

DOCKER

SUCCINCTLY

BY **ELTON STONEMAN**

Docker Succinctly

By

Elton Stoneman

Foreword by Daniel Jebaraj



Copyright © 2018 by Syncfusion, Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: John Elderkin

Acquisitions Coordinator: Morgan Weston, social media manager, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story behind the Succinctly Series of Books	7
About the Author.....	9
Chapter 1 Introducing Docker.....	10
What is Docker?.....	10
Installing Docker	10
Running containers	13
Summary.....	19
Chapter 2 Packaging Applications with Docker	20
Meet the Dockerfile	20
The main Dockerfile instructions.....	22
How Docker builds an image	25
Summary.....	29
Chapter 3 Image Registries and the Docker Hub	30
About image registries	30
Using the Docker Hub.....	30
Tags and image versions	32
Automating builds	34
Choose your base image carefully	36
Running your own registry	38
Other registry options.....	40
Summary.....	40
Chapter 4 Data Storage in Docker	41
Containers and their data.....	41
Images, containers, and volumes	41

Mounting into data volumes	43
Sharing data volumes	45
Removing containers, data volumes, and images	48
Summary.....	50
Chapter 5 Orchestrating Systems with Docker.....	51
Orchestrating multicontainer solutions.....	51
Docker networks	51
Docker Compose	55
Summary.....	60
Chapter 6 Clustering Hosts with Docker Swarm	61
Clustering with Docker swarm mode	61
Creating a Docker swarm	61
Working with Docker swarm	63
Scaling services	66
Summary.....	70
Chapter 7 Docker on Linux, Windows, and ARM.....	71
Containers and kernels	71
Docker on Linux	72
Docker on Windows	74
Docker on ARM.....	75
Hybrid Docker swarms.....	77
Summary.....	79
Chapter 8 Running Docker in the Cloud.....	80
Docker versus the cloud	80
Docker on DigitalOcean	80
Docker for Azure	84
Multicloud Docker swarms	87

Summary.....	89
Chapter 9 Continuing Your Docker Journey	91
Next steps with Docker	91
Docker and Docker, Inc.	91
Docker and DevOps.....	96
Docker and microservices.....	97
Next steps: containerize what you know.....	98

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge
As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Elton Stoneman is a Pluralsight author and Microsoft MVP, and he was a Docker Captain before he joined Docker. He has been connecting systems since 2000, and he has spent many years designing and building distributed solutions with a variety of technologies and a mixture of on-premises and cloud deliveries.

His popular [Pluralsight courses](#) cover a range of technologies, including Microsoft .NET, Nginx, AWS, Azure, Big Data, and Docker.

Docker Succinctly is Elton's fourth e-book for Syncfusion (following *Hadoop Succinctly*, *Hive Succinctly*, and *HBase Succinctly*), and it is accompanied by source code on GitHub along with links to various container images on the Docker Hub at <https://github.com/SyncfusionSuccinctlyE-Books/Docker-Succinctly>.

He's also written a comprehensive book about Windows containers, [Docker on Windows](#), published by Packt.

You'll find Elton in other places online, like blogging at <https://blog.sixeyed.com>, or tweeting [@EltonStoneman](#).

Chapter 1 Introducing Docker

What is Docker?

Docker is an application platform. It lets you package your application with everything it needs, from the operating system upwards, into a single unit that you can share and run on any computer that has Docker. Docker runs your application in a lightweight, isolated component called a container.

It's a simple proposition, but it is hugely powerful. The application package, called a Docker image, is typically only tens or hundreds of megabytes, so it's cheap to store and fast to move. When you run a container from the image, it will start in seconds and the application process actually runs on the host, which means you can run hundreds of containers on a single machine. Images can be versioned, so you can be sure the software you release to production is exactly what you've tested, and the Docker tools can even scan images for security vulnerabilities, so you will know if your application is safe.

With Docker, you can build your application image and know that it will run in the same way on your development laptop, on a VM in an on-premise test lab, or on a cluster of machines in the cloud. It's a facilitator for some of the most popular trends in software delivery. You can easily add a packaging step into your continuous integration process to generate a versioned image for every commit. You can extend that to continuous delivery, automatically deploying the latest image through environments to production. In Docker, the packaging process is where development and operations meet, which means it's a great start for the transition to DevOps. And having a framework for orchestrating work between many containers gives you the foundation for microservice architectures.

Docker is open source and cross-platform, and one of its ecosystem's most compelling aspects is the Docker Hub—a public registry where organizations and individuals share their own application container images. On the Hub, you'll find official, supported images for popular technologies such as Nginx, MariaDB, and Redis alongside custom community images, and you can share your own images, too. Images on the Hub can be as simple as a Hello World app or as complex as a fully distributed Hadoop cluster, and because the images are usually open source, navigating the Hub is a great way to get started with Docker.

In this chapter, we'll do just that—we'll get Docker installed, and we'll run some containers using images from the Docker Hub. We'll see how easy it is to get up and running with Docker, and we'll begin to understand the power of the platform. In the rest of this e-book, we'll dig deeper and walk through all you'll need to know in order to be comfortable using Docker in production.

Installing Docker

Docker is a single product which has three components—the background server that does the work; the Docker client, a command-line interface for working with the server; and a REST API for client-server communication.

The client is cross-platform, which means you can run it natively from Linux, Windows, and OS/X machines, and you can manage Docker running locally or on a remote machine. The Docker server runs on Linux and on the latest versions of Windows.

You don't need to be a Linux guru to use Docker effectively. The Docker team has put together packages for Mac and Windows that make use of virtualization technology on the host so that your Docker server runs inside a Linux VM on your OS/X or Windows machine (you run the client locally and talk to the server through the REST API exposed on the VM).



Note: The latest Docker for Mac and Docker for Windows packages require up-to-date versions of the operating systems OS/X Yosemite or Windows 10. If you're running older versions, you can still use Docker with the Docker Toolbox. It's an older package that uses VirtualBox to run the Linux VM, but you use it in the same way.

Figure 1 shows the different options for running the Docker Engine on various operating systems.

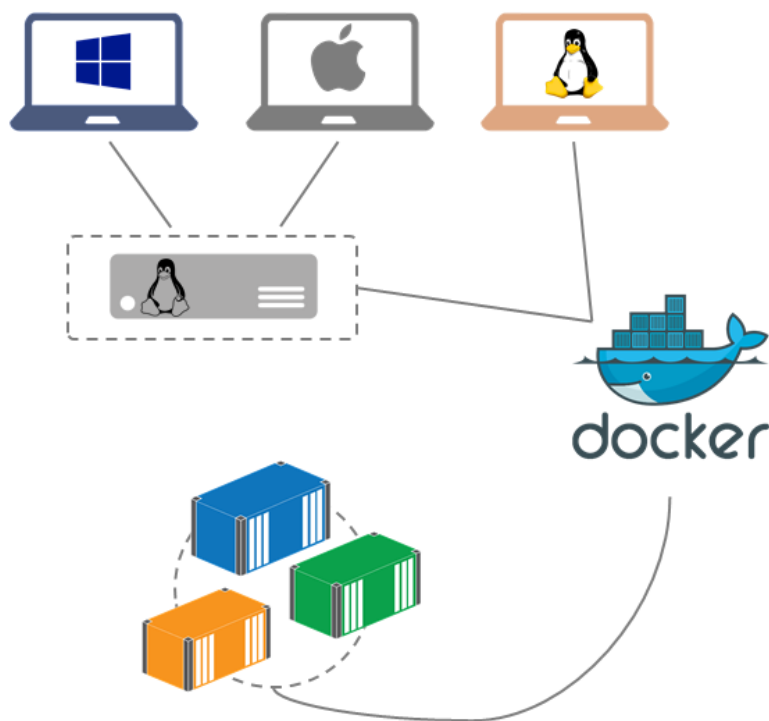


Figure 1: Running Docker on Windows, Mac, and Linux

That figure may look complex, but it's all wrapped up in simple installations. Knowing how Docker actually runs on your machine is a good idea, but the installation itself will only take a few minutes to download, followed by just a couple of clicks. Docker's documentation is first-rate, and the [Get started section on Docker Store](#) includes detailed instructions for Mac, Windows, and Linux.



Note: On Windows, you'll need to have Hardware Virtualization (VT-x) enabled in the BIOS to run the Docker Linux VM, and after installing you'll need to reboot. You can switch between Windows containers and Linux containers—the examples in this book use Linux containers.

After you've installed Docker, simply launch a command-line window (or the Docker Terminal if you're using Docker Toolbox) and you can start running Docker client commands. Code Listing 1 shows the output from running `docker version`, which gives you details on the installed version of Docker.

Code Listing 1: Checking the Version of Docker

```
$ docker version

Client:

Version:      17.11.0-ce
API version:  1.34 (downgraded from 1.35)
Go version:   go1.9.2
Git commit:
Built:        Fri Nov 24 16:01:38 2017
OS/Arch:      darwin/amd64
Orchestrator: kubernetes

Server:

Version:      17.11.0-ce
API version:  1.34 (minimum version 1.12)
Go version:   go1.8.5
Git commit:   1caf76c
Built:        Mon Nov 20 18:39:28 2017
OS/Arch:      linux/amd64
Experimental: true
```

Docker reports separately on the client and the server because you might be using a local client to manage a remote server, and those could be on different versions or different platforms. Docker is built using Go, but it ships as compiled binaries, which means you don't need to install the Go runtime beforehand.

With Docker installed, you're ready to start running some containers.

Running containers

Docker images are packaged applications. You can push them to a central store (called a registry) and pull them on any machine that has access to the registry. The image is a single logical unit that contains the application package. In order to start the app, you run a container from the image.

Images are typically built to run a single process. If your app needs to work with other services, you run those services in their own containers and orchestrate them so that all the containers can work together (which you'll learn about in [Chapter 5, Orchestrating Systems with Docker](#)).

When you run a container from an image, it may be a short-lived app that runs some functionality and then ends; it may be a long-running app that runs like a background service; or it may be an interactive container that you can connect with as though it was a remote machine.

Hello World

Let's start with the simplest container you can run. With Docker installed and an Internet connection, you can run the command in Code Listing 2 and see the Hello World container in action.

Code Listing 2: Running Hello World

```
$ docker container run hello-world

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

b04784fba78d: Pull complete

Digest: sha256:f3b3b28a45160805bb16542c9531888519430e9e6d6fffc09d72261b0d26ff74f

Status: Downloaded newer image for hello-world:latest

Hello from Docker!

...
```

You'll see some helpful text written out, but don't be underwhelmed that your first container merely writes to the console. When you run the command, there's a lot happening:

- Your local Docker client sends a request to the Docker server to run a container from the image called **hello-world**.
- The Docker server checks to see if it has a copy of the image in its cache. If not, it will download the image from Docker Hub.
- When the image is downloaded locally, the Docker server runs a container from the image, and sends the output back to the Docker client.

With this image, the process inside the container ends when the console output has been written, and Docker containers exit when there are no processes running inside. You can check that by getting a list of running containers from Docker using the **container ls** (container list) command. Because the **hello-world** container has ended, there are no running containers and the command output will be empty, as in Code Listing 3.

Code Listing 3: Checking for Running Containers

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	



Tip: You can see *all* your containers, including the ones that have exited, by running `docker container ls --all`, which lists containers in any status.

This type of container, which executes some code and then exits, is a very useful pattern. You can use this approach for containers that script repetitive tasks such as backing up data, creating infrastructure in the cloud, or processing a message from a message queue. But containers are equally well-suited for long-running background processes.

Hello Nginx

[Nginx](#) is a powerful, lightweight, open-source HTTP server. It's been growing in popularity for many years—as a Web server, it has been progressively taking Internet market share from Apache and IIS. With the growth of Docker, Nginx has seen an acceleration in popularity because it's easy to configure, builds into a very small image, and has many features that gel nicely with orchestrated container workloads.

The Docker Hub has an [official Nginx image](#) that is maintained and supported by the Nginx team. It comes in several variations, but they fundamentally do the same thing—start the Nginx server process listening on port 80, inside a container. With Code Listing 4, you can run the smallest version of the Nginx Docker image, which is based on Alpine Linux.

Code Listing 4: Running Nginx in a Container

```
$ docker container run nginx:alpine

Unable to find image 'nginx:alpine' locally

alpine: Pulling from library/nginx
019300c8a437: Pull complete
2425a41f485c: Pull complete
26e59859b15d: Pull complete
a69539b662c9: Pull complete

Digest: sha256:6cf0606c8010ed70f6a6614f8c6dfedbdb5e2d207b5dd4b0fab846bbc26f263e

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

When you run that image, a container will start in the foreground, running the Nginx process in your terminal so that you can't run any other commands. The container is listening for HTTP requests on port 80, but that's port 80 **inside** the container, so we can't reach it from the host machine. This container isn't doing much, so we can kill it by ending the process with Ctrl+C.

Docker supports long-running background processes, such as web servers, by allowing containers to run in detached mode, so the container keeps running in the background. Code Listing 5 runs a new container from the same Nginx image, which will run in the background with the **--detach** flag and with port 80 published with the **--publish** flag.

Code Listing 5: Running Nginx as a Background Container

```
$ docker container run --detach --publish 80:80 nginx:alpine

a840ccbf8c8652cb6d52b5489146a59e8468747f3372e38426fe3deb40d84372a
```

That command publishes port 80 inside the container to port 80 on the host. Ports can't be shared, so this will fail if you have another process listening on port 80. However, you can publish the container port to any free port on the host: **--publish 8081:80** maps port 8081 on the host to port 80 in the container.

The output from Docker shows the unique ID of the new container, then control returns to the terminal. You can check if the container is running with the **container ls** command, as in Code Listing 6.

Code Listing 6: Listing the Background Container

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
a840ccbf865	nginx:alpine	"nginx -g 'daemon ...'"	47 seconds ago
Up 45 seconds	0.0.0.0:80->80/tcp	heuristic_roentgen	

The output tells us a number of things: which image the container is running; a short form of the container ID—starting **a840** in this case—that Docker uniquely generates; the container name—**heuristic_roentgen**—that Docker will randomly assign unless we supply a name; and the command running in the container—**nginx**. This container is running in the background, Nginx is listening on port 80, and we’ve published port 80 from the container, mapping it to port 80 on the host machine running the container.

When requests come in to port 80 on the host now, they will be routed to the container, and the response will come from the Nginx process running inside the container. On Linux, the Docker Engine is running directly on your host machine, and Docker for Mac and Docker for Windows use native network sharing, which means you can browse to **http://localhost** and see the Nginx welcome page, as in Figure 2.

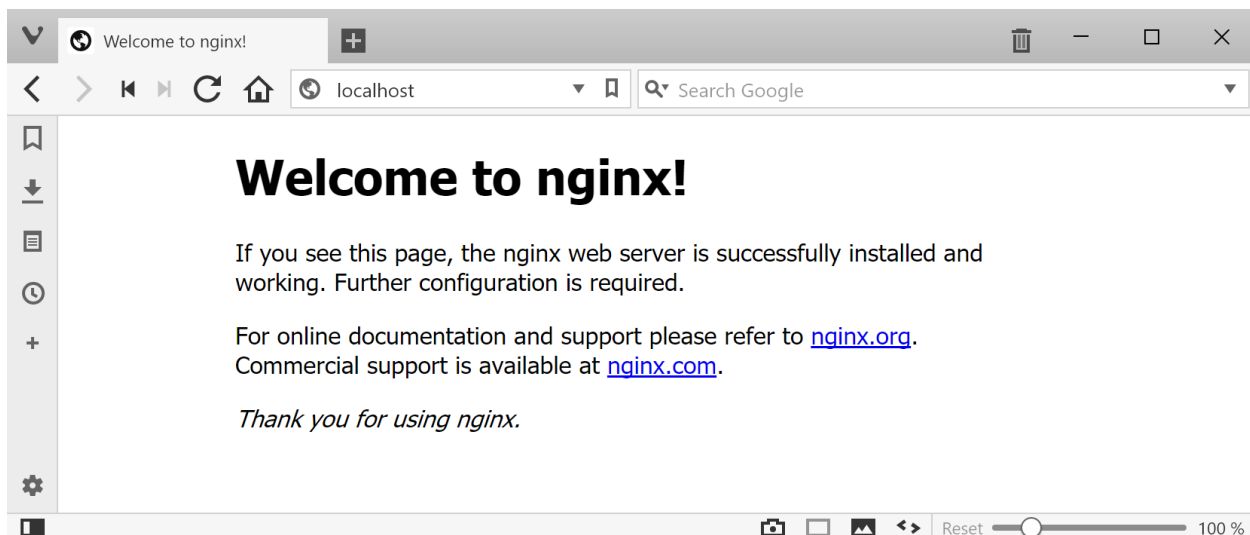


Figure 2: Browsing the Web Server Inside the Container



Tip: On older versions of Mac and Windows (using the Docker Toolbox), the Docker server is running inside a Linux VM on VirtualBox, which will have its own IP address so that you won’t use the `localhost` address. In order to access ports mapped from Docker containers, you can find the IP address of your Docker VM by running `docker-machine ip`, which will give you an IP address like `192.168.99.100`—and that’s where you browse.

Docker is an ideal platform for long-running background services. The Nginx web server running in this container uses next to zero resources unless someone is accessing the site—but the container doesn't have a resource limit, which means that under peak load the Nginx container can grab more resources, and the container process can max out 100% of CPU and memory as if it was running directly on the host.

You can run hundreds of background containers on a modestly specified server this way. Provided the usage patterns are varied and the container loads don't all peak at the same time, the host can happily share resources between all the containers.

Hello Ubuntu

The last type of container is one you run interactively. It stays alive as long as you're connected to it with the Docker CLI, and it behaves like a remote connection to a separate machine. You can use containers in this way to evaluate images, to use images as software tools, or to work through the steps when you're building up your own image.

The majority of Docker Hub images use Linux as the base OS, and Ubuntu is one of the most popular base images. Canonical publishes the official Ubuntu image, and they have integrated Docker Hub with their release cycle so that the latest Ubuntu versions are available on the Hub. You can run an interactive Ubuntu container using Code Listing 7's command.

Code Listing 7: Running an Interactive Ubuntu Container

```
$ docker container run --interactive --tty ubuntu:16.04  
root@dafaf06d4ceb:/#
```

With the **--interactive** and **--tty** flags, Docker runs the container interactively with terminal emulation (it's commonly abbreviated to **-it**). The container is still running on the Docker server, but the client maintains an open connection to it until you exit the container. We're using the Ubuntu official image, but in the run command we've specified a particular version of the image—**16.04**—that gives us the current Long Term Support version of Ubuntu.

However, if you're used to working with Ubuntu, you'll find that the version running in this container won't behave in the same way as the full Ubuntu Server edition. Code Listing 8 shows that the normal Linux commands, such as **ls** and **cat**, work as expected in the container.

Code Listing 8: Linux Commands in the Ubuntu Container

```
root@dafaf06d4ceb:/# ls -l /tmp  
total 0  
  
root@dafaf06d4ceb:/# cat /etc/hosts  
127.0.0.1        localhost  
::1             localhost ip6-localhost ip6-loopback
```

```
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.5      dafaf06d4ceb
```



Note: There are some interesting entries in the `hosts` file. Docker injects some runtime details about the container into that file in order to help with discoverability between containers. We'll see more of that in [Chapter 5, Orchestrating Systems with Docker](#).

The version of Ubuntu in the Docker image is a heavily stripped down version of Ubuntu Server, which means some of the most basic utilities aren't available. In order to edit the `hosts` file, we might expect to use the Nano text editor, but it's not installed, and if we try to install it, we'll see the software libraries aren't up to date in the image either, as shown in Code Listing 9.

Code Listing 9: Missing Utilities in the Ubuntu Image

```
root@dafaf06d4ceb:/# nano /etc/hosts
bash: nano: command not found
root@dafaf06d4ceb:/# apt-get install nano
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package nano
```

You can still use the container like any other Ubuntu installation so that you can update the package repositories with `apt-get update` and install whichever tools you like. But you're only changing this instance of the container, not the underlying image. When you run the `exit` command, the container will be stopped but your changes won't be saved. The next time you run a container from the Ubuntu image, it will be the same minimal version of the OS.

The Docker Hub has many such images that are intended to be used as a base image for your own apps. [Ubuntu](#), [Alpine](#), and [BusyBox](#) are popular, and they're deliberately minimal. Having less software installed means less bloat, which means the images are smaller, and it also means a reduced attack vector because there are fewer packages with potential vulnerabilities.

Application containers are not meant to be treated like VMs or physical servers—you wouldn't normally connect to a running container to fix an issue or patch the OS. Containers are so cheap to build and run that you would update the image instead, using a newer version of the base image if it had patches, then creating a new container and killing the old one. Not only will that fix the problem with your application, but it will also give you an updated image with the problem fixed for any future containers you run.

Summary

We've seen what Docker does, learned how to use the Docker client, and looked at how the Docker Engine runs on different platforms. We've walked through the main usage patterns with Docker, running short-lived task containers that do a single job and then exit; we've looked at long-running containers that keep background tasks running as long as the container is running; and we've examined interactive containers that exist as long as your client keeps an open connection with them.

The foundations of Docker are very simple—applications are packaged into images, images are used to run containers on the Docker server, and the Docker client manages the containers. There's a lot more to learn about Docker, and in the rest of this e-book we'll cover the features that have made Docker a revolutionary technology in software delivery.

In the next chapter, we'll look at packaging your own applications as Docker images.

Chapter 2 Packaging Applications with Docker

Meet the Dockerfile

In order to package up your own Docker image, you write a text document called a Dockerfile that lists all the steps needed to both make the image and use the Docker command line to build it. The Dockerfile uses a very simple domain-specific language that only requires a handful of instructions. Code Listing 10 shows a perfectly valid Dockerfile.

Code Listing 10: A Simple Dockerfile

```
FROM ubuntu

RUN apt-get update && apt-get install nano
```

When you build an image from that Dockerfile and run a container from the image, you'll be working in an Ubuntu container with the **nano** package installed. The **FROM** instruction specifies the base image so that your image will start from there and layer on the changes in the rest of your Dockerfile. In this case, it will run two **apt** commands to install Nano.

In order to build an image, you use the **docker image build** command. You need to specify both a repository name to identify the image, and the path Docker should use as the context for building the image. You can also tag images with labels, which explains how you can have multiple image versions in a repository (like **ubuntu:12.04** and **ubuntu:14.04**). Code Listing 11 builds an image using a file called **Dockerfile** in the local directory.

Code Listing 11: Building the Docker Image

```
$ docker image build --tag dockersuccinctly/ubuntu-with-nano .

