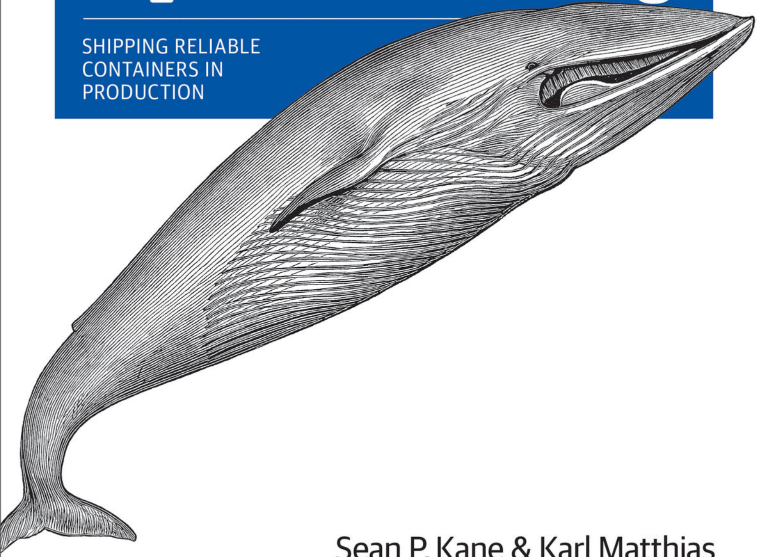
O'REILLY®

2nd Edition

Docker
Up & Running

SHIPPING RELIABLE
CONTAINERS IN
PRODUCTION



Sean P. Kane & Karl Matthias

Docker: Up & Running

SECOND EDITION

Shipping Reliable Containers in Production

Sean P. Kane and Karl Matthias

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Docker: Up and Running

by Sean P. Kane and Karl Matthias

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Development Editor: Virginia Wilson

Production Editor: Justin Billing

Copyeditor: Rachel Monaghan

Proofreader: Gillian McGarvey

Indexer: Ellen Troutman-Zaig

Interior Designer: David Futato

Cover Designer: Randy Comer

Illustrator: Rebecca Demarest

Technical Reviewers: Ksenia Burlachenko, Chelsea Frank, Mihai Todor, and Rachid Zarouali

September 2018: Second Edition

Revision History for the Second Edition

- 2018-09-07: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492036739 for release details.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-03673-9

[LSI]

Chapter 1. Introduction

Docker was first introduced to the world—with no pre-announcement and little fanfare—by Solomon Hykes, founder and CEO of a company then called dotCloud, in a five-minute lightning talk at the Python Developers Conference in Santa Clara, California on March 15, 2013. At the time of this announcement, only about 40 people outside of dotCloud had been given the opportunity to play with Docker.

Within a few weeks of this announcement, there was a surprising amount of press. The project was quickly open-sourced and made publicly available on GitHub, where anyone could download and contribute to the project. Over the next few months, more and more people in the industry started hearing about Docker and how it was going to revolutionize the way software was built, delivered, and run. And within a year, almost no one in the industry was unaware of Docker, but many were still unsure what it was exactly, and why people were so excited about it.

Docker is a tool that promises to easily encapsulate the process of creating a distributable artifact for any application, deploying it at scale into any environment, and streamlining the workflow and responsiveness of agile software organizations.

The Promise of Docker

While ostensibly viewed as a virtualization platform by people who are unfamiliar with Docker, it is far more than that. Docker's core domain spans a few crowded segments of the industry that include technologies like KVM, Xen, OpenStack, Mesos, Capistrano, Fabric, Ansible, Chef, Puppet, SaltStack, and so on. There is something very telling about the list of products that Docker competes with, and maybe you've spotted it already. Most engineers would not say that virtualization products compete with configuration management tools, yet both technologies are being disrupted by Docker. This is largely because Docker significantly alters the playing field, and this has a very direct impact on each of these traditionally isolated segments inside the DevOps pipeline. The technologies in that list are also generally acclaimed for their ability to improve productivity, and that's exactly what has given Docker so much buzz. Docker sits right in the middle of some of the most enabling technologies of the last decade and can bring significant improvements to almost every step of the pipeline.

If you were to do a feature-by-feature comparison of Docker and the reigning champion in any of these individual areas, Docker would very likely look like a middling competitor. It’s stronger in some areas than others, but what Docker brings to the table is a feature set that crosses a broad range of workflow challenges. By combining the ease of application deployment tools like Capistrano and Fabric with the ease of administrating virtualization systems, and then providing hooks that make workflow automation and orchestration easy to implement, Docker provides a very enabling feature set.

Lots of new technologies come and go, and a dose of skepticism about the newest rage is always healthy. Without digging deeper, it would be easy to dismiss Docker as just another technology that solves a few very specific problems for developers or operations teams. If you look at Docker as a pseudo-virtualization or deployment technology alone, it might not seem very compelling. But Docker is much more than it seems on the surface.

It is hard and often expensive to get communication and processes right between teams of people, even in smaller organizations. Yet we live in a world where the communication of detailed information between teams is increasingly required to be successful. Discovering and implementing a tool that reduces the complexity of that communication while aiding in the production of more robust software would be a big win. And that’s exactly why Docker merits a deeper look. It’s no panacea, and the way that you implement Docker within your organization requires some critical thought, but Docker is a good approach to solving some real-world organizational problems and helping enable companies to ship better software faster. Delivering a well-designed Docker workflow can lead to happier technical teams and real savings for the organization’s bottom line.

So where are companies feeling the most pain? Shipping software at the speed expected in today’s world is hard to do well, and as companies grow from one or two developers to many teams of developers, the burden of communication around shipping new releases becomes much heavier and harder to manage. Developers have to understand a lot of complexity about the environment they will be shipping software into, and production operations teams need to increasingly understand the internals of the software they ship. These are all generally good skills to work on because they lead to a better understanding of the environment as a whole and therefore encourage the designing of robust software, but these same skills are very difficult to scale effectively as an organization’s growth accelerates.

The details of each company’s environment often require a lot of communication that doesn’t directly build value for the teams involved. For example, requiring developers to ask an operations team for *release 1.2.1* of a particular library slows them down and provides no direct business value to the company. If developers could simply upgrade the version of the library they use, write their code, test with the new version, and ship it, the delivery time would be measurably shortened and fewer risks would be involved in deploying the change. If operations engineers could upgrade software on the host system without having to coordinate with multiple teams of application developers, they could move faster. Docker helps to build a layer of isolation in software that reduces the burden of communication in the world of humans.

Beyond helping with communication issues, Docker is opinionated about software architecture in a way that encourages more robustly crafted applications. Its architectural philosophy centers on atomic or throwaway containers. During deployment, the whole running environment of the old application is thrown away with it. Nothing in the environment of the application will live longer than the application itself, and that's a simple idea with big repercussions. It means that applications are not likely to accidentally rely on artifacts left by a previous release. It means that ephemeral debugging changes are less likely to live on in future releases that picked them up from the local filesystem. And it means that applications are highly portable between servers because all of the state has to be included directly into the deployment artifact and be immutable, or sent to an external dependency like a database, cache, or file server.

All of this leads to applications that are not only more scalable but more reliable as well. Instances of the application container can come and go with little impact on the uptime of the frontend site. These are proven architectural choices that have been successful for non-Docker applications, but the design choices enforced by Docker mean that Docker-ized applications are *required* to follow these best practices. And that's a good thing.

Benefits of the Docker Workflow

It's hard to cohesively categorize all of the things Docker brings to the table. When implemented well, it benefits organizations, teams, developers, and operations engineers in a multitude of ways. It makes architectural decisions simpler because all applications essentially look the same on the outside from the hosting system's perspective. It makes tooling easier to write and share between applications. Nothing in this world comes with benefits and no challenges, but Docker is surprisingly skewed toward the benefits. Here are some more of the benefits you get with Docker:

Packaging software in a way that leverages the skills developers already have

> Many companies have had to create positions for release and build engineers in order to manage all the knowledge and tooling required to create software packages for their supported platforms. Tools like rpm, mock, dpkg, and pbuilder can be complicated to use, and each one must be learned independently. Docker wraps up all your requirements

together into one package that is defined in a single file.

Bundling application software and required OS filesystems together in a single standardized image format

> In the past, you typically needed to package not only your application, but also many of the dependencies that it relied on, including libraries and daemons. However, you could never ensure that 100 percent of the execution environment was identical. For natively compiled code, this meant that your build system needed to have exactly the same versions of shared libraries as your production environment. All of this made packaging difficult to master, and hard for many companies to accomplish reliably. Often someone running Scientific Linux would resort to trying to deploy a community package tested on Red Hat Linux, hoping that the package was close enough to what they needed. With Docker, you deploy your application along with every single file required to run it. Docker's layered images make this an efficient process that ensures that your application is running in the expected environment.

Using packaged artifacts to test and deliver the exact same artifact to all systems in all environments

> When developers commit changes to a version control system, a new Docker image can be built, which can go through the whole testing process and be deployed to production without having to be recompiled or repackaged at any step in the process, unless that is specifically desired.

Abstracting software applications from the hardware without sacrificing resources

> Traditional enterprise virtualization solutions like VMware are typically used when people need to create an abstraction layer between the physical hardware and the software applications that run on it, at the cost of resources. The hypervisors that manage the VMs and each VM's running kernel use a percentage of the hardware system's resources, which are then no longer available to the hosted applications. A container, on the other hand, is just another process that talks directly to the Linux kernel and therefore can utilize more resources, up until the system or quota-based limits are reached.

When Docker was first released, Linux containers had been around for quite a few years, and many of the other technologies that it is built on are not entirely new. However, Docker's unique mix of strong architectural and workflow choices combine into a whole that is much more powerful than the sum of its parts. Docker finally makes Linux containers, which have been publicly available since 2008, approachable to the average technologist. It fits containers relatively easily into the existing workflow and processes of real companies. And the problems discussed earlier have been felt by so many people that interest in the Docker project has been accelerating faster than anyone could have reasonably expected.

From a standing start only a few years ago, Docker has seen rapid iteration and now has a huge feature set and is deployed in a vast number of production infrastructures across the planet. It is rapidly becoming one of the foundation layers for any modern distributed system. A large number of companies now leverage Docker as a solution to some of the serious complexity issues that they face in their application delivery processes.

What Docker Isn’t

Docker can be used to solve a wide range of challenges that other categories of tools have traditionally been enlisted to fix; however, Docker’s breadth of features often means that it lacks depth in specific functionality. For example, some organizations will find that they can completely remove their configuration management tool when they migrate to Docker, but the real power of Docker is that although it can replace some aspects of more traditional tools, it is also usually compatible with them or even enhanced in combination with them. In the following list, we explore some of the tool categories that Docker doesn’t directly replace but that can often be used in conjunction to achieve great results:

Enterprise virtualization platform (VMware, KVM, etc.)

> A container is not a virtual machine in the traditional sense. Virtual machines contain a complete operating system, running on top of a hypervisor that is managed by the underlying host operating system. The biggest advantage to hardware virtualization is that it is easy to run many virtual machines with radically different operating systems on a single host. With containers, both the host and the containers share the same kernel. This means that containers utilize fewer system resources but must be based on the same underlying operating system (i.e., Linux).

Cloud platform (OpenStack, CloudStack, etc.)

> Like enterprise virtualization, the container workflow shares a lot of similarities—on the surface—with cloud platforms. Both are traditionally leveraged to allow applications to be horizontally scaled in response to changing demand. Docker, however, is not a cloud platform. It only handles deploying, running, and managing containers on preexisting Docker hosts. It doesn’t allow you to create new host systems (instances), object stores, block storage, and the many other resources that are often managed with a cloud platform. That being said, as you start to expand your Docker tooling, you should start to experience

more and more of the benefits that one traditionally associates with the cloud.

Configuration management (Puppet, Chef, etc.)

Although Docker can significantly improve an organization’s ability to manage applications and their dependencies, it does not directly replace more traditional configuration management. Dockerfiles are used to define how a container should look at build time, but they do not manage the container’s ongoing state, and cannot be used to manage the Docker host system. Docker can, however, significantly lessen the need for complex configuration management code. As more and more servers simply become Docker hosts, the configuration management codebase can become much smaller, and Docker can be used to ship the more complex application requirements inside of Docker images.

Deployment framework (Capistrano, Fabric, etc.)

Docker eases many aspects of deployment by creating container images that encapsulate all the dependencies of an application in a manner that can be deployed, in all environments, without changes. However, Docker can’t be used to automate a complex deployment process by itself. Other tools are usually still needed to stitch together the larger workflow. That being said, because the method required to deploy containers will be consistent on all hosts, a single deployment workflow should suffice for most, if not all, of your Docker-based applications.

Workload management tool (Mesos, Kubernetes, Swarm, etc.)

An orchestration layer (including the built-in Swarm mode) must be used to coordinate work intelligently across a pool of Docker hosts, track the current state of all the hosts and their resources, and keep an inventory of running containers.

Development environment (Vagrant, etc.)

Vagrant is a virtual machine management tool for developers that is often used to simulate server stacks that closely resemble the production environment in which an application is destined to be deployed. Among other things, Vagrant makes it easy to run Linux software on macOS and Windows-based workstations. Docker Machine is sufficient for many standard Docker workflows, but it doesn’t provide the breadth of features found in Vagrant, since it is very narrowly focused on the Docker workflow. However, as with many of the previous examples, when you start to fully utilize Docker, there is a lot less need to mimic a wide variety of production systems in development, since most production systems will simply be Docker servers, which can easily be reproduced locally.

Important Terminology

Here are a few terms that we will continue to use throughout the book and whose meanings you should become familiar with:

Docker client

> This is the `docker` command used to control most of the Docker workflow and talk to remote Docker servers.

Docker server

> This is the `dockerd` command that is used to start the Docker server process that builds and launches containers via a client.

Docker images

> Docker images consist of one or more filesystem layers and some important metadata that represent all the files required to run a Dockerized application. A single Docker image can be copied to numerous hosts. An image typically has both a name and a tag. The tag is generally used to identify a particular release of an image.

Docker container

> A Docker container is a Linux container that has been instantiated from a Docker image. A specific container can exist only once; however, you can easily create multiple containers from the same image.

Atomic host

> An atomic host is a small, finely tuned OS image, like Fedora CoreOS, that supports container hosting and atomic OS upgrades.

Wrap-Up

Completely understanding Docker can be challenging when you are coming at it without a strong frame of reference. In the next chapter we will lay down a broad overview of Docker: what it is, how it is intended to be used, and what advantages it brings to the table when implemented with all of this in mind.

Chapter 2. The Docker Landscape

Before you dive into configuring and installing Docker, a broad survey is in order to explain what Docker is and what it brings to the table. It is a powerful technology, but not a tremendously complicated one at its core. In this chapter, we'll cover the generalities of how Docker works, what makes it powerful, and some of the reasons you might use it. If you're reading this, you probably have your own reasons to use Docker, but it never hurts to augment your understanding before you jump in.

Don't worry—this chapter should not hold you up for too long. In the next chapter, we'll dive right into getting Docker installed and running on your system.

Process Simplification

Because Docker is a piece of software, it may not be obvious that it can also have a big positive impact on company and team processes if it is adopted well. So, let's dig in and see how Docker can simplify both workflows and communication. This usually starts with the deployment story. Traditionally, the cycle of getting an application to production often looks something like the following (illustrated in Figure 2-1):

1. Application developers request resources from operations engineers.

2. Resources are provisioned and handed over to developers.

3. Developers script and tool their deployment.

4. Operations engineers and developers tweak the deployment repeatedly.

5. Additional application dependencies are discovered by developers.

6. Operations engineers work to install the additional requirements.

7. Loop over steps 5 and 6 *N* more times.

8. The application is deployed.

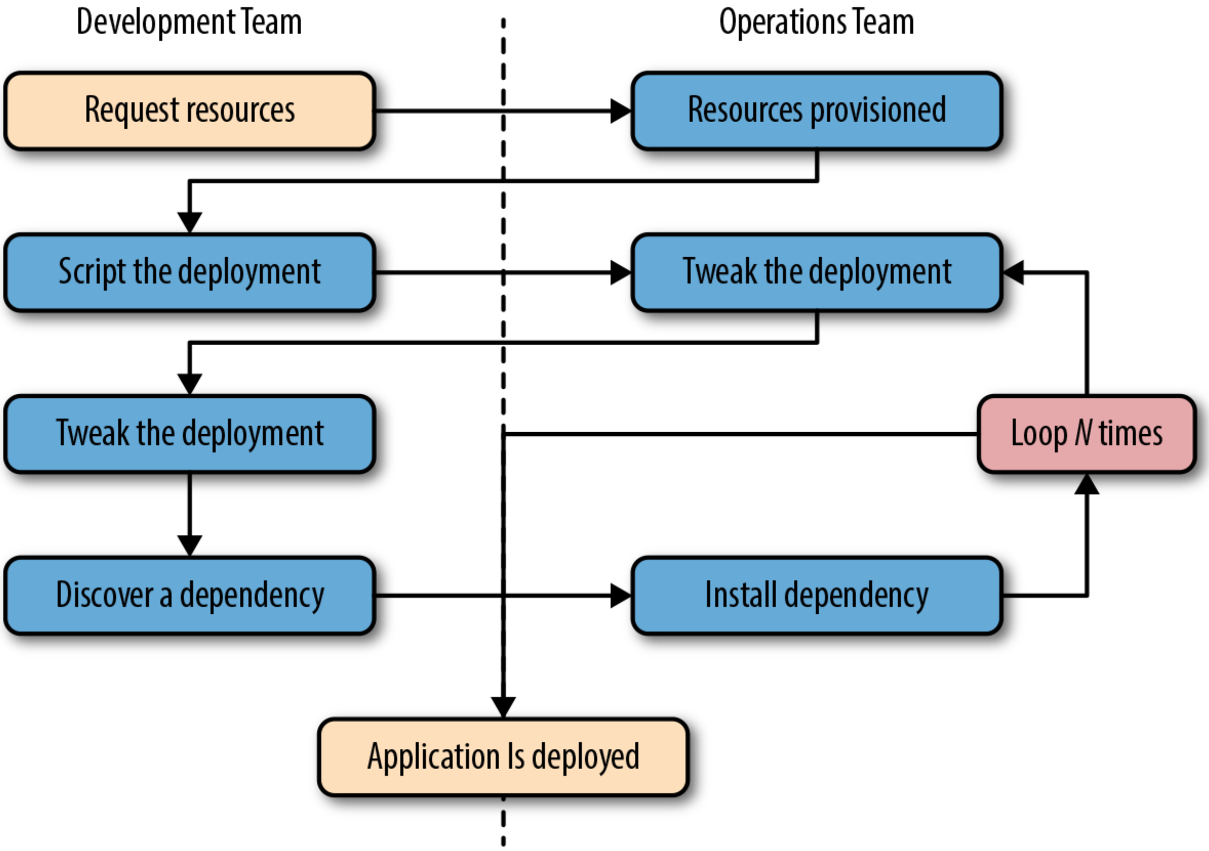


Figure 2-1. A traditional deployment workflow (without Docker)

Our experience has shown that when you are following traditional processes, deploying a brand new application into production can take the better part of a week for a complex new system. That's not very productive, and even though DevOps practices work to alleviate some of the barriers, it often requires a lot of effort and communication between teams of people. This process can be both technically challenging and expensive, but even worse, it can limit the kinds of innovation that development teams will undertake in the future. If deploying software is hard, time-consuming, and dependent on resources from another team, then developers may just build everything into the existing application in order to avoid suffering the new deployment penalty.

Push-to-deploy systems like Heroku have shown developers what the world can look like if you are in control of most of your dependencies as well as your application. Talking with developers about deployment will often turn up discussions of how easy things are on Heroku or similar systems. If you're an operations engineer, you've probably heard complaints about how much slower your internal systems are compared with deploying on Heroku.

Heroku is a whole environment, not just a container engine. While Docker doesn't try to be everything that is included in Heroku, it provides a clean separation of responsibilities and encapsulation of dependencies, which results in a similar boost in productivity. Docker also allows even more fine-grained control than Heroku by putting developers in control of everything, down to the OS distribution on which they ship their application. Some of the tooling and orchestrators built (e.g., Kubernetes, Swarm, or Mesos) on top of Docker now aim to replicate the simplicity of Heroku. But even though these platforms wrap more around Docker to provide a more capable and complex environment, a simple platform that uses only Docker still provides all of the core process benefits without the added complexity of a larger system.

As a company, Docker adopts an approach of “batteries included but removable.” This means that they want their tools to come with everything most people need to get the job done, while still being built from interchangeable parts that can easily be swapped in and out to support custom solutions.

By using an image repository as the hand-off point, Docker allows the responsibility of building the application image to be separated from the deployment and operation of the container. What this means in practice is that development teams can build their application with all of its dependencies, run it in development and test environments, and then just ship the exact same bundle of application and dependencies to production. Because those bundles all look the same from the outside, operations engineers can then build or install standard tooling to deploy and run the applications. The cycle described in Figure 2-1 then looks somewhat like this (illustrated in Figure 2-2):

1. Developers build the Docker image and ship it to the registry.

2. Operations engineers provide configuration details to the container and provision resources.

3. Developers trigger deployment.

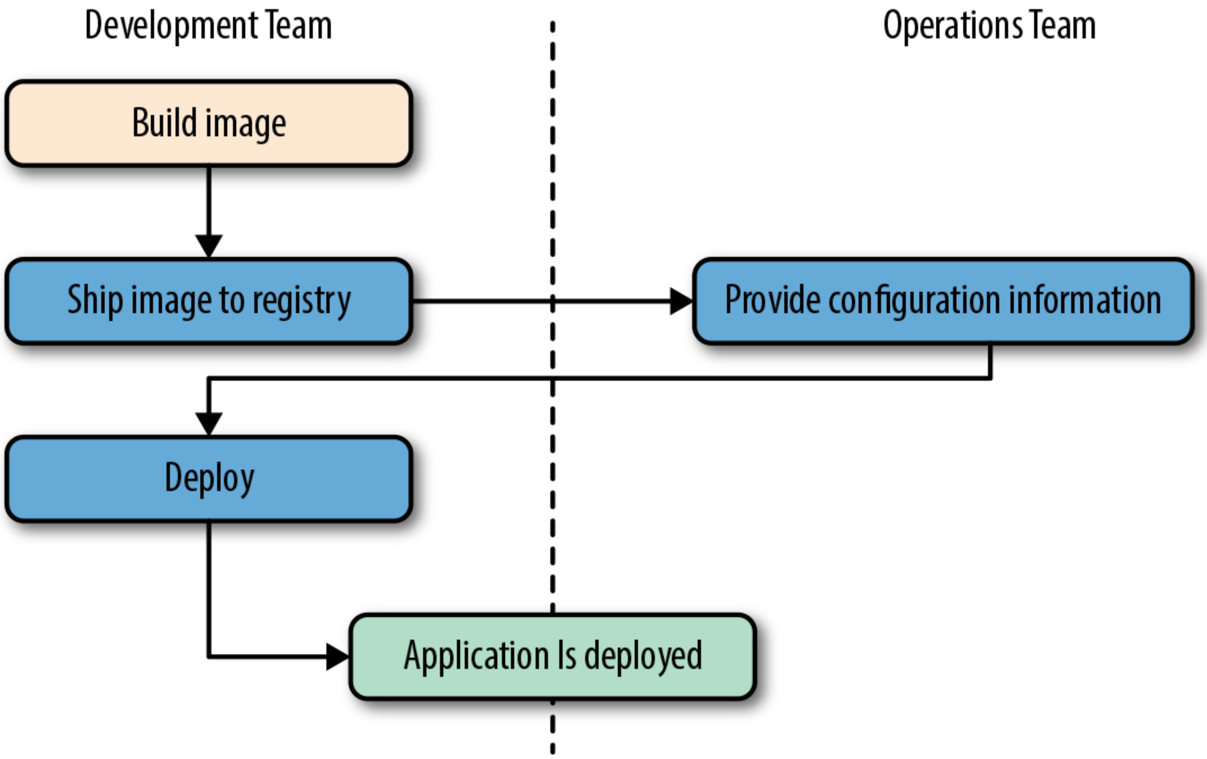


Figure 2-2. A Docker deployment workflow

This is possible because Docker allows all of the dependency issues to be discovered during the development and test cycles. By the time the application is ready for first deployment, that work is done. And it usually doesn't require as many handoffs between the development and operations teams. That's a lot simpler and saves a lot of time. Better yet, it leads to more robust software through testing of the deployment environment before release.

Broad Support and Adoption

Docker is already well supported, with the majority of the large public clouds offering some direct support for it. For example, Docker runs in AWS via multiple products like Elastic Container Service (ECS), Elastic Container Service for Kubernetes (EKS), Fargate, and Elastic Beanstalk. Docker can also be used on Google AppEngine, Google Kubernetes Engine, Red Hat OpenShift, IBM Cloud, Microsoft Azure, Rackspace Cloud, Docker's own Docker Cloud, and many more. At DockerCon 2014, Google's Eric Brewer announced that Google would be supporting Docker as its primary internal container format. Rather than just being good PR for these companies, what this meant for the Docker community was that a lot of money began to back the stability and success of the Docker platform.

Further building its influence, Docker's containers are becoming the lingua franca between cloud providers, offering the potential for "write once, run anywhere" cloud applications. When Docker released their `libswarm` development library at DockerCon 2014, an engineer from Orchard demonstrated deploying a Docker container to a heterogeneous mix of cloud providers at the same time. This kind of orchestration had not been easy before because every cloud provider provided a different API or toolset for managing instances, which were usually the smallest item you could manage with an API. What was only a promise from Docker in 2014 has since become fully mainstream as the largest companies continue to invest in the platform, support, and tooling. With most providers offering some form of Docker orchestration as well as the container runtime itself, Docker is already well supported for nearly any kind of workload in common production environments. If all of your tooling is around Docker, your applications can be deployed in a cloud-agnostic manner, allowing a huge new flexibility not previously possible.

That covers Docker containers and tooling, but what about OS vendor support and adoption? The Docker client runs directly on most major operating systems, and the server can run on Linux or Windows Server. The vast majority of the ecosystem is built around Linux servers, but other platforms are increasingly being supported. The beaten path is and will likely continue to be Linux servers running Linux containers.

> **NOTE**
>
> It is actually possible to run Windows containers natively (without a VM) on 64-bit versions of Windows Server 2016. However, 64-bit versions of Windows 10 Professional still require Hyper-V to provide the Windows Server kernel that is used for Windows containers. We dive into a little more detail about this in "Windows Containers".

In order to support the growing demand for Docker tooling in development environments, Docker has released easy-to-use implementations for macOS and Windows. These appear to run natively but are still sitting on top of a Linux kernel underneath. Docker has traditionally been developed on the Ubuntu Linux distribution, but most Linux distributions and other major operating systems are now supported where possible. RedHat, for example, has gone all-in on containers and all of their platforms have first-class support for Docker. With the near ubiquity of containers in the Linux realm, we now have distributions like Red Hat's CoreOS, which is built entirely on top of Docker containers, either running on Docker itself or under their own `rkt` runtime.

In the first years after Docker's release, a set of competitors and service providers voiced concerns about Docker's proprietary image format. Containers on Linux did not have a standard image format, so Docker, Inc., created their own according to the needs of their business.

Service providers and commercial vendors were particularly reluctant to build platforms subject to the whim of a company with somewhat overlapping interests to their own. Docker as a company faced some public challenges in that period as a result. In order to support wider adoption, Docker, Inc., then helped sponsor the Open Container Initiative (OCI) in June 2015. The first full specification from that effort was released in July 2017, and was based in large part on version 2 of the Docker image format. The OCI is working on the certification process and it will soon be possible for implementations to be OCI certified. In the meantime, there are at least four runtimes claiming to implement the spec: `runc`, which is part of Docker; `railcar`, an alternate implementation by Oracle; Kata Containers (formerly Clear Containers) from Intel, Hyper, and the OpenStack Foundation, which run a mix of containers and virtual machines; and finally, the gVisor runtime from Google, which is implemented entirely in user space. This set is likely to grow, with plans for CoreOS's `rkt` to become OCI compliant, and others are likely to follow.

The space around deploying containers and orchestrating entire systems of containers continues to expand, too. Many of these are open source and available both on premise and as cloud or SaaS offerings from various providers, either in their own clouds or yours. Given the amount of investment pouring into the Docker space, it's likely that Docker will continue to cement itself further into the fabric of the modern internet.

Architecture

Docker is a powerful technology, and that often means something that comes with a high level of complexity. And, under the hood, Docker is fairly complex; however, its fundamental user-facing structure is indeed a simple client/server model. There are several pieces sitting behind the Docker API, including `containerd` and `runc`, but the basic system interaction is a client talking over an API to a server. Underneath this simple exterior, Docker heavily leverages kernel mechanisms such as iptables, virtual bridging, cgroups, namespaces, and various filesystem drivers. We'll talk about some of these in Chapter 11. For now, we'll go over how the client and server work and give a brief introduction to the network layer that sits underneath a Docker container.

Client/Server Model

It's easiest to think of Docker as consisting of two parts: the client and the server/daemon (see Figure 2-3). Optionally there is a third component called the registry, which stores Docker images and their metadata. The server does the ongoing work of building, running, and managing your containers, and you use the client to tell the server what to do. The Docker daemon can run on any number of servers in the infrastructure, and a single client can address any number of servers. Clients drive all of the communication, but Docker servers can talk directly to image registries when told to do so by the client. Clients are responsible for telling servers what to do, and servers focus on hosting containerized applications.

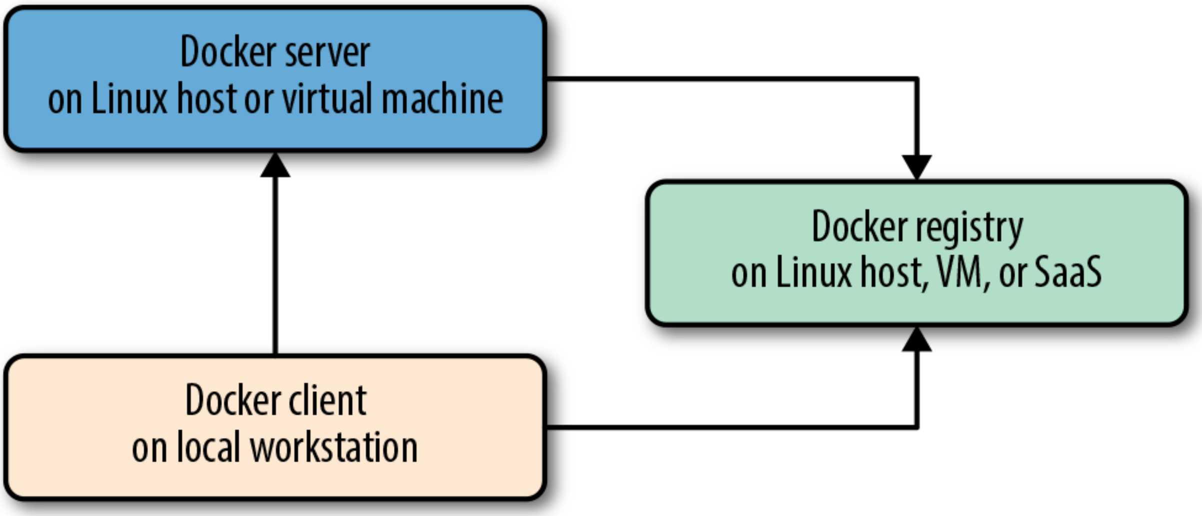


Figure 2-3. Docker client/server model

Docker is a little different in structure from some other client/server software. It has a `docker` client and a `dockerd` server, but rather than being entirely monolithic, the server then orchestrates a few other components behind the scene on behalf of the client, including `docker-proxy`, `runc`, `containerd`, and sometimes `docker-init`. Docker cleanly hides any complexity behind the simple server API, though, so you can just think of it as a client and server for most purposes. Each Docker host will normally have one Docker server running that can manage a number of containers. You can then use the `docker` command-line tool client to talk to the server, either from the server itself or, if properly secured, from a remote client. We’ll talk more about that shortly.

Network Ports and Unix Sockets

The `docker` command-line tool and `dockerd` daemon talk to each other over network sockets. Docker, Inc., has registered two ports with IANA for use by the Docker daemon and client: TCP ports 2375 for unencrypted traffic and 2376 for encrypted SSL connections. Using a different port is easily configurable for scenarios where you need to use different settings. The default setting for the Docker installer is to use only a Unix socket to make sure the system defaults to the most secure installation, but this is also easily configurable. The Unix socket can be located in different paths on different operating systems, so you should check where yours is located (often */var/run/docker.sock*). If you have strong preferences for a different location, you can usually specify this at install time or simply change the server configuration afterward and restart the daemon. If you don’t, then the defaults will probably work for you. As with most software, following the defaults will save you a lot of trouble if you don’t need to change them.

Robust Tooling

Among the many things that have led to Docker's growing adoption is its simple and powerful tooling. Since its initial release, its capabilities have been expanding ever wider, thanks to efforts from the Docker community at large. The tooling that Docker ships with supports building Docker images, basic deployment to individual Docker daemons, a distributed mode called Swarm mode, and all the functionality needed to actually manage a remote Docker server. Beyond the included Swarm mode, community efforts have focused on managing whole fleets (or clusters) of Docker servers and scheduling and orchestrating container deployments.

> **NOTE**
>
> One point of confusion for people who search for Docker Swarm information on the internet is that there are two different things sharing that name. There was an older, now deprecated project that was a standalone application, called "Docker Swarm," and there is a newer, built-in Swarm that we refer to here as "Swarm mode." You'll need to carefully look at the search results to identify the relevant ones. Over time this conflict will hopefully become moot as the older product fades into history.

Docker has also launched its own orchestration toolset, including Compose, Machine, and Swarm, which creates a cohesive deployment story for developers. Docker's offerings in the production orchestration space have been largely over-shadowed by Google's Kubernetes and the Apache Mesos project in current deployments. But Docker's orchestration tools remain useful, with Compose being particularly handy for local development.

Because Docker provides both a command-line tool and a remote web API, it is easy to add further tooling in any language. The command-line tool lends itself well to shell scripting, and a lot of power can easily be leveraged with simple wrappers around it. The tool also provides a huge amount of functionality itself, all targeted at an individual Docker daemon.

Docker Command-Line Tool

The command-line tool `docker` is the main interface that most people will have with Docker. This is a Go program that compiles and runs on all common architectures and operating systems. The command-line tool is available as part of the main Docker distribution on various platforms and also compiles directly from the Go source. Some of the things you can do with the Docker command-line tool include, but are not limited to:

- Build a container image.
- Pull images from a registry to a Docker daemon or push them up to a registry from the Docker daemon.
- Start a container on a Docker server either in the foreground or background.
- Retrieve the Docker logs from a remote server.
- Start a command-line shell inside a running container on a remote server.
- Monitor statistics about your container.
- Get a process listing from your container.

You can probably see how these can be composed into a workflow for building, deploying, and observing applications. But the Docker command-line tool is not the only way to interact with Docker, and it's not necessarily the most powerful.

Docker Engine API

Like many other pieces of modern software, the Docker daemon has a remote web application programming interface (API). This is in fact what the Docker command-line tool uses to communicate with the daemon. But because the API is documented and public, it's quite common for external tooling to use the API directly. This enables all manners of tooling, from mapping deployed Docker containers to servers, to automated deployments, to distributed schedulers. While it's very likely that beginners will not initially want to talk directly to the Docker API, it's a great tool to have available. As your organization embraces Docker over time, it's likely that you will increasingly find the API to be a good integration point for this tooling.

Extensive documentation for the API is on the Docker site. As the ecosystem has matured, robust implementations of Docker API libraries have emerged for all popular languages. Docker maintains SDKs for Python and Go, but the best libraries remain those maintained by third parties. We've used the much more extensive third-party Go and Ruby libraries, for example, and have found them to be both robust and rapidly updated as new versions of Docker are released.

Most of the things you can do with the Docker command-line tooling are supported relatively easily via the API. Two notable exceptions are the endpoints that require streaming or terminal access: running remote shells or executing the container in interactive mode. In these cases, it's often easier to use one of these solid client libraries or the command-line tool.

Container Networking

Even though Docker containers are largely made up of processes running on the host system itself, they usually behave quite differently from other processes at the network layer. Docker initially supported a single networking model, but now supports a robust assortment of configurations that handle most application requirements. Most people run their containers in the default configuration, called *bridge mode*. So let's take a look at how it works.

To understand bridge mode, it's easiest to think of each of your Docker containers as behaving like a host on a private network. The Docker server acts as a virtual bridge and the containers are clients behind it. A bridge is just a network device that repeats traffic from one side to another. So you can think of it like a mini–virtual network with each container acting like a host attached to that network.

The actual implementation, as shown in Figure 2-4, is that each container has its own virtual Ethernet interface connected to the Docker bridge and its own IP address allocated to the virtual interface. Docker lets you bind and expose individual or groups of ports on the host to the container so that the outside world can reach your container on those ports. That traffic passes over a proxy that is also part of the Docker daemon before getting to the container.

Docker allocates the private subnet from an unused RFC 1918 private subnet block. It detects which network blocks are unused on the host and allocates one of those to the virtual network. That is bridged to the host's local network through an interface on the server called `docker0`. This means that all of the containers are on a network together and can talk to each other directly. But to get to the host or the outside world, they go over the `docker0` virtual bridge interface. As we mentioned, inbound traffic goes over the proxy. This proxy is fairly high performance but can be limiting if you run high-throughput applications in containers.

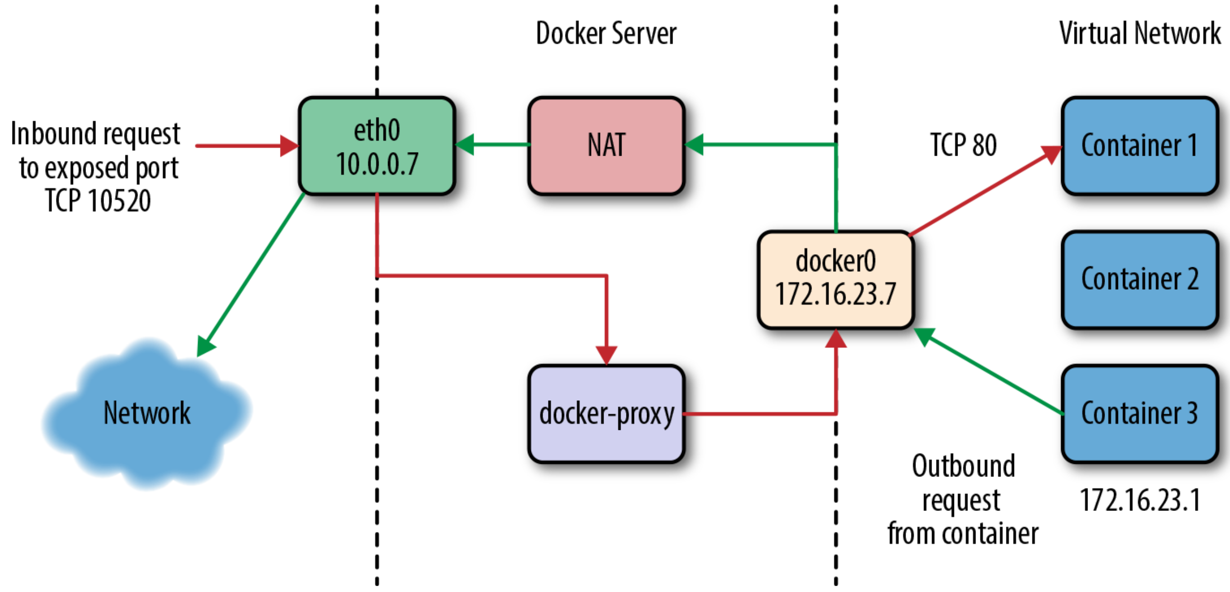


Figure 2-4. The network on a typical Docker server

There is a dizzying array of ways in which you can configure Docker's network layer, from allocating your own network blocks to configuring your own custom bridge interface. People often run with the default mechanisms, but there are times when something more complex or specific to your application is required. You can find much more detail about Docker networking in the documentation, and we will cover more details in the Chapter 11.

> **NOTE**
>
> When developing your Docker workflow, you should definitely get started with the default networking approach. You might later find that you don't want or need this default virtual network. Networking is configurable per container, and you can switch off the whole virtual network layer entirely for a container using the `--net=host` switch to `docker run`. When running in that mode, Docker containers use the host's own network devices and address and no virtual interfaces or bridge are provisioned. Note that host networking has security implications you might need to consider. Other network topologies are possible and discussed in Chapter 11.

Getting the Most from Docker

Like most tools, Docker has a number of great use cases, and others that aren't so good. You can, for example, open a glass jar with a hammer. But that has its downsides. Understanding how to best use the tool, or even simply determining if it's the right tool, can get you on the correct path much more quickly.

To begin with, Docker's architecture aims it squarely at applications that are either stateless or where the state is externalized into data stores like databases or caches. Those are the easiest to containerize. Docker enforces some good development principles for this class of application, and we'll talk later about how that's powerful. But this means doing things like putting a database engine inside Docker is a bit like swimming against the current. It's not that you can't do it, or even that you shouldn't do it; it's just that this is not the most obvious use case for Docker, so if it's the one you start with, you may find yourself disappointed early on. Databases that run well in Docker are often now deployed this way, but this is not the simple path. Some good applications for beginning with Docker include web frontends, backend APIs, and short-running tasks like maintenance scripts that might normally be handled by cron.

If you focus first on building an understanding of running stateless or externalized-state applications inside containers, you will have a foundation on which to start considering other use cases. We strongly recommend starting with stateless applications and learning from that experience before tackling other use cases. Note that the community is working hard on how to better support stateful applications in Docker, and there are likely to be many developments in this area.

Containers Are Not Virtual Machines

A good way to start shaping your understanding of how to leverage Docker is to think of containers not as virtual machines but as very lightweight wrappers around a single Unix process. During actual implementation, that process might spawn others, but on the other hand, one statically compiled binary could be all that's inside your container (see "Outside Dependencies" for more information). Containers are also ephemeral: they may come and go much more readily than a traditional virtual machine.

Virtual machines are by design a stand-in for real hardware that you might throw in a rack and leave there for a few years. Because a real server is what they're abstracting, virtual machines are often long-lived in nature. Even in the cloud where companies often spin virtual machines up and down on demand, they usually have a running lifespan of days or more. On the other hand, a particular container might exist for months, or it may be created, run a task for a minute, and then be destroyed. All of that is OK, but it's a fundamentally different approach than the one virtual machines are typically used for.

To help drive this differentiation home, if you run Docker on a Mac or Windows system you are leveraging a Linux virtual machine to run `dockerd`, the Docker server. However, on Linux `dockerd` can be run natively and therefore there is no need for a virtual machine to be run anywhere on the system (see Figure 2-5).

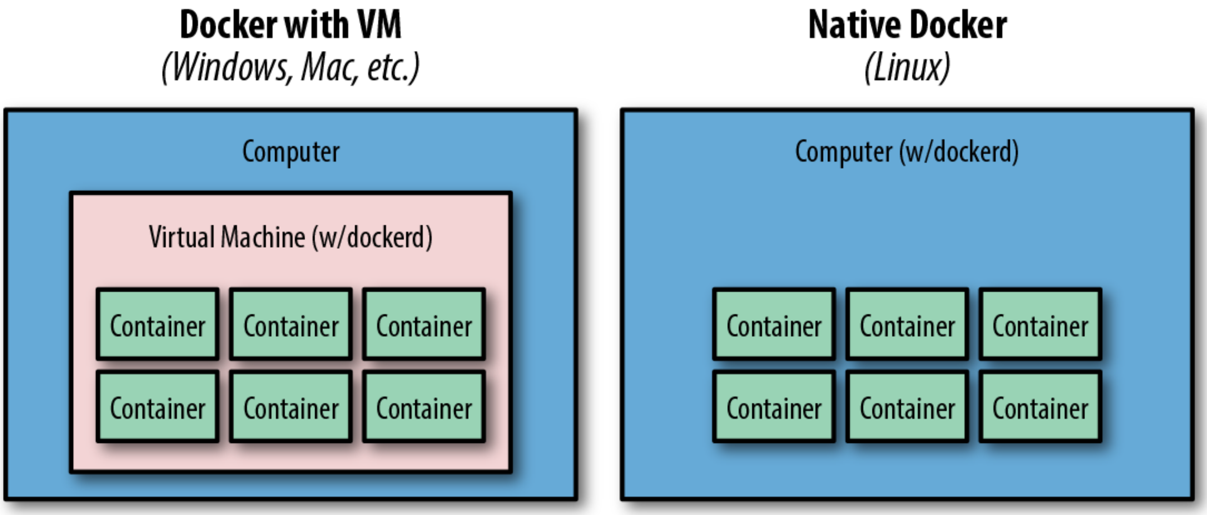


Figure 2-5. Typical Docker installations

Limited Isolation

Containers are isolated from each other, but that isolation is probably more limited than you might expect. While you can put limits on their resources, the default container configuration just has them all sharing CPU and memory on the host system, much as you would expect from colocated Unix processes. This means that unless you constrain them, containers can compete for resources on your production machines. That is sometimes what you want, but it impacts your design decisions. Limits on CPU and memory use are encouraged through Docker, but in most cases they are not the default like they would be from a virtual machine.

It's often the case that many containers share one or more common filesystem layers. That's one of the more powerful design decisions in Docker, but it also means that if you update a shared image, you may want to recreate a number of containers that still use the older one.

Containerized processes are just processes on the Docker server itself. They are running on the same exact instance of the Linux kernel as the host operating system. They even show up in the `ps` output on the Docker server. That is utterly different from a hypervisor, where the depth of process isolation usually includes running an entirely separate instance of the operating system kernel for each virtual machine.

This light containment can lead to the tempting option of exposing more resources from the host, such as shared filesystems to allow the storage of state. But you should think hard before further exposing resources from the host into the container unless they are used exclusively by the container. We'll talk about security of containers later, but generally you might consider helping to enforce isolation further by applying SELinux (Security-Enhanced Linux) or AppArmor policies rather than compromising the existing barriers.

WARNING

By default, many containers use UID 0 to launch processes. Because the container is *contained*, this seems safe, but in reality it isn't. Because everything is running on the same kernel, many types of security vulnerabilities or simple misconfiguration can give the container's `root` user unauthorized access to the host's system resources, files, and processes. Refer to "Security" for a discussion of how to mitigate this.

Containers Are Lightweight

We'll get more into the details of how this works later, but creating a container takes very little space. A quick test reveals that a newly created container from an existing image takes a whopping 12 kilobytes of disk space. That's pretty lightweight. On the other hand, a new virtual machine created from a golden image might require hundreds or thousands of megabytes. The new container is so small because it is just a reference to a layered filesystem image and some metadata about the configuration. There is no copy of the data allocated to the container. Containers are just processes on the existing system, so there may not be a need to copy any data for the exclusive use of the container.

The lightness of containers means that you can use them for situations where creating another virtual machine would be too heavyweight or where you need something to be truly ephemeral. You probably wouldn't, for instance, spin up an entire virtual machine to run a `curl` command to a website from a remote location, but you might spin up a new container for this purpose.

Toward an Immutable Infrastructure

By deploying most of your applications within containers, you can start simplifying your configuration management story by moving toward an immutable infrastructure, where components are replaced entirely rather than being changed in place. The idea of an immutable infrastructure has gained popularity in response to how difficult it is, in reality, to maintain a truly idempotent configuration management codebase. As your configuration management codebase grows, it can become as unwieldy and unmaintainable as large, monolithic legacy applications.

With Docker it is possible to deploy a very lightweight Docker server that needs almost no configuration management, or in many cases, none at all. You handle all of your application management simply by deploying and redeploying containers to the server. When the server needs an important update to something like the Docker daemon or the Linux kernel, you can simply bring up a new server with the changes, deploy your containers there, and then decommission or reinstall the old server.

Container-based Linux distributions like RedHat's CoreOS are designed around this principle. But rather than requiring you to decommission the instance, CoreOS can entirely update itself and switch to the updated OS. Your configuration and workload largely remain in your containers and you don't have to configure the OS very much at all.

Because of this clean separation between deployment and configuration of your servers, many container-based production systems are now using tools such as HashiCorp's Packer to build cloud virtual server images and then leveraging Docker to nearly or entirely avoid configuration management systems.

Stateless Applications

A good example of the kind of application that containerizes well is a web application that keeps its state in a database. Stateless applications are normally designed to immediately answer a single self-contained request, and have no need to track information between requests from one or more clients. You might also run something like ephemeral memcache instances in containers. If you think about your web application, though, it probably has local state that you rely on, like configuration files. That might not seem like a lot of state, but it means that you've limited the reusability of your container and made it more challenging to deploy into different environments, without maintaining configuration data in your codebase.

In many cases, the process of containerizing your application means that you move configuration state into environment variables that can be passed to your application from the container. Rather than baking the configuration into the container, you apply the configuration to the container at deployment time. This allows you to easily do things like use the same container to run in either production or staging environments. In most companies, those environments would require many different configuration settings, from the names of databases to the hostnames of other service dependencies.

With containers, you might also find that you are always decreasing the size of your containerized application as you optimize it down to the bare essentials required to run. We have found that thinking of anything that you need to run in a distributed way as a container can lead to some interesting design decisions. If, for example, you have a service that collects some data, processes it, and returns the result, you might configure containers on many servers to run the job and then aggregate the response on another container.

Externalizing State

If Docker works best for stateless applications, how do you best store state when you need to? Configuration is best passed by environment variables, for example. Docker supports environment variables natively, and they are stored in the metadata that makes up a container configuration. This means that restarting the container will ensure that the same configuration is passed to your application each time. It also makes the configuration of the container easily observable while it's running, which can make debugging a lot easier.

Databases are often where scaled applications store state, and nothing in Docker interferes with doing that for containerized applications. Applications that need to store files, however, face some challenges. Storing things to the container's filesystem will not perform well, will be extremely limited by space, and will not preserve state across a container lifecycle. If you redeploy a stateful service without utilizing storage external to the container, you will lose all of that state. Applications that need to store filesystem state should be considered carefully before you put them into Docker. If you decide that you can benefit from Docker in these cases, it's best to design a solution where the state can be stored in a centralized location that could be accessed regardless of which host a container runs on. In certain cases, this might mean using a service like Amazon S3, EBS volumes, HDFS, OpenStack Swift, a local block store, or even mounting EBS volumes or iSCSI disks inside the container. Docker volume plug-ins provide some options here and are briefly discussed in Chapter 11.

TIP

Although it is possible to externalize state on an attached filesystem, it is not generally encouraged by the community, and should be considered an advanced use case. It is strongly recommended that you start with applications that don't need persistent state. There are multiple reasons why this is typically discouraged, but in almost all cases it is because it introduces dependencies between the container and the host that interfere with using Docker as a truly dynamic, horizontally scalable application delivery service. If your container relies on an attached filesystem, it can only be deployed to the system that contains this filesystem. Remote volumes that can be dynamically attached are a good solution, but also an advanced use case.

The Docker Workflow

Like many tools, Docker strongly encourages a particular workflow. It's a very enabling workflow that maps well to how many companies are organized, but it's probably a little different than what you or your team are doing now. Having adapted our own organizations' workflow to the Docker approach, we can confidently say that this change is a benefit that touches many teams in the organization. If the workflow is implemented well, it can help realize the promise of reduced communication overhead between teams.

Revision Control

The first thing that Docker gives you out of the box is two forms of revision control. One is used to track the filesystem layers that comprise images, and the other is a tagging systems for images.

FILESYSTEM LAYERS

Docker containers are made up of stacked filesystem layers, each identified by a unique hash, where each new set of changes made during the build process is laid on top of the previous changes. That’s great, because it means that when you do a new build, you only have to rebuild the layers that follow the change you’re deploying. This saves time and bandwidth because containers are shipped around as layers and you don’t have to ship layers that a server already has stored. If you’ve done deployments with many classic deployment tools, you know that you can end up shipping hundreds of megabytes of the same data to a server over and over at each deployment. That’s slow, and worse, you can’t really be sure exactly what changed between deployments. Because of the layering effect, and because Docker containers include all of the application dependencies, with Docker you can be quite sure where the changes happened.

To simplify this a bit, remember that a Docker image contains everything required to run your application. If you change one line of code, you certainly don’t want to waste time rebuilding every dependency your code requires into a new image. Instead, Docker will use as many base layers as it can so that only the layers affected by the code change are rebuilt.

IMAGE TAGS

The second kind of revision control offered by Docker is one that makes it easy to answer an important question: what was the previous version of the application that was deployed? That’s not always easy to answer. There are a lot of solutions for non-Dockerized applications, from Git tags for each release, to deployment logs, to tagged builds for deployment, and many more. If you’re coordinating your deployment with Capistrano, for example, it will handle this for you by keeping a set number of previous releases on the server and then using symlinks to make one of them the current release.

But what you find in any scaled production environment is that each application has a unique way of handling deployment revisions. Many of them do the same thing, but some may be different. Worse, in heterogeneous language environments, the deployment tools are often entirely different between applications and very little is shared. So the question of “What was the previous version?” can have many answers depending on whom you ask and about which application. Docker has a built-in mechanism for handling this: it provides image tagging at deployment time. You can leave multiple revisions of your application on the server and just tag them at release. This is not rocket science, and it’s not functionality that is hard to find in other deployment tooling, as we mention. But it can easily be made standard across all of your applications, and everyone can have the same expectations about how things will be tagged for all applications. This makes communication easier between teams and it makes tooling much simpler because there is one source of truth for build versions.

> **WARNING**
>
> In many examples on the internet and in this book, you will see people use the `latest` tag. This is useful when you’re getting started and when you’re writing examples, as it will always grab the most recent version of a image. But since this is a floating tag, it is a really bad idea to use `latest` in most production workflows, as your dependencies can get updated out from under you, and it is impossible to roll back to `latest` because the old version is no longer the one tagged `latest`. It also makes it hard to verify if the same image is running on different servers. The rule of thumb is: don’t use the `latest` tag in production. It’s not even a good idea to use the `latest` tag from upstream images, for the same reasons.

Building

Building applications is a black art in many organizations, where a few people know all the levers to pull and knobs to turn in order to spit out a well-formed, shippable artifact. Part of the heavy cost of getting a new application deployed is getting the build right. Docker doesn’t solve all the problems, but it does provide a standardized tool configuration and toolset for builds. That makes it a lot easier for people to learn to build your applications, and to get new builds up and running.

The Docker command-line tool contains a `build` flag that will consume a *Dockerfile* and produce a Docker image. Each command in a *Dockerfile* generates a new layer in the image, so it’s easy to reason about what the build is going to do by looking at the *Dockerfile* itself. The great part of all of this standardization is that any engineer who has worked with a *Dockerfile* can dive right in and modify the build of any other application. Because the Docker image is a standardized artifact, all of the tooling behind the build will be the same regardless of the language being used, the OS distribution it’s based on, or the number of layers needed. The *Dockerfile* is checked into a revision control system, which also means tracking changes to the build is simplified. Modern multistage Docker builds also allow you to define the build environment separately from the final artifact image. This provides huge configurability for your build environment just like you'd have for a production container.

Most Docker builds are a single invocation of the `docker build` command and generate a single artifact, the container image. Because it's usually the case that most of the logic about the build is wholly contained in the *Dockerfile*, it's easy to create standard build jobs for any team to use in build systems like Jenkins. As a further standardization of the build process, many companies—eBay, for example—have standardized Docker containers to do the image builds from a *Dockerfile*. SaaS build offerings like Travis CI and Codeship also have first-class support for Docker builds.

Testing

While Docker itself does not include a built-in framework for testing, the way containers are built lends some advantages to testing with Docker containers.

Testing a production application can take many forms, from unit testing to full integration testing in a semi-live environment. Docker facilitates better testing by guaranteeing that the artifact that passed testing will be the one that ships to production. This can be guaranteed because we can either use the Docker SHA for the container, or a custom tag to make sure we're consistently shipping the same version of the application.

Since, by design, containers include all of their dependencies, tests run on containers are very reliable. If a unit test framework says tests were successful against a container image, you can be sure that you will not experience a problem with the versioning of an underlying library at deployment time, for example. That's not easy with most other technologies, and even Java WAR (Web application ARchive) files, for example, don't include testing of the application server itself. That same Java application deployed in a Docker container will generally also include an application server like Tomcat, and the whole stack can be smoke-tested before shipping to production.

A secondary benefit of shipping applications in Docker containers is that in places where there are multiple applications that talk to each other remotely via something like an API, developers of one application can easily develop against a version of the other service that is currently tagged for the environment they require, like production or staging. Developers on each team don't have to be experts in how the other service works or is deployed just to do development on their own application. If you expand this to a service-oriented architecture with innumerable microservices, Docker containers can be a real lifeline to developers or QA engineers who need to wade into the swamp of inter-microservice API calls.

A common practice in organizations that run Docker containers in production is for automated integration tests to pull down a versioned set of Docker containers for different services, matching the current deployed versions. The new service can then be integration-tested against the very same versions it will be deployed alongside. Doing this in a heterogeneous language environment would previously have required a lot of custom tooling, but it becomes reasonably simple to implement because of the standardization provided by Docker containers.

Packaging

Docker produces what for all intents and purposes is a single artifact from each build. No matter which language your application is written in or which distribution of Linux you run it on, you get a multilayered Docker image as the result of your build. And it is all built and handled by the Docker tooling. That's the shipping container metaphor that Docker is named for: a single, transportable unit that universal tooling can handle, regardless of what it contains. Like the container port or multimodal shipping hub, your Docker tooling will only ever have to deal with one kind of package: the Docker image. That's powerful, because it's a huge facilitator of tooling reuse between applications, and it means that someone else's off-the-shelf tools will work with your build images.

Applications that traditionally take a lot of custom configuration to deploy onto a new host or development system become totally portable with Docker. Once a container is built, it can easily be deployed on any system with a running Docker server.

Deploying

Deployments are handled by so many kinds of tools in different shops that it would be impossible to list them here. Some of these tools include shell scripting, Capistrano, Fabric, Ansible, and in-house custom tooling. In our experience with multi-team organizations, there are usually one or two people on each team who know the magical incantation to get deployments to work. When something goes wrong, the team is dependent on them to get it running again. As you probably expect by now, Docker makes most of that a nonissue. The built-in tooling supports a simple, one-line deployment strategy to get a build onto a host and up and running. The standard Docker client handles deploying only to a single host at a time, but there are a large array of tools available that make it easy to deploy into a cluster of Docker hosts. Because of the standardization Docker provides, your build can be deployed into any of these systems, with low complexity on the part of the development teams.

The Docker Ecosystem

Over the years, a wide community has formed around Docker, driven by both developers and system administrators. Like the DevOps movement, this has facilitated better tools by applying code to operations problems. Where there are gaps in the tooling provided by Docker, other companies and individuals have stepped up to the plate. Many of these tools are also open source. That means they are expandable and can be modified by any other company to fit their needs.

> **NOTE**
>
> Docker is a commercial company that has contributed much of the core Docker source code to the open source community. Companies are strongly encouraged to join the community and contribute back to the open source efforts. If you are looking for supported versions of the core Docker tools, you can find out more about its offerings on the Docker website.

ORCHESTRATION

The first important category of tools that adds functionality to the core Docker distribution contains orchestration and mass deployment tools. Mass deployment tools like New Relic's Centurion, Spotify's Helios, and the Ansible Docker tooling still work largely like traditional deployment tools but leverage the container as the distribution artifact. They take a fairly simple, easy-to-implement approach. You get a lot of the benefits of Docker without much complexity.

