to the shared storage backend (might not be required)
3. Restore DTR

Run the following commands from the node that you want to restore DTR to. This node will obviously need to be a member of the same UCP cluster that the DTR is a member of. You should also use the same version of the `docker/dtr` image that was used to create the backup.

1. Stop and delete DTR.

```
$ docker run -it --rm \
  docker/dtr:2.4.1 destroy \
  --ucp-insecure-tls

INFO[0000] Beginning Docker Trusted Registry replica destroy
ucp-url (The UCP URL including domain and port): https://34.252.195.122:443
ucp-username (The UCP administrator username): admin
ucp-password:
INFO[0020] Validating UCP cert
INFO[0020] Connecting to UCP
INFO[0021] Searching containers in UCP for DTR replicas
INFO[0023] This cluster contains the replicas: 47f20fb864cf a6a628053157
Choose a replica to destroy [47f20fb864cf]:
INFO[0030] Force removing replica
INFO[0030] Stopping containers
INFO[0035] Removing containers
INFO[0045] Removing volumes
INFO[0047] Replica removed.
```

You'll be prompted to enter the UCP URL, admin credentials, and replica ID that you want to delete.

If you have multiple replicas, you can run the command multiple times to remove them all.

2. If the images were lost from the shared backend, you will need to recover them. This step is beyond the scope of the book as it can be specific to your shared storage backend.
3. Restore DTR with the following command.

You will need to substitute the values on lines 5 and 6 with the values from your environment. Unfortunately the restore command cannot be ran interactively, so you cannot be prompted for values once the restore has started.

```
$ read -sp 'ucp password: ' UCP_PASSWORD; \
docker run -i --rm \
--env UCP_PASSWORD=$UCP_PASSWORD \
docker/dtr:2.4.1 restore \
--ucp-url <ENTER_YOUR_ucp-url> \
--ucp-node <ENTER_DTR_NODE_hostname> \
--ucp-insecure-tls \
--ucp-username admin \
< ucp.bkp
```

DTR is now recovered.

Congratulations. You now know how to backup and recover; Swarm, UCP, and DTR. Time for one final thing before wrapping up the chapter — network ports!

UCP managers, workers, and DTR nodes need to be able to communicate over the network. Figure 16.14 summarizes the port requirements.

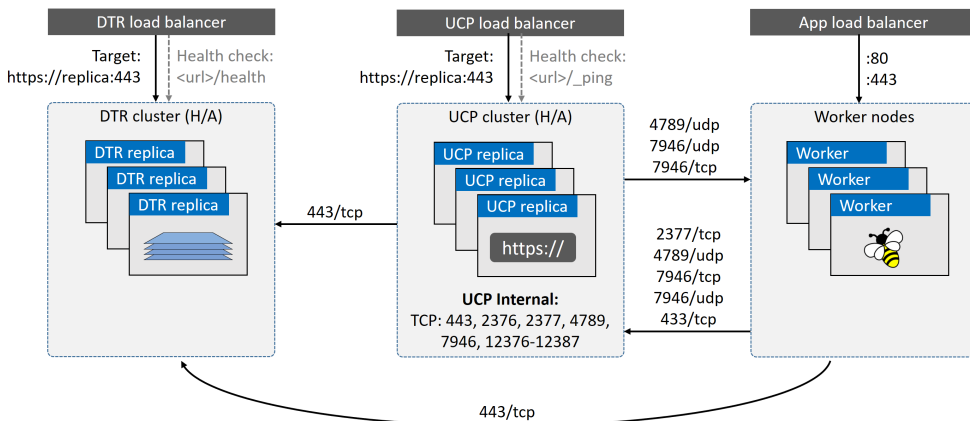


Figure 16.14 UCP cluster network port requirements

Chapter Summary

Docker Enterprise Edition (EE) is a suite of products that form an “*enterprise friendly*” container-as-a-service platform. It comprises a hardened Docker Engine,

an Operations UI, and a secure registry. All of which can be deployed on-premises and managed by the customer. It's even bundled with a support contract.

Docker Universal Control Plane (UCP) provides a simple-to-use web UI focussed at traditional enterprise Ops teams. It supports native high availability (HA) and has tools to perform backup and restore operations. Once up and running, it provides a whole suite of enterprise-grade features that we'll discuss in the next chapter.

Docker Trusted Registry (DTR) sits on top of UCP and provides a highly available secure registry. Like UCP, this can be deployed on-premises within the safety of the corporate *"firewall"*, and provides native tools for backup and recovery.

17: Enterprise-grade features

This chapter follows on from the previous chapter, and covers some of the enterprise-grade features provided by Docker Universal Control Plane (UCP) and Docker Trusted Registry (DTR).

We'll be assuming you've read the previous chapter, so know how to install and configure them, as well as perform backup and recovery operations.

We'll split this chapter into two parts:

- The TLDR
- The deep dive

Enterprise-grade features - The TLDR

Enterprises want to use Docker and containers, but they need things packaged and supported like a real enterprise app. They also need things like role-based access control and integration with enterprise directory services like Active Directory. This is where *Docker Enterprise Edition* comes into play.

Docker Enterprise Edition is a hardened version of the Docker engine, an Ops UI, a secure registry, and a bunch of enterprise-focussed features. You can deploy it on-premises or in the cloud, you manage it yourself, and you can get it with a support contract.

In summary, it's a container-as-a-service platform that you can run in the safety of your own corporate data center.

Enterprise-grade features - The Deep Dive

We'll divide this main section of the chapter as follows:

- Role-based access control (RBAC)
- Active Directory integration
- Docker Content Trust (DCT)
- Configuring Docker Trusted Registry (DTR)
- Using Docker Trusted Registry
- Image promotions
- HTTP Routing Mesh (HRM)

Role-based access control (RBAC)

I've spent the majority of the last 10 years of my career running IT in the financial services sector. Two checkboxes that are mandatory at most places I worked, are role-based access control (RBAC) and Active Directory (AD) integration. If you were trying to sell us a product, and it didn't have these two features, we wouldn't buy it!

Fortunately, Docker EE has both. In this section we'll talk about RBAC.

UCP implements RBAC via something called a *grant*. At a high level, a grant is made of three things:

- **Subject**
- **Role**
- **Collection**

The *subject* is one or more users or a team. The *role* is the set of permissions, and the *collection* is the resources these permissions apply to. See Figure 17.1.

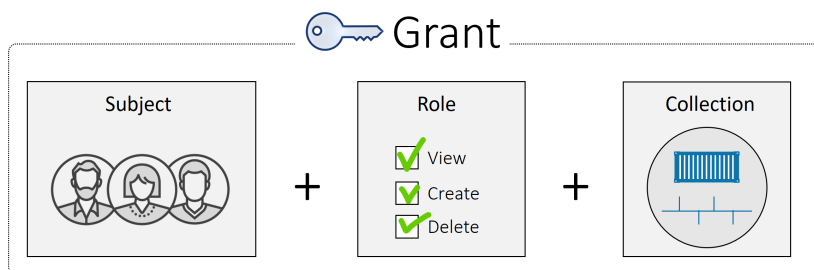
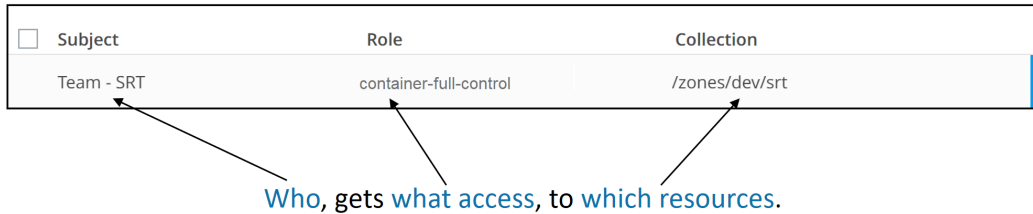


Figure 17.1 Grant

Figure 17.2, shows an example where the SRT team has container-full-control access to all resources in the /zones/dev/srt collection.



☐ Subject	Role	Collection
Team - SRT	container-full-control	/zones/dev/srt

Who, gets what access, to which resources.

Figure 17.2

Let's complete the following steps to create a grant:

- Create users and teams
- Create a custom role
- Create a collection
- Create a grant

Only UCP Admins can create and manage users, teams, roles, collections, and grants. So to follow along, you'll need to be logged in as a UCP admin.

Create users and teams

It's a best practice to group users into teams, and assign teams to grants. You *can* assign individual users to a *grant*, but it's not recommended.

Let's create some users and teams.

1. Log in to UCP.
2. Expand User Management and click Users.

From here you can create users.

3. Click Organization & Teams.

From here you can create organizations. For the examples in the next few steps, we'll be using an organization called "manufacturing".

4. Click the manufacturing organization and create a team.

Teams exist within an organization. It's not possible to create a team that isn't part of an organization, and a team can only be a member of one organization.

5. Add users to a team.

To add a user to a team, you need to click into the team and choose Add Users from the Actions menu.

Figure 17.3 shows how to add users to the SRT team in the manufacturing organization.

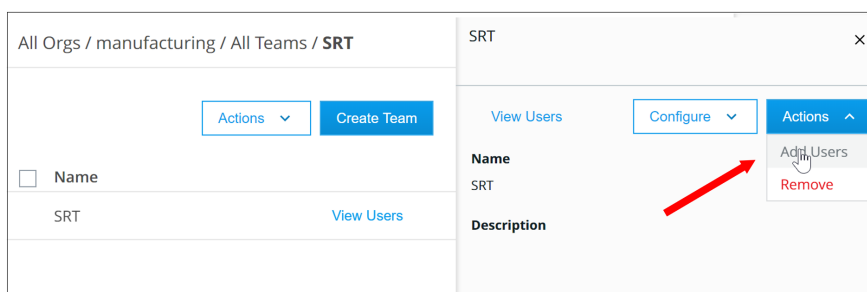


Figure 17.3 Adding users to teams

You now have some users and teams. UCP shares its user database with DTR, meaning any users and teams you create in UCP are also available in DTR.

Create a custom role

Custom roles are powerful, they let you get extremely granular with the permissions you assign. In this step we'll create a new custom role called `secret-ops` that allows subjects to create, delete, update, use, and view Docker secrets.

1. Expand the User Management tab of the left-hand navigation pane and select Roles.
2. Create a new role.
3. Give the role a name.

In this example we're going to create a new custom role called "secret-ops" with permission to perform all secret-related operations.

4. Select **Operations** and explore the list of operations that can be assigned to the role.

The list is long, and allows you to specify individual API operations.

5. Select the individual API operations you want to assign to the role.

In the example, we'll assign all secret-related API operations.

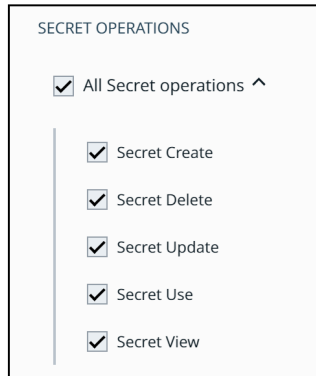


Figure 17.4 Assigning API operations to a custom role

6. Click **Create**.

The role is now on the system and can be assigned to multiple grants.