Sending build context to Docker daemon 2.048 kB

Step 1/2 : FROM ubuntu

latest: Pulling from library/ubuntu

Digest: sha256:34471448724419596ca4e890496d375801de21b0e67b81a77fd6155ce001edad

Status: Downloaded newer image for ubuntu:latest

---> ccc7a11d65b1

Step 2/2 : RUN apt-get update && apt-get install nano

---> Running in 53cccf9021fb
```

```
...
---> 9ef68677ce6a

Removing intermediate container 53cccf9021fb

Successfully built 9ef68677ce6a

Successfully tagged dockersuccinctly/ubuntu-with-nano:latest
```



Tip: *Dockerfile (with no extension) is the default filename Docker looks for, but you can call your Dockerfile anything and identify it with the `--file` option. This means you can call your file `server.dockerfile` and build it with `docker image build --file server.dockerfile`.*

The **image build** command gets executed by the Docker server—the client simply sends the details through. That’s why you must specify a path (using `.` for the current working directory in this case). The client sends the contents of the path to the server, and the server stores it in a working folder it uses to build the image. That folder is called the build context, and later in this chapter we’ll see why it’s important.

You can also give your image a repository name. When you build locally you can call your image anything you like, but the convention is to use a format such as `{user}/{application}`, where the **user** part is your account ID on Docker Hub.

The tag is the unique identifier for a particular image within a repository, which means that in the public registry on the Docker Hub you’ll see images with repository names like [microsoft/azure-cli](#) and [sixeyed/hadoop-dotnet](#) that each have many image versions. If you don’t specify a version in a tag, Docker uses the default **latest**.



Note: *Some images on the Hub don’t have a user in the repository name, which means the Ubuntu image is simply called `ubuntu` rather than `canonical/ubuntu`. These are from [official repositories](#) that are curated, verified, and security scanned. You should use official images for your base image as a preference.*

When you successfully build an image, it’s stored in the Docker server’s local cache and you can run containers from it. You can also push it to a shared image registry such as Docker Hub or your own registry (which we’ll cover in [Chapter 3, Image Registries and the Docker Hub](#)).

The Docker CLI can list all the images stored in the Docker image cache with the **image ls** command, as shown in Code Listing 12.

Code Listing 12: Listing Images in the Engine Cache

```
$ docker image ls
```

REPOSITORY SIZE	TAG	IMAGE ID	CREATED
dockersuccinctly/ubuntu-with-nano ago 165.9 MB	latest	b06d1e92b27e	3 minutes
ubuntu ago 126.4 MB	latest	f8d79ba03c00	8 days
hello-world ago 1.848 kB	latest	c54a2cc56cbb	7 weeks
nginx ago 69.3 MB	alpine	5ad9802b809e	8 weeks

The output from the **docker image ls** command tells you the repository name and the tag for each image, its unique image ID, when it was created in the cache, and the size. In this output, I have three images downloaded from the Hub for the containers I ran in Chapter 1, and I have my own newly built image with the **dockersuccinctly** account name. When you start using larger images, your local cache can use a lot of disk space—we'll see how to manage that in [Chapter 4, Data Storage in Docker](#).

The main Dockerfile instructions

The only required Dockerfile instruction is **FROM**, which specifies the base image on top of which a new image will be built. Of course, that doesn't do much on its own, but to build useful production-grade images you need only a few more instructions:

- **RUN**: Execute a command.
- **ENV**: Set environment variables.
- **COPY**: Copy files from the build context into the image.
- **EXPOSE**: Expose ports from the container that can be mapped to ports on the host.
- **VOLUME**: Create a directory inside the image that can be mapped to external storage.
- **CMD**: Specify the command to run when the container starts.

Here's a very simple Dockerfile that shows all the main instructions—this image is for a basic app that listens for input on a specific port and echoes out any input it receives to a file. Code Listing 13 shows the Dockerfile in full.

Code Listing 13: Dockerfile for an Echo Application

```
FROM ubuntu
RUN apt-get update && \
    apt-get install -y netcat-openbsd
ENV LOG_FILE echo.out
COPY ./echoserver.sh /echoserver.sh
RUN chmod +x /echoserver.sh
EXPOSE 8082
VOLUME /server-logs
CMD /echoserver.sh
```



Note: The full code is on GitHub at [SyncfusionSuccinctlyE-Books/Docker-Succinctly](https://github.com/SyncfusionSuccinctlyE-Books/Docker-Succinctly), and a built image is available on the Docker Hub in the repository [dockersuccinctly/echoserver](https://hub.docker.com/r/dockersuccinctly/echoserver). The order of instructions is important, as we'll see soon, but in this example the instructions are ordered to make them easy to follow.

As Docker builds that image, it will process each of the following instructions:

- Download the latest **ubuntu** image if it's not already in the local cache.
- Update the package list and install the **netcat** package.
- Set an environment variable called **LOG_FILE** with the value **echo.out**.
- Copy the script called **echoserver.sh** from the build context into the root folder in the container.
- Make the script executable on the container.
- Expose port 8082 from the container.
- Create a filesystem mount called **server-logs**.



Note: It's important to understand that those commands are happening inside containers during the build process, not on your local machine. After you've built this image on your machine, the image will have **netcat** installed and have a value set for the **LOG_FILE** environment variable, but your local machine won't.

Everything except the final **CMD** instruction gets executed during the build. When you run a container from the image, Docker uses the **CMD** instruction to tell it how to start—in this case by running the **echoserver.sh** script. That script starts [netcat](#) listening on port 8082 and redirects the output from client connections to a file. The file path uses the volume named in the Dockerfile and the log file name from the environment variable. Code Listing 14 shows how to start the echo server container and find the virtual IP address of the container.

Code Listing 14: Running the Echo Server Container

```
$ docker container run --detach --publish 8082:8082 --name echo-server  
dockersuccinctly/echoserver  
  
7a372ff9f350995b4bb8a84215cd8020bd87dbd196367935dab568ed1939cc5f
```

The container is now running netcat listening on port 8082, and Docker is forwarding requests to port 8082 on the local host into the container.

In Code Listing 15, we connect to the container using **netcat** on the host, specifying the localhost address and the published port 8082. Then we write a string to **netcat** and exit the connection.

Code Listing 15: Connecting to the Echo Server from the Host

```
$ nc localhost 8082  
  
Hello, Docker Succinctly!  
  
^C
```



Note: If you're running on Windows, the **netcat** utility won't be available, but you can install a version from <https://eternallybored.org/misc/netcat/>.

And lastly, in Code Listing 16, we use **docker container exec** to run a command inside the container and write the output back to the host. In this case, we read the contents of the output file that **netcat** on the container is using—this is an echo of the string we sent from the client.

Code Listing 16: Viewing the Echo Server's Log File

```
$ docker container exec echo-server cat /server-logs/echo.out  
  
Hello, Docker Succinctly!
```

By using a volume for the output location and an environment variable for the file name, we can change where the echo data is written for different instances of the container when we run them. We'll look more closely at Docker volumes in [Chapter 4, Data Storage in Docker](#).

This is a simple example of a standard approach to Dockerfiles. A typical Dockerfile for packaging an application will state a minimal base image, install the application platform, copy in the application source, compile the application, and specify how the application starts.

You can vary that approach. For example, if your platform has an official image, you can use that as the base, and if you can publish your app with all its dependencies, you can compile first, then copy the binaries into the container. There's a balance between the portability of the Dockerfile, dependencies on third-party resources, and the size of the built image.

How Docker builds an image

Docker uses a layered filesystem for images. Starting from the base image, the Docker server runs a temporary container from the image for each instruction in the Dockerfile, executes the instruction, then saves the temporary container as a new image, adding it to the local image cache. Docker uses the cache during the build process, which means that if it finds an image matching the current instruction stack—that is, one that matches the state you're asking Docker to create—it will reuse the cached image.

You can write your Dockerfile to make maximum use of the cache by ensuring that the Dockerfile is correctly structured and that the Dockerfiles for different applications each have similar structures. This way, they will use cached images as much as possible. Ideally, when you build apps with similar dependencies, Docker will need only to execute instructions in new layers that are specific to the application. For instance, the Dockerfiles in Code Listing 17 and Code Listing 18 are identical up to the final **COPY** instruction.

Code Listing 17: Dockerfile 'A'

```
FROM ubuntu

RUN touch /setup.txt

RUN echo init > /setup.txt

COPY file.txt /a.txt
```

Code Listing 18: Dockerfile 'B'

```
FROM ubuntu

RUN touch /setup.txt

RUN echo init > /setup.txt

COPY file.txt /b.txt
```

When you build an image from the first Dockerfile, it will create new layers for both of the **RUN** instructions and the final **COPY** instruction. Code Listing 19 builds an image from the first Dockerfile by using the **-f** flag to specify the source Dockerfile name.

Code Listing 19: Building Dockerfile 'A'

```
$ docker image build -t dockersuccinctly/a -f a.dockerfile .  
Sending build context to Docker daemon 4.096 kB  
Step 1 : FROM ubuntu  
---> f8d79ba03c00  
Step 2 : RUN touch /setup.txt  
---> Running in c9761757ff3c  
---> e4d6c1754277  
Removing intermediate container c9761757ff3c  
Step 3 : RUN echo init > /setup.txt  
---> Running in 0f63b9763bef  
---> 3050c9fc2760  
Removing intermediate container 0f63b9763bef  
Step 4 : COPY file.txt /a.txt  
---> f339e6dd38bb  
Removing intermediate container c1c6ba5469a5  
Successfully built f339e6dd38bb
```

Let's note a few things here. The first instruction in Step 1 finds a match in the cache because we've already downloaded the **ubuntu** image, which means Docker simply writes the ID of the cached image it's going to use (starting **f8d**).

For Step 2 there is no match, so Docker runs a temporary container from the **f8d** image, executes the command, and saves the temporary container to a new image with the ID starting **e4d**. Similarly, for Steps 3 and 4, there is no match in the cache, which means Docker runs a temporary, intermediate container from the image in the previous step, saves the container as a new image, and removes the intermediate container.

The **docker image history** command displays all the layers in an image, as in Code Listing 20, which shows the layer history for the **dockersuccinctly/a** image.

Code Listing 20: History of the 'A' Image

```
$ docker image history dockersuccinctly/a
```

IMAGE	CREATED	CREATED BY
-------	---------	------------

```

SIZE
e03337199b8c      3 seconds ago      /bin/sh -c #(nop) COPY
file:9363c0e5fcfd8d7ad  8 B

715ac6bf594b      4 seconds ago      /bin/sh -c echo init > /setup.txt
5 B

1e5d518d70c9      5 seconds ago      /bin/sh -c touch /setup.txt
0 B

f8d79ba03c00      8 days ago         /bin/sh -c #(nop) CMD ["/bin/bash"]
0 B

<missing>          8 days ago         /bin/sh -c sed -i
's/^#\s*\s*(deb.*universe\)$/' 1.895 kB

<missing>          8 days ago         /bin/sh -c rm -rf /var/lib/apt/lists/*
0 B

<missing>          8 days ago         /bin/sh -c set -xe  && echo '#!/bin/sh' >
/u      745 B

<missing>          8 days ago         /bin/sh -c #(nop) ADD
file:a2427e00553ce3905b 126.4 M

```

There is a lot of detail in there. The “missing” layers mean we don’t have the intermediate layers in our cache, because they’re part of the base Ubuntu image. Docker downloaded that image from the Hub—we didn’t build it locally, which means we don’t have all the layers. But we can see part of the instructions that went into building the final Ubuntu image, which is the one with ID **f8d** that our image started with. There are three layers above that, one for each of the instructions in the Dockerfile.

If you now build an image from the second Dockerfile, it will find matching images in the cache for the first two instructions. Docker tells you it’s found a cache hit in the build output, as we see in Code Listing 21.

Code Listing 21: Building Dockerfile ‘B’

```

$ docker image build -t dockersuccinctly/b -f b.dockerfile .

Sending build context to Docker daemon 4.096 kB

Step 1 : FROM ubuntu

---> f8d79ba03c00

Step 2 : RUN touch /setup.txt

---> Using cache

---> 1e5d518d70c9

```

Step 3 : RUN echo init > /setup.txt

---> Using cache

---> 715ac6bf594b

Step 4 : COPY file.txt /b.txt

---> 8cd641b1af84

Removing intermediate container 6d924ab8d087

Successfully built 8cd641b1af84

The Docker server has run only an intermediate container to execute the **COPY** instruction in Step 4 because there was no match in the cache for that. Everything up to that point can come from the cache because the Dockerfile instructions are identical to the cached layers. If we look at the layers in the **dockersuccinctly/b** image, we'll see the bottom seven layers match the **dockersuccinctly/a** image and that only the final layer is different, as in Code Listing 22.

Code Listing 22: History of the 'B' Image

```
$ docker image history dockersuccinctly/b
```

IMAGE SIZE	CREATED	CREATED BY
8cd641b1af84 file:9363c0e5fcfd8d7ad 8 B	About a minute ago	/bin/sh -c #(nop) COPY
715ac6bf594b 5 B	5 minutes ago	/bin/sh -c echo init > /setup.txt
1e5d518d70c9 0 B	5 minutes ago	/bin/sh -c touch /setup.txt
f8d79ba03c00 0 B	8 days ago	/bin/sh -c #(nop) CMD ["/bin/bash"]
<missing> 's/^#s*(deb.*universe\)\$/ 1.895 kB	8 days ago	/bin/sh -c sed -i
<missing> 0 B	8 days ago	/bin/sh -c rm -rf /var/lib/apt/lists/*
<missing> /u 745 B	8 days ago	/bin/sh -c set -xe && echo '#!/bin/sh' >
<missing> file:a2427e00553ce3905b 126.4 MB	8 days ago	/bin/sh -c #(nop) ADD

The image cache saves a huge amount of time when you're building many images with similar instructions. But you need to be careful with the image cache. Each instruction in a Dockerfile is a candidate for adding to the image cache, which effectively preserves the state at the time the image was built.

If Docker finds a match in the cache, it will use that match, which means you need to be sure you only cache layers that are good candidates for reuse and that don't have contents that will become stale. For example, in Code Listing 23 we have the `apt-get update` command in its own `RUN` instruction.

Code Listing 23: A Dockerfile Which Will Become Stale

```
FROM ubuntu  
  
RUN apt-get update
```

When this image gets built, it will cache a layer with an updated package list. If you build an image from a different Dockerfile that starts with the same instructions, it will use the cached image with the saved package list. If we build the new image several months after building the original image, the new image will pick up the old cached layer—it will not run `apt-get update` for the new image. Any subsequent instructions that install packages will be using an old package list.



Tip: How you structure your Dockerfile impacts the speed of the build process, but the image cache can also have a functional impact on the contents of the image. Docker's resource [Best practices for writing Dockerfiles](#) is worth getting to know. One of its key recommendations is that you combine multiple commands in a single `RUN` statement in order to prevent unintended cache hits.

Summary

Packaging applications into a Docker image is simple—you specify the base image, install dependencies, copy in your application files, and tell Docker the command to run when a container starts from the image. You will need to consider the workflow for your Dockerfile so that it builds efficiently, and you'll also need to be aware of the Docker image cache and how it can impact your builds.

The majority of the work that goes into the Dockerfile is about optimizing your image. Typically, for a production image you want the smallest possible image in order to keep your app secure and make it easy to move around. The official base images on the Docker Hub are a good place to start, and in the next chapter we'll have a closer look at the public Hub and other image registries.

Chapter 3 Image Registries and the Docker Hub

About image registries

Packaging an application into a container image is easy with the `docker image build` command, but the image only gets created in the local image cache on your own machine. In order to make that image useful, you need to be able to ship it to different hosts, and that's where image registries come in. A registry is simply a shared location for images—you can push images you've created and pull images that you or other people have created. The registry is a core concept in Docker, and you use the normal Docker command-line tool to work with registries.

The [Docker Hub](#) is a free, public registry for images maintained by Docker, Inc. It is a hugely popular service—at the time of writing there are more than 900,000 images on the Hub, and there have been more than 12 billion image pulls. Anyone can push images to the Hub, and in addition to the community images, there are also official images curated by Docker and supported by their owners.

It's best to start with the Docker Hub, which is the default source and has a very useful feature set that makes it much more than just an image store. However, if you're not comfortable storing your own application images in a third-party cloud service, there are many other options. In this chapter, we'll look at using the Hub, choosing images, and other registry options.

Using the Docker Hub

The Docker Hub is a public image registry provided by Docker Inc. (we'll look at the relationship between Docker and Docker, Inc. in [Chapter 9, Continuing Your Docker Journey](#)). Docker Hub is the default registry for the Docker Engine, and you can pull images without needing an account. So far, when we've run containers, the images have been downloaded from the Docker Hub without us having to configure anything.

You will need your own account on the Hub if you want to push images and share them. You can register with the Hub at <https://hub.docker.com> and create a free account. The username you choose will be the user part of the repository name when you build your own images (my username on the Hub is [sixeyed](#), and I can only push images which are prefixed **sixeyed**). The Hub can be used for storing private repositories, too, and the free plan lets you store one private repository and unlimited public ones.



Tip: Don't be confused by the terminology here. A “registry” is a service for hosting lots of image repositories, potentially from different users. A “repository” is an image library that potentially holds many different tagged images. For instance, the Docker Hub

is a public registry, and one of its users is Microsoft. Microsoft has a repository for .NET Core, [microsoft/dotnet](https://hub.docker.com/r/microsoft/dotnet), offering many image versions. You can pull a specific .NET Core image from the microsoft/dotnet repository on the Docker Hub registry.

When you have an account registered with the Docker Hub, you will need to add your credentials to the Docker CLI using **docker login**, as shown in Code Listing 24.

Code Listing 24: Logging in to Docker Hub

```
$ docker login

Login with your Docker ID to push and pull images from Docker Hub. If you don't have
a Docker ID, head over to https://hub.docker.com to create one.