Fully automatic schedulers like Apache Mesos—when combined with a scheduler like Singularity, Aurora, Marathon, or Google's Kubernetes—are more powerful options that take nearly complete control of a pool of hosts on your behalf. Other commercial entries are widely available, such as HashiCorp's Nomad, CoreOS's Tectonic, Mesosphere's DCOS, and Rancher.[1] The ecosystems of both free and commercial options continue to grow rapidly.

ATOMIC HOSTS

One additional idea that you can leverage to enhance your Docker experience is atomic hosts. Traditionally, servers and virtual machines are systems that an organization will carefully assemble, configure, and maintain to provide a wide variety of functionality that supports a broad range of usage patterns. Updates must often be applied via nonatomic operations, and there are many ways in which host configurations can diverge and introduce unexpected behavior into the system. Most running systems are patched and updated in place in today's world. Conversely, in the world of software deployments, most people deploy an entire copy of their application, rather than trying to apply patches to a running system. Part of the appeal of containers is that they help make applications even more atomic than traditional deployment models.

What if you could extend that core container pattern all the way down into the operating system? Instead of relying on configuration management to try to update, patch, and coalesce changes to your OS components, what if you could simply pull down a new, thin OS image and reboot the server? And then if something breaks, easily roll back to the exact image you were previously using?

This is one of the core ideas behind Linux-based atomic host distributions, like Red Hat's CoreOS and Red Hat's original Project Atomic. Not only should you be able to easily tear down and redeploy your applications, but the same philosophy should apply for the whole software stack. This pattern helps provide very high levels of consistency and resilience to the whole stack.

Some of the typical characteristics of an atomic host are a minimal footprint, a design focused on supporting Linux containers and Docker, and atomic OS updates and rollbacks that can easily be controlled via multihost orchestration tools on both bare-metal and common virtualization platforms.

In Chapter 3, we will discuss how you can easily use atomic hosts in your development process. If you are also using atomic hosts as deployment targets, this process creates a previously unheard of amount of software stack symmetry between your development and production environments.

ADDITIONAL TOOLS

Docker is not just a standalone solution. It has a massive feature set, but there is always a case where someone needs more than it can deliver. There is a wide ecosystem of tools to either improve or augment Docker's functionality. Some good production tools leverage the Docker API, like Prometheus for monitoring, or Ansible or Spotify's Helios for orchestration. Others leverage Docker's plug-in architecture. Plug-ins are executable programs that conform to a specification for receiving and returning data to Docker. Some examples include Rancher's Convoy plug-in for managing persistent volumes on Amazon EBS volumes or over NFS mounts, Weaveworks' Weave Net network overlay, and Microsoft's Azure File Service plug-in.

There are many more good tools that either talk to the API or run as plug-ins. Many plug-ins have sprung up to make life with Docker easier on the various cloud providers. These really help with seamless integration between Docker and the cloud. As the community continues to innovate, the ecosystem continues to grow. There are new solutions and tools available in this space on an ongoing basis. If you find you are struggling with something in your environment, look to the ecosystem!

Wrap-Up

There you have it: a quick tour through Docker. We’ll return to this discussion later on with a slightly deeper dive into the architecture of Docker, more examples of how to use the community tooling, and an exploration of some of the thinking behind designing robust container platforms. But you’re probably itching to try it all out, so in the next chapter we’ll get Docker installed and running.

[1] Some of these commercial offerings have free editions of their platforms.

Chapter 3. Installing Docker

We’re now at the point where you hopefully understand roughly what Docker is and what it isn’t, and it’s time for some hands-on work. Let’s get Docker installed so we can work with it. The steps required to install Docker vary depending on the platform you use for development and the Linux distribution you use to host your applications in production.

In this chapter, we discuss the steps required to get a fully working Docker development environment set up on most modern desktop operating systems. First we’ll install the Docker client on your native development platform, and then we’ll get a Docker server running on Linux. Finally, we’ll test out the installation to make sure it works as expected.

Although the Docker client can run on Windows and macOS to control a Docker server, Linux containers can be built and launched only on a Linux system.[1] Therefore, non-Linux systems will require a virtual machine or remote server to host the Linux-based Docker server. Docker Community Edition, Docker Machine, and Vagrant, which are all discussed later in this chapter, provide some approaches to address this issue. It is also possible to run Windows containers natively on Windows systems, and we will specifically discuss this in “Windows Containers”, but most of the book’s focus will be on Linux containers.

> **NOTE**
>
> The Docker ecosystem is changing very rapidly as the technology evolves to become more robust and solve a broader range of problems. Some features discussed in this book and elsewhere may become deprecated. To see what has been tagged for deprecation and eventual removal, refer to the documentation.

Docker Client

The Docker client natively supports 64-bit versions of Linux, Windows, and macOS.

The majority of popular Linux distributions can trace their origins to either Debian or Red Hat. Debian systems utilize the deb package format and Advanced Package Tool (apt) to install most prepackaged software. On the other hand, Red Hat systems rely on rpm (Red Hat Package Manager) files and Yellowdog Updater, Modified (yum), or Dandified yum (dnf) to install similar software packages.

On macOS and Microsoft Windows, native GUI installers provide the easiest method to install and maintain prepackaged software. Homebrew for macOS and Chocolatey for Windows are also very popular options among technical users.

> **WARNING**
>
> We will be discussing a few approaches to installing Docker in this section. Make sure that you pick the first one in this list that best matches your needs. Installing more than one may cause you problems if you are not well versed in how to switch between them properly.
>
> Choose one of these: Docker Community Edition, docker-machine, OS package manager, or vagrant.

Linux

It is strongly recommended that you run Docker on a modern release of your preferred Linux distribution. It is possible to run Docker on some older releases, but stability may be a significant issue. Generally a 3.8 or later kernel is required, and we advise you to use the newest stable version of your chosen distribution. The following directions assume you are using a recent, stable release.

UBUNTU LINUX 17.04 (64-BIT)

> **NOTE**
>
> For other versions of Ubuntu, see the Docker Community Edition for Ubuntu.

Let’s take a look at the steps required to install Docker.

This first command will ensure that you aren’t running older versions of Docker. The package has been renamed a few times, so you’ll need to specify several possibilities here:

```
$ sudo apt-get remove docker docker-engine docker.io
```

NOTE

Docker ships in two flavors: Community Edition (CE) and Enterprise Edition (EE). Enterprise Edition is the commercially supported variant and has a slightly different set of features because they more carefully control the release schedule, much like Red Hat Enterprise and Fedora work in the Linux distribution arena. Here we’ll cover the Community Edition of Docker since that is largely what people will be using. Enterprise Edition users should find that almost all of what we’ll discuss in this book works fine for them as well, however, and should not be put off by the focus on Community Edition.

Next you will need to add the required software dependencies and *apt* respository for Docker Community Edition. This lets us fetch and install packages for Docker and validate that they are signed.

```
$ sudo apt-get update
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg |\
    sudo apt-key add -
$ sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) \
    stable"
```

Now that you have the repository set up, run the following commands to install Docker:

```
$ sudo apt-get update
$ sudo apt-get install docker-ce
```

Assuming you don’t get any error messages, you now have Docker installed!

FEDORA LINUX 26 (64-BIT)

> **NOTE**
>
> For other versions of Fedora, see the Docker Community Edition for Fedora.

Let’s take a look at the steps needed in order to install the correct Docker packages on your system.

This first command will ensure that you aren’t running older versions of Docker. As on Ubuntu systems, the package has been renamed a few times, so you’ll need to specify several possibilities here:

```
$ sudo dnf remove docker \
    docker-common \
    docker-selinux \
    docker-engine-selinux \
    docker-engine
```

Next you will need to add the required software dependencies and *dnf* respository for Docker Community Edition.

```
$ sudo dnf -y install dnf-plugins-core
$ sudo dnf config-manager \
    --add-repo \
    https://download.docker.com/linux/fedora/docker-ce.repo
```

Now you can install the current version of Docker Community Edition.

```
$ sudo dnf install docker-ce
```

macOS, Mac OS X

To install Docker Community Edition on macOS, you should use the GUI installer provided by Docker.

GUI INSTALLER

Download the latest Docker Community Edition for Mac installer and then double-click on the downloaded program icon. Follow all of the installer prompts until the installation is finished.

Docker Community Edition for Mac relies on the xhyve project to provide a native lightweight virtualization layer for the Linux server component, which macOS requires to launch Linux virtual machines that can build Docker images and run containers.

HOMEBREW INSTALLATION

You can also install the Docker CLI tools using the popular Homebrew package management system for macOS. If you take this approach, you should consider installing Docker Machine for creating and managing your Linux VM. We'll discuss that shortly in "Non-Linux VM-Based Server".

Microsoft Windows 10 Professional

Download the latest Docker Community Edition for Windows installer and then double-click on the downloaded program icon. Follow all of the installer prompts until the installation is finished.

> **TIP**
>
> If prompted, select support for "Linux Containers," as almost all the examples in the book expect this, and Linux containers are still the most common type of Docker container. You can easily change this at any time if you want or need to switch over to Windows containers in the future.

Docker Community Edition for Windows relies on Hyper-V to provide a native virtualization layer for the Linux server component, which Windows requires to launch Linux virtual machines that can build Docker images and run containers.

CHOCOLATEY INSTALLATION

You can also install the Docker CLI tools using the popular Chocolatey package management system for Windows. If you take this approach, you should consider installing Docker Machine for creating and managing your Linux VM. We'll discuss that shortly in "Non-Linux VM-Based Server".

NOTE

Installation directions for additional operating systems can be found at *store.docker.com*.

Docker Server

The Docker server is a separate binary from the client and is used to manage most of the work that Docker is typically used for. Next we will explore the most common ways to manage the Docker server.

NOTE

Docker Community Edition already sets up the server for you, so if you took that route, you do not need to do anything else besides ensure that Docker Community Edition is running.

systemd-Based Linux

Current Fedora and Ubuntu releases use `systemd` to manage processes on the system. Because you have already installed Docker, you can ensure that the server starts every time you boot the system by typing:

```
$ sudo systemctl enable docker
```

This tells `systemd` to enable the `docker` service and start it when the system boots or switches into the default run level. To start the Docker server, type the following:

```
$ sudo systemctl start docker
```

Non-Linux VM-Based Server

If you are using Microsoft Windows or macOS in your Docker workflow, you will need a virtual machine so that you can set up a Docker server for testing. Docker Community Edition is convenient because it sets up this VM for you using the native virtualization technology on these platforms. If you are running an older version of Windows or cannot use Docker

Community Edition for other reasons, you should investigate Docker Machine to help you create and manage your Docker server Linux VM.

In addition to Docker Machine, it is also possible to use other tools to set up the Docker server, depending on your preferences and needs:

- Vagrant
- Manually maintained virtual machines

DOCKER MACHINE

In early 2015, Docker announced the beta release of Docker Machine, a tool that makes it much easier to set up Docker hosts on bare-metal, cloud, and virtual machine platforms.

> **WARNING**
>
> You only need to follow the Docker Machine directions if you are on a Mac or Windows system and decided not to install Docker Community Edition, since that already installs a virtual machine–based server for you.

The easiest way to install Docker Machine is to visit the GitHub releases page and download the correct binary for your operating system and architecture. Currently, there are variants for 64-bit versions of Linux, Windows, and macOS, among a few other options.

> **NOTE**
>
> Docker Machine is an advanced tool. Determining which hypervisor is appropriate for your system and how to install it is impossible to cover in adequate detail here.

For these demonstrations, you will need to have a recent release of a hypervisor, like VirtualBox, installed on your system. VirtualBox is freely downloadable and easily installed on most systems, so for this section, the examples will all use a Unix-based system with VirtualBox. The following example shows how to do the install on macOS.

First, you need to download and install the `docker-machine` executable. This example downloads v0.13.0, but you can determine the current release on GitHub.

```
$ mkdir ~/bin
$ curl -L https://github.com/docker/machine/releases/\
download/v0.13.0/docker-machine-`uname -s`-`uname -m` \
  > ~/bin/docker-machine
$ export PATH=${PATH}:~/bin
$ chmod u+rx ~/bin/docker-machine
```

TIP

We’ve had to line-wrap the URL to fit the format of this book. If you have trouble running that in your shell as is, try removing the backslashes and joining it into one line without any spaces in the URL.

Once you have the `docker-machine` executable in your path, you can start to use it to set up Docker hosts. Here we’ve just put it temporarily into your path. If you want to keep running it in the future, add it to your `.profile` or `.bash_profile` file. Now that you can run the tool, the next thing that you need to do is create a virtual machine to host your Docker server.

TIP

Under some circumstances, this command might exit early and then recommend that you run `docker-machine regenerate-certs local`. If it does, go ahead and run that command and then try again.

You can do this using the `docker-machine create` command:

```
$ docker-machine create --driver virtualbox local

Creating CA: /home/skane/.docker/machine/certs/ca.pem
Creating client certificate: /home/skane/.docker/machine/certs/cert.pem
Running pre-create checks...
(local) Image cache directory does not exist, creating it at ...
(local) No default Boot2Docker ISO found locally, downloading the latest..
(local) Latest release for github.com/boot2docker/boot2docker is v17.09.1-
(local) Downloading /home/skane/.docker/machine/cache/boot2docker.iso from
(local) 0%..10%..20%..30%..40%..50%..60%..70%..80%..90%..100%
Creating machine...
(local) Copying /home/me/.docker/machine/cache/boot2docker.iso to ...
(local) Creating VirtualBox VM...
```

```
(local) Creating SSH key...
(local) Starting the VM...
(local) Check network to re-create if needed...
(local) Found a new host-only adapter: "vboxnet0"
(local) Waiting for an IP...
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 boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on t
  virtual machine, run: docker-machine env local
```

> **NOTE**
>
> If you already have a `docker-machine` VM created, you can simply start it with the command `docker-machine start local`.

This downloads a virtual machine image and then creates a VirtualBox VM that you can use as a Docker host. If you look at the output from the `create` command, you will see that it instructs you to run `docker-machine env local`.

This command will produce output that includes all of the environment variables that you must set in your shell to configure the Docker client to use your `docker-machine` virtual machine.

```
$ docker-machine env local
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://172.17.42.10:2376"
export DOCKER_CERT_PATH="/Users/me/.docker/machine/machines/local"
export DOCKER_MACHINE_NAME="local"
# Run this command to configure your shell:
# eval $(docker-machine env local)
```

The last line in the preceding output includes a command that will automatically set up those environment variables in your current shell. This command is different across platforms, but on Unix-based systems it will look like this:

```
$ eval $(docker-machine env local)
```

Because we’re evaluating the output of the command with `eval`, this command produces no output. If you’re curious, feel free to run the part of the command in parentheses to see what is being evaluated. Generally speaking, you will want to also put this whole `eval` block in your `.profile` or `.bash_profile` file as well, since it makes working with your Docker VM much easier going forward.

Now that you have that all set up, you can use the following command to confirm what machines you have running:

```
$ docker-machine ls
NAME  ACTIVE DRIVER     STATE   URL                       SWARM DOCKER
local -      virtualbox Running tcp://172.17.42.10:2376         v17.09.1-c
```

This tells you that you have one machine, named local, that is active and running.

Now you can pass commands to the new Docker machine by leveraging the regular `docker` command, since you have set the proper environment variables.

If you did not want to set the Docker environment variables from `docker-machine env local` for some reason, you could also use the `docker` and `docker-machine` commands in conjunction with one another to configure the `docker` CLI at runtime, like so: `docker $(docker-machine config local) ps`.

This command embeds the output from `docker-machine` into the middle of the `docker` command. If you run the `docker-machine config` command on its own, you can see what it is adding to the `docker` command:

```
$ docker-machine config local
--tlsverify
--tlscacert="/home/skane/.docker/machine/machines/local/ca.pem"
--tlscert="/home/skane/.docker/machine/machines/local/cert.pem"
--tlskey="/home/skane/.docker/machine/machines/local/key.pem"
-H=tcp://172.17.42.10:2376
```

Although you can see the Docker host’s IP address in the output, you can ask for it explicitly with the following command:

```
$ docker-machine ip local
```

```
   172.17.42.10
```

If you want to log in to the system, you can easily do this by running:

```
$ docker-machine ssh local
                        ##         .
                  ## ## ##        ==
               ## ## ## ## ##    ===
           /"""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
           \______ o           __/
             \    \         __/
              \____\_______/
 _                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 17.09.1-ce, build HEAD : e7de9ae - Fri Dec  8 19:41:36
Docker version 17.09.1-ce, build 19e2cf6
docker@local:~$ exit
```

To stop your Docker machine, you can run:

```
$ docker-machine stop local
Stopping "local"...
Machine "local" was stopped.
```

And then you can run this to restart it (you need it to be running):

```
$ docker-machine start local
```

Once a machine is down, you can also remove it to clean up after yourself entirely:

```
$ docker-machine rm local
```

If you want to explore the other options that `docker-machine` provides, you can simply run `docker-machine` without any other options to see the command help.

VAGRANT

If you need more flexibility with the way your Docker development environment is set up, you might want to consider using Vagrant instead of `docker-machine`. Like Docker Machine, Vagrant provides support for multiple hypervisors, but it is much more extensible, and can often be leveraged to mimic even the most complex environments.

A common use case for leveraging Vagrant during Docker development is to support testing on images that match your production environment. Vagrant supports everything from broad distributions like CentOS 7 and Ubuntu 16.04 to finely focused atomic host distributions like CoreOS and Project Atomic.

You can easily install Vagrant on most platforms by downloading a self-contained package from vagrantup.com. You will also need to have a hypervisor, like VirtualBox, installed on your system.

In the following example, you will create a CoreOS-based Docker host running the Docker daemon on the unencrypted port 2375. You could use your favorite distribution here instead, but CoreOS ships with working Docker out of the box and the image is quite small.

> **NOTE**
>
> In production, Docker should always be set up to use only encrypted remote connections. Although Docker Machine now uses encrypted communications by default, setting up Vagrant to do this in CoreOS is currently a bit too complicated for this installation example. You can find more information on how to set this up properly at coreos.com.

After Vagrant is installed, create a host directory with a name similar to `docker-host` and then move into that directory:

```
$ mkdir docker-host
$ cd docker-host
```

To install the *coreos-vagrant* files, you need the version control tool named Git. If you don't already have Git, you can install it via your package manager of choice, or download and install it from git-scm.com. When Git is installed, you can grab the *coreos-vagrant* files and then change into the new directory with the following commands:

```
$ git clone https://github.com/coreos/coreos-vagrant.git
$ cd coreos-vagrant
```

Inside the *coreos-vagrant* directory, you'll need to leverage the built-in `ignition` tool or the built-in `cloud-init` tool to add a `systemd` unit file that will enable the Docker daemon on TCP port 2375. The Vagrantfile will determine whether to use `ignition` or `cloud-init` based on your hypervisor, so it is easiest to simply set them both up.

TIP

You can read more about how `ignition` and `cloud-init` are used by `coreos-vagrant` in the README.

For `ignition`, create a file called *config.ign* that contains all of the following:

```
{
  "ignition": {
    "version": "2.0.0",
    "config": {}
  },
  "storage": {},
  "systemd": {
    "units": [
      {
        "name": "docker-tcp.socket",
        "enable": true,
        "contents": "[Unit]\nDescription=Docker Socket for the API\n..
."
      }
    ]
  },
  "networkd": {},
  "passwd": {}
}
```

NOTE

The complete *contents* key and value pair should be a single line in the file. The above file will not work correctly, as printed, because the contents line has been truncated.

You can get a complete copy of the above JSON file by running:

```
git clone https://github.com/bluewhalebook/\
docker-up-and-running-2nd-edition.git
```

For `cloud-init`, create a file called *user-data* that contains all of the following, including the very first line:

```
#cloud-config

coreos:
  units:
    - name: docker-tcp.socket
      command: start
      enable: true
      content: |
        [Unit]
        Description=Docker Socket for the API

        [Socket]
        ListenStream=2375
        Service=docker.service
        BindIPv6Only=both

        [Install]
        WantedBy=sockets.target
```

When you have saved both of these files, you can start up the Vagrant-based virtual machine by running:

WARNING

You will want to stop Docker Community Edition or any other Docker daemon you have running locally before running `vagrant up`, so that there are no potential port conflicts.

```
$ vagrant up
Bringing machine 'core-01' up with 'vmware_fusion' provider...
==> core-01: Cloning VMware VM: 'coreos-alpha'. This can take some time...
==> core-01: Checking if box 'coreos-alpha' is up to date...
==> core-01: Verifying vmnet devices are healthy...
==> core-01: Preparing network adapters...
==> core-01: Fixed port collision for 22 => 2222. Now on port 2202.
==> core-01: Starting the VMware VM...
==> core-01: Waiting for machine to boot. This may take a few minutes...
    core-01: SSH address: 172.17.42.10:22
    core-01: SSH username: core
    core-01: SSH auth method: private key
==> core-01: Machine booted and ready!
==> core-01: Forwarding ports...
```

```
    core-01: -- 22 => 2202
    core-01: SSH address: 172.17.42.10:22
    core-01: SSH username: core
    core-01: SSH auth method: private key
==> core-01: Setting hostname...
==> core-01: Configuring network adapters within the VM...
==> core-01: Running provisioner: file...
==> core-01: Running provisioner: shell...
    core-01: Running: inline script
==> core-01: Running provisioner: shell...
    core-01: Running: inline script
```

To set up your shell environment so that you can easily use your local Docker client to talk to the Docker daemon on your virtual machine, you can set the following variables:

```
$ unset DOCKER_TLS_VERIFY
$ unset DOCKER_CERT_PATH
$ export DOCKER_HOST=tcp://127.0.0.1:2375
```

WARNING

Keep an eye out for a message like this when running `vagrant up`. If you see this, it means that Vagrant remapped the Docker port to a new port.

```
... Fixed port collision for 2375 ... Now on port 2202
```

This might happen if you have Docker Community Edition running, among other reasons. In the previous case, you can deal with this by changing the `export` command to this:

```
export DOCKER_HOST=tcp://127.0.0.1:2202
```

If everything is running properly, you should now be able to run the following to connect to the Docker daemon:

```
$ docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 17.11.0-ce
Storage Driver: overlay2
 Backing Filesystem: extfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk sy
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 992280e8e265f491f7a624ab82f3e238be086e49
runc version: 0351df1c5a66838d0c392b4ac4cf9450de844e2d
init version: v0.13.2 (expected: 949e6facb77383876aeff8a6944dde66b3089574)
Security Options:
 seccomp
  Profile: default
 selinux
Kernel Version: 4.14.4-coreos
Operating System: Container Linux by CoreOS 1618.0.0 (Ladybug)
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 996.3MiB
Name: core-01
ID: MFKI:CLTN:3KDF:JXF4:FV74:GZAM:VJD7:35CW:TS3J:LTYE:QBLW:X26A
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Experimental: false
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
```

To connect to a shell on the Vagrant-based virtual machine, you can run:

```
$ vagrant ssh
Last login: Sun Dec 17 20:23:34 UTC 2017 from 192.168.180.1 on ssh
Container Linux by CoreOS alpha (1618.0.0)
core@core-01 ~ $ exit
```

You can stop your Vagrant-based VM by running:

```
$ vagrant halt
==> core-01: Attempting graceful shutdown of VM...
```

NOTE

If you shut down Docker Community Edition earlier to test Vagrant but do not plan on using Vagrant for your Docker environment, then now would be a good time to start Docker Community Edition back up and run `unset DOCKER_HOST` in your shell so that you are no longer pointing at the Vagrant host.

Testing the Setup

You are now ready to test that everything is working. You should be able to run any one of the following commands on your local system to tell the Docker daemon to download the latest official container for that distribution and then launch it with a running Unix shell process.

This step is important to ensure that all the pieces are properly installed and communicating with each other as expected. It also shows off one of the features of Docker: we can run containers based on any distribution we like. In the next few steps we'll run Docker containers based on Ubuntu, Fedora, and Alpine Linux. You don't need to run them all to prove that this works; running one of them will suffice.

NOTE

If you want to run these commands on the server, be sure that you prepend each `docker` command with `sudo`. Alternatively, since most Docker installs create a `docker` group that can be used to manage who has access to the `dockerd` Unix socket, you can simply add your user to that group as well.

Ubuntu

```
$ docker run --rm -ti ubuntu:latest /bin/bash
```

Fedora

```
$ docker run --rm -ti fedora:latest /bin/bash
```

Alpine Linux

```
$ docker run --rm -ti alpine:latest /bin/sh
```

> **NOTE**
>
> `ubuntu:latest`, `fedora:latest`, and `alpine:latest` all represent a Docker image name followed by an image tag.

Exploring the Docker Server

Although the Docker server is often installed, enabled, and run automatically, it's useful to see that running the Docker daemon manually on a Linux system is as simple as typing something like this:

```
$ sudo dockerd -H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375
```

> **NOTE**
>
> If you are using Docker Community Edition on a Windows or Mac, you won't be able to easily interact with the `dockerd` executable, as it is intentionally hidden from the end user, but we'll show you a trick in just a moment.

This command starts the Docker daemon, creates and listens to a Unix domain socket (`-H unix:///var/run/docker.sock`), and binds to all system IP addresses using the default unencrypted traffic port for docker (`-H tcp://0.0.0.0:2375`). You're not likely to have to start the Docker server yourself, but that's what going on behind the scenes. On non-Linux systems, you need a Linux-based virtual machine to host the Docker server. Docker Community Edition sets up this virtual machine for you in the background.

NOTE

If you already have Docker running, executing the daemon again will fail because it can’t use the same network port twice.

In most cases, it is very easy to SSH into your new Docker server and look around, but the seamless experience of Docker Community Edition on a non-Linux system means it is very hard to even notice that Docker is still leveraging a local virtual machine to run the Docker daemon on because the server appears to be running on the host system itself. You aren’t talking to the server over a TCP socket in that case, so you can’t just SSH to it.

If you are curious, or ever have a need to access the underlying VM, you can do it, but it requires a little advanced knowledge. We will talk about the command `nsenter` in much more detail in “nsenter”, but for now, if you would like to see the virtual machine (or underlying host) you can run these commands:

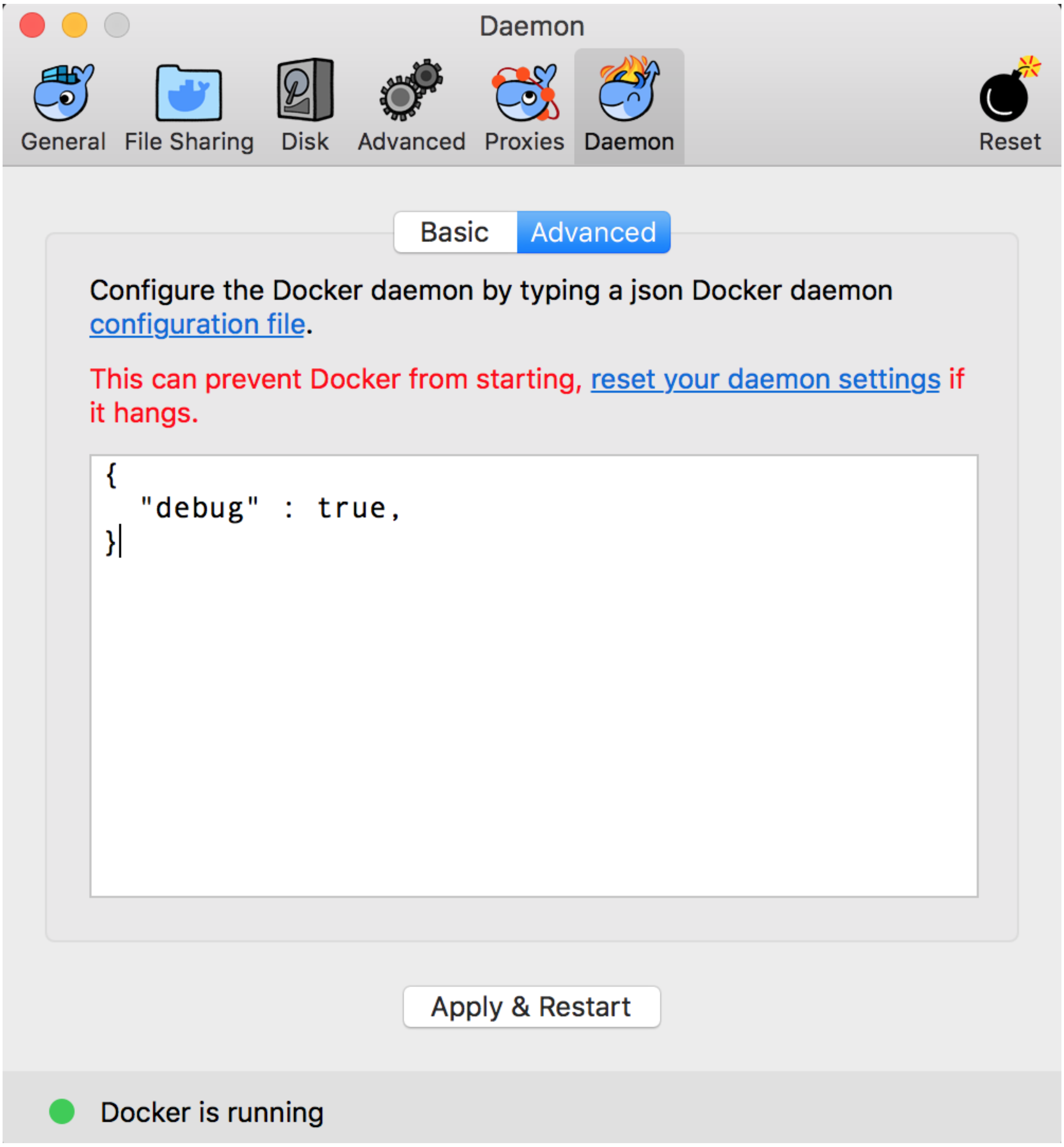
```
$ docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -n -i sl

/ # cat /etc/os-release
PRETTY_NAME="Docker for Mac"

/ # ps | grep dockerd
 1096 root       0:16 /usr/local/bin/dockerd -H unix:///var/run/docker.socl
                          --config-file /run/config/docker/daemon.json
                          --swarm-default-advertise-addr=eth0
                          --userland-proxy-path /usr/bin/vpnkit-expose-por
                          --storage-driver overlay2

/ # exit
```

This command uses a privileged Debian container that contains the `nsenter` command to manipulate the Linux kernel namepsaces so that we can navigate the filesystem of the underlying virtual machine or host.

WARNING

This container is privileged to allow us to navigate the underlying host, but you should not get into the habit of using privileged containers when adding individual capabilities or system call privileges will suffice. We discuss this more in "Security".

The Docker daemon configuration is stored in */etc/docker/daemon.json*. Docker uses sane defaults for all its settings, so this file may be very small or even completely absent. If you are using Docker Community Edition, you can edit this file by clicking on the Docker icon and selecting Preferences… → Daemon → Advanced, as shown in Figure 3-1.



Configure the Docker daemon by typing a json Docker daemon configuration file.

This can prevent Docker from starting, reset your daemon settings if it hangs.

```
{
  "debug" : true,
}
```

Figure 3-1. Community Edition server configuration

Wrap-Up

Now that you have a running Docker setup, you can start to look at more than the basic mechanics of getting it installed. In the next chapter, you'll explore how to build and manage Docker images, which provide the basis for every container you will ever launch with Docker.

TIP

In the rest of the book, when you see `docker` on the command line, assume you will need to have the correct configuration in place either as environment variables or via the `-H` command-line flag to tell the `docker` client how to connect to the `dockerd` server process.

[1] There is actually a project that is working toward making it possible to build Linux containers on Windows. If you are interested, take a look at lcow.

Chapter 4. Working with Docker Images

Every Docker container is based on an image. Images are the underlying definition of what gets reconstituted into a running container, much like a virtual disk becomes a virtual machine when you start it up. Docker images provide the basis for everything that you will ever deploy and run with Docker. To launch a container, you must either download a public image or create your own. You can think of the image as the filsystem for the container. But under the covers, every Docker image consists of one or more filesystem layers that generally have a direct one-to-one mapping to each individual build step used to create that image.

Because images are layered, they put special demand on the Linux kernel, which must provide the drivers that Docker needs to run the storage backend. For image management, Docker relies heavily on this storage backend, which communicates with the underlying Linux filesystem to build and manage the multiple layers that combine into a single usable image. The primary storage backends that are supported include: AUFS, BTRFS, Device-mapper, and overlay2. Each storage backend provides a fast copy-on-write (CoW) system for image management. We discuss the specifics of various backends in Chapter 11. For now we'll use the default backend and explore how images work, since they make up the basis for almost everything else that you will do with Docker, including:

- Building images
- Uploading (pushing) images to an image registry
- Downloading (pulling) images from an image registry
- Creating and running containers from an image

Anatomy of a Dockerfile

To create a custom Docker image with the default tools, you will need to become familiar with the *Dockerfile*. This file describes all the steps that are required to create an image and would usually be contained within the root directory of the source code repository for your application.

A typical *Dockerfile* might look something like the one shown here, which creates a container for a Node.js-based application:

```
FROM node:0.10
```

```
LABEL "maintainer"="anna@example.com"
LABEL "rating"="Five Stars" "class"="First Class"

USER root

ENV AP /data/app
ENV SCPATH /etc/supervisor/conf.d

RUN apt-get -y update

# The daemons
RUN apt-get -y install supervisor
RUN mkdir -p /var/log/supervisor

# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/

WORKDIR $AP

RUN npm install

CMD ["supervisord", "-n"]
```

Dissecting this *Dockerfile* will provide some initial exposure to a number of the possible instructions for controlling how an image is assembled. Each line in a *Dockerfile* creates a new image layer that is stored by Docker. This layer contains all of the changes that are a result of that command being issued. This means that when you build new images, Docker will only need to build layers that deviate from previous builds: you can reuse all the layers that haven't changed.

Although you could build a Node instance from a plain, base Linux image, you can also explore the Docker Hub for official images for Node. The Node.js community maintains a series of Docker images and tags that allows you to quickly determine what versions are available. If you want to lock the image to a specific point release of Node, you could point it at something like `node:0.10.33`. The base image that follows will provide you with an Ubuntu Linux image running Node 0.10.x.

```
FROM node:0.10
```

Applying labels to images and containers allows you to add metadata via key/value pairs that can later be used to search for and identify Docker images and containers. You can see the labels applied to any image using the `docker inspect` command.

```
LABEL "maintainer"="anna@example.com"
LABEL "rating"="Five Stars" "class"="First Class"
```

NOTE

MAINTAINER is a deprecated field in the *Dockerfile* specification. It was designed to provided contact information for the *Dockerfile*'s author, but Docker now recommends simply using a label for this purpose.

By default, Docker runs all processes as `root` within the container, but you can use the `USER` instruction to change this:

```
USER root
```

CAUTION

Even though containers provide some isolation from the underlying operating system, they still run on the host kernel. Due to potential security risks, production containers should almost always be run under the context of a nonprivileged user.

The `ENV` instruction allows you to set shell variables that can be used by your running application for configuration and during the build process to simplify the *Dockerfile* and help keep it DRYer:[1]

```
ENV AP /data/app
ENV SCPATH /etc/supervisor/conf.d
```

In the following code, you'll use a collection of `RUN` instructions to start and create the required file structure that you need, and install some required software dependencies. You'll also start to use the build variables you defined in the previous section to save you a bit of work and help protect you from typos.

```
RUN apt-get -y update

# The daemons
RUN apt-get -y install supervisor
RUN mkdir -p /var/log/supervisor
```

WARNING

While we’re demonstrating it here for simplicity, it is not recommended that you run commands like `apt-get -y update` or `yum -y update` in your application’s *Dockerfile*. This is because it requires crawling the repository index each time you run a build, and means that your build is not guaranteed to be repeatable since package versions might change between builds. Instead, consider basing your application image on another image that already has these updates applied to it and where the versions are in a known state. It will be faster and more repeatable.

The `ADD` instruction is used to copy files from either the local filesystem or a remote URL into your image. Most often this will include your application code and any required support files. Because `ADD` actually copies the files into the image, you no longer need access to the local filesystem to access them once the image is built.

```
# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/
```

TIP

Remember that every instruction creates a new Docker image layer, so it often makes sense to combine a few logically grouped commands onto a single line. It is even possible to use the `ADD` instruction in combination with the `RUN` instruction to copy a complex script to your image and then execute that script with only two commands in the *Dockerfile*.

With the `WORKDIR` instruction, you change the working directory in the image for the remaining build instructions and the default process that launches with any resulting containers:

```
WORKDIR $AP

RUN npm install
```

CAUTION

The order of commands in a *Dockerfile* can have a very significant impact on ongoing build times. You should try to order commands so that things that change between every single build are closer to the bottom. This means that adding your code and similar steps should be held off until the end. When you rebuild an image, every single layer after the first introduced change will need to be rebuilt.

And finally you end with the `CMD` instruction, which defines the command that launches the process that you want to run within the container:

```
CMD ["supervisord", "-n"]
```

NOTE

Though not a hard and fast rule, it is generally considered a best practice to try to run only a single process within a container. The core idea is that a container should provide a single function so that it remains easy to horizontally scale individual functions within your architecture. In the example, you are using `supervisord` as a process manager to help improve the resiliency of the node application within the container and ensure that it stays running. This can also be useful for troubleshooting your application during development, so that you can restart your service without restarting the whole container.

Building an Image

To build your first image, go ahead and clone a Git repo that contains an example application called *docker-node-hello*, as shown here:[2]

```
$ git clone https://github.com/spkane/docker-node-hello.git \
    --config core.autocrlf=input
Cloning into 'docker-node-hello'...
remote: Counting objects: 41, done.
remote: Total 41 (delta 0), reused 0 (delta 0), pack-reused 41
Unpacking objects: 100% (41/41), done.
$ cd docker-node-hello
```

NOTE

Git is frequently installed on Linux and macOS systems, but if you do not already have Git available, you can download a simple installer from git-scm.com.

The `--config core.autocrlf=input` option we use helps ensure that the line endings are not accidentally altered from the Linux standard that is expected.

This will download a working *Dockerfile* and related source code files into a directory called *docker-node-hello*. If you look at the contents while ignoring the Git repo directory, you should see the following:

```
$ tree -a -I .git
.
├── .dockerignore
├── .gitignore
├── Dockerfile
├── Makefile
├── README.md
├── Vagrantfile
├── index.js
├── package.json
└── supervisord
    └── conf.d
        ├── node.conf
        └── supervisord.conf
```

Let's review the most relevant files in the repo.

The *Dockerfile* should be exactly the same as the one you just reviewed.

The *.dockerignore* file allows you to define files and directories that you do not want uploaded to the Docker host when you are building the image. In this instance, the *.dockerignore* file contains the following line:

```
.git
```

This instructs `docker build` to exclude the *.git* directory, which contains the whole source code repository, including Git configuration data and every single change that you have ever made to your code. The rest of the files reflect the current state of your source code on the checked-out branch. You don't need the contents of the *.git* directory to build the Docker image, and since it can grow quite large over time, you don't want to waste time copying it every time you do a build. *package.json* defines the Node.js application and lists any dependencies that it relies on. *index.js* is the main source code for the application.

The *supervisord* directory contains the configuration files for `supervisord` that you will use to start and monitor the application.

NOTE

Using `supervisord` in this example to monitor the application is overkill, but it is intended to provide a bit of insight into some of the techniques you can use in a container to provide more control over your application and its running state.

As we discussed in Chapter 3, you will need to have your Docker server running and your client properly set up to communicate with it before you can build a Docker image. Assuming that this is all working, you should be able to initiate a new build by running the upcoming command, which will build and tag an image based on the files in the current directory.

Each step identified in the following output maps directly to a line in the *Dockerfile*, and each step creates a new image layer based on the previous step. The first build that you run will take a few minutes because you have to download the base node image. Subsequent builds should be much faster unless a newer node 0.10 base image has been released. Let’s run the build:

```
$ docker build -t example/docker-node-hello:latest .

Sending build context to Docker daemon  15.87kB
Step 1/14 : FROM node:0.10
 ---> 0f9a0e09fbda
Step 2/14 : LABEL "maintainer"="anna@example.com"
 ---> Running in 1d874dd2d5fa
Removing intermediate container 1d874dd2d5fa
 ---> 6113004a627c
Step 3/14 : LABEL "rating"="Five Stars" "class"="First Class"
 ---> Running in 99b5cf62f37a
Removing intermediate container 99b5cf62f37a
 ---> 9b674b79f9f8
Step 4/14 : USER root
 ---> Running in d8cc28917a0c
Removing intermediate container d8cc28917a0c
 ---> a73840c164af
Step 5/14 : ENV AP /data/app
 ---> Running in 879df989d503
Removing intermediate container 879df989d503
 ---> 022f5be79fb0
Step 6/14 : ENV SCPATH /etc/supervisor/conf.d
 ---> Running in 1386343bbcc5
Removing intermediate container 1386343bbcc5
 ---> 181728649ade
Step 7/14 : RUN apt-get -y update
 ---> Running in 460ddd2c7cd6
Get:1 http://security.debian.org jessie/updates
...
Reading package lists...
```

```
Removing intermediate container 460ddd2c7cd6
 ---> 9907f012a8e8
Step 8/14 : RUN apt-get -y install supervisor
 ---> Running in 03e9d7c7bd1e
Reading package lists...
...
Processing triggers for systemd (215-17+deb8u5) ...
Removing intermediate container 03e9d7c7bd1e
 ---> 4517cc222b0b
Step 9/14 : RUN mkdir -p /var/log/supervisor
 ---> Running in b0646ddd804a
Removing intermediate container b0646ddd804a
 ---> c45f74d9b9d7
Step 10/14 : ADD ./supervisord/conf.d/* $SCPATH/
 ---> 653eeae68288
Step 11/14 : ADD *.js* $AP/
 ---> 54ced594abe5
Step 12/14 : WORKDIR $AP
Removing intermediate container c065270d7e4b
 ---> d1b5e1d93364
Step 13/14 : RUN npm install
 ---> Running in 0c2dc15cab8d
npm WARN deprecated connect@2.7.9: connect 2.x series is deprecated
...
bytes@0.2.0, formidable@1.0.13)
Removing intermediate container 0c2dc15cab8d
 ---> 6b51bb6d8872
Step 14/14 : CMD ["supervisord", "-n"]
 ---> Running in a7be8f7416a1
Removing intermediate container a7be8f7416a1
 ---> 3a7881d5536e
Successfully built 3a7881d5536e
Successfully tagged example/docker-node-hello:latest
```

TIP

To improve the speed of builds, Docker will use a local cache when it thinks it is safe. This can sometimes lead to unexpected issues because it doesn't always notice that something changed in a lower layer. In the preceding output you will notice lines like `---> Running in 0c2dc15cab8d`. If instead you see `---> Using cache`, you know that Docker decided to use the cache. You can disable the cache for a build by using the `--no-cache` argument to the `docker build` command.

If you are building your Docker images on a system that is used for other simultaneous processes, you can limit the resources available to your builds by using many of the same cgroup methods that we will discuss in Chapter 5. You can find detailed documentation on the `docker build` arguments in the official documentation.

Troubleshooting Broken Builds

We normally expect builds to just work, especially when we've scripted them, but in the real world things go wrong. Let's spend a little bit of time discussing what you can do to troubleshoot a Docker build that is failing.

We need a patient for the next exercise, so let's create a failing build. To do that, edit the *Dockerfile* so that the line that reads:

```
RUN apt-get -y update
```

now reads:

```
RUN apt-get -y update-all
```

If you try to build the image now, you should get the following error:

```
$ docker build -t example/docker-node-hello:latest --no-cache .

Sending build context to Docker daemon  15.87kB
...
Step 6/14 : ENV SCPATH /etc/supervisor/conf.d
 ---> Running in 9c0a385269cf
Removing intermediate container 9c0a385269cf
 ---> 8a773166616c
Step 7/14 : RUN apt-get -y update-all
 ---> Running in cd57fc47503d
E: Command line option 'y' [from -y] is not known.
The command '/bin/sh -c apt-get -y update-all' returned a non-zero code: 100
```

This is a good example of a broken build, because the error is actually very misleading, and if we just assumed that `-y` was the problem, we would quickly discover that it was not. So, how can we troubleshoot this, especially if we are not developing on a Linux system? The real trick here is to remember that almost all Docker images are layered on top of other Docker images, and that you can start a container from any image. Although the meaning is not obvious on the surface, if you look at the output for step 6, you will see this:

```
Step 6/14 : ENV SCPATH /etc/supervisor/conf.d
 ---> Running in 9c0a385269cf
Removing intermediate container 9c0a385269cf
 ---> 8a773166616c
```

The first line that reads `Running in 9c0a385269cf` is telling you that the build process has started a new container, based on the image created in step 5. The next line, which reads `Removing intermediate container 9c0a385269cf`, is telling you that Docker is now removing the container, after having altered it based on the instruction in step 6. In this case, it was simply adding a default environment variable via `ENV SCPATH /etc/supervisor/conf.d`. The final line, which reads `--→ 8a773166616c`, is the one we really care about, because this is giving us the image ID for the image that was generated by step 6. You need this to troubleshoot the build, because it is the image from the last successful step in the build.

With this information, it is possible to run an interactive container so that you can try to determine why your build is not working properly. Remember that every container image is based on the image layers below it. One of the great benefits of that is that we can just run the lower layer as a container itself, using a shell to look around!

```
$ docker run --rm -ti 8a773166616c /bin/bash
root@464e8e35c784:/#
```

From inside the container, you can now run any commands that you might need to determine what is causing your build to fail and what you need to do to fix your *Dockerfile*.

```
root@464e8e35c784:/# apt-get -y update-all
E: Command line option 'y' [from -y] is not known.

root@464e8e35c784:/# apt-get update-all
E: Invalid operation update-all

root@464e8e35c784:/# apt-get --help
apt 1.0.9.8.3 for amd64 compiled on Mar 12 2016 13:31:17
Usage: apt-get [options] command
       apt-get [options] install|remove pkg1 [pkg2 ...]
       apt-get [options] source pkg1 [pkg2 ...]

apt-get is a simple command line interface for downloading and
installing packages. The most frequently used commands are update
and install.

Commands:
   update - Retrieve new lists of packages
...

Options:
...
  -y  Assume Yes to all queries and do not prompt
...

See the apt-get(8), sources.list(5) and apt.conf(5) manual
pages for more information and options.
                       This APT has Super Cow Powers.

root@464e8e35c784:/# apt-get -y update
Get:1 http://security.debian.org jessie/updates InRelease [63.1 kB]
...
Reading package lists... Done

root@464e8e35c784:/# exit
exit
```

Once the root cause has been determined, the *Dockerfile* can be fixed, so that `RUN apt-get -y update-all` now reads `RUN apt-get -y update`, and then rebuilding the image should result in success.

```
$ docker build -t example/docker-node-hello:latest .
Sending build context to Docker daemon  15.87kB
...
Successfully built 69f5e83bb86e
Successfully tagged example/docker-node-hello:latest
```

Running Your Image

Once you have successfully built the image, you can run it on your Docker host with the following command:

```
$ docker run -d -p 8080:8080 example/docker-node-hello:latest
```

The preceding command tells Docker to create a running container in the background from the image with the `example/docker-node-hello:latest` tag, and then map port 8080 in the container to port 8080 on the Docker host. If everything goes as expected, the new Node.js application should be running in a container on the host. You can verify this by running `docker ps`. To see the running application in action, you will need to open up a web browser and point it at port 8080 on the Docker host. You can usually determine the Docker host IP address by simply printing out the value of the `DOCKER_HOST` environment variable unless you are only running Docker locally, in which case 127.0.0.1 should work. Docker Machine users can also simply use `docker-machine ip`.

```
$ echo $DOCKER_HOST
tcp://127.0.0.1:2376
```

Get the IP address and enter something like *http://127.0.0.1:8080/* (or your remote Docker address if it's different than that) into your web browser address bar. You should see the following text:

```
Hello World. Wish you were here.
```

WARNING

If you are running Docker CE locally, you may not have `$DOCKER_HOST` set and can just assume localhost or 127.0.0.1.

Environment Variables

If you read the *index.js* file, you will notice that part of the file refers to the variable `$WHO`, which the application uses to determine who the application is going to say Hello to:

```
var DEFAULT_WHO = "World";
var WHO = process.env.WHO || DEFAULT_WHO;

app.get('/', function (req, res) {
  res.send('Hello ' + WHO + '. Wish you were here.\n');
});
```

Let's quickly cover how you can configure this application by passing in environment variables when you start it. First you need to stop the existing container using two commands. The first command will provide you with the container ID, which you will need to use in the second command:

```
$ docker ps
CONTAINER ID  IMAGE                             STATUS       ...
b7145e06083f  example/centos-node-hello:latest  Up 4 minutes ...
```

NOTE

You can format the output of `docker ps` by using a Go template so that you see only the information that you care about. In the preceding example you might decide to run something like `docker ps --format "table {{.ID}}\t{{.Image}}\t{{.Status}}"` to limit the output to the three fields you care about. Additionally, running `docker ps --quiet` with no format options will limit the output to only the container ID.

And then, using the container ID from the previous output, you can stop the running container by typing:

```
$ docker stop b7145e06083f
b7145e06083f
```

You can then restart the container by adding one argument to the previous `docker run` command:

```
$ docker run -d -p 8080:8080 -e WHO="Sean and Karl" \
    example/docker-node-hello:latest
```

If you reload your web browser, you should see that the text on the web page now reads:

```
Hello Sean and Karl. Wish you were here.
```

Custom Base Images

Base images are the lowest-level images that other Docker images will build upon. Most often, these are based on minimal installs of Linux distributions like Ubuntu, Fedora, CentOS, or Alpine Linux, but they can also be much smaller, containing a single statically compiled binary. For most people, using the official base images for their favorite distribution or tool is a great option.

However, there are times when it is more preferable to build your own base images rather than using an image created by someone else. One reason to do this would be to maintain a consistent OS image across all your deployment methods for hardware, VMs, and containers. Another would be to get the image size down substantially. There is no need to ship around an entire Ubuntu distribution, for example, if your application is a statically built C or Go application. You might find that you only need the tools you regularly use for debugging and some other shell commands and binaries. Making the effort to build such an image could pay off in better deployment times and easier application distribution.

A common middle ground between these two approaches is to build images using Alpine Linux, which is designed to be very small and is popular as a basis for Docker images. To keep the distribution size very small, Alpine Linux is based on the modern, lightweight musl standard library, instead of the more traditional GNU libc. In general, this is not a big issue, since many packages support musl, but it is something to be aware of. It has the largest impact on Java-based applications and DNS resolution. It’s widely used in production, however, because of its diminutive image size. It’s highly optimized for space, which is the reason that Alpine Linux ships with */bin/sh*, instead of */bin/bash*, by default. However, you can also install glibc and bash in Alpine Linux if you really need it, and this is often done in the case of JVM containers.

In the official Docker documentation, there is some good information about how you can build base images on the various Linux distributions.

Storing Images

Now that you have created a Docker image that you’re happy with, you’ll want to store it somewhere so that it can be easily accessed by any Docker host that you want to deploy it to. This is also the normal hand-off point between building images and storing them somewhere for future deployment. You don’t normally build the images on a production server and then run them. This process was described when we talked about handoff between teams for application deployment. Ordinarily, deployment is the process of pulling an image from a repository and running it on one or more Docker servers. There are a few ways you can go about storing your images into a central repository for easy retrieval.

Public Registries

Docker provides an image registry for public images that the community wants to share. These include official images for Linux distributions, ready-to-go WordPress containers, and much more.

If you have images that can be published to the internet, the best place for them is a public registry, like Docker Hub. However, there are other options. When the core Docker tools were first gaining popularity, Docker Hub did not exist. To fill this obvious void in the community, Quay.io was created. Since then, Quay.io was purchased by CoreOS[3] and has been used to create the CoreOS Enterprise Registry product, which we will discuss in a moment. Cloud vendors like Google also have their own registries, and a number of third parties have also joined the fray. Here we'll just talk about the two most popular.

Both Docker Hub and Quay.io provide centralized Docker image registries that can be accessed from anywhere on the internet, and provide a method to store private images in addition to public ones. Both have nice user interfaces and the ability to separate team access permissions and manage users. Both also offer reasonable commercial options for private SaaS hosting of your images, much in the same way that GitHub sells private registries on their systems. This is probably the right first step if you're getting serious about Docker but are not yet shipping enough code to need an internally hosted solution.

> **TIP**
>
> As you become comfortable with Docker Hub, you may also want to consider investigating Docker Cloud, which adds a significant feature set on top of the Docker Hub registry.

For companies that use Docker heavily, one of the biggest downsides to these registries is that they are not local to the network on which the application is being deployed. This means that every layer of every deployment might need to be dragged across the internet in order to deploy an application. Internet latencies have a very real impact on software deployments, and outages that affect these registries could have a very detrimental impact on a company's ability to deploy smoothly and on schedule. This is mitigated by good image design where you make thin layers that are easy to move around the internet.

Private Registries

The other option that many companies consider is to host some type of Docker image registry internally, which can interact with the Docker client to support pushing, pulling, and searching images. The version 1 registry, Docker Registry, has been replaced with the version 2 registry, called Docker Distribution.

Other strong contenders in the private registry space include the Docker Trusted Registry and the Quay Enterprise Registry. In addition to the basic Docker registry functionality, these products have solid GUI interfaces and many additional features, like image verification.

Authenticating to a Registry

Communicating with a registry that stores container images is a part of daily life with Docker. For many registries, this means you'll need to authenticate to gain access to images. But Docker also tries to make it easy to automate things so it can store your login information and use it on your behalf when you request things like pulling down a private image. By default, Docker assumes the registry will be Docker Hub, the public repository hosted by Docker, Inc.

CREATING A DOCKER HUB ACCOUNT

For these examples, you will create an account on Docker Hub. You don't need an account to use publicly shared images, but you will need one to upload your own public or private containers.

To create your account, use your web browser of choice to navigate to Docker Hub.

From there, you can log in via an existing account or create a new login based on your email address. When you create your account, Docker Hub sends a verification email to the address that you provided during signup. You should immediately log in to your email account and click the verification link inside the email to finish the validation process.

At this point, you have created a public registry to which you can upload new images. The Settings option under your profile picture will allow you to change your registry into a private one if that is what you need.

LOGGING IN TO A REGISTRY

Now let's log in to the Docker Hub registry using our account:

```
$ 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: <hub-username>
Password: <hub-password>
Login Succeeded
```

When you get `Login Succeeded` back from the server, you know you're ready to pull images from the registry. But what happened under the covers? It turns out that Docker has written a dotfile for us in our home directory to cache this information. The permissions are set to 0600 as a security precaution against other users reading your credentials. You can inspect the file with something like:

```
$ ls -la ~/.docker/config.json
-rw-------@ 1 ...  158 Dec 24 10:37 /Users/someuser/.docker/config.json
$ cat ~/.docker/config.json
```

```
{
    "auths": {
    "https://index.docker.io/v1/": {
      "auth":"cmVsaEXamPL3hElRmFCOUE=",
      "email":"someuser@example.com"
    }
  }
}
```

> **NOTE**
>
> Docker is constantly evolving and has added support for many OS native secret management systems like the macOS keychain. So, your *config.json* file might look significantly different than the example. There is also a set of credentials managers for different platforms that can make your life easier here.

Here you can see the *.docker/config.json* file, owned by someuser, and the stored credentials in JSON format. Note that this can support multiple registries at once. In this case, you just have one entry, for Docker Hub, but you could have more if you needed it. From now on, when the registry needs authentication, Docker will look in *.docker/config.json* to see if you have credentials stored for this hostname. If so, it will supply them. You will notice that one value is completely lacking here: a timestamp. These credentials are cached forever or until you tell Docker to remove them, whichever comes first.

As with logging in, you can also log out of a registry if you no longer want to cache the credentials:

```
$ docker logout
Removing login credentials for https://index.docker.io/v1/
$ cat ~/.docker/config.json
```

```
{
  "auths": {
  }
}
```

Here you have removed the cached credentials and they are no longer stored by Docker. Some versions of Docker may even remove this file if it is completely empty. If you were trying to log in to something other than the Docker Hub registry, you could supply the hostname on the command line:

```
$ docker login someregistry.example.com
```

This would then add another auth entry into our *.docker/config.json* file.

PUSHING IMAGES INTO A REPOSITORY

The first step required to push your image is to ensure that you are logged into the Docker repository you intend to use. For this example we will focus on Docker Hub, so ensure that you are logged into

Docker Hub with your personal credentials.

```
$ docker login
Username: <your_user>
Password: <your_password>
Email: <your_email@your_domain.com>
Login Succeeded
```

Once you are logged in, you can upload an image. Earlier you used the command `docker build -t example/docker-node-hello:latest .` to build the `docker-node-hello` image. The `example` portion of that command refers to a repository. When this is local, it can be anything that you want. However, when you are going to upload your image to a real repository, you need that to match the login.

You can easily edit the tags on the image that you already created by running the following command:

```
$ docker tag example/docker-node-hello:latest \
    ${<myuser>}/docker-node-hello:latest
```

If you need to rebuild the image with the new naming convention or simply want to give it a try, you can accomplish this by running the following command in the *docker-node-hello* working directory that was generated when you performed the Git checkout earlier in the chapter.

NOTE

For the following examples, you will need to replace `${<myuser>}` in all the examples with the user that you created in Docker Hub (or whatever repository you decided to use).

```
$ docker build -t ${<myuser>}/docker-node-hello:latest .
...
```

On the first build this will take a little time. If you rebuild the image, you may find that it is very fast. This is because most, if not all, of the layers already exist on your Docker server from the previous build. We can quickly verify that our image is indeed on the server by running `docker image ls ${<myuser>}/docker-node-hello`:

```
$ docker image ls ${<myuser>}/docker-node-hello
REPOSITORY                 TAG      IMAGE ID       CREATED              SIZE
myuser/docker-node-hello   latest   f683df27f02d   About an hour ago    649MB
```

TIP

It is possible to format the output of `docker image ls` to make it more concise by using the `--format` argument, like this: `docker images --format="table {{.ID }}\t {{.Repository }}"`.

At this point you can upload the image to the Docker repository by using the `docker push` command:

```
$ docker push ${<myuser>}/docker-node-hello:latest
The push refers to a repository [docker.io/myuser/docker-node-hello]
f93b977faea8: Pushed
...
fe4c16cbf7a4: Mounted from library/node
latest: digest: sha256:4ccb6995f9470553fcaa85674455b30370f092a110... size: 2840
```

If this image was uploaded to a public repository, anyone in the world can now easily download it by running the `docker pull` command.

TIP

If you uploaded the image to a private repository, then users must log into the private repository using the `docker login` command before they will be able to pull the image down to their local system.

```
$ docker pull ${<myuser>}/docker-node-hello:latest
latest: Pulling from myuser/docker-node-hello
Digest: sha256:4ccb6995f9470553fcaa85674455b30370f092a11029f6c7648dc45776d6af06
Status: Image is up to date for myuser/docker-node-hello:latest
```

Running a Private Registry

In keeping with the spirit of the open source community, Docker encourages the community to share Docker images via Docker Hub by default. There are times, however, when this is not a viable option due to commerical, legal, or reliability concerns.

In these cases, it makes sense to host an internal private registry. Setting up a basic registry is not difficult, but for production use, you should take the time to familiarize yourself with all the available configuration options for Docker Distribution.

For this example, we are going to create a very simple secure registry using SSL and HTTP basic auth.

First let’s create a few directories and files on our Docker server. If you are using a virtual machine or cloud instance to run your Docker server, then you will need to SSH to that server for the next few commands. If you are using Docker Community Edition, then you should be able to run these on your local system.

> **NOTE**
>
> Windows users: You may need to download additional tools, like `htppaswd`, or alter the non-Docker commands to accomplish the same tasks on you local system.