Let's create a collection.

Create a collection

In the previous chapter we learned that networks, volumes, secrets, services, and nodes are Swarm resources — they get stored in the Swarm config at `/var/lib/docker/swarm`. *Collections* let you group these in ways that match your organizational structure and IT requirements. For example, your IT infrastructure might be divided into three zones; prod, test, and dev. If this is the case, you could create three collections and assign resources to each, as shown in Figure 17.5.

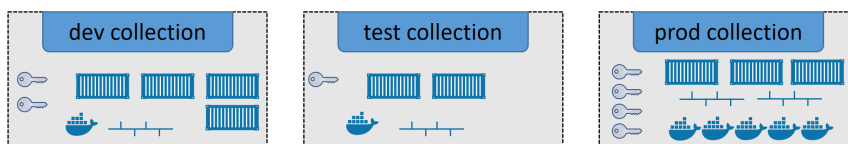


Figure 17.5 High-level collections

Each resource can only be in one collection.

In the next steps, we'll create a new collection called `zones/dev/srt` and assign a secret to it. Collections are hierarchical by nature, so you'll need to create three nested collections like this: `zones > dev > srt`.

Perform all of the following steps from the Docker UCP web UI.

1. Select `Collections` from the left navigation pane, and choose `Create Collection`.
2. Create the root collection called `zones`.
3. Click `View Children` for the `/zones` collection.
4. Create a nested child collection called `dev`.
5. Click `View Children` for the `/zones/dev` collection.
6. Create the final nested child collection called `srt`.

You now have a collection called `/zones/dev/srt`. However, it's currently empty. In the next steps we'll add a *secret* to it.

1. Create a new secret.

You can create it from the command line or the UCP web UI. We'll explain the web UI method.

From the UCP web UI click: `Secrets > Create Secret`. Give it a name, some data and click `Save`.

It's possible to configure the *collection* at the same time you create the secret. But we're not doing it that way.

2. Locate and select the secret in the UCP web UI.
3. Click `Collection` from the `Configure` drop-down menu.

4. Navigate through the View Children hierarchy until the `/zones/dev/srt` collection is selected and click Save.

The secret is now part of the `/zones/dev/srt` collection. It cannot be a member of any other collections.

One final thing about *collections* before we create the *grant*. Collections have an inheritance model where access to any collection automatically implies access to nested child collections. In Figure 17.6, the `dev` team has access to the `/zones/dev` collection, and as such, it automatically gets access to the resources in the `srt`, `hellcat` and `daemon` child collections.

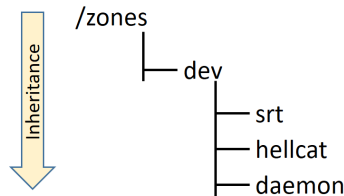
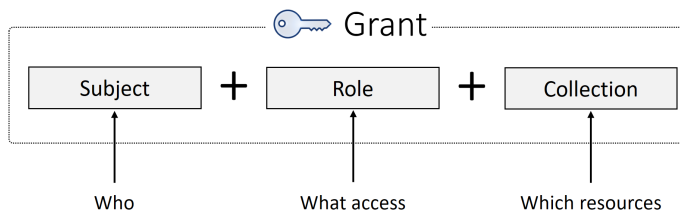


Figure 17.6 Collection inheritance

Create a grant

Now that you have users and teams, a custom role, and a collection, you can create a grant. In this example, we'll create a grant for the `srt-dev` team to have the custom `secret-ops` role against all resources in the `/zones/dev/srt` collection.

Grants are about *who*, gets *what access*, to *which resources*.



1. Expand the User Management tab on the left navigation pane and click Grants.

2. Create a new grant.
3. Click Subject and choose the SRT team from the manufacturing organization.

It’s possible to select an entire organization. If you do this, all teams within the organization will be included in the grant.
4. Click Role and select the custom secret-ops role.
5. Click Collections and select the /zones/dev/srt collection.

You may have to view the children of the top-level Swarm collection before you see /zones.

6. Click Save to create the grant.

The grant is now created and can be viewed in the list of all grants on the system. Members of the manufacturing/SRT team can now perform all secret-related operations on resources in the /zones/dev/srt collection.

☐ Subject	Role	Collection
Org - docker-datacenter	Scheduler	/
admin	Restricted Control	/Shared/Private/admin
Team - SRT	secret-ops	/zones/dev/srt

Who, gets what access, to which resources.

You can modify components of a grant while the grant is live. For example, you can add users to the team, and resources to the collection. But you cannot alter the API operations assigned to the role. If you want to change the permissions of the role, you will need to create a new one with the desired permissions.

RBAC for nodes

One final thing on RBAC. It’s possible to group the worker nodes in your cluster for scheduling purposes. For example, you might run a single cluster for dev, testing, and QA workloads — a single cluster might reduce admin overheads and make it easier to assign nodes to the three different environments. But you might also want

worker nodes divided up so that only members of the dev team can schedule work onto nodes in the dev collection etc.

As you'd expect, you accomplish this with *grants*. First of all, you'd assign UCP Worker nodes to a custom *collection*. Then you'd create a grant comprising the collection, the built-in Scheduler *role*, and the team that you want to assign the grant to. This lets you control which users can schedule work to which nodes in the cluster.

As a simple example, the grant shown in Figure 17.9 will allow members of the dev team to be able to schedule services and containers onto worker nodes in the /zones/dev collection.

☐ Subject	Role	Collection
Team - dev	Scheduler	/zones/dev

Who, gets what access, to which resources.

Figure 17.9 RBAC for nodes

That's it! You know how to implement RBAC in Docker UCP!

Active Directory integration

Like all good enterprise tools, UCP integrates with Active Directory and other LDAP directory services. This allows it to leverage existing users and groups from your organization's established single-sign-on system.

Before going any further in this section, it is vital that you discuss any AD/DLAP integration plans with the teams responsible for directory services in your organization. Get them involved from the start, so that your planning and implementation can be as smooth as possible!

Out-of-the-box, UCP user and group data is stored in a local database that's leveraged by DTR for a single-sign-on (SSO) experience. This authenticates all access requests locally, and allows you to login to DTR without having to enter your UCP credentials again. However, **UCP admins** can configure UCP to leverage existing corporate user

accounts stored in AD or other LDAP directory services — offloading authentication and account management to existing teams and processes.

The following procedure will show you how to configure UCP to leverage AD for user accounts. At a high level, the process tells UCP to search for user accounts, in a specific directory, and copy them into UCP. As previously stated, co-ordinate this work with your directory services team.

Let's do it.

1. Expand the Admin drop-down in the left navigation pane and select Admin Settings.
2. Select Authentication & Authorization and click Yes under the **LDAP Enabled** heading.
3. Configure LDAP Server settings.

At a high level, you can think of the LDAP Server Settings as *where to search*. E.g. which directories to look in for user accounts.

The values entered here will be specific to your environment.

LDAP Server URL is the name of an LDAP server in the domain you'll be searching for accounts in. For example, `ad.mycompany.internal`.

Reader DN and **Reader Password** are the credentials for an account in the directory with permission to search it. The account must exist in, or be trusted by, the directory you are searching. It's best practice for it to have *read-only* permissions in the directory.

You can use the Add LDAP Domain + button to add additional domains to search. Each one needs its own LDAP Server URL and Reader account.

4. Configure LDAP User Search Configuration.

If LDAP Server Settings is *where to search*, then LDAP User Search Configuration is *what to search for*.

Base DN Specifies the LDAP node to start the search from.

Username Attribute is the LDAP attribute to use as the UCP username.

Full Name Attribute is the LDAP Attribute to use as the full name of the UCP account.

See the documentation for other more advanced settings. You should also consult with your directory services team when configuring LDAP integration.

5. Once you've configured the LDAP settings, UCP will search for matching users and create them in the UCP user database. It will then perform periodic sync operations according to the Sync Interval (Hours) setting.

If you checked the Just-In-Time User Provisioning box, UCP will defer the creation of user accounts until each account's first login event.

6. Before clicking Save, you should always perform a test login under the LDAP Test Login section.

The test login needs to be with a valid user account in the LDAP system you're configuring UCP To use. The test will apply all of the configuration values defined in the sections above (the LDAP config you're about to save).

Only save the configuration if the test login succeeds.

7. Save the configuration.

At this point, UCP will search the LDAP system and create the user accounts matching the Base DN and other criteria provided.

Local user accounts created prior to configuring LDAP will still be present on the system and can still be used.

Docker Content Trust (DCT)

In the modern IT world, *trust* is a big deal! And going forward, it's going to get bigger. Fortunately, Docker implements trust through a feature called Docker Content Trust (DCT).

At a very high level, publishers of Docker images can sign their images when pushing them to a repo. Consumers can then verify them when they pull them, or perform build and run operations. To cut a long story short, DCT enables consumers to guarantee they're getting what they ask for!

Figure 17.10 shows the high level architecture.

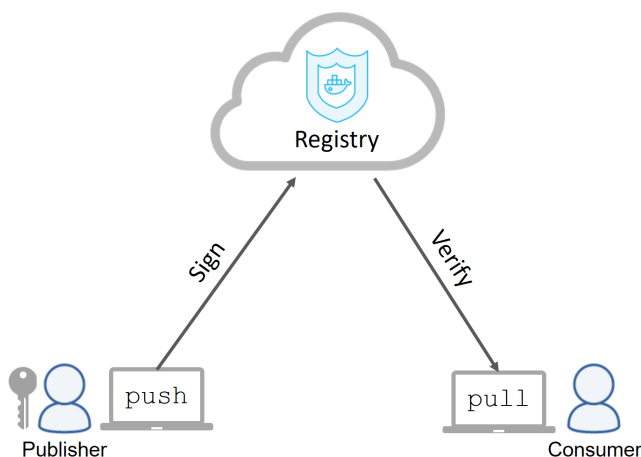


Figure 17.10 High level DCT architecture

DCT implements *client-side* signing and verification operations, meaning the Docker client performs them.

While it's obvious that cryptographic guarantees like this are important when pulling and pushing software across the internet, it's increasingly important at every level of the stack and at every step in the software delivery pipeline. Hopefully it won't be long before all aspects of the delivery chain are infused with cryptographic trust guarantees.

Let's walk through a quick example of configuring DCT and seeing it in action.

You'll need a single Docker client and a repository that you can push an image to. A repository on Docker Hub will work.

DCT is turned on and off via the `DOCKER_CONTENT_TRUST` environment variable. Setting it to a value of "1" will turn DCT **on** in your current session. Setting it to any other value will turn it **off**. The following example will turn it on a Linux-based Docker host.

```
$ export DOCKER_CONTENT_TRUST=1
```

All future `docker push` commands will automatically sign images as part of the push operation. Likewise, all `pull`, `build`, and `run` commands will only work if the image they are acting on is signed.