Username: sixeyed

Password:

Login Succeeded
```

You can now push images as well as pull them. My Docker Hub account also has access to the [dockersuccinctly](https://hub.docker.com/r/dockersuccinctly) account, so I can push images there too. The **image push** command is very simple, as we see in Code Listing 25.

Code Listing 25: Pushing a Local Image to Docker Hub

```
$ docker image push dockersuccinctly/echoserver

The push refers to a repository [docker.io/dockersuccinctly/echoserver]

b9ff8817e1e9: Pushed
b859a9603b97: Pushed
4e31876babbb: Pushing [=====] 16.87 MB/39.35 MB
4699cbd1a947: Mounted from library/ubuntu
2bed5b3ec49f: Mounted from library/ubuntu
3834bde7e567: Mounted from library/ubuntu
d8d865b23727: Mounted from library/ubuntu
```

The push works intelligently, pushing only layers that aren't already available in the registry. In this case, the base image layers are from the official Ubuntu repository (technically called **library/ubuntu**), which means they're not pushed, and the command flags them as "mounted" instead. Only the new layers created by building my Dockerfile get pushed. If I modify the Dockerfile and rebuild, and if only the top layer changes, when I push a new version only the top layer will be uploaded.

That repository is now created in the Hub, and anyone can run containers from my image with a **docker container run** command specifying the image tag. As we have seen, when we run a container, Docker checks the image cache to see if it has a copy of the image locally. If not, it will pull the image from the Docker Hub.

Image size can be a few megabytes or many gigabytes, so pulling large images might take a while, which means startup time for a new container can jump from seconds to minutes. In order to prevent that, you can explicitly pull images from the Hub and download them before they are needed, so that when you do run a container it uses the local image, as in Code Listing 26.

Code Listing 26: Pulling an Image

```
$ docker image pull dockersuccinctly/echoserver

Using default tag: latest

latest: Pulling from dockersuccinctly/echoserver

Digest: sha256:9f6b1e1fb9955d4a985f04b1f85166421faf4fe414e14921ccd3f431e35833e6

Status: Downloaded newer image for dockersuccinctly/echoserver:latest
```

Tags and image versions

So far we've mostly referred to images by their repository name without specifying a particular tag. Docker image repositories use the basic format **{user}/{app}**, but often the tag is used to specify a release version, or a variant of the image, in the format **{user}/{app}:{tag}**. If you don't specify a tag when you refer to an image, the default **latest** tag is used. You can see this when you list out your image cache with the **docker image ls**. Code Listing 27 shows the various Ubuntu images in my local cache, using the filter flag.

Code Listing 27: Listing Images for a Repository

```
$ docker image ls --filter reference=ubuntu
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu 126.4 MB	16.04	f8d79ba03c00	8 days ago	
ubuntu 126.4 MB	latest	f8d79ba03c00	8 days ago	
ubuntu 126.4 MB	xenial	f8d79ba03c00	8 days ago	
ubuntu 126.4 MB	xenial-20160809	f8d79ba03c00	8 days ago	

ubuntu MB	14.04	90d5884b1ee0	3 months ago	188
--------------	-------	--------------	--------------	-----



Note: One image can have multiple tags. At the time of writing, if you pull `ubuntu`, `ubuntu:xenial`, or `ubuntu:16.04`, you'll get the same image, one with the ID that starts `f8d`. When Canonical releases a patch to Ubuntu, it uploads a new version with multiple tags—by codename and version. This is useful because you can base your images on `ubuntu` and you'll always use the latest patched version when you build. If you're more specific, your build will use the same image. If you have an old image based on `ubuntu:14.04` and rebuild it now, it will still be based on 14.04.

Tags can be version numbers, commit labels, codenames—any useful way to classify images. An established, active image repository will have multiple images that make good use of the tag in order to tell you how the images are different. Table 1 shows some of the images available in the [official Elasticsearch repository](#), with image sizes and age at the time of writing.

Table 1: Image Tags for Elasticsearch

Tag Name	Compressed Size	Last Updated
5	157 MB	5 days ago
5.0	157 MB	5 days ago
5.0.0	157 MB	5 days ago
5.0.0-alpha5	157 MB	5 days ago
latest	152 MB	5 days ago
2	152 MB	5 days ago
2.3	152 MB	5 days ago
2.3.5	152 MB	5 days ago
2.2	154 MB	5 days ago

From that output we can clearly see that there is an old version **2.2**, there's a prerelease version **5.0.0-alpha5**, and there's a **latest** tag for the current release. They're all the same age, which suggests Elastic has an automated process that builds and pushes all versions at the same time. On the main repository page on the Hub, we can also see that many of the tags point to the same image, as in Figure 3.

- 1.5.2 , 1.5 (1.5/Dockerfile)
- 1.6.2 , 1.6 (1.6/Dockerfile)
- 1.7.5 , 1.7 , 1 (1.7/Dockerfile)
- 2.0.2 , 2.0 (2.0/Dockerfile)
- 2.1.2 , 2.1 (2.1/Dockerfile)
- 2.2.2 , 2.2 (2.2/Dockerfile)
- 2.3.5 , 2.3 , 2 , latest (2.3/Dockerfile)
- 5.0.0-alpha5 , 5.0.0 , 5.0 , 5 (5.0/Dockerfile)

Figure 3: Elasticsearch Images with Multiple Tags

We can use `docker container run elasticsearch` to start a container using the latest image, which is currently version 2.3.5; if we want to try the alpha 5 release of version 5, we specify the tag `docker container run elasticsearch:5.0.0-alpha5`; and if we want to stick with an old version, we can use `docker container run elasticsearch:1.7`. The same is true if we're using that image as a base for our own containers. The `FROM` instruction in the Dockerfile supports image tags, too.



Tip: It's a good idea to specify a tag for base images in your `FROM` instruction. If you don't, the image tagged as `latest` will be used. That tag can change to a different image without warning, and there can be breaking changes between image versions, which means your own images might fail to build. If you have an old image built on `ubuntu:14.04` but the Dockerfile specifies `FROM ubuntu`, the base image will be `ubuntu:16.04` the next time you build it, which is functionally different.

Automating builds

Manually building and pushing images is fine for apps that don't change frequently, but if you're doing regular releases you will want to automate building and pushing your images. Docker Hub supports automated builds for images when the Dockerfile and source is in GitHub or Bitbucket, which means you can set up the Hub to build and tag images whenever you commit changes.

The automated build setup is only available when you're logged in to the Docker Hub. From the top menu, select **Create > Create Automated Build**. From here, you can link your GitHub or Bitbucket account and select the source git repository—you'll see the screen in Figure 4.

Repository Namespace & Name*

sixeyed

hadoop-succinctly

Visibility

public

Short Description*

Max 100 Characters

By default Automated Builds will match branch names to Docker build tags. [Click here to customize](#) behavior.

Customize Autobuild Tags

Your image will build automatically when your source repository is pushed based on the following rules. [Revert to default settings](#)

Push Type	Name	Dockerfile Location	Docker Tag	
Branch	master	/	latest	+
Tag	2.7.2	/	2.7.2	-

Create

Figure 4: Creating an Automated Build in Docker Hub

After the build is set up, any pushes to the git repo will trigger a build on Docker Hub. You can also manually trigger a build, and Docker Hub will pull in the source files, run **docker image build** on its own servers, and upload the image to your Docker Hub repository.

Automated builds can be set up to create different image tags from different branches or tags from the source git repository. Figure 5 shows the automated build configuration for one of my public images.

PUBLIC | AUTOMATED BUILD


sixeyed/hadoop-dotnet ☆

Last pushed: a month ago

[Repo Info](#)[Tags](#)[Dockerfile](#)[Build Details](#)[Build Settings](#)[Collaborators](#)[Webhooks](#)[Settings](#)

Build Settings

☒ When active, builds will happen automatically on pushes.
The build rules below specify how to build your source into Docker images. The name can be a string or a regex. The Docker Tag name may contain variables. We currently support {sourceref}, which refers to the source branch/tag name. [Show more](#)

 Source Repository
[sixeyed/dockers](#)

Type	Name	Dockerfile Location	Docker Tag Name	
<div>Branch</div>	<div>master</div>	<div>/hadoop-dotnet</div>	<div>latest</div>	<div>+ Trigger</div>
<div>Tag</div>	<div>2.7.2</div>	<div>/hadoop-dotnet</div>	<div>2.7.2</div>	<div>- Trigger</div>

Figure 5: An Automated Build with Multiple Image Tags

There are two builds set up, one that uses the GitHub **master** branch as the source for the **latest** image tag and one that uses the git tag **2.7.2** for an image with the same tag name. I use that image for Hadoop training, specifying the tagged version **2.7.2** in the material. If I update the image with a new version of Hadoop, I'll use a new tag. The old image won't change and people using old content can still work along with it. But if I find a problem with the **2.7.2** image, I can update it with the same tag so that it remains correct for the training.

Choose your base image carefully

Docker Hub is the default registry, and there are thousands of high-quality images available there, but only the official repositories are curated. There are no quality gates on any other images in the Hub, which means anyone can build anything into an image. You need to be careful what you choose to run or to use as a base image.

The official repositories are the best place to start. Not only are they actively supported, but the Docker Hub scans the images against a security vulnerability database and will flag any issues it finds in the image.

You can see the results of the security scan if you're logged in to the Docker Hub. Figure 6 shows the current status of the official Ubuntu image and some other interesting details.





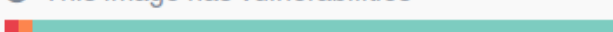

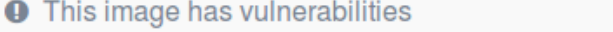
16.10 Compressed size: 39 MB Scanned 9 days ago	! This image has vulnerabilities 
latest Compressed size: 50 MB Scanned 9 days ago	! This image has vulnerabilities 
xenial Compressed size: 50 MB Scanned 9 days ago	! This image has vulnerabilities 
xenial-20160809 Compressed size: 50 MB Scanned 9 days ago	! This image has vulnerabilities 
16.04 Compressed size: 50 MB Scanned 9 days ago	! This image has vulnerabilities 
trusty-20160802 Compressed size: 66 MB Scanned 9 days ago	! This image has vulnerabilities 
14.04 Compressed size: 66 MB Scanned 9 days ago	! This image has vulnerabilities 

Figure 6: Security Scan Results for Ubuntu Images

We can see that the image for 16.04 is 16 MB smaller than version 14.04, and it currently has two known security vulnerabilities that you can drill into to find. The image for 14.04 currently has 11 vulnerabilities. If you have images built from **ubuntu:14.04**, this information tells you it's worth upgrading and rebuilding your images in order to get a smaller image and close off some attack vectors. The image for 16.10 is smaller still, which means Canonical is actively minimizing the image size with each new release.

Only a small number of official repositories exist, however. In the rest of the Hub, you may find several images that fit the requirements for a base image for your app, but you must be sure they are trustworthy.

If the image is set up as an automated build, you can check the Dockerfile in GitHub and be confident that the image came from that Dockerfile because it was built and pushed by Docker's servers. For images that aren't automated, you can't guarantee that.

Many images have a link to the Dockerfile, but if the image was manually built and pushed by a user, the contents might not match what's in the Dockerfile. Some images don't have a Dockerfile link at all. That doesn't mean they're poor quality or untrustworthy—the star rating and number of pulls might suggest they're fine—but pulling the image first is a good precaution. After doing that, you can run **docker image history** and look through the instructions to get an idea what went into building the image.



Note: *If you find the perfect base image and it's not from an official repository, the publisher is not required to keep that image up-to-date or even available. If you build from another user's base image on the Hub, that repository can be deleted, leaving you unable to build your own image. If that's a concern, you should consider cloning the source and building your own version of the base image so that you can control the image stack up to a reliable, official source.*

Running your own registry

Nearly all of the images on the Docker Hub are packaged versions of free, open-source software. For commercial or sensitive applications, you might not want your images built and hosted on a public service, even in a private repository. In that case, you can run your own registry on your own hardware with Docker's **registry** image.

Yes, it's a Docker image available on the public Docker Hub that packages an application for running a private image registry you can use on-premise. Code Listing 28 shows how to run a simple local registry.

Code Listing 28: Running Your Own Image Registry

```
$ docker container run -d -p 5000:5000 registry:2
```

Docker can use different registries from the default Docker Hub, and you need only to specify the registry location as a prefix to the image repository name. The Docker Registry runs on port 5000, which means that when you have a container running locally with port 5000 mapped, you can reference it at **localhost:5000**. The registry address is actually part of the full image tag, so in order to push an existing image from your cache to your registry, you will need to tag it with the address, as in Code Listing 29.

Code Listing 29: Tagging an Image for a Different Registry

```
$ docker image tag dockersuccinctly/echoserver  
localhost:5000/dockersuccinctly/echoserver
```

You can now push the image to your registry simply by specifying the full image name, as in Code Listing 30.

Code Listing 30: Pushing to a Local Registry

```
$ docker image push localhost:5000/dockersuccinctly/echoserver

The push refers to a repository [localhost:5000/sixeyed/docker-succinctly-echoserver]

b9ff8817e1e9: Pushed
b859a9603b97: Pushed
4e31876babbb: Pushed
4699cbd1a947: Pushed
2bed5b3ec49f: Pushed
3834bde7e567: Pushed
d8d865b23727: Pushed

latest: digest:
sha256:9f6b1e1fb9955d4a985f04b1f85166421faf4fe414e14921ccd3f431e35833e6 size: 1776
```

This gives you a lot of flexibility. You can pull the Ubuntu base image, tag it to give it a completely different name, and push it to your local registry. If you mandate that image as the base for all your in-house container images, they will all use a version of Ubuntu that is in your control. If, in the future, you decide to move to Debian, you can tag the official Debian image with your custom name, replace the version in your registry, and rebuild your containers. They'll all be rebased to use Debian.

With Docker's Registry image, you get the basic push and pull functionality but not the command-line search option, and there is no UI, so it's not feature-equivalent to the Docker Hub. The [REST API](#) for the Registry gives you a lot of functionality, but without a friendly interface. You can list all the repositories in your local registry with an HTTP GET to the `_catalog` endpoint, as in Code Listing 31.

Code Listing 31: Querying the Local Registry with cURL

```
$ curl localhost:5000/v2/_catalog

{"repositories":["dockersuccinctly/echoserver"]}
```

The Docker Registry is quite a technical option. You can set it up as production-grade, load-balanced, secured cluster, but configuring it correctly and maintaining it well requires a good bit of effort.

Other registry options

Because the Registry API is public, compatible registries are available in other products. You can use the integrated container registry in [GitLab](#) as part of a build pipeline so that when you push code, GitLab builds a container and pushes it to the registry. JFrog has a Docker registry add-on for [Artifactory](#). And Docker, Inc. has the [Trusted Registry](#) product, which we'll look at in [Chapter 9, Continuing Your Docker Journey](#).

Summary

We've now seen how to push and pull Docker images using different registries. The most popular registry is the Docker Hub, which has hundreds of thousands of images for all kinds of applications. The Hub has a free service level, and you can push your own images or make use of the automated build system in which the Hub will build images from a GitHub or Bitbucket repository whenever you commit a change.

If you're packaging applications for internal use, you can host your own registry rather than using the public Docker Hub. Doing so can be as simple as running the **registry:2** image on a designated server in your network or running a scalable set of instances on multiple servers or using a commercial registry product.

For cases in which you're running your image registry inside a container, you'll obviously want all your data permanently persisted. However, when it comes to saving data, Docker containers may not behave as you expect. In the next chapter, we'll get a better understanding of Docker volumes.

Chapter 4 Data Storage in Docker

Containers and their data

Docker uses a layered filesystem for containers called the Union File System. When you run a container from an image, all the layers going into the image are mounted in the container as read-only parts of the filesystem. Next, a read-write layer is created on top for this instance of the container. You can change the data inside your container, but this will not change the image itself. If you create a file in a running container and run a second container from the same image, the new container won't have the file you created in the first container.

That restriction guarantees that every container you run from an image will start with exactly the same state because the only way to change an image is to explicitly create a new one. In Docker, if you want to persist data in your container and have it available the next time you run a container from that image, or if you want to share it with other containers or your host machine, you do it with volumes.

Volumes are like virtual disk drives inside the container. When you write to them, you're actually writing to a permanent location on the host machine's disk (or another storage platform), but inside the container it will look just like a local file path. Volumes are owned by one container, but they can be shared with other containers. In this chapter, we'll see how Docker volumes work and what you can do with them.

Images, containers, and volumes

When you run a container, volumes can be added to it or they can be configured in the Dockerfile so that they're part of the image—the same as explicitly specifying ports to be made available with the **EXPOSE** instruction. The **VOLUME** instruction specifies a data volume with a named path that you can use as a file location in the container. Code Listing 32 shows a simple Dockerfile with a single data volume with one file created in it.

Code Listing 32: A Dockerfile with a Volume Instruction

```
FROM ubuntu:16.04

RUN mkdir /v1 && touch /v1/file1.es.txt

VOLUME /v1
```



Note: *touch is a Linux command that creates a new, empty file or updates the timestamp on an existing file. It's a useful utility that Windows doesn't have in the command shell or in PowerShell.*

We can build that image and run an interactive container from it in order to see what happens when we write data inside a container. Code Listing 33 runs a container from that image, which I built with the tag **dockersuccinctly/ubuntu-with-volume**. The container runs interactively (with the **-it** options), and the Docker command also uses the **--rm** option, which will remove the container once it exits.

Code Listing 33: Running a Container with a Volume

```
$ docker container run -it --rm dockersuccinctly/ubuntu-with-volume
```

Now, we can write some data in our running container. This is an Ubuntu container, and we can write to the usual paths—**/** is the filesystem root, and **/v1** is the name of the volume we specified in the Dockerfile. Code Listing 34 shows a new file being written to the volume.

Code Listing 34: Creating a New File in a Volume

```
root@bf780f3e8a15:/# touch /v1/file2.es.txt
```

As long as that container is running, both the files in the volume exist and can be used inside the container. Code Listing 35 shows the output of searching for all the **.es.txt** files on the filesystem, and we can see two files are there.

Code Listing 35: Finding Files in Volumes

```
root@bf780f3e8a15:/# find / -iname '*.es.txt'

/v1/file2.es.txt

/v1/file1.es.txt
```

Now, if we exit the container, what will happen to the data? The container is removed because we ran it with the **--rm** flag, but the files in the volume are in a location on the host, which raises the question: can they still be used? In Code Listing 36, I run a second container from the same image and repeat the **find** command.

Code Listing 36: Checking the Volume in a New Container

```
$ docker container run -it --rm dockersuccinctly/ubuntu-with-volume

root@9dd1e7d81018:/# find / -iname '*.es.txt'

/v1/file1.es.txt
```

Because the new container isn't sharing the volume from the first container, only the original file from the image exists. It has its own data volume on the host. The file from the image was copied into the volume for the new container, but any changes this container makes won't change the image. The only way to add data into a volume that all containers will see is by creating the files in the image using the Dockerfile.



Note: The position of the `VOLUME` instruction in the `Dockerfile` is important. If you try to work with files in a volume after the `VOLUME` instruction, the work will be done in a temporary container during the build process, and because the volume is container-specific, the changes won't be committed to the image. Instead, you should set up the data on the filesystem first, so that it's part of the image, then expose the volume.

Figure 7 shows how data volumes compare with the filesystem layers in an image and how a running container makes use of both, together with its own, dedicated read-write layer.

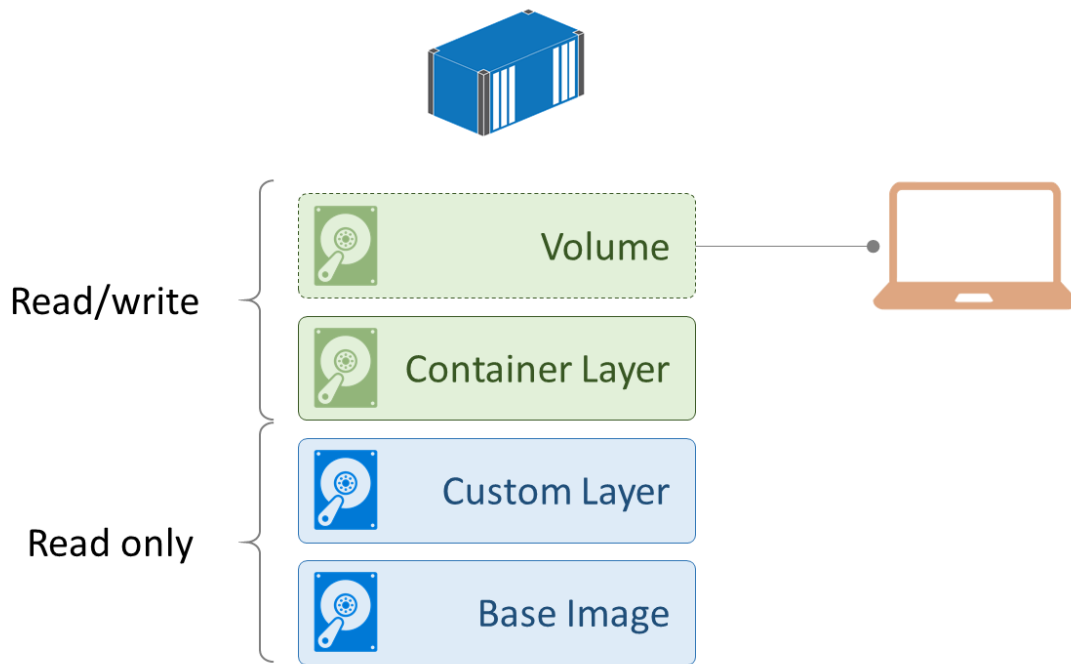


Figure 7: Layers in the Union File System

The setup is different when you map an image volume to a location on the host.

Mounting into data volumes

If you want to look at the data inside a container, you typically must attach to it or execute a command to run inside it. With data volumes, when you run a container, you can mount the volume from a specified path on the host, which means you have a shared folder between the host and any containers using the same mounted volume.

You can mount a volume using an option in the `container run` command similar to publishing a port—you specify the path on the host and the name of the volume to map. Code Listing 37 creates a folder in my home directory, then runs a container from the previous image but mounts the `v1` container from the local directory on the host with the `-v` option.