First let’s clone a Git repository that contains the basic files required to set up a simple, authenticated Docker registry.

```
$ git clone https://github.com/spkane/basic-registry --config core.autocrlf=inpu
Cloning into 'basic-registry'...
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 10 (delta 0), reused 10 (delta 0), pack-reused 0
Unpacking objects: 100% (10/10), done.
```

Once you have the files locally, you can change directories and examine the files that you have just downloaded.

```
$ cd basic-registry
$ ls
Dockerfile          config.yml.sample   registry.crt.sample
README.md           htpasswd.sample     registry.key.sample
```

The *Dockerfile* simply takes the upstream registry image from Docker Hub and copies some local configuration and support files into a new image.

For testing, you can use some of the included sample files, but *do not* use these in production.

If your Docker server is available via localhost (127.0.0.1), then you can use these files unmodified by simply copying each of them like this:

```
$ cp config.yml.sample config.yml
$ cp registry.key.sample registry.key
$ cp registry.crt.sample registry.crt
$ cp htpasswd.sample htpasswd
```

If, however, your Docker server is on a remote IP address, then you will need to do a little additional work.

First copy *config.yaml.sample* to *config.yaml*.

```
$ cp config.yml.sample config.yml
```

Then edit *config.yaml* and replace `127.0.0.1` with the IP address of your Docker server so that:

```
http:
  host: https://127.0.0.1:5000
```

becomes something like this:

```
http:
  host: https://172.17.42.10:5000
```

> **NOTE**
>
> It is easy to create a registry using a fully qualified domain name (FQDN), like `my-registry.example.com`, but for this example working with IP addresses is easier because no DNS is required.

Next you need to create an SSL keypair for your registry's IP address:

One way to do this is with the following OpenSSL command. Note that you will need to set the IP address in this portion of the command `/CN=172.17.42.10` to match your Docker server's IP address.

```
openssl req -x509 -nodes -sha256 -newkey rsa:4096 \
        -keyout registry.key -out registry.crt \
        -days 14 -subj '{/CN=172.17.42.10}'
```

Finally, you can either use the example `htpasswd` file by copying it:

```
$ cp htpasswd.sample htpasswd
```

or you can create your own username and password pair for authentication by using a command like the following, replacing `${<username>}` and `${<password>}` with your preferred values.

```
docker run --entrypoint htpasswd spkane/basic-registry:latest \
  -Bbn ${<username>} ${<password>} > htpasswd
```

If you look at the directory listing again, it should now look like this:

```
$ ls
Dockerfile          config.yml.sample   registry.crt        registry.key.sample
README.md           htpasswd            registry.crt.sample
config.yml          htpasswd.sample     registry.key
```

If any of these files are missing, review the previous steps, to ensure that you did not miss one, before moving on.

If everything looks correct, then you should be ready to build and run the registry.

```
$ docker build -t my-registry .
$ docker run -d -p 5000:5000 --name registry my-registry
$ docker logs registry
```

TIP

If you see errors like `docker: Error response from daemon: Conflict. The container name "/registry" is already in use`, then you need to either change the container name above, or remove the existing container with that name. You can remove the container by running `docker rm registry`.

TESTING THE PRIVATE REGISTRY

Now that the registry is running, you can test it. The very first thing that you need to do is authenticate against it. You will need to make sure that the IP address in the `docker login` matches the IP address of your Docker server that is running the registry.

NOTE

`myuser` is the default username, and `myuser-pw!` is the default password. If you generated your own `htpasswd`, then these will be whatever you choose.

```
$ docker login 127.0.0.1:5000
Username: <registry-username>
```

```
Password: <registry-password>
Login Succeeded
```

WARNING

This registry container is not using any external storage, which means that when you delete the running container, all your images will also be deleted. This is by design. In production you will want to use some type of redundant external storage, like an object store. If you want to keep your development registry images between containers, you could add something like `-v /tmp/registry-data:/var/lib/registry` to your `docker run` command to store the registry data on the Docker server.

Now, let's see if you can push the image you just built into your local private registry.

TIP

In all of these commands, ensure that you use the correct IP address for your registry.

```
$ docker tag my-registry 127.0.0.1:5000/my-registry
$ docker push 127.0.0.1:5000/my-registry
The push refers to a repository [127.0.0.1:5000/my-registry]
3198e7f79e44: Pushed
...
52a5560f4ca0: Pushed
latest: digest: sha256:b34f35f1810a8b8b64e0eac729a0aa875ca3ccbb4... size: 2194
```

You can then try to pull the same image from your repository.

```
$ docker pull 127.0.0.1:5000/my-registry
Using default tag: latest
latest: Pulling from my-registry
Digest: sha256:b34f35f1810a8b8b64e0eac729038f43be2f7d2a0aa875ca3ccbb493f14ea159
Status: Image is up to date for 127.0.0.1:5000/my-registry:latest
```

TIP

It’s worth keeping in mind that both Docker Hub and Docker Distribution expose an API endpoint that you can query for useful information. You can find out more information about the API via the official documentation.

If you have not encountered any errors, then you have a working registry for development and could build on this foundation to create a production registry. At this point you may want to stop the registry for the time being. You can easily accomplish this by running:

```
$ docker stop registry
```

TIP

As you become comfortable with Docker Distribution, you may also want to consider exploring VMware’s open source project, called Harbor, which extends the Docker Distribution with a lot of security- and reliability-focused features.

Advanced Building Techniques

After you have spent a little bit of time working with Docker, you will quickly notice that keeping your image sizes small and your build times fast can be very beneficial in decreasing the time required to build and deploy new versions of your software into production. In this section we will talk a bit about some of the considerations you should always keep in mind when designing your images, and a few techniques that can help achieve these goals.

Keeping Images Small

In most modern businesses, downloading a single 1 GB file from a remote location on the internet is not something that people often worry about. It is so easy to find software on the internet that people will often rely on simply redownloading it if they need it again, instead of keeping a local copy for the future. This may often be acceptable, when you truly need a single copy of this software on a single server, but it can quickly become a scaling problem, when you need the same software on 100+ nodes and you deploy new releases multiple times a days. Downloading these large files can quickly cause network congestion and slower deployment cycles that have a real impact on the production environment.

For convenience, a large number of Docker containers inherit from a base image that contains a minimal Linux distribution. Although this is an easy starting place, it isn’t actually required. Containers only need to contain the files that are required to run the application on the host kernel, and nothing else. The best way to explain this is to explore a very minimal container.

Go is a compiled programming language that by default will generate statically compiled binary files. For this example we are going to use a very small web application written in Go that can be found on GitHub.

Let’s go ahead and try out the application, so that you can see what it does. Run the following command and then open up a web browser and point it to your Docker host on port 8080 (e.g., *http://127.0.0.1:8080* for Docker Community Edition):

```
$ docker run -d -p 8080:8080 adejonge/helloworld
```

If all goes well, you should see the following message in your web browser: `Hello World from Go in minimal Docker container`. Now let’s take a look at what files this container comprises. It would be fair to assume that at a minimum it will include a working Linux environment and all the files required to compile Go programs, but you will soon see that this is not the case.

While the container is still running, execute the following command to determine what the container ID is. The following command returns the information for the last container that you created:

```
$ docker container ls -l
CONTAINER ID IMAGE               COMMAND       CREATED              ...
ddc3f61f311b adejonge/helloworld "/helloworld" 4 minutes ago        ...
```

You can then use the container ID that you obtained from running the previous command to export the files in the container into a tarball, which can be easily examined.

```
$ docker export ddc3f61f311b -o web-app.tar
```

Using the `tar` command, you can now examine the contents of your container at the time of the export.

```
tar -tvf web-app.tar
-rwxr-xr-x  0 0      0           0 Jan  7 15:54 .dockerenv
drwxr-xr-x  0 0      0           0 Jan  7 15:54 dev/
-rwxr-xr-x  0 0      0           0 Jan  7 15:54 dev/console
drwxr-xr-x  0 0      0           0 Jan  7 15:54 dev/pts/
drwxr-xr-x  0 0      0           0 Jan  7 15:54 dev/shm/
drwxr-xr-x  0 0      0           0 Jan  7 15:54 etc/
-rwxr-xr-x  0 0      0           0 Jan  7 15:54 etc/hostname
-rwxr-xr-x  0 0      0           0 Jan  7 15:54 etc/hosts
lrwxrwxrwx  0 0      0           0 Jan  7 15:54 etc/mtab -> /proc/mounts
-rwxr-xr-x  0 0      0           0 Jan  7 15:54 etc/resolv.conf
-rwxr-xr-x  0 0      0     3604416 Jul  2  2014 helloworld
drwxr-xr-x  0 0      0           0 Jan  7 15:54 proc/
drwxr-xr-x  0 0      0           0 Jan  7 15:54 sys/
```

The first thing you might notice here is that there are almost no files in this container, and almost all of them are zero bytes in length. All of the files that have a zero length are required to exist in every Linux container and are automatically bind-mounted from the host into the container when it is first created. All of these files, except for *.dockerenv*, are critical files that the kernel actually needs to do its job properly. The only file in this container that has any actual size and is related to our application is the statically compiled `helloworld` binary.

The take-away from this exercise is that your containers are only required to contain exactly what they need to run on the underlying kernel. Everything else is unnecessary. Because it is often useful for troubleshooting to have access to a working shell in your container, people will often compromise and build their images from a very lightweight Linux distribution like Alpine Linux.

To dive into this a little deeper, let's look at that same container again so that we can dig into the underlying filesystem and compare it with the popular `alpine` base image.

Although we could easily poke around in the `alpine` image by simply running `docker run -ti alpine:latest /bin/sh`, we cannot do this with the `adejonge/helloworld` image, because it does not contain a shell or SSH. This means that we can't use `ssh`, `nsenter`, or `docker exec` to examine it. Earlier, we took advantage of the `docker export` command to create a *.tar* file that contained a copy of all the files in the container, but this time around we are going to examine the container's filesystem by connecting directly to the Docker server, and then looking into the container's filesystem itself. To do this, we need to find out where the image files reside on the server's disk.

To determine where on the server our files are actually being stored, run `docker image inspect` on the `alpine:latest` image:

```
$ docker image inspect alpine:latest
```

```
[
    {
        "Id": "sha256:3fd...353",
        "RepoTags": [
            "alpine:latest"
        ],
        "RepoDigests": [
            "alpine@sha256:7b8...f8b"
        ],
...
        "GraphDriver": {
            "Data": {
                "MergedDir":
                "/var/lib/docker/overlay2/ea8...13a/merged",
                "UpperDir":
                "/var/lib/docker/overlay2/ea8...13a/diff",
                "WorkDir":
                "/var/lib/docker/overlay2/ea8...13a/work"
            },
            "Name": "overlay2"
...
]
```

And then on the `adejonge/helloworld:latest` image:

```
$ docker image inspect adejonge/helloworld:latest
```

```
[
    {
        "Id": "sha256:4fa...06d",
        "RepoTags": [
            "adejonge/helloworld:latest"
        ],
        "RepoDigests": [
            "adejonge/helloworld@sha256:46d...a1d"
        ],
...
        "GraphDriver": {
            "Data": {
                "LowerDir":
                "/var/lib/docker/overlay2/37a...84d/diff:
                /var/lib/docker/overlay2/28d...ef4/diff",
                "MergedDir":
                "/var/lib/docker/overlay2/fc9...c91/merged",
                "UpperDir":
                "/var/lib/docker/overlay2/fc9...c91/diff",
                "WorkDir":
                "/var/lib/docker/overlay2/fc9...c91/work"
            },
            "Name": "overlay2"
...
]
```

NOTE

In this particular example we are going to use Docker Community Edition running on macOS, but this general approach will work on most modern Docker servers. You'll just need to access your Docker server via whatever method is easiest.

Since we are using Docker Community Edition, we need to use our `nsenter` trick to enter the SSH-less virtual machine and explore the filesystem.

```
$ docker run -it --privileged --pid=host debian nsenter -t 1  -m -u -n -i sh

/ #
```

Inside the VM, we should now be able to explore the various directories listed in the `GraphDriver` section of the `docker image inspect` commands.

In this example, if we look at the first entry for the `alpine` image will see that it is labeled `MergedDir` and lists the folder */var/lib/docker/overlay2/ea86408b2b15d33ee27d78ff44f82104705286221f055ba1331b58673f4b313a/merge* If we list that directory we will actually get an error, but from listing the parent directory we quickly discover that we actually want to look at the *diff* directory.

```
/ # ls -lFa /var/lib/docker/overlay2/ea...3a/merged

ls: /var/lib/docker/overlay2/ea..3a/merged: No such file or directory

/ # ls -lF /var/lib/docker/overlay2/ea...3a/

total 8
drwxr-xr-x   18 root     root          4096 Mar 15 19:27 diff/
-rw-r--r--    1 root     root            26 Mar 15 19:27 link

/ # ls -lF /var/lib/docker/overlay2/ea...3a/diff

total 64
drwxr-xr-x    2 root     root          4096 Jan  9 19:37 bin/
drwxr-xr-x    2 root     root          4096 Jan  9 19:37 dev/
drwxr-xr-x   15 root     root          4096 Jan  9 19:37 etc/
drwxr-xr-x    2 root     root          4096 Jan  9 19:37 home/
drwxr-xr-x    5 root     root          4096 Jan  9 19:37 lib/
drwxr-xr-x    5 root     root          4096 Jan  9 19:37 media/
drwxr-xr-x    2 root     root          4096 Jan  9 19:37 mnt/
dr-xr-xr-x    2 root     root          4096 Jan  9 19:37 proc/
drwx------    2 root     root          4096 Jan  9 19:37 root/
drwxr-xr-x    2 root     root          4096 Jan  9 19:37 run/
drwxr-xr-x    2 root     root          4096 Jan  9 19:37 sbin/
drwxr-xr-x    2 root     root          4096 Jan  9 19:37 srv/
drwxr-xr-x    2 root     root          4096 Jan  9 19:37 sys/
drwxrwxrwt    2 root     root          4096 Jan  9 19:37 tmp/
drwxr-xr-x    7 root     root          4096 Jan  9 19:37 usr/
drwxr-xr-x   11 root     root          4096 Jan  9 19:37 var/

/ # du -sh  /var/lib/docker/overlay2/ea...3a/diff
4.5M    /var/lib/docker/overlay2/ea...3a/diff
```

Now `alpine` happens to be a very small base image, weighing in at only 4.5 MB, and it is actually ideal for building containers on top of. However, we can see that there is still a lot of stuff in this container before we have started to build anything from it.

Now, let’s take a look at the files in the `adejonge/helloworld` image. In this case, we want to look at the first directory from the `LowerDir` entry of the `docker image inspect` output, which you’ll notice also ends in a directory called *diff*.

```
/ # ls -lFh /var/lib/docker/overlay2/37...4d/diff
```

```
total 3520
-rwxr-xr-x    1 root     root         3.4M Jul  2  2014 helloworld*

/ # exit
```

You'll notice that there is only a single file in this directory and it is 3.4 MB. This helloworld binary is the only file shipped in this container and is smaller than the starting size of the `alpine` image before any application files have been added to it.

> **NOTE**
>
> It is actually possible to run the `helloworld` application right from that directory on your Docker server, because it does not require any other files. You really don't want to do this on anything but a development box, but it can help drive the point home about how useful these types of statically compiled applications can be.

MULTISTAGE BUILDS

There is a way you can constrain containers to an even smaller size in many cases: multistage builds. This is, in fact, how we recommend that you build most production containers. You don't have to worry as much about bringing in extra resources to build your application, and can still run a lean production container. Multistage containers also encourage doing builds inside of Docker, which is a great pattern for repeatability in your build system.

As the author of this helloworld application has written about, the release of multistage build support in Docker itself has made the process of creating small containers much easier than it used to be. In the past, to do the same thing that multistage delivers for nearly free, you were required to build one image that compiled your code, extract the resulting binary, and then build a second image without all the build dependencies that you would then inject that binary into. This was often difficult to set up and did not always work out-of-the-box with standard deployment pipelines.

Today, you can now achieve similar results using a *Dockerfile* as simple as this one:

```
# Build container
FROM golang:alpine as builder
RUN apk update && \
    apk add git && \
    CGO_ENABLED=0 go get -a -ldflags '-s' github.com/adriaandejonge/helloworld

# Production container
FROM scratch
COPY --from=builder /go/bin/helloworld /helloworld
EXPOSE 8080
CMD ["/helloworld"]
```

The first thing you’ll notice about this *Dockerfile* is that it looks a lot like two files that have been combined into one. Indeed this is the case, but there is more to it. The `FROM` command has been extended, so that you can name the image during the build phase. In this example, the first line, which reads `FROM golang as builder`, means that you want to base your build on the `golang` image and will be referring to this build image/stage as `builder`.

On the fourth line, you’ll see another `FROM` line, which was not allowed before the introduction of multistage builds. This `FROM` line uses a special image name, called `scratch`, that tells Docker to start from an empty image, which includes no additional files. The next line, which reads `COPY --from=builder /go/bin/helloworld /helloworld`, allows you to copy the binary that you built in the *builder* image directly into the current image. This will ensure that you end up with the smallest container possible.

Let’s try to build this and see what happens. First, create a directory where you can work and then, using your favorite text editor, paste the content from the preceding example into a file called *Dockerfile*.

```
$ mkdir /tmp/multi-build
$ cd /tmp/multi-build
$ vi Dockerfile
```

We can now start the multistage build.

```
$ docker build .
Sending build context to Docker daemon  2.048kB
Step 1/6 : FROM golang:alpine as builder
 ---> f421e93ece9c
Step 2/6 : RUN apk update &&     apk add git &&     CGO_ENABLED=0 go get -a \
               -ldflags '-s' github.com/adriaandejonge/helloworld
 ---> Running in 8472fecfb61e
...
OK: 25 MiB in 17 packages
 ---> 074c432f8d99
Removing intermediate container 8472fecfb61e
Step 3/6 : FROM scratch
 --->
Step 4/6 : COPY --from=builder /go/bin/helloworld /helloworld
 ---> 0d995eddb962
Step 5/6 : EXPOSE 8080
 ---> Running in 544d8feedcde
 ---> 0421d47b8884
Removing intermediate container 544d8feedcde
Step 6/6 : CMD /helloworld
 ---> Running in f5284d729d8a
 ---> 8970a3ac4276
Removing intermediate container f5284d729d8a
Successfully built 8970a3ac4276
```

You'll notice that the output looks like most other builds and still ends by reporting the successful creation of our final, very minimal image.

> **TIP**
>
> You are not limited to two stages, and in fact, none of the stages need to even be related to each other. They will be run in order. You could, for example, have a stage based on the public Go image that builds your underlying Go application to serve an API, and another stage based on the Angular container to build your frontend web UI. The final stage could then combine outputs from both. If compiling binaries with shared libs, you need to be careful about the underlying OS you are compiling on versus the one you will ship in the container. Alpine Linux is rapidly becoming the common ground here.

Layers Are Additive

Something that is not apparent until you dig much deeper into how images are built is that the filesystem layers that make up your images are strictly additive in nature. Although you can mask files in previous layers, you cannot delete those files. In practice this means that you cannot make your image smaller by simply deleting files that were generated in earlier steps.

The easiest way to explain this is by using some practical examples. In a new directory, create the following file, which will generate a image that launches the Apache web server running on Fedora Linux:

```
FROM fedora
RUN dnf install -y httpd
CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
```

and then build it like this:

```
$ docker build .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM fedora
 ---> 422dc563ca32
Step 2/3 : RUN dnf install -y httpd
 ---> Using cache
 ---> 44a66eb9f002
Step 3/3 : CMD /usr/sbin/httpd -DFOREGROUND
 ---> Running in 9888e6524355
 ---> 8d29ec43dc5a
Removing intermediate container 9888e6524355
Successfully built 8d29ec43dc5a
```

Let's go ahead and tag the resulting image so that you can easily refer to it in subsequent commands:

```
$ docker tag 8d29ec43dc5a size1
```

Now let’s take a look at our image with the `docker history` command. This command will give us some insight into the filesystem layers and build steps that our image uses.

```
$ docker history size1
IMAGE        CREATED       CREATED BY                                      SIZE
8d29ec43dc5a 5 minutes ago /bin/sh -c #(nop)  CMD ["/usr/sbin/httpd"...  0B
44a66eb9f002 6 minutes ago /bin/sh -c dnf install -y httpd                182MB
422dc563ca32 7 weeks ago   /bin/sh -c #(nop) ADD file:3b33f77d83b76f3... 252MB
<missing>    7 weeks ago   /bin/sh -c #(nop)  ENV DISTTAG=f27containe... 0B
<missing>    2 months ago  /bin/sh -c #(nop)  MAINTAINER [Adam Miller... 0B
```

You’ll notice that three of the layers added no size to our final image, but two of them added a great deal of size. The layer that is 252 MB makes sense, as this is the base Fedora image that includes a minimal Linux distribution; however, the 182 MB layer is surprising. The Apache web server shouldn’t be nearly that large, so what’s going on here, exactly?

If you have experience with package managers like `apk`, `apt`, `dnf`, or `yum`, then you may know that most of these tools rely heavily on a large cache that includes details about all the packages that are available for installation on the platform in question. This cache uses up a huge amount of space and is completely useless once you have installed the packages you need. The most obvious next step is to simply delete the cache. On Fedora systems, you could do this by editing your *Dockerfile* so that it looks like this:

```
FROM fedora
RUN dnf install -y httpd
RUN dnf clean all
CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
```

and then building, tagging, and examining the resulting image:

```
$ docker build .
Sending build context to Docker daemon  2.048kB
...
Successfully built 6ad08c11b3f5
$ docker tag 6ad08c11b3f5 size2
$ docker history size2
IMAGE        CREATED        CREATED BY                                     SIZE
6ad08c11b3f5 About a minute /bin/sh -c #(nop)  CMD ["/usr/sbin/httpd...  0B
29b7f0acbb91 About a minute /bin/sh -c dnf clean all                     1.84MB
44a66eb9f002 24 minutes ago /bin/sh -c dnf install -y httpd              182MB
422dc563ca32 7 weeks ago    /bin/sh -c #(nop) ADD file:3b33f77d83b76f... 252MB
<missing>    7 weeks ago    /bin/sh -c #(nop)  ENV DISTTAG=f27contain... 0B
<missing>    2 months ago   /bin/sh -c #(nop)  MAINTAINER [Adam Mille... 0B
```

If you look carefully at the output from the `docker history` command, you'll notice that you have created a new layer that adds 1.84 MB to the image, but you have not decreased the size of the problematic layer at all. What is happening exactly?

The important thing to understand is that image layers are strictly *additive* in nature. Once a layer is created, nothing can be removed from it. This means that you cannot make earlier layers in an image smaller by deleting files in subsequent layers. When you delete or edit files in subsequent layers, you're simply masking the older version with the modified or removed version in the new layer. This means that the only way you can make a layer smaller is by removing files before you save the layer.

The most common way to deal with this is by stringing commands together on a single *Dockerfile* line. You can do this very easily by taking advantage of the `&&` operator. This operator acts like a Boolean `AND` statement and basically translates into English as "and if the previous command ran successfully, run this command." In addition to this, you can also take advantage of the `/` operator, which is used to indicate that a command continues after the newline. This can help improve the readability of long commands.

With this knowledge in hand, you can rewrite the *Dockerfile* like this:

```
FROM fedora
RUN dnf install -y httpd && \
    dnf clean all
CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
```

Now you can rebuild the image and see how this change has impacted the size of the layer that includes the `http` daemon:

```
$ docker build .
Sending build context to Docker daemon  2.048kB
...
Successfully built b064ee5618c9
$ docker tag b064ee5618c9 size3
$ docker history size3
IMAGE         CREATED        CREATED BY                                     SIZE
b064ee5618c9  52 seconds ago /bin/sh -c #(nop)  CMD ["/usr/sbin/http...    0B
7eb37b2de3cf  53 seconds ago /bin/sh -c dnf install -y httpd &&      ...    17.7MB
422dc563ca32  2 months ago   /bin/sh -c #(nop) ADD file:3b33f77d83b7...    252MB
<missing>     2 months ago   /bin/sh -c #(nop)  ENV DISTTAG=f27conta...    0B
<missing>     2 months ago   /bin/sh -c #(nop)  MAINTAINER [Adam Mil...    0B
```

In the first two examples the layer in question was 182 MB in size, but now that you have removed many unnecessary files that were added to that layer, you are able to shrink the layer all the way down to 17.7 MB. This is a very large saving of space, especially when you consider how many servers might be pulling the image down during any given deployment. Of course, 252 MB is still pretty big for a production base layer, but hopefully you get the idea.

Optimizing for the Cache

The final building technique that we will cover here is related to keeping build times as fast as possible. One of the important goals of the DevOps movement is to keep feedback loops as tight as possible. This means that it is important to try to ensure that problems are discovered and reported as quickly as possible so that they can be fixed when people are still completely focused on the code in question and haven't moved on to other unrelated tasks.

During any standard build process, Docker uses a layer cache to try to avoid rebuilding any image layers that it has already built and that do not contain any noticeable changes. Because of this cache, the order in which you do things inside your *Dockerfile* can have a dramatic impact on how long your builds take on average.

For starters let's take the *Dockerfile* from the previous example and customize it just a bit, so that it looks like this:

```
FROM fedora
RUN dnf install -y httpd && \
    dnf clean all
RUN mkdir /var/www && \
    mkdir /var/www/html
ADD index.html /var/www/html
CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
```

Now, in the same directory, let's also create a new file called *index.html* that looks like this:

```
<html>
  <head>
    <title>My custom Web Site</title>
  </head>
  <body>
    <p>Welcome to my custom Web Site</p>
  </body>
</html>
```

For the first test, let's time the build without using the Docker cache at all, by using the following command:

```
$ time docker build --no-cache .
Sending build context to Docker daemon  3.072kB
...
Step 2/5 : RUN dnf install -y httpd &&     dnf clean all
...
Step 3/5 : RUN mkdir /var/www &&     mkdir /var/www/html
 ---> Running in 3bfc31e235da
Removing intermediate container 3bfc31e235da
 ---> a21db49f0442
Step 4/5 : ADD index.html /var/www/html
 ---> ff677486fe44
...
```

```
Successfully built 8798731d106c

real 1m7.611s
user 0m0.088s
sys  0m0.074s
```

TIP

Windows users should be able to use the PowerShell `Measure-Command` function to replace the Unix `time` command used in these examples.

The output from the `time` command tells us that the build without the cache took about one minute and seven seconds. If you rebuild the image immediately afterward, and allow Docker to use the cache, you will see that the build is very fast.

```
time docker build .
Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM fedora
 ---> 422dc563ca32
Step 2/5 : RUN dnf install -y httpd &&     dnf clean all
 ---> Using cache
 ---> 88ae32ca622c
Step 3/5 : RUN mkdir /var/www &&     mkdir /var/www/html
 ---> Using cache
 ---> c3dc1fc9eb8b
Step 4/5 : ADD index.html /var/www/html
 ---> Using cache
 ---> d3dcfe6bc6d6
Step 5/5 : CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
 ---> Using cache
 ---> 1974815809c4
Successfully built 1974815809c4

real 0m0.266s
user 0m0.051s
sys  0m0.052s
```

Since none of the layers changed and the cache could be fully leveraged, the build took only a fraction of a second to complete. Now, let’s make a small improvement to the *index.html* file so that it looks like this:

```
<html>
  <head>
    <title>My custom Web Site</title>
  </head>
  <body>
    <div align="center">
      <p>Welcome to my custom Web Site!!!</p>
    </div>
  </body>
</html>
```

and then let’s time the rebuild again:

```
$ time docker build .
Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM fedora
 ---> 422dc563ca32
Step 2/5 : RUN dnf install -y httpd &&     dnf clean all
 ---> Using cache
 ---> 88ae32ca622c
Step 3/5 : RUN mkdir /var/www &&     mkdir /var/www/html
 ---> Using cache
 ---> c3dc1fc9eb8b
Step 4/5 : ADD index.html /var/www/html
 ---> dec032325309
Step 5/5 : CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
 ---> Running in ea4e4d246cda
Removing intermediate container ea4e4d246cda
 ---> 59f4f67cd756
Successfully built 59f4f67cd756

real  0m0.933s
user  0m0.050s
sys  0m0.048s
```

If you look at the output carefully, you will see that the cache was used for most of the build. It wasn’t until Docker step 4, when Docker needed to copy *index.html*, that the cache was invalidated and the layers had to be recreated. Because the cache could be used for most of the build, the build still did not exceed a second.

But what would happen if you changed the order of the commands in the *Dockerfile* so that they looked like this:

```
FROM fedora
RUN mkdir /var/www && \
    mkdir /var/www/html
ADD index.html /var/www/html
RUN dnf install -y httpd && \
    dnf clean all
CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
```

Let’s quickly time another test build without the cache to get a baseline:

```
$ time docker build --no-cache .
Sending build context to Docker daemon  3.072kB
...
Successfully built c949ed0f036e

real 0m48.824s
```

```
user 0m0.076s
sys  0m0.066s
```

In this case, the build took 48 seconds to complete. The difference in time from the very first test is entirely due to fluctuating network speeds and has nothing to do with the changes that you have made to the *Dockerfile*.

Now, let’s edit *index.html* again like so:

```
<html>
  <head>
    <title>My custom Web Site</title>
  </head>
  <body>
    <div align="center" style="font-size:180%">
      <p>Welcome to my custom Web Site</p>
    </div>
  </body>
</html>
```

And now, let’s time the image rebuild, while using the cache:

```
$ time docker build .
Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM fedora
 ---> 422dc563ca32
Step 2/5 : RUN mkdir /var/www &&     mkdir /var/www/html
 ---> Using cache
 ---> 04bee02fe0ba
Step 3/5 : ADD index.html /var/www/html
 ---> acb3cb2e9cee
Step 4/5 : RUN dnf install -y httpd &&     dnf clean all
 ---> Running in cc569be95c12
...
Successfully built 6597845f8514

real 0m48.511s
user 0m0.077s
sys  0m0.066s
```

The first time that you rebuilt the image, after editing the *index.html* file, it took only .933 seconds, but this time it took 48.511 seconds, almost exactly as long as it took to build the whole image without using the cache at all.

This is because you have modified the *Dockerfile* so that the *index.html* file is copied into the image very early in the process. The problem with doing it this way is that the *index.html* file changes frequently and will often invalidate the cache. The other issue it that it is unnecessarily placed before a very time-consuming step in our *Dockerfile*: installing the Apache web server.

The important lesson to take away from all of this is that order matters, and in general, you should always try to order your *Dockerfile* so that the most stable and time-consuming portions of your build process happen first and your code is added as late in the process as possible.

For projects that require you to install dependencies based on your code using tools like `npm` and `bundle`, it is also a good idea to do some research about optimizing your Docker builds for those platforms. This often includes locking down your dependency versions and storing them along with your code so that they do not need to be downloaded for each and every build.

Wrap-Up

At this point you should feel comfortable with the basic day-to-day activities around image creation for Docker. In the next chapter, we will start to dig into how you can use your images to create containerized processes for your projects.

[1] Don’t Repeat Yourself.

[2] This code was forked from GitHub.

[3] Which in turn has been purchased by Red Hat.

Chapter 5. Working with Docker Containers

In the previous chapter, we learned how to build a Docker image and the very basic steps required for running the resulting image within a container. In this chapter, we'll first take a look at the history of container technology and then dive deeper into running containers and exploring the Docker commands that control the overall configuration, resources, and privileges that your container receives.

What Are Containers?

You might be familiar with virtualization systems like VMware or KVM that allow you to run a complete Linux kernel and operating system on top of a virtualized layer, commonly known as a *hypervisor*. This approach provides very strong isolation between workloads because each virtual machine hosts its own operating system kernel that sits in a separate memory space on top of a hardware virtualization layer.

Containers are fundamentally different, since they all share a single kernel, and isolation between workloads is implemented entirely within that one kernel. This is called *operating system virtualization*. The `libcontainer` README provides a good, short definition of a container: "A container is a self-contained execution environment that shares the kernel of the host system and which is (optionally) isolated from other containers in the system." One of the major advantages of containers is resource efficiency, because you don't need a whole operating system instance for each isolated workload. Since you are sharing a kernel, there is one less layer of indirection between the isolated task and the real hardware underneath. When a process is running inside a container, there is only a little bit of code that sits inside the kernel managing the container. Contrast this with a virtual machine where there would be a second layer running. In a VM, calls by the process to the hardware or hypervisor would require bouncing in and out of privileged mode on the processor twice, thereby noticeably slowing down many calls.

But the container approach does mean that you can only run processes that are compatible with the underlying kernel. For example, unlike hardware virtualization provided by technologies like VMware or KVM, Windows applications cannot run inside a Linux container on a Linux host. Windows applications can, however, run inside Windows containers on a Windows host.

So containers are best thought of as an OS-specific technology where, at least for now, you can run any of your favorite applications or daemons that are compatible with the container server. When thinking of containers, you should try very hard to throw out what you might already know about virtual machines and instead conceptualize a container as a wrapper around a process that actually runs on the server.

> **NOTE**
>
> In addition to being able to run containers inside virtual machines, it is actually completely feasible to run a virtual machine inside a container. If you do this, then it is actually possible to run a Windows application inside a Windows VM that is running inside a Linux container.

History of Containers

It is often the case that a revolutionary technology is an older technology that has finally arrived in the spotlight. Technology goes in waves, and some of the ideas from the 1960s are back in vogue. Similarly, Docker is a newer technology and it has an ease of use that has made it an instant hit, but it doesn't exist in a vacuum. Much of what underpins Docker comes from work done over the last 30 years in a few different arenas. We can easily trace the conceptual evolution of containers from a simple system call that was added to the Unix kernel in the late 1970s all the way to the modern container tooling that powers many huge internet firms, like Google, Twitter, and Facebook. It's worth taking some time for a quick tour through how the technology evolved and led to the creation of Docker, because understanding that helps you place it within the context of other things you might be familiar with.

Containers are not a new idea. They are a way to isolate and encapsulate a part of the running system. The oldest technology in this area includes the very first batch processing systems. When using these early computers, the system would literally run one program at a time, switching to run another program once the previous program had finished or a set time period had been reached. With this design there was enforced isolation: you could make sure your program didn't step on anyone else's program, because it was only possible to run one thing at a time. Although modern computers still switch tasks constantly, it is incredibly fast and completely unnoticeable to most users.

We would argue that the seeds for today's containers were planted in 1979 with the addition of the `chroot` system call to Version 7 Unix. `chroot` restricts a process's view of the underlying filesystem to a single subtree. The `chroot` system call is commonly used to protect the operating system from untrusted server processes like FTP, BIND, and Sendmail, which are publicly exposed and susceptible to compromise.

In the 1980s and 1990s, various Unix variants were created with mandatory access controls for security reasons.[1] This meant you had tightly controlled domains running on the same Unix kernel. Processes in each domain had an extremely limited view of the system that precluded them from interacting across domains. A popular commercial version of Unix that implemented this idea was the Sidewinder firewall built on top of BSDI Unix, but this was not possible with most mainstream Unix implementations.

That changed in 2000 when FreeBSD 4.0 was released with a new command, called `jail`, which was designed to allow shared-environment hosting providers to easily and securely create a separation between their processes and those of their individual customers. FreeBSD `jail` expanded `chroot`'s capabilities and also restricted everything a process could do with the underlying system and other jailed processes.

In 2004, Sun released an early build of Solaris 10, which included Solaris Containers, and later evolved into Solaris Zones. This was the first major commercial implementation of container technology and is still used today to support many commercial container implementations. In 2005 OpenVZ for Linux was released by the company Virtuozzo, followed in 2007 by HP's Secure Resource Partitions for HP-UX, which was later renamed to HP-UX Containers. Finally, in 2008, Linux Containers (LXC) were released in version 2.6.24 of the Linux kernel. The phenomenal growth of Linux Containers across the community did not really start to grow until 2013 with the inclusion of user namespaces in version 3.8 of the Linux kernel and the release of Docker one month later.

Companies, like Google, that had to deal with scaling applications for broad internet consumption started pushing container technology in the early 2000s in order to facilitate distributing their applications across global data centers full of computers. A few companies maintained their own patched Linux kernels with container support for internal use, but as the need for these features became more evident within the Linux community, Google contributed some of its own work supporting containers into the mainline Linux kernel.

Creating a Container

So far we've started containers using the handy `docker run` command. But `docker run` is really a convenience command that wraps two separate steps into one. The first thing it does is create a container from the underlying image. We can accomplish this separately using the `docker create` command. The second thing `docker run` does is execute the container, which we can also do separately with the `docker start` command.

The `docker create` and `docker start` commands both contain all the options that pertain to how a container is initially set up. In Chapter 4, we demonstrated that with the `docker run` command you could map network ports in the underlying container to the host using the `-p` argument, and that `-e` could be used to pass environment variables into the container.

This only just begins to touch on the array of things that you can configure when you first create a container. So let's take a look at some of the options that `docker` supports.

Basic Configuration

Let's start by exploring some of the ways we can tell Docker to configure our container when we create it.

CONTAINER NAME

When you create a container, it is built from the underlying image, but various command-line arguments can affect the final settings. Settings specified in the *Dockerfile* are always used as defaults, but you can override many of them at creation time.

By default, Docker randomly names your container by combining an adjective with the name of a famous person. This results in names like *ecstatic-babbage* and *serene-albattani*. If you want to give your container a specific name, you can use the `--name` argument.

```
$ docker create --name="awesome-service" ubuntu:latest sleep 120
```

After creating this container, you could then start it by using the `docker start awesome-service`. It will automatically exit after 120 seconds, but you could stop it before then by running `docker stop awesome-service`. We will dive a bit more into each of these commands a little later in the chapter.

> **WARNING**
>
> You can only have one container with any given name on a Docker host. If you run the preceding command twice in a row, you will get an error. You must either delete the previous container using `docker rm` or change the name of the new container.

LABELS

As mentioned in Chapter 4, labels are key/value pairs that can be applied to Docker images and containers as metadata. When new Docker containers are created, they automatically inherit all the labels from their parent image.

It is also possible to add new labels to the containers so that you can apply metadata that might be specific to that single container.

```
docker run -d --name has-some-labels -l deployer=Ahmed -l tester=Asako \
  ubuntu:latest sleep 1000
```

You can then search for and filter containers based on this metadata, using commands like `docker ps`.

```
$ docker ps -a -f label=deployer=Ahmed
CONTAINER ID  IMAGE         COMMAND       ... NAMES
845731631ba4  ubuntu:latest "sleep 1000"  ... has-some-labels
```

You can use the `docker inspect` command on the container to see all the labels that a container has.

```
$ docker inspect 845731631ba4
...
```

```
        "Labels": {
            "deployer": "Ahmed",
            "tester": "Asako"
        },
```

```
...
```

Note that this container runs the command `sleep 1000`, so after 1,000 seconds it will stop running.

HOSTNAME

By default, when you start a container, Docker copies certain system files on the host, including */etc/hostname*, into the container's configuration directory on the host,[2] and then uses a bind mount to link that copy of the file into the container. We can launch a default container with no special configuration like this:

```
$ docker run --rm -ti ubuntu:latest /bin/bash
```

This command uses the `docker run` command, which runs `docker create` and `docker start` in the background. Since we want to be able to interact with the container that we are going to create for demonstration purposes, we pass in a few useful arguments. The `--rm` argument tells Docker to delete the container when it exits, the `-t` argument tells Docker to allocate a pseudo-TTY, and the `-i` argument tells Docker that this is going to be an interactive session, and we want to keep STDIN open. The final argument in the command is the executable that we want to run within the container, which in this case is the ever-useful `/bin/bash`.

If we now run the `mount` command from within the resulting container, we'll see something similar to this:

```
root@ebc8cf2d8523:/# mount
overlay on / type overlay (rw,relatime,lowerdir=...,upperdir=...,workdir..
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev type tmpfs (rw,nosuid,mode=755)
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=65536k)
mqueue on /dev/mqueue type mqueue (rw,nosuid,nodev,noexec,relatime)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,...,ptmxmode=666
sysfs on /sys type sysfs (ro,nosuid,nodev,noexec,relatime)
/dev/sda9 on /etc/resolv.conf type ext4 (rw,relatime,data=ordered)
/dev/sda9 on /etc/hostname type ext4 (rw,relatime,data=ordered)
/dev/sda9 on /etc/hosts type ext4 (rw,relatime,data=ordered)
devpts on /dev/console type devpts (rw,nosuid,noexec,relatime,...,ptmxmode=
proc on /proc/sys type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sysrq-trigger type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/irq type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/bus type proc (ro,nosuid,nodev,noexec,relatime)
tmpfs on /proc/kcore type tmpfs (rw,nosuid,mode=755)
root@ebc8cf2d8523:/#
```

NOTE

When you see any examples with a prompt that looks something like *root@hashID*, it means that you are running a command within the container instead of on the Docker host. Note that there are occasions when a container will have been configured with a different hostname instead (e.g., using `--name` on the CLI), but in the default case, it's a hash.

There are quite a few bind mounts in a container, but in this case we are interested in this one:

```
/dev/sda9 on /etc/hostname type ext4 (rw,relatime,data=ordered)
```

While the device number will be different for each container, the part we care about is that the mount point is */etc/hostname*. This links the container's */etc/hostname* to the hostname file that Docker has prepared for the container, which by default contains the container's ID and is not fully qualified with a domain name.

We can check this in the container by running the following:

```
root@ebc8cf2d8523:/# hostname -f
ebc8cf2d8523
root@ebc8cf2d8523:/# exit
```

NOTE

Don't forget to `exit` the container shell to return to the Docker host when finished.

To set the hostname specifically, we can use the `--hostname` argument to pass in a more specific value.

```
$ docker run --rm -ti --hostname="mycontainer.example.com" \
    ubuntu:latest /bin/bash
```

Then, from within the container, we'll see that the fully qualified hostname is defined as requested.

```
root@mycontainer:/# hostname -f
mycontainer.example.com
root@mycontainer:/# exit
```

DOMAIN NAME SERVICE

Just like */etc/hostname*, the *resolv.conf* file that configured Domain Name Service (DNS) resolution is managed via a bind mount between the host and container.

```
/dev/sda9 on /etc/resolv.conf type ext4 (rw,relatime,data=ordered)
```

By default, this is an exact copy of the Docker host’s *resolv.conf* file. If you didn’t want this, you could use a combination of the `--dns` and `--dns-search` arguments to override this behavior in the container:

```
$ docker run --rm -ti --dns=8.8.8.8 --dns=8.8.4.4 --dns-search=example1.co
    --dns-search=example2.com ubuntu:latest /bin/bash
```

NOTE

If you want to leave the search domain completely unset, then use `--dns-search=.`

Within the container, you would still see a bind mount, but the file contents would no longer reflect the host’s *resolv.conf*; instead, it would now look like this:

```
root@0f887071000a:/# more /etc/resolv.conf
nameserver 8.8.8.8
nameserver 8.8.4.4
search example1.com example2.com
root@0f887071000a:/# exit
```

MAC ADDRESS

Another important piece of information that you can configure is the media access control (MAC) address for the container.

Without any configuration, a container will receive a calculated MAC address that starts with the *02:42:ac:11* prefix.

If you need to specifically set this to a value, you can do so by running something similar to this:

```
$ docker run --rm -ti --mac-address="a2:11:aa:22:bb:33" ubuntu:latest /bin
```

Normally you will not need to do that. But sometimes you want to reserve a particular set of MAC addresses for your containers in order to avoid other virtualization layers that use the same private block as Docker.

WARNING

Be very careful when customizing the MAC address settings. It is possible to cause ARP contention on your network if two systems advertise the same MAC address. If you have a strong need to do this, try to keep your locally administered address ranges within some of the official ranges, like *x2-xx-xx-xx-xx-xx*, *x6-xx-xx-xx-xx-xx*, *xA-xx-xx-xx-xx-xx*, and *xE-xx-xx-xx-xx-xx* (with *x* being any valid hexidecimal character).

Storage Volumes

There are times when the default disk space allocated to a container, or the container's ephemeral nature, is not appropriate for the job at hand, so you'll need storage that can persist between container deployments.

WARNING

Mounting storage from the Docker host is not generally advisable because it ties your container to a particular Docker host for its persistent state. But for cases like temporary cache files or other semi-ephemeral states, it can make sense.

For times like this, you can leverage the `-v` command to mount directories and individual files from the host server into the container. The following example mounts */mnt/session_data* to */data* within the container:

```
$ docker run --rm -ti -v /mnt/session_data:/data ubuntu:latest /bin/bash
root@0f887071000a:/# mount | grep data
/dev/sda9 on /data type ext4 (rw,relatime,data=ordered)
root@0f887071000a:/# exit
```

TIP

By default, volumes are mounted read-write, but you can easily modify this command to make it mount the directory read-only:

```
docker run --rm -ti -v /mnt/session_data:/data:ro \
  ubuntu:latest /bin/bash
```

Neither the host mount point nor the mount point in the container needs to preexist for this command to work properly. If the host mount point does not exist already, then it will be created as a directory. This could cause you some issues if you were trying to point to a file instead of a directory.

In the mount options, you can see that the filesystem was mounted read-write on */data* as expected.

SELINUX AND VOLUME MOUNTS

If you have SELinux enabled on your Docker host, you may get a “Permission Denied” error when trying to mount a volume into your container. You can handle this by using one of the `z` options to the Docker command for mounting volumes:

- The lowercase `z` option indicates that the bind mount content is shared among multiple containers.

- The uppercase `Z` option indicates that the bind mount content is private and unshared.

If you are going to share a volume between containers, you can use the `z` option to the volume mount:

```
docker run -v /app/dhcpd/etc:/etc/dhcpd:z dhcpd
```

However, the best option is actually the `Z` option to the volume mount command, which will set the directory with the exact MCS label (e.g., chcon … -l s0:c1,c2) that the container will be using. This provides for the best security and will allow only a single container to mount the volume:

```
docker run -v /app/dhcpd/etc:/etc/dhcpd:Z dhcpd
```

WARNING

Use extreme caution with the `z` options. Bind-mounting a system directory such as */etc* or */var* with the `Z` option will very likely render your system inoperable and require you to relabel the host machine manually.

If the container application is designed to write into */data*, then this data will be visible on the host filesystem in */mnt/session_data* and will remain available when this container stops and a new container starts with the same volume mounted.

It is possible to tell Docker that the root volume of your container should be mounted read-only so that processes within the container cannot write anything to the root filesystem. This prevents things like logfiles, which a developer may be unaware of, from filling up the container’s allocated disk in production. When it’s used in conjunction with a mounted volume, you can ensure that data is written only into expected locations.

In the previous example, we could accomplish this simply by adding `--read-only=true` to the command.

```
$ docker run --rm -ti --read-only=true -v /mnt/session_data:/data \
    ubuntu:latest /bin/bash
root@df542767bc17:/# mount | grep " / "
overlay on / type overlay (ro,relatime,lowerdir=...,upperdir=...,workdir=.
root@df542767bc17:/# mount | grep data
/dev/sda9 on /data type ext4 (rw,relatime,data=ordered)
root@df542767bc17:/# exit
```

If you look closely at the mount options for the root directory, you'll notice that they are mounted with the `ro` option, which makes it read-only. However, the */session_data* mount is still mounted with the `rw` option so that our application can successfully write to the one volume to which it's designed to write.

Sometimes it is necessary to make a directory like */tmp* writeable, even when the rest of the container is read-only. For this use case, you can use the `--tmpfs` argument with `docker run`, so that you can mount a *tmpfs* filesystem into the container. Any data in these *tmpfs* directories will be lost when the container is stopped. The following example shows a container being launched with a *tmpfs* filesystem mounted at */tmp* with the `rw`, `noexec`, `nodev`, `nosuid`, and `size=256M` mount options set:

```
$ docker run --rm -ti --read-only=true --tmpfs \
    /tmp:rw,noexec,nodev,nosuid,size=256M ubuntu:latest /bin/bash
root@25b4f3632bbc:/# df -h /tmp
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           256M     0  256M   0% /tmp
root@25b4f3632bbc:/# grep /tmp /etc/mtab
tmpfs /tmp tmpfs rw,seclabel,nosuid,nodev,noexec,relatime,size=262144k 0 0
root@25b4f3632bbc:/# exit
```

WARNING

Containers should be designed to be stateless whenever possible. Managing storage creates undesirable dependencies and can easily make deployment scenarios much more complicated.

Resource Quotas

When people discuss the types of problems they must often cope with when working in the cloud, the “noisy neighbor” is often near the top of the list. The basic problem this term refers to is that other applications running on the same physical system as yours can have a noticeable impact on your performance and resource availability.

Virtual machines have the advantage that you can easily and very tightly control how much memory and CPU, among other resources, are allocated to the virtual machine. When using Docker, you must instead leverage the cgroup functionality in the Linux kernel to control the resources that are available to a Docker container. The `docker create` and `docker run` commands directly support configuring CPU, memory, swap, and storage I/O restrictions when you create a container.

> **NOTE**
>
> Constraints are normally applied at the time of container creation. If you need to change them, you can use the `docker container update` command or deploy a new container with the adjustments.

There is an important caveat here. While Docker supports various resource limits, you must have these capabilities enabled in your kernel in order for Docker to take advantage of them. You might need to add these as command-line parameters to your kernel on startup. To figure out if your kernel supports these limits, run `docker info`. If you are missing any support, you will get warning messages at the bottom, like:

```
WARNING: No swap limit support
```

> **NOTE**
>
> The details regarding getting cgroup support configured for your kernel are distribution-specific, so you should consult the Docker documentation if you need help configuring things.

CPU SHARES

Docker has several ways to limit CPU usage by applications in containers. The original method, and one still commonly used, is the concept of *cpu shares*. Below we'll present other options as well.

The computing power of all the CPU cores in a system is considered to be the full pool of shares. Docker assigns the number *1024* to represent the full pool. By configuring a container's CPU shares, you can dictate how much time the container gets to use the CPU for. If you want the container to be able to use at most half of the computing power of the system, then you would allocate it *512* shares. Note that these are not exclusive shares, meaning that assigning all *1024* shares to a container does not prevent all other containers from running. Rather, it's a hint to the scheduler about how long each container should be able to run each time it's scheduled. If we have one container that is allocated *1024* shares (the default) and two that are allocated *512*, they will all get scheduled the same number of times. But if the normal amount of CPU time for each process is 100 microseconds, the containers with *512* shares will run for 50 microseconds each time, whereas the container with *1024* shares will run for 100 microseconds.

Let's explore a little bit how this works in practice. For the following examples, we'll use a new Docker image that contains the `stress` command for pushing a system to its limits.

When we run `stress` without any cgroup constraints, it will use as many resources as we tell it to. The following command creates a load average of around 5 by creating two CPU-bound processes, one I/O-bound process, and two memory allocation processes. Note that in the following code, we are running on a system with two CPUs.

```
$ docker run --rm -ti progrium/stress \
  --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

WARNING

This should be a reasonable command to run on any modern computer system, but be aware that it is going to stress the host system. So, don't do this in a location that can't take the additional load, or even a possible failure, due to resource starvation.

If you run the `top` or `htop` command on the Docker host, near the end of the two-minute run, you can see how the system is affected by the load created by the `stress` program.

```
$ top -bn1 | head -n 15
top - 20:56:36 up 3 min,  2 users,  load average: 5.03, 2.02, 0.75
```

```
Tasks:  88 total,   5 running,  83 sleeping,   0 stopped,   0 zombie
%Cpu(s): 29.8 us, 35.2 sy,  0.0 ni, 32.0 id,  0.8 wa,  1.6 hi,  0.6 si,  0
KiB Mem:   1021856 total,   270148 used,   751708 free,    42716 buffers
KiB Swap:        0 total,        0 used,        0 free.    83764 cached Me

  PID USER       PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAN
  810 root       20   0    7316     96      0 R  44.3  0.0   0:49.63 stress
  813 root       20   0    7316     96      0 R  44.3  0.0   0:49.18 stress
  812 root       20   0  138392  46936    996 R  31.7  4.6   0:46.42 stress
  814 root       20   0  138392  22360    996 R  31.7  2.2   0:46.89 stress
  811 root       20   0    7316     96      0 D  25.3  0.0   0:21.34 stress
    1 root       20   0  110024   4916   3632 S   0.0  0.5   0:07.32 system
    2 root       20   0       0      0      0 S   0.0  0.0   0:00.04 kthrea
    3 root       20   0       0      0      0 S   0.0  0.0   0:00.11 ksofti
```

NOTE

Docker Community Edition users on non-Linux systems may discover that Docker has made the VM filesystem read-only and it does not contain many useful tools for monitoring the VM. For these demos where you want to be able to monitor the resource usage of various processes, you can work around this by doing something like this:

```
$ docker run -it --privileged --pid=host alpine sh
/ # apk update
/ # apk add htop
/ # htop -p $(pgrep stress | tr '\n' ',')
/ # exit
```

Be aware that the preceding `htop` command will give you an error unless `stress` is actively running when you launch `htop`, since no processes will be returned by the `pgrep` command.

If you want to run the exact same `stress` command again, with only half the amount of available CPU time, you can do so like this:

```
$ docker run --rm -ti --cpu-shares 512 progrium/stress \
    --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

The `--cpu-shares 512` is the flag that does the magic, allocating 512 CPU shares to this container. Note that the effect might not be noticeable on a system that is not very busy. That's because the container will continue to be scheduled for the same time-slice length whenever it has work to do, unless the system is constrained for resources. So in our case, the results of a `top` command on the host system will likely look exactly the same, unless you run a few more containers to give the CPU something else to do.

WARNING

Unlike virtual machines, Docker's cgroup-based constraints on CPU shares can have unexpected consequences. They are not hard limits; they are a relative limit, similar to the `nice` command. An example is a container that is constrained to half the CPU shares but is on a system that is not very busy. Since the CPU is not busy, the limit on the CPU shares would have only a limited effect because there is no competition in the scheduler pool. When a second container that uses a lot of CPU is deployed to the same system, suddenly the effect of the constraint on the first container will be noticeable. Consider this carefully when constraining containers and allocating resources.

CPU PINNING

It is also possible to pin a container to one or more CPU cores. This means that work for this container will be scheduled only on the cores that have been assigned to this container. That is useful if you want to hard-shard CPUs between applications or if you have applications that need to be pinned to a particular CPU for things like cache efficiency.

In the following example, we are running a stress container pinned to the first of two CPUs, with 512 CPU shares. Note that everything following the container image here are parameters to the `stress` command, not the `docker` command.

```
$ docker run --rm -ti --cpu-shares 512 --cpuset=0 progrium/stress \
    --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

WARNING

The `--cpuset` argument is zero-indexed, so your first CPU core is 0. If you tell Docker to use a CPU core that does not exist on the host system, you will get a `Cannot start container` error. On a two-CPU example host, you could test this by using `--cpuset=0,1,2`.

If you run `top` again, you should notice that the percentage of CPU time spent in user space (`us`) is lower than it previously was, since we have restricted two CPU-bound processes to a single CPU.

```
%Cpu(s): 18.5 us, 22.0 sy,  0.0 ni, 57.6 id,  0.5 wa,  1.0 hi,  0.3 si,  0
```

NOTE

When you use CPU pinning, additional CPU sharing restrictions on the container only take into account other containers running on the same set of cores.

Using the CPU CFS (Completely Fair Scheduler) within the Linux kernel, you can alter the CPU quota for a given container by setting the `--cpu-quota` flag to a valid value when launching the container with `docker run`.

SIMPLIFYING CPU QUTOAS

While CPU shares were the original mechanism in Docker for managing CPU limits, Docker has evolved a great deal since and one of the ways that it now makes users' lives easier is by greatly simplifying how CPU quotas can be set. Instead of trying to set CPU shares and quotas correctly, you can now simply tell Docker how much CPU you would like to be available to your container, and it will do the math required to set the underlying cgroups correctly.

The `--cpus` command can be set to a floating-point number between 0.01 and the number of CPU cores on the Docker server.

```
$ docker run -d --cpus=".25" progrium/stress \
    --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 60s
```

If you try to set the value too high, you’ll get an error message from Docker (not the `stress` application) that will give you the correct range of CPU cores that you have to work with.

```
$ docker run -d --cpus="40.25" progrium/stress \
    --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 60s
docker: Error response from daemon: Range of CPUs is from
    0.01 to 4.00, as there are only 4 CPUs available.
See 'docker run --help'.
```

The `docker update` command can be used to dynamically adjust the resource limits of one of more containers. You could adjust the CPU allocation on two containers simultaneously, for example, like so:

```
docker update --cpus="1.5" 6b785f78b75e 92b797f12af1
```

MEMORY

We can control how much memory a container can access in a manner similar to constraining the CPU. There is, however, one fundamental difference: while constraining the CPU only impacts the application’s priority for CPU time, the memory limit is a *hard* limit. Even on an unconstrained system with 96 GB of free memory, if we tell a container that it may have access only to 24 GB, then it will only ever get to use 24 GB regardless of the free memory on the system. Because of the way the virtual memory system works on Linux, it’s possible to allocate more memory to a container than the system has actual RAM. In this case, the container will resort to using swap, just like a normal Linux process.

Let’s start a container with a memory constraint by passing the `--memory` option to the `docker run` command:

```
$ docker run --rm -ti --memory 512m progrium/stress \
    --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

When you use the `--memory` option alone, you are setting both the amount of RAM and the amount of swap that the container will have access to. So by using `--memory 512m` here, we’ve constrained the container to 512 MB of RAM and 512 MB of additional swap space. Docker supports `b`, `k`, `m`, or `g`, representing bytes, kilobytes, megabytes, or gigabytes, respectively. If your system somehow runs Linux and Docker and has multiple terabytes of memory, then unfortunately you’re going to have to specify it in gigabytes.

If you would like to set the swap separately or disable it altogether, you need to also use the `--`

memory-swap option. This defines the total amount of memory and swap available to the container. If we rerun our previous command, like so:

```
$ docker run --rm -ti --memory 512m --memory-swap=768m progrium/stress \
    --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

Then we’re telling the kernel that this container can have access to 512 MB of memory and 256 MB of additional swap space. Setting the `--memory-swap` option to `-1` will disable the swap completely within the container. We can also specifically limit the amount of kernel memory available to a container by using the `--kernel-memory` argument to `docker run` or `docker create`.

WARNING

Again, unlike CPU shares, memory is a hard limit! This is good, because the constraint doesn’t suddenly have a noticeable effect on the container when another container is deployed to the system. But it does mean that you need to be careful that the limit closely matches your container’s needs because there is no wiggle room. An out-of-memory container causes the kernel to behave just like it would if the system were out of memory. It will try to find a process to kill in order to free up space. This is a common failure case where containers have their memory limits set too low. The telltale sign of this issue is a container exit code of 137 and kernel out-of-memory (OOM) messages in the `dmesg` output.

So, what happens if a container reaches its memory limit? Well, let’s give it a try by modifying one of our previous commands and lowering the memory significantly:

```
$ docker run --rm -ti --memory 100m progrium/stress --cpu 2 --io 1 --vm 2
    --vm-bytes 128M --timeout 120s
```

While all of our other runs of the `stress` container ended with the line:

```
stress: info: [1] successful run completed in 120s
```

We see that this run quickly fails with the line:

```
stress: FAIL: [1] (452) failed run completed in 1s
```

This is because the container tries to allocate more memory than it is allowed, and the Linux out-of-memory (OOM) killer is invoked and starts killing processes within the cgroup to reclaim memory. Since our container has only one running process, this kills the container.

Docker has features that allow you to tune and disable the Linux OOM killer by using the `--oom-kill-disable` and the `--oom-score-adj` arguments to `docker run`.

If you access your Docker server, you can see the kernel message related to this event by running `dmesg`. The output will look something like this:

```
[ 4210.403984] stress invoked oom-killer: gfp_mask=0x24000c0 ...
[ 4210.404899] stress cpuset=5bfa65084931efabda59d9a70fa8e88 ...
[ 4210.405951] CPU: 3 PID: 3429 Comm: stress Not tainted 4.9 ...
[ 4210.406624] Hardware name:   BHYVE, BIOS 1.00 03/14/2014
...
[ 4210.408978] Call Trace:
[ 4210.409182]  [<ffffffff94438115>] ? dump_stack+0x5a/0x6f
....
[ 4210.414139]  [<ffffffff947f9cf8>] ? page_fault+0x28/0x30
[ 4210.414619] Task in /docker-ce/docker/5...3
killed as a result of limit of /docker-ce/docker/5...3
[ 4210.416640] memory: usage 102380kB, limit 102400kB, failc ...
[ 4210.417236] memory+swap: usage 204800kB, limit 204800kB,  ...
[ 4210.417855] kmem: usage 1180kB, limit 9007199254740988kB, ...
[ 4210.418485] Memory cgroup stats for /docker-ce/docker/5...3:
cache:0KB rss:101200KB rss_huge:0KB mapped_file:0KB dirty:0KB
writeback:11472KB swap:102420KB inactive_anon:50728KB
active_anon:50472KB inactive_file:0KB active_file:0KB unevictable:0KB
...
[ 4210.426783] Memory cgroup out of memory: Kill process 3429...
[ 4210.427544] Killed process 3429 (stress) total-vm:138388kB,
anon-rss:44028kB, file-rss:900kB, shmem-rss:0kB
[ 4210.442492] oom_reaper: reaped process 3429 (stress), now
anon-rss:0kB, file-rss:0kB, shmem-rss:0kB
```

This out-of-memory event will also be recorded by Docker and viewable via `docker events`.