Let's push an image to a repo with a new tag.

The image being pushed can be any image. In fact, the one I'm using is the current `alpine:latest` that I just pulled a minute ago. At the moment, it's not signed by me!

1. Tag the image so it can be pushed to your desired repo. I'm going to push it to a new repo within the namespace of my personal Docker Hub account.

```
$ docker image tag alpine:latest nigelpoulton/dockerbook:v1
```

2. Login to Docker Hub (or another registry) so you can push the image in the next step.

```
$ docker login
```

Login with your Docker ID to push and pull images from Docker Hub.

Username: nigelpoulton

Password:

Login Succeeded

3. Push the newly tagged image.

```
$ docker image push nigelpoulton/dockerbook:v1
```

The push refers to a repository [docker.io/nigelpoulton/dockerbook]

cd7100a72410: Mounted from library/alpine

v1: digest: sha256:8c03...acbc size: 528

Signing and pushing trust metadata

<Snip>

Enter passphrase for new root key with ID 865e4ec:

Repeat passphrase for new root key with ID 865e4ec:

Enter passphrase for new repository key with ID bd0d97d:

Repeat passphrase for new repository key with ID bd0d97d:

Finished initializing "docker.io/nigelpoulton/sign"

Successfully signed "docker.io/nigelpoulton/sign":v1

With DCT enabled, the image was automatically signed as part of the push operation.

Two sets of keys were created as part of the signing operation:

- Root key

- Repository key

By default, both are stored below a hidden folder in your home directory called `docker`. On Linux this is `~/.docker/trust`.

The **root key** is the master key (of sorts). It's used to create and sign new repository keys, and should be kept safe. This means you should protect it with a strong passphrase, and you should store it offline in a secure place when not in use. If it gets compromised, you'll be in world of pain! You would normally only have one per person, or may be even one per team or organization, and you'll normally only use it to create new repository keys.

The **repository key**, also known as the *tagging key* is a per-repository key that is used to sign tagged images pushed to a particular repository. As such, you'll have one per repository. It's quite a bit easier to recover from a loss of this key, but you should still protect it with a strong passphrase and keep it safe.

Each time you push an image to a **new repository**, you'll create a new repository tagging key. You need your **root key** to do this, so you'll need to enter the root key's passphrase. Subsequent pushes to the same repository will only require you to enter the passphrase for the repository tagging key.

There's another key called the timestamp key. This gets stored in the remote repository and is used in more advanced use-cases to ensure things like *freshness*.

Let's have a look at pulling images with DCT enabled.

Perform the following commands from the same Docker host that has DCT enabled.

Pull an unsigned image.

```
$ docker image pull nigelpoulton/dockerbook:unsigned
Error: trust data does not exist for docker.io/nigelpoulton/dockerbook:
notary.docker.io no trust data for docker.io/nigelpoulton/dockerbook
```

Note: Sometimes the error message will be No trust data for unsigned.

See how Docker has refused to download the image because it is not signed.

You'll get similar errors if you try to build new images or run new containers from unsigned images. Let's test it.

Pull the unsigned image by using the `--disable-content-trust` flag to override DCT.

```
$ docker image pull --disable-content-trust nigelpoulton/dockerbook:unsigned
```

The `--disable-content-trust` flag overrides DCT on a per-command basis. Use it wisely.

Now try and run a container from the unsigned image.

```
$ docker container run -d --rm nigelpoulton/dockerbook:unsigned  
docker: No trust data for unsigned.
```

This proves that Docker Content Trust enforces policy on push, pull and run operations. Try a build to see it work there as well.

Docker UCP also supports DCT, allowing you to set a UCP-wide signing policy.

To enable DCT across an entire UCP, expand the Admin drop-down and click Admin Settings. Select the Docker Content Trust option and tick the Run Only Signed Images tickbox. This will enforce a signing policy across the entire cluster that will only allow you to deploy services using signed images.

The default configuration will allow any image signed by a valid UCP user. You can optionally configure a list of teams that are authorized to sign images.

That's the basics of Docker Content Trust. Let's move on to configuring and using Docker Trusted Registry (DTR).

Configuring Docker Trusted Registry (DTR)

In the previous chapter we installed DTR, plugged it in to a shared storage backend, and configured HA. We also learned that UCP and DTR share a common single-sign-on sub-system. But there's a few other important things you should configure. Let's take a look.

Most of the DTR configuration settings are located on the `Settings` page of the DTR web UI.

From the `General` tab you can configure:

- Automatic update settings
- Licensing
- Load balancer address
- Certificates
- Single-sign-on

The `TLS Settings` under `Domains & proxies` allows you to change the certificates used by UCP. By default, DTR uses self-signed certificates, but you can use this page to configure the use of custom certificates.

The `Storage` tab lets you configure the backend used for **image storage**. We saw this in the previous chapter when we configured a shared Amazon S3 backend so that we could configure DTR HA. Other storage options include object storage services from other cloud providers, as well as volumes and NFS shares.

The `Security` tab is where you enable and disable *Image Scanning* — binary-level scans that identify known vulnerabilities in images. When you enable *image scanning*, you have the option of updating the vulnerability database *online* or *offline*. Online will automatically sync the database over the internet, whereas the offline method is for DTR instances that do not have internet access and need to update the database manually.

See the *Security in Docker* chapter for more information on Image Scanning.

Last but not least, the `Garbage Collection` tab lets you configure when DTR will perform garbage collection on image layers that are no longer referenced in the Registry. By default, unreferenced layers are not garbage collected, resulting in large amounts of wasted disk space. If you enable garbage collection, layers that are no longer referenced by an image will be deleted, but layers that are referenced by at least one image manifest will not.

See the chapter on Images for more information about how image manifests reference image layers.

Now that we know how to configure DTR, let's use it!

Using Docker Trusted Registry

Docker Trusted Registry is a secure on-premises registry that you configure and manage yourself. It's integrated into UCP for smooth out-of-the-box experience.

In this section, we'll look at how to push and pull images from it, and we'll learn how to inspect and manage repositories using the DTR web UI.

Log in to the DTR UI and create a repo and permissions

Let's log in to DTR and create a new repository that members of the `technology/devs` team can push and pull images from.

Log on to DTR. The DTR URL can be found in the UCP web UI under Admin > Admin Settings > Docker Trusted Registry. Remember that the DTR web UI is accessible over HTTPS on TCP port 443.

Create a new organization and team, and add a user to it. The example will create an organization called `technology`, a team called `devs`, and add the `nigelpoulton` user to it. You can substitute these values in your environment.

1. Click Organizations in the left navigation pane.
2. Click New organization and call it `technology`.
3. Select the new `technology` organization and click the + button next to TEAMS as shown in Figure 17.11.

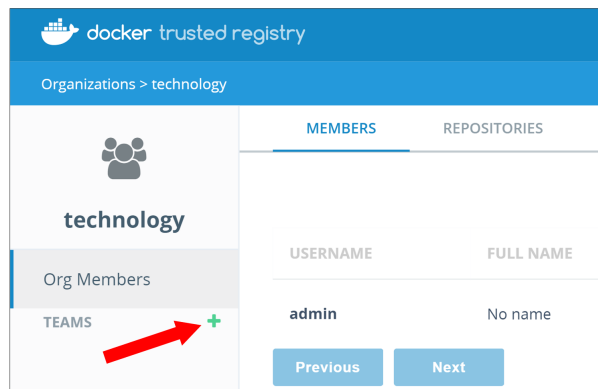


Figure 17.11

4. With the devs team selected, add an existing user.

The example will add the nigelpoulton user. Your user will be different in your environment.

The organization and team changes you have made in DTR will be reflected in UCP. This is because they share the same accounts database.

Let's create a new repository and add the technology/devs team with read/write permission.

Perform all of the following in the DTR web UI.

1. If you aren't already, navigate to Organizations > technology > devs.
2. Select the Repositories tab and create a new repository.
3. Configure the repository as follows.

Make it a **New** repository called **test** under the **technology** organization. Make it **public**, enable **scan on push** and assign **read/write** permissions. Figure 17.12 shows a screenshot of how it should look.

The screenshot shows the 'Add repository' form in the DTR web UI. At the top, there are two radio buttons: 'Existing' (unselected) and 'New' (selected). Below this, the form is divided into several sections. The 'ACCOUNT' section has a dropdown menu showing 'technology'. The 'REPOSITORY NAME' section has a text input field containing 'test'. The 'VISIBILITY' section has two buttons: 'Public' (selected, with a globe icon) and 'Private' (with a lock icon). Below these, there are two sub-sections: 'SCAN ON PUSH' with a toggle switch turned on, and 'PERMISSIONS' with a dropdown menu showing 'Read-write'. At the bottom left, there is a 'DESCRIPTION (OPTIONAL)' text area containing 'Test for Docker Deep Dive book'. Below that, there is a link 'Hide advanced settings'. At the bottom, there is an 'IMMUTABILITY' section with two buttons: 'On' (with a lock icon) and 'Off' (selected, with a lock icon). Below these buttons, there are two sub-sections: 'Tags are immutable' and 'Tags can be overwritten'. At the bottom right, there are two buttons: 'Cancel' and 'Save'.

Figure 17.12 Creating a new DTR image repo

4. Save changes.

Congratulations! You have an image repo on DTR called `<dtr-url>/technology`, and members of the technology/devs team have read/write access, meaning they can push and pull from it.

Push an image to the DTR repo

In this step we'll push a new image to the repo you just created. To do this, we'll complete the following steps:

1. Pull an image and re-tag it.
2. Configure a client to use a certificate bundle.
3. Push the re-tagged image to the DTR repo.
4. Verify the operation in the DTR web UI.

Let's pull an image and tag it so that it can be pushed to the DTR repo.

It doesn't matter what image you pull. The example uses the `alpine:latest` image because it's small.

```
$ docker pull alpine:latest
latest: Pulling from library/alpine
ff3a5c916c92: Pull complete
Digest: sha256:7df6...b1c0
Status: Downloaded newer image for alpine:latest
```

In order to push an image to a specific repo, you need to tag the image with the name of the repo. The example DTR repo has a fully qualified name of `dtr.mydns.com/technology/test`. This is made by combining the DNS name of the DTR and the name of the repo. Yours will be different.

Tag the image so it can be pushed to the DTR repo.

```
$ docker image tag alpine:latest dtr.mydns.com/technology/test:v1
```

The next job is to configure a Docker client to authenticate as a user in the group that has read/write permission to the repository. The high-level process is to create a certificate bundle for the user and configure a Docker client to use those certificates.

1. Login to UCP as admin, or a user that has read/write permission to the DTR repo.

2. Navigate to the desired user account and create a `client` bundle.
3. Copy the bundle file to the Docker client you want to configure.
4. Login to the Docker client and perform the following commands from the client.
5. Unzip the bundle and run the appropriate shell script to configure your environment.