Code Listing 37: Mounting a Volume from the Host

```
$ mkdir ~/v1  
$ docker container run -it --rm -v ~/v1:/v1 dockersuccinctly/ubuntu-with-volume
```

This action results in different behavior. When a data volume is mounted from the host, files from the image do not get copied into the container volume when it starts because this is a shared location and copying files could overwrite data. So, in this new container we won't have any data in the volume, as Code Listing 38 shows.

Code Listing 38: Listing the Contents of a Mounted Volume

```
root@1f5bf6d15d52:/# ls -l /v1  
  
total 0
```

When we create a file, however, it will be available to the container and to the host because the data will persist in a known location when the container is removed. In Code Listing 39, we add a new file to the volume and leave the container.

Code Listing 39: Writing Data in a Mounted Volume

```
root@1f5bf6d15d52:/# touch /v1/mappedfile1.es.txt  
  
root@1f5bf6d15d52:/# ls -l /v1  
  
total 8  
  
-rw-r--r-- 1 root root 0 Aug 20 18:42 mappedfile1.es.txt  
  
root@1f5bf6d15d52:/# exit
```

Now we have the file on the host, which means that if we start a new container with the same data volume mount, we will see the file from the first container, as in Code Listing 40.

Code Listing 40: Running a New Container with a Mounted Volume

```
$ docker container run -it --rm -v ~/v1:/v1 dockersuccinctly/ubuntu-with-volume  
  
root@642cebc0e1be:/# ls -l /v1  
  
total 8  
  
-rw-r--r-- 1 root root 0 Aug 20 18:42 mappedfile1.es.txt
```

The original file from the image has still not been copied in, but the file from the previous container is available even though that container has been removed.



Tip: *It's good to be clear on the different ways of using Docker volumes. Many images on the Hub use volumes for variable data, such as configuration files, or for permanent state, such as database files. The [Nginx](#) image is a good example—it has a volume defined for website content, and if you mount the volume from a local directory containing your website files, you can run your site in an Nginx container directly from the official image without having to build your own image.*

Mounting a local directory into a container enables many scenarios, but mounting is limited if you need to move your container to a different host—the data it needs will no longer be available on the new host. For those scenarios, Docker supports shared storage engines as the source for volume mounts through the [volume plugin](#) architecture. You can choose a volume plugin that matches your architecture and mount volumes from a shared location available from any host.

For cloud deployments, there are plugins that let you share storage using the platform's services—for example, [Azure File Storage](#) in Azure or [Persistent Disks](#) in Google Compute. Those options are useful when you have clusters of hosts running in a Docker swarm, which we'll look at in [Chapter 6, Clustering Hosts with Docker Swarm](#).

Sharing data volumes

Data volumes exist outside of a container, which means any data written to them persists even when the container isn't running or has been removed. That also means a container's volumes can be shared between containers from the same image or different images, which is useful for scenarios such as backing up databases. You might have a database running in a long-running container and run a task container to back up the data files, sharing the volume from the database container.



Tip: *You may find documentation that recommends using a data-only container that runs a container from a lightweight image, such as [BusyBox](#) with a data volume. In order to make use of this as a data container, stop the container immediately so that it doesn't use any compute—use it only for sharing the volume to other containers. This was common with earlier versions of Docker, before host volume mounting was introduced, but now it isn't needed.*

We can demonstrate sharing volumes for the backup scenario with the official MongoDB image by using the command in Code Listing 41 to run a Mongo database in a background container.

Code Listing 41: Running MongoDB in a Container

```
$ docker container run -d --name mongo mongo
```

The Mongo image stores database files in a volume named **/data/db** that we can access in a separate container by running it with the **--volumes-from** option, which shares the volumes from the named container in the new container. In Code Listing 42, we run an interactive container with access to the database's volumes and list the Mongo data directory contents.

Code Listing 42: Viewing Mongo Data from Another Container

```
$ docker container run -it --volumes-from mongo ubuntu
root@c1bec125113a:/# ls -l /data/db
total 136
-rw-r--r-- 1 999 999    46 Aug 20 20:04 WiredTiger
-rw-r--r-- 1 999 999    21 Aug 20 20:04 WiredTiger.lock
-rw-r--r-- 1 999 999   915 Aug 20 20:06 WiredTiger.turtle
-rw-r--r-- 1 999 999 40960 Aug 20 20:06 WiredTiger.wt
-rw-r--r-- 1 999 999  4096 Aug 20 20:04 WiredTigerLAS.wt
-rw-r--r-- 1 999 999 16384 Aug 20 20:05 _mdb_catalog.wt
-rw-r--r-- 1 999 999 16384 Aug 20 20:05 collection-0--6297223593488391649.wt
drwxr-xr-x 2 999 999  4096 Aug 20 20:06 diagnostic.data
-rw-r--r-- 1 999 999 16384 Aug 20 20:05 index-1--6297223593488391649.wt
drwxr-xr-x 2 999 999  4096 Aug 20 20:04 journal
-rw-r--r-- 1 999 999     2 Aug 20 20:04 mongod.lock
-rw-r--r-- 1 999 999 16384 Aug 20 20:06 sizeStorer.wt
-rw-r--r-- 1 999 999    95 Aug 20 20:04 storage.bson
```

In the new container, the data volume from the Mongo container has the same path—**/data/db**. The new container can see the files created by Mongo in order to physically store the databases, which means you can simply use **tar** to back up all of them to a single archive. This is obviously not a robust backup strategy for a production database, but it is a simple approach that can be useful for testing or development, or to back up and restore the known state.

The volume for this container is stored on the host machine, but because it's not using a mount source, we don't know where the data is physically stored. The **docker container inspect** command gives us that information along with a lot of extra details about the container. Code Listing 43 shows an extract of the output that focuses on the **Mounts** section, which lists the volumes in use by the container.

Code Listing 43: Inspecting Volume Mounts

```
$ docker container inspect mongo
...
    "Mounts": [
      {
        "Name":
"bb05dc88f914ee0129efc68cc1221659a279cb26de0c9742712bdcd6faac464e",
        "Source":
"/var/lib/docker/volumes/bb05dc88f914ee0129efc68cc1221659a279cb26de0c9742712bdcd6faac
464e/_data",
        "Destination": "/data/db",
        "Driver": "local",
        "Mode": "",
        "RW": true,
        "Propagation": ""
      }
    ]
...

```

The key information here is the **RW** flag, which tells us this is a read-write volume (Docker also allows containers to run with read-only volumes), and the **Source**, which tells us where the data is actually stored on the host. Code Listing 44 shows that we can view the content of the volume source on the host machine too, provided we have permission to access it.

Code Listing 44: Listing Volume Contents from the Host

```
$ sudo ls -l
/var/lib/docker/volumes/bb05dc88f914ee0129efc68cc1221659a279cb26de0c9742712bdcd6faac4
64e/_data
total 120
drwxr-xr-x 2 999 docker  4096 Aug 20 21:40 diagnostic.data
-rw-r--r-- 1 999 docker 16384 Aug 20 21:05 index-1--6297223593488391649.wt
-rw-r--r-- 1 999 docker 16384 Aug 20 21:05 collection-0--6297223593488391649.wt
drwxr-xr-x 2 999 docker  4096 Aug 20 21:04 journal
...

```

This is the physical storage location for the files on the host. Containers can see this data in the volume thanks to the virtual filesystem mounting the data into the container. The container doesn't get a copy of the files—it uses the originals directly. If you're running Linux containers on Docker for Mac or Docker for Windows, remember the “host” is really a Linux VM, so you won't see the files on your physical machine.



Note: From the host, and from containers that share the data volume, file permissions are preserved from the original container. In this case, all the data files are writable from outside the Mongo container, which means that an external process can modify or delete data and potentially corrupt the database. If you need to read data in another container, you can use the `:ro` flag to make the target volume read-only, e.g., `--volumes-from mongo:ro`. Checking the contents of the volume on the host is sometimes useful for tracing issues, but in general you shouldn't poke around in `/var/lib/docker`, which is where the Docker Engine keeps its data.

Removing containers, data volumes, and images

Containers end when the process inside them finishes or when they are externally ended with the `docker container kill` or `docker container stop` commands. But the container isn't removed—it's just stopped. The `docker container ls` command won't show the container unless you add the `--all` flag, which will list all containers, including the stopped ones.

If you view Docker as a host for short-lived task containers, this might seem surprising, but there is a consistent approach for all types of containers. A background service container might be stopped and restarted, and while it is stopped you'd expect the container to exist with all its data. For single-task and interactive containers, it's good practice to use the `--rm` flag when you start the container so that Docker will automatically remove the container when it ends.

To explicitly remove containers, use the `docker container rm` command and specify containers by their ID or name, as in Code Listing 45.

Code Listing 45: Stopping and Removing a Container

```
$ docker container stop mongo
mongo
$ docker container rm mongo
mongo
```

Running containers are not deleted when you run the `rm` command, which means you can chain `rm` with `docker container ls` to remove all stopped containers, as in Code Listing 46.

Code Listing 46: Removing All Stopped Containers

```
$ docker container rm $(docker container ls -a -q)
c28b88ef8354
c1bec125113a
c960a3f90043
...
```



Note: This is Linux syntax, but on Windows machines you can use the exact same syntax in PowerShell.

You should also be aware that when you remove a container that has a data volume, the data volume is not automatically removed. In order to remove the data volume along with the container, you need to explicitly state it with the **-v** option: **docker container rm -v {container}**.

We've stopped the Mongo database container and removed it, but we didn't use the **-v** flag, so the data volume with all the database files continues to exist as a directory on the host machine. This is called a dangling volume because there's no container available and it's simply using up space on your host drive. With the Docker CLI you can list all the dangling volumes and chain that command with **volume rm** to remove all the dangling volumes, as in Code Listing 47.

Code Listing 47: Listing and Removing Dangling Volumes

```
$ docker volume ls -qf dangling=true
06d3a6e04d0dfc6034a5c589ac4d5f1ae2a55079052c9b7370c7f060d3e39f5a
09f6df5ecb28f92127d06f2ff3a8724f018e82e0c9bbbc4eb4e28917bd4d8542
0d322921340a086be82c413ac047c841d834a1592aac1266f5c4caf0cb644d44
...

$ docker volume rm $(docker volume ls -qf dangling=true)
06d3a6e04d0dfc6034a5c589ac4d5f1ae2a55079052c9b7370c7f060d3e39f5a
09f6df5ecb28f92127d06f2ff3a8724f018e82e0c9bbbc4eb4e28917bd4d8542
0d322921340a086be82c413ac047c841d834a1592aac1266f5c4caf0cb644d44
...
```

In development and test environments, it's easy to find yourself with a large number of stopped containers and dangling volumes, so it's good practice to put these commands in a cleanup script that you run regularly. The machines on which you regularly build images or pull images to try out can end up with a large image cache. Images are never automatically removed by Docker—they must always be explicitly deleted with the **docker image rm** command.

As with volumes, the Docker CLI includes the concept of dangling images—Docker images that are in the cache but have no containers based off them. You can combine a list of dangling images with the **image rm** command in order to remove unused images and reclaim a lot of disk space, as in Code Listing 48.

Code Listing 48: Listing and Deleting Dangling Images

```
$ docker image rm $(docker image ls -f "dangling=true" -q)

Deleted: sha256:9417bb0f80a5c5eeb72e9aa8fd1a43dc13ce322cf49b1640a3c39f0f994a8b9b

Deleted: sha256:b445166061334c8c206e1d7be26cab455e022f781dce1f916f195d2292b18df2

Deleted: sha256:f4b91f407b83bd8bc0c20063fdadab06af1f2d6c6de7a3fd6bf6c2f55d8928ca

...
```



Note: The CLI also has the *docker system prune* command, which you can use to remove all unused containers, volumes, and images.

Summary

We've explored storage mechanisms in Docker and looked at data volumes and the different ways volumes can be mounted and shared. We've seen that you can provide access to data in containers by mounting volumes from paths on the host or on a storage platform using a volume plugin. Alternatively, you can explicitly share the volumes from one container with other containers, allowing many containers to access the same data.

Volumes and containers persist even when the container has stopped running, and this can clog up your disk. We saw some useful commands for cleaning up volumes, containers, and images. You can run a system prune on a schedule that ensures your environments will stay clean.

If you share volumes between containers, you must be mindful that many processes can attempt to access the same files and end up overwriting or corrupting data. In order to prevent that, data volumes can be shared with read-only access that supports scenarios such as creating backups or indexing data without the risk of modifying it. Docker also has functionality for connecting containers at a higher level so that they can share state without sharing data. We will look at orchestrating with Docker in the next chapter.

Chapter 5 Orchestrating Systems with Docker

Orchestrating multicontainer solutions

When you break down a large application into smaller parts with each part running in a separate container, you'll need a way for containers to work together. That's called orchestration. Orchestration requires a framework that allows communication between containers, configuration of how containers need to be connected, scheduling for container creation, monitoring of health, and scaling. There are external platforms for container orchestration—[Mesos](#) and [Kubernetes](#) are popular—but Docker has orchestration built into the platform.

In this chapter, we'll look at Docker networking, which is a virtual networking layer that underpins communication between containers. With Docker networks you can segregate traffic on your host, allowing you to isolate groups of containers while allowing them to talk to each other. For defining the relationships between containers, we'll look at Docker Compose, which is a client-side tool that works with the Docker service API.

The other side of orchestration is managing containers across multiple hosts, as you would do in a production environment in order to provide scale and reliability. We'll cover that in [Chapter 6, Clustering Hosts with Docker Swarm](#), and we'll see how to apply what we learn in this chapter to a cluster of Docker hosts.

Docker networks

Docker is the platform for your application runtime, and it controls how your container interacts with resources from the host. You can set memory and CPU restrictions on containers, and when your app runs it only sees the resources allocated to it. At the network level, Docker creates a bridge network on your host. A bridge network gives every container its own IP address, and it allows communication between containers and also between containers and the host.



Note: *There are other types of networks supported by Docker. The bridge network is for working with containers on a single host. The overlay network is for working with containers that run across multiple hosts. There is also a plugin system that lets you use third-party network types.*

Creating a network in Docker is easy. Bridge networks are the default, which means you simply need to give the network a name, as in Code Listing 49.

Code Listing 49: Creating a Bridge Network

```
$ docker network create ch05  
ffe636469bc2820e19438f09c9fe484e129879e37c7f93059b7d7fa03ae164fe
```

As usual, when you create an object, Docker responds with a large ID that you can use to refer to the object (or you can refer to the object by its name as well). The bridge network is created with a designated IP address range, and every container in the network will have a dedicated IP address in that range. This is the simplest way to connect containers. In Code Listing 50, we start an Nginx container in the background that joins the new network.

Code Listing 50: Running a Container in the Network

```
$ docker container run -d --name webcontainer --network ch05 nginx:alpine
```

The **--network** option tells Docker to join this container into the **ch05** network we've created, which means it will get an IP address from the network's range. Note that we haven't published port 80 on the container, so we can't browse to **localhost** to see the site, but we can browse to the container's IP address.

We can get the IP address either by inspecting the container or the network. The **network inspect** command in Code Listing 51 lists all the containers in the network and their IP addresses.

Code Listing 51: Inspecting the Network

```
$ docker network inspect ch05  
[  
  {  
    "Name": "ch05",  
    "Id": "ffe636469bc2820e19438f09c9fe484e129879e37c7f93059b7d7fa03ae164fe",  
    "Scope": "local",  
    "Driver": "bridge",  
    ...  
    "Containers": {  
      "e7c511c50943853a9307b859f5290f08ce208380e459d17a858dad6f995ee250": {  
        "Name": "webcontainer",  
        "EndpointID":  
"a65f4cb72fb3c28452fbde375b9a0224605daf89501d0d9e26b2788d03137aa9",
```

```


        "MacAddress": "02:42:ac:14:00:02",
        "IPv4Address": "172.20.0.2/16",
        "IPv6Address": ""
    },
    "Options": {},
    "Labels": {}
}
]

```

The subnet range for this network is **172.20.0.0**, and it's a class B network that allows 65536 IP addresses. The Nginx container, called **webcontainer**, has the IP address **172.20.0.2**, and, because it's a bridge network, we can access that from the host, as we see in Figure 8.



Figure 8: Browsing to the Container's IP Address

 **Note:** The bridge is between the container and the host. If you're running on Linux you can access containers by their IP address. On Docker for Mac and Docker for Windows, remember the host is a Linux VM, so you can't connect directly to the container IP address from your host machine.

Now we can start a second container in the same network, but we'll run it interactively so that we can see how the network connections are set up with the Linux **ip** command, as in Code Listing 52.

Code Listing 52: Running a Second Container in the Network

```
$ docker container run -it --rm --network ch05 sixeyed/ubuntu-with-utils
root@acda23507c84:/# ip -f inet address
...
16: eth0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default link-netnsid 0

    inet 172.20.0.3/16 scope global eth0
```

This container has IP address **172.20.0.3**, which means it's in the same subnet as the Nginx container. Docker has a built-in DNS server, so containers on the same network can access each other by name as well as by IP address, as we can see with the **dig** and **ping** commands in Code Listing 53.

Code Listing 53: Querying the Docker Network

```
# dig webcontainer
...
;; ANSWER SECTION:
webcontainer.      600    IN     A      172.20.0.2

# ping webcontainer -c 2
PING webcontainer (172.20.0.2) 56(84) bytes of data.
64 bytes from webcontainer.ch05 (172.20.0.2): icmp_seq=1 ttl=64 time=0.086 ms
64 bytes from webcontainer.ch05 (172.20.0.2): icmp_seq=2 ttl=64 time=0.092 ms
```



Note: Before Docker had a DNS server, you connected containers by using the `--link` flag when you ran them, and that created entries in the `hosts` file on the container so that you could refer to linked containers by name. That functionality is preserved for backwards compatibility, but it's a basic mechanism and it's no longer needed. If you're following an old tutorial that uses `--link`, merely replace it with `--network` and your containers can communicate by name.

The DNS server in Docker makes application configuration easy in a multicontainer system because, in your applications, you simply refer to dependencies by their container name. If you have a database container called **mysql**, the connection string in your web app can always use **mysql** as the server name. You don't need to switch connection strings for different environments—provided you orchestrate the same way in every environment, the landscape will always look the same to the apps.

When you make assumptions about server names in your system, orchestration becomes a key part of your delivery. Using Docker gives you a consistent set of packaged applications, but you don't want to rely on a deployment document to ensure they get orchestrated correctly. For that, you can use Docker Compose.

Docker Compose

As you begin delivering more complex apps with more moving parts, the orchestration of your containers becomes more involved, and you will need to capture the dependencies so that you can deploy reliably and repeatedly. One option for doing so is [Docker Compose](#), a separate client tool that lets you define your solution architecture in a YAML file, and then manage a whole set of containers with a single command.



Tip: *Docker Compose comes bundled with Docker for Mac, Docker for Windows, and the Docker Toolbox. On Linux, it's a separate install, so you'll need to follow the instructions to [install Docker Compose](#).*

The Docker Compose syntax takes the arguments available in the `docker container run` command and structures them as [YAML](#). You specify an image to run from, container names, ports to expose, volumes to map, and networks—all the major command-line options are supported by Compose. Code Listing 54 shows a full Docker Compose file that will orchestrate the two containers we started manually.

Code Listing 54: Docker Compose YAML File

```
version: '2'

services:

  web:

    image: nginx:alpine

  util:

    image: sixeyed/ubuntu-with-utils

    container_name: util

    command: ping web
```

The first line defines this as using version 2 of the Docker Compose file format. In Compose, you define containers to run as services, then you specify the image name and any other optional properties, such as the container name and command in this example.

By convention, Compose files are named `docker-compose.yml`. From the directory where the YAML file is saved, you can start all the services in the Compose file as containers by running the `up` command, as in Code Listing 55.

Code Listing 55: Starting Services with Docker Compose

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

As with the Docker CLI, we can specify **-d** to daemonize all the containers and keep them running in the background. The container service called **util** has a **command** specified in the Compose file that is a long-running command, which means both containers will stay running until we stop them.

Although we started the containers using Docker Compose, these are just normal Docker containers and we can work with them with the Docker CLI in the normal way. Code Listing 56 shows the **container ls** command listing all the running containers.

Code Listing 56: Containers Running from Docker Compose

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
420b4928a92d	nginx:alpine	"nginx -g 'daemon off'"	2 minutes ago
Up 2 minutes	0.0.0.0:80->80/tcp, 443/tcp	sample01_web_1	
7262c0b6367d	sixeyed/ubuntu-with-utils	"ping web"	2 minutes ago
Up 2 minutes		util	

The Nginx container has a name generated by Compose that uses the project name **sample01**, which is derived by combining the folder where the **docker-compose.yml** file is on my machine with the service name **web** and the instance of the container: **sample01_web_1**. The utility container has the fixed name **util**, as specified in the Compose file.