```
$ docker events
2018-01-28T15:56:19.972142371-08:00 container oom \
    d0d803ce32c4e86d0aa6453512a9084a156e96860e916ffc2856fc63ad9cf88b \
    (image=progrium/stress, name=loving_franklin)
```

BLOCK I/O

Many containers are just stateless applications and won't have a need for I/O restrictions. But Docker also supports limiting block I/O in a few different ways via the cgroups mechanism.

The first way is applying some prioritization to a container's use of block device I/O. You enable this by manipulating the default setting of the `blkio.weight` cgroup attribute. This attribute can have a value of 0 (disabled) or a number between 10 and 1000, the default being 500. This limit acts a bit like CPU shares, in that the system will divide all of the available I/O between every process within a cgroup slice by 1000, with the assigned weights impacting how much available I/O is available to each process.

To set this weight on a container, you need to pass the `--blkio-weight` to your `docker run` command with a valid value. You can also target a specific device using the `--blkio-weight-device` option.

As with CPU shares, tuning the weights is hard to get right in practice, but we can make it vastly simpler by limiting the maximum number of bytes or operations per second that are available to a container via its cgroup. The following settings let us control that:

```
--device-read-bps     Limit read rate (bytes per second) from a device
--device-read-iops    Limit read rate (IO per second) from a device
--device-write-bps    Limit write rate (bytes per second) to a device
--device-write-iops   Limit write rate (IO per second) to a device
```

You can test how these impact the performance of a container by running some of the following commands, which use the Linux I/O tester `bonnie`.

```
$ time docker run -ti --rm spkane/train-os:latest bonnie++ -u 500:500 \
    -d /tmp -r 1024 -s 2048 -x 1
...
real    0m27.715s
user    0m0.027s
sys     0m0.030s

$ time docker run -ti --rm --device-write-iops /dev/sda:256 \
    spkane/train-os:latest bonnie++ -u 500:500 -d /tmp -r 1024 -s 2048 -x
...
real    0m58.765s
user    0m0.028s
sys     0m0.029s

$ time docker run -ti --rm --device-write-bps /dev/sda:5mb \
    spkane/train-os:latest bonnie++ -u 500:500 -d /tmp -r 1024 -s 2048 -x
...
```

TIP

Windows users should be able to use the PowerShell `Measure-Command` function to replace the Unix `time` command used in these examples.

In our experience, the `--device-read-ops` and `--device-write-ops` are the most effective way to set limits, and the one we recommend. Of course there could be reasons why one of the other methods is better for your use case, so you should know about them.

ULIMITS

Before Linux cgroups, there was another way to place a limit on the resources available to a process: the application of user limits via the `ulimit` command. That mechanism is still available and still useful for all of the use cases where it was traditionally used.

The following code is a list of the types of system resources that you can usually constrain by setting soft and hard limits via the `ulimit` command:

```
$ ulimit -a
core file size (blocks, -c) 0
data seg size (kbytes, -d) unlimited
scheduling priority (-e) 0
file size (blocks, -f) unlimited
pending signals (-i) 5835
max locked memory (kbytes, -l) 64
max memory size (kbytes, -m) unlimited
open files (-n) 1024
pipe size (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority (-r) 0
stack size (kbytes, -s) 10240
cpu time (seconds, -t) unlimited
max user processes (-u) 1024
virtual memory (kbytes, -v) unlimited
file locks (-x) unlimited
```

It is possible to configure the Docker daemon with the default user limits that you want to apply to every container. The following command tells the Docker daemon to start all containers with a soft limit of 50 open files and a hard limit of 150 open files:

```
$ sudo dockerd --default-ulimit nofile=50:150
```

You can then override these ulimits on a specific container by passing in values using the `--ulimit` argument.

```
$ docker run -d --ulimit nofile=150:300 nginx
```

There are some additional advanced commands that you can use when creating containers, but this covers many of the more common use cases. The Docker client documentation lists all the available options and is updated with each Docker release.

Starting a Container

Before we got into the details of containers and constraints, we created our container using the `docker create` command. That container is just sitting there without doing anything. There is a configuration, but no running process. When we're ready to start the container, we can do so using the `docker start` command.

Let's say that we needed to run a copy of Redis, a common key/value store. We won't really do anything with this Redis container, but it's a lightweight, long-lived process and serves as an example of something we might do in a real environment. We could first create the container:

```
$ docker create -p 6379:6379 redis:2.8
Unable to find image 'redis:2.8' locally
30d39e59ffe2: Pull complete
...
868be653dea3: Pull complete
511136ea3c5a: Already exists
redis:2.8: The image you are pulling has been verified. Important: ...
Status: Downloaded newer image for redis:2.8
6b785f78b75ec2652f81d92721c416ae854bae085eba378e46e8ab54d7ff81d1
```

The result of the command is some output, the last line of which is the full hash that was generated for the container. We could use that long hash to start it, but if we failed to note it down, we could also list all the containers on the system, whether they are running or not, using:

```
$ docker ps -a
CONTAINER ID  IMAGE                    COMMAND               ...
6b785f78b75e  redis:2.8                "/entrypoint.sh redi  ...
92b797f12af1  progrium/stress:latest   "/usr/bin/stress --v  ...
```

We can identify our container by the image we used and the creation time (truncated here for formatting). We can then start the container with the following command:

```
$ docker start 6b785f78b75e
```

NOTE

Most Docker commands will work with the container name, the full hash, the short hash, or even just enough of the hash to make it unique. In the previous example, the full hash for the container is *6b785f78b75ec2652f81d92… bae085eba378e46e8ab54d7ff81d1*, but the short hash that is shown in most command output is *6b785f78b75e*. This short hash consists of the first 12 characters of the full hash. In the previous example, running `docker start 6b7` would have worked just fine.

That *should* have started the container, but with it running it the background we won't necessarily know if something went wrong. To verify that it's running, we can run:

```
$ docker ps
CONTAINER ID   IMAGE       COMMAND                     ...   STATUS         ...
6b785f78b75e   redis:2.8   "/entrypoint.sh redi        ...   Up 2 minutes   ...
```

And, there it is: running as expected. We can tell because the status says "Up" and shows how long the container has been running.

Auto-Restarting a Container

In many cases, we want our containers to restart if they exit. Some containers are very short-lived and come and go quickly. But for production applications, for instance, you expect them to be up and running at all times after you've told them to run. If you are running a more complex system, a scheduler may do this for you.

In the simple case, we can tell Docker to manage restarts on our behalf by passing the `--restart` argument to the `docker run` command. It takes four values: `no`, `always`, or `on-failure`, or `unless-stopped`. If restart is set to `no`, the container will never restart if it exits. If it is set to `always`, the container will restart whenever it exits, with no regard to the exit code. If restart is set to `on-failure:3`, then whenever the container exits with a nonzero exit code, Docker will try to restart the container three times before giving up. `unless-stopped` is the most command choice, and will restart the container unless it is intentionally stopped with something like `docker stop`.

We can see this in action by rerunning our last memory-constrained stress container without the `--rm` argument, but with the `--restart` argument.

```
$ docker run -ti --restart=on-failure:3 --memory 100m progrium/stress \
    --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

In this example, we'll see the output from the first run appear on the console before it dies. If we run a `docker ps` immediately after the container dies, we'll see that Docker is attempting to restart the container.

```
$ docker ps
...  IMAGE                      ...  STATUS                               ..
...  progrium/stress:latest     ...  Restarting (1) Less than a second ago ..
```

It will continue to fail because we haven't given it enough memory to function properly. After three attempts, Docker will give up and we'll see the container disappear from the output of `docker ps`.

Stopping a Container

Containers can be stopped and started at will. You might think that starting and stopping a container is analogous to pausing and resuming a normal process, but it's not quite the same in reality. When stopped, the process is not paused; it actually exits. And when a container is stopped, it no longer shows up in the normal `docker ps` output. On reboot, Docker will attempt to start all of the containers that were running at shutdown. If you need to prevent a container from doing any additional work, without actually stopping the process, then you can pause the Docker container with `docker pause` and `unpause`, which will be discussed in more detail later. For now, go ahead and stop our container:

```
$ docker stop 6b785f78b75e
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

Now that we have stopped the container, nothing is in the running `ps` list! We can start it back up with the container ID, but it would be really inconvenient to have to remember that. So `docker ps` has an additional option `(-a)` to show all containers, not just the running ones.

```
$ docker ps -a
CONTAINER ID  IMAGE                  STATUS                 ...
6b785f78b75e  progrium/stress:latest Exited (0) 2 minutes ago ...
```

That `STATUS` field now shows that our container exited with a status code of 0 (no errors). We can start it back up with all of the same configuration it had before:

```
docker start 6b785f78b75e
6b785f78b75e
$ docker ps -a
CONTAINER ID IMAGE                       ... STATUS    ...
6b785f78b75e progrium/stress:latest  Up 15 seconds ...
```

Voilà, our container is back up and running, and configured just as it was before.

NOTE

Remember that containers exist as a blob of configuration in the Docker system even when they are not started. That means that as long as the container has not been deleted, you can restart it without needing to recreate it. Although memory contents will have been lost, all of the container's filesystem contents and metadata, including environment variables and port bindings, are saved and will still be in place when you restart the container.

By now we've probably thumped on enough about the idea that containers are just a tree of processes that interact with the system in essentially the same way as any other process on the server. But it's important to point it out here again because it means that we can send Unix signals to our process in the containers that they can then respond to. In the previous `docker stop` example, we're sending the container a `SIGTERM` signal and waiting for the container to exit gracefully. Containers follow the same process group signal propagation that any other process group would receive on Linux.

A normal `docker stop` sends a `SIGTERM` to the process. If you want to force a container to be killed if it hasn't stopped after a certain amount of time, you can use the `-t` argument, like this:

```
$ docker stop -t 25 6b785f78b75e
```

This tells Docker to initially send a `SIGTERM` signal as before, but then if the container has not stopped within 25 seconds, to send a `SIGKILL` signal to forcefully kill it.

Although `stop` is the best way to shut down your containers, there are times when it doesn't work and you'll need to forcefully kill a container, just as you might have to do with any process outside of a container.

Killing a Container

When a process is misbehaving, `docker stop` might not cut it. You might just want the container to exit immediately.

In these circumstances, you can use `docker kill` . As you'd expect, it looks a lot like `docker stop`:

```
$ docker kill 6b785f78b75e
6b785f78b75e
```

A `docker ps` command now shows that the container is no longer running, as expected:

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

Just because it was killed rather than stopped does not mean you can't start it again, though. You can just issue a `docker start` like you would for a nicely stopped container. Sometimes you might want to send another signal to a container, one that is not `stop` or `kill`. Like the Linux `kill` command, `docker kill` supports sending any Unix signal. Let's say we wanted to send a `USR1` signal to our container to tell it to do something like reconnect a remote logging session. We could do the following:

```
$ docker kill --signal=USR1 6b785f78b75e
6b785f78b75e
```

If our container actually did something with the `USR1` signal, it would now do it. Since we're just running a bash shell, though, it just continues on as if nothing happened. Try sending a `HUP` signal, though, and see what happens. Remember that `HUP` is the signal that is sent when the terminal closes on a foreground process.

Pausing and Unpausing a Container

There are a few reasons why we might not want to completely stop our container. We might want to pause it, leave its resources allocated, and leave its entries in the process table. That could be because we're taking a snapshot of its filesystem to create a new image, or just because we need some CPU on the host for a while. If you are used to normal Unix process handling, you might wonder how this actually works since containerized processes are just processes.

Pausing leverages the `cgroups` freezer, which essentially just prevents your process from being scheduled until you unfreeze it. This will prevent the container from doing anything while maintaining its overall state, including memory contents. Unlike stopping a container, where the processes are made aware that they are stopping via the `SIGSTOP` signal, pausing a container doesn't send any information to the container about its state change. That's an important distinction. Several Docker commands use pausing and unpausing internally as well. Here is how we pause a container:

```
$ docker pause 6b785f78b75e
```

> **NOTE**
>
> To pause and unpause containers in Windows, you must be using Hyper-V as the underlying virtualization technology.

If we look at the list of running containers, we will now see that the Redis container status is listed as (Paused).

```
# docker ps
CONTAINER ID   IMAGE                      ...  STATUS                      ...
6b785f78b75e   progrium/stress:latest     ...  Up 36 minutes  (Paused)    ...
```

Attempting to use the container in this paused state would fail. It's present, but nothing is running. We can now resume the container by using the `docker unpause` command.

```
$ docker unpause 6b785f78b75e
6b785f78b75e
$ docker ps
CONTAINER ID   IMAGE                      ... STATUS ...
6b785f78b75e   progrium/stress:latest     ... Up 38 minutes ...
```

It’s back to running, and `docker ps` correctly reflects the new state. Note that it shows “Up 38 minutes” now, because Docker still considers the container to be running even when it is paused.

Cleaning Up Containers and Images

After running all these commands to build images, create containers, and run them, we have accumulated a lot of image layers and container folders on our system.

We can list all the containers on our system using the `docker ps -a` command and then delete any of the containers in the list. We must stop all containers that are using an image before removing the image itself. Assuming we’ve done that, we can remove it as follows, using the `docker rm` command:

```
$ docker ps -a
CONTAINER ID  IMAGE                    ...
92b797f12af1  progrium/stress:latest   ...
...
$ docker rm 92b797f12af1
```

We can then list all the images on our system using:

```
$ docker images
REPOSITORY        TAG      IMAGE ID      CREATED         VIRTUAL SIZE
ubuntu            latest   5ba9dab47459  3 weeks ago     188.3 MB
redis             2.8      868be653dea3  3 weeks ago     110.7 MB
progrium/stress   latest   873c28292d23  7 months ago    281.8 MB
```

We can then delete an image and all associated filesystem layers by running:

```
$ docker rmi 873c28292d23
```

WARNING

If you try to delete an image that is in use by a container, you will get a `Conflict, cannot delete` error. You should stop and delete the container(s) first.

There are times, especially during development cycles, when it makes sense to completely purge all the images or containers from your system. The easiest way to do this is by running the `docker system prune` command.

```
$ docker system prune
WARNING! This will remove:
        - all stopped containers
        - all networks not used by at least one container
        - all dangling images
        - all build cache
Are you sure you want to continue? [y/N] y
Deleted Containers:
cbbc42acfe6cc7c2d5e6c3361003e077478c58bb062dd57a230d31bcd01f6190
...
Deleted Images:
deleted: sha256:bec6ec29e16a409af1c556bf9e6b2ec584c7fb5ffbfd7c46ec00b30bf
untagged: spkane/squid@sha256:64fbc44666405fd1a02f0ec731e35881465fac395e7
...
Total reclaimed space: 1.385GB
```

TIP

To remove all unused images, instead of only dangling images, try `docker system prune -a`

It is also possible to craft more specific commands to accomplish similar goals.

To delete all of the containers on your Docker hosts, use the following command:

```
$ docker rm $(docker ps -a -q)
```

And to delete all the images on your Docker host, this command will get the job done:

```
$ docker rmi $(docker images -q)
```

The `docker ps` and `docker images` commands both support a `filter` argument that can make it easy to fine-tune your delete commands for certain circumstances.

To remove all containers that exited with a nonzero state, you can use this filter:

```
$ docker rm $(docker ps -a -q --filter 'exited!=0')
```

And to remove all untagged images, you can type:

```
$ docker rmi $(docker images -q -f "dangling=true")
```

NOTE

You can read the official Docker documentation to explore the filtering options. At the moment there are very few filters to choose from, but more will likely be added over time. And if you're really interested, Docker is an open source project, so it is always open to public code contributions.

You can also make your own very creative filters by stringing together commands using pipes (|) and other similar techniques.

In production systems that see a lot of deployments, you can sometimes end up with old containers or unused images lying around and filling up disk space. It can be useful to script some of these `docker rm` and `docker rmi` commands to run on a schedule (e.g., running under `cron` or via a `systemd` timer). You can use what you've learned to do that for yourself, or you could look at something like Spotify's docker-gc tool to keep your servers nice and neat. `docker-gc` has some nice options that let you keep a certain number of images or containers around, exempt certain images from garbage collection, or exclude recent containers. The tool includes a lot of what you would probably implement on your own, but it has been heavily battle-hardened and works well.

Windows Containers

Up to now we have focused entirely on Docker commands for Linux containers, since this is the most common use case and works on all Docker platforms. However, since 2016, the Microsoft Windows platform has supported running Windows Docker containers that include native Windows applications and can be managed with the usual set of Docker commands.

Windows containers are not really the focus of this book, since they make up only a very tiny portion of production containers at this point and they aren't compatible with the rest of the Docker ecosystem because they require Windows-specific container images. However, they're a growing and important part of the Docker world, so we'll take a brief look at how they work. In fact, except for the actual contents of the containers, almost everything else works the same as on Linux Docker containers. In this section we'll run through a quick example of how you can run a Windows container on Windows 10 with Hyper-V and Docker.

> **TIP**
>
> For this to work, you must be using Docker Community Edition (or Enterprise Edition) on a 64-bit edition of Windows 10 (Professional or better).

The first thing you'll need to do is to switch Docker from Linux Containers to Windows Containers. To do this, right-click on the Docker whale icon in your taskbar and select `Switch to Windows Containers`…. You should get a notification that Docker is switching, and this process might take some time, although usually it happens almost immediately. Unfortunately, there is no notification that the switch has completed, so you'll just need to try using the Windows version. If you right-click on the Docker icon again, you should now see `Switch to Linux Containers`… in place of the original option.

> **NOTE**
>
> If the first time you right-click on the Docker icon, it reads `Switch to Linux Containers`…, then you are already configured for Windows containers.

We can test a simple Windows container by opening up PowerShell and trying to run the following command:

```
PS C:\> docker run -it microsoft/nanoserver powershell 'Write-Host "Hello

Hello World
```

This will download and launch a base container for Windows Server Nano Server and then use PowerShell scripting to print `Hello World` to the screen.

If you want to build an image that accomplishes the same task, you can create the following *Dockerfile*:

```
FROM microsoft/nanoserver

RUN powershell.exe Add-Content C:\\helloworld.ps1 'Write-Host "Hello W
orld"'

CMD ["powershell", "C:\\helloworld.ps1"]
```

When we build this *Dockerfile* it will base the container on `microsoft/nanoserver`, create a very small PowerShell script, and then set the image to run the script by default, when this image is used to launch a container.

> **WARNING**
>
> You may have noticed that we had to escape the backslash (\) with an additional backslash in the preceding *Dockerfile*. This is because Docker has its roots in Unix and the backslash has a special meaning in Unix shells. So, we escape all backslashes to ensure that Docker does not interpret them to mean that this command is continued on the next line.

If you build this *Dockerfile* now, you'll see something similar to this:

```
PS C:\> docker build -t windows-helloworld:latest .

Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM microsoft/nanoserver
 ---> 8a62949f0058
Step 2/3 : RUN powershell.exe Add-Content C:\\helloworld.ps1 'Write-Host \
             "Hello World"'
 ---> Using cache
 ---> 930ed3b401fb
Step 3/3 : CMD ["powershell", "C:\\helloworld.ps1"]
 ---> Using cache
 ---> 445764524411
Successfully built 445764524411
Successfully tagged windows-helloworld:latest
```

And now if you run the resulting image, you'll see this:

```
PS C:\> docker run --rm -ti windows-helloworld:latest

Hello World
```

Microsoft maintains good documentation about Windows containers that also includes an example of building a container that launches a .NET application.

> **TIP**
>
> On the Windows platform, it is also useful to know that you can get improved isolation for your container by launching it inside a dedicated and very lightweight Hyper-V virtual machine. You can do this very easily, by simply adding the `--isolation=hyperv` option to your `docker create` and `docker run` commands. There is a small performance and resource penalty for this, but it does significantly improve the isolation of your container. You can read more about this in the documentation.

Even if you plan to mostly work with Windows Containers, for the rest of the book you should switch back to Linux Containers, so that all the examples work as expected. When you are done reading and are ready to dive into building your own containers, you can always switch back.

> **TIP**
>
> Remember that you can reenable Linux Containers by right-clicking on the Docker icon, and selecting `Switch to Linux Containers...`.

Wrap-Up

In the next chapter, we'll continue our exploration of what Docker brings to the table. For now it's probably worth doing a little experimentation on your own. We suggest exercising some of the container control commands we covered here so that you're familiar with the command-line options and the overall syntax. Now would even be a great time to try to design and build a small image and then launch it as a new container. When you are ready to continue, head on to Chapter 6!

[1] SELinux is one current implementation.

[2] Typically under */var/lib/docker/containers*.

Chapter 6. Exploring Docker

Now that you have some experience working with containers and images, we can explore some of Docker's other capabilities. In this chapter, we'll continue to use the `docker` command-line tool to talk to the running `dockerd` server that you've configured, while visiting some of the other fundamental commands.

Docker provides commands to do a number of additional things easily:

- Printing the Docker version
- Viewing the server information
- Downloading image updates
- Inspecting containers
- Entering a running container
- Returning a result
- Viewing logs
- Monitoring statistics
- And much more…

Let's take a look at these and some of the additional community tooling that augments Docker's native capabilities.

Printing the Docker Version

If you completed the last chapter, you have a working Docker daemon on a Linux server or virtual machine, and you've started a base container to make sure it's all working. If you haven't set that up already and you want to try out the steps in the rest of the book, you'll want to follow the installation steps in Chapter 3 before you move on with this section.

The absolute simplest thing you can do with Docker is print the versions of the various components. It might not sound like much, but this is a useful tool to have because the server and API are often not backward compatible with older clients. Knowing how to show the version will help you troubleshoot certain types of connection issues. Sometimes, for example, the Docker client will give you a cryptic message about mismatched API versions and it's nice to be able to translate that into Docker versions so you know which component to change. Note that this command actually talks to the remote Docker server. If you can't connect to the server for any reason, the client will complain. If you find that you have a connectivity problem, you should probably revisit the steps in the last chapter.

> **NOTE**
>
> You can always directly log in to the Docker server and run `docker` commands from a shell on the server if you are troubleshooting issues or simply do not want to use the `docker` client to connect to a remote system. On most Docker servers, this will require either `root` privileges or membership in the `docker` group in order to connect to the Unix domain socket that Docker is listening on.

Since we just installed all of the Docker components at the same time, when we run `docker version`, we should see that all of our versions match:

```
$ docker version
Client:
 Version:      17.09.1-ce
 API version:  1.32
 Go version:   go1.8.3
 Git commit:   19e2cf6
 Built:        Thu Dec  7 22:24:23 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.09.1-ce
 API version:  1.32 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   19e2cf6
 Built:        Thu Dec  7 22:23:00 2017
 OS/Arch:      linux/amd64
 Experimental: false
```

Notice how we have different sections representing the client and server. In this case we have a matching client and server since we just installed them together. But it's important to note that this won't always be the case. Hopefully, in your production system you can manage to keep the same version running on most systems. But it's not uncommon for development environments and build systems to have slightly different versions.

Another important point is that different versions of the command-line tools might use the same Docker API version. Even when they do, sometimes the Docker CLI tool won't let you talk to a remote server that doesn't exactly match. Usually the command-line tool requires an exact match. But other API clients and libraries will usually work across a large number of Docker versions, depending on which API version they require. In the `Server` section, we can see that it's telling us that the current API version is 1.32 and the minimum API it will serve is 1.12. This is useful information when you're working with third-party clients. Now you know how to verify this information.

Server Information

We can also find out a lot about the Docker server via the Docker client. Later we'll talk more about what all of this means, but you can find out which filesystem backend the Docker server is running, which kernel version it is on, which operating system it is running on, which plug-ins are installed, which runtime is being used, and how many containers and images are currently stored there. `docker info` will present you with something something similar to this, which has been shortened for brevity:

```
$ docker info
Containers: 43
...
Images: 53
Server Version: 17.09.1-ce
Storage Driver: overlay2
...
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk sy
Runtimes: cc-runtime runc
Default Runtime: runc
...
Kernel Version: 4.4.0-104-generic
Operating System: Ubuntu 16.04.3 LTS
Docker Root Dir: /var/lib/docker
...
```

Depending on how your Docker daemon is set up, this might look somewhat different. Don't be concerned about that; this is just to give you an example. Here we can see that our server is an Ubuntu 16.04.3 LTS release running the 4.4.0 Linux kernel and backed with the overlay2 filesystem driver. We also have a lot of images and containers! With a fresh install, this number should be zero.

The information about plug-ins is worth pointing out here. It's telling us about all the things this installation of Docker supports. On a fresh install, this will look more or less like this, depending on which new plug-ins are distributed with Docker. Docker itself is made up of many different plug-ins all working together. This is powerful because it means it's also possible to install a number of other plug-ins contributed by members of the community. It's useful to be able to see which are installed even if just for the purpose of making sure Docker has recognized one that you added.

In most installations, */var/lib/docker* will be the default root directory used to store images and containers. If you need to change this, you can edit your Docker startup scripts to launch the daemon, with the `--data-root` argument pointing to a new storage location. To test this by hand, you could run something like this:

```
$ sudo dockerd \
    -H unix:///var/run/docker.sock \
    -H tcp://0.0.0.0:2375 --data-root="/data/docker"
```

We will talk more abut runtimes later, but here you can see that we have two runtimes installed. The `runc` runtime is the default Docker runtime. If you think of Docker containers, you are usually thinking about the type of container that `runc` builds. On this server we also have the Intel Clear Containers runtime installed, which works a little differently. We'll talk more about that in Chapter 11.

Downloading Image Updates

We're going to use an Ubuntu base image for the following examples. Even if you have already grabbed the Ubuntu base image once, you can `pull` it again and it will automatically pick up any updates that have been published since you last ran it. That's because `latest` is a tag that, by convention, is always moved to the most recent version of the image that has been published to the image registry. Invoking `pull` will look like this:

```
$ docker pull ubuntu:latest

Pulling repository ubuntu
5506de2b643b: Download complete
```

```
511136ea3c5a: Download complete
d497ad3926c8: Download complete
ccb62158e970: Download complete
e791be0477f2: Download complete
3680052c0f5c: Download complete
22093c35d77b: Download complete
```

That command pulled down only the layers that have changed since we last ran the command. You might see a longer or shorter list, or even an empty list, depending on when you ran it and what changes have been pushed to the registry since then.

> **TIP**
>
> It’s good to remember that even though you pulled `latest`, Docker won’t automatically keep the local image up to date for you. You’ll be responsible for doing that yourself. However, if you deploy an image based on a newer copy of `ubuntu:latest`, the Docker client will download the missing layers during the deployment just like you would expect. Keep in mind that this is the behavior of the Docker client, and other libraries or API tools may not behave this way. It’s always safest to deploy production code using a fixed tag rather than the `latest` tag. This helps guarantee that you got the version you expected.

In addition to referring to items in the registry by the `latest` tag or another version number tag, you can also refer to them by their content-addressable tag. These look like `sha256:2f9a…82cf` (where we’ve shortened a very long ID). These are actually generated as a hashed sum of the contents of the image and are a very precise identifier. This is by far the safest way to refer to Docker images where you need to make sure you are getting the exact version you expected, because these can’t be moved like a version tag. The syntax for pulling them from the registry is very similar, but note the `@` in the tag.

```
docker pull ubuntu@sha256:2f9a...82cf
```

Note that unlike most Docker commands where you may shorten the hash, you cannot do that with SHA256 hashes. You must use the full hash here.

Inspecting a Container

Once you have a container created, running or not, you can now use `docker` to see how it was configured. This is often useful in debugging, and also has some other information that can be useful for identifying a container.

For this example, go ahead and start up a container:

```
$ docker run -d -t ubuntu /bin/bash
3c4f916619a5dfc420396d823b42e8bd30a2f94ab5b0f42f052357a68a67309b
```

We can list all our running containers with `docker ps` to ensure everything is running as expected, and to copy the container ID.

```
$ docker ps
CONTAINER ID  IMAGE         COMMAND     ... STATUS         ...  NAMES
3c4f916619a5  ubuntu:latest "/bin/bash" ... Up 31 seconds  ...  angry_mesto
```

In this case, our ID is `3c4f916619a5`. We could also use `angry_mestorf`, which is the dynamic name assigned to our container. Many underlying tools need the unique container ID, though, so it's useful to get into the habit of looking at that first. As we mentioned earlier, the ID as shown is actually the truncated version, but Docker treats these interchangeably with the long ones. As is the case in many version control systems, this hash is actually just the prefix of a much longer hash. Internally, the kernel uses a 64-byte hash to identify the container. But that's painful for humans to use, so Docker supports the shortened hash.

The output to `docker inspect` is pretty verbose, so we'll cut it down in the following code block to a few values worth pointing out. You should look at the full output to see what else you think is interesting:

```
$ docker inspect 3c4f916619a5
```

```
[{
    "Id": "3c4f916619a5dfc420396d823b42e8bd30a2f94ab5b0f42f052357a68a6
7309b",
    "Created": "2018-11-07T22:06:32.229471304Z",
    ...
    "Args": [],
    ...
    "Image": "sha256:ed889b69344b43252027e19b41fb98d36...a9098a6d"
    ...
    "Config": {
        "Hostname": "3c4f916619a5",
        ...
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "Cmd": [
            "/bin/bash"
        ],
        ...
        "Image": "ubuntu",
        ...
    },
    ...
}]
```

Note that long "Id" string. That's the full unique identifier of this container. Luckily we can use the short version, even if that's still not especially convenient. We can also see the exact time when the container was created in a much more precise way than docker ps gives us.

Some other interesting things are shown here as well: the top-level command in the container, the environment that was passed to it at creation time, the image on which it's based, and the hostname inside the container. All of these are configurable at container creation time if you need to do so. The usual method for passing configuration to containers, for example, is via environment variables, so being able to see how a container was configured via docker inspect can reveal a lot when you're debugging.

Exploring the Shell

Let's get a container running with just an interactive bash shell so we can take a look around. We'll do that, as we did before, by just running something like:

```
$ docker run -i -t ubuntu:16.04 /bin/bash
```

That will run an Ubuntu 16.04 LTS container with the bash shell as the top-level process. By specifying the 16.04 tag, we can be sure to get a particular version of the image. So, when we start that container, what processes are running?

```
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 22:12 ?        00:00:00 /bin/bash
root        12     1  0 22:16 ?        00:00:00 ps -ef
```

Wow, that's not much, is it? It turns out that when we told docker to start bash, we didn't get anything but that. We're inside a whole Linux distribution image, but no other processes started for us automatically. We only got what we asked for. It's good to keep that in mind going forward.

WARNING

Docker containers don't, by default, start anything in the background like a full virtual machine would. They're a lot lighter weight than that and therefore don't start an `init` system. You can, of course, run a whole `init` system if you need to, or Docker's own tini init system, but you have to ask for it. We'll talk about that more in Chapter 7.

That's how we get a shell running in a container. You should feel free to poke around and see what else looks interesting inside the container. Note that you might have a pretty limited set of commands available. You're in an Ubuntu distribution, though, so you can fix that by using `apt-get` to install more packages. Note that these are only going to be around for the life of this container. You're modifying the top layer of the container, not the base image! Containers are by nature ephemeral, so anything you do inside this container won't outlast it.

Returning a Result

Would you spin up a whole virtual machine in order to run a single process and get the result? You usually wouldn't do that because it would be very time-consuming and require booting a whole operating system to simply execute one command. But Docker doesn't work the same way as virtual machines: containers are very lightweight and don't have to boot up like an operating system. Running something like a quick background job and waiting for the exit code is a normal use case for a Docker container. You can think of it as a way to get remote access to a containerized system and have access to any of the individual commands inside that container with the ability to pipe data to and from them and return exit codes.

This can be useful in lots of scenarios: you might, for instance, have system health checks run this way remotely, or have a series of machines with processes that you spin up via Docker to process a workload and then return. The *docker* command-line tools proxy the results to the local machine. If you run the remote command in foreground mode and don't specify doing otherwise, *docker* will redirect its `stdin` to the remote process, and the remote process's `stdout` and `stderr` to your terminal. The only things we have to do to get this functionality are to run the command in the foreground and not allocate a TTY on the remote. This is actually the default configuration! No command-line options are required. We do need to have a container configured and ready to run beforehand, using `docker create`, or invoking a container we previously created with `docker run` and specifying the image.

If we do that, we can then reinvoke that container and just give it the command that we intend to run. Keep in mind that when we run these commands, Docker starts up the container, executes the command that we requested inside the container's namespaces and cgroups, and then exits, so that no process is left running between invocations. Let's say we created a Docker image whose ID was `8d12decc75fe`. The following code shows what you can then do:

```
$ docker run 8d12decc75fe /bin/false
$ echo $?
1
```

```
$ docker run 8d12decc75fe /bin/true
$ echo $?
0
```

```
$ docker run 8d12decc75fe /bin/cat /etc/passwd

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
nobody:x:99:99:Nobody:/:/sbin/nologin

$ docker run 8d12decc75fe /bin/cat /etc/passwd | wc -l

8
```

Here we executed `/bin/false` on the remote server, which will always exit with a status of 1. Notice how *docker* proxied that result to us in the local terminal. Just to prove that it returns other results, we also run `/bin/true`, which will always return a 0. And there it is.

Then we actually ask `docker` to run `cat /etc/passwd` on the remote container. What we get is a printout of the */etc/passwd* file contained inside that container's filesystem. Because that's just regular output on `stdout`, we can pipe it into local commands just like we would anything else.

WARNING

The previous code pipes the output into the local `wc` command, not a `wc` command in the container. The pipe itself is not passed to the container. If you want to pass the whole command, including the pipes, to the server, you need to invoke a complete shell on the remote side and pass a quoted command, like `bash -c "<your command> | <something else>"`. In the previous code, that would be: `docker run 8d12decc75fe /bin/bash -c "/bin/cat /etc/passwd | wc -l"`.

Getting Inside a Running Container

You can pretty easily get a shell running in a new container as we demonstrated earlier with `docker run`. But it's not the same as getting a new shell inside an existing container that is actively running your application. Every time you use `docker run`, you get a new container. But if you have an existing container that is running an application and you need to debug it from inside the container, you need something else.

Since Docker containers are Linux namespaces, there is both a Docker-native way to get into a container (`docker exec`), and a more Linux-native way to do it, called `nsenter`. Let's take a look at both.

docker exec

First, let's take a look at the easiest and best way to get inside a running container. The `dockerd` server and `docker` command-line tool support remotely executing a new process in a running container via the `docker exec` command. So let's start up a container in background mode, and then enter it using `docker exec` and invoking a shell. The command you invoke doesn't have to be a shell: it's possible to run individual commands inside the container and see their results outside it using `docker exec`. But if you want to get inside the container to look around, a shell is the easiest way to do that.

To run `docker exec`, we'll need our container's ID, like we did previously when we inspected it. For this demo, we listed out the containers and our ID is `589f2ad30138`. We can now use that to get inside the container with `docker exec`. The command line for that, unsurprisingly, looks a lot like the command line for `docker run`. We request an interactive session and a pseudo-TTY with the `-i` and `-t` flags:

```
$ docker exec -i -t 589f2ad30138 /bin/bash
root@589f2ad30138:/#
```

Note that we got a command line back that tells us the ID of the container we're running inside. That's pretty useful for keeping track of where we are. We can now run a normal Linux `ps` to see what else is running inside our container. We should see our other `bash` process that we backgrounded earlier when starting the container.

```
root@589f2ad30138:/# ps -ef
UID          PID   PPID  C STIME TTY          TIME CMD
root           1      0  0 23:13 ?        00:00:00 /bin/bash
root           9      0  1 23:14 ?        00:00:00 /bin/bash
root          17      9  0 23:14 ?        00:00:00 ps -ef
```

WARNING

You can also run additional processes in the background via `docker exec`. You use the `-d` option just like with `docker run`. But you should think hard about doing that for anything but debugging because you lose the repeatability of the image deployment if you depend on this mechanism. Other people would then have to know what to pass to `docker exec` to get the desired functionality. If you're tempted to do this, you would probably reap bigger gains from rebuilding your container image to launch both processes in a repeatable way. If you need to signal to software inside the container to take some action like rotating logs or reloading a configuration, it is cleaner to leverage `docker kill -s <SIGNAL>` with the standard Unix signal name to pass information to the process inside the container.

nsenter

Part of the core `util-linux` package from kernel.org is `nsenter`, short for "Namespace Enter," which allows you to enter any Linux namespace. In Chapter 11, we'll go into more detail on namespaces. But they are the core of what makes a container a container. Using `nsenter`, therefore, we can get into a Docker container from the server itself, even in situations where the `dockerd` server is not responding and we can't use `docker exec`. `nsenter` can also be used to manipulate things in a container as `root` on the server that would otherwise be prevented by `docker exec`, for example. This can be really useful when you are debugging. Most of the time, `docker exec` is all you need, but you should have `nsenter` in your tool belt.

Most Linux distributions ship with the `util-linux` package that contains `nsenter`. But not all of them ship with one that is new enough to have `nsenter` itself installed, because it’s a recent addition to the package. Ubuntu 14.04, for example, still has `util-linux` from 2012. Ubuntu 16.04, however, has the newest package. If you are on a distribution that does not have it, the easiest way to get ahold of `nsenter` is to install it via a third-party Docker container.

This works by pulling a Docker image from the Docker Hub registry and then running a specially crafted Docker container that will install the `nsenter` command-line tool into */usr/local/bin*. This might seem strange at first, but it’s a clever way to allow you to install `nsenter` to any Docker server remotely using nothing more than the `docker` command.

NOTE

The following example is intended to be directly run only on a Linux-based Docker server. It will not work as intended on Docker Community Edition for Mac or Windows. On those systems you can play around with `nsenter` a bit by running something like `docker run -it --privileged --pid=host debian nsenter` and passing whatever additional command-line arguments you want to `nsenter`.

This code shows how we install `nsenter` to */usr/local/bin* on your Docker server:

```
$ docker run --rm -v /usr/local/bin:/target jpetazzo/nsenter
Unable to find image 'jpetazzo/nsenter' locally
Pulling repository jpetazzo/nsenter
9e4ef84f476a: Download complete
511136ea3c5a: Download complete
71d9d77ae89e: Download complete
Status: Downloaded newer image for jpetazzo/nsenter:latest
Installing nsenter to /target
Installing docker-enter to /target
```

WARNING

You should be very careful about doing this! It’s always a good idea to check out what you are running, and particularly what you are exposing part of your filesystem to, before you run a third-party container on your system. With `-v`, we’re telling Docker to expose the host’s */usr/local/bin* directory into the running container as */target*. When the container starts, it is then copying an executable into that directory on our host’s filesystem. In Chapter 11, we will discuss some security frameworks and commands that can be leveraged to prevent potentially nefarious container activities.

Unlike `docker exec`, which can be run remotely, `nsenter` requires that you run it on the server itself. The `README` in the GitHub repo explains how to set this up to work over SSH automatically if you want to do that. For our purposes, we’ll log in to our Docker server and then invoke the command from there. In any case, like with `docker exec`, we need to have a container running. You should still have one running from earlier. If not, go back and start one, and then log into your server.

`docker exec` is pretty simple, but `nsenter` is a little inconvenient to use. It needs to have the PID of the actual top-level process in your container. That’s less than obvious to find and requires a few steps. Luckily there’s a convenience wrapper installed by that Docker container we just ran, called `docker-enter`, which takes away the pain. But before we jump to the convenience wrapper, let’s run `nsenter` by hand so you can see what’s going on.

First we need to find out the ID of the running container, because `nsenter` needs to know that to access it. This is the same as previously shown for `docker inspect` and `docker exec`:

```
$ docker ps
CONTAINER ID  IMAGE          COMMAND      ...  NAMES
3c4f916619a5  ubuntu:latest  "/bin/bash"  ...  grave_goldstine
```

The ID we want is that first field, `3c4f916619a5`. Armed with that, we can now find the PID we need. We do that like this:

```
$ PID=$(docker inspect --format \{{.State.Pid\}} 3c4f916619a5)
```

This will store the PID we care about into the PID environment variable. We need to have `root` privilege to do what we’re going to do. So you should either su to root or use sudo on the command line. Now we invoke nsenter:

```
$ sudo nsenter --target $PID --mount --uts --ipc --net --pid
root@3c4f916619a5:/#
```

NOTE

For Docker Community Edition on Windows and Mac, something like this should accomplish the same thing:

```
$ docker run -it --privileged --pid=host debian nsenter \
    --target $PID --mount --uts --ipc --net --pid
```

If the end result looks a lot like docker exec, that’s because it does almost exactly the same thing under the hood!

There are a lot of command-line options there; what they’re doing is telling nsenter which parts of the container we need access to. Generally you want all of them, so you might expect that to be the default, but it’s not, so we specify them all.

WARNING

Neither nsenter or docker exec work well for exploring a container that does not contain a Unix shell. In this case you usually need to explore the container from the Docker server by navigating directly to where the container filesystem resides on storage. This will typically look something like */var/lib/docker/overlay/365c…*, but will vary based on the Docker setup, storage backend, and container hash. You can determine your Docker root directory by running docker info.

Back at the beginning of this section, we mentioned that there is a convenience wrapper called docker-enter that you install by running the installation Docker container. Having now seen the mechanism involved with running nsenter, you can now appreciate that if you actually just want to enter all the namespaces for the container and skip several steps, you can do this:

```
$ sudo docker-enter 3c4f916619a5 /bin/bash
root@3c4f916619a5:/#
```

Much easier and a lot less to remember. It’s still not anywhere nearly as convenient as `docker exec`, but in some situations it will be all you have if the Docker daemon is not responding, so it’s a very worthwhile tool to have available.

docker volume

Docker supports a `volume` subcommand that makes it possible to list all of the volumes stored in your root directory and then discover additional information about them, including where they are physically stored on the server.

These volumes are not bind-mounted volumes, but special data containers that are useful for persisting data.

If we run a normal `docker` command that bind mounts a directory, we’ll notice that it does not create any Docker volumes.

```
$ docker volume ls
DRIVER              VOLUME NAME

$ docker run -d -v /tmp:/tmp ubuntu:latest sleep 120
6fc97c50fb888054e2d01f0a93ab3b3db172b2cd402fc1cd616858b2b5138857

$ docker volume ls
DRIVER              VOLUME NAME
```

However, you can easily create a new volume with a command like this:

```
$ docker volume create my-data
```

If you then list all your volumes, you should see something like this:

```
# docker volume ls
DRIVER              VOLUME NAME
local               my-data

# docker volume inspect my-data
```

```
[
    {
        "CreatedAt": "2018-07-14T21:01:05Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-data/_data",
        "Name": "my-data",
        "Options": {},
        "Scope": "local"
    }
]
```

Now you can start a container with this data volume attached to it, by running the following:

```
 $ docker run --rm \
     --mount source=my-data,target=/app \
     ubuntu:latest touch /app/my-persistent-data
```

That container created a file in the data volume and then immediately exited.

If we now mount that data volume to a different container, we will see that our data is still there.

```
$ docker run --rm \
    --mount source=my-data,target=/app \
    fedora:latest ls -lFa /app/persistent-data

-rw-r--r-- 1 root root 0 Jul 14 21:05 /app/my-persistent-data
```

And finally, you can delete the data volume when you are done with it by running:

```
$ docker volume rm my-data

my-data
```

NOTE

If you try to delete a volume that is in use by a container (whether it is running or not), you'll get an error like this:

```
Error response from daemon: unable to remove volume:
    remove my-data: volume is in use - [
    d0763e6e8d79e55850a1d3ab21e9d...,
    4b40d52978ea5e784e66ddca8bc22...]
```

These commands should help you to explore your containers in great detail. Once we've explained namespaces more in Chapter 11, you'll get a better understanding of exactly how all these pieces interact and combine to create a container.

Logging

Logging is a critical part of any production application. When things go wrong, logs can be a critical tool in restoring service, so they need to be done well. There are some common ways in which we expect to interact with application logs on Linux systems, some better than others. If you're running an application process on a box, you might expect the output to go to a local logfile that you could read through. Or perhaps you might expect the output to simply be logged to the kernel buffer where it can be read from `dmesg`. Or, as on many modern Linux distributions with `systemd`, you might expect logs to be available from `journalctl`. Because of the container's restrictions and how Docker is constructed, none of these will work without at least some configuration on your part. But that's OK, because logging has first-class support in Docker.

Docker makes logging easier in a few critical ways. First, it captures all of the normal text output from applications in the containers it manages. Anything sent to `stdout` or `stderr` in the container is captured by the Docker daemon and streamed into a configurable logging backend. Secondly, like many other parts of Docker, this system is pluggable and there are lots of powerful options available to you as plug-ins. But let's not dive into the deep end just yet.

docker logs

We'll start with the simplest Docker use case: the default logging mechanism. There are limitations to this mechanism, which we'll explain in a minute, but for the simple case it works well, and it's very convenient. If you are running Docker in development, this is probably the only logging strategy you'll use there. This logging method has been there from the very beginning and is well understood and supported. The mechanism is the `json-file` method. The `docker logs` command exposes most users to this.

As is implied by the name, when you run the default `json-file` logging plug-in, your application's logs are streamed by the Docker daemon into a JSON file for each container. This lets us retrieve logs for any container at any time.

We can display some logs from our example container running `nginx` with the following command:

```
$ docker logs 3c4f916619a5
```

```
2017/11/20 00:34:56 [notice] 12#0: using the "epoll" ...
2017/11/20 00:34:56 [notice] 12#0: nginx/1.0.15
2017/11/20 00:34:56 [notice] 12#0: built by gcc 4.4.7 ...
2017/11/20 00:34:56 [notice] 12#0: OS: Linux 3.8.0-35-generic
```

This is nice because Docker allows you to get the logs remotely, right from the command line, on demand. That's really useful for low-volume logging.

> **NOTE**
>
> To limit the log output to more recent logs, you can use the `--since` option to display only logs after a specified RFC 3339 date (e.g., 2002-10-02T10:00:00-05:00), Unix timestamp (e.g., 1450071961), or Go duration string (e.g., 5m45s). You may also use `--tail` followed by a number of lines to tail.

The actual files backing this logging are on the Docker server itself, by default in */var/lib/docker/containers/*`<container_id>`*/* where the `<container_id>` is replaced by the actual container ID. If you take a look at one of those files, you'll see it's a file with each line representing a JSON object. It will look something like this:

```
{"log":"2018-02-04 23:58:51,003 INFO success: running.\r\n",
"stream":"stdout",
"time":"2018-02-04T23:58:51.004036238Z"}
```

That `log` field is exactly what was sent to `stdout` on the process in question; the `stream` field tells us that this was `stdout` and not `stderr`; and the precise time that the Docker daemon received it is provided in the `time` field. It's an uncommon format for logging, but it's structured rather than just a raw stream, which is beneficial if you want to do anything with the logs later.

Like a logfile, you can also tail the Docker logs live with `docker logs -f`:

```
$ docker logs -f 3c4f916619a5
nginx stderr | 2017/11/20 00:34:56 [notice] 12#0: using the "epoll" ...
nginx stderr | 2017/11/20 00:34:56 [notice] 12#0: nginx/1.0.15
nginx stderr | 2017/11/20 00:34:56 [notice] 12#0: built by gcc 4.4.7 ...
nginx stderr | 2017/11/20 00:34:56 [notice] 12#0: OS: Linux 3.8.0-35-gener
```

This looks identical to the usual `docker logs`, but the client then blocks, waiting on and displaying any new logs to appear, much like the Linux command line `tail -f`.

> **TIP**
>
> By configuring the tag log option similar to `--log-opt tag="{{.ImageName}}/{{.ID}}"`, it is possible to change the default log tag (which every log line will start with) to something more useful. By default, Docker logs will be tagged with the first 12 characters of the container ID.

For single-host logging, this mechanism is pretty good. Its shortcomings are around log rotation, access to the logs remotely once they've been rotated, and disk space usage for high-volume logging. Despite being backed by a JSON file, this mechanism actually performs well enough that most production applications can log this way if that's the solution that works for you. But if you have a more complex environment, you're going to want something more robust, and with centralized logging capabilities.

> **WARNING**
>
> The default settings do not currently enable log rotation. You'll want to make sure you specify the `--log-opt max-size` and `--log-opt max-file` settings if running in production. Those limit the largest file size before rotation and the maximum number of logfiles to keep, respectively. `max-file` does not do anything unless you've also set `max-size` to tell Docker when to rotate the logs. Note that when this is enabled, the `docker logs` mechanism will return data only from the current logfile.

More Advanced Logging

For those times when the default mechanism isn't enough—and at scale it's probably not—Docker also supports configurable logging backends. This list of plug-ins is constantly growing. Currently supported are the `json-file` we described earlier, as well as `syslog`, `fluentd`, `journald`, `gelf`, `awslogs`, `splunk`, `etwlogs`, `gcplogs`, and `logentries`, which are used for sending logs to various popular logging frameworks and services.

That's a big list of plug-ins we just threw out there. The supported option that currently is the simplest for running Docker at scale is the option to send your container logs to `syslog` directly from Docker. You can specify this on the Docker command line with the `--log-driver=syslog` option or set it as the default in the *daemon.json* file for all containers.

> **TIP**
>
> The *daemon.json* file is the configuration for the `dockerd` server. It can usually be found in the */etc/docker/* directory on the server. For Docker Community Edition, this file can be edited in Preferences → Daemon → Advanced from the UI. If you change this file, you will need to restart Docker Community Edition or the `dockerd` daemon.

There are also a number of third-party plug-ins available. We've seen mixed results from third-party plug-ins, primarily because they complicate installing and maintaining Docker. However, you may find that there is a third-party implementation that's perfect for your system, and it might be worth the installation and maintenance hassle.

> **WARNING**
>
> There are some caveats that apply to all of the logging drivers. For example, Docker supports only one at a time. This means that you can use the `syslog` or `gelf` logging friver, but not along with the `json-file` driver. Unless you run `json-file` or `journald`, you will lose the ability to use the `docker logs` command! This may not be expected and is a big consideration when you are changing the driver. There are too many gotchas to go through for each driver, but you should keep in mind the tradeoff between guaranteed delivery of logs and the potential for breaking your Docker deployment. UDP-based solutions or other nonblocking options are recommended.

Traditionally, most Linux systems have some kind of syslog receiver, whether it be `syslog`, `rsyslog`, or any of the many other options. This protocol in its various forms has been around for a long time and is fairly well supported by most deployments. When migrating to Docker from a traditional Linux or Unix environment, many companies already have syslog infrastructure in place, which means this is often the easiest migration path as well.

NOTE

Many newer Linux distributions are based on the `systemd` init system and therefore use `journald` for logging by default, which is different from `syslog`.

While we think that syslog is the easiest solution, it has its problems. The Docker syslog driver supports TLS, TCP, and UDP connection options, which sounds great, but you should be cautious about streaming logs from Docker to a remote log server over TCP or TLS. The problem with this is that they are both run on top of connection-oriented TCP sessions, and Docker tries to connect to the remote logging server at the time of container startup. If it fails to make the connection, it will block trying to start the container. If you are running this as your default logging mechanism, this can strike at any time on any deployment.

We don’t find that to be a particularly usable state for production systems and thus encourage you to use the UDP option for syslog logging if you intend to use the `syslog` driver. This does mean your logs are not encrypted and not guaranteed delivery. There are various philosophies around logging, and you’ll need to balance your need for logs against the reliability of your system. We err on the side of reliability, but if you run in a secure audit environment you may have different priorities. In a tighter security environment, you could run a syslog relay on the Docker host to receive over UDP and relay the logs off box over TLS.

TIP

You can log directly to a remote syslog-compatible server from a single container by setting the log option `syslog-address` similar to this: `--log-opt syslog-address=udp://192.168.42.42:123`.

One final caveat to be aware of regarding most of the logging plug-ins: they are blocking by default, which means that logging back-pressure can cause issues with your application. You can change this behavior by setting `--log-opt mode=non-blocking` and then setting the maximum buffer size for logs to something like `--log-opt max-buffer-size=4m`. Once these are set, the application will no longer block when that buffer fills up. Instead, the oldest loglines in memory will be dropped. Again, reliability needs to be weighed here against your business need to receive all logs.

Non-Plug-In Community Options

Outside of the Docker and third-party plug-ins, there are community contributions with many alternate ways of shipping your logs at scale. The most common non-Docker solution is to use a method to send your logs directly to a syslog-compatible server, bypassing Docker. There are several mechanisms in use:

- Log directly from your application.
- Have a process manager in your container relay the logs (e.g., `systemd`, `upstart`, `supervisor`, or `runit`).
- Run a logging relay in the container that wraps `stdout/stderr` from the container.
- Relay the Docker JSON logs themselves to a remote logging framework from the server or another container.

> **WARNING**
>
> Some third-party libraries and programs, like `supervisor`, write to the filesystem for various (and sometimes unexpected) reasons. If you are trying to design clean containers that do not write directly into the container filesystem, you should consider utilizing the `--read-only` and `--tmpfs` options to `docker run` that we discussed in Chapter 4. Writing logs *inside* the container is not recommended. It makes them hard to get to, prevents them from being preserved beyond the container lifespan, and can wreak havoc with the Docker filesystem backend.

Many of these non-plug-in options share the same drawbacks as changing the logging driver in Docker itself: they hide logs from `docker logs` so that you cannot access them as easily during debugging without relying on an external application. Let’s see how these options stack up.

Logging directly from your application to syslog might make sense, but if you’re running any third-party applications, this approach probably won’t work and is inflexible, since you must redeploy all your application whenever you want to make changes to how logging is handled. And unless you also emit all your logs on `stdout` and `stderr` inside the container, they will not be visible in `docker logs`.

Spotify has a simple, statically linked Go relay to handle logging your `stderr` and `stdout` to syslog for one process inside the container. Generally you run this from the `CMD` line in the *Dockerfile*. Because it's statically compiled, it has no dependencies, which makes it very flexible. By default, this relay swallows the loglines, so they are not visible in `docker logs` unless you start it with the `-tee=true` flag.

The `svlogd` daemon from the `runit` init system can collect logs from your process's `stdout` and ship them to remote hosts over UDP. It's simple to set up and available in many Linux distributions via native packages, which makes it easy to install.

If you want to have one system to support all your containers, a popular option is Logspout, which runs in a separate container, talks to the Docker daemon, and logs all of the system's container logs to syslog (UDP, TCP, TLS). The advantage of this approach is that it does not preclude `docker logs`, but it does require that you set up log rotation. It also *will not* block starting up any containers when the remote syslog server is unavailable!

Finally, while you really should be capturing your logs somewhere, there are rare situations where you simply don't want any logging. You can turn them off completely using the `--log-driver=none` switch.

Monitoring Docker

Among the most important requirements for production systems is that they be observable and measurable. A production system where you are blind to how it's behaving won't work well in the long run. In the modern operations environment, we monitor everything meaningful and report as many useful statistics as we can. Docker supports container health checks and some nice, basic reporting capabilities via `docker stats` and `docker events`. We'll show you those and then look at a community offering from Google that does some nice graphing output, and a—currently experimental—feature of Docker that exports container metrics to the Prometheus monitoring system.

Container Stats

Let's start with the CLI tools that ship with Docker itself. The `docker` CLI has an endpoint for viewing stats of running containers. The command-line tool can stream from this endpoint and every few seconds report back on one or more listed containers, giving basic statistics information about what's happening. `docker stats`, like the Linux `top` command, takes over the current terminal and updates the same lines on the screen with the current information. It's hard to show that in print so we'll just give an example, but this updates every few seconds by default.

COMMAND-LINE STATS

```
$ docker stats b668353c3af5
CONTAINER     CPU %    MEM USAGE/LIMIT    MEM %  NET I/O     BLK I/O    PID
b668353c3af5  1.50%    60.39MiB/200MiB  30.19% 335MB/9.1GB  45.4MB/0B  17
```

Let’s break that rather dense output down into some manageable chunks. What we have is:

1. The container ID (but not the name).

2. The amount of CPU it’s currently consuming.

3. The amount of memory it has in use followed by the maximum amount it’s allowed to use.

4. Network and block I/O statistics.

5. The number of active processes inside the container.

Some of these will be more useful than others for debugging, so let’s take a look at what we’d do with them.

One of the more helpful pieces of output here is the percentage of memory used versus the limit that was set for the container. One common problem with running production containers is that overly aggressive memory limits can cause the kernel OOM (out of memory) killer to stop the container over and over again. The `stats` command can really help with tracking that down.

With regard to I/O statistics, if you run all of your applications in containers, then this summary can make it very clear where your I/O is going from the system. Before containers, this was much harder to figure out!

The number of active processes inside the container is helpful for debugging as well. If you have an application that is spawning children without reaping them, this can expose it pretty quickly.

One great feature of `docker stats` is that it can show not just one container, but all of them in a single summary. That can be pretty revealing, even on boxes where you think you know what they are doing.

That is all useful and easy to digest because it’s human formatted and available on the command line. But there is an additional endpoint on the Docker API that provides a *lot* more information than is shown in the client. We’ve steered away from hitting the API in this book so far, but in this case the data provided by the API is so much richer that we’ll use `curl` to call it and see what our container is doing. It’s nowhere near as nice to read, but there is a lot more detail. This is a good intro to calling the API yourself as well.

STATS API ENDPOINT

The `/stats/` endpoint that we’ll hit on the API will continue to stream stats to us as long as we keep the connection open. Since as humans we can’t really parse the JSON, we’ll just ask for one line and then use Python to “pretty-print” it—we’ll show that shortly. In order for this command to work, you’ll need to have Python installed (version 2.6 or later). If you don’t and you still want to see the JSON output, you can skip the pipe to Python, but you’ll get plain, ugly JSON back. If you already have a favorite JSON pretty-printer, you should feel free to use that instead.

TIP

Another useful tool for pretty-printing JSON on the command line is jq, which is available for most platforms. This is actually an incredibly powerful tool for working with JSON and is worth familiarizing yourself with if you work with JSON much.

Most Docker daemons will be installed with the API available only on the Unix domain socket and not published on TCP. So we’ll use `curl` from the host itself to call the API. If you plan to monitor this endpoint in production, you’ll want to expose the Docker API on a TCP port, usually over SSL and requiring credentials. The Docker documentation will walk you through this.

NOTE

You can usually inspect the contents of the `DOCKER_HOST` environment variable, using something like `echo $DOCKER_HOST`, to discover the hostname or IP address of the Docker server that you are using.

First, start up a container that you can read stats from:

```
$ docker run -d ubuntu:latest sleep 1000
91c86ec7b33f37da9917d2f67177ebfaa3a95a78796e33139e1b7561dc4f244a
```

Now that the container is running, you can get an ongoing stream of statistics about the container in JSON format by running something like `curl` with your container’s hash.

NOTE

In the following examples, we are running `curl` against the Docker socket, but you could just as easily run it against the Docker port, if you have it bound.