The following will work on Mac and Linux.

```
$ eval "$(<env.sh)"
```

6. Run a `docker version` command to verify the environment has been configured and the certificates are being used.

As long as the `Server` section of the output shows the `Version` as `ucp/x.x.x` it is working. This is because the shell script configured the Docker client to talk to a remote daemon on a UCP manager. It also configured the Docker client to sign all commands with the certificates.

The next job is to log in to DTR. The DTR URL and username will be different in your environment.

```
$ docker login dtr.mydns.com
Username: nigelpoulton
Password:
Login Succeeded
```

You are now ready to push the re-tagged image to DTR.

```
$ docker image push dtr.mydns.com/technology/test:v1
The push refers to a repository [dtr.mydns.com/technology/test]
cd7100a72410: Pushed
v1: digest: sha256:8c03...acbc size: 528
```

The push looks successful, but let's verify the operation in the DTR web UI.

1. If you aren't already, login to the DTR web UI.

2. Click **Repositories** in the left navigation pane.
3. Click **View Details** for the `technology/test` repository.
4. Click the **IMAGES** tab.

Figure 17.13 shows what the image looks like in the DTR repo. We can see that the image is a Linux-based image and that it has 3 major vulnerabilities. We know about the vulnerabilities because we configured the repository to scan all newly-pushed images.

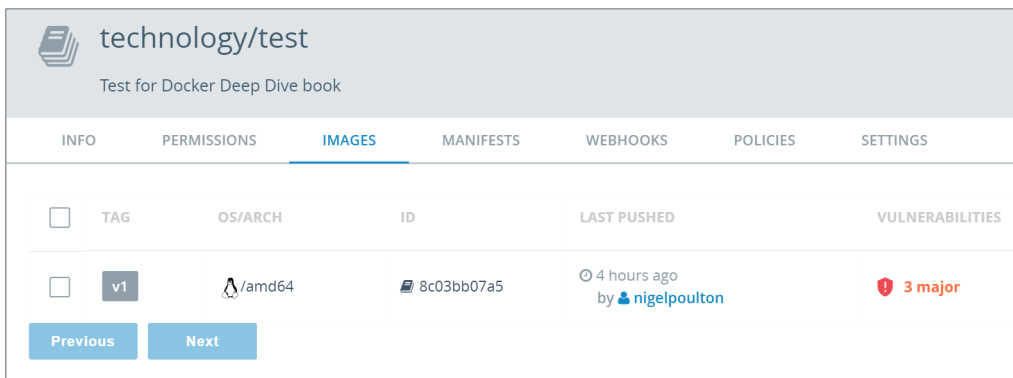


Figure 17.13

Congratulations. You've successfully pushed an image to a new repository on DTR. You can select the checkbox to the left of the image and delete it. Be certain before doing this, as the operation cannot be undone.

Image promotions

DTR has a couple other interesting features:

- Image promotions
- Immutable repos

Image promotions let you build policy-based automated pipelines that promote images through a set of repositories in the same DTR.

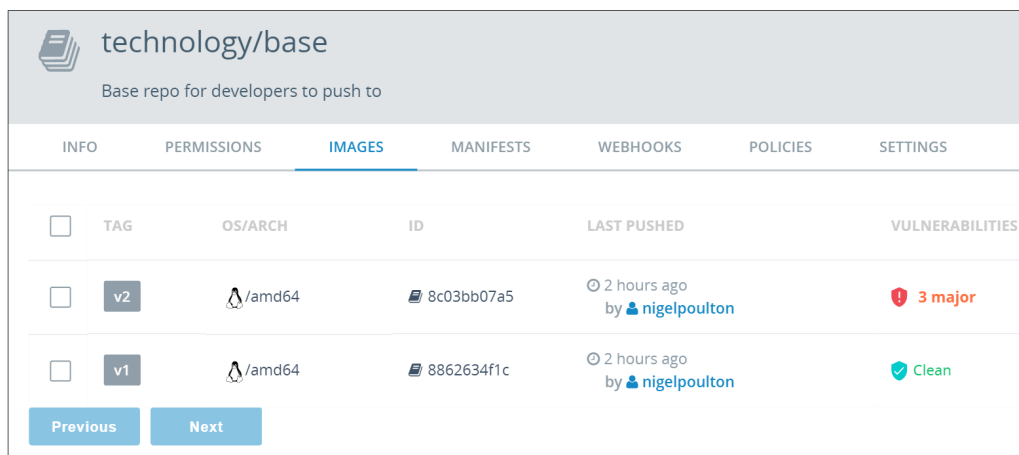
As an example, you might have developers pushing images to a repository called `base`. But you don't want them to be able to push images straight to production in case they contain vulnerabilities. To help with situations like this, DTR allows you to assign a policy to the `base` repo, that will scan all pushed images, and promote them to another repo based on the results of the scan. If the scan highlights issues, the policy can *promote* the image to a quarantined repo, whereas if the scan results are clean, it can promote it to a QA or prod repo. You can even re-tag the image as it passes through the pipeline.

Let's see it in action.

The example that we'll walk through has a single DTR with 3 image repos:

- `base`
- `good`
- `bad`

The `good` and `bad` repos are empty, but the `base` repo has two images in it, shown in Figure 17.14.



technology/base						
Base repo for developers to push to						
INFO	PERMISSIONS	IMAGES	MANIFESTS	WEBHOOKS	POLICIES	SETTINGS
☐	TAG	OS/ARCH	ID	LAST PUSHED	VULNERABILITIES	
☐	v2	/amd64	8c03bb07a5	2 hours ago by nigelpoulton	3 major	
☐	v1	/amd64	8862634f1c	2 hours ago by nigelpoulton	Clean	

Previous Next

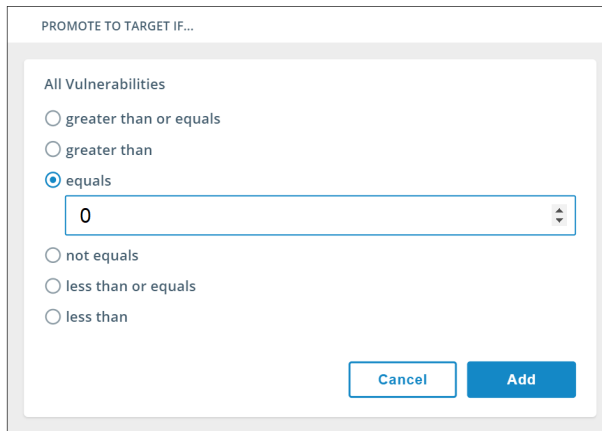
Figure 17.14

As we can see, both images have been scanned, `v1` is clean and has no known vulnerabilities, but `v2` has 3 majors.

We'll create two policies on the base repo so that images with a clean bill-of-health are promoted to the good repo, and images with known vulnerabilities are promoted to the bad repo.

Perform all of the following actions on the base repo.

1. Click the Policies tab and make sure that Is source is selected.
2. Click New promotion policy.
3. Under "PROMOTE TO TARGET IF..." select All Vulnerabilities and create a policy for equals 0.



PROMOTE TO TARGET IF...

All Vulnerabilities

☐ greater than or equals

☐ greater than

☒ equals

0

☐ not equals

☐ less than or equals

☐ less than

Cancel Add

This will create a policy that acts on all images with zero vulnerabilities.

Don't forget to click Add before moving to the next step.

4. Select the TARGET REPOSITORY as technology/good and hit Save & Apply.

Clicking Save will apply the policy to the repo and enforce it for all new images pushed the repo, but it will not affect images already in the repo. Save & Apply will do the same, **but also for images already in the repo.**

If you click Save & Apply, the policy will immediately evaluate all images in the repo and promote those that are clean. This means the v1 image will be promoted to the technology/good repo.

5. Inspect the technology/good repo.

As you can see in Figure 17.16, the v1 image has been promoted and is showing in the UI as PROMOTED.

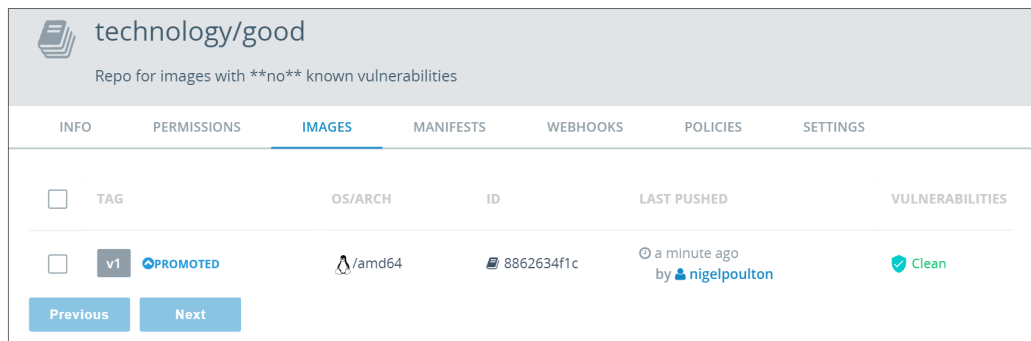


Figure 17.16

The promotion policy is working. Let's create another one to *promote* images that do have vulnerabilities to the technology/bad repo.

Perform all of the following from the technology/base repo.

1. Create another new promotion policy.
2. Create a policy criteria for All Vulnerabilities > 0 and click Add.

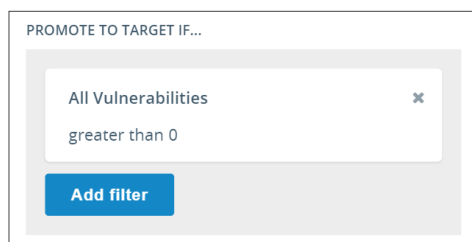


Figure 17.17

3. Add the target repo as technology/bad, and add “-dirty” to the TAG NAME IN TARGET box so that it is now “%n-dirty”. This last bit will re-tag the image as part of the promotion.
4. Click Save & Apply.

5. Check the technology/bad repo to confirm that the policy is enforcing and the v2 image has been promoted and re-tagged.

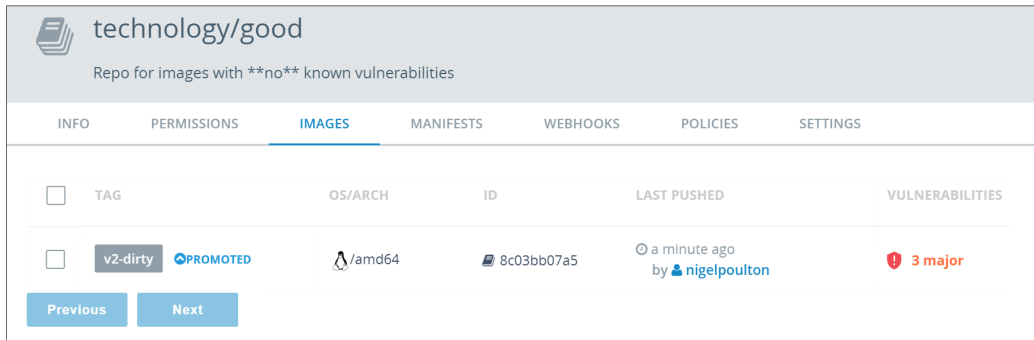


Figure 17.18

Now that images are being promoted to the technology/good repo if they have no vulnerabilities, it might be a good idea to make the repo immutable. This will prevent images from being overwritten and deleted.