The utility container pings the Nginx container by the service name **web**, which is specified in the Compose file—not by the container name, which isn't known until runtime. If we check the logs, we'll see the Nginx container is responding to pings on the service name, as in Code Listing 57.

Code Listing 57: Checking Logs from a Container

```
$ docker container logs util
```

PING web (172.21.0.3) 56(84) bytes of data.

64 bytes from sample01_web_1.sample01_default (172.21.0.3): icmp_seq=1 ttl=64 time=0.092 ms

64 bytes from sample01_web_1.sample01_default (172.21.0.3): icmp_seq=2 ttl=64 time=0.055 ms

64 bytes from sample01_web_1.sample01_default (172.21.0.3): icmp_seq=3 ttl=64 time=0.057 ms



Note: Docker has different logging providers that can be used to direct the standard output and standard error streams to different destinations. If you don't specify a provider, the default allows you to see the output from containers using the `Logs` command.

The Nginx container has a generated name because Compose lets you scale up by adding more containers and because it needs to ensure the name is unique. We can add more Nginx containers with the `scale` option to the `up` command in Code Listing 58, which specifies the desired number of containers.

Code Listing 58: Scaling Services with Docker Compose

```
$ docker-compose up -d --scale web=5

Creating and starting sample01_web_2 ... done
Creating and starting sample01_web_3 ... done
Creating and starting sample01_web_4 ... done
Creating and starting sample01_web_5 ... done
```

If we now connect to the `util` container, we can see that the DNS server in Docker returns all five Nginx containers when you query it, as in Code Listing 59.

Code Listing 59: Querying the Docker DNS Server

```
$ docker container exec -it util bash

root@b9594d7abca1:/# dig web

...

;; ANSWER SECTION:

web.                600    IN     A      172.21.0.5
web.                600    IN     A      172.21.0.4
web.                600    IN     A      172.21.0.6
web.                600    IN     A      172.21.0.3
web.                600    IN     A      172.21.0.7
```



Note: The DNS server in Docker actually supports load balancing, too, which means that if you repeat the `dig` command, the addresses in the response come back in a different order, which is a simple way of balancing load across the containers.

The IP addresses of the containers start with **172.21.0**—a different range from the network we explicitly created earlier in the chapter. Docker Compose creates a different network for each project, each named after the directory in which the Compose file lives.

When I list my networks with the **network ls** command in Code Listing 60, I can see some default networks created by Docker, the **ch05** network I created manually and the **sample01_default** network created by Docker Compose.

Code Listing 60: Listing Docker Networks

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
ffe636469bc2	ch05	bridge	local
ef5358de1913	docker_default	bridge	local
5c600195c84b	host	host	local
e78c673dd3f9	none	null	local
30771928757e	sample01_default	bridge	local

You can administer all the containers in your Compose project with single commands. These are typically the Docker CLI commands transplanted to Docker Compose. In Code Listing 61, we stop all the containers, check that none are running, then start them again.

Code Listing 61: Stopping and Starting Containers with Compose

```
$ docker-compose stop
```

Stopping sample01_web_5 ... done

Stopping sample01_web_4 ... done

Stopping sample01_web_2 ... done

Stopping sample01_web_3 ... done

Stopping sample01_web_1 ... done

Stopping util ... done


```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	

```
$ docker-compose start  
Starting util ... done  
Starting web ... done
```



Tip: Other commands, such as `kill` and `rm`, are available in Docker Compose, but it's important to remember that Compose is a client-side tool. When you create services with Compose, Docker has no notion that they're related (other than being in the same network). In order to manage the services as a single unit, you work in the directory on your client machine where the `docker-compose.yml` file lives.

With Docker Compose, you can orchestrate complex applications by linking containers together and running them inside a Docker network. Distributed solutions such as the example in Figure 9 can be captured at the infrastructure level in a Compose file.

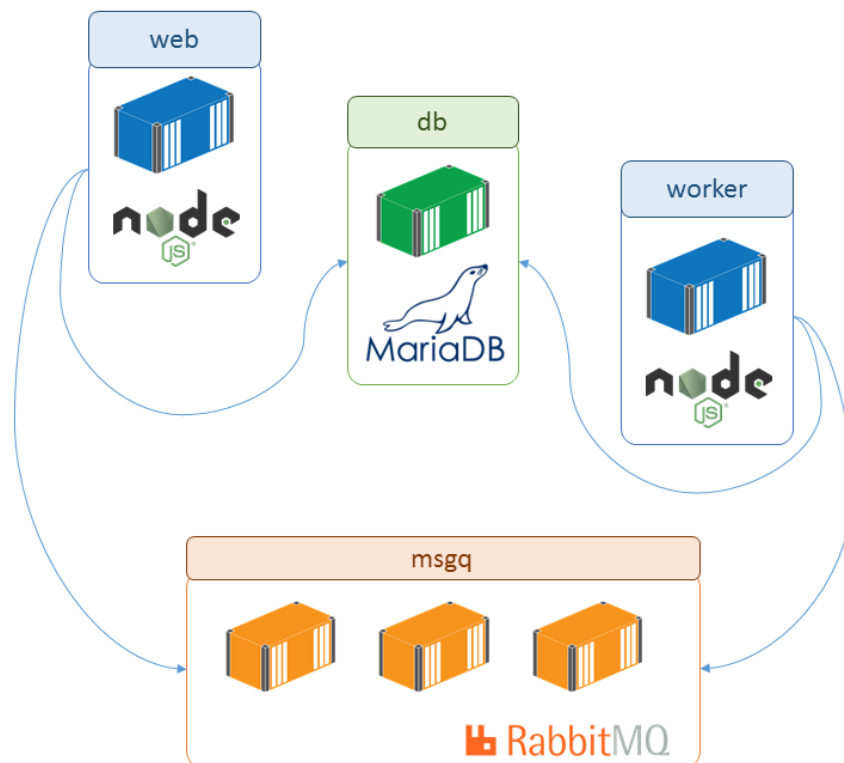


Figure 9: A Distributed System in Docker

Each component in the stack refers to its dependencies by name (`msgq` and `db`), and that will be consistent for every environment in which the containers are in the same network, even if they are running at different levels of scale (which means there's no change to application configuration).

More advanced features exist than we can cover in a *Succinctly* title, but the key concept is that the network looks the same from the inside, so containers only need to know each other's names. From the outside, the network might be a simple bridge network on a single host, such as a developer's laptop or an overlay network across multiple servers in a datacenter—the application code and the container orchestration are the same.

Summary

Application containers are lightweight, portable units. Breaking a large system down into many small containers means you can scale components independently at runtime so that you can direct compute power where it will bring the most benefit. It also means different parts of your app can have different release cycles, which allows you to release high-value components more frequently and give value to the business more quickly.

Docker supports that with simple but powerful mechanisms for orchestrating multicontainer solutions. With Docker networking, you can set up an isolated network between a group of containers with built-in DNS so that containers can communicate with each other transparently by their names. That lets you run multiple distributed systems on a single set of infrastructure.

For cases in which you have a complex deployment with many containers, you can use Docker Compose to explicitly orchestrate the dependencies. With Compose, you define all your containers in a single place and you can manage an entire system as a single entity. Now that you know how to manage multiple containers with Docker, in the next chapter we'll look at managing multiple Docker hosts in a cluster, using a Docker swarm.

Chapter 6 Clustering Hosts with Docker Swarm

Clustering with Docker swarm mode

You start to see the real power of Docker when your application is packaged and you're ready to deploy to production. In most cases, you'll want redundancy for your app so that if the server hosting it goes down, another server can take over; and you'll want the ability to scale so that multiple servers can all run the app and share the workload between them. Docker has native support for that with Docker swarm mode, which lets you join multiple machines running Docker into a cluster that provides scale and redundancy.

Swarm mode is an optional feature in Docker. You can run Docker as a standalone host, as we've done so far, or you can join that same host to a swarm with a single command. When you have multiple machines running in a swarm, you start containers in a similar way, but you can specify how many instances of the container you want to run. You make a request to the swarm, and, based on the resources available, Docker decides which hosts will actually run the containers.

It's incredibly easy to set up a Docker Swarm, and you can add and remove nodes while the swarm is running in order to manage your overall capacity. The swarm supports Docker networking, which means your containers can all communicate, and it monitors instances the entire time. If a host goes down and the number of containers running your app falls below what you requested, Docker will start new instances on other hosts in order to ensure the service level is correct. In this chapter, we'll see how swarm mode works and how to run containers on a swarm.

Creating a Docker swarm

Docker swarm follows a manager-worker architecture in which the manager schedules work on the workers. In a swarm, the type of work is always the same—running containers—and Docker is slightly unusual in that containers can run on the manager nodes, too. A manager node must be running for the swarm to operate correctly. You can have multiple nodes designated as managers in order to provide high availability, but only one is active—the leader.



Tip: Docker swarm mode uses the [Raft Consensus Algorithm](#) for electing a new leader manager node if the current one is lost. Like similar algorithms, Raft relies on a majority election, which means you should have an odd number of manager nodes in order to avoid tied elections—three or five is common. Because Docker swarm managers can run containers, too, your secondary managers can work and you won't have wasted compute in your swarm.

Creating a swarm is easy. You'll need a number of machines (or VMs) set up, and the only prerequisites for the machines are:

- Docker installed, version **1.12** or higher
- Ports **2377**, **7946**, and **4789** open so the machines can communicate

When you create your swarm, Docker will provision certificates and ensure that communication between all the nodes is encrypted. The specified ports need to be open between the nodes, but they don't need to be public, and you can restrict access at the infrastructure layer. For instance, in Microsoft Azure you would create all your VMs in a virtual network and only open the ports to other machines from the virtual network, which means access from the public Internet would be blocked.



Note: Docker 1.12 introduced swarm mode, which means that is the minimum version you can use. Prior to 1.12, Docker Swarm was a separate product. You can still use the old Docker Swarm to cluster multiple Docker hosts, but it is far more complicated to set up, and there are no advantages to using it over the newer swarm mode.

With the machines ready to go, choose one to be the master, note its IP address, and create a swarm using the **swarm init** command. I have three VMs in Azure running Ubuntu, which have Docker installed, and I'm going to make them into a swarm. In Code Listing 62, I create the swarm from the machine with the internal IP address 10.0.0.4, which switches this Docker host into swarm mode and sets it as the manager for the swarm.

Code Listing 62: Creating a Docker Swarm

```
$ docker swarm init --advertise-addr 10.0.0.4

Swarm initialized: current node (am2avhyv52mr05yernr2t9ug8) is now a manager.

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

    docker swarm join \

        --token SWMTKN-1-637apaa2992zallyfvw6iudobr6vai830h65tkk16mee8nw2u0-
15ofjpt6mt4r1yy1jgg9buyb6 \

    10.0.0.4:2377
```

In the **init** command, you need only to specify the IP address from which the master will listen for nodes if the manager has multiple IP addresses, but specifying the address is useful as a matter of course so that you'll know exactly where the manager is advertising for new joiners. That command is all you need. This is a functional swarm now, albeit with a single node, and now you can add the rest of your worker nodes.

The output from **swarm init** gives you a token that you use to join more nodes to the swarm. The token is a security measure that prevents any nodes joining the swarm unless you share the token with them—you should treat it carefully, like any other security secret. In Code Listing 63, I join another Azure VM to the swarm.

Code Listing 63: Joining the Swarm

```
$ docker swarm join \
> --token SWMTKN-1-637apaa2992zallyfvw6iudobr6vai830h65tkk16mee8nw2u0-
15ofjpt6mt4r1yy1jgg9buyb6 \
> 10.0.0.4:2377

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

The **swarm join** command specifies the secret token and the advertised address of the master, and the output shows the nodes have been accepted as workers, so now I have a multinode swarm. Now that we're in swarm mode, managing nodes and containers is a little different, and certain commands work only on the manager. Because that is true, and also because worker nodes can come and go, it's best to work with Docker on the manager.

Working with Docker swarm

Using Docker in swarm mode brings us two new concepts. First, all the machines in the swarm are called nodes, and you manage them with a set of **node** commands. Second, you don't run containers on the swarm, you work at a higher level with **services**. Running a service is like running a container—it gets created from a Docker image in the same way—but a service can run as multiple instances of a container.

From the swarm manager, you can see all the nodes in the swarm and the current status with the **node ls** command, as in Code Listing 64 (in which I've added a third node to the swarm).

Code Listing 64: Listing Nodes in a Swarm

```
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
8smbcf360or1257jt5d3fj32i	swarm-02	Ready	Active	
aedcdczhcppw7qva7fdtvxmmj	swarm-01	Ready	Active	
am2avhyv52mr05yernr2t9ug8 *	swarm-00	Ready	Active	Leader

The output here tells us that machine **swarm-00** is the leader manager node and that **swarm-01** and **swarm-02** are functioning worker nodes—with Active availability and Ready status. We'll see what happens when a node goes offline later in this chapter.

To start a service on the swarm, use the **service create** command. Code Listing 65 shows the usual Nginx web application being started as a service.

Code Listing 65: Running a Service on the Swarm

```
$ docker service create --name website --publish 80:80 nginx:alpine
cixz0vah486d81zo1ecw5fe2p
```

Creating a service is a lot like running a container. You specify the image and any options for the engine—for example, the service name and port publishing in this example. Nginx exposes port 80, and my Azure VMs are configured to allow port 80 requests from the Internet, which means I should be able to browse my website from any computer so long as I know the address of the node running the container.

The **service ls** command tells me which services are running, **service ps** tells me which hosts are running which containers, and Code Listing 66 shows me which VM I need to hit in order to access the website.

Code Listing 66: Listing Services and Containers

```
$ docker service ls
```

ID	NAME	REPLICAS	IMAGE	COMMAND
cixz0vah486d	website	1/1	nginx:alpine	

```
$ docker service ps website
```

ID	NAME	IMAGE	NODE	DESIRED STATE
CURRENT STATE	ERROR			
1w5lqx0lp4pnftqzl04m0kqiq	website.1	nginx:alpine	swarm-00	Running
minutes ago				2

The one container running on **swarm-00** happens to be the manager. Now, if I browse to the manager's domain address, the port 80 request will be forwarded to the container running on the node and I can see the website. Figure 10 shows the result. As we should expect, notice that the hostname **swarm-00** of the machine serving the response is the same as the VM to which I sent the request.



Figure 10: Browsing the Website on the Swarm Container

We can map our public web address to a specific host with a CNAME DNS entry, but we don't want to do that. If that host goes down, the swarm will recreate the container on another host, and we'd need to change the CNAME, which isn't very flexible or scalable. However, one of the great things about Docker swarm mode is its built-in request routing, which lets you send a request to any node in the swarm, and if that node isn't running a container that can service the request, it will transparently route the request to another node where the container is running.

In Code Listing 67, I'm making a web request with cURL to node **swarm-01** that doesn't have the web container running. I still get the correct response, though—node **swarm-01** has routed the request to node **swarm-00**, where the web container is running, and it has relayed the response.

Code Listing 67: Querying Another Swarm Node

```
$ curl http://swarm-01.northeurope.cloudapp.azure.com/

<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>
```

This feature is called the routing mesh, and it enables load balancing and failover. You can put a load balancer in front of your swarm so that requests can be forwarded to any node, and if a node gets a request it can't handle, it will intelligently reroute the request to a node that can. Similarly, if a node fails, its containers will be started on other nodes, and the services will still be reachable through the routing mesh.

Scaling services

The services we've created so far have run with a single container instance on the swarm—that's the default setting for a new service. We can run multiple instances of a container using the **replicas** option. Replica is the swarm terminology for an instance of a container, and as the name suggests, these are replicas running from the same image with the same setup. The services should be stateless so that requests can be handled by any container with the same results.

Running multiple replicas with a load balancer means you can scale up your service by running it on more nodes. The replica level can be specified when services are created, and it can be changed when services are running. The **service scale** command in Code Listing 68 adds another four instances of my website container to the Swarm.

Code Listing 68: Scaling a Swarm Service

```
$ docker service scale website=5  
  
website scaled to 5
```

Now, when I check the service with the **service ps** command, as in Code Listing 69, I can see where the instances are running. In this case, I have more replicas than there are nodes in the swarm, which means nodes **swarm-00** and **swarm-02** have two replicas running and the node **swarm-01** just has one.

Code Listing 69: Listing Service Replicas

```
$ docker service ps website
```

ID CURRENT STATE	NAME ERROR	IMAGE	NODE	DESIRED	STATE
1w5lqx0lp4pnftqzl04m0kqiq 13 minutes ago	website.1	nginx:alpine	swarm-00	Running	Running
80qaxmk6fi836qyxc9n4i3o2n Preparing 12 seconds ago	website.2	nginx:alpine	swarm-01	Running	
eqvkkgax24vxpo6hq1wt17199 4 seconds ago	website.3	nginx:alpine	swarm-02	Running	Running
28z50bvlx7kakk403qib86dl4 4 seconds ago	website.4	nginx:alpine	swarm-02	Running	Running
4fp3992k7iq72nn0fobkdzwhs 9 seconds ago	website.5	nginx:alpine	swarm-00	Running	Running

If I now make a request to a specific node, I'll get a response from that node—all nodes have an instance of the container running, which means they won't need to reroute requests internally. I have a load balancer set up in Azure that shares requests among all the VMs, which is outside of Docker at the infrastructure level. Using [Apache Bench](#) to test performance, with five replicas running across three nodes, I get served more than 180 requests per second, as Code Listing 70 shows.

Code Listing 70: Benchmarking the Swarm Service

```
$ ab -n 3000 -c 150 http://docker-succinctly.northeurope.cloudapp.azure.com/
...

Benchmarking docker-succinctly.northeurope.cloudapp.azure.com (be patient)
Completed 300 requests
Completed 600 requests
...

Concurrency Level:      150
Time taken for tests:    16.442 seconds
Complete requests:      3000
Failed requests:         0
Total transferred:      2535000 bytes
HTML transferred:       1836000 bytes
Requests per second:   182.46 [#/sec] (mean)
Time per request:       822.088 [ms] (mean)
Time per request:       5.481 [ms] (mean, across all concurrent requests)
```

If the service scale is reduced to a single replica, performance drops to fewer than 120 requests per second, as Code Listing 71 shows.

Code Listing 71: Benchmarking a Single Replica

```
$ ab -n 3000 -c 150 http://docker-succinctly.northeurope.cloudapp.azure.com/
...
```

```
Benchmarking docker-succinctly.northeurope.cloudapp.azure.com (be patient)
```

```
Completed 300 requests
```

```
Completed 600 requests
```

```
...
```

```
Concurrency Level:      150
```

```
Time taken for tests:    25.840 seconds
```

```
Complete requests:      3000
```

```
Failed requests:        0
```

```
Total transferred:      2535000 bytes
```

```
HTML transferred:       1836000 bytes
```

```
Requests per second:    116.10 [#/sec] (mean)
```

```
Time per request:        1292.008 [ms] (mean)
```

```
Time per request:        8.613 [ms] (mean, across all concurrent requests)
```

Nginx is a highly performant web server in any circumstance, but running it as a service on a Docker swarm allows you to scale up in seconds with a single command and thereby drastically improve performance. For more complex systems that perform more compute-to-serve requests, the benefits from scaling up are even greater.



Tip: *The routing mesh works at port level—if a node gets a request on a port from which there's no container listening, it will forward the request on to another host that does have a container listening on that port. If you run only one service per port on your swarm, you don't need a proxy—your load balancer and the swarm can handle it. If you're running multiple services with the same port, such as several websites all running on port 80, you'll need a proxy, like Nginx, running in the swarm. The proxy listens on port 80 while all your other services listen on custom ports, and Nginx routes traffic to different ports based on the incoming HTTP request.*

The manager node in the swarm constantly checks in with nodes and verifies that all services have the correct replica level. When a node goes down, the swarm will create more containers on other nodes in order to ensure the services run as requested. In Code Listing 72, I shut down node **swarm-02** in the swarm, which is currently running two of the Nginx containers.

Code Listing 72: Shutting Down a Worker Node

```
elton@swarm-02:~$ sudo shutdown 0
```

When the node shuts down, the heartbeats to the manager stop and the manager quickly sees that the node is unavailable. With that node down, my website service falls below the required replica level, which means the manager will schedule new containers on the remaining nodes. Code Listing 73 shows the **node ls** command that tells me node **swarm-02** is down, but the service details show that there are still five containers running.

Code Listing 73: Listing Nodes and Running Replicas

```
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
8smbcf360or1257jt5d3fj32i	swarm-02	Down	Active	
aedcdczhcppw7qva7fdtvxmmj	swarm-01	Ready	Active	
am2avhyv52mr05yernr2t9ug8 *	swarm-00	Ready	Active	Leader


```
$ docker service ps -f "desired-state=running" website
```

ID	NAME	IMAGE	NODE	DESIRED	STATE
CURRENT	STATE	ERROR			
1w5lqx0lp4pnftqzl04m0kqiq 36 minutes ago	website.1	nginx:alpine	swarm-00	Running	Running
bpeyfyjd5hl129rdiborxryj1t 12 minutes ago	website.2	nginx:alpine	swarm-01	Running	Running
5c2q4odd5vw62eyb6g4clrrz1 13 seconds ago	website.3	nginx:alpine	swarm-00	Running	Running
ehlxvdq66adtjd9p53zs9i7y5 13 seconds ago	website.4	nginx:alpine	swarm-00	Running	Running
7pok5wg9ntfo4iy1zq9a983n6 12 minutes ago	website.5	nginx:alpine	swarm-01	Running	Running

The containers that had been on node **swarm-02** have been rescheduled on the other nodes so that node **swarm-00** now has three containers and node **swarm-01** has two.