```
$ curl --unix-socket /var/run/docker.sock \
    http://v1/containers/91c86ec7b33f/stats
```

NOTE

This JSON stream of statistics will not stop on its own. So for now, we can use the Ctrl-C key combination to stop it.

To get a single group of statistics, we can run something similar to this:

```
$ curl --unix-socket /var/run/docker.sock \
    http://v1/containers/91c86ec7b33f/stats | head -1
```

And finally, if we have Python or another tool capable of pretty-printing JSON, we can make this output human-readable, as shown here:

```
$ curl --unix-socket /var/run/docker.sock \
    http://v1/containers/91c86ec7b33f/stats \
    | head -1 | python -m json.tool
```

```
{
    "blkio_stats": {
        "io_merged_recursive": [],
        "io_queue_recursive": [],
        "io_service_bytes_recursive": [
            {
                "major": 8,
                "minor": 0,
                "op": "Read",
                "value": 6098944
            },
...
        ],
        "io_service_time_recursive": [],
        "io_serviced_recursive": [
            {
                "major": 8,
                "minor": 0,
                "op": "Read",
                "value": 213
            },
...
        ],
        "io_time_recursive": [],
        "io_wait_time_recursive": [],
        "sectors_recursive": []
    },
    "cpu_stats": {
        "cpu_usage": {
...
        },
        "system_cpu_usage": 1884140000000,
        "throttling_data": {
            "periods": 0,
            "throttled_periods": 0,
            "throttled_time": 0
        }
    },
    "memory_stats": {
        "failcnt": 0,
        "limit": 1035853824,
        "max_usage": 7577600,
        "stats": {
...
            "total_active_anon": 1368064,
            "total_active_file": 221184,
            "total_cache": 6148096,
            "total_inactive_anon": 24576,
            "total_inactive_file": 5890048,
            "total_mapped_file": 2215936,
            "total_pgfault": 2601,
            "total_pgmajfault": 46,
            "total_pgpgin": 2222,
            "total_pgpgout": 390,
            "total_rss": 1355776,
            "total_unevictable": 0,
            "unevictable": 0
        },
        "usage": 7577600
    },
    "network": {
        "rx_bytes": 936,
        "rx_dropped": 0,
        "rx_errors": 0,
        "rx_packets": 12,
        "tx_bytes": 468,
        "tx_dropped": 0,
        "tx_errors": 0,
        "tx_packets": 6
    },
    "read": "2018-02-11T15:20:22.930379289-08:00"
}
```

There is *a lot* of information in there. We’ve cut it down to prevent wasting any more trees or electrons than necessary, but even so, there is a lot to digest. The main idea is to let you see how much data is available from the API about each container. We won’t spend much time going into the details, but you can get quite detailed memory usage information, as well as block I/O and CPU usage information.

If you are doing your own monitoring, this is a great endpoint to hit as well. A drawback, however, is that it’s one endpoint per container, so you can’t get the stats about all containers from a single call.

Container Health Checks

As with any other application, when you launch a container it is possible that it will start and run, but never actually enter a healthy state where it could receive traffic. Production systems also fail and your application may become unhealthy at some point during its life, so you need to be able to deal with that.

Many production environments have standardized ways to health-check applications. Unfortunately, there’s no clear standard for how to do that across organizations and so it’s unlikely that many companies do it in the same way. For this reason, monitoring systems have been built to handle that complexity so that they can work in a lot of different production systems. It’s a clear place where a standard would be a big win.

To help remove this complexity and standardize on a universal interface, Docker has added a health-check mechanism. Following the shipping container metaphor, Docker containers should really look the same to the outside world no matter what is inside the container, so Docker’s health-check mechanism not only standardizes health checking for containers, but also maintains the isolation between what is inside the container and what it looks like on the outside. This means that containers from Docker Hub or other shared repositories can implement a standardized health-checking mechanism and it will work in any other Docker environment designed to run production containers.

Health checks are a build-time configuration item and are created with a check definition in the

Dockerfile. This directive tells the Docker daemon what command it can run inside the container to ensure the container is in a healthy state. As long as the command exits with a code of zero (0), Docker will consider the container to be healthy. Any other exit code will indicate to Docker that the container is not in a healthy state, at which point appropriate action can be taken by a scheduler or monitoring system.

We will be using the following project to explore Docker Compose in a few chapters. But, for the moment, it includes a useful example of Docker health checks. Go ahead and pull down a copy of the code and then navigate into the *rocketchat-hubot-demo/mongodb/docker/* directory:

```
$ git clone https://github.com/spkane/rocketchat-hubot-demo.git \
    --config core.autocrlf=input
$ cd rocketchat-hubot-demo/mongodb/docker
```

In this directory, you will see a *Dockerfile* and a script called `docker-healthcheck`. If you view the *Dockerfile*, this is all that you will see:

```
<code class="k">FROM</code> <code class="s2">mongo:3.2</code>

<code class="k">COPY</code> <code class="s2">docker-healthcheck /usr/local

<code class="k">HEALTHCHECK CMD</code> <code class="s2">["docker-healt
hcheck"]</code>
```

It is very short because we are basing this on the upstream Mongo image, and our image inherits a lot of things from that including the entry point, default command, and port to expose:

```
ENTRYPOINT ["docker-entrypoint.sh"]
EXPOSE 27017
CMD ["mongod"]
```

So, in our *Dockerfile* we are only adding a single script that can health-check our container, and defining a health-check command that runs that script.

You can build the container like this:

```
$ docker build -t mongo-with-check:3.2 .
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM mongo:3.2
 ---> 56d7fa068c3d
Step 2/3 : COPY docker-healthcheck /usr/local/bin/
 ---> 217c13baf475
```

```
Step 3/3 : HEALTHCHECK CMD ["docker-healthcheck"]
 ---> Running in d18658520abf
Removing intermediate container d18658520abf
 ---> f69ff65ac29c
Successfully built f69ff65ac29c
Successfully tagged mongo-with-check:3.2
```

And then run the container and looking at the `docker ps` output:

```
$ docker run -d --name mongo-hc mongo-with-check:3.2
1ad213256ab2f24fd65d91e5ce47c8a6c3c47a74cf0251e3afa6fdfc2fbadf0e

$ docker ps
... IMAGE                 ... STATUS                          PORTS     ...
... mongo-with-check:3.2 ... Up 1 second (health: starting) 27017/tcp ...
```

You should notice that the STATUS column now has a `health` section in parentheses. Initially this will display `health: starting` as the container is starting up. You can change the amount of time that Docker waits for the container to initialize using the `--health-start-period` argument to `docker run`. The status will change to `healthy` once the container is up and the health check is successful.

```
$ docker ps
... IMAGE                 ... STATUS                      PORTS     ...
... mongo-with-check:3.2 ... Up About a minute (healthy) 27017/tcp ...
```

You can query this status directly, using the `docker inspect` command.

```
$ docker inspect --format='{{.State.Health.Status}}' mongo-hc
healthy

$ docker inspect --format='{{json .State.Health}}' mongo-hc | jq
```

```
{
  "Status": "healthy",
  "FailingStreak": 0,
  "Log": [
    ...
  ]
}
```

If your container began failing its health check, the status would change to `unhealthy` and you could then determine how to handle the situation.

```
$ docker ps
... IMAGE                ... STATUS                   PORTS     ...
... mongo-with-check:3.2 ... Up 9 minutes (unhealthy) 27017/tcp ...
```

> **TIP**
>
> As with most systems, you can configure a lot of details about your health checks, including how often Docker checks the health (`--health-interval`), how many failures are required to cause the container to be marked unhealthy (`--health-retries`), and more. You can even disable the health check completely (`--no-healthcheck`) if needed.

This feature is very useful, and you should strongly consider using it in all of your containers. This will help you improve both the reliability of your environment and the visibility you have into how things are running in it. It is also supported by most production schedulers and many monitoring systems, so it should be easy to implement.

> **WARNING**
>
> As always, the usefulness of a health check is largely determined by how well-written it is, and how good a job it does at accurately determining the state of the service.

Docker Events

The `dockerd` daemon internally generates an events stream around the container lifecycle. This is how various parts of the system find out what is going on in other parts. You can also tap into this stream to see what lifecycle events are happening for containers on your Docker server. This, as you probably expect by now, is implemented in the `docker` CLI tool as another command-line argument. When you run this command, it will block and continually stream messages to you. Behind the scenes, this is a long-lived HTTP request to the Docker API that returns messages in JSON blobs as they occur. The `docker` CLI tool decodes them and prints some data to the terminal.

This events stream is useful in monitoring scenarios or in triggering additional actions, like wanting to be alerted when a job completes. For debugging purposes, it allows you to see when a container died even if Docker restarts it later. Down the road, this is a place where you might also find yourself directly implementing some tooling against the API. Here's how we use it on the command line:

```
$ docker events
2018-02-18T14:00:39-08:00 1b3295bf300f: (from 0415448f2cc2) die

2018-02-18T14:00:39-08:00 1b3295bf300f: (from 0415448f2cc2) stop

2018-02-18T14:00:42-08:00 1b3295bf300f: (from 0415448f2cc2) start
```

TIP

As with the Docker statistics, you can access the Docker events via `curl` using a command like `curl --unix-socket /var/run/docker.sock` *`http://v1/events`*.

In this example, we initiated a `stop` signal with `docker stop`, and the events stream logs this as a "die" message. The "die" message actually marks the beginning of the shutdown of the container. It doesn't stop instantaneously. So, following the "die" message is a "stop" message, which is what Docker says when a container has actually stopped execution. Docker also helpfully tells us the ID of the image that the container is running on. This can be useful for tying deployments to events, for example, because a deployment usually involves a new image.

Once the container was completely down, we initiated a `docker start` to tell it to run again. Unlike the "die/stop" operations, this is a single command that marks the point at which the container is actually running. We don't get a message telling us that someone explicitly started it. So what happens when we try to start a container that fails?

```
2015-02-18T14:03:31-08:00 e64a279663aa: (from e426f6ef897e) die
```

Note that here the container was actually asked to start, but it failed. Rather than seeing a "start" and a "die," all we see is a "die."

If you have a server where containers are not staying up, the `docker events` stream is pretty helpful in seeing what's going on and when. But if you're not watching it at the time, Docker very helpfully caches some of the events and you can still get at them for some time afterward. You can ask it to display events after a time with the `--since` option, or before with the `--until` option. You can also use both to limit the window to a narrow scope of time when an issue you are investigating may have occurred. Both options take ISO time formats like those in the previous example (e.g., 2018-02-18T14:03:31-08:00).

cAdvisor

`docker stats` and `docker events` are useful but don't yet get us graphs to look at. And graphs are pretty helpful when we're trying to see trends. Of course, other people have filled some of this gap. When you begin to explore the options for monitoring Docker, you will find that many of the major monitoring tools now provide some functionality to help you improve the visibility into your containers' performance and ongoing state.

In addition to the commercial tooling provided by companies like DataDog, GroundWork, and New Relic, there are plenty of options for free, open source tools like Prometheus or even Nagios. We'll talk about Prometheus in the next section, but first we'll start with a nice offering from Google. A few years ago they released their own internal container advisor as a well-maintained open source project on GitHub, called cAdvisor. Although cAdvisor can be run outside of Docker, by now you're probably not surprised to hear that the easiest implementation of cAdvisor is to simply run it as a Docker container.

To install cAdvisor on most Linux systems, all you need to do is run this code:

```
$ docker run \
    --volume=/:/rootfs:ro \
    --volume=/var/run:/var/run:rw \
    --volume=/sys:/sys:ro \
    --volume=/var/lib/docker/:/var/lib/docker:ro \
    --publish=8080:8080 \
    --detach=true \
    --name=cadvisor \
    google/cadvisor:latest

Unable to find image 'google/cadvisor:latest' locally
Pulling repository google/cadvisor
f0643dafd7f5: Download complete
...
ba9b663a8908: Download complete
Status: Downloaded newer image for google/cadvisor:latest
f54e6bc0469f60fd74ddf30770039f1a7aa36a5eda6ef5100cddd9ad5fda350b
```

> **NOTE**
>
> On RHEL and CentOS-based systems, you will need to add the following line to the `docker run` command shown here: `--volume=/cgroup:/cgroup \`.

Once you have done this, you will be able to navigate to your Docker host on port 8080 to see the cAdvisor web interface (i.e., *http://172.17.42.10:8080/*) and the various detailed charts it has for the host and individual containers (see Figure 6-1).

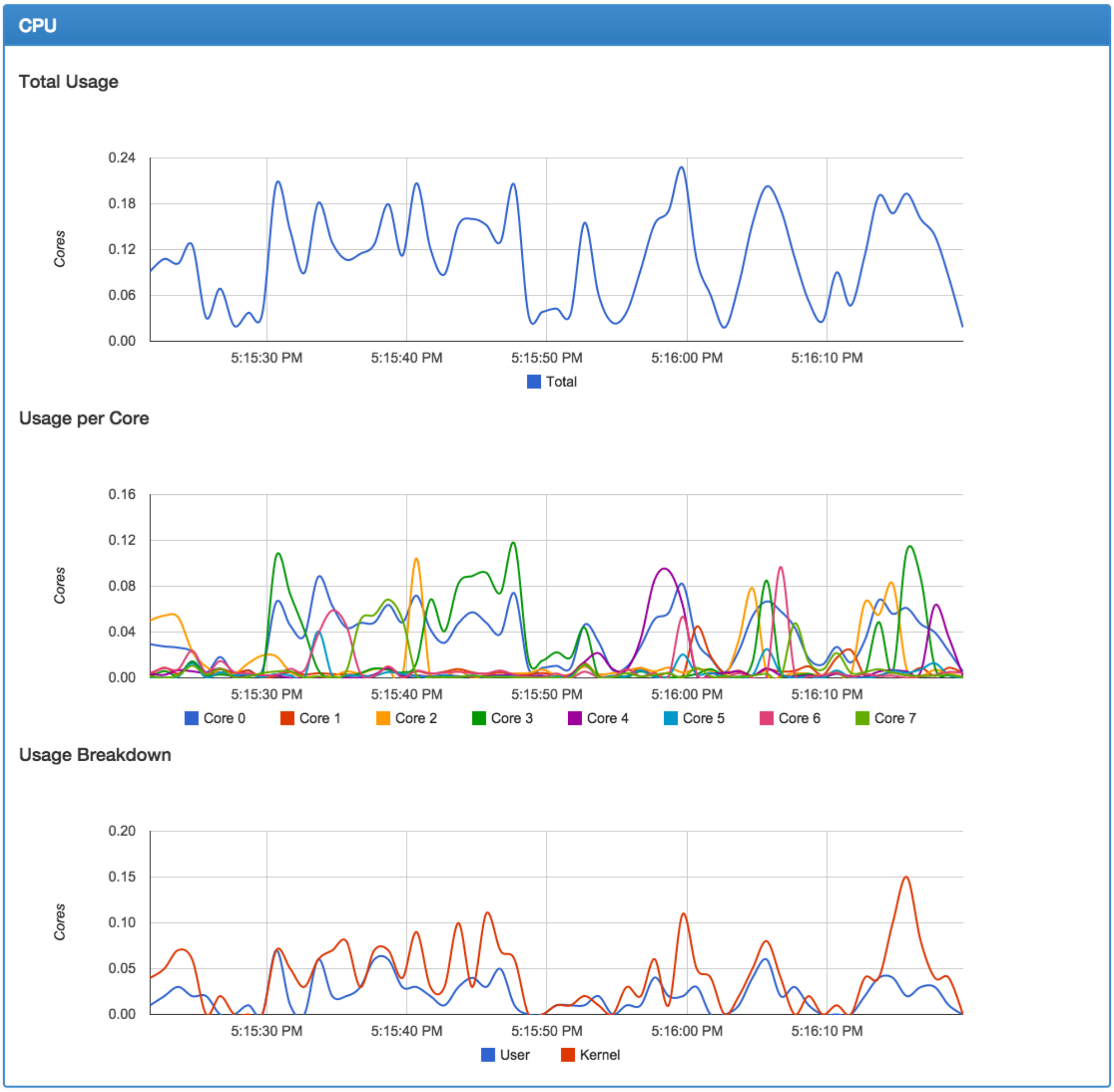


Figure 6-1. cAdvisor CPU graphs

cAdvisor provides a REST API endpoint, which can easily be queried for detailed information by your monitoring systems:

```
$ curl http://172.17.42.10:8080/api/v1.3/containers/
```

```
{
  "name": "/",
  "subcontainers": [
    {
      "name": "/docker"
    }
  ],
  "spec": {
    "creation_time": "2015-04-05T00:05:40.249999996Z",
    "has_cpu": true,
    "cpu": {
      "limit": 1024,
      "max_limit": 0,
      "mask": "0-7"
    },
    "has_memory": true,
    "memory": {
      "limit": 2095771648,
      "swap_limit": 1073737728
    },
    "has_network": true,
    "has_filesystem": true,
    "has_diskio": true
  },
  "stats": [
    {
      "timestamp": "2015-04-05T00:26:50.679218419Z",
      "cpu": {
        "usage": {
          "total": 123375166639,
          "per_cpu_usage": [
            41967365270,
...
            9229283688
          ],
          "user": 22990000000,
          "system": 43890000000
        },
        "load_average": 0
      },
      "diskio": {},
      "memory": {
        "usage": 1910095872,
        "working_set": 1025523712,
        "container_data": {
...
        },
        "hierarchical_data": {
...
```

```
        }
      },
      "network": {
...
      },
      "filesystem": [
        {
          "device": "/dev/sda1",
          "capacity": 19507089408,
          "usage": 2070806528,
...
        }
      ],
      "task_stats": {
        "nr_sleeping": 0,
        "nr_running": 0,
        "nr_stopped": 0,
        "nr_uninterruptible": 0,
        "nr_io_wait": 0
      }
    },
    ...
    }
  ]
}
```

As you can see from the preformatted output, the amount of detail provided here should be sufficient for many of your graphing and monitoring needs.

Prometheus Monitoring

The Prometheus monitoring system has become a popular solution for monitoring distributed systems. It works largely on a pull model, where it reaches out and gathers statistics from endpoints on a timed basis. Docker has an endpoint that was built for Prometheus and makes it really easy to integrate your container stats into a Prometheus monitoring system. At the time of this writing, the endpoint is currently experimental and not enabled in the `dockerd` server by default. This feature seems pretty popular, and we suspect it will become a permanent feature of the daemon shortly. It’s up to you if you want to use it as it stands. Our brief experience with it shows that it seems to work well, and it’s a pretty slick solution, as we’ll show you. We should point out that this solution is for monitoring the `dockerd` server, in contrast to the other solutions, which exposed information about the containers.

In order to export metrics to Prometheus, we need to reconfigure the `dockerd` server to enable the experimental features, and additionally to expose the metrics listener on a port of our choice. This is nice, because we don’t have to expose the whole Docker API on a TCP listener in order to get metrics out of the system—a security win at the expense of a little more configuration. To do that, we can either provide the `--experimental` and `--metrics-addr=` options on the command line, or we can put them into the *daemon.json* file that the daemon uses to configure itself. Because many current distributions run `systemd` and changing configurations there is highly dependent on your installation, we'll use the *daemon.json* option since it's more portable. We'll demonstrate this on Ubuntu Linux 16.04 LTS. On this distribution, the file is usually not present to begin with. So let's put one there using your favorite editor.

TIP

As previously mentioned, the *daemon.json* file for Docker Community Edition can be edited in Preferences → Daemon → Advanced from the UI. If you change this file, you will need to restart Docker Community Edition or the `dockerd` daemon.

Paste in the following, removing anything else that might be in there:

```
{
  "experimental": true,
  "metrics-addr": "0.0.0.0:9323"
}
```

You should now have a file that contains only what you just pasted and nothing else.

WARNING

Any time you make a service available on the network, you need to consider what security risks you might introduce. We believe the benefit of making metrics available is worth the tradeoff, but you should think through the repercussions in your scenario. For example, making them available on the public internet is probably not a good idea in most cases.

When we restart Docker we'll now have a listener on all addresses on port 9323. That's where we'll have Prometheus connect to get our metrics. But first we need to restart the `dockerd` server and test out the endpoint:

```
$ systemctl restart docker
$ curl -s http://localhost:9323/metrics | head -15
```

```
# HELP builder_builds_failed_total Number of failed image builds
# TYPE builder_builds_failed_total counter
builder_builds_failed_total{reason="build_canceled"} 0
builder_builds_failed_total{reason="build_target_not_reachable_error"} 0
builder_builds_failed_total{reason="command_not_supported_error"} 0
builder_builds_failed_total{reason="dockerfile_empty_error"} 0
builder_builds_failed_total{reason="dockerfile_syntax_error"} 0
builder_builds_failed_total{reason="error_processing_commands_error"} 0
builder_builds_failed_total{reason="missing_onbuild_arguments_error"} 0
builder_builds_failed_total{reason="unknown_instruction_error"} 0
# HELP builder_builds_triggered_total Number of triggered image builds
# TYPE builder_builds_triggered_total counter
builder_builds_triggered_total 0
# HELP engine_daemon_container_actions_seconds The number of seconds it ta
# process each container action
# TYPE engine_daemon_container_actions_seconds histogram
```

You should not get any errors returned from the `systemctl` call in the first line. If you do, you may have done something wrong when writing the *daemon.json* file. You should get some output from the `curl` command that looks something like what we show here. It might not be identical, and that's OK as long as you get something that is not error messages.

So now we have a place where Prometheus could get to our statistics. But we need to have Prometheus running somewhere, right? We can easily do that by spinning up a container. But first we need to write a simple config. We'll put it in */tmp/prometheus/prometheus.yaml*. You can either use the `cat` trick we used before, or use your favorite editor to put the following into the file:

```
# Scrape metrics every 5 seconds and name the monitor 'stats-monitor'
global:
  scrape_interval: 5s
  external_labels:
    monitor: 'stats-monitor'

# We're going to name our job 'DockerStats' and we'll connect to the d
ocker0
# bridge address to get the stats. If your docker0 has a different IP
address
# then use that instead. 127.0.0.1 and localhost will not work.
scrape_configs:
  - job_name: 'DockerStats'
    static_configs:
    - targets: ['172.17.0.1:9323']
```

NOTE

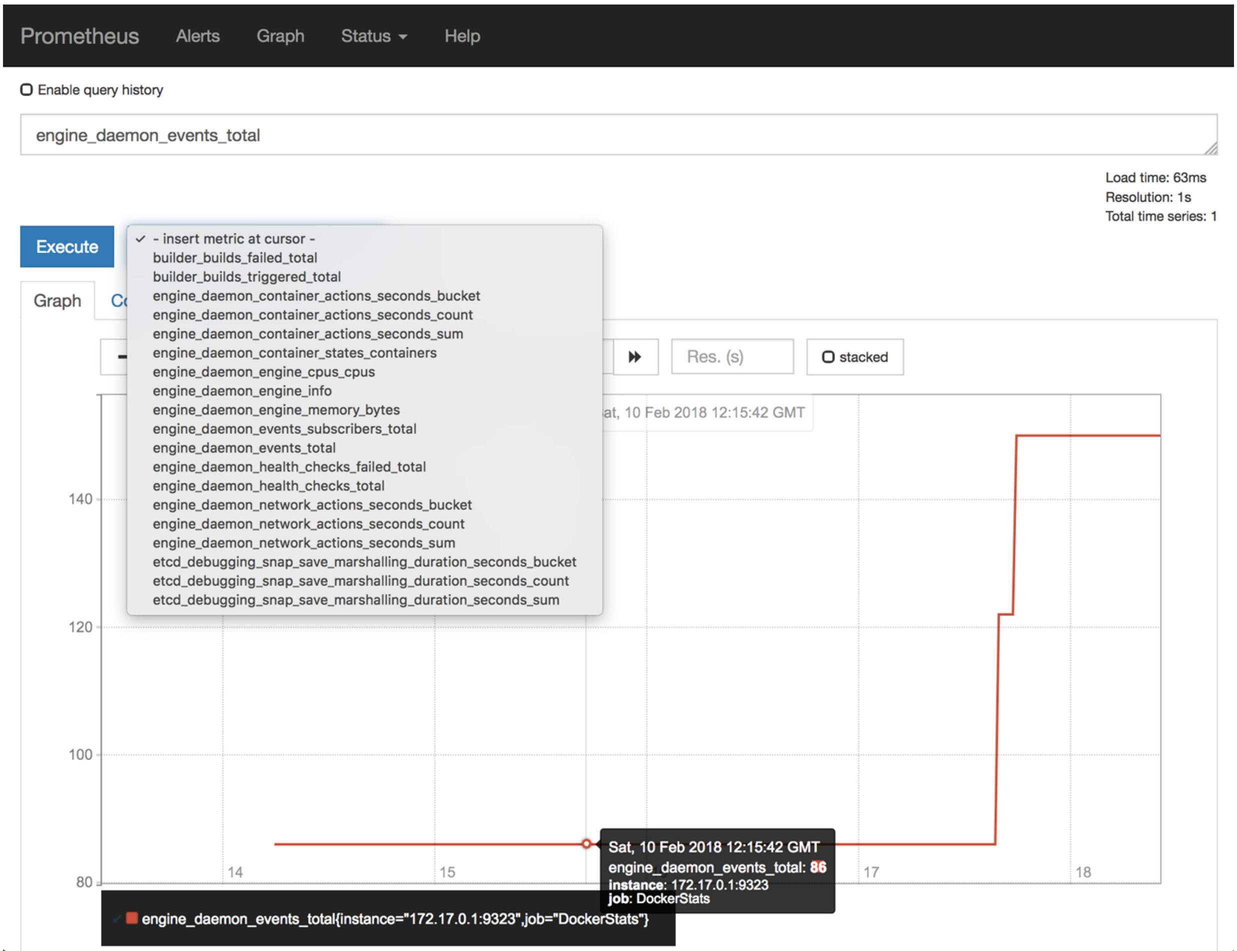
In Docker Community Edition for the Mac and Windows, you can use `host.docker.internal:9323` in place of the `172.17.0.1:9323` shown here.

As noted in the file, you should use the IP address of your `docker0` bridge here, or the IP address of your `ens3` or `eth0` interface since *localhost* and `127.0.0.1` are not routable from the container. The address we used here is the usual default for `docker0`, so it's probably the right one for you.

Now that we've written that out, we need to start up the container using this config:

```
$ docker run -d -p 9090:9090 \
    -v /tmp/prometheus/prometheus.yaml:/etc/prometheus.yaml \
    prom/prometheus --config.file=/etc/prometheus.yaml
```

That will run the container and volume mount the config file we made into the container so that it will find the settings we need it to have to monitor our Docker endpoint. If it starts up cleanly, you should now be able to open your browser and navigate to port 9090 on your Docker host. There you will get a Prometheus window something like Figure 6-2.


Prometheus
Alerts
Graph
Status
Help
Execute
Graph
Res. (s)
stacked
✓ - insert metric at cursor -
builder_builds_failed_total
builder_builds_triggered_total
engine_daemon_container_actions_seconds_bucket
engine_daemon_container_actions_seconds_count
engine_daemon_container_actions_seconds_sum
engine_daemon_container_states_containers
engine_daemon_engine_cpus_cpus
engine_daemon_engine_info
engine_daemon_engine_memory_bytes
engine_daemon_events_subscribers_total
engine_daemon_events_total
engine_daemon_health_checks_failed_total
engine_daemon_health_checks_total
engine_daemon_network_actions_seconds_bucket
engine_daemon_network_actions_seconds_count
engine_daemon_network_actions_seconds_sum
etcd_debugging_snap_save_marshalling_duration_seconds_bucket
etcd_debugging_snap_save_marshalling_duration_seconds_count
etcd_debugging_snap_save_marshalling_duration_seconds_sum
at, 10 Feb 2018 12:15:42 GMT
140
120
100
80
14
15
17
18

○ Enable query history

engine_daemon_events_total

Load time: 63ms
Resolution: 1s
Total time series: 1

Sat, 10 Feb 2018 12:15:42 GMT
engine_daemon_events_total: 86
instance: 172.17.0.1:9323
job: DockerStats

🟥 engine_daemon_events_total{instance="172.17.0.1:9323",job="DockerStats"}

Figure 6-2. Prometheus event graph

Here we’ve selected one of the metrics, the `engine_daemon_events_total`, and graphed it over a short period. You can easily query any of the other metrics in the drop-down. Further work and exploration with Prometheus would allow you to define alerts and alerting policies based on these metrics as well. And you don’t need to stop with monitoring the `dockerd` server. You can also expose metrics for Prometheus from your applications. If you’re intrigued and want to look at something more advanced, you might take a look at DockProm, which leverages Grafana to make nice dashboards and also queries your container metrics like those in the Docker API `/stats` endpoint.

Exploration

That gives you all the basics you need to start running containers. It’s probably worth downloading a container or two from the Docker Hub registry and exploring a bit on your own to get used to the commands we just learned. There are many other things you can do with Docker, including but not limited to:

- Copying files in and out of the container with `docker cp`
- Saving a container’s filesystem to a tarball with `docker export`

- Saving an image to a tarball with `docker save`

- Loading an image from a tarball with `docker import`

Docker has a huge feature set that you will likely grow into over time. Each new release adds more functionality as well. We'll get into a lot more detail later on about many of the other commands and features, but keep in mind that Docker's whole feature set is very large.

Wrap-Up

In the next chapter we will dive into some of the more technical details about how Docker works and how you can use this knowledge to debug your containerized applications.

Chapter 7. Debugging Containers

Once you've shipped an application to production, there will come a day when it's not working as expected. It's always nice to know ahead of time what to expect when that day comes. It's also important to have a good understanding of debugging containers before moving on to more complex deployments. Without debugging skills, it will be difficult to see where orchestration systems have gone wrong. So let's take a look at debugging containers.

In the end, debugging a containerized application is not all that different from debugging a normal process on a system except that the tools are somewhat different. Docker provides some pretty nice tooling to help you out! Some of these map to regular system tools, and some go further.

It is also critical to understand that your application is not running in a separate system from the other Docker processes. They share a kernel, likely a filesystem, and depending on your container configuration, they may share network interfaces. That means you can get a lot of information about what your container is doing.

If you're used to debugging applications in a virtual machine environment, you might think you would need to enter the container to inspect in detail an application's memory or CPU use, or to debug its system calls. Not so! Despite feeling in many ways like a virtualization layer, processes in containers are just processes on the Docker host itself. If you want to see a process list across all of the Docker containers on a machine, you can just run `ps` with your favorite command-line options right on the server, for example. By using the `docker top` command, you can see the process list as your container understands it. Let's take a look at some of the things that you can do when debugging a containerized application that don't require the use of `docker exec` and `nsenter`.

Process Output

One of the first things you'll want to know when debugging a container is what is running inside it. Docker has a built-in command for doing that: `docker top`. This is nice because it works even from remote hosts because it is exposed over the Docker Remote API. This is not the only way to see what’s going on inside a container, but it is the easiest to use. Let’s see how that works:

```
$ docker ps

CONTAINER ID   IMAGE         COMMAND    ...  NAMES
106ead0d55af   test:latest  /bin/bash  ...  clever_hypatia

$ docker top 106ead0d55af

UID        PID    PPID    C  STIME  TTY TIME     CMD
root       4548   1033    0  13:29  ?   00:00:00 /bin/sh -c nginx
root       4592   4548    0  13:29  ?   00:00:00 nginx: master process ngi
www-data   4593   4592    0  13:29  ?   00:00:00 nginx: worker process
www-data   4594   4592    0  13:29  ?   00:00:00 nginx: worker process
www-data   4595   4592    0  13:29  ?   00:00:00 nginx: worker process
www-data   4596   4592    0  13:29  ?   00:00:00 nginx: worker process
```

To run `docker top`, we need to pass it the ID of our container, so we get that from `docker ps`. We then pass that to `docker top` and get a nice listing of what is running inside our container, ordered by PID just as we’d expect from Linux `ps` output.

There are some oddities here, though. The primary one is namespacing of user IDs and filesystems.

For example, a user might exist in a container’s */etc/passwd* directory, but that same user might not exist simultaneously on the host machine, or have an entirely different user ID on the host machine versus inside the container. In the case where that user is running a process in a container, the `ps` output on the host machine will show a numeric ID rather than a username. In some cases, two containers might have users squatting on the same numeric ID, or mapping to an ID that is a completely different user on the host system. For example, if `haproxy` were installed on the host system, it would be possible for `nginx` in the container to appear to be running as the `haproxy` user on the host.

Let’s look at a more concrete example. Let’s consider a production Docker server running CentOS 7 and a container running on it that has an Ubuntu distribution inside. If you ran the following commands on the CentOS host, you would see that UID 7 is named `halt`:

```
$ id 7

uid=7(halt) gid=0(root) groups=0(root)
```

NOTE

There is nothing special about the UID number we are using here. You don’t need to take any particular note of it. It was chosen simply because it is used by default on both platforms but represents a different username.

If we then enter the standard Ubuntu container on that Docker host, you will see that UID 7 is set to `lp` in */etc/passwd*. By running the following commands, you can see that the container has a completely different perspective of who UID 7 is:

```
$ docker run -i -t ubuntu:latest /bin/bash

root@f86f8e528b92:/# grep x:7: /etc/passwd
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin

root@f86f8e528b92:/# id lp

uid=7(lp) gid=7(lp) groups=7(lp)
root@409c2a8216b1:/# exit
```

If we then run `ps au` on the Docker host while that container is running as UID 7 (`-u 7`), we would see that the Docker host would show the container process as being run by `halt` instead of `lp`:

```
$ docker run -d -u 7 ubuntu:latest sleep 1000

5525...06c6

$ ps ua | grep sleep

 1185 halt     sleep 1000
 1192 root     grep sleep
```

This could be particularly confusing if a well-known user like `nagios` or `postgres` were configured on the host system but not in the container, yet the container ran its process with the same ID. This namespacing can make the `ps` output look quite strange. It might, for example, look like the `nagios` user on your Docker host is running the `postgresql` daemon that was launched inside a container, if you don’t pay close attention.

TIP

One solution to this is to dedicate a nonzero UID to your containers. On your Docker hosts, you can create a `container` user as UID 5000 and then create the same user in your base container images. If you then run all your containers as UID 5000 (`-u 5000`), not only will you improve the security of your system by not running container processes as UID 0, but you will also make the `ps` output on the Docker host easier to decipher by displaying the `container` user for all of your running container processes. Some systems use the `nobody` or `daemon` user for the same purpose, but we prefer `container` for clarity. There is a little more detail about how this works in "Namespaces".

Likewise, because the process has a different view of the filesystem, paths that are shown in the `ps` output are relative to the container and not the host. In these cases, knowing it is in a container is a big win.

So that's how you use the Docker tooling to look at what's running in a container. But that's not the only way, and in a debugging situation, it might not be the best way. If you hop onto a Docker server and run a normal Linux `ps` to see what's running, you get a full list of everything containerized and not containerized just as if they were all equivalent processes. There are some ways to look at the process output to make things a lot clearer. For example, you can facilitate debugging by looking at the Linux `ps` output in tree form so that you can see all of the processes descended from Docker. Here's what that might look like when you use the BSD command-line flags; we'll chop the output to just the part we care about:

```
$ ps axlfww

...  /usr/bin/dockerd -H fd://
...  \_ docker-containerd -l unix:///var/run/docker/l
...  |   \_ docker-containerd-shim b668353c3af5d62350
...  |   |   \_ /usr/bin/cadvisor -logtostderr
...  |   \_ docker-containerd-shim dd72ecf1c9e4c22bf7
...  |       \_ /bin/s6-svscan /etc/services
...  |           \_ s6-supervise nginx.svc
...  |               \_ ./nginx
...  \_ /usr/bin/docker-proxy -proto tcp -host-ip 0.0
...  \_ /usr/bin/docker-proxy -proto tcp -host-ip 0.0
```

NOTE

Many of the ps commands in the preceding example work only on Linux distributions with the full ps command. Some stripped-down versions of Linux run the Busybox shell, which does not have full ps support and won't show some of this output. We recommend running a full distribution on your host systems like CoreOS, Ubuntu, or CentOS.

Here you can see that we're running one Docker daemon and two instances of the docker-proxy, which we will discuss in more detail in "Network Inspection". There is one containerd running, which is the main container runtime inside Docker. Everything else under those processes represents Docker containers and processes inside them. In this example, we have two containers. They show up as docker-containerd-shim followed by the container ID. In this case, we are running one instance of Google's cadvisor, and one instance of nginx in another container, being supervised by the S6 supervisor. Each container has a related docker-proxy process that is used to map the required network ports between the container and the host Docker server.

Because of the tree output from ps, it's pretty clear how they are related to each other, and we know they're running in a container because they are in dockerd's process tree. If you're a bigger fan of Unix SysV command-line flags, you can get a similar, but not as nice-looking, tree output with ps -ejH:

```
$ ps -ejH

40643 ...   docker
43689 ...     docker
43697 ...     docker
43702 ...     start
43716 ...       java
46970 ...     docker
46976 ...     supervisord
46990 ...       supervisor_remo
46991 ...       supervisor_stdo
46992 ...       nginx
47030 ...         nginx
47031 ...         nginx
47032 ...         nginx
47033 ...         nginx
46993 ...       ruby
47041 ...         ruby
47044 ...         ruby
```

You can get a more concise view of the `docker` process tree by using the `pstree` command. Here, we'll use `pidof` to scope it to the tree belonging to `docker`:

```
$ pstree `pidof dockerd`

dockerd... docker-containe──cadvisor─┬─15*[{cadvisor}]
       ...          │                └─9*[{docker-containe}]
       ...          └─docker-containe─┬─s─nginx───13*[{nginx}...]
       ...                             └─9*[{docker-containe}]
```

This doesn't show us PIDs and therefore is useful only for getting a sense of how things hang together in our containers. But this is pretty nice output when there are a lot of containers running on a host. It's far more concise and provides a nice high-level map of how things connect. Here we can see the same containers that were shown in the previous `ps` output, but the tree is collapsed so we get multipliers like `10*` when there are 10 duplicate processes.

We can actually get a full tree with PIDs if we run `pstree`, as shown here:

```
$ pstree -p `pidof dockerd`

dockerd(4086)... ─┬─dockerd(6529)─┬─{dockerd}(6530)
             ...  │               ├─...
             ...  │               └─{dockerd}(6535)
             ...  ├─...
             ...  ├─mongod(6675)─┬─{mongod}(6737)
             ...  │              ├─...
             ...  │              └─{mongod}(6756)
             ...  ├─redis-server(6537)─┬─{redis-server}(6576)
             ...  │                    └─{redis-server}(6577)
             ...  ├─{dockerd}(4089)
             ...  ├─...
             ...  └─{dockerd}(6738)
```

This output provides us with a very good look at all the processes attached to Docker and what they are running. It is, however, difficult to see the `docker-proxy` in this output, since it is really just another forked `dockerd` process.

Process Inspection

If you're logged in to the Docker server, you can inspect running processes in many of the same ways that you would on the host. Common debugging tools like `strace` work as expected. In the following code, we'll inspect a `unicorn` process running inside a Ruby web app container:

```
$ strace -p 31292

Process 31292 attached - interrupt to quit
select(11, [10], NULL, [7 8], {30, 103848}) = 1 (in [10], left {29, 176592
fcntl(10, F_GETFL)                          = 0x802 (flags O_RDWR|O_NONBLOCK)
accept4(10, 0x7fff25c17b40, [128], SOCK_CLOEXEC) = -1 EAGAIN (...)
getppid()                                   = 17
select(11, [10], NULL, [7 8], {45, 0})  = 1 (in [10], left {44, 762499})
fcntl(10, F_GETFL)                          = 0x802 (flags O_RDWR|O_NONBLOCK)
accept4(10, 0x7fff25c17b40, [128], SOCK_CLOEXEC) = -1 EAGAIN (...)
getppid()                                   = 17
```

You can see that we get the same output that we would from noncontainerized processes on the host. Likewise, an `lsof` shows us that the files and sockets open in a process work as expected:

```
$ lsof -p 31292

COMMAND ...  NAME
ruby    ...  /data/app
ruby    ...  /
ruby    ...  /usr/local/rbenv/versions/2.1.1/bin/ruby
ruby    ...  /usr/.../iso_8859_1.so (stat: No such file or directory)
ruby    ...  /usr/.../fiber.so (stat: No such file or directory)
ruby    ...  /usr/.../cparse.so (stat: No such file or directory)
ruby    ...  /usr/.../libsasl2.so.2.0.23 (path dev=253,0, inode=1443531)
ruby    ...  /lib64/libnspr4.so (path dev=253,0, inode=655717)
ruby    ...  /lib64/libplc4.so (path dev=253,0, inode=655718)
ruby    ...  /lib64/libplds4.so (path dev=253,0, inode=655719)
ruby    ...  /usr/lib64/libnssutil3.so (path dev=253,0, inode=1443529)
ruby    ...  /usr/lib64/libnss3.so (path dev=253,0, inode=1444999)
ruby    ...  /usr/lib64/libsmime3.so (path dev=253,0, inode=1445001)
ruby    ...  /usr/lib64/libssl3.so (path dev=253,0, inode=1445002)
ruby    ...  /lib64/liblber-2.4.so.2.5.6 (path dev=253,0, inode=655816)
ruby    ...  /lib64/libldap_r-2.4.so.2.5.6 (path dev=253,0, inode=655820)
```

Note that the paths to the files are all relative to the container's view of the backing filesystem, which is not the same as the host view. Therefore, the version of the file on the host will not match the one the container sees. In this case, it's probably best to enter the container using `docker exec` to look at the files with the same view that the processes inside it have.

It's possible to run the GNU debugger (`gdb`) and other process inspection tools in the same manner as long as you're `root` and have proper permissions to do so.

Controlling Processes

When you have a shell directly on the Docker server, you can, in many ways, treat containerized processes just like any other process running on the system. If you're remote, you might send signals with `docker kill` because it's expedient. But if you're already logged in to a Docker server for a debugging session or because the Docker daemon is not responding, you can just `kill` the process like you would any other.

Note that unless you kill the top-level process in the container (PID 1 inside the container), killing a process will not terminate the container itself. That *might* be desirable if you were killing a runaway process, but might leave the container in an unexpected state. Developers probably expect that all the processes are running if they can see their container in `docker ps` and it could also confuse a scheduler like Mesos or Kubernetes or any other system that is health-checking your application. Keep in mind that containers are supposed to look to the outside world like a single bundle. If you need to kill off something inside the container other than for debugging purposes, it's best to replace the whole container. Containers offer an abstraction that tools interoperate with. They expect the internals of the container to remain consistent.

Terminating processes is not the only reason to send signals. And since containerized processes are just normal processes in many respects, they can be passed the whole array of Unix signals listed in the manpage for the Linux `kill` command. Many Unix programs will perform special actions when they receive certain predefined signals. For example, `nginx` will reopen its logs when receiving a `SIGUSR1` signal. Using the Linux `kill` command, you can send any Unix signal to a container process on the local Docker server.

TIP

Unless you run an orchestrator like Kubernetes that can handle multiple containers in a larger abstraction like a pod, we consider it a best practice to run some kind of process control in your production containers. Whether it be `systemd`, `upstart`, `runit`, `s6`, or your own homegrown tools, this allows you to treat containers atomically even when they contain more than one process. You should, however, try very hard not to run more than one thing inside your container, to ensure that your container is scoped to handle one well-defined task and does not grown into a monolithic container.

But in either case, you want `docker ps` to reflect the presence of the whole container and don't want to worry if one of the processes inside it has died. If you can assume that the presence of a container and absence of error logs means that things are working, you can treat `docker ps` output as the truth about what's happening on your Docker systems. It also means any orchestration system you use can do the same.

It is also a good idea to ensure that you understand the complete behavior of your preferred process control service, including memory or disk utilization, Unix single handling, and so on, since this can impact your container's performance and behavior. Generally, the lightest-weight systems are the best.

Because containers work just like any other process, it's important to understand how they can interact with your application in a less helpful way. There are some special needs in a container for processes that spawn background children—that is, anything that forks and daemonizes so the parent no longer manages the child process lifecycle. Jenkins build containers are one common example where people see this go wrong. When daemons fork into the background, they become children of PID 1 on Unix systems. Process 1 is special and is usually an `init` process of some kind.

PID 1 is responsible for making sure that children are reaped. In your container, by default your main process will be PID 1. Since you probably won't be handling reaping of children from your application, you can end up with zombie processes in your container. There are a few solutions to this problem. The first is to run an init system in the container of your own choosing—one that is capable of handling PID 1 responsibilities. `s6`, `runit`, and others described in the preceding Note can be easily used inside the container.

But Docker itself provides an even simpler option that solves just this one case without taking on all the capabilities of a full init system. If you provide the `--init` flag to `docker run`, Docker will launch a very small init process based on the tini project that will act as PID 1 inside the container on startup. Whatever you specify in your *Dockerfile* as the `CMD` is passed to `tini` and otherwise works in the same way you would expect. It does, however, replace anything you might have in the `ENTRYPOINT` section of your *Dockerfile*.

When you launch a Docker container without the `--init` flag, you get something like this in your process list:

```
$ docker run -i -t alpine:3.6 sh
/ # ps -ef

PID   USER     TIME   COMMAND
    1 root       0:00 sh
    5 root       0:00 ps -ef

/ # exit
```

Notice that in this case, the `CMD` we launched is PID 1. That means it is responsible for child reaping. If we are launching a container where that is important, we can pass `--init` to make sure that when the parent process exits, children are reaped.

```
$ docker run -i -t --init alpine:3.6 sh
/ # ps -ef

PID   USER     TIME   COMMAND
    1 root       0:00 /dev/init -- sh
    5 root       0:00 sh
    6 root       0:00 ps -ef

/ # exit
```

Here, you can see that the PID 1 process is actually `/dev/init`. That has in turn launched the shell binary for us as specified on the command line. Because we now have an init system inside the container, the PID 1 responsibilities fall to it rather than the command we used to invoke the container. In most cases this is what you want. You may not need an init system, but it’s small enough that you should consider having at least `tini` inside your containers in production.

Network Inspection

Compared to process inspection, debugging containerized applications at the network level can be more complicated. Unlike traditional processes running on the host, Docker containers can be connected to the network in a number of ways. If you are running the default setup, as the vast majority of people are, then your containers are all connected to the network via the default bridge network that Docker creates. This is a virtual network where the host is the gateway to the rest of the world. We can inspect these virtual networks with the tooling that ships with Docker. You can get it to show you which networks exist by calling the `docker network ls` command:

```
$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
a4ea6aeb7503        bridge              bridge              local
b5b5fa6889b1        host                host                local
08b8b30a20da        none                null                local
```

Here we can see the default bridge network: the host network, which is for any containers running in `host` network mode, where containers share the same network namespace as the host; and the none network, which disables network access entirely for the container. If you run `docker-compose` or other orchestration tools, they may create additional networks here with different names.

But seeing which networks exist doesn't make it any easier to see what's on those networks. So, you can see which containers are attached to any particular named network with the `docker network inspect` command. This produces a fair amount of output. It shows you all of the containers that are attached to the network specified, and a number of details about the network itself. Let's take a look at the default bridge network:

```
$ docker network inspect bridge
```

```
[
    {
        "Name": "bridge",
        ...
        "Driver": "bridge",
        "EnableIPv6": false,
        ...
        "Containers": {
            "6a0f439...9a9c3": {
                "Name": "inspiring_johnson",
                ...
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            },
            "8720cc2...e91b5": {
                "Name": "zealous_keller",
                ...
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            ...
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            ...
        },
        "Labels": {}
    }
]
```

We've excluded some of the details here to shrink the output a bit. But what we can see is that there are two containers on the bridge network, and they are attached to the `docker0` bridge on the host. We can also see the IP addresses of each container, and the host network address they are bound to (`host_binding_ipv4`). This is useful when you are trying to understand the internal structure of the bridged network. Note that if you have containers on different networks, they may not have connectivity to each other, depending on how the networks were configured.

> **TIP**
>
> In general we recommend leaving your containers on the default bridge network unless you have a good reason not to, or are running `docker-compose` or a scheduler that constructs networks in its own manner. In addition, naming your containers in some identifiable way really helps here because we can't see the image information. The name and ID are the only reference we have in this output that can tie us back to the `docker ps` listing. Some schedulers don't do a good job of naming containers, which is too bad because it can be really helpful for debugging.

As we've seen, containers will normally have their own network stack and their own IP address, unless they are running in host networking mode, which we will discuss further in "Networking". But what about when we look at them from the host machine itself? Because containers have their own network and addresses, they won't show up in all `netstat` output on the host. But we know that the ports you map to your containers are bound on the host. Running `netstat -an` on the Docker server works as expected, as shown here:

```
$ sudo netstat -an

Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.0.3.1:53             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp6       0      0 :::23235                :::*                    LISTEN
tcp6       0      0 :::2375                 :::*                    LISTEN
tcp6       0      0 :::4243                 :::*                    LISTEN
tcp6       0      0 fe80::389a:46ff:fe92:53 :::*                    LISTEN
tcp6       0      0 :::22                   :::*                    LISTEN
udp        0      0 10.0.3.1:53             0.0.0.0:*
udp        0      0 0.0.0.0:67              0.0.0.0:*
udp        0      0 0.0.0.0:68              0.0.0.0:*
udp6       0      0 fe80::389a:46ff:fe92:53 :::*
```

Here we can see all of the interfaces that we're listening on. Our container is bound to port 23235 on IP address 0.0.0.0. That shows up. But what happens when we ask `netstat` to show us the process name that's bound to the port?

```
$ netstat -anp

Active Internet connections (servers and established)
Proto ... Local Address           Foreign Address State  PID/Program name
tcp   ... 10.0.3.1:53             0.0.0.0:*       LISTEN 23861/dnsmasq
tcp   ... 0.0.0.0:22              0.0.0.0:*       LISTEN 902/sshd
tcp6  ... :::23235                :::*            LISTEN 24053/docker-prox
tcp6  ... :::2375                 :::*            LISTEN 954/docker
tcp6  ... :::4243                 :::*            LISTEN 954/docker
tcp6  ... fe80::389a:46ff:fe92:53 :::*            LISTEN 23861/dnsmasq
tcp6  ... :::22                   :::*            LISTEN 902/sshd
udp   ... 10.0.3.1:53             0.0.0.0:*              23861/dnsmasq
udp   ... 0.0.0.0:67              0.0.0.0:*              23861/dnsmasq
udp   ... 0.0.0.0:68              0.0.0.0:*              880/dhclient3
udp6  ... fe80::389a:46ff:fe92:53 :::*                   23861/dnsmasq
```

We see the same output, but notice what is bound to the port: `docker-proxy`. That's because, in its default configuration, Docker actually has a proxy written in Go that sits between all of the containers and the outside world. That means that when we look at output, we see only `docker-proxy`. Notice that there is no clue here about which container that `docker-proxy` is handling. Luckily, `docker ps` shows us which containers are bound to which ports, so this isn't a big deal. But it's not necessarily expected, and you probably want to be aware of it before you're debugging a production failure. Still, passing the `p` flag to netstat is helpful in identifying which ports are tied to containers.

> **NOTE**
>
> If you’re using host networking in your container, then this layer is skipped. There is no docker-proxy, and the process in the container can bind to the port directly. It also shows up as a normal process in netstat -anp output.

Other network inspection commands work largely as expected, including tcpdump, but it’s important to remember that docker-proxy is there, in between the host’s network interface and the container, and that the containers have their own network interfaces on a virtual network.

Image History

When you’re building and deploying a single container, it’s easy to keep track of where it came from and what images it’s sitting on top of. But this rapidly becomes unmanageable when you’re shipping many containers with images that are built and maintained by different teams. How can you tell what layers are actually underneath the one your container is running on? Your container’s image version hopefully shows you which build you’re running of the application, but that doesn’t reveal anything about the images it’s built on. docker history does just that. You can see each layer that exists in the inspected image, the sizes of each layer, and the commands that were used to build it:

```
$ docker history redis:latest

IMAGE        CREATED     CREATED BY                                      SIZ
33c26d72bd74 3 weeks ago /bin/sh -c #(nop)  CMD ["redis-server"]         0B
<missing>    3 weeks ago /bin/sh -c #(nop)  EXPOSE 6379/tcp              0B
<missing>    3 weeks ago /bin/sh -c #(nop)  ENTRYPOINT ["docker-entry… 0B
<missing>    3 weeks ago /bin/sh -c #(nop) COPY file:9c29fbe8374a97f9… 344
<missing>    3 weeks ago /bin/sh -c #(nop) WORKDIR /data                 0B
<missing>    3 weeks ago /bin/sh -c #(nop)  VOLUME [/data]               0B
<missing>    3 weeks ago /bin/sh -c mkdir /data && chown redis:redis … 0B
<missing>    3 weeks ago /bin/sh -c set -ex;   buildDeps='   wget    … 24.
<missing>    3 weeks ago /bin/sh -c #(nop)  ENV REDIS_DOWNLOAD_SHA=ff… 0B
<missing>    3 weeks ago /bin/sh -c #(nop)  ENV REDIS_DOWNLOAD_URL=ht… 0B
<missing>    3 weeks ago /bin/sh -c #(nop)  ENV REDIS_VERSION=4.0.8     0B
<missing>    3 weeks ago /bin/sh -c set -ex;   fetchDeps='ca-certific… 3.1
<missing>    3 weeks ago /bin/sh -c #(nop)  ENV GOSU_VERSION=1.10       0B
<missing>    3 weeks ago /bin/sh -c groupadd -r redis && useradd -r -… 330
<missing>    3 weeks ago /bin/sh -c #(nop)  CMD ["bash"]                 0B
<missing>    3 weeks ago /bin/sh -c #(nop) ADD file:a0f72eb6710fe45af… 79.
```

Using `docker history` can be useful, for example, when you are trying to determine why the size of the final image is much larger than expected. The layers are listed in order, with the first one at the bottom of the list and the last one at the top.

Here we can see that the command output has been truncated in a few cases. For long commands, adding the `--no-trunc` option to the preceding command will let you see the complete command that was used to build each layer.

Inspecting a Container

In Chapter 4, we showed you how to read the `docker inspect` output to see how a container is configured. But underneath that is a directory on the host's disk that is dedicated to the container. Usually this is */var/lib/docker/containers*. If you look at that directory, it contains very long SHA hashes, as shown here:

```
$ ls /var/lib/docker/containers

106ead0d55af55bd803334090664e4bc821c76dadf231e1aab7798d1baa19121
28970c706db0f69716af43527ed926acbd82581e1cef5e4e6ff152fce1b79972
3c4f916619a5dfc420396d823b42e8bd30a2f94ab5b0f42f052357a68a67309b
589f2ad301381b7704c9cade7da6b34046ef69ebe3d6929b9bc24785d7488287
959db1611d632dc27a86efcb66f1c6268d948d6f22e81e2a22a57610b5070b4d
a1e15f197ea0996d31f69c332f2b14e18b727e53735133a230d54657ac6aa5dd
bad35aac3f503121abf0e543e697fcade78f0d30124778915764d85fb10303a7
bc8c72c965ebca7db9a2b816188773a5864aa381b81c3073b9d3e52e977c55ba
daa75fb108a33793a3f8fcef7ba65589e124af66bc52c4a070f645fffbbc498e
e2ac800b58c4c72e240b90068402b7d4734a7dd03402ee2bce3248cc6f44d676
e8085ebc102b5f51c13cc5c257acb2274e7f8d1645af7baad0cb6fe8eef36e24
f8e46faa3303d93fc424e289d09b4ffba1fc7782b9878456e0fe11f1f6814e4b
```

That's a bit daunting. But those are just the container IDs in long form. If you want to look at the configuration for a particular container, you just need to use `docker ps` to find its short ID, and then find the directory that matches:

```
$ docker ps

CONTAINER ID        IMAGE                             COMMAND
8720cc2f0502        alpine:3.6   "/bin/sh -c nginx"     ...
```

You can view the short ID from `docker ps`, then match it to the `ls /var/lib/docker/containers` output to see that you want the directory beginning with `8720cc2f0502`. Command-line tab completion is helpful here. If you need exact matching, you can do a `docker inspect 8720cc2f0502` and grab the long ID from the output. This directory contains some pretty interesting files related to the container:

```
$ cd /var/lib/docker/containers/\
8720cc2f05021643fb1f78710ec5ebc245ca42d3c64dd4e73dd2534226be91b5

$ ls -la

total 36
drwx------  4 root root 4096 Dec 28 04:26 .
drwx------ 35 root root 4096 Dec 28 15:32 ..
-rw-r-----  1 root root    0 Dec 28 04:26 8720cc2f05021643fb1f78710ec5ebc.
drwx------  2 root root 4096 Dec 28 04:26 checkpoints
-rw-------  1 root root 2620 Dec 28 04:26 config.v2.json
-rw-r--r--  1 root root 1153 Dec 28 04:26 hostconfig.json
-rw-r--r--  1 root root   13 Dec 28 04:26 hostname
-rw-r--r--  1 root root  174 Dec 28 04:26 hosts
-rw-r--r--  1 root root  194 Dec 28 04:26 resolv.conf
-rw-r--r--  1 root root   71 Dec 28 04:26 resolv.conf.hash
drwxrwxrwt  2 root root   40 Dec 28 04:26 shm
```

As we discussed in Chapter 5, this directory contains some files that are bind-mounted directly into your container, like *hosts*, *resolv.conf*, and *hostname*. If you are running the default logging mechanism, then this directory is also where Docker stores the JSON file containing the log that is shown with the `docker logs` command, the JSON configuration that backs the `docker inspect` output (*config.v2.json*), and the networking configuration for the container (*hostconfig.json*). The *resolv.conf.hash* file is used by Docker to determine when the container's file has diverged from the current one on the host so it can be updated.

This directory can also be really helpful in the event of severe failure. Even if we're not able to enter the container, or if `docker` is not responding, we can look at how the container was configured. It's also pretty useful to understand where those files are mounted from inside the container. Keep in mind that it's not a good idea to modify these files. Docker expects them to contain reality, and if you alter that reality, you're asking for trouble. But it's another avenue for information on what's happening in your container.

Filesystem Inspection

Docker, regardless of the backend actually in use, has a layered filesystem that allows it to track the changes in any given container. This is how the images are actually assembled when you do a build, but it is also useful when you're trying to figure out if a Docker container has changed anything and, if so, what. A common problem with Dockerized applications is that they continue to write things into the filesystem. Normally you don't want your containers to do that, to the extent possible, and it can be helpful for debugging to figure out if they have been writing into the container. Sometimes this is helpful in turning up stray logfiles that exist in the container as well. As with most of the core tools, this kind of inspection is built into the `docker` command-line tooling and is also exposed via the API. Let's take a look at what this shows us. We'll assume that we already have the ID of the container we're concerned with.