1. Navigate to the technology/good repo and click the Settings tab.
2. Set IMMUTABILITY to On and click Save.
3. Try and delete the image.

You'll get the following error.

Tags can't be deleted because this repository is immutable. To delete tags, change the repository settings.

Time for one last feature!

HTTP Routing Mesh (HRM)

Docker Swarm features a layer-4 routing mesh called the Swarm Routing Mesh. This exposes Swarm services on all nodes in the cluster and balances incoming

traffic across service replicas. The end results is a moderately even balance of traffic to all service replicas. However, it has no application intelligence. For example, it cannot route based on data at layer 7 in the HTTP headers. To overcome this, UCP implements a layer-7 routing mesh called the HTTP Routing Mesh, or HRM for short. This builds on top of the Swarm Routing Mesh.

The HRM allows multiple Swarm services to be published on the same Swarm-wide port, with ingress traffic being routed to the right service based on hostname data stored in the HTTP headers of incoming requests.

Figure 17.20 shows a simple two-service example.

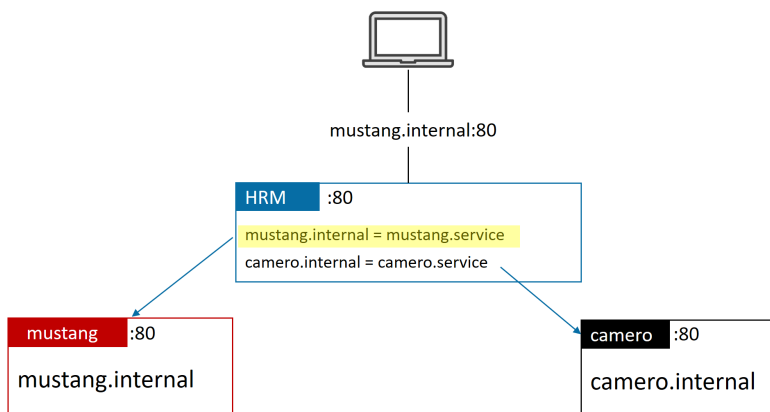


Figure 17.20

In the picture, the laptop client is making an HTTP request to `mustang.internal` on TCP port 80. The UCP cluster has two services that are both listening on port 80. The `mustang` service is published on port 80 and configured to receive traffic intended for the `mustang.internal` hostname. The `camero` service is also published on port 80, but is configured to receive traffic coming in to `camero.internal`.

There is a third service called HRM that maintains the mapping between hostnames and UCP services. It is the HRM that receives incoming traffic on port 80, inspects the HTTP headers and makes the decision of which service to route it to.

Let's walk through an example, then explain a bit more detail when we're done.

We'll build the example shown in Figure 17.20. The process will be as follows: Enable the HRM on port 80. Deploy a *service* called “mustang” using the `nigelpoulton/-`

`dockerbook:mustang` image and create a hostname route for the `mustang` service so that requests to “`mustang.internal`” get routed to it. Deploy a second service called “`camero`” based on the `nigelpoulton/dockerbook:camero` image and create a hostname route for this one that maps it to requests for “`camero.internal`”.

You can use publicly resolvable DNS names such as `mustang.mycompany.com`, all that is required is that you have name resolution configured so that requests to those addresses resolve to the load balancer in front of your UCP cluster. IF you don’t have a load balancer, you can point traffic to the IP of any node in the cluster.

Let’s see it.

1. If you aren’t already, log on to the UCP web UI.
2. Navigate to Admin > Admin Settings > Routing Mesh.
3. Tick the Enable Routing Mesh tickbox and make sure that the HTTP Port is configured to 80.
4. Click Save.

That’s the UCP cluster configured to use the HRM. Behind the scenes this has deployed a new *system service* called `ucp-hrm`, and a new overlay network called `ucp-hrm`.

If you inspect the `ucp-hrm` system service, you’ll see that it’s publishing port 80 in *ingress mode*. This means the `ucp-hrm` is deployed on the cluster and bound to port 80 on all nodes in the cluster. This means **all traffic** coming into the cluster on port 80 will be handled by this service. When the `mustang` and `camero` services are deployed, the `ucp-hrm` service will be updated with hostname mappings so that it knows how to route traffic to those services.

Now that the HRM is deployed, it’s time to deploy our services.

1. Select Services in the left navigation pane and click Create Service.
2. Deploy the “`mustang`” service as follows:
 - **Details/Name:** `mustang`
 - **Details/Image:** `nigelpoulton/dockerbook:mustang`
 - **Network/Ports/Publish Port:** Click the option to Publish Port +
 - **Network/Ports/Internal Port:** 8080

- **Network/Ports/Add Hostname Based Routes:** Click on the option to add a hostname based route
 - **Network/Ports/External Scheme:** `Http://`
 - **Network/Ports/Routing Mesh Host:** `mustang.internal`
 - **Network/Ports/Networks:** Make sure that the service is attached to the `ucp-hrm` network
3. Click **Create** to deploy the service.
 4. Deploy the “camero” service.

Deploy this service with the same settings as the “mustang” service, but with the following differences:

- **Details/Name:** `camero`
 - **Details/Image:** `nigelpoulton/dockerbook:camero`
 - **Network/Ports/Routing Mesh Host:** `camero.internal`
5. Click **Create**.

It’ll take a few seconds for each service to deploy, but when they’re done, you’ll be able to point a web browser at `mustang.internal` and reach the `mustang` service, and `camero.internal` and reach the `camero` service.

Note: You will obviously need name resolution configured so that `mustang.internal` and `camero.internal` resolve to your UCP cluster. This can be to a load balancer sitting in front of your cluster forwarding traffic to the cluster on port 80, or you’re in a lab without a load balancer, it can be a simple local `hosts` file resolving the DNS names to the IP address of a cluster node.

Figure 17.21 shows the `mustang` service being reached via `mustang.internal`.

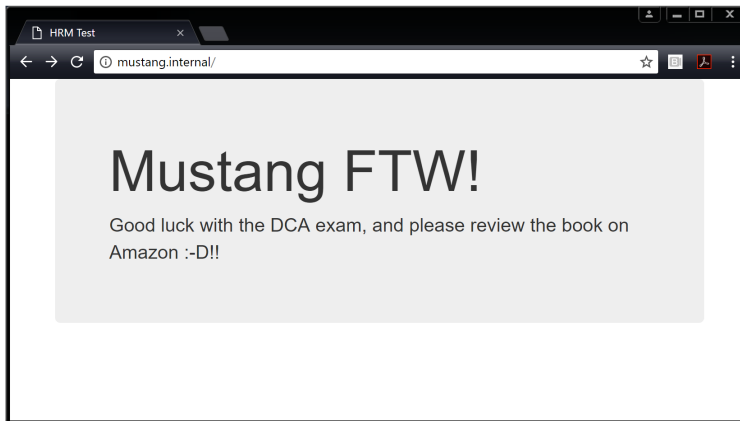


Figure 17.21

Let's remind ourselves of how this works.

The HTTP Routing Mesh is a Docker UCP feature that builds on top of the transport layer Swarm Routing Mesh. Specifically, the HRM adds application layer intelligence in the form of hostname rules.

Enabling the HRM deploys a new UCP *system service* called `ucp-hrm`. This service is published *swarm-wide* on port 80 and 8443. This means that all traffic arriving at the cluster on either of those ports will be sent to the `ucp-hrm` service. This puts the `ucp-hrm` service in a position to receive, inspect, and route all traffic entering the cluster on those ports.

We then deployed two *user services*. As part of deploying each service, we created a hostname mapping that was added to the `ucp-hrm` service. The “mustang” service created a mapping so that it would receive all traffic arriving on the cluster on port 80 with “mustang.internal” in the HTTP header. The “camero” service did the same thing for traffic arriving on port 80 with “camero.internal” in the HTTP header. This resulted in the `ucp-hrm` service having two entries effectively saying the following:

- All traffic arriving on port 80 for “mustang.internal” gets sent to the “mustang” service.
- All traffic arriving on port 80 for “camero.internal” gets sent to the “camero” service.

Let's show Figure 17.20 again.

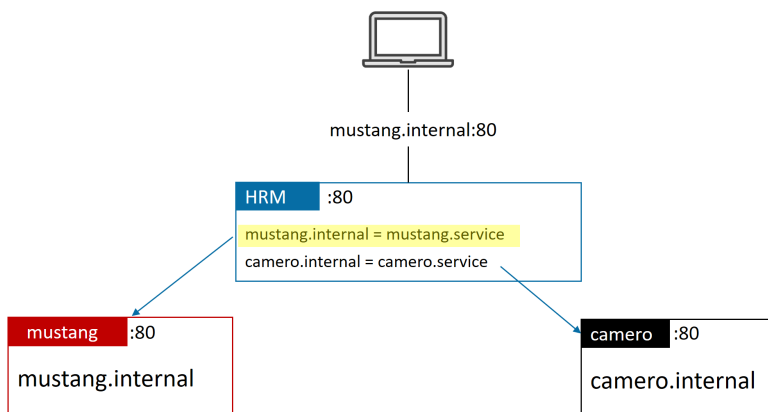


Figure 17.20

Hopefully this should be clear now!

Chapter Summary

UCP and DTR join forces to provide a great suit of features that are valuable to most enterprise organizations.

Strong role-based access control is a fundamental part of UCP, with the ability be extremely granular with permissions – down to individual API operations. Integration with Active Directory and other corporate LDAP solutions is also supported.

Docker Content Trust (DCT) brings cryptographic guarantees to image-based operations. These include `push`, `pull`, `build`, and `run`. When DCT is enabled, all images pushed to remote repos are signed, and all images pulled are verified. This gives you cryptographic certainty that the image you get is the one you asked for. UCP can be configured to enforce a cluster-wide policy requiring all images to be signed.

DTR can be configured to use self-signed certificates, or certificates from trusted 3rd-party CAs. You can configure it to perform binary-level image scans that identify known vulnerabilities. And you can configure policies to automate the promotion of images through your build pipelines.

Finally, we looked at the HTTP Routing mesh that performs application layer routing based on hostnames in HTTP headers.

Appendix A: Securing client and daemon communication

This was originally going to be a section in the “Installing Docker” chapter, or the “Security in Docker” chapter. But it got too long, so I’ve added it here as an appendix.

Docker implements a client-server model. The client implements the CLI, and the server (daemon) implements the functionality, including the public-facing REST API.