The swarm takes care of routing traffic to nodes and repairing services when nodes are unavailable, which means much of the hard work of managing distributed systems has been dealt with.

Summary

Docker swarm mode is a clustering technology built into the Docker platform. You can create a swarm from multiple Docker hosts in minutes and have a cluster that can intelligently route traffic between nodes and run scalable services as multiple instances of containers.

Communication between nodes is all encrypted, and nodes don't have to be on the same operating system or even the same platform, which means you can build hybrid swarms with Linux and Windows Server hosts.

Running a hybrid swarm with Windows and Linux nodes lets you migrate multiple existing workloads into a single swarm, containerizing apps to use different base images—such as Windows Server Core for .NET Framework apps and Ubuntu Server for Linux apps. Your applications are all Docker images, and you can manage them in the same way, but at runtime you can only run Windows-based containers on Windows hosts and Linux-based containers on Linux hosts.

In the next chapter, we'll see how containers actually work and what Docker looks like on different host platforms.

Chapter 7 Docker on Linux, Windows, and ARM

Containers and kernels

When you run a Docker container on a Linux host, it runs in an isolated process boundary called a namespace. Inside the container, the app thinks it's running on its own computer with no other processes, but in fact the boundary is only a logical boundary and there can be many other processes running on the same physical host.

You can see all the processes running inside and outside of your containers on the host machine, but in the container you'll only see the processes within the namespace boundary. Figure 11 shows how that looks on a Linux machine running multiple Nginx containers.

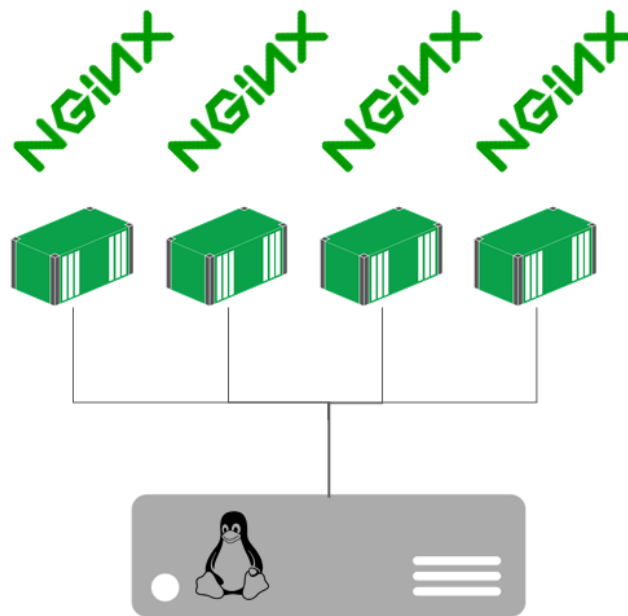


Figure 11: Multiple Containers Running Nginx

If you were to run the **top** command inside one of the containers, you would see only a single **nginx** process. But if you run **top** on the host, you'll see four **nginx** processes and the **dockerd** process, which is the server part of the Docker Engine, along with any other processes you have running.

This is why Docker containers can run so efficiently—they use the underlying operating system kernel of the host machine so that processes inside the container are actually running on the host. For a host, running multiple containers is the same as running multiple processes (unlike virtual machines, for which each VM has its own kernel and a hypervisor running on the host in order to translate between the virtual kernel and the real kernel).

That is why you can't run Linux containers on Windows or run Windows containers on Linux. A container using Ubuntu as its base image needs to run on a Linux machine so that when the container launches an executable, the host is capable of running it. Linux executables aren't compatible with Windows, which means you can't run an Ubuntu-based container on Windows. In this chapter, we'll look at how Docker runs on different operating systems and see how you can mix architectures.

Docker on Linux

Linux is the native operating system for Docker. Linux has long had the technology for running containers, but Docker brought the concept of packaging up images and running them as containers, which made it simple to take advantage of the underlying Linux features. When you're running Linux-based containers, you use a server-grade Linux distribution for your hosts, and Docker might be the only software you install. Everything else you'd want to run on your server would run as a container in Docker.

Docker is available in the package repositories for the major Linux distributions, and the only requirement is that you have a reasonably recent version of the Linux kernel running on your host. At the time of writing, the minimum Linux kernel is 3.10, which is an entire major version behind the current release, which means Docker doesn't demand an up-to-date operating system version.

The quickest way to install Docker on Linux is to run the helper script from **get.docker.com** in Code Listing 74.

Code Listing 74: Installing Docker on Linux

```
$ curl -sSL http://get.docker.com | sh
```

Doing this will install Docker and the CLI on your machine, and you can run containers and create or join a swarm using the commands we've seen so far. The Docker package doesn't install Docker Compose, but that's a client-side tool that you'd normally have running on a separate machine.



Tip: *It's important that you install the correct version of Docker. The version included with your Linux distribution is not likely to be the latest. If you don't care to run a shell script from the Internet, the [Docker Store](#) website has detailed instructions for all the supported Linux distros.*

You can administer a Docker host remotely, and all the clients are compatible with the Engine so that you can run the Docker command line from a Windows or Mac machine to work with containers on a remote Linux host.

If you use **systemd** to run Docker as a service on Linux, which is the default init system with Ubuntu, you get a lot of information from the status output. Code Listing 75 shows the output from a Docker Engine that isn't currently running any containers.

Code Listing 75: Checking the Status of the Docker Daemon

```
$ sudo service docker status

docker.service - Docker Application Container Engine

   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset:
enabled)

   Active: active (running) since Sun 2016-08-21 09:10:21 BST; 6h ago

     Docs: https://docs.docker.com

Main PID: 3091 (dockerd)

   Tasks: 41

  Memory: 1.6G

     CPU: 2min 35.083s

   CGroup: /system.slice/docker.service

           └─3091 /usr/bin/dockerd -H fd://

               └─3175 docker-containerd -l unix:///var/run/docker/libcontainerd/docker-
containerd.sock --shim docker-containerd-
```

Each running container will be listed in the **CGroup** section—the control groups Docker is using. You don't need to know the details of kernel namespaces and control groups in order to use Docker successfully, but it is useful to be aware of how the runtime structures affect security.

A note on security

Docker takes a lot of the pain away from building, sharing, and running software systems, but it doesn't achieve this by taking security shortcuts. The Docker daemon runs with root privilege on Linux, and, in the early days of Docker, users expressed concern that this was a dangerous attack vector—could a vulnerable application be compromised and gain access to other containers on the host?

Docker worked with the Center for Internet Security to address those concerns and to build a comprehensive set of recommendations in a benchmark document. These are publicly available for different versions of Docker, such as the [CIS Docker 1.12.0 Benchmark](#).

Successive releases of Docker have increased security. You can now run containers in a user namespace without root privileges, and you can build your images to have the processes inside containers running as nonroot users. The Docker documentation has a useful section on [Docker Security](#) that covers the technical elements of securing the container environment.



Tip: There are features of Docker that actually increase the overall security of your systems. The [Docker Bench](#) tool runs as a container and scans your Linux host for vulnerabilities. [Docker Security Scanning](#) is an optional feature on the Docker Hub that scans images in your repositories and reports on known vulnerabilities for the software inside your images.

Docker on Windows

Windows Server 2016 and Windows 10 have native support for containers and Docker. There are different ways to run containers on Windows: either as Windows Containers, which share the operating system kernel and run in a similar way to Linux containers, or as Hyper-V containers, which can use the same images as Windows Containers but run in a very thin hypervisor layer so that each container has its own kernel.

You can't run Linux-based containers on Windows, but Windows Containers are powered by Docker, which means you can package applications into Docker image binaries and run them in Windows using the same Docker command-line tool. In order for containers to run on Windows, they need to be based from a Windows image, and Microsoft has base images available on the Docker Hub.

The Hub has two base images for Windows containers: [microsoft/windowsservercore](#) and [microsoft/nanoserver](#). Windows Server Core is a full version of Windows Server that has all the standard Windows Server features except the UI. With Server Core, you can install software from MSIs and run the full .NET Framework, which means you can use it as a base image to containerize an existing Windows application.

Nano Server is a minimal installation of Windows that follows the approach of Linux base images on the Hub. Nano Server has a limited feature set—you can't run MSIs or the .NET Framework, and you can only use PowerShell to administer it. That means the image weighs in at around 350 MB instead of 3 GB for Server Core, and it has a much smaller attack surface.

Windows Containers are a new addition to the Docker ecosystem, but Windows uses the same Docker platform, so it's an established technology. You can build a simple image from Windows Nano Server using the Dockerfile in Code Listing 76.

Code Listing 76: A Dockerfile for a Windows Container Image

```
FROM microsoft/nanoserver

CMD ["powershell", "Write-Host 'Hello from Windows Nano Server!'"]
```



Note: Remember that when you build an image from a Dockerfile, Docker creates temporary containers to run the instructions. Since the base image here is a Windows container, you can only build this image from a Windows machine. If you're using Docker for Windows, be sure to switch to the Windows Containers option.

When the image is built, you can run it in the normal way. I've built and pushed an image from this Dockerfile onto the Hub so that you can run it with the command in Code Listing 77.

Code Listing 77: Running a Windows Container

```
> docker container run sixeyed/nanoserver-helloworld
```

```
Hello from Windows Nano Server!
```



Note: Again, this is a Windows-based container. You can't run it if your host is a Mac or a Linux machine, or even if you're using Windows with Docker running in a Linux VM. You can use Windows containers directly with [Docker for Windows](#) and [Docker on Windows Server 2016](#).

These are standard Docker containers that you can manage in the normal way so that **docker container ls** will list the running containers and **docker image ls** will show you all the images in your local cache. The Nano Server base image is a good starting point for new apps when you don't need to install heavy software and can benefit from the reduced footprint.

The Windows Server Core image doesn't easily fit with the philosophy of having lightweight portable containers because you can easily end up with a 5 GB image once you've packaged your app. But it's a great opportunity to containerize existing Windows apps without having to rewrite them so that you can move existing workloads off their own servers and into a Docker swarm.

Docker on ARM

You can also run Docker on low-cost, low-power computing devices based on the Advanced RISC Machine (ARM) CPU architecture, such as the [Raspberry Pi](#) or [Pine64](#). ARM devices tend to be well supported with Linux operating systems, but unfortunately you can't run normal Linux-based Docker images on ARM devices. The Linux kernel for a container may be the same as the kernel on the ARM device, but the binary image was probably compiled for the x86/x64 processor instruction set, which is not compatible with ARM.

In order to run Docker containers on ARM, the container must have been compiled for the ARM chipset, which in practice means building the image from a Dockerfile on your ARM device. Once the image is built, you can ship it in the usual way, including pushing it to the Docker Hub (and there are many ARM-compatible images on the Hub). Currently, ARM images are not easy to identify because you can't search the Hub by chipset or operating system.

The Raspberry Pi is a good ARM device to get started with because the setup script on [get.docker.com](#) has support for Raspbian, the native Linux distribution for the Pi. So, in order to install the Docker command line and Docker engine, simply use the same command for other Linux distributions, as in Code Listing 78.

Code Listing 78: Installing Docker on the Raspberry Pi

```
$ curl -sSL get.docker.com | sh
```

This is good for Raspberry Pi 2, 3, and Zero models—so you can get Docker running on a \$5 computer! There are ARM base images on the Docker Hub, which means you can containerize any application that runs on ARM. And because popular platforms such as Java, Go, and NodeJS have ARM support, you can build Docker images for those apps. The new .NET Core platform from Microsoft has ARM support on the roadmap for the next release.

The Dockerfile in Code Listing 79 can be used to build the obligatory Hello World app in an ARM-compatible Docker image based on the **resin/rpi-raspbian** image on the Hub.

Code Listing 79: Dockerfile for an ARM Image

```
FROM resin/rpi-raspbian:jessie  
  
CMD ["echo", "Hello World from the Pi!"]
```

You can build and run that image with the Docker CLI as usual, but as with Windows containers, you must run the commands on the correct architecture. In this case, you need to build the image and run containers on the Pi itself, as in Code Listing 80.

Code Listing 80: Building Images and Running Containers on ARM

```
pi@raspberrypi:~ $ docker build -t sixeyed/armhf-helloworld .  
  
Sending build context to Docker daemon 9.728 kB  
  
Step 1 : FROM resin/rpi-raspbian:jessie  
  
---> e353bbe6de96  
  
Step 2 : CMD echo Hello World from the Pi!  
  
---> Running in 41038ed95981  
  
---> cc950743d17a  
  
Removing intermediate container 41038ed95981  
Successfully built cc950743d17a  
  
pi@raspberrypi:~ $ docker run sixeyed/armhf-helloworld  
  
Hello World from the Pi!
```

There are a number of ARM-based images on the Docker Hub. The [armhf](#) account has repositories for Node and Redis running on ARM, but they are flagged as experimental. The number of ARM-compatible images on the Hub is small at the moment, and building your own images can be time-consuming on an ARM device, but once you have an image you can run containers from it in the normal way.

Docker doesn't have full feature parity on ARM, but you can join multiple devices together in a swarm. Interestingly, the swarm supports heterogeneous hosts, which means you can build a hybrid swarm of ARM, x64, Linux, and Windows hosts.

Hybrid Docker swarms

You can't mix containers for different OS kernels and CPU architectures on a single Docker host, but you can build a swarm with hosts from multiple architectures and operating systems, and you can run different types of containers on the swarm. Figure 12 shows a sample architecture for a hybrid swarm.

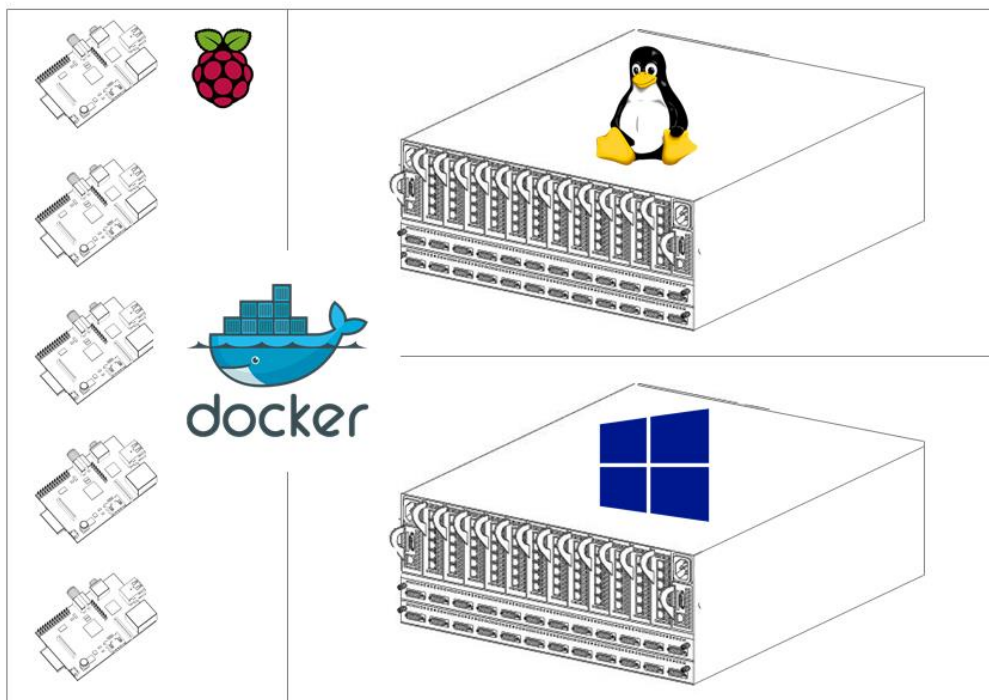


Figure 12: A (Very) Hybrid Docker Swarm

This scenario is not quite as extreme as it looks. Provided you have a fast enough network connection between the hosts, this swarm is highly functional and gives you a huge amount of flexibility. You can run an existing ASP.NET app in Windows Server Core containers on the Windows nodes, front them with a caching proxy in Nginx running on the Linux nodes, and use the ARM nodes for noncritical workloads, such as collecting metrics from the other containers and running a performance dashboard.

You create a hybrid swarm in the same way as a normal swarm, starting with a manager and joining new nodes. But in order to make sure containers run only on the correct nodes, you need to give Docker some extra information. You can add arbitrary labels to Docker hosts—for example, you can add flags like **cpu=armhf** or **os=linux** and **os=windows** to the hosts.

In Code Listing 81, I have created a hybrid swarm with an x64/Linux node as the manager and a Raspberry Pi as the worker. I use the **update** command with the **--label-add** option to label the Pi with a custom **cpu=armhf** label.

Code Listing 81: Labeling Swarm Nodes

```
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
0bvd2x0lh4e5kbpzpydvev02r *	sc-ub-brix	Ready	Active	Leader
3he5xk2ve5fewk4j0pmwhqril	raspberrypi	Ready	Active	

```
$ docker node update --label-add cpu=armhf raspberrypi
raspberrypi
```

Services can be scheduled with constraints to tell Docker that they have special host requirements. Node labels can be used in constraints so that we can run three replicas of the x64-based Nginx container with the command in Code Listing 82 (which excludes nodes with the **cpu=armhf** label).

Code Listing 82: Creating a Service with a Node Label Constraint

```
$ docker service create --name website --constraint 'node.labels.cpu != armhf' --replicas 3 nginx:alpine
dqtfwk7zmx44r1ztrh2k116m
```

When the scheduler selects nodes, it will exclude any that have a label with the key **cpu** and the value **armhf**. In this case, that only leaves the manager node, and as we see in Code Listing 83, all three containers are running on the manager.

Code Listing 83: Listing Service Instances

```
$ docker service ps website
```

ID	NAME	IMAGE	NODE	DESIRED STATE
CURRENT STATE	ERROR			
7z9a5k5vaepdqr9eyi7izg16	website.1	nginx:alpine	sc-ub-brix	Running
Running 6 seconds ago				

```
el30vk6euevq7swx1e9fw4axe  website.2  nginx:alpine  sc-ub-brix  Running
Running 5 seconds ago

5g2p0f9d8i28jqfh01of1hft0  website.3  nginx:alpine  sc-ub-brix  Running
Running 6 seconds ago
```

Docker supports two kind of labels: node labels that were introduced with swarm mode for scheduling constraints, and engine labels for backwards compatibility on older hosts. In order to set the Docker Engine labels, you need to amend the environment configuration on the host, which will differ for each operating system. Node labels can be administered on the manager, which means they are a better approach.



Note: *In order to span systems across hybrid swarms, you need to create an overlay network and start the services with the `--network` option. Different platforms have different networking stacks, but as of Docker 17.06 you can network containers across hybrid swarms.*

Summary

We've looked at how Docker containers work on different operating systems and CPU architectures. The main takeaway is that you can't mix and match—images based on Windows can run only on Windows hosts, images based on Linux can run only on Linux hosts, and images compiled for x64 can run only on x64 hosts.

With Docker swarm mode, you can have nodes running on different architectures and operating systems, which lets you run a single cluster capable of hosting Linux, Windows, and even ARM applications with a single management interface. That's a particularly important option when you're paying for compute. If you want to run Dockerized Windows apps in the cloud, you can have Windows VMs as worker nodes while using cheaper Linux VMs for the manager nodes.

Running cloud solutions on Docker is particularly attractive because you get a consistent packaging and deployment model that you can use on any cloud, which means you can avoid being locked into a particular cloud platform. Next, we'll look at Docker in the cloud.

Chapter 8 Running Docker in the Cloud

Docker versus the cloud

A production setup for Docker makes very few demands from your infrastructure. You need only a few machines that can communicate with each other over a network—they can be physical machines connected with Ethernet or VMs connected in a virtual network. When you're hosting in the cloud, Docker makes a very good case for using Infrastructure-as-a-Service (IaaS) over Platform-as-a-Service (PaaS).

Typically, PaaS is the preferred option because it removes the administrative burden of managing virtual machines, and cloud providers add value with their PaaS offerings, which makes it easy to deploy and manage applications. However, all the PaaS offerings are custom. If you build an app to run on Amazon's Elastic Beanstalk, you can't lift the app and deploy it on Microsoft Azure instead—the packaging, deployment, and management models are not compatible. That means on-premise environments can't run the app on the same platform as production, which means that in development and testing you must run the app in a different way.

By packaging your app in Docker images, you can run the same images locally and in any cloud using basic IaaS services and without taking a dependency on a particular vendor's platform. In this chapter, we'll look at using Docker Machine to build a Docker swarm on the DigitalOcean cloud, we'll cover the Docker for Azure product, and we'll look at running a multicloud swarm.

Docker on DigitalOcean

DigitalOcean is a relatively new cloud provider. It initially made its name by providing fast SSD-backed Linux VMs at a much lower cost than AWS, Azure, and Google. Its services are mostly in the IaaS space, which means you can spin up VMs, connect them in a virtual network, and set up persistent storage, but there are no higher-level PaaS services such as a message queue or a managed database platform.

When you sign up for a DigitalOcean account, you can create an API access token, which you use with the `do` command-line tool to create and manage components in the cloud. You can manually create a VM, log in, and install Docker, but it's easier to use the Docker Machine tool.

[Docker Machine](#) is an open-source tool that abstracts the creation of a VM running Docker. Docker Machine has drivers for VirtualBox, AWS, Azure, DigitalOcean, and more, so you can create VMs in the same way on different platforms on your machines or in the cloud.



Note: *Docker Machine is bundled with the Docker for Mac, Docker for Windows, and Docker Toolbox suites, but not on Linux. If you're running Linux, the Docker documentation covers how to [Install Docker Machine](#).*

The provider plugins for Docker Machine use the relevant authentication mechanism. In order to create a new VM (called a “droplet” in DigitalOcean), you can use the command in Code Listing 84 and supply your own access token.

Code Listing 84: Creating a Docker Host on DigitalOcean with Docker Machine