```
$ sudo docker diff 89b8e19707df

C /var/log/redis
A /var/log/redis/redis.log
C /var/run
A /var/run/cron.reboot
A /var/run/crond.pid
C /var/lib/logrotate.status
C /var/lib/redis
A /var/lib/redis/dump.rdb
C /var/spool/cron
A /var/spool/cron/root
```

Each line begins with either A or C, which are just shorthand for added or changed, respectively. We can see that this container is running `redis`, that the `redis` log is being written to, and that someone or something has been changing the `crontab` for `root`. Logging to the local filesystem is not a good idea, especially for anything with high-volume logs. Being able to find out what is writing to your Docker filesystem can really help you understand where things are filling up, or give you a preview of what would be added if you were to build an image from it.

Further detailed inspection requires jumping into the container with `docker exec` or `nsenter` and the like, in order to see what exactly is in the filesystem. But `docker diff` gives you a good place to start.

Wrap-Up

At this point, you should have a good idea how to deploy and debug individual containers in development and production, but how do you start to scale this for larger application ecosystems? In the next chapter we'll take a look at one of the simpler Docker orchestration tools: Docker Compose. This tool is a nice bridge between a single Docker container and a production orchestration system. It delivers a lot of value in development environments and throughout the DevOps pipeline.

Chapter 8. Exploring Docker Compose

At this point you should have a good feel for the `docker` command and how to use it to build, launch, monitor, and debug your applications. Once you are comfortable working with individual containers it won’t be long before you’ll want to share your projects and start building more complex projects that require multiple containers to function properly. This is particularly the case in development environments, where having a whole stack of containers running can easily simulate many production environments on your local machine.

If you’re running a whole stack of containers, however, every container needs to be run with the proper setup to ensure that the underlying application is configured correctly and will run as expected. Getting these settings correct each and every time can be challenging, especially when you are not the person who originally wrote the application. To help with this during development, people often resort to writing shell scripts that can build and run their containers in a consistent manner. Although this works, it can become difficult to understand for a newcomer and hard to maintain as the project changes over time. It’s also not necessarily repeatable between projects.

To help address this problem, Docker, Inc. released a tool primarily aimed at developers called Docker Compose. This tool is included with Docker Community Edition, but you can also install it by following the online installation directions.

Docker Compose is an incredibly useful tool that can streamline all sorts of development tasks that have traditionally been very cumbersome and error-prone. It can easily be leveraged to help developers quickly spin up complicated application stacks, compile applications without the need for setting up complex local development environments, and much more.

In this chapter, we’ll do a run-through of how to use Compose to its best advantage. We’ll be using a GitHub repository in all of the following examples. If you want to run the examples as we go through them, you should run the following command to download the code, if you didn’t already do that in Chapter 6:

```
$ git clone https://github.com/spkane/rocketchat-hubot-demo.git \
    --config core.autocrlf=input
```

NOTE

In the example shell script and *docker-compose.yaml* files below, some lines have been truncated to fit in the margins. Make sure that you use the files from the preceding git repository, if you plan to try these examples yourselves.

This repository contains the configuration we'll need to launch a complete web service that includes a MongoDB datastore, the open source RocketChat communications server, a Hubot chatops bot, and a `zmachine-api` instance for a little surprise entertainment value.

Configuring Docker Compose

Before we dive into using the `docker-compose` command, it is useful to see the kind of ad hoc tooling it replaces. So, let's take a moment to look at a shell script that could be used to build and deploy a local copy of our service for development and local testing via Docker. This output is long and detailed, but it's important in order to prove the point about why Docker Compose is a huge leap over shell scripting.

```
#!/bin/bash

set -e
set -u

if [ $# -ne 0 ] && [ ${1} == "down" ]; then
  docker rm -f hubot || true
  docker rm -f zmachine || true
  docker rm -f rocketchat || true
  docker rm -f mongo-init-replica || true
  docker rm -f mongo || true
  docker network rm botnet || true
  echo "Environment torn down..."
  exit 0
fi

# Global Settings
export PORT="3000"
export ROOT_URL="http://127.0.0.1:3000"
export MONGO_URL="mongodb://mongo:27017/rocketchat"
export MONGO_OPLOG_URL="mongodb://mongo:27017/local"
export MAIL_URL="smtp://smtp.email"
export RESPOND_TO_DM="true"
export HUBOT_ALIAS=". "
export LISTEN_ON_ALL_PUBLIC="true"
export ROCKETCHAT_AUTH="password"
export ROCKETCHAT_URL="rocketchat:3000"
export ROCKETCHAT_ROOM=""
export ROCKETCHAT_USER="hubot"
export ROCKETCHAT_PASSWORD="HughTheBot"
export BOT_NAME="bot"
export EXTERNAL_SCRIPTS="hubot-help,hubot-diagnostics,hubot-zmachine"
export HUBOT_ZMACHINE_SERVER="http://zmachine:80"
export HUBOT_ZMACHINE_ROOMS="zmachine"
export HUBOT_ZMACHINE_OT_PREFIX="ot"

docker build -t spkane/mongo:3.2 ./mongodb/docker

docker push spkane/mongo:3.2
docker pull spkane/zmachine-api:latest
docker pull rocketchat/rocket.chat:0.61.0
docker pull rocketchat/hubot-rocketchat:latest

docker rm -f hubot || true
docker rm -f zmachine || true
docker rm -f rocketchat || true
docker rm -f mongo-init-replica || true
docker rm -f mongo || true

docker network rm botnet || true

docker network create -d bridge botnet

docker run -d \
  --name=mongo \
  --network=botnet \
  --restart unless-stopped \
  -v $(pwd)/mongodb/data/db:/data/db \
  spkane/mongo:3.2 \
  mongod --smallfiles --oplogSize 128 --replSet rs0
sleep 5
docker run -d \
  --name=mongo-init-replica \
  --network=botnet \
  spkane/mongo:3.2 \
  'mongo mongo/rocketchat --eval "rs.initiate({ ..."'
sleep 5
docker run -d \
  --name=rocketchat \
  --network=botnet \
  --restart unless-stopped  \
  -v $(pwd)/rocketchat/data/uploads:/app/uploads \
  -p 3000:3000 \
  -e PORT=${PORT} \
  -e ROOT_URL=${ROOT_URL} \
  -e MONGO_URL=${MONGO_URL} \
  -e MONGO_OPLOG_URL=${MONGO_OPLOG_URL} \
  -e MAIL_URL=${MAIL_URL} \
  rocketchat/rocket.chat:0.61.0
docker run -d \
  --name=zmachine \
  --network=botnet \
  --restart unless-stopped  \
  -v $(pwd)/zmachine/saves:/root/saves \
  -v $(pwd)/zmachine/zcode:/root/zcode \
  -p 3002:80 \
  spkane/zmachine-api:latest
docker run -d \
  --name=hubot \
  --network=botnet \
  --restart unless-stopped  \
  -v $(pwd)/hubot/scripts:/home/hubot/scripts \
  -p 3001:8080 \
  -e RESPOND_TO_DM="true" \
  -e HUBOT_ALIAS=". " \
  -e LISTEN_ON_ALL_PUBLIC="true" \
  -e ROCKETCHAT_AUTH="password" \
  -e ROCKETCHAT_URL="rocketchat:3000" \
  -e ROCKETCHAT_ROOM="" \
  -e ROCKETCHAT_USER="hubot" \
  -e ROCKETCHAT_PASSWORD="HughTheBot" \
  -e BOT_NAME="bot" \
  -e EXTERNAL_SCRIPTS="hubot-help,hubot-diagnostics,hubot-zmachine" \
  -e HUBOT_ZMACHINE_SERVER="http://zmachine:80" \
  -e HUBOT_ZMACHINE_ROOMS="zmachine" \
  -e HUBOT_ZMACHINE_OT_PREFIX="ot" \
  rocketchat/hubot-rocketchat:latest
echo "Environment setup..."
exit 0
```

At this point you can probably follow most of this script pretty easily. As you may already have noticed, this is a hassle to read, is not very flexible, will be a pain to edit, and might fail in several places in an unexpected way. If we were to follow shell script best practices and handle all the possible errors here in an effort to guarantee it was repeatable, it would also be two to three times as long as it already is. Without a lot of work extracting common functionality for the error handling, you'd also have to rewrite much of that logic every time you have a new project like this. It's not a very good way to approach a process that you need to work every time. This is where good tooling comes in. You can accomplish exactly the same thing with Docker Compose, while also making it much more repeatable and easier to read, understand, and maintain.

In contrast to this messy shell script, which is very repetitive and prone to easily breaking, Docker Compose is typically configured with a single, declarative YAML file for each project, called *docker-compose.yaml*. This configuration file is very easy to read and will work in a very repeatable fashion, so that each user has the same experience when they run it. Here you can see an example *docker-compose.yaml* file that could be used to replace the preceding brittle shell script:

```yaml
version: '2'
services:
  mongo:
    build:
      context: ../../mongodb/docker
    image: spkane/mongo:3.2
    restart: unless-stopped
    command: mongod --smallfiles --oplogSize 128 --replSet rs0
    volumes:
      - "../../mongodb/data/db:/data/db"
    networks:
      - botnet
  mongo-init-replica:
    image: spkane/mongo:3.2
    command: 'mongo mongo/rocketchat --eval "rs.initiate({ ..."'
    depends_on:
      - mongo
    networks:
      - botnet
  rocketchat:
    image: rocketchat/rocket.chat:0.61.0
    restart: unless-stopped
    volumes:
      - "../../rocketchat/data/uploads:/app/uploads"
    environment:
      PORT: 3000
      ROOT_URL: "http://127.0.0.1:3000"
      MONGO_URL: "mongodb://mongo:27017/rocketchat"
      MONGO_OPLOG_URL: "mongodb://mongo:27017/local"
      MAIL_URL: "smtp://smtp.email"
    depends_on:
      - mongo
    ports:
      - 3000:3000
    networks:
      - botnet
  zmachine:
    image: spkane/zmachine-api:latest
    restart: unless-stopped
    volumes:
      - "../../zmachine/saves:/root/saves"
      - "../../zmachine/zcode:/root/zcode"
    depends_on:
      - rocketchat
    expose:
      - "80"
    networks:
      - botnet
  hubot:
    image: rocketchat/hubot-rocketchat:latest
    restart: unless-stopped
    volumes:
      - "../../hubot/scripts:/home/hubot/scripts"
    environment:
      RESPOND_TO_DM: "true"
      HUBOT_ALIAS: ". "
      LISTEN_ON_ALL_PUBLIC: "true"
      ROCKETCHAT_AUTH: "password"
      ROCKETCHAT_URL: "rocketchat:3000"
      ROCKETCHAT_ROOM: ""
      ROCKETCHAT_USER: "hubot"
      ROCKETCHAT_PASSWORD: "HughTheBot"
      BOT_NAME: "bot"
      EXTERNAL_SCRIPTS: "hubot-help,hubot-diagnostics,hubot-zmachine"
      HUBOT_ZMACHINE_SERVER: "http://zmachine:80"
      HUBOT_ZMACHINE_ROOMS: "zmachine"
      HUBOT_ZMACHINE_OT_PREFIX: "ot"
    depends_on:
      - zmachine
    ports:
      - 3001:8080
    networks:
      - botnet
networks:
  botnet:
    driver: bridge
```

The *docker-compose.yaml* file makes it easy to describe all the important requirements for each of your services and how they need to communicate with each other. And we get a lot of validation and checking logic for free that we didn’t even have time to write into our shell script and which we’d probably get wrong on occasion, no matter how careful we were.

So, what did we tell Compose to do in that YAML file? The first line of our file simply tells Docker Compose what version of the Compose configuration language this file was designed for.

```
version: '2'
```

The rest of our document is divided into two sections: `services` and `networks`.

For starters, let’s take a quick look at the `networks` section. In this *docker-compose.yaml* file, we are defining a single named Docker network.

```
networks:
  botnet:
    driver: bridge
```

This is a very simple configuration that tells Docker Compose to create a single network, named `botnet`, using the (default) bridge driver, which will bridge the Docker network with the host's networking stack.

The `services` section is the most important part of the configuration and tells Docker Compose what applications you want to launch. Here the `services` section defines five services: `mongo`, `mongo-init-replica`, `rocketchat`, `zmachine`, and `hubot`. Each named service then contains sections that tell Docker how to build, configure, and launch that service.

If you take a look at the `mongo` service, you will see that the first subsection is called `build` and contains a `context` key. This informs Docker Compose that it can build this image and that the files required for the build are located in the *../../mongodb/docker* directory, which is two levels above the directory containing the *docker-compose.yaml* file.

```
    build:
      context: ../../mongodb/docker
```

The next setting, `image`, defines the image tag that you want either to apply to your build or to download (if you're not building an image) and then run.

```
    image: spkane/mongo:3.2
```

With the `restart` option, you tell Docker when you want it to restart your containers. In most cases, you'll want Docker to restart your containers any time that you have not specifically stopped them.

```
    restart: unless-stopped
```

The `command` option allows you to define the command that your container should run at startup. For this service, we'll start MongoDB with a few command-line arguments.

```
    command: mongod --smallfiles --oplogSize 128 --replSet rs0
```

A lot of services have at least some data that should be persisted during development, despite the ephemeral nature of containers. To accomplish this, it is easiest to mount a local directory into the containers. The `volumes` section allows you to list all the local directories that you would like to have mounted into a container, and define where they go.

```
    volumes:
      - "../../mongodb/data/db:/data/db"
```

The final subsection for the mongo service, networks, tells Docker Compose which network this container should be attached to.

```
    networks:
      - botnet
```

At this point, let’s jump down to the rocketchat service. This service does not have a build subsection. It only defines an image without a build line, which tells Docker Compose that it cannot build this image and must instead try to pull and launch a standard preexisting Docker image with the defined tag.

The first new subsection that you will notice in this service is called environment. This is where you can define any environment variables that you want to pass into your container.

Note that the value for the MONGO_URL does not use an IP address or fully qualified domain name. This is because all of these services are running on the same Docker network and Docker configures each container so that it can find the others via their service names. This means that we can easily configure URLs like this to simply point at the service name and internal port for the container we need to connect to. And, if we rearrange things later on, these names will continue to point to the right container in our stack. They are also nice because they make it quite explicit to the reader what the dependency is for this container.

```
    environment:
      PORT: 3000
      ROOT_URL: "http://127.0.0.1:3000"
      MONGO_URL: "mongodb://mongo:27017/rocketchat"
      MONGO_OPLOG_URL: "mongodb://mongo:27017/local"
      MAIL_URL: "smtp://smtp.email"
```

TIP

The *docker-compose.yaml* file can also refer to environment variables using the `${<VARIABLE_NAME>}` format, which makes it possible to pull in secrets without actually storing them in this file. Newer versions of Docker Compose also support an .env file, which can be very useful for handling secrets and environment variables that change between developers, for example.

The `depends_on` section defines a container that must be running before this container can be started. Normally `docker-compose` ensures only that the container is running, not that it is actually healthy; however, you can use health-check functionality in both Docker and Docker Compose, if needed.

```
    depends_on:
      - mongo
```

NOTE

We discuss Docker's health-check functionality in more detail in "Container Health Checks". You can also find more information in the documentation for Docker and Docker Compose.

The `ports` subsection allows you to define all the ports that you want mapped from the container to the host.

```
    ports:
      - 3000:3000
```

The `zmachine` service uses only one new subsection, called `expose`. This section allows us to tell Docker that we want to expose this port to the other containers on the Docker network, but not to the underlying host. This is why you do not provide a host port to map this port to.

```
    expose:
      - "80"
```

You might notice at this point that, while we expose a port for `zmachine`, we didn't expose a port in the `mongo` service. It wouldn't have hurt anything to expose the `mongo` port, but we didn't need to because it is already exposed by the upstream mongo *Dockerfile*. This is sometimes a little opaque. `docker history` on the built image can be helpful here.

Here we've used an example that is complex enough to expose you to some of the power of Docker Compose, but it is by no means exhaustive. There is a great deal else that you can configure in a *docker-compose.yaml* file, including security settings, resource quotas, and much more. You can find a lot of detailed information about configuration for Compose in the official Docker Compose documentation.

Launching Services

We configured a set of services for our application in the YAML file. That tells Compose what we're going to launch and how to configure it. So, let's get it up and running! To run our first Docker Compose command, we need to be sure that we are in the same directory as the *docker-compose.yaml* file that is appropriate for our operating system.

- Unix/OS X users run:

```
$ cd rocketchat-hubot-demo/compose/unix
```

- Windows users run:

```
PS C:\> cd rocketchat-hubot-demo\compose\windows
```

The only difference between these two *docker-compose.yaml* files is that the Windows version does not mount a volume for the MongoDB datastore. This is because MongoDB cannot use the Windows filesystem to store its data properly. The primary result of this is that when you delete the container using a command like `docker-compose down`, all of the data in the MongoDB instance will be lost.

Once you are in the appropriate directory, you can confirm that the configuration is correct by running:

```
$ docker-compose config
```

If everything is fine, the command will print out your configuration file. If there is a problem, the command will print an error with details about the problem, like so:

```
ERROR: The Compose file './docker-compose.yaml' is invalid because:
Unsupported config option for services.hubot: 'networkss'
```

You can build any containers that you need by using the `build` option. Any services that use images will be skipped.

```
$ docker-compose build
Building mongo
...
Successfully built a7b0d2b7b1b9
Successfully tagged spkane/mongo:3.2
mongo-init-replica uses an image, skipping
rocketchat uses an image, skipping
zmachine uses an image, skipping
hubot uses an image, skipping
```

You can start up your web service in the background by running the following command:

```
$ docker-compose up -d
Creating network "unix_botnet" with driver "bridge"
Creating unix_mongo-init-replica_1 ... done
Creating unix_rocketchat_1         ...
Creating unix_zmachine_1           ... done
Creating unix_zmachine_1           ...
Creating unix_hubot_1              ... done
```

NOTE

Docker Compose prefixes the network and container names with the name of the directory that contains your *docker-compose.yaml* file. So, on Windows you will see `windows_botnet`, `windows_mongo-init-replica_1`, and so on, in your output instead of what we show here.

Windows users: When you first bring up the services, Windows will most likely prompt you to authorize vpnkit, and Docker for Windows will prompt you to share your disk. You must click both the “Allow access” and the “Share it” buttons for the network and volume shares to work and everything to come up properly.

Once everything comes up, we can take a quick look at the logs for all of the services:

```
$ docker-compose logs
...
rocketchat_1           | → |              Platform: linux                |
mongo_1                | 2018-02-05T01:05:40.200+0000 I REPL     [Repli ...
rocketchat_1           | → |           Process Port: 3000                |
mongo_1                | 2018-02-05T01:05:40.200+0000 I REPL     [Repli ...
rocketchat_1           | → |              Site URL: http://127.0.0.1:3000  |
hubot_1                | npm info preuninstall hubot-diagnostics@0.0.2
mongo_1                | 2018-02-05T01:05:40.200+0000 I REPL     [Repli ...
rocketchat_1           | → |      ReplicaSet OpLog: Enabled              |
...
```

You can't see it in print here, but if you're following along note that all of the logs are color-coded by service and interlaced by the time Docker received the loglines. This makes it a lot easier to follow what's happening, even though there are a number of services logging at once.

At this point, we have successfully launched a reasonably complex application that makes up a stack of containers. We'll take a look at this simple application now so that you can see what we built and get a more complete understanding of the Compose tooling. While this next section does not strictly have anything to do with Docker itself, it is intended to show you how easy it is to use Docker Compose to set up complex and fully functioning web services.

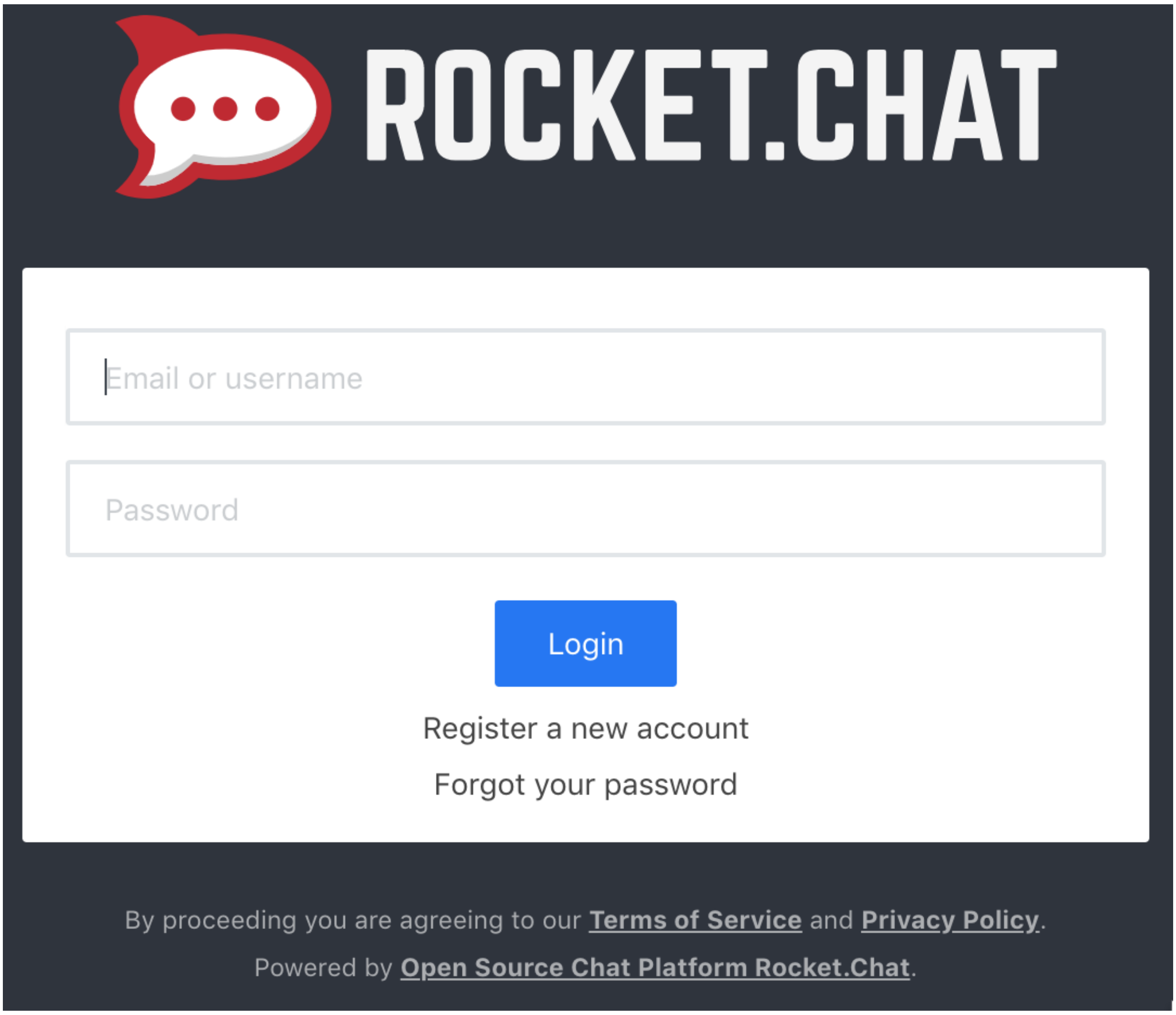
Exploring RocketChat

NOTE

In this section we're going to diverge from Docker for a moment and take a look at RocketChat. We'll spend a few pages on it, so that you know enough about it that you can hopefully start to appreciate how much easier it is to set up a complex environment using Docker Compose. Feel free to skip down to "Exercising Docker Compose", if you would like.

We'll shortly dig further into what's happening under the covers of our setup. But in order to do that effectively, we should now take a brief moment to explore the application stack we built. The application we launched with Docker Compose, RocketChat, is an open source chat client/server application. To see how it works, let's launch a web browser and navigate to *http://127.0.0.1:3000*.

When you get there, you are prompted with a login screen for RocketChat (Figure 8-1).


ROCKET.CHAT

Email or username

Password

By proceeding you are agreeing to our Terms of Service and Privacy Policy.

Powered by Open Source Chat Platform Rocket.Chat.

Figure 8-1. RocketChat login screen

Click the "Register a new account" link and fill out the following form:

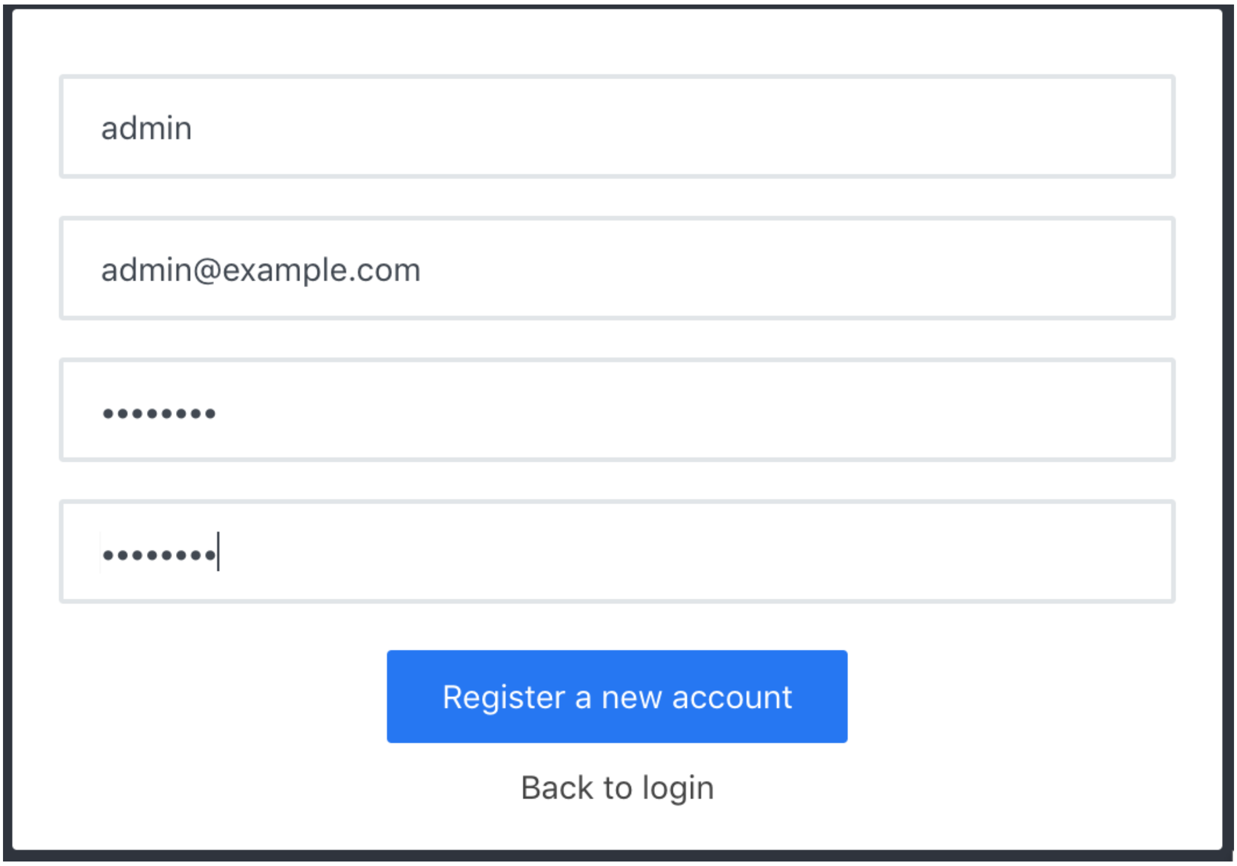
Name: **`admin`**

Email: **`admin@example.com`**

Password: **`p2ssw0rd`**

Re-enter Password: **`p2ssw0rd`**

Then click the large blue button, labeled "Register a new account" (Figure 8-2).



admin

admin@example.com

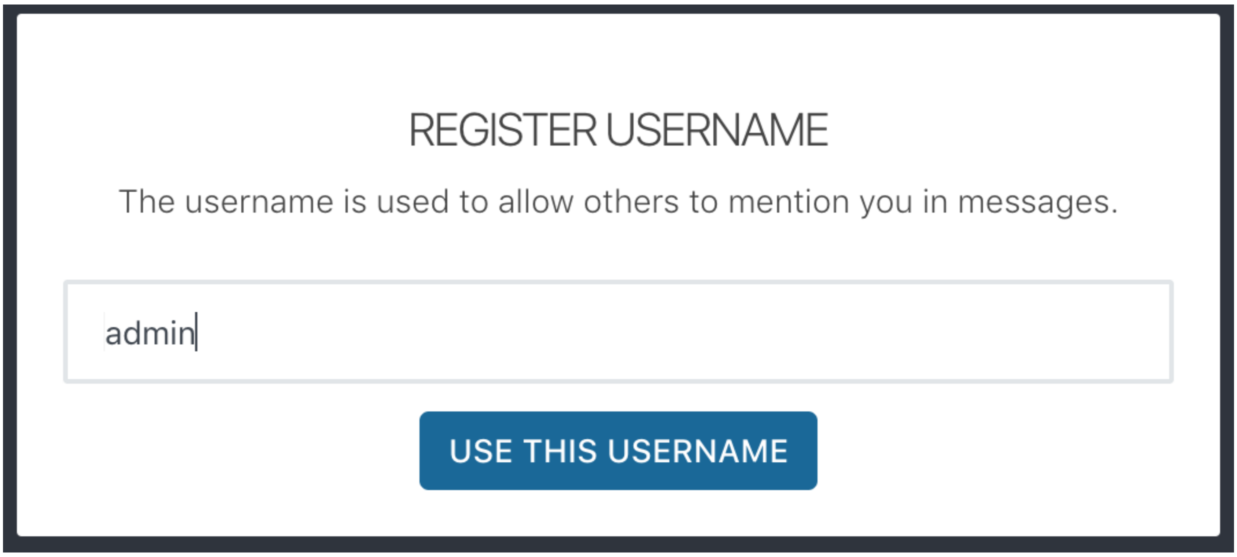
••••••••

••••••••

Figure 8-2. RocketChat registration screen

You now have to wait. The account setup will take at least a minute, while the server sets everything up for the first time. Try to resist clicking the blue button that now says “Please wait…”. It won’t really hurt anything, but it won’t help speed anything up either.

Once everything is set up, you will be prompted by a dialog window entitled “Register Username.” You can leave the username as is, unless you want to change it, and simply click the button labeled “Use This Username” (Figure 8-3).



REGISTER USERNAME

The username is used to allow others to mention you in messages.

admin

Figure 8-3. RocketChat username registration screen

Congratulations—you are now logged into a fully functional chat client, but you aren't done yet. The Docker Compose configuration launched an instance of a hubot chat assistant and the mysterious zmachine, so let's take a look at those.

Since the RocketChat server is brand new, it does not have a user yet that our bot can use. Let's remedy that.

Start by clicking the top of the left sidebar, where you see your username and status displayed. This makes a menu pop up, within which you should be able to click "Administration" (Figure 8-4).

> **WARNING**
>
> During testing, we noticed that some web browsers or operating systems did not always show the smaller icons in the RocketChat UI. They were clickable, but it was hard to tell that there was anything there. If you are seeing this, the buttons still work and moving your mouse around slowly should show some of the buttons' tool tips.

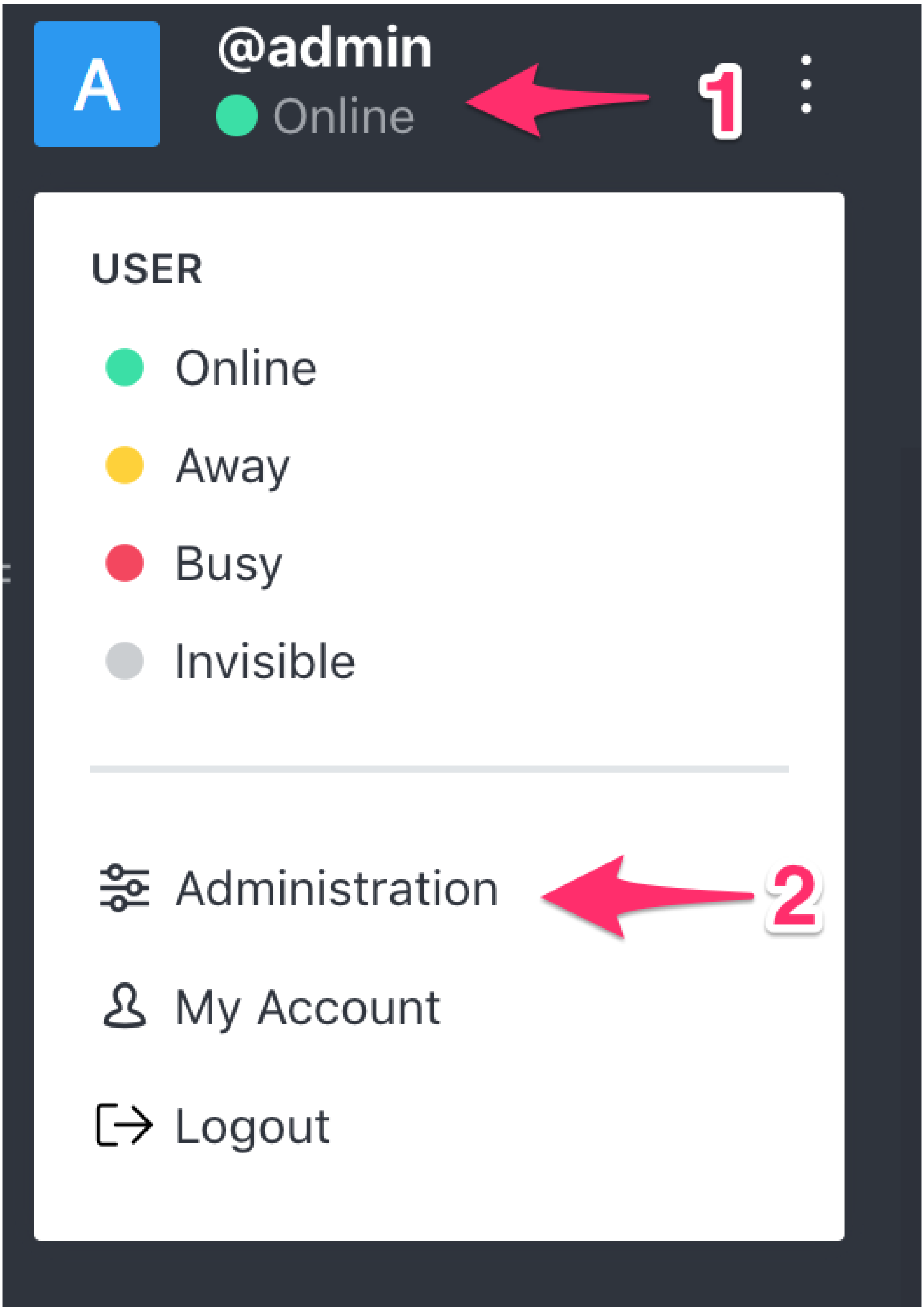
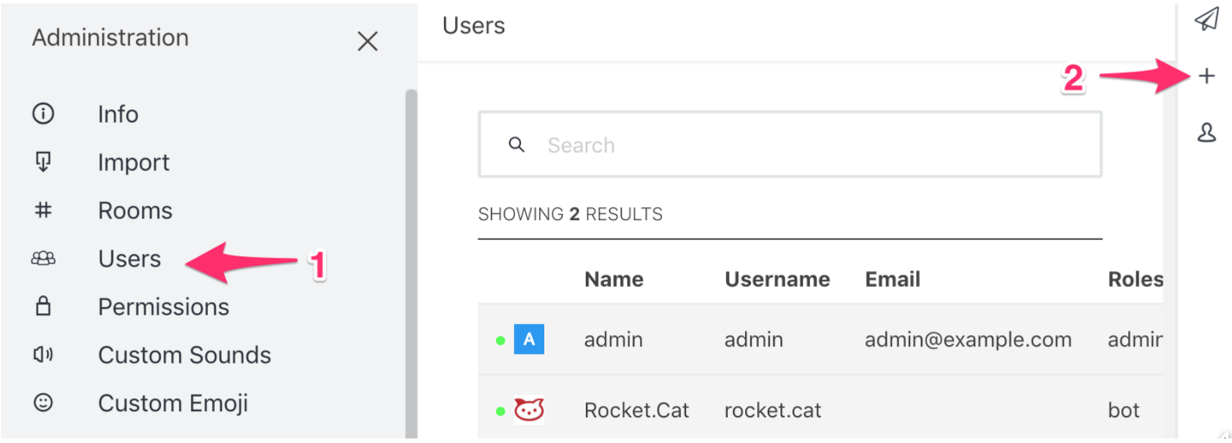


Figure 8-4. RocketChat administration sidebar

The left sidebar is now replaced by the Administration panel. In the Administration panel, click "Users" (Figure 8-5).



SHOWING 2 RESULTS

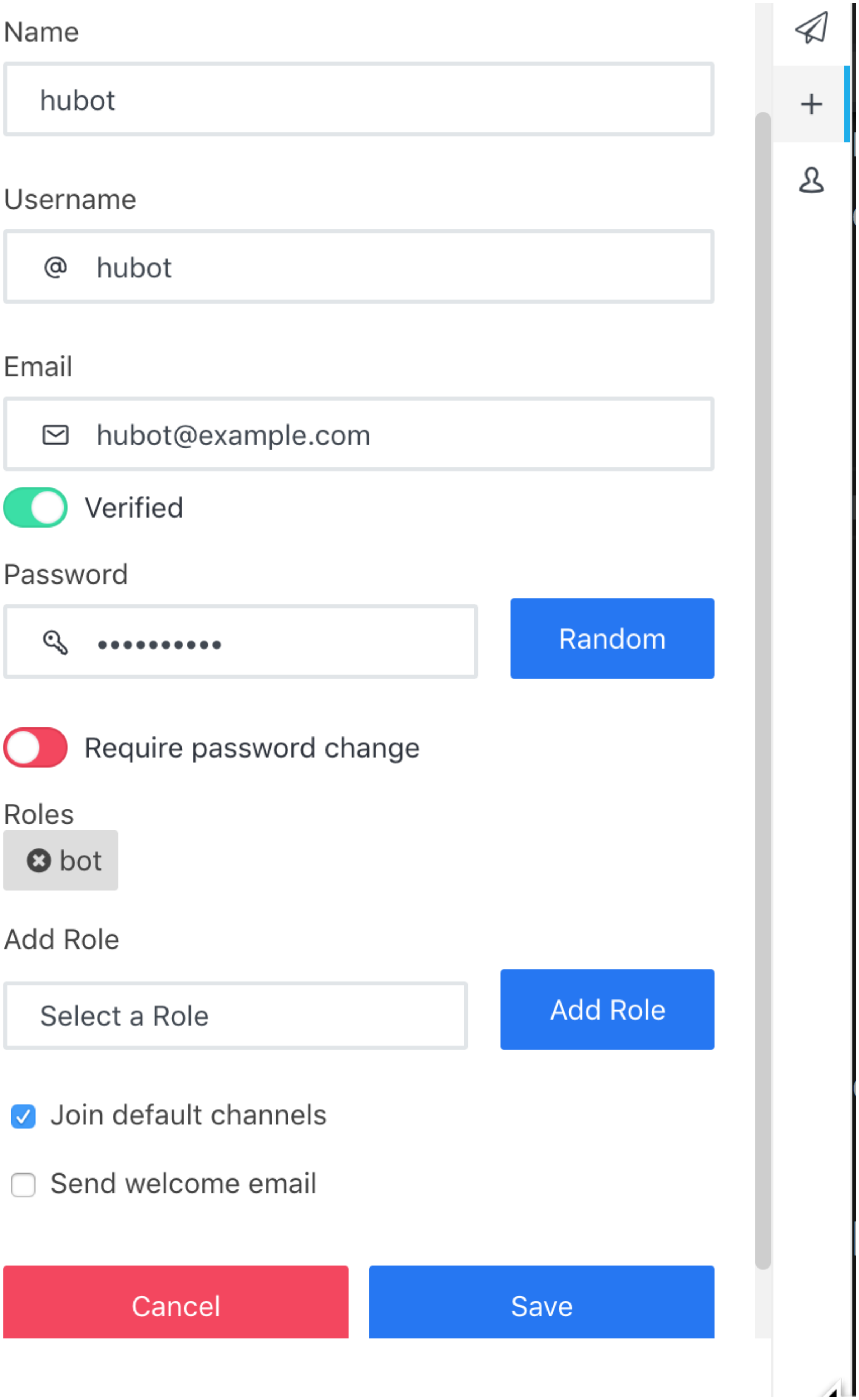
	Name	Username	Email	Roles
A	admin	admin	admin@example.com	admir
	Rocket.Cat	rocket.cat		bot

Figure 8-5. RocketChat Add User screen

On the far right side of the screen, click the + symbol to add a user, and then fill out the form as follows:

- Name: **`hubot`**
- Username: **`hubot`**
- Email: **`hubot@example.com`**
- Click: **`Verified`** **(Green)**
- Password: **`HughTheBot`**
- Click: **`Require password change`** **(Red)**
- Selec a_role_: **`bot`**
- Click: **`Add Role`**
- Uncheck: **`Send welcome email`**

Click “Save” at the bottom to create the user (Figure 8-6).



Name

hubot

Username

@ hubot

Email

hubot@example.com

Verified

Password

••••••••••

Require password change

Roles

bot

Add Role

Select a Role

☑ Join default channels

□ Send welcome email

Figure 8-6. RocketChat Create Bot User screen

At the top of the left side Administration panel, click the X to close the panel (Figure 8-7).

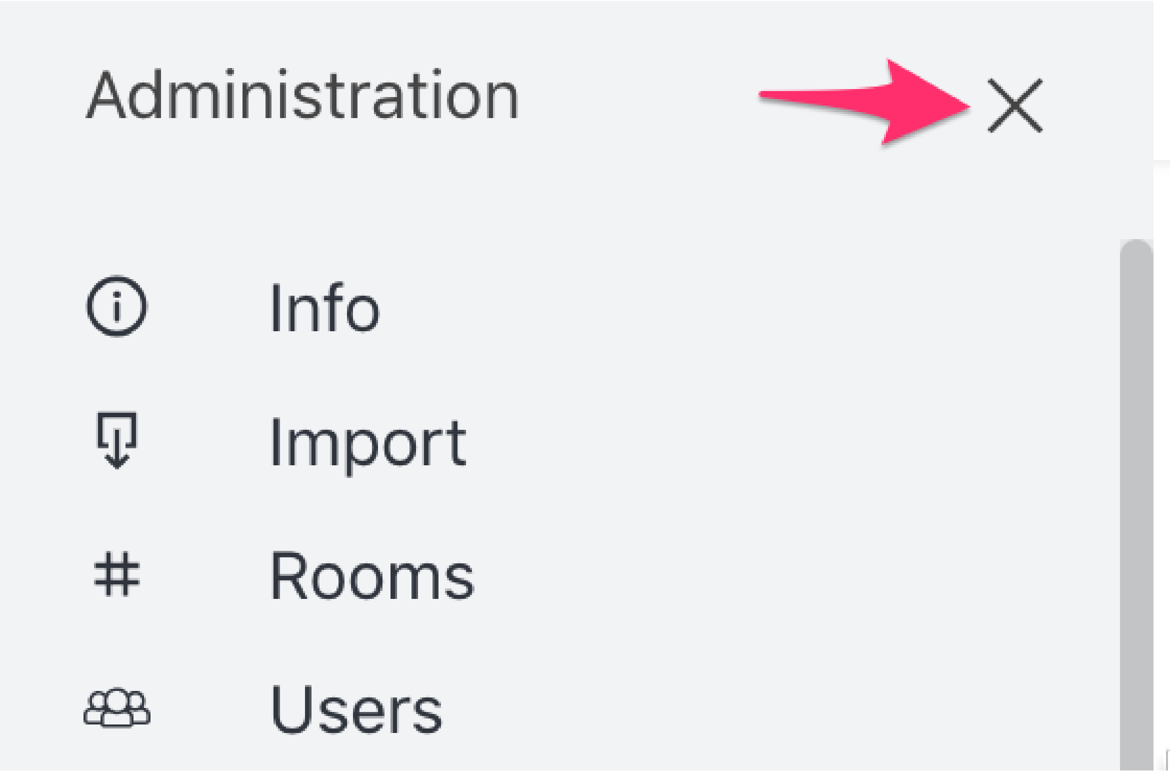
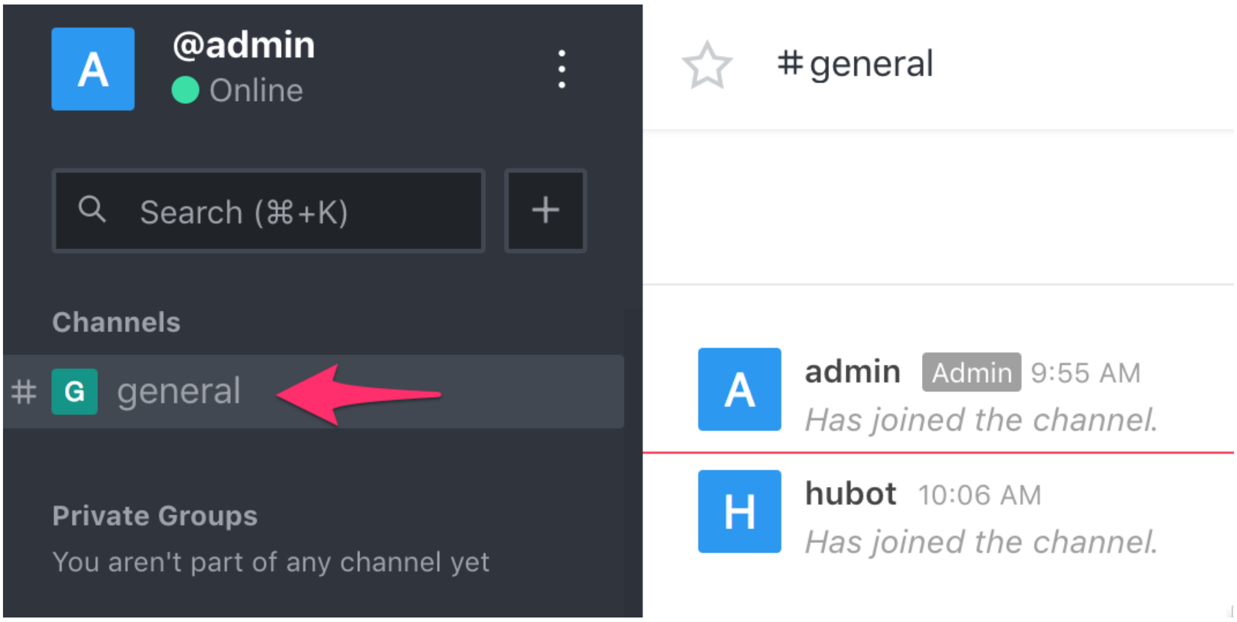


Figure 8-7. RocketChat Close Administration panel

In the left side panel under channels, click “general” (Figure 8-8).



admin Admin 9:55 AM
Has joined the channel.

hubot 10:06 AM
Has joined the channel.

Figure 8-8. RocketChat General channel

And finally, let’s try to get hubot to connect to the chat server. The easiest way to accomplish this is to use a terminal and tell Docker Compose to restart the hubot container.

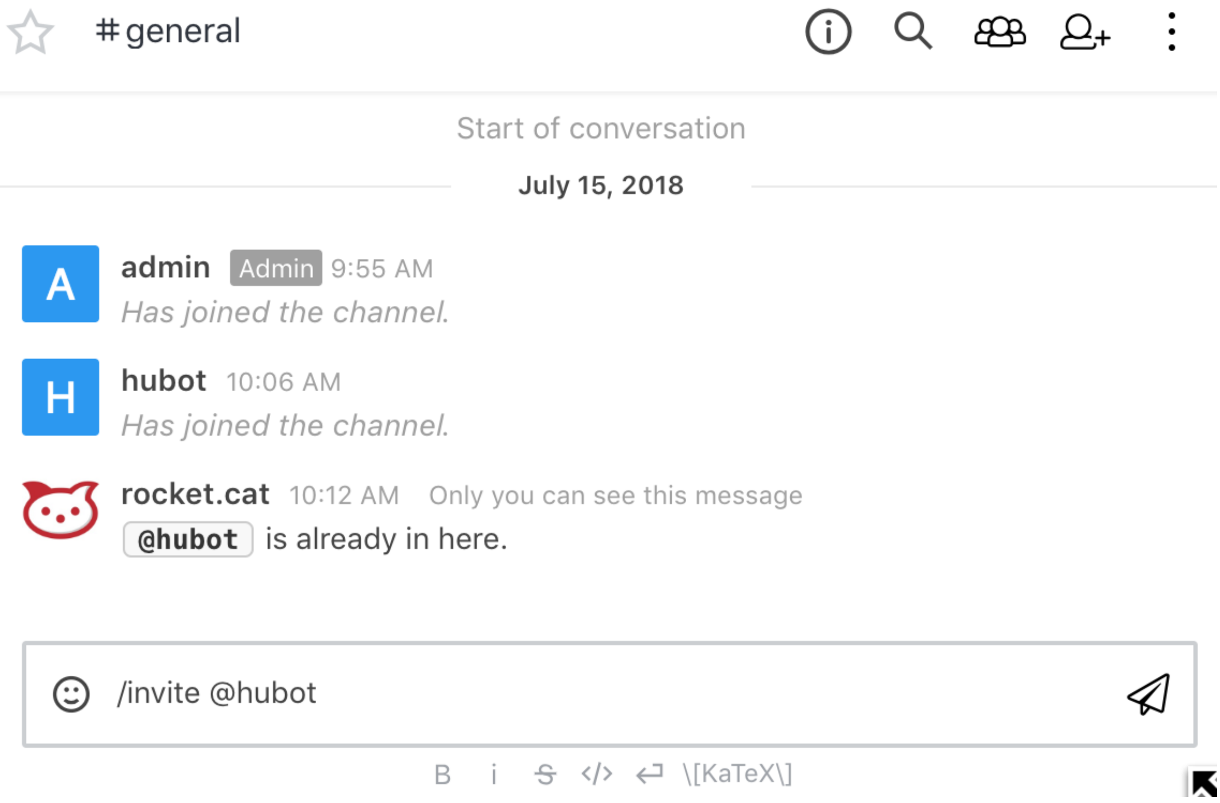
```
$ docker-compose restart hubot
```

```
Restarting unix_hubot_1 ... done
```

If everything went according to plan, you should now be able to navigate back to your web browser and send commands to your bot in the chat window.

In the message window at the bottom of the General chat screen, start by typing **`/invite @hubot`** to ensure that our bot is listening in the room (Figure 8-9).

> **NOTE**
>
> You may get a message from the internal admin `rocket.cat` that says `@hubot is already in here`. This is perfectly fine.



Start of conversation

July 15, 2018

admin Admin 9:55 AM
Has joined the channel.

hubot 10:06 AM
Has joined the channel.

rocket.cat 10:12 AM Only you can see this message
@hubot is already in here.

/invite @hubot

Figure 8-9. RocketChat Invite hubot screen

The environment variables used to configure hubot defined it’s alias as a period. So you can now try typing `. help` to test that the bot is responding. If everything is working you should get a list of commands that the bot understands and will respond to.

```
> . help
. adapter - Reply with the adapter
. echo <text> - Reply back with <text>
. help - Displays all of the help commands that this bot knows about.
. help <query> - Displays all help commands that match <query>.
...
. meme One does not simply <text> - Lord of the Rings Boromir
...
. ping - Reply with pong
. time - Reply with current time
. z <action> - Performs the action
...
. z stop - Ends the game for this channel
```

Finally, try typing:

```
. meme One does not simply launch a complex web service with docker-compos
```

So, for one last diversion, try creating a new chat channel by typing `/create zmachine` in the chat window. You should now be able to click on the new `zmachine` channel in the left sidebar and invite hubot with the chat command `/invite @hubot`.

NOTE

When you do this, hubot might say:

```
There's no game for zmachine!
```

This is nothing to be concerned about.

Next try typing the following commands into the chat window to play a chat-based version of the famous game, *Colossal Cave Adventure*:

```
. z start adventure

more
look
go east
examine keys
get keys

. z save firstgame
. z stop
. z start adventure
. z restore firstgame

inventory
```

WARNING

Interactive fiction can be addictive and a huge time sink. You have been warned… That being said, if you are aren’t already familiar with it and are interested in learning more, take a look at some of these resources.

- Definiton of Interactive Fiction
- Emulator
- Development
- Games
- Competition

You’ve now seen how easy it can be to configure, launch, and manage complex web services that require multiple components to accomplish their jobs using Docker Compose. In the next section, we will explore a few more of the features that Docker Compose includes.

Exercising Docker Compose

Now that you have that running and understand what the application is doing, we can dig in to get a little more insight into how the services are running. Some of the common Docker commands are also exposed as Compose commands, but for a specific stack rather than a single container or all of the containers on a host. You can run `docker-compose top` to see an overview of your containers and the processes that are running in them.

```
$ docker-compose top
unix_hubot_1
PID    USER   TIME                                      COMMAND
--------------------------------------------------------------------------
1353   1001   0:00   /bin/sh -c node -e
                     "console.log(JSON.stringify('$EXTERNAL_SCRIPTS'.split
                     external-scripts.json && npm install $(node -e
                     "console.log('$EXTERNAL_SCRIPTS'.split(',').join(' ')
                     bin/hubot
                     -n $BOT_NAME -a rocketchat
1521   1001   0:02   node node_modules/.bin/coffee node_modules/.bin/hubot
                     -n bot -a rocketchat

unix_mongo_1
 PID    USER   TIME                        COMMAND
-----------------------------------------------------------------------
23104   999    0:46   mongod --smallfiles --oplogSize 128 --replSet rs0

unix_rocketchat_1
 PID    USER    TIME     COMMAND
-----------------------------------
23399   99999   0:34   node main.js

unix_zmachine_1
 PID    USER   TIME                              COMMAND
--------------------------------------------------------------------------
2808    0      0:00   /root/src/../frotz/dfrotz -S 0 /root/src/../zcode/ad
23710   0      0:00   node /root/src/server.js
```

Similarly to how you would enter a running Docker container, you can run commands inside containers via the Docker Compose tooling using a `docker-compose exec` command:

```
$ docker-compose exec mongo bash
root@6f12015f1dfe:/# mongo
MongoDB shell version: 3.2.18
connecting to: test
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
        http://docs.mongodb.org/
Questions? Try the support group
        http://groups.google.com/group/mongodb-user
rs0:PRIMARY> exit
bye
root@6f12015f1dfe:/# exit
```

> **TIP**
>
> `docker-compose logs` and `docker-compose exec` are probably the most useful commands for troubleshooting. If `docker-compose` cannot build your image or start your container at all, you will need to fall back to the standard `docker` commands to debug your image and container, like we discussed in “Troubleshooting Broken Builds” and “Getting Inside a Running Container”.

You can also use Docker Compose to `start` and `stop` and, in most environments, `pause` and `unpause` either a single container or all of your containers, depending on what you need.