The client is called `docker` (`docker.exe` on Windows) and the daemon is called `dockerd` (`dockerd.exe` on Windows). A default installation puts them on the same host and configures them to communicate over a secure local PIC socket:

- `/var/run/docker.sock` on Linux
- `\\.\pipe/docker_engine` on Windows

However, it’s possible to configure them to communicate over the network. But the default daemon network configuration uses an unsecured HTTP socket on port `2375/tcp`.

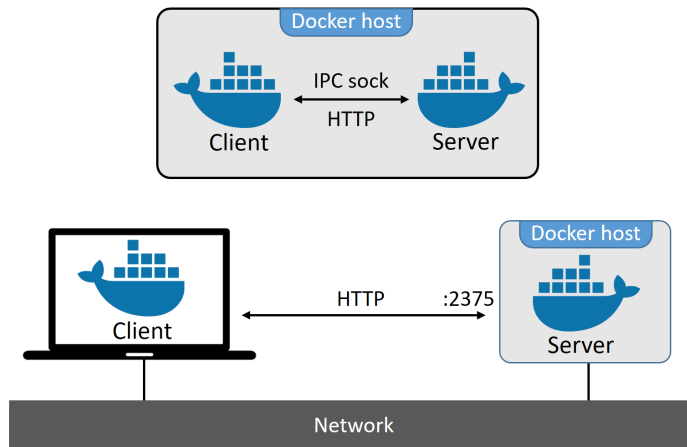


Figure A1.1

Note: It's convention to use 2375 for unencrypted communication between the client and daemon, and 2376 for encrypted traffic.

This might be fine for labs, but it's unacceptable for production.

TLS to the rescue!

Docker let's you configure the client and daemon to only accept network connections that are secured with TLS. This is recommended for production environments, even if you're using trusted internal networks!

Docker offers two modes for securing client-daemon traffic with TLS:

- **Daemon mode:** The Docker daemon will only accept connections from authenticated clients.
- **Client mode:** The Docker client will only connect to Docker daemons that have certificates signed by a trusted CA.

A combination of the two provides the highest security.

We'll use a simple lab environment to walk through the process of configuring Docker for **daemon mode** and **client mode** TLS.

Lab setup

We'll use a simple lab setup for the remainder of the chapter. It's a three-node Linux lab with a CA, Docker client, and Docker daemon. It's vital that all hosts can resolve each other by name.

We'll configure `node1` to be the secure Docker client, and `node3` to be the secure Docker daemon. `node2` will be the CA.

You can follow along in your own environment, but all of the examples shown will use the names and IPs from the lab diagram in Figure A1.2.

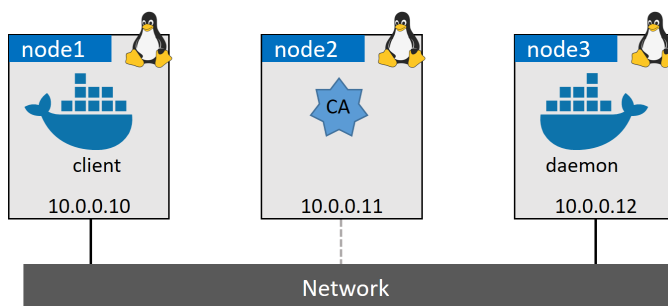


Figure A1.2 Sample lab setup

The high-level process will be as follows:

1. **Configure a CA and certificates**
2. Create a CA (self-signed certs)
3. Create and sign keys for the Daemon
4. Create and sign keys for the Client
5. Distribute keys
6. **Configure Docker to use TLS**
7. Configure daemon mode
8. Configure client mode

Create a CA (self-signed certs)

You only need to complete this step if you are following along in a lab and need to build a CA to sign certificates. Also, we're building a simple CA to help demonstrate how to configure Docker, we're **not** attempting to build a production-grade PKI.

Run the following commands from the CA node in the lab.

1. Create a new private key for the CA.

You will set a passphrase as part of the operation. Don't forget it!

```
$ openssl genrsa -aes256 -out ca-key.pem 4096

Generating RSA private key, 4096 bit long modulus
.....++
..++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
```

You will have a new file in your current directory called `ca-key.pem`. This is the CA's private key.

2. Use the CA's private key to generate a public key (certificate).

You will need to enter the passphrase from the previous step. Hopefully you haven't forgotten it already :-D

```
$ openssl req -new -x509 -days 730 -key ca-key.pem -sha256 -out ca.pem
```

This has added a second file to your working directory called `ca.pem`. This is the CA's public key, a.k.a. "certificate".

You now have two files in your current directory: `ca-key.pem` and `ca.pem`. These are the CA's private and public keys, and form the *identity* of the CA.

Create a key pair for the daemon

In this step, we'll generate a new key-pair for node3. This is the node that will run the secure Docker daemon. It's a four-step process:

1. Create the private key
2. Create the signing request
3. Add IP addresses and make it valid for *server authorization*
4. Generate the certificate

Let's do it.

Run all commands from the CA node (node2).

1. Create the private key for the daemon.

```
$ openssl genrsa -out daemon-key.pem 4096  
<Snip>
```

This has created new file in your working directory called `daemon-key.pem`. This is the private key for the daemon node.

2. Create a certificate signing request (CSR) for the CA to create and sign a certificate for the daemon. Be sure to use the correct DNS name of the node that you intend to run your secure Docker daemon on. The example uses node3.

```
$ openssl req -subj "/CN=node3" \  
-sha256 -new -key daemon-key.pem -out daemon.csr
```

You now have a fourth file in your working directory. This one is the CSR and it is called `daemon.csr`.

3. Add required attributes to the certificate.

We need to create a file that will add a couple of extended attributes to the daemon's certificate when it gets signed by the CA. These attributes will add the daemon's DNS name and IP address, as well as configure the certificate to be used for *server authentication*.

Create a new file called `extfile.cnf` with the following values. The example uses the DNS name and IP of the daemon node in the lab from Figure A1.2. The values in your environment might be different.

```
subjectAltName = DNS:node3,IP:10.0.0.12
extendedKeyUsage = serverAuth
```

4. Generate the certificate.

This step uses the CSR file, CA keys, and the `extfile.cnf` file to sign and configure the daemon's certificate. It will output the daemon's public key (certificate) as a new file called `daemon-cert.pem`

```
$ openssl x509 -req -days 730 -sha256 \
  -in daemon.csr -CA ca.pem -CAkey ca-key.pem \
  -CAcreateserial -out daemon-cert.pem -extfile extfile.cnf
```

At this point, you have a working CA, as well as a key-pair for `node3` which will run the secure Docker daemon.

Delete the CSR and `extfile.cnf` before moving on.

```
$ rm daemon.csr extfile.cnf
```

Create a key pair for the client

In this section, we'll repeat what we just did for the `node3`, but this time we'll do it for `node1` which will run our Docker client.

Run all commands from the CA (`node2`).

1. Create a private key for `node1`.

This will generate a new file in your working directory called `client-key.pem`.

```
$ openssl genrsa -out client-key.pem 4096
```

2. Create a CSR. Be sure to use the correct DNS name of the node that will be your secure Docker client. The example uses `node1`.

```
$ openssl req -subj '/CN=node1' -new -key client-key.pem -out client.csr
```

This will create a new file in your current directory called `client.csr`.

3. Create a file called `extfile.cnf` and populate it with the following value. This will make the certificate valid for client authentication.

```
extendedKeyUsage = clientAuth
```

4. Create the certificate for node1 using the CSR, CA's public and private keys, and the `extfile.cnf` file. This will create the client's signed public key as a new file in your current directory called `client-cert.pem`.

```
$ openssl x509 -req -days 730 -sha256 \  
-in client.csr -CA ca.pem -CAkey ca-key.pem \  
-CAcreateserial -out client-cert.pem -extfile extfile.cnf
```

Delete the CSR and `extfile.cnf` files, as these are no longer needed.

```
$ rm client.csr extfile.cnf
```

At this point you should have the following 7 files in your working directory:

<code>ca-key.pem</code>	<< CA private key
<code>ca.pem</code>	<< CA public key (cert)
<code>ca.srl</code>	<< Tracks serial numbers
<code>client-cert.pem</code>	<< client public key (Cert)
<code>client-key.pem</code>	<< client private key
<code>daemon-cert.pem</code>	<< daemon public key (cert)
<code>daemon-key.pem</code>	<< daemon private key

Before moving on, you should remove write permission from the keys, and make them only readable to you and other accounts that are members of your group.

```
$ chmod 0400 ca-key.pem client-key.pem daemon-key.pem
```

Distribute keys

Now that you've got all of the keys and certificates, it's time to distribute them to the client and daemon nodes. We'll be copying the following files:

- `ca.pem`, `daemon-cert.pem`, and `daemon-key.pem` from the CA to the node3 (the daemon node).

- `ca.pem`, `client-cert.pem`, and `client-key.pem` from the CA to node1 (the client node).

We'll show you how to do it using `scp`, but feel free to use a different tool.

Run the following commands from the directory containing the keys on node2 (the CA node).

```
// Daemon files
$ scp ./ca.pem ubuntu@daemon:/home/ubuntu/.docker/ca.pem
$ scp ./daemon-cert.pem ubuntu@daemon:/home/ubuntu/.docker/cert.pem
$ scp ./daemon-key.pem ubuntu@daemon:/home/ubuntu/.docker/key.pem

//Client files
$ scp ./ca.pem ubuntu@client:/home/ubuntu/.docker/ca.pem
$ scp ./client-cert.pem ubuntu@client:/home/ubuntu/.docker/cert.pem
$ scp ./client-key.pem ubuntu@client:/home/ubuntu/.docker/key.pem
```

A few things to note about the commands:

1. The 2nd, 3rd, 5th, and 6th commands are renaming the files as part of the copy operation. This is important, as Docker expects the files to have these names.
2. These commands will work on Ubuntu Linux, and they assume you are using the `ubuntu` user account.
3. You may have to pre-create the `/home/ubuntu/.docker` hidden directory on the daemon and client nodes before executing the commands. You may also have to change permissions on the `.docker` directory to enable the copy — `chmod 777 .docker` will work, but is not secure. **Remember, we're building a quick CA and certificates so you can follow along. We're not trying to build a secure PKI.**
4. If you're working in something like AWS, you'll need to specify the instance's private key with the `-i <key>` flag for each copy command. For example:

The lab now looks like Figure A1.3

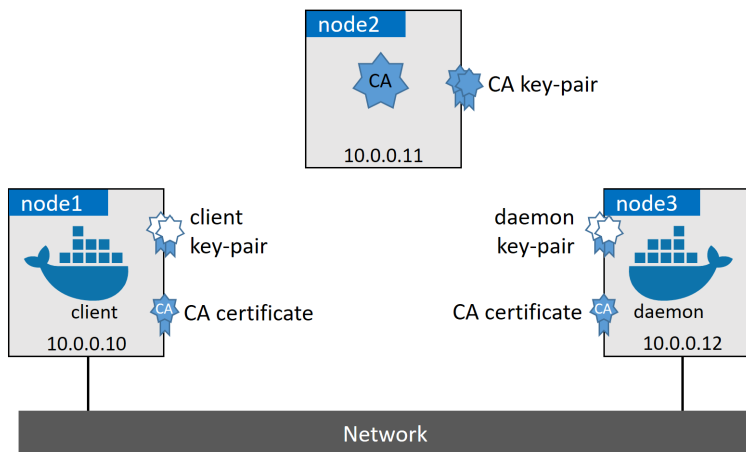


Figure A1.3 Updated lab with keys

The presence of the CA's public key (`ca.pem`) on node1 and node3 is what will tell them to trust the CA and all certificates signed by it.