```
$ docker-machine create \
  --driver digitalocean \
  --digitalocean-access-token=[YOUR-API-TOKEN] \
  --digitalocean-region lon1 \
  --digitalocean-image ubuntu-16-04-x64 \
  do-node-01

Running pre-create checks...
Creating machine...
(do-node-01) Creating SSH key...
(do-node-01) Creating Digital Ocean droplet...
(do-node-01) Waiting for IP address to be assigned to the Droplet...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with ubuntu(systemd)...
Installing Docker...
```

When that completes, you’ll have a new VM in DigitalOcean that runs the latest version of Docker. There are lots of options that the **create** command accepts, and they’re flexible for different VM providers. For DigitalOcean, I’ve specified the London datacenter and the latest longterm support version of Ubuntu, 16.04, as the operating system image for the VM.

Docker Machine locally stores details for the nodes it creates, and you can view them with the **docker-machine env** command. The output of that command is structured so that you can run it and set environment variables for your local Docker CLI to communicate with the remote Docker Engine you’ve just created. In Code Listing 85, I do this to work with the DigitalOcean droplet.

Code Listing 85: Configuring Docker CLI to Manage Remote Hosts

```
$ docker-machine env do-node-01

export DOCKER_TLS_VERIFY="1"

export DOCKER_HOST="tcp://139.59.189.184:2376"

export DOCKER_CERT_PATH="/home/elton/.docker/machine/machines/do-node-01"

export DOCKER_MACHINE_NAME="do-node-01"

# Run this command to configure your shell:

# eval $(docker-machine env do-node-01)


$ eval $(docker-machine env do-node-01)
```

Docker CLI uses the **DOCKER_HOST** and other environment variables to connect to the Docker Engine server. With those values set, my local command line is working with the DigitalOcean VM, and I can use all the usual commands. In Code Listing 86, I check the Docker versions.

Code Listing 86: Checking Docker Engine Versions

```
$ docker version

Client:

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

Server:

Version:      1.12.1
API version:  1.24
Go version:   go1.6.3
```

```
Git commit: 23cf638
Built:      Thu Aug 18 05:33:38 2016
OS/Arch:    linux/amd64
```

I have the latest version of Docker on my local client and the remote machine, and I can use the normal Docker commands to administer the remote machine. Versions don't need to match—you can use a newer client with an older engine, but not vice versa.



Tip: When you work with multiple Docker hosts, it's easy to lose track and find yourself killing containers on remote hosts when you meant to work locally. The `docker-machine env -u` command shows you how to reset the environment variables and point the CLI back to the local host.

You can repeat the command to create more VMs running Docker, then use the Docker Machine `ls` command to view them all. In Code Listing 87, you can see I have created two more DigitalOcean droplets through Docker Machine.

Code Listing 87: Listing Docker Machine Hosts

```
$ docker-machine ls -q
do-node-01
do-node-02
do-node-03
```

I won't use these hosts yet—we'll see how to join them to a multicloud swarm later in the chapter.

Docker Machine is a good choice for creating virtual machines as hosts—the commands are fundamentally the same for different platforms with only the options changing to support different feature sets. For most providers, the options can be specified in environment variables instead of the command, which means your actual `docker-machine` commands can be generic. You can capture your entire host setup in one script and have another script to tear it down, letting you easily spin up and remove environments.

Because it's focused at the host level, there are limitations with Docker Machine, which means you'll need to fall back to the platform's CLI if you want to create more complex IaaS environments with subnets in your virtual network or custom port configurations. Docker addresses the two main cloud providers with products that build out the entire infrastructure.

Docker for Azure

The “Docker for” products all follow the same basic principle—they create a consistent Docker environment using the correct tools for the platform. [Docker for Mac](#) runs a Linux VM using HyperKit and [Docker for Windows](#) runs a Linux VM using Hyper-V. The end-user experience for each is the same because Docker makes use of the native platform tools.

[Docker for Azure](#) creates a Docker swarm on Azure, and the deployment is based on an Azure Resource Management template. Like the other products, Docker for Azure makes best use of the platform. When you deploy the template, it creates a functional environment for the swarm to work in. This includes:

- A top-level resource group.
- A virtual network for all the nodes.
- A single VM to be the manager node.
- A Virtual Machine Scale Set for the worker nodes.
- A Network Security Group to configure routing rules.
- A load balancer sitting in front of the worker nodes.
- Public IP addresses for the manager VM and the load balancer.



Note: *With AWS, the approach is very similar. [Docker for AWS](#) provides a CloudFormation template that creates all the resources. Docker for GCP is the latest edition.*

In the Azure Portal, you can vary parameters for the deployment—for example, the size of the VMs and the initial number of worker nodes in the VM scale set, as we see in Figure 13.

Custom deployme...

Deploy from a custom template

Parameters

Customize your template parameters

* Template

Edit template

* Parameters

Edit parameters

* Subscription

Azure Free Trial

* Resource group

Create new

Use existing

DOCKER4AZURE

* Resource group location

North Europe

* Legal terms

Review legal terms

SWARMNAME (string)

dockerswarm

MANAGERCOUNT (int)

1

WORKERCOUNT (int)

3

WORKERVM SIZE (string)

Standard_D1_v2

MANAGERVM SIZE (string)

Standard_D1_v2

* AD SERVICE PRINCIPAL APP ID (string)

<YOUR-APP-ID>

* AD SERVICE PRINCIPAL APP SECRET (securestring)

.....

* SSH PUBLIC KEY (string)

<YOUR-SSH-KEY>

Figure 13: Creating a Swarm with Docker for Azure

The deployment is secured in two ways: with an Azure Active Directory Service Principal, which Docker for Azure uses to work with the Azure APIs from your subscription, and a normal SSH key, which you use for connecting with the swarm from your client machine. The [Docker for Azure](#) documentation details how to set up the security.

When deployment completes, you can connect to the manager node by SSH. In the Azure Portal, the Output section specifies the SSH command, as in Figure 14.

The screenshot displays the Azure portal interface for the 'docker-for-azure' deployment history. On the left, a sidebar contains navigation links: Overview, Activity log, Access control (IAM), Tags, SETTINGS, Quickstart, Resource costs, Deployments (selected), and Properties. The main area shows a deployment history table with one entry: 'Microsoft.Template' deployed on 8/21/2016 at 8:34:54 PM. To the right, the deployment details are shown, including a summary of the deployment date, status (Succeeded), resource group, related events, and template link. Below the summary, the 'Outputs' section lists two items: 'SSH' with the command 'ssh docker@52.169.123.238' and 'DEFAULTDNSTARGET' with the IP '52.169.127.220'.

Figure 14: Docker for Azure Output

Code Listing 88 shows the connection to the manager node, where we can run all the standard Docker commands, such as listing out the nodes in the swarm.

Code Listing 88: Connecting to the Docker for Azure Manager Node

```
$ ssh docker@52.169.123.238
Welcome to Docker!
dockerswarm-manager0:~$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY
MANAGER	STATUS		
17i6hbl89u3m4ridw9p6ykip6	_dockerswarm-worker-vmss_0	Ready	Active
33xg9i02xukght223215iprtc *	_dockerswarm-manager0	Ready	Active
Leader			
8q53wxmdo4umveppzazx0ntd7	_dockerswarm-worker-vmss_2	Ready	Active
czocuohj8r06q9ge29x21rfgy	_dockerswarm-worker-vmss_1	Ready	Active



Note: This is an Azure deployment, but the VMs are running Linux, not Windows. We can run any Linux x64-based container images on this swarm, but we can't run Windows-based images. The [Docker EE template on Azure Marketplace](#) lets you create a hybrid swarm with Linux and Windows nodes.

[VM Scale Sets in Azure](#) can be configured to autoscale based on properties such as amount of CPU use. When a scaling event is triggered, a new VM is created from the same image as the other VMs. For example, in the case of Docker for Azure, a new VM would be created from the worker image and join the swarm. Currently, there is no functionality to rebalance the swarm when a new node joins, which means you'd need to manually scale services on the swarm to make use of the new node(s).

Multicloud Docker swarms

We touched on Docker networks in [Chapter 5, Orchestrating Systems with Docker](#), but the functionality goes beyond creating a virtual network for machines that are already within the same physical network. You can create an overlay network that spans physical networks, which means you can create a Docker swarm with nodes running on different clouds.



Note: *The fact that you can do this doesn't mean that you should. Swarm nodes are chatty, sending regular heartbeat messages, and a high latency link between different datacenters, or even different continents, could cause problems if the manager thinks worker nodes have dropped out. In the worst case, you could have a network partition between clouds, which means manager nodes can't reach each other and you'll end up with different managers in different clouds. But with careful planning and testing, this architecture can be useful in certain scenarios.*

I'll illustrate this with a simple swarm. I have a VM running Ubuntu with Docker in Azure that has the firewall ("Network Security Group" in Azure terminology) set up to allow public Internet access to the swarm ports. I can create a swarm in the usual way, but I need to specify two IP addresses, as in Code Listing 89.

Code Listing 89: Initializing a Swarm with Two IP Addresses

```
$ docker swarm init --advertise-addr 13.74.254.111 --listen-addr 10.3.0.4
```

The `--advertise-addr` option specifies the public IP address that the manager should listen to for connections, but that's a virtual IP address managed by Azure—the VM itself doesn't know about the address. The `--listen-addr` option specifies the actual IP address the manager should listen on, which means the manager can bind to one IP address but advertise another.

I can now join one of my DigitalOcean droplets to the swarm using the same `join` command and by specifying the public virtual IP address of the Azure manager node, as seen in Code Listing 90.

Code Listing 90: Joining a Swarm from Another Cloud

```
$ docker swarm join --token SWMTKN-1-3jsfck3x9s9sme8rmtk31321z28qu46o66399c555x9hqewmvx-bfsgl92blu2rng12o5xdvli9813.74.254.111:2377
```

This node joined a swarm as a worker.

This gives me a swarm with two nodes running in different countries on different clouds. Code Listing 91 shows the node listing from the Azure manager.

Code Listing 91: Listing the Nodes in a Multicloud Swarm

```
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
0ejba0pkdtkqjsqs5vkskvt91	do-node-01	Ready	Active	
47950fi8w0evsk20ypkj4adk9 *	node0-ub	Ready	Active	Leader

I can use this multicloud swarm like any other, creating services with many replicas that span across the nodes. In Code Listing 92, I create a network for containers to communicate in.

Code Listing 92: Creating an Overlay Network on the Swarm

```
$ docker network create -d overlay website-net
```



Note: The default network type is *bridge*, which is used for single hosts. For networks that span multiple hosts, Docker has the *overlay* network, which is what you need to use for Docker swarm networks.

The overlay network can be specified when a service is created, adding all the containers to the network. In Code Listing 93, I create the basic Nginx service with enough replicas to span both nodes, then I list out the service processes.

Code Listing 93: Creating a Service Using an Overlay Network

```
$ docker service create --name website --publish 80:80 --replicas 4 --network website-net nginx:alpine
```

...

```
$ docker service ps website
```

ID	NAME	IMAGE	NODE	DESIRED STATE
CURRENT STATE	ERROR			
aq1ug1p1zknrfn3ucawbcfxhh	website.1	nginx:alpine	node0-ub	Running
Running 13 seconds ago				
1w7gpilweobxzjh8lc993ce8w	website.2	nginx:alpine	do-node-01	Running
Running 13 seconds ago				
9b52hijql5ugd723rql21bvvd	website.3	nginx:alpine	node0-ub	Running
Running 13 seconds ago				
5oiscpz73secc7hpe1bfdazg	website.4	nginx:alpine	do-node-01	Running

Running 13 seconds ago

We see that there are four containers running, two each on the manager in Azure and the worker in DigitalOcean. In order to verify that the website is running, I can make an HTTP GET request to the public IP addresses for the nodes, as in Code Listing 94.

Code Listing 94: Checking the Service with cURL

```
$ curl http://138.68.129.206/ #DigitalOcean

<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>

...