```
$ docker-compose stop zmachine
Stopping unix_zmachine_1 ... done
$ docker-compose start zmachine
Starting zmachine ... done
$ docker-compose pause
Pausing unix_mongo_1      ... done
Pausing unix_rocketchat_1 ... done
Pausing unix_zmachine_1   ... done
Pausing unix_hubot_1      ... done
$ docker-compose unpause
Unpausing unix_hubot_1      ... done
Unpausing unix_zmachine_1   ... done
Unpausing unix_rocketchat_1 ... done
Unpausing unix_mongo_1      ... done
```

Finally, when you want to tear everything down and delete all the containers created by `docker-compose`, you can run the following command:

```
$ docker-compose down
Stopping unix_hubot_1      ... done
Stopping unix_zmachine_1   ... done
Stopping unix_rocketchat_1 ... done
Stopping unix_mongo_1      ... done
Removing unix_hubot_1              ... done
Removing unix_zmachine_1           ... done
Removing unix_rocketchat_1         ... done
Removing unix_mongo-init-replica_1 ... done
Removing unix_mongo_1              ... done
Removing network unix_botnet
```

WARNING

Windows users: When you delete the MongoDB container using the `docker-compose down` command, all data in the MongoDB instance will be lost.

Wrap-Up

You should now have a very good feel for the types of things you can accomplish with Docker Compose and how this tool can be used to decrease the toil and increase the repeatability of your development environments.

In the next chapter, we will explore some of the tools that are available to help you scale Docker inside your data center and in the cloud.

Chapter 9. The Path to Production Containers

Now that we've explored tooling for bringing up a stack of containers on a single host, we need to look at how we'd do this in a large scale production environment. In this chapter, our goal is to show you how you might take containers to production based on our own experiences. There are a myriad of ways in which you will probably need to tailor this to your own applications and environments, but this all should provide you with a solid starting point to help you understand the Docker philosophy in practical terms.

Getting to Production

Getting an application from the point where it is built and configurable to the point where it is actually running on production systems is one of the most mine-ridden steps in going from zero to production. This has traditionally been complicated but is vastly simplified by the shipping container model. If you can imagine what it was like to load goods into a ship to take across the ocean before shipping containers existed, you have a sense of what most traditional deployment systems look like. In that old shipping model, randomly-sized boxes, crates, barrels, and all manner of other packaging were all loaded by hand onto ships. They then had to be manually unloaded by someone who could tell which pieces needed to be unloaded first so that the whole pile wouldn't collapse like a Jenga puzzle.

Shipping containers changed all that: we have a standardized box with well-known dimensions. These containers can be packed and unloaded in a logical order and whole groups of items arrive together when expected. We've built machinery to make managing them extremely efficient. This is the Docker deployment model. All Docker containers support the same external interface, and the tooling just drops them on the servers they are supposed to be on without any concern for what's inside.

In the new model, when we have a running build of our application, we don't have to write much custom tooling to kick off deployment. If we only want to ship it to one server, the `docker` command-line tooling will handle most of that for us. If we want to send it to more servers, then we will have look at some of more advanced tooling from the broader container ecosystem.

There is a progression you will follow while getting your applications to production on Docker:

1. Locally build and test a Docker image on your development box.

2. Build your official image for testing and deployment, usually from a CI or build system.

3. Push the image to a registry.

4. Deploy your Docker image to your server, then configure and start the container.

As your workflow evolves, you will eventually collapse all of those steps into a single fluid workflow:

1. Orchestrate the deployment of images and creation of containers on production servers.

But there is a lot more to the story than that. At the most basic level, a production story must encompass three things:

- It must be a repeatable process. Each time you invoke it, it needs to do the same thing.

- It needs to handle configuration for you. You must be able to define your application's configuration in a particular environment and then guarantee that it will ship that configuration on each deployment.

- It must deliver an executable artifact that can be started.

To accomplish that, there are a number of things you need to think about. We'll try to help with that by presenting a framework you can use to think about your application in its environment.

Docker's Role in Production Environments

We've covered a lot of capabilities that Docker brings to the table, and talked about some general production strategies. Before we dive deeper into production containers, let's look at how Docker fits into both a traditional and more modern production environment. If you are moving to Docker from a more traditional system, you can pick and choose which pieces you will delegate to Docker, to a deployment tool, or to a larger platform like Mesos or Kubernetes, or perhaps even leave on your more traditional infrastructure. We have successfully transitioned multiple systems from traditional deployments to containerized systems, and there is a wide spectrum of good solutions. But understanding the required components and what makes up the modern and traditional variants will put you on the right path to make good choices.

In Figure 9-1 we describe a number of concerns that need to be filled in a production system, the modern components that fill them, and the systems they might replace in a more traditional environment. We divide these up into concerns that are addressed by Docker itself, and those we ascribe to what we call the "platform." The platform is a system that usually wraps around more than one instance of Docker and presents a common interface across Docker instances. This might be a single system like Mesos, Kubernetes, or Docker Swarm, or it may be a deployment system, a separate monitoring system, and a separate orchestration system. In transition from a more traditional system to a fully containerized system with a scheduler, the platform might be more than one thing at a time. So let's take a look at each of these concerns and see how they fit together.

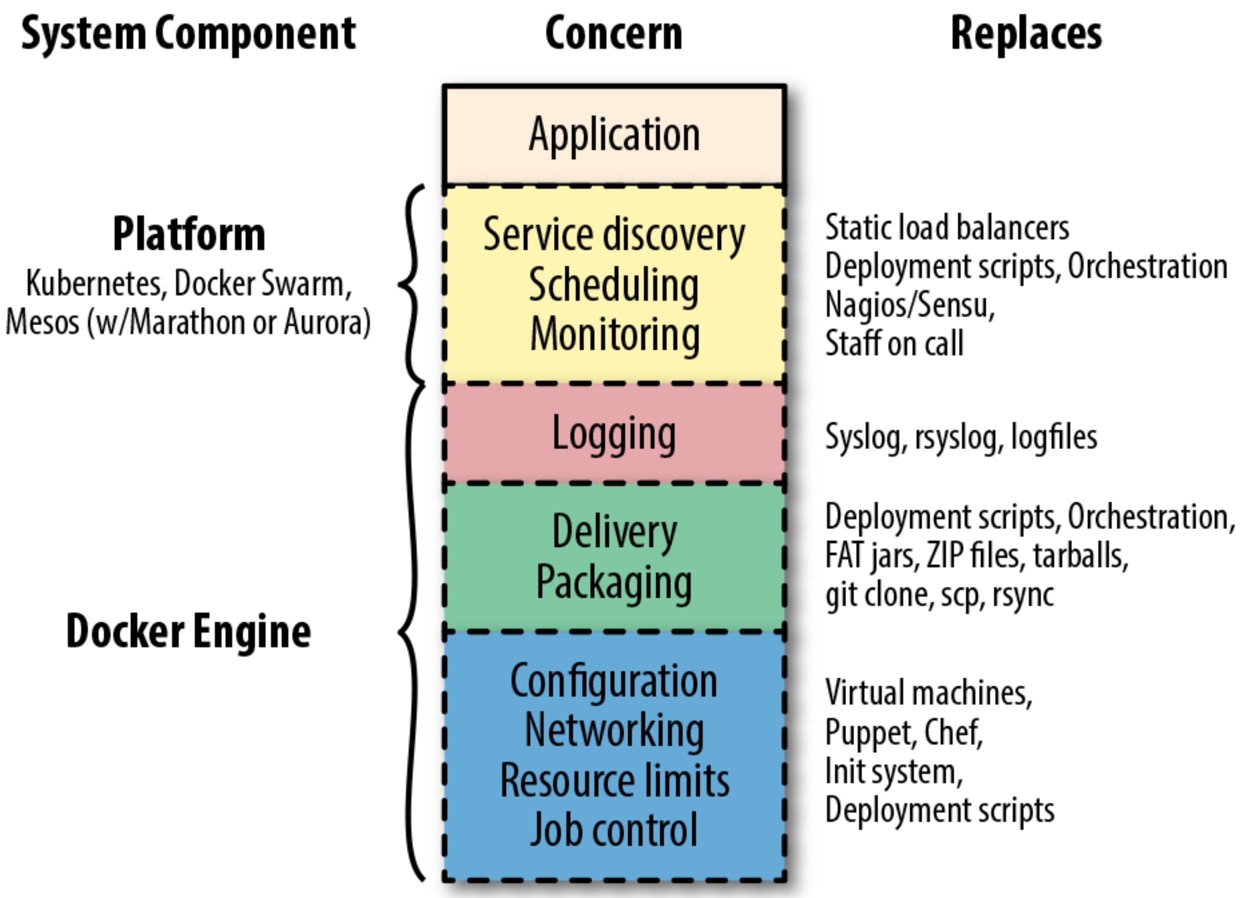


Figure 9-1. Docker's role in a production system

In the diagram, you can see that the application is sitting on the top of the stack. It relies on all of the concerns below it in a production system. In some cases your environment may call these concerns out specifically, and in others they may be addressed by something you don't necessarily think of as filling that concern. But your production applications will rely on most of these in one way or another, and they will need to be addressed in your production environment. If you want to transition from an existing environment to a Docker-based environment, you'll want to think about how you are providing these today and how they might be addressed in the new system.

We'll start with familiar territory, and then go from the bottom to the top. That familiar territory is your application. Your application is on top! Everything else is there to deliver functionality to your application. After all, it's the application that delivers business value and everything else is there to make that possible, to facilitate doing it at scale and reliably, and to standardize how it works across applications. While the items are ordered underneath your application in an intentional way, it's not the case that each layer provides functionality to the one above. They are all providing that functionality to the application itself.

Because Docker provides a lot of this stack, containerizing your system will make many of the choices easier. As we get closer to the platform part of the stack, we'll have more to think about, but understanding everything that lies below there will make that much more manageable.

Let's start with application job control.

Job Control

Job control really makes up just the table stakes for a modern deployment. This is part of the blue block in the drawing of concerns. You basically can't have a system of any kind without job control. It's something we've more traditionally left to the operating system, or the Linux init system (`upstart`, `systemd`, System V `init`, `runit`, etc.) more specifically. We tell the operating system about a process we want to have running and then we configure how the behavior should be around restarting it, reloading its configuration, and managing the lifecycle of the application. When we want to start or stop the application, we rely on these systems to handle that. We also rely on them in some cases to keep the application running in a more robust way by, for example, restarting it when it fails. Different applications require different job control. In a traditional Linux system you might use `cron` to start and stop jobs on a timed basis. How it does so is down to the specifics of that system, and there are many different systems to deal with.

But that's not great. If we're moving to the shipping container model, we want to be able to treat all jobs more or less the same way from the outside. We might need a little more metadata about them to get them to do the right thing, but we don't want to look inside the container. The Docker engine provides a strong set of primitives around job control—for example, `docker start`, `docker stop`, `docker run`, and `docker kill`—which map to most of the critical steps in the lifecycle of an application. We've placed this at the bottom of the stack of concerns because it's the lowest abstraction that Docker really provides for your application. Even if we didn't use any other part of Docker, this would be a big win because it's the same for all applications.

Resource Limits

Sitting above job control are resource limits. In Linux systems we can use cgroups directly if we want to, and production environments have sometimes done that before Docker popularized them. But more traditionally we relied on things like ulimit and the different settings of application runtime environments like the Java, Ruby, or Python virtual machines. In cloud systems, one of the early wins was that we could spin up individual virtual servers to limit the resources around a single business application. Compared to containers, of course, that is a pretty coarse-grained control.

With Docker you can easily apply a wide set of resource controls to your containers. It's up to you to decide whether or not you'll restrict your application's access to things like memory, disk space, or I/O when running in production. However, we highly recommend that you take the time to do this once you're comfortable with the needs of your application. As we've discussed, Docker gives this to you for free and it's a core part of what makes a container valuable. You can review the specific arguments that Docker uses to manage these resources in Chapter 5.

Networking

There is a lot of detail about Docker networking in Chapter 11, so we won't touch on it too heavily here, but your containerized system will need to somehow manage connecting your applications together on the network. Docker provides a rich set of configuration options for networking. You should really decide on one mechanism to use in your production environment and standardize that across containers. Trying to mix them is not an easy path to success.

Configuration

All applications need to somehow have access to their configuration. There are two levels of configuration for an application. The lowest level is how it expects the Linux environment around it to be configured. Containers handle this by providing a *Dockerfile* that we can use to build the same environment repeatably. In a more traditional system we might have used a configuration management system like Chef, Puppet, or Ansible to do this. You may still use those systems in a containerized world, but you are usually not using them to provide dependencies to applications. That job belongs to Docker and the *Dockerfile*. Even if the contents of the *Dockerfile* are different for different applications, the mechanism and tooling are all the same—and that's a huge win.

The next level of configuration is the configuration directly applied to the application. We talked earlier about this in detail. Docker's native mechanism is to use environment variables, and this works across all of the modern platforms. Some systems, notably, make it easier to rely on more traditional configuration files. We find that this can really impact the observability of the application and discourage you from relying on that crutch. There is more about the reasoning behind environment variables in Chapter 12.

Packaging and Delivery

We'll lump packaging and delivery together in our discussion here. This is an area where a containerized system has major advantages over a traditional one. Here we don't have to stretch our imaginations to see the parallels to the shipping container model: we have a consistent package, the container image, and a standardized way to get them places—Docker's registry and `pull` and `push` facilities. In more traditional systems we would have built handcrafted deployment tooling, some of which we hopefully standardized across our applications. But if we needed to have a multilanguage environment, this would have been trouble. In your containerized environment, you'll need to consider how you handle packaging your applications into images and how you store those images.

The easiest path for the latter is a paid subscription to a hosted, commercial registry. If that's acceptable to your company, then you should consider that. Several cloud providers, including Amazon, have image-hosting services that you can deploy inside your own environment, which are another good option. You can, of course, also build and maintain your own registry as we talked about in "Running a Private Registry".

Logging

Logging sits on the boundary of concerns that you can rely on Docker to provide in your containerized environment and concerns that the platform needs to manage. That's because, as we detailed in Chapter 6, Docker can collect all the logs from your containers and ship them somewhere. But by default, that somewhere is not even off of the local system. That might be great for a limited-size environment, and you could stop considering it there if this sounds like it's good enough for you. But your platform will be responsible for handling logs from lots of applications on lots of systems, so you'll probably want more than this. When designing this, refer back to Chapter 6 for more details on logging.

Monitoring

The first part of the system not fully solved by Docker still relies on the standardization that Docker brings to the table. The ability to health-check applications in a standardized way, as discussed in Chapter 6, means that monitoring applications for health is vastly simplified. In many systems the platform itself handles monitoring, and the scheduler will dynamically shut down unhealthy containers and potentially move the workload to a different server or restart the workload on the same system. In older systems, containers are often monitored by existing systems like Nagios, Zabbix, or other traditional monitoring systems. As we showed in

Chapter 6, there are also newer options, including systems like Prometheus.

In older systems, it is generally engineers who are paged, respond to issues, and make decisions about how to handle failed applications. In dynamic systems, this work generally moves into more automated processes that belong in the platform. In a transitional period, your system may have both while moving more and more to an automated system where engineers are paged only when the platform really can't intervene.

Scheduling

How do you decide which services run on which servers? Containers are easy to move around because Docker provides such good mechanisms for doing so. And that opens up lots of possibilities for better resource usage, better reliability, self-healing services, and dynamic scaling. But something has to make those decisions.

In older systems this was often handled with dedicated servers per service. You often configured a list of servers into the deployment scripts and the same set of servers would receive the new application on each deployment. One-service-per-server models drove early virtualization in private data centers. Cloud systems encouraged the one-service-per-server model by making it easy to slice and dice servers into commodity virtual servers. Autoscaling in systems like AWS handled part of this dynamic behavior. But if you move to containers, where many services may be running on the same virtual server, then scaling and dynamic behavior at the server level doesn't help you.

DISTRIBUTED SCHEDULERS

Distributed schedulers leverage Docker to let you reason about your entire network of servers almost as if it were a single computer. The idea here is that you define some policies about how you want your application to run, and you let the system figure out where to run it and how many instances of it to run. If something goes wrong on a server or with the application, you let the scheduler start it up again on resources that are healthy. This fits more into Docker, Inc. founder Solomon Hykes's original vision for Docker: a way to run your application anywhere without worrying about how it gets there. Generally, zero downtime deployment in this model is done in the blue-green style, where you launch the new generation of an application alongside the old generation, and then slowly migrate work from the old stack to the new one.

Using the metaphor now made famous by Kelsey Hightower, the scheduler is the system that plays Tetris for you, placing services on servers for best fit, on the fly.

Apache Mesos, which was originally written at the University of California, Berkeley, and most publicly adopted by Twitter and Airbnb, is the most mature option. Mesos is a resource pool abstraction that lets you run multiple frameworks on the same cluster of hosts. You can, for example, run Docker applications and Hadoop jobs on the same compute cluster. Mesos uses Zookeeper (or CoreOS's zetcd) and has been around for much longer than most of the other options because it actually predates Docker. Despite being on the scene for a number of years before Docker arrived, Mesos has outstanding support for Docker. In Mesos systems the actual scheduling is handled by the framework you run on top (e.g., HubSpot's Singularity, Mesosphere's Marathon, Mesos's Chronos, or Apache Aurora). All of these frameworks have full support for Docker—it's almost expected at this point.

All of these Mesos frameworks take quite different approaches, and it's worth evaluating at least the four main options to see which works best for you. Schedulers often can handle different kinds of workloads, including long-running services, one-off commands, and scheduled jobs.

Kubernetes is another popular scheduler and came out of Google in 2014. It inherits a lot of what they learned on their own internal Borg system. It was built on Docker from the beginning and not only supports Docker but also a few of the other runtimes, all of which run Docker containers. It's a big system with a lot of moving pieces. There are now at least two dozen different commercial distributions of Kubernetes and at least a dozen cloud environments. The Kubernetes ecosystem has fragmented at a really high rate, with lots of vendors trying to stake out their own territory early on. This space continues to change rapidly and while Kubernetes is really powerful, it's a moving target that's hard to stay on top of. If you are building a brand new system from scratch, you will probably want to consider Kubernetes. We believe it's a harder system to retrofit, and at the moment it is not clear which distributions will win out in the long run. Docker, Inc. is heavily integrating Kubernetes into their own tooling, so this might be a safer bet if you're looking for a platform to carry forward.

Docker Swarm came out of Docker, Inc. in 2015 and is built as a Docker native system from the ground up. It's an attractive option if you're looking to stay completely on a Docker platform and get support from a single vendor. It has not had the adoption in the market that Kubernetes has recently seen, and since Docker is integrating Kubernetes so heavily into their own tooling, this is probably not as clear a path as it once was.

ORCHESTRATION

When we talk about schedulers, we often talk about not just their ability to match jobs to resources but their orchestration capabilities as well. By that, we mean the ability to command and organize applications and deployments across a whole system. Your scheduler might move jobs for you on the fly or allow you to run tasks on each server specifically. This was more commonly handled in older systems by specific orchestration tools. You might find that even with a scheduler in your system, you want a tool explicitly for orchestration. Or you might find that you can get away without a scheduler and simply deploy applications using orchestration tools. You and your team will then act as schedulers. This might not sound space age, but it's a reasonable early approach for a small or transitional system. With these tools, you tell them what to do, and then they do it at scale while you wait, much like the deployment model of Capistrano or Fabric, for example. These tools generally provide the simplest way to get into production with Docker.

In this category are tools like:

- Spotify's Helios
- Ansible's Docker tooling
- New Relic's Centurion

Of all the features delivered by the platform, scheduling is undoubtedly the most powerful. It also has the most impact on applications when moving them into containers. Many applications are not designed to have service discovery and resource allocation change underneath them and require a significant number of changes to work well in a truly dynamic environment. For this reason, your move to a containerized system may not necessarily encompass moving to a scheduled platform initially. Often the best path to production containers lies in containerizing your applications while running inside the traditional system, and then moving on to a more dynamic, scheduled system.

Service Discovery

You can think of service discovery as the mechanism by which the application finds all the other services and resources it needs on the network. Rare is the application that has no dependency on anything else. Stateless, static websites are perhaps one of the only systems that may not need any service discovery. Nearly everything else needs to know something about the surrounding system and a way to find out that information. Most of the time this involves more than one system, but they are usually tightly coupled.

You might not think of them this way, but in traditional systems, load balancers were one of the primary means for service discovery. Load balancers are used for reliability and scaling, but they also keep track of all of the endpoints associated with a particular service. This is sometimes manually configured and sometimes more dynamic, but the way other systems find endpoints for a service is by using a known address or name for the load balancer. That's a form of service discovery, and load balancers are a common way to do this in older systems. They often are in modern environments, too, even if they don't look much like traditional load balancers. Other means for service discovery in older systems are static database configurations or application configuration files.

As you can see back in Figure 9-1, Docker does not itself address service discovery in your environment, except when using Docker Swarm mode. But for the vast majority of systems, service discovery is left to the platform. This means it's one of the first things you'll need to resolve in a more dynamic system. Containers are by nature easily moved, and that can break traditional systems easily if they were built around more statically deployed applications. Each platform handles this differently and you'll want to understand what works best with your system.

> **NOTE**
>
> Docker Swarm and Docker Swarm mode are not exactly the same thing. We will discuss Docker Swarm mode in more detail in Chapter 10.

Some examples of service discovery mechanisms you might be familiar with include:

- Load balancers with well-known addresses
- Round-robin DNS
- DNS SRV records
- Dynamic DNS systems
- Multicast DNS
- Overlay networks with well-known addresses
- Gossip protocols (e.g., in Cassandra, Seneca JS, or Sidecar from Nitro/New Relic)
- Apple's Bonjour protocol
- Apache Zookeeper
- HashiCorp's Consul
- CoreOS's etcd

That's a big list and there are a lot more options than that. Some of these systems also do a lot more than just service discovery, which can confuse the issue. An example of service discovery that may be closer to hand while you're trying to understand this concept is the linking mechanism used by Docker Compose in Chapter 8. This mechanism relies on a DNS system that the `dockerd` server supplies, which allows you to call out to known service names and find the right container at the resolved address.

Often you find that the interface to these systems basically relies on having well-known names and/or ports for a service. You might call out to *http://service-a.example.com* to reach service A on a well-known name. Or you might call out to *http://services.example.com:service-a-port* to reach the same service on a well-known name and port. Different modern environments handle this differently. Usually within the new system, this process will be managed and fairly seamless. And it's frequently easy for new applications to call out of the platform to more traditional systems, but sometimes it's not as easy going the other way. Often the best initial system (though not necessarily longer-term) is a system where you present dynamically configured load balancers that are easily reachable by systems in your older environment.

Examples of this include:

- The Mesos backend for the Traefik load balancer
- Nixy for `nginx` and Mesos
- Kubernetes's ingress controllers
- Standalone Sidecar service discovery with Lyft's Envoy proxy
- Istio and Lyft's Envoy

If you are running a blended modern and traditional system, getting traffic into the newer containerized system is generally the harder problem to solve and the one you should think through first.

Production Wrap-Up

Many people will start by using simple Docker orchestration tools. However, as the number of containers and frequency with which you deploy containers grow, the appeal of distributed schedulers will quickly become apparent. Tools like Mesos allow you to abstract individual servers and whole data centers into large pools of resources in which to run container-based tasks.

There are undoubtedly many other worthy projects out there in the deployment space. But these are the most commonly cited and have the most publicly available information at the time of this writing. It's a fast-evolving space, so it's worth taking a look around to see what new tools are being shipped.

In any case, you should start by getting a Docker infrastructure up and running and then look at outside tooling. Docker's built-in tooling might be enough for you. We suggest using the lightest-weight tool for the job, but having flexibility is a great place to be, and Docker is increasingly supported by more and more powerful tooling.

Docker and the DevOps Pipeline

So if we wade through all of those waters, we can get our production environment into robust shape. But how do we know it works? One of the key promises of Docker is the ability to test your application and all of its dependencies in exactly the operating environment it would have in production. It can't guarantee that you have properly tested external dependencies like databases, nor does it provide any magical test framework, but it can make sure that your libraries and other code dependencies are all tested together. Changing underlying dependencies is a critical place where things go wrong, even for organizations with strong testing discipline. With Docker, you can build your image, run it on your development box, and then test the exact same image with the same application version and dependencies before shipping it to production servers.

Testing your Dockerized application is not much more complicated than testing your application itself, but you need to make sure that your test environment has a Docker server you can run things on and that your application will allow you to use environment variables or command-line arguments to switch on the correct testing behavior. Next we'll cover one example of how you might do this.

Quick Overview

Let's draw up an example production environment for a fictional company. We'll try to describe something that is similar to the environment at a lot of companies, with Docker thrown into the mix for illustration purposes.

Our fictional company's environment has a pool of production servers that run Docker daemons, and an assortment of applications deployed there. There is a build server and test worker boxes that are tied to the test server. We'll ignore deployment for now and talk about it once we have our fictional application tested and ready to ship.

Figure 9-2 shows what a common workflow looks like for testing Dockerized applications, including the following steps:

1. A build is triggered by some outside means.

2. The build server kicks off a Docker build.

3. The image is created on the local `docker`.

4. The image is tagged with a build number or commit hash.

5. A container is configured to run the test suite based on the newly built image.

6. The test suite is run against the container and the result is captured by the build server.

7. The build is marked as passing or failing.

8. Passed builds are shipped to an image store (e.g., registry).

You'll notice that this isn't too different from common patterns for testing applications. At a minimum you need to have a job that can kick off a test suite. The steps we're adding here are just to create a Docker image first and invoke the test suite inside the container rather than on the raw system itself.

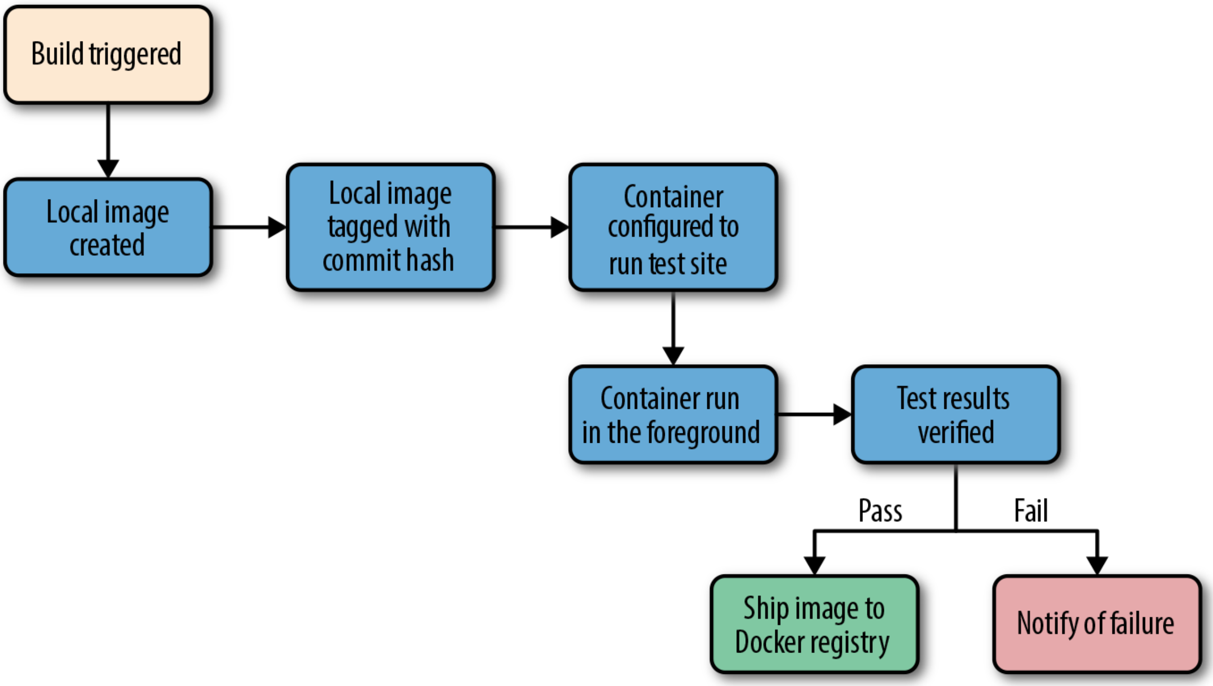

Build triggered
Local image created
Local image tagged with commit hash
Container configured to run test site
Container run in the foreground
Test results verified
Pass
Fail
Ship image to Docker registry
Notify of failure

Figure 9-2. Docker testing workflow chart

Let's look at how this works for the application we're deploying at our fictional company. We just updated our application and pushed the latest code to our Git repository. We have a post-commit hook that triggers a build on each commit, so that job is kicked off on the build server. The job on the test server is set up to talk to a `docker` on a particular test worker server. Our test server doesn't have `docker` running, but it has the `docker` command-line tool installed. So we run our `docker build` against that remote Docker server and it runs our *Dockerfile*, generating a new image on the remote Docker server. We could run `docker` on the test server itself if we had a smaller environment.

> **NOTE**
>
> You should build your container image exactly as you'll ship it to production. If you need to make concessions for testing, they should be externally provided switches, either via environment variables or through command-line arguments. The whole idea is to test the exact build that you'll ship, so this is a critical point.

Once the image has been built, our test job will create and run a new container based on our new production image. Our image is configured to run the application in production, but we need to run a different command for testing. That's OK! Docker lets us do that simply by providing the command at the end of the `docker run` command. In production, we'd start `supervisor` and it would start up an `nginx` instance and some Ruby unicorn web server instances behind that. But for testing, we don't need that `nginx` and we don't need to run our web application. Instead, our build job invokes the container like this:

```
$ docker run -e ENVIRONMENT=testing -e API_KEY=12345 \
    -i -t awesome_app:version1 /opt/awesome_app/test.sh
```

We called `docker run`, but we did a couple of extra things here, too. We passed a couple of environment variables into the container: `ENVIRONMENT` and `API_KEY`. These can either be new or overrides for the ones Docker already exports for us. We also asked for a particular tag—in this case, `version1`. That will make sure we build on top of the correct image even if another build is running simultaneously. Then we override the command that our container was configured to start in the *Dockerfile*'s CMD line. Instead, we call our test script, */opt/awesome_app/test.sh*. We don't have to here, but you should note that in some cases you will actually need to override the *Dockerfile*'s `ENTRYPOINT` (`--entrypoint`) to run something other than the default command for that container. Using `ENTRYPOINT` to run the default command is a bad practice but, sadly, a common one.

TIP

Always pass the precise Docker tag (usually a version or commit hash) for your image into the test job. If you always use `latest`, then you won't be able to guarantee that another job has not moved that tag just after your build was kicked off. If you use the precise tag, you can be sure you're testing the right version of the application.

A critical point to make here is that `docker run` will exit with the exit status of the command that was invoked in the container. That means we could just look at the exit status to see if our tests were successful. If your test suite is properly designed, this is probably all you need. If you need to run multiple steps, or the exit code can't be relied on, one way to handle this is to capture all of the output of the test run into a file and then sift through the output to look for status messages. Our fictional build system does just that. We write out the output from the test suite and our *test.sh* echoes either `Result: SUCCESS!` or `Result: FAILURE!` on the last line to signify if our tests passed. If you need to rely on this mechanism, be sure to look for some output string that won't appear by happenstance in your normal test suite output. If we need to look for "success," for example, we had best limit it to looking at the last line of the file, and maybe also anchored to the beginning of the line. In this case, we look at just the last line of the file and find our success string, so we mark the build as passed.

There is one more Docker-specific step. We want to take our passed build and push that image to our registry. The registry is the interchange point between builds and deployments. It also allows us to share the image with other builds that might be stacked on top of it. But for now, let's just think of it as the place where we put and tag successful builds. Our build script will now do a `docker tag` to give the image the right build tag(s), including `latest`, and then a `docker push` to push the build to the registry.

That's it! As you can see, there is not much to this compared with testing a normal application. We take advantage of the client/server model of Docker to invoke the test on a different server from the test master server, and we have to wrap up our test output in a shell script to generate our output status. But other than that, it's a lot like a common build system anywhere.

But, critically, our fictional company's system makes sure they only ship applications whose test suite has passed on the same Linux distribution, with the same libraries and the same exact build settings. That container might then also be tested against any outside dependencies like databases or caches without having to mock them. None of this guarantees success, but it gets them a lot closer to that guarantee than the dependency roulette often experienced by production deployment systems.

> **NOTE**
>
> If you use Jenkins for continuous integration or are looking for a good way to test scaling Docker, there are many plug-ins for Docker, Mesos, and Kubernetes that are worth investigating.

Outside Dependencies

But what about those external dependencies we glossed over? Things like the database, or Memcache or Redis instances that we need to run our tests against our container? If our fictional company's application needs a database to run, or a Memcache or Redis instance, we need to solve that external dependency to have a clean test environment. It would be nice to use the container model to support that dependency. With some work, you can do this with tools like Docker Compose, which we described in detail in Chapter 8. In Docker Compose, our build job could express some dependencies between containers, and then Compose will connect them together seamlessly.

Being able to really test your application in an environment that looks like where it will live is a huge win. Compose makes this pretty easy to set up. You'll still need to rely on your own language's testing framework for the tests, but the environment is really easy to orchestrate.

Wrap-Up

Now that we have covered many of the things you need to consider when moving your containerized applications into production, it is time to explore how Docker clusters can be built to support the global, always-on, on-demand nature of many modern technology operations.

Chapter 10. Docker at Scale

One of Docker's major strengths is its ability to abstract away the underlying hardware and operating system so that your application is not constrained to any particular host or environment. It facilitates scaling a stateless application not just horizontally within your data center, but also across cloud providers without many of the traditional barriers you would encounter. True to the shipping container metaphor, a container on one cloud looks like a container on another.

Many organizations find turnkey cloud deployments of Docker appealing because they can gain many of the immediate benefits of a scalable container-based platform without needing to completely build something in-house. Even though this is true, the barrier is actually pretty low for building your own platform in the cloud or in your own data center, and we'll cover some options for doing that shortly.

The major public cloud providers have all worked to support Linux containers natively in their offerings. Some of the biggest efforts to implement Docker containers in the public cloud include:

- Amazon Elastic Container Service
- Google Kubernetes Engine
- Azure Container Service
- Red Hat OpenShift

Even cloud providers running on non-Linux operating systems like SmartOS and Windows are actively supporting the Docker ecosystem:

- Joyent Triton
- Azure Container Service (Azure supports both Windows and Linux operating systems)

It’s trivial to install Docker on a Linux instance in one of the public clouds. But getting Docker onto the server is usually just one step in the creation of a full production environment. You could do this completely on your own, or you could avail yourself of the many tools available from the major cloud providers, Docker, Inc., and the Docker community. Much of the tooling will work equally well in either a public cloud or your own data center.

A first step using built-in tooling might be to leverage Docker Engine’s Swarm mode to deploy containers easily across a large pool of Docker hosts. If you want to start with something even simpler, you could also try some of the deployment-oriented community tools like Ansible’s Docker tooling, New Relic’s Centurion, or Spotify’s Helios to quickly facilitate multihost container deployment without all the complexity of a full-blown scheduler.

In the realm of schedulers and more complex tooling systems, we are spoiled for choice with systems that replicate much of the functionality you would get from a public cloud provider, including Kubernetes, DC/OS Community Edition, and a standard Mesos cluster, as we discussed in the last chapter. Even if you run in a public cloud, there are some compelling reasons for why you might choose to run your own Docker environment rather than use one of the off-the-shelf offerings.

In this chapter, we’ll cover some options for running Docker at scale, first going through some of the simpler tools like Centurion and Docker Swarm mode, and then diving into some more advanced tools like Kubernetes and Amazon Elastic Container Service (ECS) with Fargate. All of these examples should give you a view of how you can leverage Docker to provide an incredibly flexible platform for your application workloads.

Centurion

Centurion is one of many tools that enables repeatable deployment of applications to a group of hosts. While most scheduler platforms treat the cluster as a single machine, you instead tell Centurion about each individual host that you want it to know about. Its focus is on simplifying and guaranteeing the repeatability of zero-down-time deployments with containers. Centurion does not do much more than manage your container deployment in a reliable manner, and this makes it very easy to get started with. It assumes that a load balancer sits in front of your application instances. Centurion is an easy first step in moving from traditional deployments to a Docker workflow and is a great option for people who aren’t ready for, or simply don’t need, the features of Swarm, Kubernetes, or Mesos.

NOTE

We could equally be covering Spotify's Helios here, but we believe that Centurion is the simplest of these tools to get up and running. We encourage you to see if there are tools that are a better fit for your deployment needs, but this section should give you a taste of what you might achieve as a first step.

Let's look at deploying a simple application with Centurion. Here you'll deploy the public `adejonge/helloworld` container as your web application. It won't do much more than serve up a welcome page that you can see in a browser. You could easily switch in your custom application. The only requirement is that it be deployed to a registry.

Before you can deploy, you have to satisfy the dependencies for the tool and get it installed. Centurion depends on the Docker command-line tool and requires that you have Ruby 2.0 or higher, so you'll want to make sure you have a system with these installed. Centurion is known to run well on Linux or macOS. It can conceivably run on Windows, but this is untested by the package maintainers. Ruby runtime packages are available via `yum`, `dnf`, `apk`, and `apt-get` on all popular Linux distributions. Generally, any distribution with a kernel new enough to run Docker will ship with packages that meet this requirement. On recent macOS versions, you will already have the right version of Ruby. If you are on an older release, you can install a recent Ruby with Homebrew—which we installed in Chapter 3. Most Linux distributions that are capable of running Docker also ship with a modern enough Ruby to run Centurion. You can check to see if you have Ruby installed, and if you have a version new enough, like this:

```
$ ruby -v

ruby 2.4.2p198 (2017-09-14 revision 59899) [x86_64-darwin17]
```

Here, we have Ruby 2.4.2, which is plenty recent. Once you have Ruby running, install Centurion with the Ruby package manager:

```
$ gem install centurion

Fetching: trollop-2.1.2.gem (100%)
Successfully installed trollop-2.1.2
...
Parsing documentation for centurion-1.9.0
Installing ri documentation for centurion-1.9.0
Done installing documentation for trollop, excon, logger-colors,
  net-ssh, net-scp, sshkit, centurion after 5 seconds
```

```
7 gems installed
```

You can now invoke `centurion` from the command line to make sure it's available:

```
$ centurion --help

Options:
  -p, --project=<s>          project (blog, forums...)
  -e, --environment=<s>      environment (production, staging...)
  -a, --action=<s>           action (deploy, list...) (default: list)
  -i, --image=<s>            image (yourco/project...)
  -t, --tag=<s>              tag (latest...)
  -h, --hosts=<s>            hosts, comma separated
  -d, --docker-path=<s>      path to docker executable (default: docker)
  -n, --no-pull              Skip the pull_image step
  --registry-user=<s>        user for registry auth
  --registry-password=<s>    password for registry auth
  -o, --override-env=<s>     override environment variables, comma separat
  -l, --help                 Show this message
```

There are a lot of options there, but right now you're just making sure that it's installed and working. If Centurion is not yet available and you get an error, then you can add it to your path like so:

```
$ gempath=`gem environment | grep '\- INSTALLATION DIRECTORY' | awk '{prin
$ export PATH=$gempath/bin:$PATH
```

You should now be able to invoke `centurion --help` and see the output of the help.

To begin with, you'll just make a directory in which you'll store the Centurion configuration. If this were your own custom application, this might be the application's directory, or it might be a directory in which you store all the deployment configs for all of your applications. We encourage the latter for larger installations because it facilitates conversations between teams about the deployment configuration, and helps promote good patterns between teams. Since you're just going to deploy the public `adejonge/helloworld` container, let's create a temporary directory to house the configs. Then you'll change directories into it and tell Centurion to scaffold a basic config for you with the `centurionize` tool:

```
$ cd /tmp
$ mkdir helloworld
$ cd helloworld
$ gem install bundle
```

```
...
Done installing documentation for bundler, bundle after 4 seconds
2 gems installed

$ centurionize -p helloworld

Creating /tmp/helloworld/config/centurion
Writing example config to /tmp/helloworld/config/centurion/helloworld.rake
Writing new Gemfile to /Users/skane/tmp/helloworld/Gemfile
Adding Centurion to the Gemfile

Remember to run `bundle install` before running Centurion

Done!
```

We can ignore the Gemfile for now and just open the config it generated for us in *config/centurion/helloworld.rake*. You might take a look at it to see what it put in place in order to get an idea of what Centurion can do. The scaffolded config contains examples of how to use many of Centurion's features. We'll just edit it down to the basics we care about:

```
namespace :environment do
  desc 'Development environment'
  task :development do
    set_current_environment(:development)
    set :image, 'adejonge/helloworld'

    env_vars MY_ENV_VAR: 'something important'

    host_port 8080, container_port: 8080

    host '192.168.2.10'
    host '192.168.2.25'
  end
end
```

NOTE

It is important that you edit the `host` lines to match the hostnames or IP addresses for your Docker servers; otherwise, it won't deploy anything! In theory, you can get Centurion working with Docker Community Edition for Mac and Windows, but at the moment, it is not straightforward or recommended.

There are two ways to use this. As is, it will require that you have Docker's API open over the network so that Centurion can connect. Unless you've already got that secured, you probably want to use it over SSH instead. If you have a user who can use the Docker command locally on your servers, you can make Centurion SSH into your hosts to deploy your containers instead of requiring an exposed port. To enable this feature, simply, add something like this to the task in your Rakefile.

```
# Add on the line after 'set_current_environment(:development)'
ssh_user = ENV['USER']
puts "[Info] Will SSH using: #{ssh_user}"
set :ssh, true
set :ssh_user, ssh_user
set :ssh_log_level, Logger::WARN
```

If you are going to use SSH, you'll want to add the required private keys to a local SSH agent so that you avoid any password prompts.

Centurion supports multiple environments in the same config. Here you're just going to deploy to *development*. You could add as many environments as you like. The default file also uses a pattern where common configurations among environments are put into a *common* section that is called by each environment. For demonstration purposes, we've cut this config down to a bare minimum.

You now have a config that will let you deploy the `adejonge/helloworld` image from the public registry to two hosts, while setting the environment variable `MY_ENV_VAR` and mapping port 8080 inside the container to port 8080 of the server. Centurion supports any number of environment variables, hosts, ports, or volume mounts. The idea is to store a repeatable configuration for your application that can be stamped out onto as many Docker hosts as needed.

Centurion supports a rolling deployment model out of the box for web applications. It will cycle through a set of hosts, taking one container down at a time to help ensure that your application stays available throughout the deployment. It uses a defined health-check endpoint on a container to enable rolling deployments, health-checking each container as it comes up and not moving on to the next host until it has been successful. By default, this endpoint is set to "/" and that's good enough for our simple welcome page application. Nearly all of this is configurable, but we'll keep it simple for this demonstration. Once you're more familiar with the tool, you should go back to the README to read about all of the other options.

You're now ready to deploy this to your development environment. You are going to tell Centurion to use the `helloworld` project, use the `development` environment, and do a web application zero-downtime deployment with `rolling_deploy`. Centurion will initiate a `docker pull` on the hosts in parallel; then, on each host in turn, it will create a new container, tear down the old one, and start up the new one. We'll cut down the very verbose output to get a clearer idea of the process:

```
$ centurion -p helloworld -e development -a rolling_deploy

...
I, [... #22324]  INFO -- : Fetching image adejonge/helloworld:latest IN PA
I, [... #22324]  INFO -- : Using CLI to pull
I, [... #22324]  INFO -- : Using CLI to pull
latest: Pulling from adejonge/helloworld
...
I, [... #22324]  INFO -- : ----- Connecting to Docker on 192.168.2.10 ----
...
I, [... #22324]  INFO -- : ----- Connecting to Docker on 192.168.2.10 ----
I, [... #22324]  INFO -- : Image sha256:4 found on 192.168.2.10
...
I, [... #22324]  INFO -- : ----- Connecting to Docker on 192.168.2.10 ----
I, [... #22324]  INFO -- : Looking for containers listening on port 8080
I, [... #22324]  INFO -- : Stopping container(s): []
I, [... #22324]  INFO -- : Creating new container for adejonge/helloworld:
I, [... #22324]  INFO -- : Starting new container 965f0947
I, [... #22324]  INFO -- : Inspecting new container 965f0947:
...
I, [... #22324]  INFO -- : Waiting for the port to come up
I, [... #22324]  INFO -- : Found container up for 0 seconds
I, [... #22324]  INFO -- : Container is up!
...
I, [... #22324]  INFO -- : ----- Connecting to Docker on 192.168.2.20 ----
I, [... #22324]  INFO -- : Service name helloworld
```

Here you can see Centurion pulling the requested image; verifying that it pulled properly; and then connecting to the hosts to stop the old container, create a new one, start it up, and then continuously health-check it until it reports back that it is healthy. At the very end, it cleans up the old containers so that they don't hang around forever.

Now your application is up and running on both of your servers. You can connect to it with a web browser by hitting the IP address of either server on port 8080.

```
http://192.168.2.20:8080/
```

If everything is working correctly, you should see the simple message `Hello World from Go in minimal Docker container` in your web browser.

In real production, you'll want a load balancer configured to sit in front of these hosts and redirect your clients to either of the instances. There is nothing dynamic to the Centurion setup, but it gets your application deployed with all the basic benefits of Docker for a minimal investment of time.

That's all there is to a very basic setup. This class of tooling is very easy to get started with and will get you to a production infrastructure quickly. But growing your Docker deployment to a vast scale will likely involve a distributed scheduler or one of the cloud provider's container platforms.

In that vein, let's take a look at Docker's Swarm tooling and see how this built-in Docker cluster mode can bridge the gap between simple tools like Centurion and the more full-featured robust platforms like Kubernetes and Amazon Elastic Container Service.

Docker Swarm Mode

After building the container runtime in the form of the Docker engine, the engineers at Docker turned to the problems of orchestrating a fleet of individual Docker hosts and effectively packing those hosts full of containers. The first tool that evolved from this work was called Docker Swarm. As we explained early on, and rather confusingly, there are now two things called "Swarm," both of which come from Docker, Inc.

The original standalone Docker Swarm is officially Docker Swarm, but there is a second "Swarm," which is more specifically called Swarm mode. This is actually built into the Docker Engine. The built-in Swarm mode is a lot more capable than the original Docker Swarm and is intended to replace it entirely. Swarm mode has the major advantage of not requiring you to install anything separately. You already have this on your Docker box! This is the Docker Swarm we'll focus on here. Hopefully now that you know there are two different Docker Swarms, you won't get confused by contradictory information on the internet.

The idea behind Docker Swarm is to present a single interface to the `docker` client tool, but have that interface be backed by a whole cluster rather than a single Docker daemon. Swarm is primarily aimed at managing clustered computing resources via the Docker tools. It has grown a lot since its first release and now contains several scheduler plug-ins with different strategies for assigning containers to hosts, and with basic service discovery built in. But it remains only one building block of a more complex solution. Docker has also added native support for deploying containers to Kubernetes into some Docker clients,[1] which means that for some situations it is possible to start with a simpler solution like Swarm and then graduate to Kubernetes without needing to change as many of the tools that users rely on.

Swarm clusters can contain one or more managers that act as the central management hub for your Docker cluster. It is best to set up an odd number of managers. Only one manager will act as the cluster leader at a time. As you add more nodes to Swarm, you are merging them into a single, cohesive cluster that can be easily controlled with the Docker tooling.

Let's get a Swarm cluster up and running. To start, you will need three or more Linux servers that can talk to each other over the network. Each of these servers should be running recent releases of Docker Community Edition from the official Docker software repositories.

TIP

Refer to Chapter 3 for details on installing the `docker-ce` packages on Linux.

For this example, we will use three Ubuntu servers running `docker-ce`. The very first thing you'll need to do is `ssh` to the server that you want to use as the Swarm manager and then run the `swarm init` command using the IP address for your Swarm manager.

```
$ ssh 172.17.4.1
...

ubuntu@172.17.4.1:$ sudo docker swarm init --advertise-addr 172.17.4.1

Swarm initialized: current node (hypysglii5syybd2zew6ovuwq) is now a manag

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

    docker swarm join --token SWMTKN-1-14......a4o55z01zq 172.17.4.1:2377

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

WARNING

In many of this chapter's examples, it's important that you use the correct IP addresses for your manager and worker nodes.

This step will initialize the Swarm manager and give you the token that is required for nodes that want to join the cluster. Make note of this token somewhere safe, like a password manager. Don't worry too much if you lose this token; you can always get it again by running the following command on the manager:

```
sudo docker swarm join-token --quiet worker
```

You can inspect your progress so far by running your local `docker` client pointed at the new manager node's IP address.

```
$ docker -H 172.17.4.1 info

...
Swarm: active
 NodeID: usjngewgh5syybd2ws6ovqwe
 Is Manager: true
 ClusterID: k2ujqs3vf8l1f31qx5fh2pouh
 Managers: 1
 Nodes: 1
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Number of Old Snapshots to Retain: 0
  Heartbeat Tick: 1
  Election Tick: 3
 Dispatcher:
  Heartbeat Period: 5 seconds
 CA Configuration:
  Expiry Duration: 3 months
  Force Rotate: 0
 Autolock Managers: false
 Root Rotation In Progress: false
 Node Address: 172.17.4.1
 Manager Addresses:
  172.17.4.1:2377
...
```

You can also list all of the nodes that are currently in the cluster with the following command:

```
$ docker -H 172.17.4.1 node ls

ID          HOSTNAME       STATUS  AVAILABILITY  MANAGER STATUS
hyp...   *  ip-172-17-4-1  Ready   Active        Leader
```

At this point, you can add the two additional servers as workers to the Swarm cluster. This is what you'd do in production if you were going to scale up, and Swarm makes this pretty easy.

```
$ ssh 172.17.4.2 \
    "sudo docker swarm join --token SWMTKN-1-14......a4o55z01zq 172.17.4.1

This node joined a swarm as a worker.

$ ssh 172.17.4.3 \
    "sudo docker swarm join --token SWMTKN-1-14......a4o55z01zq 172.17.4.1

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

If you rerun `docker node ls` you should now see that you have a total of three nodes in your cluster, and only one of them is marked as the `Leader`.

```
$ docker -H 172.17.4.1 node ls

ID          HOSTNAME       STATUS  AVAILABILITY  MANAGER STATUS
hyp... *    ip-172-17-4-1  Ready   Active        Leader
qwe...      ip-172-17-4-2  Ready   Active
poi...      ip-172-17-4-3  Ready   Active
```

This is all that's required to get a Swarm cluster up and running in Swarm mode (Figure 10-1)!

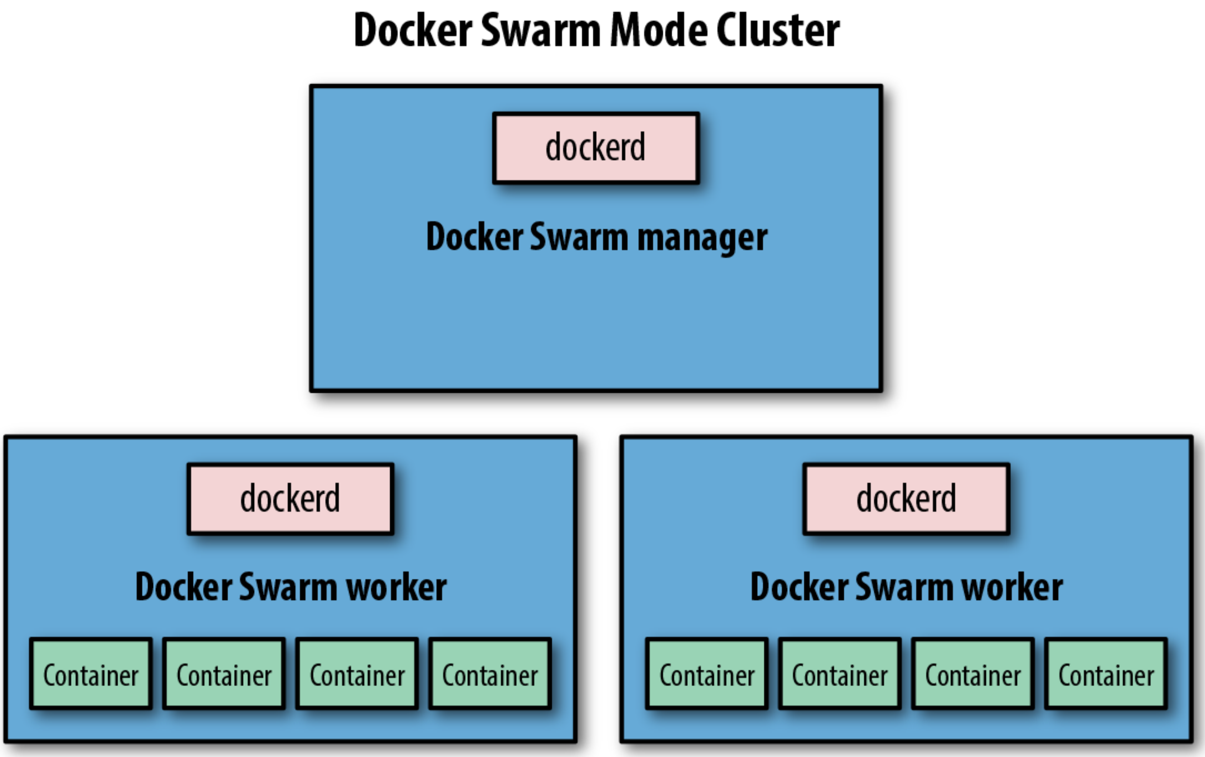


Figure 10-1. Simple Docker Swarm Mode cluster

The next thing you should do is create a default network for your services to use. Again, it's pretty simple to do that:

```
$ docker -H 172.17.4.1 network create --driver=overlay default-net

ckwh5ph4ksthvx6843ytrl5ik
```

Up to this point, we've just been getting the underlying pieces running, and so far we haven't deployed any real business logic. So let's launch your first service into the cluster. You can do that with a command like this:

```
$ docker -H 172.17.4.1 service create --detach=true --name quantum \
    --replicas 2 --publish published=80,target=8080 --network default-net
    spkane/quantum-game:latest

tiwtsbf270mh83032kuhwv07c
```

The service we're launching with starts containers that host the Quantum web game. This is a browser-based puzzle game that uses real quantum mechanics. We hope that this is a more interesting example than another Hello World!

> **WARNING**
>
> Although we're using the `latest` tag in many of these examples, it is important that you don't use this tag in production. It is convenient for the book, since we can easily push out updates to the code, but this tag floats and cannot be pinned to a specific release over a long period of time. That means if you use `latest`, then your deployments are not repeatable! It can also easily lead to a situation where you don't have the same version of an application running on all the servers.

Let's now see where those containers ended up, by running `docker service ps` against the service name you created:

```
$ docker -H 172.17.4.1 service ps quantum

ID   NAME      IMAGE     NODE          DESIRED STATE CURRENT STATE
v... quantum.1 spkane/... ip-172-17-4-2 Running       Running 56 seconds a
u... quantum.2 spkane/... ip-172-17-4-3 Running       Running 56 seconds a
```

Swarm mode uses a routing mesh between the nodes to automatically route traffic to a container that can serve the request. When you specify a published port in the `docker service create` command, the mesh makes it possible to hit this port on any of your three nodes and will route you to the web application. Notice that we said any of the *three* nodes even though you only have two instances running. Traditionally, you would have had to also set up a separate reverse proxy layer to accomplish this, but its batteries are included with Swarm mode.

To prove it, you can test the service now by pointing a web browser to the IP address of any of your nodes.

```
http://172.17.4.1/
```

If everything is working as expected, you the splash screen for “The Quantum Game” will greet you, quickly followed by the puzzle board.

Here we only have one instance running, but you would probably have a number of instances in a real-world scenario. To get a list of all the services, we can use `service ls`:

```
$ docker -H 172.17.4.1 service ls

ID      NAME     MODE        REPLICAS  IMAGE                            PORTS
tn...   quantum  replicated  2/2       spkane/quantum-game:latest       *:80->80
```

This gives us a summary view with the most commonly needed information, but sometimes that’s not enough. Docker maintains a lot of other metatadata about services, just like it does for containers. Mimicking the Docker CLI, we can get detailed information about a service with `service inspect`:

```
$ docker -H 172.17.4.1 service inspect --pretty quantum

ID:             tn5075th07tuncwqa4po4a514
Name:           quantum
Service Mode:   Replicated
 Replicas:      2
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:         spkane/quantum-game:latest@sha256:257286d3126622397f...
Resources:
Networks: default-net
Endpoint Mode:  vip
Ports:
 PublishedPort = 80
  Protocol = tcp
  TargetPort = 8080
  PublishMode = ingress
```

There is a lot of info here, so let’s point out some of the more important things. First we can see that this is a replicated service with two replicas, just like we saw in the `service ls` command. We can also see that Docker is health-checking the service at five-second intervals. Running an update to the service will use the `stop-first` method, which means it will take our service first to *N*–1 and then spin up a new instance to take us back to *N*. You might want to always run in *N*+1 mode so that you are never down a node during updates in production. You can change that with the `--update-order=start-first` option to the `service update` command. It will exhibit the same behavior in a rollback scenario, and we can likewise change that with `--rollback-order=start-first`.

In a real-world scenario, we need to be able to not only launch our service but also to scale it up and down. It would be a shame if we had to redeploy it to do that, not to mention it could introduce any number of issues. Luckily, Swarm makes it easy to scale our services with a single command. To double the number of instances you have running from two to four, you can simply run this:

```
$ docker -H 172.17.4.1 service scale --detach=false quantum=4

quantum scaled to 4
overall progress: 4 out of 4 tasks
1/4: running   [==================================================>]
2/4: running   [==================================================>]
3/4: running   [==================================================>]
4/4: running   [==================================================>]
verify: Service converged
```

NOTE

We used `--detach=false` in the previous command so that it was easier to see what was happening.

We can now use `service ps` to show us that Swarm did what we asked. This is the same command we ran earlier, but now we should have more copies running! But wait, didn't we ask for more copies than we have nodes?

```
$ docker -H 172.17.4.1 service ps quantum

ID   NAME      IMAGE      NODE          DESIRED STATE CURRENT STATE
v... quantum.1 spkane/... ip-172-17-4-2 Running       Running 56 seconds a
u... quantum.2 spkane/... ip-172-17-4-3 Running       Running 56 seconds a
e... quantum.3 spkane/... ip-172-17-4-1 Running       Running 2 minutes ag
q... quantum.4 spkane/... ip-172-17-4-1 Running       Running 2 minutes ag
```

You'll notice that you have two services running on the same host. Did you expect that? This may not be ideal for host resiliency, but by default Swarm will prioritize ensuring that you have the number of instances that you requested over spreading individual containers across hosts when possible. If you don't have enough nodes, you will get multiple copies on each node. In a real-world scenario, you need to think carefully about placement and scaling. You might not be able to get away with running multiple copies on the same host in the event that you lose a whole node. Would your application still serve users at that reduced scale?

When you need to deploy a new release of your software, you will want to use the `docker service update` command. There are a lot of options to this command, but here's one example:

```
$ docker -H 172.17.4.1 service update --update-delay 10s \
    --update-failure-action rollback --update-monitor 5s \
    --update-order start-first --update-parallelism 1 \
    --detach=false \
    --image spkane/quantum-game:latest-plus quantum

quantum
overall progress: 4 out of 4 tasks
1/4: running   [==================================================>]
2/4: running   [==================================================>]
3/4: running   [==================================================>]
4/4: running   [==================================================>]
```

```
verify: Service converged
```

Running this command will cause Swarm to update your service one container at a time, pausing in between each update. Great, but what if something were to go wrong? We might need to deploy a previous release to get back to working order. You could now roll back to the previous version with the `service rollback` command, which we discussed in passing a little bit earlier:

```
$ docker -H 172.17.4.1 service rollback quantum

quantum
rollback: manually requested rollback
overall progress: rolling back update: 4 out of 4 tasks
1/4: running   [>                                                  ]
2/4: running   [>                                                  ]
3/4: running   [>                                                  ]
4/4: running   [>                                                  ]
verify: Service converged
```

That's about as nice a rollback mechanism as you could ask for. You don't have to keep track of the previous version; Docker does that for you. All you need to do is tell it to roll back and it pulls the previous metadata out of its internal storage and performs the rollback. Note that just like during deployment, it can health-check your nodes to make sure the rollback is working successfully.

So, what happens if one of your servers is experiencing an issue and you need to take it offline? In this case, you can easily drain all the services off of a single node by using the `--availability` option to the `docker node update` command.

Let's take a look at the nodes that you have in the cluster again:

```
 docker -H 172.17.4.1 node ls

ID          HOSTNAME        STATUS  AVAILABILITY  MANAGER STATUS
hyp... *    ip-172-17-4-1   Ready   Active        Leader
qwe...      ip-172-17-4-2   Ready   Active
poi...      ip-172-17-4-3   Ready   Active
```

Let's also check where our containers are currently running:

```
$ docker -H 172.17.4.1 service ps -f "desired-state=running" quantum

ID   NAME      IMAGE      NODE          DESIRED STATE CURRENT STATE
```

```
f... quantum.1 spkane/... ip-172-17-4-1 Running       Running 6 minutes ag
7... quantum.2 spkane/... ip-172-17-4-1 Running       Running 6 minutes ag
r... quantum.3 spkane/... ip-172-17-4-2 Running       Running 6 minutes ag
6... quantum.4 spkane/... ip-172-17-4-3 Running       Running 6 minutes ag
```

TIP

In the previous command, we used a filter so that the output showed only the currently running processes. By default, Docker will also show you the previous containers that were running in a tree format, so that you can see things like updates and rollbacks in the output.

If you have determined that the server at 172.17.4.3 is in need of downtime, you could drain the tasks of that node and move them to another host by modifying the `availability` state to `drain` in Swarm:

```
$ docker -H 172.17.4.1 node update --availability drain ip-172-17-4-3

ip-172-17-4-3
```

If we inspect the node, we can see that the availability is now set to `drain`.

```
$ docker -H 172.17.4.1 node inspect --pretty ip-172-17-4-3

ID:                      l9j10ijs5e4843iwghjsplqae
Hostname:                ip-172-17-4-3
Joined at:               2018-03-11 18:57:17.663732624 +0000 utc
Status:
 State:                  Ready
 Availability:           Drain
 Address:                172.17.4.3
Platform:
 Operating System:       linux
 Architecture:           x86_64
Resources:
 CPUs:                   2
 Memory:                 7.797GiB
Plugins:
 Log:            awslogs, fluentd, gcplogs, gelf, journald, json-file, ...
 Network:                bridge, host, macvlan, null, overlay
 Volume:                 local
Engine Version:          17.12.1-ce
```

```
TLS Info:
 TrustRoot:
...

 Issuer Subject:        ...
 Issuer Public Key:     ...
```

You might be wondering what effect that has on the service. We told one of the nodes to stop running copies of the service, and they either have to go away or migrate somewhere else. What did it do? We can look at the details of our service again and see that all the running containers on that host have been moved to a different node.