With the certificates in place, it's **finally time to configure Docker so that the client and daemon use TLS!**

Configure Docker for TLS

As we mentioned previously, Docker has two TLS modes:

- **daemon mode**
- **client mode**

Daemon mode tells the daemon process to only allow connections from clients with a valid certificate. Client mode tells the client only to connect to daemons that have a valid certificate.

We'll configure the daemon process on node1 for *daemon mode*, and test it. After that, we'll configure the client process on node2 for *client mode*, and test that.

Configuring the Docker daemon for TLS

Securing the daemon is as simple as setting a few daemon flags in the `daemon.json` configuration file:

- `tlsverify` enables TLS verification
- `tlscacert` tells the daemon which CA to trust
- `tlscert` tells Docker where the daemon's certificate is
- `tlskey` tells Docker where the daemon's private key is
- `hosts` tells Docker which sockets to bind the daemon on

We'll configure these in the platform-independent `daemon.json` configuration file. This is found in `/etc/docker/` on Linux, and `C:\ProgramData\Docker\config\` on Windows.

Perform all of the following operations on the node that will run your secure Docker daemon (node3 in the example lab).

Edit the `daemon.json` file and add the following lines.

```
{
  "hosts": ["tcp://node3:2376"],
  "tls": true,
  "tlsverify": true,
  "tlscacert": "/home/ubuntu/.docker/ca.pem",
  "tlscert": "/home/ubuntu/.docker/cert.pem",
  "tlskey": "/home/ubuntu/.docker/key.pem"
}
```

Warning! Linux systems running `systemd` do not allow you to use the “hosts” option in `daemon.json`. Instead, you have specify it in a `systemd` override file. The simplest way to do this is with the `sudo systemctl edit docker` command. This will open a new file called `/etc/systemd/system/docker.service.d/override.conf` in an editor. Add the following three lines and save the file.

[Service]`ExecStart=``ExecStart=/usr/bin/dockerd -H tcp://node3:2376`

Now that the TLS and host options are set, it's time to restart Docker.

Once Docker has restarted, you can check that the new hosts value is in effect by inspecting the output of a `ps` command.

```
$ ps -elf | grep dockerd
4 S root    ... /usr/bin/dockerd -H tcp://node3:2376
```

The presence of “`-H tcp://node3:2376`” in the command output is evidence that the daemon is listening on the network. Port 2376 is the standard port for Docker using TLS. 2375 is the default unsecured port.

If you run a normal command, such as `docker version`, it will not work. This is because we've just configured the **daemon** to listen on the network, but the **Docker client** is still trying use the local IPC socket. Try the command again, but this time specifying the `-H tcp://node3:2376` flag.

```
$ docker -H tcp://node3:2376 version
Client:
 Version:      18.01.0-ce
 API version:  1.35
<Snip>
Get http://daemon:2376/v1.35/version: net/http: HTTP/1.x transport connection
n broken: malformed HTTP response "\x15\x03\x01\x00\x02\x02".
* Are you trying to connect to a TLS-enabled daemon without TLS?
```

The command looks better, but it's still not working. This is because the daemon is rejecting all connections from unauthenticated clients.

Congratulations. The Docker daemon is configured to listen on the network, and is rejecting connections from unauthenticated clients.

Let's configure the Docker client on `node1` to use TLS.

Configuring the Docker client for TLS

In this section, we'll configure the Docker client on `node1` for two things:

- To connect to a remote daemon over the network
- To sign all docker commands

Perform all of the following from the node that will run your secure Docker client (`node1` in the example lab).

Export the following environment variable to configure the client to connect to the remote daemon over the network.

```
export DOCKER_HOST=tcp://node3:2376
```

Try the following command.

```
$ docker version
```

```
Client:
```

```
Version:      18.01.0-ce
```

```
<Snip>
```

```
Get http://daemon:2376/v1.35/version: net/http: HTTP/1.x transport connection  
broken: malformed HTTP response "\x15\x03\x01\x00\x02\x02".
```

```
* Are you trying to connect to a TLS-enabled daemon without TLS?
```

The Docker client is now sending commands to the remote daemon across the network, but the remote daemon will only accept authenticated connections.

Export one more environment variable to tell the Docker client to sign all commands with its certificate.

```
export DOCKER_TLS_VERIFY=1
```

Run the `docker version` command again.

Appendix A: Securing client and daemon communication

398

```

$ docker version
Client:
  Version:      18.01.0-ce
<Snip>
Server:
  Engine:
    Version:     18.01.0-ce
    API version: 1.35 (minimum version 1.12)
    Go version:  go1.9.2
    Git commit:  03596f5
    Built:       Wed Jan 10 20:09:37 2018
    OS/Arch:     linux/amd64
    Experimental: false

```

Congratulations. The client is successfully talking to the remote daemon over a secure connection. The final configuration of the lab is shown in Figure A1.4.

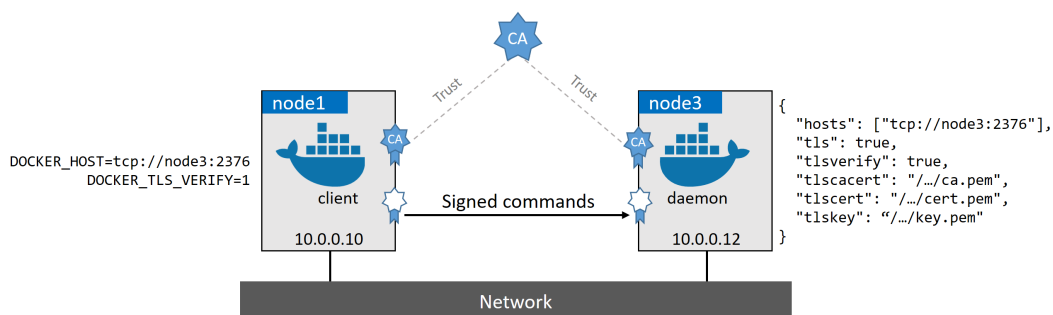


Figure A1.4

A couple of final points before we do a quick recap.

1. This last example works because we copied the clients TLS keys to the folder that Docker expects them to be in. This is a hidden folder in your user's home directory called `.docker`. We also gave the keys the default filenames that Docker expects (`ca.pem`, `cert.pem`, and `key.pem`). You can specify a different folder by exporting `DOCKER_CERT_PATH`.
2. You will probably want to make the environment variables (`DOCKER_HOST` and `DOCKER_TLS_VERIFY`) more permanent fixtures of your environment.

Docker TLS Recap

Docker supports two TLS modes:

- daemon mode
- client mode

Daemon mode will refuse connections from clients that do not sign commands with a valid certificate. Client mode will not connect to remote daemons that do not possess a valid certificate.

Configuring a daemon for TLS is done through the Docker daemon configuration file. The file is called `daemon.json` and it's platform agnostic.

The following `daemon.json` should work on most systems:

```
{
  "hosts": ["tcp://node3:2376"],
  "tls": true,
  "tlsverify": true,
  "tlscacert": "/home/ubuntu/.docker/ca.pem",
  "tlscert": "/home/ubuntu/.docker/cert.pem",
  "tlskey": "/home/ubuntu/.docker/key.pem"
}
```

- `hosts` tells Docker which socket to bind the daemon on. The example binds it to a network socket on port 2376. You can use any free port, but it's convention to use 2376 for secured Docker connections. Linux systems running `systemd` cannot use this flag and require the use of a `systemd` override file.
- `tls` and `tlsverify` force the daemon to only use encrypted and authenticated connections.
- `tlscacert` tells Docker which CA to trust. This causes Docker to trust all certificates signed by that CA.
- `tlscert` tells Docker where the daemon's certificate is located.
- `tlskey` tells Docker where the daemon's private key is located.

Making any changes to these values requires a Docker restart for them to take effect.

Configuring the **Docker client** for TLS is as simple as setting two environment variables:

- DOCKER_HOST
- DOCKER_TLS_VERIFY

DOCKER_HOST tells the client where to find the daemon. `export DOCKER_HOST=tcp://node3:2376` will tell the Docker client to connect to the daemon on a remote host called node3 on port 2376.

`export DOCKER_TLS_VERIFY=1` will tell the Docker client to sign all of the commands it issues.

Appendix B: The DCA Exam

This appendix will be updated over time with tips and advice for taking the DCA exam.

I'm also starting a new website and LinkedIn group for you to share your exam experiences and tips.

- The website is www.dockercerts.com and is currently under development
- The [LinkedIn group](#)²⁸ is called **Docke Certified Associate (DCA)

Other resources to help with the exam

At the time of writing, this is the only resource available that covers all DCA exam objectives.

I also have an [excellent video training course](#)²⁹ that covers most of the exam objectives and is a great way to help you remember what you've learned in this book.

The video course is fast-paced, fun, and has excellent reviews!



Jose Gomez @pipoe2h · Jan 13



@Docker Deep Dive by @nigelpoulton in @pluralsight is a master piece. Great job, IMHO your best course at the moment. #Docker #Containers #DevOps

²⁸<https://www.linkedin.com/groups/13578221>

²⁹<https://app.pluralsight.com/library/courses/docker-deep-dive-update/table-of-contents>



Deepak koshal @Deepakkoshal · Jan 16



Replying to @nigelpoulton @Docker

I feel so lucky to have you as my trainer. You got me from zero to Docker in true sense and now DCA. In-between is it normal to see flying whales 🐳 in the dream?



Rubén Campos @rucamzu · Jan 18



I completed the first three @pluralsight courses in the @Docker path by @nigelpoulton, including the Deep Dive!! Tons of good stuff in there, and extra kudos to Nigel for making the courses so much the more enjoyable. Thanks!!



Elias Valdez @sailevc · Jan 28



Just finished your Docker Deep Dive Architecture & Theory module recap. Answering to your "Is this good?" question: yeah, it's awesome. It's always good to know the how and why and you made it not boring at all. It was quite the contrary, actually. @nigelpoulton



Emmanuel Ballerini @emballerini · Feb 4



Just finished "Docker deep dive" on @pluralsight by @nigelpoulton Great course! Highly recommended to anyone who wants to learn how Docker really works!



Justin Hartman @STLJHartman · Jan 19



From zero to Proficient in a few weeks, thanks @nigelpoulton for creating amazing @pluralsight #docker courses. Keep on creating!! Your training style is excellent and enthusiasm is addictive. I'm all in and looking forward to getting an expert score in the near future.