$ curl http://13.74.254.111/ #Azure

<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>
```

With a multicloud setup like this, you can load balance across the nodes using a DNS service configured to serve either the Azure or the DigitalOcean endpoints. With an advanced DNS system, such as [Amazon's Route 53](#), you can put a health check on each endpoint so that you can configure Azure as your primary endpoint but have traffic routing to DigitalOcean if Azure is unavailable or busy.

Multicloud swarms, or swarms that are a mixture of cloud and on-premise hosts, offer some very interesting opportunities. But you should be aware of the security implications and the potential performance issue with swarm heartbeats and control plane messages crossing datacenters.

Summary

Running Docker in the cloud gives you the benefit of huge scale on-tap, which means you can easily spin up new nodes and add them to your swarm. This is even easier in the Docker for AWS and Docker for Azure products, as the swarms make use of the native scaling options for each platform, such as VM Scale Sets in Azure.

Centralizing on Docker for your packaging model also isolates you from vendor lock-in in the cloud. Your infrastructure deployment scripts will be specific to whichever cloud you use, but your application packages will run the same way on any cloud. So, if you start with AWS but then cut a good deal with Azure, you can easily shift your whole workload without any application code changes.

This IaaS approach to the cloud means taking on the burden of administering a fleet of VMs, but there are products available that simplify the management and add additional layers of security to your application estate.

In the final chapter, we'll look at the product suite from Docker, Inc., which is aimed at running safe and healthy production deployments of Docker, and we'll see where to go next on our Docker journey.

Chapter 9 Continuing Your Docker Journey

Next steps with Docker

Docker is a simple technology, and this *Succinctly* e-book has enough detail that you should be comfortable starting to use it. This final chapter will look at some useful next steps to consider when you're ready to try Docker with your own applications.

Docker is an established technology with technical support and financial backing from major enterprises, such as Amazon, Microsoft, and IBM, and it's in production use at Netflix, Uber, and other cutting-edge tech companies. Application containers are a new model of compute, and Docker can genuinely revolutionize the way you build, run, and support software—but it doesn't have to be a violent revolution.

In this final chapter, we'll look at products from Docker, Inc. that simplify the administration of Docker in your own datacenters and in the cloud while providing commercial support for your Docker hosts. We'll cover some of the main use cases for Docker that are emerging from successful implementations, and I'll offer advice about where you can go next with Docker.

Docker and Docker, Inc.

The core technologies in the Docker ecosystem—the Docker server, the CLI, Docker Compose, Docker Machine, and the “Docker for” range—are all free, open-source tools. The major contributor to the open-source projects is Docker, Inc., a commercial company created by the founders of Docker. In addition to the free tools, Docker, Inc. has a set of commercial products for enterprises and companies that want software support.

The Docker, Inc. products make it easier to manage multiple hosts, run a secure image registry, work with different infrastructures, and monitor the activity of your systems. Docker Enterprise Edition (Docker EE) runs on-premises, and in the cloud.

Docker EE

Docker Enterprise Edition comes in two flavors—Basic gives you production support for your container runtime, and Advanced gives you a full Containers-as-a-Service solution which you can run in the datacenter or in any cloud. [Docker EE Basic](#) has the same feature set the free version of Docker (called Docker CE or Community Edition), so you can run containers and swarms with production support. Docker EE Advanced has a much larger feature set, which will accelerate your move to production.

Docker EE Advanced has two components—the Universal Control Plane (UCP), which is where you manage containers and hosts, and the Docker Trusted Registry (DTR), which is your own secure, private image registry. UCP and DTR are integrated to provide a secure software supply chain.

UCP sits on top of the orchestration layer, and it supports multiple schedulers. At the time of writing it supports the classic Docker Swarm and Docker swarm mode. Both technologies run on the same cluster. Support for [Kubernetes](#) is coming to Docker EE, and you will be able to deploy applications on Kubernetes and Docker swarm across the same set of servers.

UCP has a web UI for managing your on-premise Docker estate. Figure 15 depicts the homepage with the dashboard showing how many resources UCP is managing—hosts, images, containers, and applications. It also shows how hard the hosts are working.

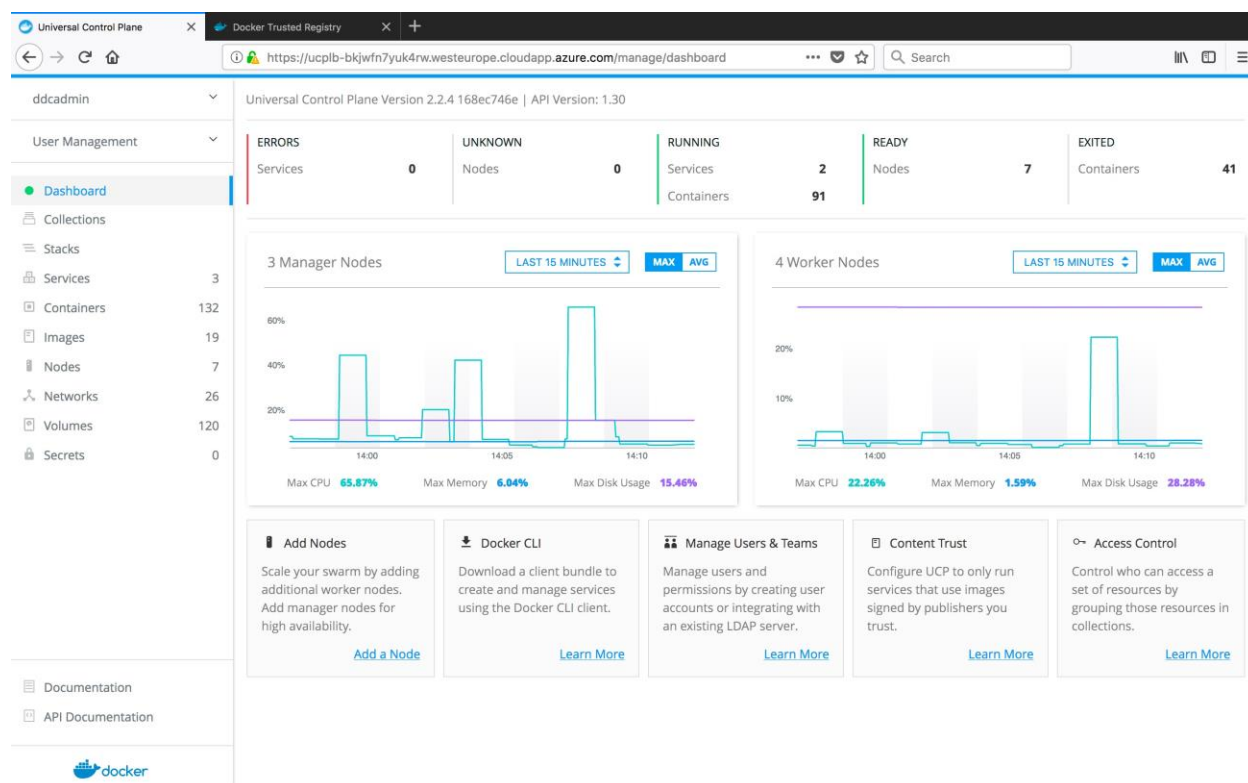


Figure 15: The Universal Control Plane Dashboard

UCP is a consolidated management portal. You can create volumes and networks, run individual containers from images, or run a multicontainer application from a Docker Compose file. From the node view, you can see the overview of the machines in the cluster. In Figure 16 we can see a hybrid cluster with Linux manager nodes, Linux worker nodes, and Windows worker nodes.

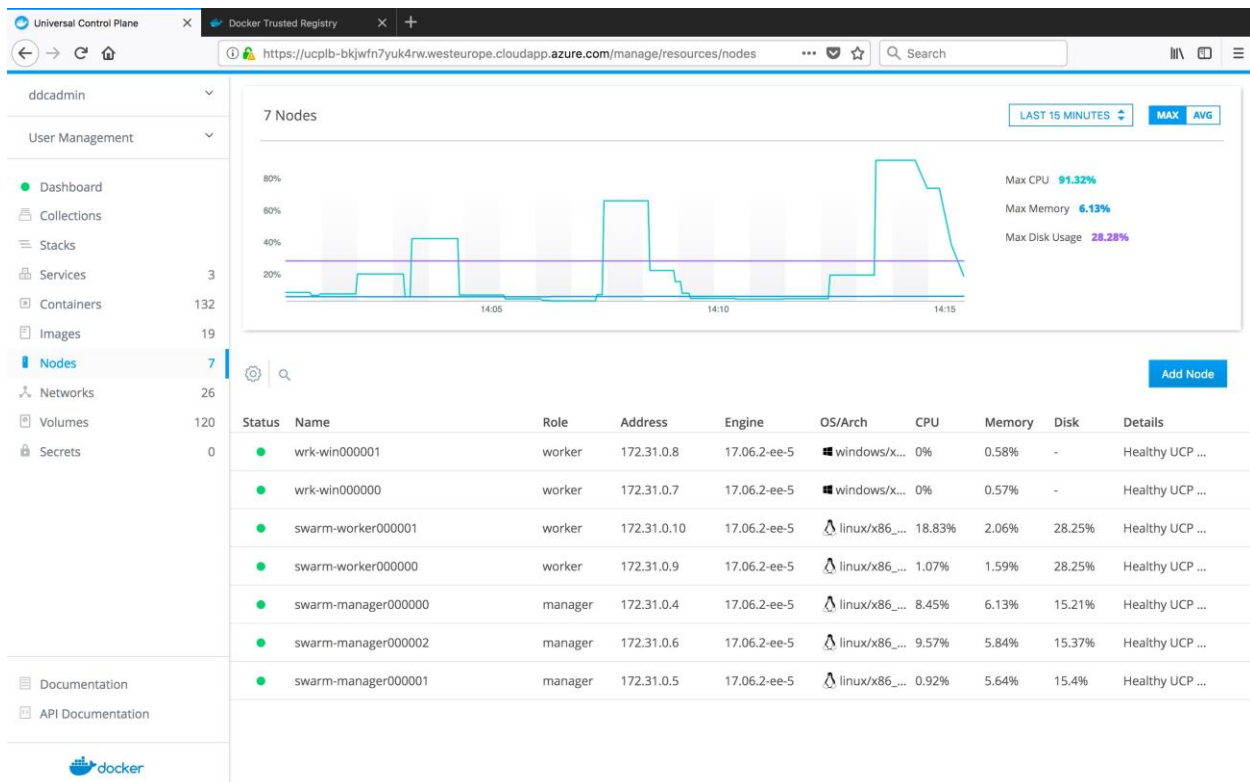


Figure 16: Checking Swarm Nodes in UCP

On the containers page, you can drill down into a running container to check the CPU memory and network usage, connect with a console, or check the logs, as in Figure 17.

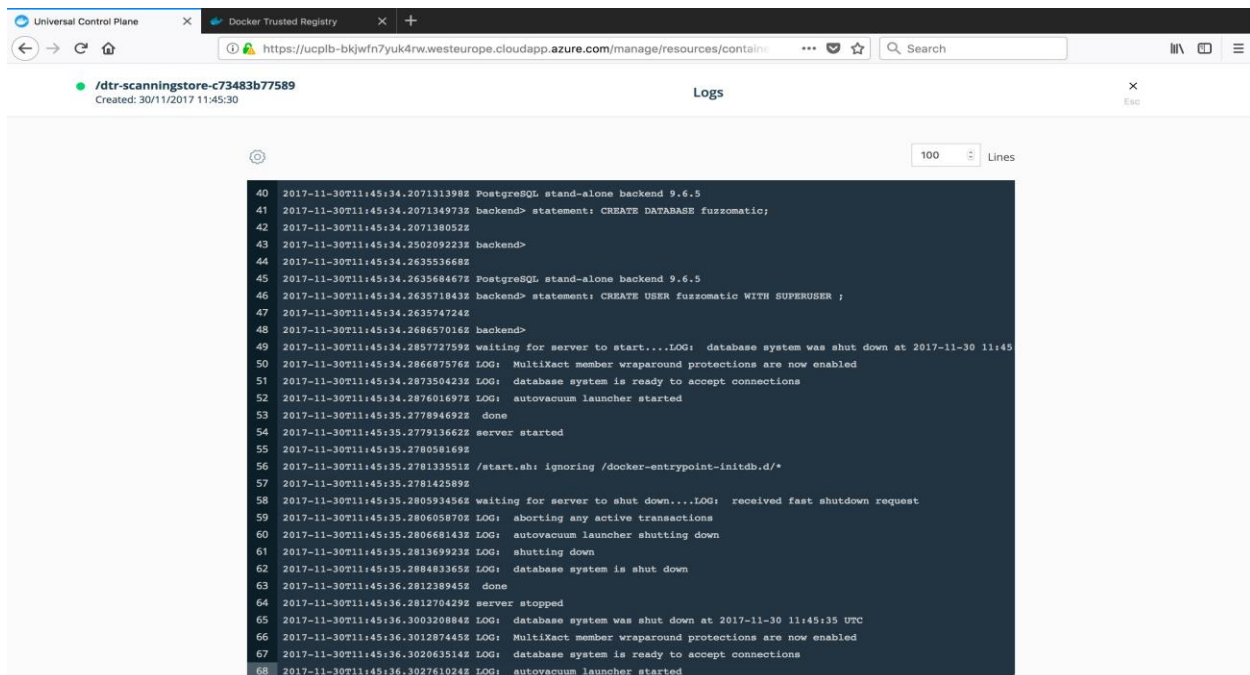


Figure 17: Viewing Container Logs from UCP

UCP gives you role-based access control for your Docker resources. All resources can have labels applied that are arbitrary key-value pairs. Permissions are applied to labels. You can create users and teams inside UCP or connect to an LDAP source to import user details and grant permissions to resource labels at the team or user level.

You can have UCP managing multiple environments and configure permissions so that developers can run containers on the integration environment, but only testers can run them on the staging environment, and only the production support team can run containers in production. UCP communicates with Docker Engine using the same APIs, but the engines are secured, too. In order to manage UCP engines with the Docker CLI, you need a certificate generated by UCP, which means you can't bypass the UCP permissions.

You secure the images that run on the hosts with the other part of Docker Datacenter—Docker Trusted Registry (DTR). DTR uses the same Registry API as the Hub, which means you work with it in the same way, but it also has its own web UI. The DTR UI is similar to the Docker Hub UI, but as a private registry, and therefore you have more control. DTR shares the authentication you have set up in UCP. Users can have their own repositories, and you can create organizations for shared repositories. Figure 18 shows the DTR homepage for an admin user with access to all repositories.

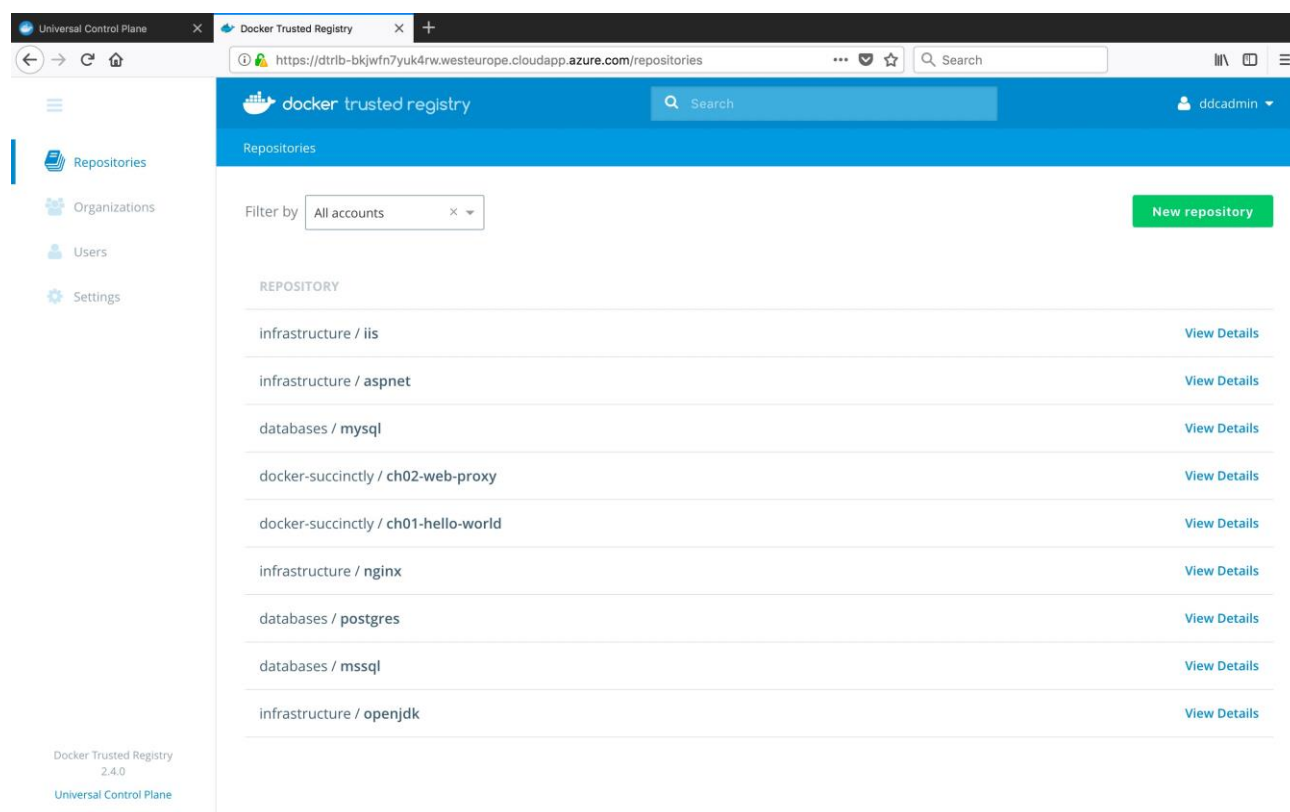


Figure 18: The Docker Trusted Registry

DTR and UCP are closely integrated, which means you can use your own image repositories as the source for running containers and applications on the UCP swarm.



Tip: DTR has the same security-scanning functionality as the Docker Hub, which means your own private images can be scanned for vulnerabilities. And because DTR also has content trust, you can securely sign images and configure your Docker Engines so that they run only containers from signed images.

Docker Cloud

[Docker Cloud](#) provides a similar set of functionality to Docker EE, but it's for managing images and nodes in public clouds. Docker EE is a good fit in the enterprise, where you will need fine-grained access control and security for a large number of images and hosts. For smaller companies or project teams that are deploying to public clouds, Docker Cloud gives you a similar level of control but without the commercial support.

Docker Cloud works similarly to Docker Machine for provisioning new nodes. It has providers for all the major clouds, and you can spin up new hosts running Docker from Docker Cloud. When you use Docker Cloud, you have two commercial relationships: your infrastructure is billed by the cloud provider and the additional Docker features are billed by Docker, Inc.

Figure 19 shows me logged in to <https://cloud.docker.com> (you can use your Docker Hub credentials and try out Docker Cloud; you don't need to start with a paid plan) with my DigitalOcean and Azure accounts set up.

The screenshot displays the Docker Cloud interface. The top navigation bar includes 'docker cloud', a search bar, and a user profile for 'sixeyed'. The left sidebar contains a 'Swarm Mode' toggle and a menu with categories: BUILD (Repositories), APPLICATIONS (Stacks, Services, Containers), INFRASTRUCTURE (Node Clusters, Nodes), and SETTINGS (Cloud Settings, Status). The 'Cloud Settings' page features a 'General' tab and a user profile for 'Elton Stoneman'. The 'Cloud providers' section lists five providers: Amazon Web Services, Digital Ocean, Microsoft Azure, SoftLayer, and Packet. Each provider has a link to 'Add new credentials' and a 'Free Tier' or '\$25 Code' option. The 'Source providers' section lists GitHub and Bitbucket, with GitHub linked to the 'sixeyed' account and Bitbucket showing 'No account linked'.

Provider	Account	Actions	Plan
Amazon Web Services	Add new credentials		Free Tier
Digital Ocean	elton@sixeyed.com		\$20 Code
Microsoft Azure	3481cd2a-ed2d-4afd-8bbb-96ea65c835f2		Global Admin Free trial
SoftLayer	Add new credentials		Free trial
Packet	Add new credentials		\$25 code

Provider	Account	Actions
GitHub	sixeyed	
Bitbucket	No account linked	

Figure 19: Cloud Providers in Docker Cloud

You can create clusters of nodes on your chosen cloud provider, and Docker Cloud will create a VM with the commercially supported Docker Engine installed. Docker Cloud integrates with the public Docker Hub in the same way that UCP and DTR integrate on-premise. You can deploy containers from your images in the Hub, and you can create private repositories that are only visible to you or your organization.

In Figure 20, my Docker Hub repositories are shown in the Docker Cloud UI, and I can deploy to my cloud Docker nodes directly from here.

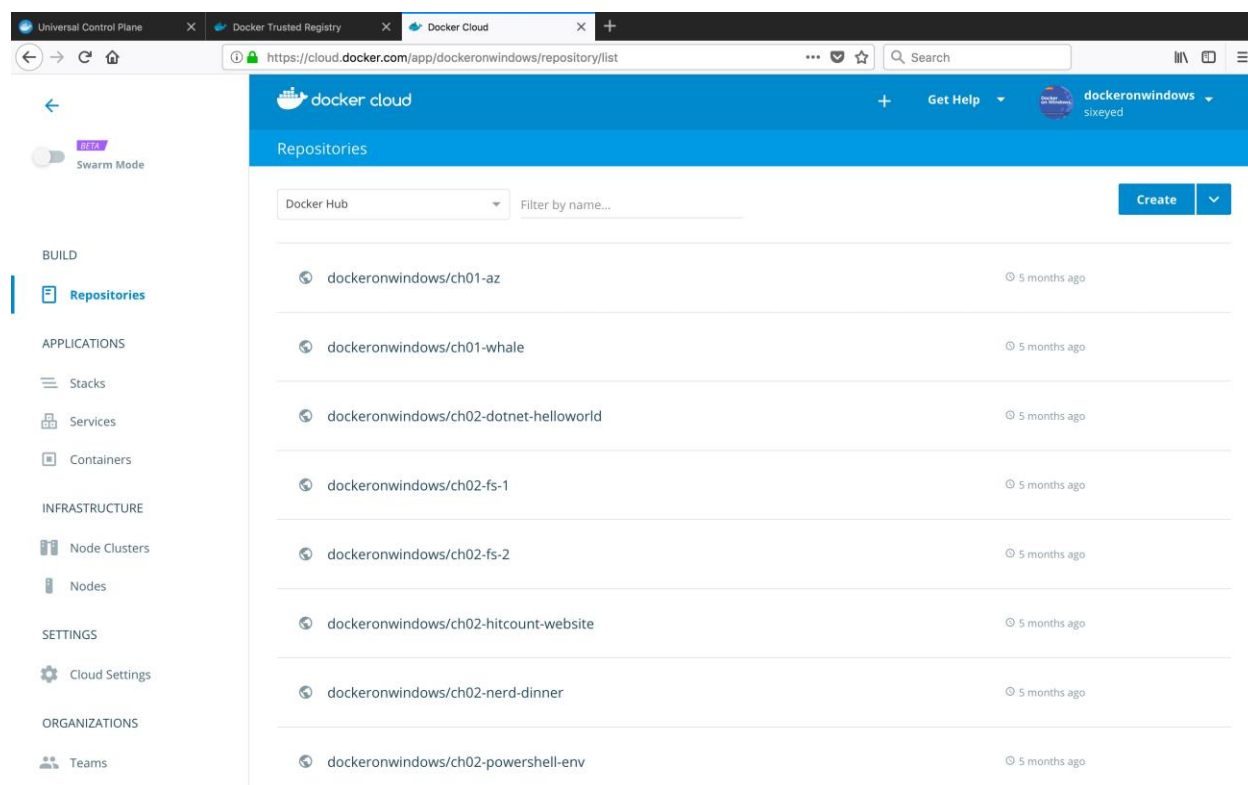


Figure 20: Image Repositories in Docker Cloud

With a paid plan for Docker Cloud, you get the same image security scanning for your own repositories that's available in Docker Trusted Registry and for official repositories on the Hub.



Tip: This was a very quick look at Docker Enterprise Edition and Docker Cloud, but both are worth further investigation. Commercial support and secure access are prerequisites for many enterprises adopting new technologies.

Docker and DevOps

Docker is facilitating major changes in the way software is built and managed. The capabilities of the Docker product suite, together with the commercial support from Docker, Inc., are foundations for building better software and delivering it more quickly.

The Dockerfile isn't often held up as a major contribution to the software industry, but it can play a key role for organizations looking to adopt DevOps. A transition to DevOps means unifying existing development and operations teams, and the Dockerfile is an obvious central point where different concerns can meet.

In essence, the Dockerfile is a deployment guide. But instead of being a vague document open to human interpretation, it's a concise, clear, and actionable set of steps for packaging an application into a runnable container image. It's a good first place for dev and ops teams to meet—developers can start with a simple Dockerfile to package the app, then operations can take it over, hardening the image or swapping out to a compatible but customized base image.

A smooth deployment pipeline is crucial to successful DevOps. Adopting Docker as your host platform means you can generate versioned images from each build, deploy them to your test environment, and, when your test suite passes, promote to staging and production knowing that you will be running the exact same codebase that passed your tests.



Tip: Docker swarm mode supports rolling service updates, which means that when you upgrade a running app to a new version, Docker will incrementally take down instances of the old version and bring up new ones. This makes for very safe upgrades of stateless apps. See the [rolling update tutorial](#) to learn more about rolling upgrades.

Docker also makes a big contribution to DevOps through the simplicity of the host environment. Within the datacenter, your VM setup can be as lean as simply installing a base OS and Docker—everything else, including business applications and operational monitoring, runs in containers on the swarm.

New operating systems are likely to grow in popularity to support Docker's simplicity. Canonical has Ubuntu Snappy Core, and Microsoft has Nano Server. Both of them are lightweight operating systems that can be used for container-based images, but they can also be used as the host operating system on the server. They have a minimal feature set that equates to a minimal attack surface and greatly reduced patching requirements.

The key technical practices of DevOps—infrastructure-as-code, continuous delivery, continuous monitoring—are all made simpler with Docker, which reduces your application environment to a few simple primitives: the image, the container, and the host.

Docker and microservices

Containerized solutions are not only suited to greenfield application development. Taking an existing application and breaking suitable parts out into separate application containers can transform a heavyweight, monolithic app into a set of lean, easily upgradable services. Lightweight containers and the built-in orchestration from Docker provide the backbone for microservice architectures.

The microservice architecture is actually easier to apply to an existing application than to a new system because you already have a clear understanding of where to find the logical service boundaries and the physical pain points. There are two good candidates for breaking out from a monolith into separate services without a major rewrite of your application:

- Small, well-contained components that add business value if they are easy to change.
- Larger, heavily integrated components that add business value if they stay the same.

As an example, consider a web app used by a business that frequently wants to change the homepage in order to try increasing customer engagement while rarely wanting to change the checkout stage because it feels that stage does all it needs to do. If that app is currently a deployed as single, monolithic unit, you can't make a fast, simple change to the homepage—you need to test and deploy the entire app, including the checkout component that you haven't changed.

Breaking the homepage and the checkout component into separate application containers means you can make changes to the homepage and deploy frequently while knowing you won't impact the checkout component. You should only test the changes you make, and you can easily roll back if the new version isn't well received.

The cost of microservices is the additional complexity of having many small services working together. However, running those services in Docker containers makes it a problem of orchestration, and that concern is well served by the Docker ecosystem.

Next steps: containerize what you know

You don't need to embark on a DevOps transition or re-architect your key systems in order to make good use of Docker. If you start by migrating applications you use regularly and understand well, you'll soon gain a lot of valuable experience—and confidence—with running applications in containers.

Development toolsets are a good place to start, and it's surprising how often a tool you rely on has already been wrapped into a Docker image on the Hub. At minimum, all you need to do is pull the image and check it out. At most, you might build a new image using your preferred base OS and tools.

Lots of teams start by containerizing their own core systems—[GitLab](#) and [Jenkins](#) have very popular repositories on the Hub, and you can find plenty of sample images for tools such as file shares, wikis, and code analysis.

Using containers every day is the quickest way to get comfortable with Docker, and you'll soon be finding all sorts of scenarios that Docker improves. Enjoy the journey!