Let me say two things if you're unsure about spending money on a video course:

1. It's worth it if it helps you pass the DCA exam!
2. Pluralsight always has a free trial. Sign up for the trial and see if you like it — I think you'll love it!

Mapping exam objectives to chapters

Here is a list of the exam objectives and which chapters they are covered in. Almost all objectives will be covered in more chapters than shown here, but these are the main chapters where they are covered in the most detail.

Domain 1: Orchestration (25% of exam)

- Complete the setup of a swarm mode cluster, with managers and worker nodes: **CHAPTER 10**
- State the differences between running a container vs running a service: **CHAPTERS 10 and 14**
- Demonstrate steps to lock a swarm cluster: **CHAPTER 10**
- Extend the instructions to run individual containers into running services under swarm: **CHAPTERS 10 and 14**
- Interpret the output of “docker inspect” commands: **Several chapters**
- Convert an application deployment into a stack file using a YAML compose file with docker stack deploy: **CHAPTER 14**
- Increase # of replicas: **CHAPTERS 10 and 14**
- Add networks, publish ports: **CHAPTERS 9, 11, 12, and 14**
- Mount volumes: **CHAPTERS 9 and 13**
- Illustrate running a replicated vs global service: **CHAPTER 10**
- Identify the steps needed to troubleshoot a service not deploying: **CHAPTER 14**
- Apply node labels to demonstrate placement of tasks: **CHAPTER 14**
- Sketch how a Dockerized application communicates with legacy systems: **CHAPTER 11**
- Paraphrase the importance of quorum in a swarm cluster: **CHAPTERS 10 and 16**
- Demonstrate the usage of templates with “docker service create”: **CHAPTER 10**

Domain 2: Image Creation, Management, and Registry (20% of exam)

- Describe Dockerfile options [add, copy, volumes, expose, entrypoint, etc): **CHAPTERS 8 and 9**
- Show the main parts of a Dockerfile: **CHAPTERS 8 and 9**
- Give examples on how to create an efficient image via a Dockerfile: **CHAPTER 8**
- Use CLI commands such as list, delete, prune, rmi, etc to manage images: **CHAPTER 6**
- Inspect images and report specific attributes using filter and format: **CHAPTER 6**
- Demonstrate tagging an image: **CHAPTERS 6 and 17**
- Utilize a registry to store an image: **CHAPTER 17**
- Display layers of a Docker image: **CHAPTER 6**
- Apply a file to create a Docker image: **CHAPTER 8**
- Modify an image to a single layer: **CHAPTER 8**
- Describe how image layers work: **CHAPTER 8**
- Deploy a registry (not architect): **CHAPTER 16**
- Configure a registry: **CHAPTERS 16 and 17**
- Log into a registry: **CHAPTERS 6 and 17**
- Utilize search in a registry: **CHAPTER 6**
- Tag an image: **CHAPTERS 6 and 17**
- Push an image to a registry: **CHAPTERS 8 and 17**
- Sign an image in a registry: **CHAPTER 17**
- Pull an image from a registry: **CHAPTER 6**
- Describe how image deletion works: **CHAPTERS 6 and 17**
- Delete an image from a registry: **CHAPTER 17**

Domain 3: Installation and Configuration (15% of exam)

- Demonstrate the ability to upgrade the Docker engine: **CHAPTER 3**
- Complete setup of repo, select a storage driver, and complete installation of Docker engine on multiple platforms: **CHAPTER 3**
- Setup swarm, configure managers, add nodes, and setup backup schedule: **CHAPTERS 10 and 16**
- Create and manager user and teams: **CHAPTERS 16 and 17**
- Outline the sizing requirements prior to installation: **CHAPTER 16**
- Understand namespaces, cgroups, and configuration of certificates: **CHAPTERS 5, 15, 16 and Appendix A**
- Use certificate-based client-server authentication to ensure a Docker daemon has the rights to access images on a registry: **CHAPTER 17**
- Consistently repeat steps to deploy Docker engine, UCP, and DTR on AWS and on premises in an HA config: **CHAPTERS 16 and 17**
- Complete configuration of backups for UCP and DTR: **CHAPTER 16**
- Configure the Docker daemon to start on boot: **CHAPTER 3**

Domain 4: Networking (15% of exam)

- Create a Docker bridge network for a developer to use for their containers: **CHAPTER 11**
- Troubleshoot container and engine logs to understand a connectivity issue between containers: **CHAPTER 11**
- Publish a port so that an application is accessible externally: **CHAPTERS 7, 9, 10, 11, 14, and 17**
- Identify which IP and port a container is externally accessible on: **CHAPTERS 7, 9, 11, 17**
- Describe the different types and use cases for the built-in network drivers: **CHAPTER 11**
- Understand the Container Network Model and how it interfaces with the Docker engine and network and IPAM drivers: **CHAPTER 11**

- Configure Docker to use external DNS: **CHAPTER 11**
- Use Docker to load balance HTTP/HTTPS traffic to an application (Configure L7 load balancing with Docker EE): **CHAPTER 17**
- Understand and describe the types of traffic that flow between the Docker engine, registry, and UCP controllers: **CHAPTERS 6, 17, and Appendix A**
- Deploy a service on a Docker overlay network: **CHAPTERS 10, 12, and 14**
- Describe the difference between “host” and “ingress” port publishing mode: **CHAPTERS 11 and 14**

Domain 5: Security (15% of exam)

- Describe the process of signing an image: **CHAPTERS 6, 15, and 17**
- Demonstrate that an image passes a security scan: **CHAPTERS 15 and 17**
- Enable Docker Content Trust: **CHAPTERS 15 and 17**
- Configure RBAC in UCP: **CHAPTER 17**
- Integrate UCP with LDAP/AD: **CHAPTER 17**
- Demonstrate creation of UCP client bundles: **CHAPTERS 16 and 17**
- Demonstrate creation of UCP client bundles: **CHAPTER 15**
- Describe swarm default security: **CHAPTERS 10 and 15**
- Describe MTLS: **CHAPTERS 15 and 17**
- Identity roles: **CHAPTER 17**
- Describe the difference between UCP workers and managers: **CHAPTERS 16 and 17**
- Describe process to use external certificates with UCP and DTR: **CHAPTERS 16 and 17**

Domain 6: Storage and Volumes (10% of exam)

- State which graph driver should be used on which OS: **CHAPTERS 3 and 13**
- Demonstrate how to configure devicemapper: **CHAPTER 3**
- Compare object storage to block storage, and explain which one is preferable when available: **CHAPTER 13**

- Summarize how an application is composed of layers and where those layers reside on the filesystem: **CHAPTERS 6 and 13**
- Describe how volumes are used with Docker for persistent storage: **CHAPTERS 13 and 14**
- Identify the steps you would take to clean up unused images on a filesystem, also on DTR: **CHAPTERS 13 and 17**
- Demonstrate how storage can be used across cluster nodes: **CHAPTERS 13 and 17**

Appendix C: What next

Hopefully you're feeling confident with Docker and ready to take the DCA exam! Fortunately, taking your container journey to the next step has never been easier!

Practice

It's never been easier to spin-up infrastructure and workloads. Docker for Mac and Docker for Windows make it easy to play and develop with Docker on your laptop. [Play with Docker](#)³⁰ is a free-to-use online playground where you can practice with Docker until you're a world authority!

Video training

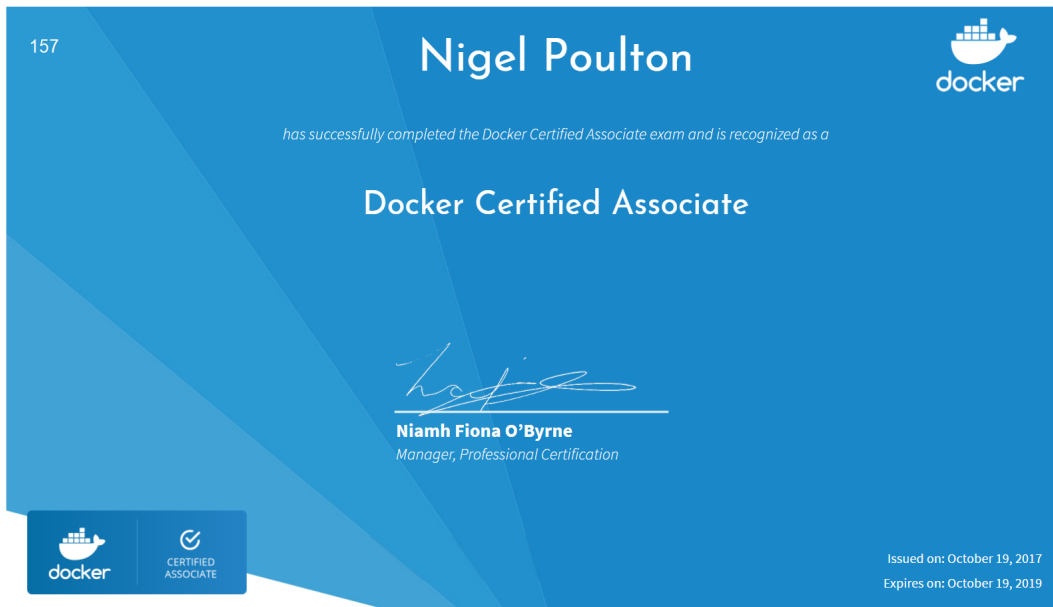
I've created a ton of highly-praised video training courses at [Pluralsight](#)³¹. If you're not a member of Pluralsight then become one! Yes, it costs money, but it's worth it! And if you're unsure... they always have a free trial where you can get free access to my courses for a limited period.

Certifications

There's now an official way to prove your Docker expertise! I did, and I recommend you do too.

³⁰<https://play-with-docker.com/>

³¹<http://app.pluralsight.com/author/nigel-poulton>



Community events

I highly recommend you attend events like [Dockercon](https://www.dockercon.com)³² and your local [Docker meetups](https://www.docker.com/community/meetup-groups)³³. Make sure you come and say “Hi” if you see me there!

Feedback

Massive thanks for reading my book. I really hope it was useful!

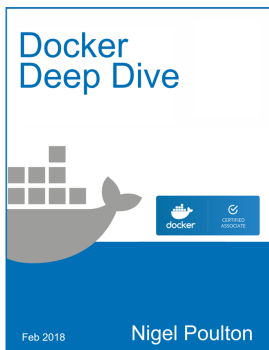
Now let me ask a favor...

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!

³²<https://www.dockercon.com>

³³<https://www.docker.com/community/meetup-groups>

Need these



★★★★☆ ▾ 40 customer reviews

Docker Deep Dive

by Nigel Poulton ▾ (Author)

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](#)³⁴ as well, but stars and cars are what I dream about :-D



³⁴<https://twitter.com/nigelpoulton>

The ~~end~~. beginning...

... of the most exciting chapter of your career!