```
$ docker -H 172.17.4.1 service ps -f "desired-state=running" quantum

ID   NAME      IMAGE      NODE          DESIRED STATE CURRENT STATE
f... quantum.1 spkane/... ip-172-17-4-1 Running       Running 19 minutes a
7... quantum.2 spkane/... ip-172-17-4-1 Running       Running 19 minutes a
r... quantum.3 spkane/... ip-172-17-4-2 Running       Running 19 minutes a
4... quantum.4 spkane/... ip-172-17-4-2 Running       Running 5 minutes ag
```

At this point, it is safe to bring down the node and do whatever work is required to make it healthy again. When you are ready to add the node back into the Swarm cluster, you can do so by running:

```
$ docker -H 172.17.4.1 node update --availability active ip-172-17-4-3

ip-172-17-4-3
```

We'll spare you from reinspecting the node at the moment, but you can always rerun the `node inspect` command if you want to see what this looks like.

WARNING

When you add a node back to the cluster, containers will not automatically balance! However, a new deploy or update should result in the containers being evenly spread across the nodes.

Once you are done, you can remove your service with the following command:

```
$ docker -H 172.17.4.1 service rm quantum

quantum
```

and then verify that it is indeed completely gone:

```
$ docker -H 172.17.4.1 service ps quantum

no such service: quantum
```

That's all for now! At this point, you can safely tear down all of the servers that were a part of your Swarm cluster if you no longer need them.

That was kind of a whirlwind tour, but covers the basics of using Swarm mode in Docker Engine and should help get you started building your own Docker cluster for deployment.

Amazon ECS and Fargate

One of the most popular cloud providers is Amazon via their AWS offering. Support for running containers natively has existed in Elastic Beanstalk since mid-2014. But that service assigns only a single container to an Amazon instance, which means that it's not ideal for short-lived or lightweight containers. EC2 itself is a great platform for hosting your own Docker environment, though, and because Docker is powerful, you don't necessarily need much on top of your instances to make this a productive environment to work in. But Amazon has spent a lot of engineering time building a service that treats containers as first-class citizens: the Elastic Container Service (ECS). In the last few years they have built upon this support with products like the ECS for Kubernetes (EKS) and, more recently, Fargate.

> **NOTE**
>
> Fargate is simply a marketing label Amazon uses for the new feature of ECS that makes it possible for AWS to automatically manage all the nodes in your container cluster so that you can focus on deploying your service.

The Elastic Container Service is a set of tools that coordinates several AWS components. With ECS, you have a choice of whether or not you will run the Fargate tooling on top. If you do, then you don't need to handle as much of the work. If you don't, then in addition to the cluster nodes to handle your workload you will also need to add one or more EC2 instances to the cluster running Docker and Amazon's special ECS agent. If you run Fargate, then the cluster is automatically managed for you. In either case, you spin up the cluster and then push your containers into it.

The Amazon ECS agent we just mentioned works with the ECS service to coordinate your cluster and schedule containers to your hosts. You will only be directly exposed to this when you manage a traditional non-Fargate ECS cluster.

Core AWS Setup

The rest of this section assumes that you have access to an AWS account and some familiarity with the service. You can learn about pricing and create a new account at aws.amazon.com/free/. Amazon offers a free service tier, which may be enough for you to experiment with if you don't already have a paid account. After you have your AWS account set up, you will need at least one administrative user, a key pair, a virtual private cloud (VPC), and a default security group in your environment. If you do not already have these set up, follow the directions in the Amazon documentation.

IAM Role Setup

AWS's Identity and Access Management (IAM) roles are used to control what actions a user can take within your cloud environment. We need to make sure we can grant access to the right actions before moving on with the Elastic Container Service.

To work with the ECS, you need to create a role called `ecsInstanceRole` that has the `AmazonEC2ContainerServiceforEC2Role` managed role attached to it. The easiest way to do this is by logging into the AWS console and then navigating to Identity and Access Management.

> **TIP**
>
> Check to ensure that you don't already have the proper role. If it already exists, then you should double-check that it is set up properly, as these directions have changed a bit over the years.

1. In the left sidebar, click Roles.

2. Then, click the "Create role" button.

3. Under AWS Service, select Elastic Container Service.

4. Under "Select your use case," select Elastic Container Service.

5. Click Next: Permissions.

6. Click Next: Review.

7. In Role Name, type: **`ecsInstanceRole`**.

8. Click "Create role."

If you are interested in storing container configuration in an S3 object storage bucket, take a look at the Amazon ECS Container Agent Configuration documentation.

AWS CLI Setup

Amazon supplies command-line tools that make it easy to work with their API-driven infrastructure. You will need to install a very recent version of the AWS command-line interface (CLI) tools. Amazon has detailed documentation that covers installation of their tools, but the basic steps are as follows.

INSTALLATION

Here we'll cover installation on a few different OSes. You can feel free to skip to the one you care about. If you're curious or just like installation instructions, by all means read them all!

macOS

> In Chapter 3, we discussed installing Homebrew. If you previously did this, you can install the AWS CLI using the following commands:
>
> ```
> $ brew update
> $ brew install awscli
> ```

Windows

> Amazon provides a standard MSI installer for Windows, which can be downloaded from Amazon S3 for your architecture:

- 32-bit Windows

- 64-bit Windows

Other

The Amazon CLI tools are written in Python. So on most platforms, you can install the tools with the Python pip package manager by running the following from a shell:

```
$ pip install awscli --upgrade --user
```

Some platforms won't have pip installed by default. In that case, you can use the easy_install package manager, like this:

```
$ easy_install awscli
```

CONFIGURATION

Quickly verify that your AWS CLI version is at least 1.7.0 with the following command:

```
$ aws --version

aws-cli/1.14.50 Python/3.6.4 Darwin/17.3.0 botocore/1.9.3
```

To configure the AWS CLI tool, ensure that you have access to your AWS access key ID and AWS secret access key, and then run the configure command. You will be prompted for your authentication information and some preferred defaults:

```
$ aws configure

AWS Access Key ID [None]: EXAMPLEEXAMPLEEXAMPLE
AWS Secret Access Key [None]: ExaMPleKEy/7EXAMPL3/EXaMPLeEXAMPLEKEY
Default region name [None]: us-east-1
Default output format [None]: json
```

At this point, it's a really good idea to test that the CLI tools are working correctly before proceeding. We can easily do that by running the following command to list the IAM users in your account:

```
$ aws iam list-users
```

Assuming everything went according to plan and you chose JSON as your default output format, you should get something like this:

```
{
    "Users": [
        {
            "Path": "/",
            "UserName": "administrator",
            "UserId": "ExmaPL3ExmaPL3ExmaPL3Ex",
            "Arn": "arn:aws:iam::936262807352:user/myuser",
            "CreateDate": "2017-12-14T19:33:23Z"
        }
    ]
}
```

Container Instances

The first thing you need to do after installing the required tools is to create at least a single cluster that your Docker hosts will register with when they are brought online.

> **NOTE**
>
> The default cluster name is imaginatively named "default." If you keep this name, you do not need to specify `--cluster-name` in many of the commands that follow.

The first thing you need to do is create a cluster in the container service. You will then launch your tasks in the cluster once it's up and running. For these examples, you should start by creating a cluster called `fargate-testing`:

```
$ aws ecs create-cluster --cluster-name fargate-testing
```

```
{
    "cluster": {
        "clusterArn": "arn:aws:ecs:us-east-1:1...2:cluster/fargate-tes
ting",
        "clusterName": "fargate-testing",
        "status": "ACTIVE",
        "registeredContainerInstancesCount": 0,
        "runningTasksCount": 0,
        "pendingTasksCount": 0,
        "activeServicesCount": 0
    }
}
```

Before AWS Fargate was released, you were required to create AWS EC2 instances running `docker` and the `ecs-agent` and add them into your cluster. You can still use this approach if you want (`EC2 launch type`), but Fargate makes it much easier to run a dynamic cluster that can scale fluidly with your workload.

Tasks

Now that our container cluster is set up, we need to start putting it to work. To do this, we need to create at least one task definition. The Amazon Elastic Container Service defines the term *task definition* as a list of containers grouped together.

To create your first task definition, open up your favorite editor, copy in the following JSON, and then save it as *webgame-task.json* in your current directory, as shown here:

```json
{
  "containerDefinitions": [
    {
      "name": "web-game",
      "image": "spkane/quantum-game",
      "cpu": 0,
      "portMappings": [
        {
          "containerPort": 8080,
          "hostPort": 8080,
          "protocol": "tcp"
        }
      ],
      "essential": true,
      "environment": [],
      "mountPoints": [],
      "volumesFrom": []
    }
  ],
  "family": "fargate-game",
  "networkMode": "awsvpc",
  "volumes": [],
  "placementConstraints": [],
  "requiresCompatibilities": [
    "FARGATE"
  ],
  "cpu": "256",
  "memory": "512"
}
```

TIP

You can also check out these files and a few others by running:

```
git clone \
    https://github.com/bluewhalebook/\
docker-up-and-running-2nd-edition.git
```

The URL above has been continued on the following line so that it fits in the margins. You may find that you need to re-assemble the URL and remove the backslashes for the command to work properly.

In this task definition, we are saying that we want to create a task family called `fargate-game` running a single container called `web-game` that is based on the Quantum web game. This Docker image launches a browser-based puzzle game that uses real quantum mechanics.

TIP

Fargate limits some of the options that you can set in this configuration, including `networkMode` and the `cpu` and `memory` settings. You can find more out about the options in the task definition from the official AWS documentation.

In this task definition, we define some constraints on memory and CPU usage for the container, in addition to telling Amazon whether this container is essential to the task. The `essential` flag is useful when you have multiple containers defined in a task, and not all of them are required for the task to be successful. If `essential` is true and the container fails to start, then all the containers defined in the task will be killed and the task will be marked as failed. We can also use the task definition to define almost all of the typical variables and settings that would be included in a *Dockerfile* or on the `docker run` command line.

To upload this task definition to Amazon, you will need to run a command similar to what is shown here:

```
$ aws ecs register-task-definition --cli-input-json file://./webgame-task.
```

```
{
    "taskDefinition": {
        "taskDefinitionArn": "arn:aws:ecs:...:task-definition/fargate-game:1",
        "containerDefinitions": [
            {
                "name": "web-game",
                "image": "spkane/quantum-game",
                "cpu": 0,
                "portMappings": [
                    {
                        "containerPort": 8080,
                        "hostPort": 8080,
                        "protocol": "tcp"
                    }
                ],
                "essential": true,
                "environment": [],
                "mountPoints": [],
                "volumesFrom": []
            }
        ],
        "family": "fargate-game",
        "networkMode": "awsvpc",
        "revision": 2,
        "volumes": [],
        "status": "ACTIVE",
        "requiresAttributes": [
            {
                "name": "com.amazonaws.ecs.capability.docker-remote-api.1.18"
            },
            {
                "name": "ecs.capability.task-eni"
            }
        ],
        "placementConstraints": [],
        "compatibilities": [
            "EC2",
            "FARGATE"
        ],
        "requiresCompatibilities": [
            "FARGATE"
        ],
        "cpu": "256",
        "memory": "512"
    }
}
```

We can then list all of our task definitions by running the following:

```
$ aws ecs list-task-definitions
```

```
{
    "taskDefinitionArns": [
        "arn:aws:ecs:us-east-1:012345678912:task-definition/fargate-ga
me:1",
    ]
}
```

Now you are ready to create your first task in your cluster. You do so by running a command like the one shown next. The `count` argument in the command allows you to define how many copies of this task you want deployed into your cluster. For this job, one is enough.

You will need to modify the following command to reference a valid subnet ID and security-group ID from your AWS VPC. You should be able to find these in the AWS console or by using the AWS CLI commands `aws ec2 describe-subnets` and `aws ec2 describe-security-groups`. You can also tell AWS to assign your tasks a public IP address by using a network configuration similar to this:

`awsvpcConfiguration={subnets=[subnet-abcd1234],securityGroups=[sg-abcd1234],assignPublicIp=ENABLED}`.

```
$ aws ecs create-service --cluster fargate-testing --service-name \
    fargate-game-service --task-definition fargate-game:1 --desired-count
    --launch-type "FARGATE" --network-configuration \
    "awsvpcConfiguration={subnets=[subnet-abcd1234],\
    securityGroups=[sg-abcd1234]}"
```

```
{
    "service": {
        "serviceArn": "arn:aws:ecs:...:service/fargate-game-service",
        "serviceName": "fargate-game-service",
        "clusterArn": "arn:aws:ecs:...:cluster/fargate-testing",
        "loadBalancers": [],
        "status": "ACTIVE",
        "desiredCount": 1,
        "runningCount": 0,
        "pendingCount": 0,
        "launchType": "FARGATE",
        "platformVersion": "LATEST",
        "taskDefinition": "arn:aws:ecs:...:task-definition/fargate-gam
e:1",
        "deploymentConfiguration": {
            "maximumPercent": 200,
            "minimumHealthyPercent": 100
        },
        "deployments": [
            {
...
            }
        ],
        "roleArn": "arn:...role/ecs.amazonaws.com/AWSServiceRoleForECS
",
        "events": [],
        "createdAt": 1520727776.555,
        "placementConstraints": [],
        "placementStrategy": [],
        "networkConfiguration": {
            "awsvpcConfiguration": {
                "subnets": [
                    "subnet-abcd1234"
                ],
                "securityGroups": [
                    "sg-abcd1234"
                ],
                "assignPublicIp": "DISABLED"
            }
        }
    }
}
```

TIP

Fargate and the `awsvpc` network require that you have a service linked role for ECS. In the preceding output, you should see a line that ends like this:

```
"role/aws-service-role/ecs.amazonaws.com/AWSServiceRoleForECS"
```

Most of the time this will be autogenerated for you, but you can create it manually using the following command:

```
aws iam create-service-linked-role \
  --aws-service-name ecs.amazonaws.com
```

You can now list all of the services in your cluster with the following command:

```
$ aws ecs list-services --cluster fargate-testing
```

```
{
    "serviceArns": [
        "arn:aws:ecs:us-east-1:012345678912:service/fargate-game-service"
    ]
}
```

To retrieve all the details about your service, run:

```
$ aws ecs describe-services --cluster fargate-testing \
    --services fargate-game-service
```

```
{
    "services": [
        {
...
            "deployments": [
                {
                    "id": "ecs-svc/9223370516126999252",
                    "status": "PRIMARY",
                    "taskDefinition": "arn:...:task-definition/fargate-game:1",
                    "desiredCount": 1,
                    "pendingCount": 1,
                    "runningCount": 0,
                    "createdAt": 1520727776.555,
                    "updatedAt": 1520727776.555,
                    "launchType": "FARGATE",
                    "platformVersion": "1.0.0",
                    "networkConfiguration": {
                        "awsvpcConfiguration": {
                            "subnets": [
                                "subnet-abcd1234"
                            ],
                            "securityGroups": [
                                "sg-abcd1234"
                            ],
                            "assignPublicIp": "DISABLED"
                        }
                    }
                }
            ],
            "roleArn": "...role/ecs.amazonaws.com/AWSServiceRoleForECS",
            "events": [
                {
                    "id": "2781cc1c-bdae-46f3-a767-f53013cc3801",
                    "createdAt": 1520727990.202,
                    "message": "(...game-service) has reached a steady
 state."
                }
            ],
...
        }
    ],
    "failures": []
}
```

This output will tell you a lot about all the tasks in your service. In this case we have a single task, which we can see has “reached a steady state.”

> ### NOTE
>
> The `task-definition` value is a name followed by a number (`fargate-game:1`). The number is the revision. If you edit your task and reregister it with the `aws ecs register-task-definition` command, you will get a new revision, which means that you will want to reference that new revision in various commands like `aws ecs update-service`. If you don’t change that number, you will continue to launch containers using the older JSON. This versioning makes it very easy to roll back changes and test new revisions without impacting all future instances.

If you want to see what individual tasks are running in your cluster, you can run the following:

```
$ aws ecs list-tasks --cluster fargate-testing
```

```
{
    "taskArns": [
        "arn:aws:ecs:...:task/2781cc1c-bdae-46f3-a767-f53013cc3801"
    ]
}
```

Since you only have a single task in your cluster at the moment, this list is very small. You will also notice that the task ID matches the ID for the task that is listed in “a steady state” in your service.

To get more details about the individual task, you can run the following command after substituting the task ID with the correct one from your cluster:

```
$ aws ecs describe-tasks --cluster fargate-testing \
  --task 2781cc1c-bdae-46f3-a767-f53013cc3801
```

```
{
    "tasks": [
        {
            "taskArn": "arn:aws:...:task/2781cc1c-bdae-46f3-a767-f53013cc3801",
            "clusterArn": "arn:aws:ecs:...:cluster/fargate-testing",
            "taskDefinitionArn": "arn:aws:...:task-definition/fargate-game:1",
            "overrides": {
                "containerOverrides": [
                    {
                        "name": "web-game"
                    }
                ]
            },
            "lastStatus": "RUNNING",
            "desiredStatus": "RUNNING",
            "cpu": "256",
            "memory": "512",
            "containers": [
                {
...
                    "name": "web-game",
                    "lastStatus": "RUNNING",
                    "networkBindings": [],
                    "networkInterfaces": [
                        {
                            "attachmentId": "a0d40aec-...-0c4086ed10d7",

                            "privateIpv4Address": "10.11.6.240"
                        }
                    ]
                }
            ],
            "startedBy": "ecs-svc/9223370516124771373",
            "version": 4,
            "connectivity": "CONNECTED",
...
            "group": "service:fargate-game-service",
            "launchType": "FARGATE",
            "platformVersion": "1.0.0",
            "attachments": [
                {
                    "id": "a0d40aec-3469-4eb0-8bd3-0c4086ed10d7",
                    "type": "ElasticNetworkInterface",
                    "status": "ATTACHED",
                    "details": [
                        {
                            "name": "subnetId",
                            "value": "subnet-abcd1234"
                        },
                        {
                            "name": "networkInterfaceId",
                            "value": "eni-abcd1234"
                        },
                        {
                            "name": "macAddress",
                            "value": "0f:c7:7d:ac:d1:ff"
                        },
                        {
                            "name": "privateIPv4Address",
                            "value": "10.4.0.100"
                        }
                    ]
                }
            ]
        }
    ],
    "failures": []
}
```

If you notice that the `lastStatus` key is displaying a value of `PENDING`, this most likely means that your service is still starting up. You can describe the task again to ensure that it has completed transitioning into a `RUNNING` state. After verifying that the `lastStatus` key is set to `RUNNING`, you should be able to test your container.

Testing the Task

You will need a modern web browser installed on your system to connect to the container and test the web game.

In the previous output, you’ll notice that the `privateIPv4Address` for the example task was listed as `10.4.0.100`. Yours will be different, and you may also have a `publicIPv4Address` if you configured your service for that.

Ensure that you are connected to a network that can reach either the public or private IP address of your host, then launch your web browser and navigate to port 8080 on that IP address.

In the example, this URL would look like:

```
http://10.4.0.100:8080/
```

If everything is working as expected, you should be greeted by the splash screen for “The Quantum Game,” which will then be quickly followed by the puzzle board.

The official version of the game can be found at *http://quantumgame.io*.

> **NOTE**
>
> We completely understand if you get distracted at this point and stop reading for a few hours while trying to solve some puzzles and learn a little bit of quantum mechanics at the same time. The book won’t notice! Put it down, play the puzzles, pick it back up later.

Stopping the Task

Right, so we have a running task. Now let’s take a look at stopping it. To do that, you need to know the task ID. One way to obtain this is by relisting all the tasks running in your cluster.

```
$ aws ecs list-tasks --cluster fargate-testing
```

```
{
    "taskArns": [
        "arn:aws:ecs:...:task/2781cc1c-bdae-46f3-a767-f53013cc3801"
    ]
}
```

You can also obtain it from the service information:

```
$ aws ecs describe-services --cluster fargate-testing \
    --services fargate-game-service
```

```
{
...
                {
                    "id": "a63e7114-9592-417c-96cf-559b82096cbc",
                    "createdAt": 1520730006.052,
                    "message": "(service fargate-game-service) has sta
rted 1..."
                }
...
}
```

Finally, we can stop the task by running the following command with the correct task ID:

```
$ aws ecs stop-task --cluster fargate-testing \
    --task 2781cc1c-bdae-46f3-a767-f53013cc3801
```

```
{
...
        "lastStatus": "RUNNING",
        "desiredStatus": "STOPPED",
...
}
```

If you describe the task again using the same task ID, you should now see that the `lastStatus` key is set to `STOPPED`:

```
$ aws ecs describe-tasks --cluster fargate-testing \
    --task 2781cc1c-bdae-46f3-a767-f53013cc3801
```

```
{
...
            "lastStatus": "STOPPED",
            "desiredStatus": "STOPPED",
...
}
```

And finally, listing all the tasks in our cluster should return an empty set:

```
$ aws ecs list-tasks --cluster fargate-testing
```

```
{
    "taskArns": []
}
```

At this point, you can start creating more complicated tasks that tie multiple containers together and rely on the ECS and Fargate tooling to spin up hosts and deploy the tasks into your cluster as needed.

Kubernetes

Now let’s take some time to look at Kubernetes. Since its release to the public during DockerCon 2014, Kubernetes has grown rapidly and is now probably the most widely adopted of the container platforms. It is not the most mature product today—that distinction goes to Mesos, which first launched in 2009 before containers were in widespread use—but Kubernetes has a great mix of functionality and a very strong community that includes many early Docker adopters. This mix has helped significantly increase its popularity over the years. At DockerCon EU 2017, Docker, Inc. announced that Kubernetes support will be coming to the Docker Engine tooling itself. Supported Docker clients can mix deployments between Swarm and Kubernetes from the same tooling, which makes Kubernetes even more attractive as a large-scale platform for Docker deployments.

But Docker's Kubernetes implementation is only the latest in a large string of releases. Like Linux itself, Kubernetes is available in a number of distributions, both free and commercial. There are currently a few dozen that are available and supported to varying degrees. Its widespread adoption means Kubernetes now has some pretty nice tooling for running it locally, including a whole distribution that can be controlled and installed locally with a single binary: Minikube. That's the distribution we'll use to do a quick runthrough of a Kubernetes cluster. Most of the concepts you'll learn while working with Minikube can be applied to any Kubernetes implementation, so it's a great place to start.

What Is Minikube?

Minikube is a whole distribution of Kubernetes for a single instance. It runs on a virtual machine on your own computer and allows you to use all the same tooling against the cluster that you would use in a production system. In scope, it's a little bit like Docker Compose: it will let you stand up a whole stack locally. It goes one step further than Compose, though, in that it actually has all the production APIs. As a result, if you run Kubernetes in production, you can have an environment on your desktop that is reasonably close in fucntion, if not in scale, to what you are running in production.

Minikube is fairly unique in that all of the distribution is controlled from a single binary you download and run locally. It will autodetect which virtual machine (VM) manager you have locally and will set up and run a VM with all of the necessary tooling on it. That means getting started with it is pretty simple.

So let's install it!

Installing Minikube

Most of the installation is the same across all platforms because once you have the tools installed, they will be your gateway to the VM running your Kubernetes installation. Just skip to the section that applies to your operating system. Once you have the tool up and running, you can follow the shared documentation.

We need two tools to use Minikube effectively: `minikube` and `kubectl`. For the purposes of our simple installation, we're going to leverage the fact that both of these commands are static binaries with no outside dependencies, which makes them easy to install.

> **NOTE**
>
> There are a number of other ways to install Minikube. We're going to show you what we think is the simplest path on each platform. If you have strong preferences about how to do this, you should feel free to do so. On Windows, for example, you might prefer to use the Chocolatey package manager, or the Snap package system on Linux.

MACOS

Just as in Chapter 3, you will need to have Homebrew installed on your system. If you don't, go back to Chapter 3 and make sure you have it set up. Once you do, it's trivial to install the `minikube` client:

```
$ brew cask install minikube
```

This will cause Hombrew to install a new cask, and then look there for the tooling to download Minikube. It will look something like this depending, on your configuration:

```
==> Auto-updated Homebrew!
...
==> Updated Formulae
...
==> Satisfying dependencies
All Formula dependencies satisfied.
==> Downloading https://storage.googleapis.com/minikube/.../minikube-darwi
######################################################################## 1
==> Verifying checksum for Cask minikube
==> Installing Cask minikube
==> Linking Binary 'minikube-darwin-amd64' to '/usr/local/bin/minikube'.
```

That's it! Let's test to make sure it's in your path:

```
$ which minikube
```

```
/usr/local/bin/minikube
```

If you don’t get a response, you will need to make sure you have */usr/local/bin* in your `PATH` environment variable. Assuming that passes, you now have the `minikube` tool installed.

Now you just need to get `kubectl` installed, and you can do that with `brew` as well. Generally, the version of Kubernetes in Homebrew will match the current release of Minikube, so using `brew install` should help prevent mismatches:

```
$ brew install kubernetes-cli
==> Downloading https://.../kubernetes-cli-1.9.3.high_sierra.bottle.tar.gz
Already downloaded: /.../kubernetes-cli-1.9.3.high_sierra.bottle.tar.gz
==> Pouring kubernetes-cli-1.9.3.high_sierra.bottle.tar.gz
==> Caveats
Bash completion has been installed to:
  /usr/local/etc/bash_completion.d

zsh completions have been installed to:
  /usr/local/share/zsh/site-functions
==> Summary
  /usr/local/Cellar/kubernetes-cli/1.9.3: 172 files, 65.4MB
```

We’ll test that the same way we tested `minikube`:

```
$ which kubectl
/usr/local/bin/kubectl
```

We’re good to go!

WINDOWS

As with installing Docker on Windows, you’ll need to have Hyper-V or another supported virtualization platform installed in order to run the Kubernetes virtual machine. But there is no additional tooling needed on Windows. You’ll simply download the binary and put it in a place you have in your `PATH` so that you can execute it on the command line. As of this writing, the URL is hosted on googleapis, which is usually very reliable about maintaining APIs. You’ll want to rename that file to *minikube.exe* once you’ve downloaded it; otherwise, you’ll be doing a lot more typing than you probably want!

You then need to get the latest Kubernetes CLI tool, `kubectl`, in order to actually control your distribution. Unfortunately, there is not a */latest* path for downloading that. So, to make sure you have the latest version, you need to get the latest version from the website and then plug it into a

URL like this:

https://storage.googleapis.com/kubernetes-release/release/<VERSION>/bin/windows/amd64/kubectl.exe.

Once you've downloaded that, you again need to make sure it's somewhere accessible from your `PATH` to make the rest of our exploration easier.

LINUX

You can actually run Minikube on the Linux box itself. We don't recommend doing that at first, because it's harder to clean up afterward if you are just experimenting. So we need to have a version of Linux that has either KVM (Linux's Kernel-based Virtual Machine) or VirtualBox. Because `minikube` is just a single binary, once you have it installed, there is no need to install any additional packages. And, because `minikube` is a statically linked binary, it should pretty much work on any distribution you want to run it on. Huzzah! We could be cool and do the installation all in a one-liner, but then if something goes wrong you might find it hard to understand the error. So let's do it in a few steps. Note that at the time of this writing the binary is hosted on googleapis, which usually maintains very stable URLs. So, here we go:

```
# Download the file, save as 'minikube'
$ curl -Lo minikube \
    https://storage.googleapis.com/minikube/releases/latest/minikube-linux-

# Make it executable
$ chmod +x minikube

# Move it to /usr/local/bin
$ sudo mv minikube /usr/local/bin/
```

You'll need to make sure that */usr/local/bin* is in your path. Now that we have `minikube` we also need to fetch `kubectl`, which we can do like this:

```
# Get the latest version number
$ KUBE_VERSION=$(curl -s \
    https://storage.googleapis.com/kubernetes-release/release/stable.txt)

# Fetch the executable
$ curl -LO \
    https://storage.googleapis.com/kubernetes-release/\
release/$(KUBE_VERSION)/bin/linux/amd64/kubectl

# Make it executable
$ chmod +x kubectl
```

```
# Move it to /usr/local/bin
$ sudo mv kubectl /usr/local/bin/
```

NOTE

One of the URLs in the previous example has been continued on the following line so that it fits in the margins. You may find that you need to re-assemble the URL and remove the back slashes for the command to work properly in your environment.

That’s it for installation—we’re ready to go.

Running Kubernetes

Now that we have the `minikube` tool, we can use it to bootstrap our Kubernetes cluster. This is normally pretty straightforward. You usually don’t need to do any configuration beforehand. You can simply run:

```
$ minikube start
Starting local Kubernetes v1.9.0 cluster...
Starting VM...
Downloading Minikube ISO
 142.22 MB / 142.22 MB [============================================] 100.
Getting VM IP address...
Moving files into cluster...
Downloading localkube binary
 162.41 MB / 162.41 MB [============================================] 100.
 0 B / 65 B [----------------------------------------------------------]
 65 B / 65 B [=====================================================] 100.
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
```

So what did we just do? Minikube packs a lot into that one command. We installed a virtual machine that has a properly configured version of Docker on it. It then runs all of the necessary components of Kubernetes inside Docker containers on the host.

We can look at those containers to see what we got:

```
$ minikube ssh
                         _             _
            _         _ ( )           ( )
  ___ ___  (_)  ___  (_)| |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| || , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)

$
```

On your Kubernetes cluster you probably won't be SSHing into the command line that often. But we want to see what's installed and get a handle on the fact that when we run `minikube`, we're controlling a virtual machine. Let's take a look at what is running on the Docker instance on our Kubernetes cluster:

```
$ docker ps
CONTAINER ID   IMAGE                              COMMAND
6039cd53ec91   gcr.io/k8s.../storage-provisioner   "/storage-provisioner"
a28e64d209f7   fed89e8b4248                        "/sidecar --v=2 --..."
e84b6d75105b   459944ce8cc4                        "/dnsmasq-nanny -v..."
539530cbe6e7   512cd7425a73                        "/kube-dns --domai..."
e73d514c68bf   e94d2f21bc0c                        "/dashboard --inse..."
21e4b12c144f   gcr.io/google_.../pause-amd64:3.0   "/pause"
696ac03d09f5   gcr.io/google_.../pause-amd64:3.0   "/pause"
47282c695e9e   gcr.io/google_.../pause-amd64:3.0   "/pause"
92b1a4d2cd0c   d166ffa9201a                        "/opt/kube-addons.sh"
97bab4a81ea8   gcr.io/google_.../pause-amd64:3.0   "/pause"
```

We won't dive too much into what each components is. But by now you should hopefully see how the mechanism works. Also, it's pretty easy to upgrade the components since they are just containers, are versioned, and can be pulled from Google's container repository.

MINIKUBE COMMANDS

In the interest of space and time, we won't go through all of the commands for `minikube`. We encourage you to run it without any options, take a look at the output, and play around with what's available. That being said, let's take a quick look at some of the most interesting commands. We'll cover a few more later in the course of installing an application stack, but here's a quick survey.

To see what was going on inside the VM, earlier we used `minikube ssh`, which is great for debugging or inspecting containers directly. Without logging into the Minikube VM over SSH, we can always check on the cluster status using another `minikube` command:

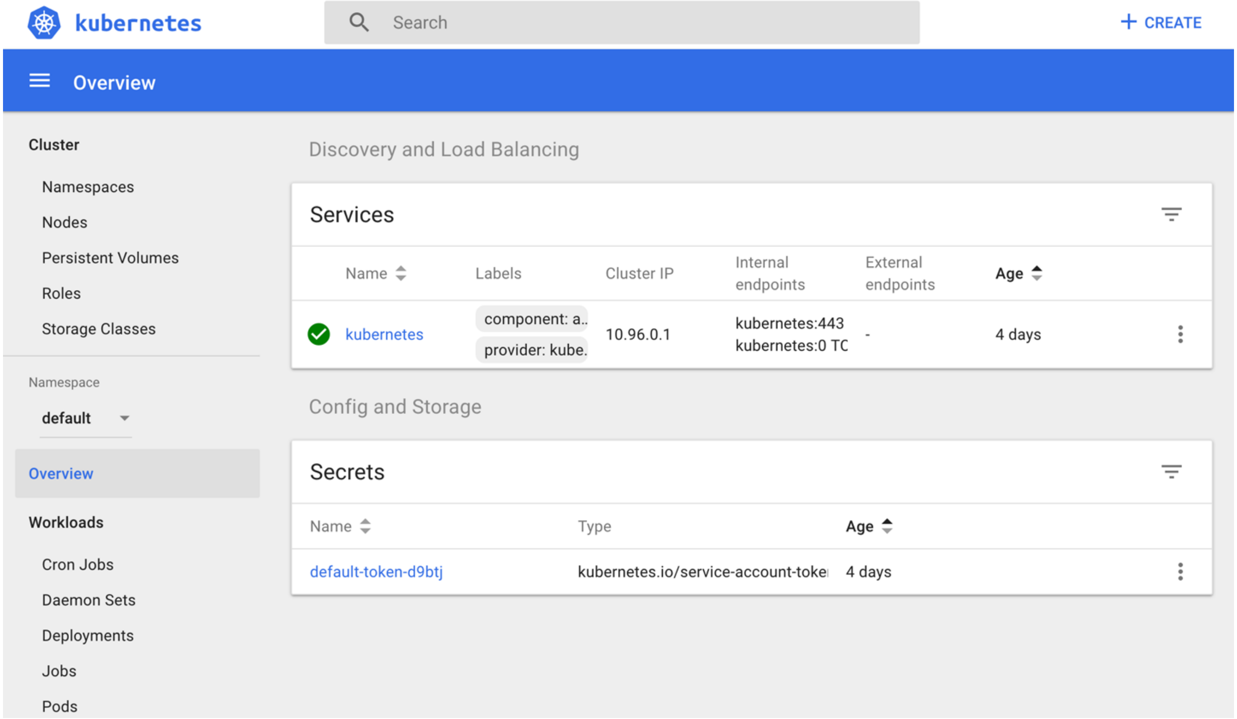
```
$ minikube status
minikube: Running
cluster: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.99.100
```

This shows us that everything is looking good, and also gives us the IP address of our virtual machine. For scripting purposes you can also retrieve just the IP address by calling `minikube ip`. At any time in the future, you can check your version of `minikube` by running `minikube update-check` and using the same mechanism you used to install it on your OS to upgrade it. Critically, the `minikube status` command also shows us that `kubectl` is installed and working properly.

We started the Kubernetes cluster with `minikube start`. As you might expect, following the style of Docker CLI arguments, `minikube stop` will stop all the Kubernetes components and the virtual machine and leave your system in a clean state.

Kubernetes Dashboard

Now that we have Minikube up and running, we don't just have the command-line tools to interact with, we actually have a whole UI installed that we can connect to. We reach it via the `minikube dashboard` command. Go ahead and run that—it should launch your web browser, pointed to the correct IP address and port of the Kubernetes dashboard! There is a lot of stuff on the dashboard and we're not able to cover it all, but you should feel free to click around and explore. Many of the terms on the left will be familiar to you at this point, and some will be totally foreign. If you don't have a computer in front of you, Figure 10-2 shows a screenshot of what an empty Minikube installation looks like from the dashboard.



Services

Name	Labels	Cluster IP	Internal endpoints	External endpoints	Age
✓ kubernetes	component: a.. provider: kube.	10.96.0.1	kubernetes:443 kubernetes:0 TC	-	4 days

Secrets

Name	Type	Age
default-token-d9btj	kubernetes.io/service-account-toke	4 days

Figure 10-2. Kubernetes dashboard

If you explore the Nodes link on the lefthand menu, you should see a single node in the cluster, named `minikube`. This is the virtual machine we started, and the dashboard, like the other components, is hosted in one of the containers we saw when we SSH'd into the host earlier. We'll take another look at the dashboard when we've actually deployed something into our cluster.

NOTE

Kubernetes exposes almost everything that you see on the dashboard with the `kubectl` command as well, which makes it very scriptable with shell scripts.

While clicking around, you may notice that Kubernetes itself shows up as a component inside the system, just like your applications will.

Kubernetes Containers and Pods

Now that we have a Kubernetes cluster up and running and you've seen how easy that is to do locally, we need to pause to talk about a concept that Kubernetes adds on top of the container abstraction. Kubernetes came out of the experiences that Google had running their own massive platform. They encountered most of the situations you might see in a production platform and had to work out concepts to make it easier to understand and solve the kinds of problems you run into when managing a large installation. In doing so, they created a complex set of new abstractions. Kubernetes embraces many of these and thus has a whole vocabulary unto itself. We won't try to get into all of these, but it's important to understand the most central of these new abstractions—a concept that sits a layer above the container and is known as a pod.

In Kubernetes parlance, a *pod* is one or more containers sharing the same cgroups and namespaces. You can also isolate the containers themselves from each other inside the same pod using cgroups and namespaces. The idea is that you may have applications that need to be deployed together all the time and that the abstraction your scheduler needs to work with is the group of applications, not just a single container. All of the containers in the pod can talk to each other on `localhost`, which eliminates any need to discover each other. So why not just deploy a big container with all the applications inside it? The advantage of a pod over a supercontainer is that you can still resource-limit the individual application separately, and leverage the large library of public Docker containers to construct your application.

Additionally, Kubernetes administrators often leverage the pod abstraction to have a container run on pod startup to make sure things are configured properly for the others, to maintain a shared resource, or to announce the application to others, for example. This allows you to make finer-grained containers than you might if you have to group things into the same container. Another nice part of the pod abstraction is the ability to share mounted volumes.

Pods have a lifespan much like a Docker container. They are essentially ephemeral and can be moved between hosts according to the lifecycle of the application or the host it runs on. Containers in a pod even share the same IP address when facing the outside world, which means they look like a single entity from the network level. Just as you would run only one instance of an application per container, you generally run one instance of that container per pod. The easiest way to think about pods is that they are a group of Docker containers that work tegether as if they were one container, for most purposes. If you need only one container, then you still get a pod deployed by Kubernetes, but that pod contains only one container. The nice thing about this is that there really is only one abstraction as far as the Kubernetes scheduler is concerned: the pod. Containers are managed by some of the runtime pieces that construct the pod and also by the configuration that you use to define them.

One critical difference between a pod and a container is that you don't construct pods in a build step. They are a runtime abstraction that lives only inside Kubernetes. So you build your Docker containers and send them to a registry, then define and deploy your pods using Kubernetes. In reality you don't usually directly describe a pod, either; the tools generate it for you through the concept of a deployment. But the pod is the unit of execution and scheduling in a Kubernetes cluster. There is a lot more to it, but that's the basic concept and it's probably easiest to understand with a simple example. The pod abstraction is more complicated than thinking of your system in terms of individual containers, but it can be pretty powerful.

Let’s Deploy Something

When actually working with pods in Kubernetes, we usually manage them through the abstraction of a deployment. A deployment is just a pod definition with some health monitoring and replication. It contains the definition of the pod and a little metadata about it. So let’s look at a basic deployment and get it running.

The simplest thing we can deploy on Kubernetes is a pod that contains just one container. The Minikube project ships a sample application called `echoserver` that we can use to explore the basics of deployment on Kubernetes. We’ll call our deployment `hello-minikube` just like the Minikube documentation does.

We’ve used the `minikube` command, but to get things done on Kubernetes itself, we now need to leverage the `kubectl` command we installed earlier.

```
$ kubectl run hello-minikube --image=k8s.gcr.io/echoserver:1.4 --port=8080
deployment "hello-minikube" created
```

To see what that did for us, we can use the `kubectl get` command to list what’s now in our cluster. We’ll trim this down to the most interesting parts:

```
$ kubectl get all
NAME                     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/hello-minikube    1         1         1            1           2m

NAME                           DESIRED   CURRENT   READY     AGE
rs/hello-minikube-c6c6764d     1         1         1         2m

NAME                                 READY     STATUS    RESTARTS   AGE
po/hello-minikube-c6c6764d-cjcb4     1/1       Running   0          2m
```

With that one command, Kubernetes created a deployment, a replica set to manage scaling, and a pod. We ran this a couple minutes after running the first command so our pod is in the `READY` state. If yours isn’t, just wait and run the command a couple more times until you see it become `READY`. The bottom entry is a running service that represents Kubernetes itself. But where is our service? We can’t get to it yet. It’s essentially in the same state a Docker container would be if you didn’t tell it to expose any ports. So we need to tell Kubernetes to do that for us:

```
$ kubectl expose deployment hello-minikube --type=NodePort
service "hello-minikube" exposed
```

This has now created a service we can reach and interact with. A service is a wrapper for one or more deployments of an application and can tell us how to contact the application. In this case, we get a NodePort, which works a lot like `EXPOSE` would for a container on Docker itself, but this time for the whole deployment. Let’s get Kubernetes to tell us how to get to it:

```
$ kubectl get svc
NAME             TYPE        CLUSTER-IP       EXTERNAL-IP    PORT(S)
hello-minikube   NodePort    10.101.150.217   <none>         8080:30616/TCP
kubernetes       ClusterIP   10.96.0.1        <none>         443/TCP
```

You might think you could now connect to *http://10.101.150.217:8080* to get to our service. But those addresses are not reachable from your host system because of the virtual machine in which Minikube is running. So we need to get `minikube` to tell us where to find the service:

```
$ minikube service hello-minikube --url
http://192.168.99.100:30616
```

The nice thing about this command, like many of the other Kubernetes commands, is that it is scriptable and command-line-friendly. If we want to open it with `curl` on the command line, we can just include the `minikube` command call in our request:

```
$ curl $(minikube service hello-minikube --url)
CLIENT VALUES:
client_address=172.17.0.1
command=GET
real path=/
query=nil
request_version=1.1
request_uri=http://192.168.99.100:8080/

SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001

HEADERS RECEIVED:
accept=*/*
host=192.168.99.100:30616
user-agent=curl/7.54.0
BODY:
-no body in request-
```

This uses `nginx` to grab some information from the request and play it back to us. Not the world's most exciting application, admittedly, but you can see that we get back our own address, visible inside the Minikube virtual machine, and the request we passed with `curl`.

This is really the simplest use case. We didn't really configure or define anything and relied on Kubernetes to do the right thing using its defaults. Next we'll take a look at something more complicated. But first, let's shut down our new service and deployment. It takes two commands to do that: one to remove the service and the other to delete it.

```
$ kubectl delete service hello-minikube
service "hello-minikube" deleted
$ kubectl delete deployment hello-minikube
deployment "hello-minikube" deleted
$ kubectl get all
NAME                               READY     STATUS        RESTARTS   AGE

NAME               TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
svc/kubernetes     ClusterIP   10.96.0.1    <none>        443/TCP   4d
```

Deploying a Realistic Stack

Let's now deploy something that looks more like a production stack. We'll deploy an application that can fetch PDF documents from an S3 bucket, cache them on disk locally, and rasterize individual pages to PNG images on request, using the cached document. To run this application, we'll want to write our cache files somewhere other than inside the container. We want to have them go somewhere a little more permanent and stable. And this time we want to make things repeatable, so that we're not deploying our application through a series of CLI commands that we need to remember and hopefully get right each time. Kubernetes, much like Docker Compose, lets us define our stack in one or more YAML files that contain all of the definitions we care about in one place. This is what you want in a production environment and is similar to what you've seen for the other production tools.

The service we'll now create will be called `lazyraster` (as in, "rasterize on demand"), and so each time you see that in the YAML definition, you'll know we're referring to our application. Our persistent volume will be called *cache-data*. Again, Kubernetes has a huge vocabulary that we can't entirely address here, but to make it clear what we're looking at we need to introduce two more concepts: `PersistentVolume` and `PersistentVolumeClaim`. A `PersistentVolume` is a physical resource that we provision inside the cluster. Kubernetes has support for many kinds of volumes, from local storage on a node to EBS volumes on AWS and similar on other cloud providers. It also supports NFS and other more modern network filesystems. A `PersistentVolume` stores data whose lifecycle is independent from our application or deployments. This lets us store data that persists between application deployments. For our cache, that's what we'll use. A `PersistentVolumeClaim` is a link between the physical resource of the `PersistentVolume` and the application that needs to consume it. We can set a policy on the claim that allows either a single read/write claim or many read claims. For our application we just want a single read/write claim to our *cache-data* `PersistentVolume`.

TIP

If you want more detail about some of the concepts we've talked about here, the Kubernetes project maintains a glossary of all the terms involved in operating Kubernetes. This can be really helpful. Each entry in the glossary is also linked to much more in-depth detail on other pages.

So let's take a look at that big YAML file, which we'll call *lazyraster-service.yaml* when it's all pasted together. We'll break it into sections so we can explain each part more easily. You can either pass each of these to `kubectl` separately or join them together into a single file, separated by `---`, and pass them all at once, as we'll do next. But it's a lot easier to reason about the definitions separately.

TIP

You can also check out these files and a few others by running:

```
git clone \
    https://github.com/bluewhalebook/\
docker-up-and-running-2nd-edition.git
```

The URL above has been continued on the following line so that it fits in the margins. You may find that you need to re-assemble the URL and remove the back slashes for the command to work properly.

Service Definition

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: lazyraster
  labels:
    app: lazyraster
spec:
  type: NodePort
  ports:
    - port: 8000
      targetPort: 8000
      protocol: TCP
  selector:
    app: lazyraster
```

The first section defines our `Service`. The second and third sections, which we'll see in a moment, respectively define our `PersistentVolumeClaim` and then our actual `Deployment`. We've told Kubernetes that our service will be called `lazyraster` and that it will be exposed on port 8000 which maps to the actual 8000 in our container. We've exposed that with the `NodePort` mechanism, which simply makes sure that our application is exposed on the same port on each host, much like the `-p` flag to Docker itself. This is helpful with `minikube` since we'll run only one instance, and the `NodePort` type makes it easy for us to access it from our own computer just like we did earlier. As with many part of Kubernetes, there are a number of options other than `NodePort`, and you can probably find a mechanism that's ideal for your production environment. `NodePort` is good for `minikube`, but it might work well for more statically configured load balancers as well.

So, back to our `Service` definition. The `Service` is going to be connected to the `Deployment` via the *selector*, which we apply in the *spec* section. Kubernetes widely uses labels as a way to reason about similar components and to tie them together. Labels are key/value pairs that are arbitrarily defined and which can then be queried to identify pieces of your system. Here the *selector* tells Kubernetes to look for `Deployments` with the label `app: lazyraster`. Notice that we also apply the same label to the `Service` itself. That's helpful if we want to identify all the components together later, but it's the *selector* section that actually ties the `Deployment` to our `Service`. Great, we have a `Service`, but it doesn't do anything yet. We need more definitions to make Kuberenetes do what we want.

PersistentVolumeClaim Definition

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cache-data-claim
  labels:
    app: lazyraster
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

The next section defines our `PersistentVolumeClaim` and likewise the `PersistentVolume` that backs it. A `PersistentVolumeClaim` is a way to name a volume and claim that you have a token to access that particular volume in a particular way. Notice, though, that we didn’t actually define the `PersistentVolume` here. That’s because Kubernetes is doing that work for us using what it calls *Dynamic Volume Provisioning*. In our case the use is pretty simple: we want a read/write claim to a volume and we’ll let Kubernetes put that in a volume container for us. But you can imagine a scenario where an application is going be to deployed into a cloud provider and where dynamic provisioning would really come into its own. In that scenario, we really don’t want to have to make separate calls to have our volume created in the cloud for us. We want Kubernetes to handle that. That’s what Dynamic Volume Provisioning is all about. Here it will just create a container for us to hold our persistent data, and mount it into our pod when we stake our claim. We don’t do a lot in this section except name it, ask for 100 MB of data, and tell Kubernetes it’s a read/write mount-once-only volume.

> **NOTE**
>
> There’s a large number of possible volume providers in Kubernetes. Which ones are available to you is in part determined by which provider or cloud service you are running on. You should take a look and see which ones make the most sense for you when you are preparing to head into production.

Deployment Definition

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: lazyraster
  labels:
    app: lazyraster
spec:
  selector:
    matchLabels:
      app: lazyraster
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: lazyraster
    spec:
      containers:
      - image: relistan/lazyraster:demo
        name: lazyraster
        env:
        - name: RASTER_RING_TYPE
          value: memberlist
        - name: RASTER_BASE_DIR
          value: /data
        ports:
        - containerPort: 8000
          name: lazyraster
        volumeMounts:
        - name: cache-data
          mountPath: /data
      volumes:
      - name: cache-data
        persistentVolumeClaim:
          claimName: cache-data-claim
```

The `Deployment` actually creates the pods for us and uses the Docker container for our application. We define some metadata about the application, including its name and a label, just like we did for the other definitions. We also apply another `selector` here to find the other resources we're tied to. In the `strategy` section, we say we want to have a `RollingUpdate`, which is a strategy that causes our pods to be cycled through one-by-one during deployment. We could also pick `Recreate`, which would simply destroy all existing pods and then create new ones afterward.

In the `template` section, we define how to actually stamp out copies of this deployment. The container definition includes the Docker image name, the ports to map, volumes to mount, and some environment variables that the `lazyraster` application needs. The very last part of the `spec` asks to have our `PersistentVolumeClaim` named `cache-data-claim`.

And that's it for the application definition. Now let's stand it up!

NOTE

There are many more options and a rich set of directives you can specify here to tell Kubernetes how to handle your application. We've walked through a couple of simple options, but we encourage you to explore the Kubernetes documentation to learn more.

Deploying the Application

Before we continue, let's see what's in our Kubernetes cluster, using the `kubectl` command:

```
$ kubectl get all
NAME             TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
svc/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   14d
```

We have only one thing defined at the moment, a service called `svc/kubernetes`. A naming convention used widely in Kubernetes is to preface the type of object with the two- or three-letter abbreviation for it. Here, `svc` means it's a `Service`. So let's go ahead and get our service, deployment, and volume into the cluster!

```
$ kubectl create -f lazyraster-service.yaml
service "lazyraster" created
persistentvolumeclaim "cache-data-claim" created
deployment "lazyraster" created
```

That output looks like what we expected: we got a service, a persistent volume claim, and a deployment. So let's see what's in the cluster now:

```
$ kubectl get all
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/lazyraster   1         1         1            1           33s

NAME                       DESIRED   CURRENT   READY     AGE
rs/lazyraster-6b8dd8cb66   1         1         1         33s

NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/lazyraster   1         1         1            1           33s

NAME                       DESIRED   CURRENT   READY     AGE
rs/lazyraster-6b8dd8cb66   1         1         1         33s

NAME                             READY     STATUS    RESTARTS   AGE
```

```
po/lazyraster-6b8dd8cb66-mfzsr   1/1       Running   0          33s

NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
svc/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP
svc/lazyraster   NodePort    10.100.32.231   <none>        8000:32185/TCP
```

You can see that a bunch more happened behind the scenes. And also, where is our volume or persistent volume claim? We have to ask for that separately:

```
$ kubectl get pvc
NAME       STATUS   VOLUME          CAPACITY   ACCESS MODES   STORAGECLASS   AGE
cache-...  Bound    pvc-b0582...    100Mi      RWO            standard       2m
```

> **TIP**
>
> `kubectl get all` does nothing of the sort. It would be more aptly named `get all-the-most-common-resources`, but there are a number of other resources you can fetch. The Kubernetes project hosts a handy cheat sheet to make this more discoverable.

So what about that `rs` and `po` that appeared in the `get all` output? Those are a `ReplicaSet` and `Pod`, respectively. A `ReplicaSet` is a piece of Kubernetes that is responsible for making sure that our application is running the right number of instances all the time and that they are healthy. We don't normally have to worry about what happens inside the `ReplicaSet` because the `Deployment` we created manages one of them for us. You can actually manage the `ReplicaSet` yourself if need be, but most of the time you won't need to or want to.

We didn't tell `kubectl` any specific number of instances, so we got one. And we can see that both the desired and current states match. We'll take a look at that in a moment. But first, let's connect to our application and see what we've got.

```
$ minikube service --url lazyraster
http://192.168.99.100:32185
```

You will probably get a different IP address and port back. That's totally fine! This is very dynamic stuff. And that's why we use the `minikube` command to manage it for us. So grab the address that came back, open your web browser, and paste it into the URL bar like this: *http://<192.168.99.100:32185>/documents/docker-up-and-running-public/sample.pdf?page=1*. You'll need to substitute the IP and port into the URL to make it work for you.

You'll need to be connected to the internet because the `lazyraster` application is going to go out to the internet, fetch a PDF from a public S3 bucket, and then render in the first page from the document as a PNG in a process called *rasterization*. If everything worked, you should see a copy of the front cover of this book! This particular PDF has two pages, so feel free to try changing the argument to `?page=2`. If you do that, you may notice it renders *much* faster than the first page. That's because the application is using our persistent volume to cache the data. You can also specify `width=2048`, or ask for a JPEG instead of a PNG with `?imageType=image/jpeg`. You could rasterize the front page as a very large JPEG like this:

http://<192.168.99.100:32185>/documents/docker-up-and-running-public/sample.pdf?page=1&imageType=image/jpeg&width=2048

If you have a public S3 bucket with other PDFs in it, you can simply substitute the bucket name for `docker-up-and-running-public` in the URL to hit your own bucket instead. If you want to play with the application some more, check out its repo on GitHub.

Scaling Up

In real life you don't just deploy applications, you operate them as well. One of the huge advantages of scheduled workloads is the ability to scale them up and down at will, within the resource constraints available to the system. In our case we only have the one Minikube node, but we can still scale up our service to better handle load and provide more reliability during deployments. Kubernetes, as you might imagine, allows scaling up and down quite easily. For our service we will need only one command to do it. Then we'll take another look at the `kubectl` output and also at the Kubernetes dashboard we introduced earlier so we can prove that the service scaled.

In Kubernetes the thing we will scale is not the `Service`, it's the `Deployment`. Here's what that looks like:

```
$ kubectl scale --replicas=2 deploy/lazyraster
deployment "lazyraster" scaled
```

Great, that did something! But what did we get?

```
$ kubectl get deploy/lazyraster
```

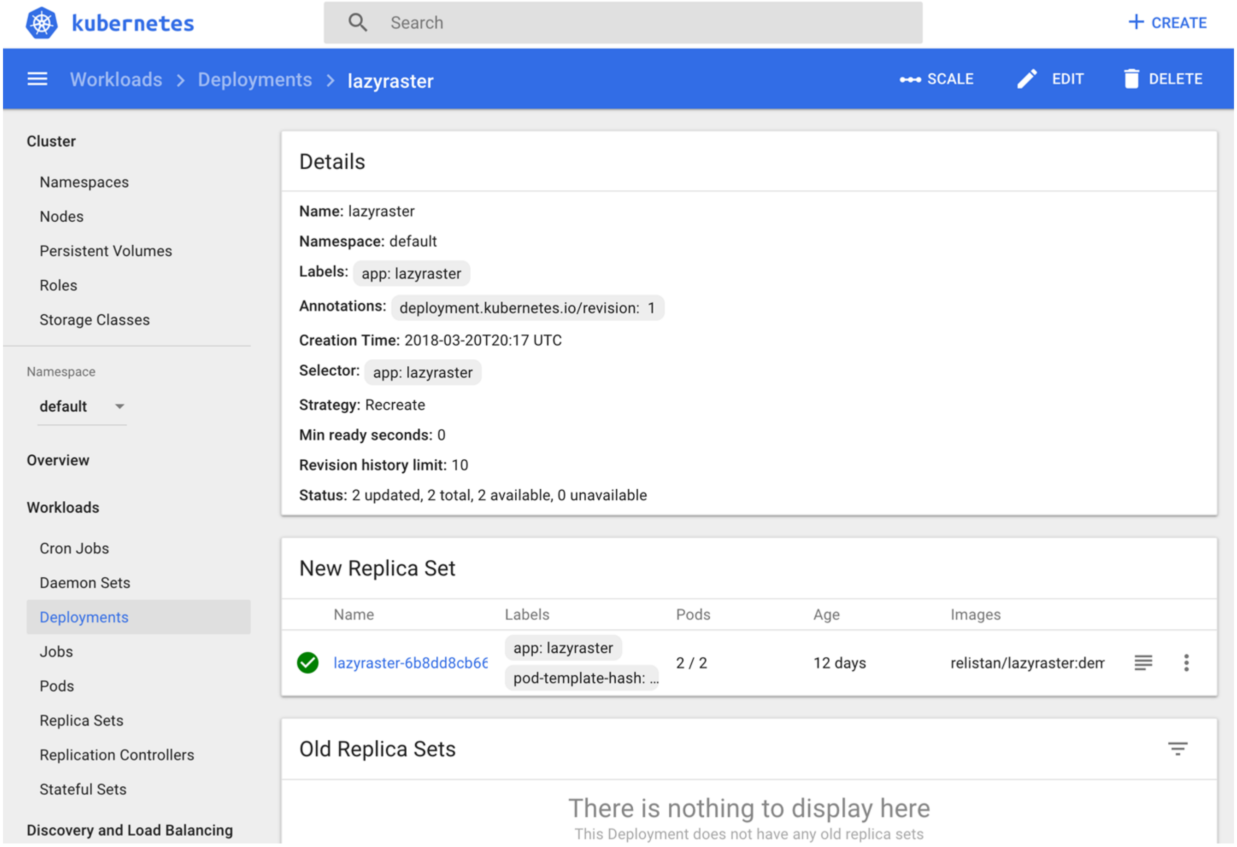
```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
lazyraster   2         2         2            2           12d
```

We now have two instances of our application running. Let’s see what we got in the logs:

```
$ kubectl logs deploy/lazyraster
Found 2 pods, using pod/lazyraster-6b8dd8cb66-mfzsr
Trying to clear existing Lazyraster cached files (if any) in the backgroun
Launching Lazyraster service...
time="..." level=info msg="Settings -----------------------"
time="..." level=info msg="  * BaseDir: /data"
time="..." level=info msg="  * HttpPort: 8000"
time="..." level=info msg="  * AdvertiseHttpPort: 8000"
...
time="..." level=info msg=------------------------------------------------
time="..." level=info msg="No New Relic license found, not starting an age
time="..." level=warning msg="Ring manager was nil in delegate!"
time="..." ... msg="No URL signing secret passed...running in insecure mod
time="..." level=info msg="Listening on tcp://:6379"
time="..." ... msg="Using anonymous credentials since no AWS env vars foun
```

We asked for logs for the deployment, but Kubernetes tells us there are two running so it picked one, the most recent in this case. We can see our new replica starting up. If we want to specify a particular instance to look at, we can ask for the logs for that pod with something like `kubectl logs po/lazyraster-6b8dd8cb66-rqmfx`, using the output from `kubectl get po` to find the pod in question.

We now have a couple of copies of our application running. What does that look like on the Kubernetes dashboard? Let’s navigate there with `minikube dashboard`. Once we’re there, we’ll click on the `layraster` Deployment and should see a screen that looks like Figure 10-3.



Details

Name: lazyraster
Namespace: default
Labels: app: lazyraster
Annotations: deployment.kubernetes.io/revision: 1
Creation Time: 2018-03-20T20:17 UTC
Selector: app: lazyraster
Strategy: Recreate
Min ready seconds: 0
Revision history limit: 10
Status: 2 updated, 2 total, 2 available, 0 unavailable

New Replica Set

Name	Labels	Pods	Age	Images
✓ lazyraster-6b8dd8cb6€	app: lazyraster pod-template-hash: ...	2 / 2	12 days	relistan/lazyraster:dem

Old Replica Sets

There is nothing to display here
This Deployment does not have any old replica sets

Figure 10-3. Lazyraster service dashboard

We encourage you to click around some more in the Kubernetes dashboard to see what else is presented. With the concepts you've picked up here, there should be a lot that is clearer now and you can probably figure out some more on your own. Likewise, `kubectl` has a lot of other options available as well, many of which you'll need in a real production system. The cheat sheet we linked earlier is a real lifesaver here!

kubectl API

We haven't shown you an API yet and, as we've discussed with Docker, it can be really useful to have a simple API to interact with for scripting and operational needs. You could write programs to talk to the individual components yourself. But for most cases, you can use `kubectl` as a nice proxy to Kubernetes, and it presents a clean API that is accessible with `curl` and JSON command-line tools. Here's an example of what you can do:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

We've now got `kubectl` itself serving up a web API on the local system! You'll need to read more about what's possible, but let's get it to show us the individual instances of the `lazyraster` application. We can do that by opening the following URL in a browser, or with `curl`: *http://localhost:8001/api/v1/namespaces/default/endpoints/lazyraster*.

There is a lot of output here, but the part we care about is the `subsets` section:

```
{
...
   "subsets" : [
      {
         "addresses" : [
            {
               "targetRef" : {
                  "resourceVersion" : "291386",
                  "kind" : "Pod",
                  "namespace" : "default",
                  "uid" : "b05b79e4-2c7b-11e8-915a-0800274dbc88",
                  "name" : "lazyraster-6b8dd8cb66-mfzsr"
               },
               "nodeName" : "minikube",
               "ip" : "172.17.0.2"
            },
            {
               "ip" : "172.17.0.5",
               "nodeName" : "minikube",
               "targetRef" : {
                  "resourceVersion" : "291702",
                  "kind" : "Pod",
                  "uid" : "de00121d-36aa-11e8-b37c-0800274dbc88",
                  "namespace" : "default",
                  "name" : "lazyraster-6b8dd8cb66-rqmfx"
               }
            }
         ],
         "ports" : [
            {
               "protocol" : "TCP",
               "port" : 8000
            }
         ]
      }
   ]
}
```

What’s interesting here is that we can see that both instances are running on the Minikube host, and that they have different IP addresses. If we were building a cloud-native application that needed to know where the other instances of the application were running, this would be a good way to do that. We could, in fact, run a copy of `kubectl proxy` in our pod and talk to it directly. We would normally use something like the `hyperkube` container to do that. If you are interested further, you can explore the documentation some more.

TIP

Kubernetes is a really big system, with great community involvement. There is a big overlap with the Docker ecosystem, but it has also developed a number of components of its own. There is increasing integration between Docker itself and Kubernetes. We've just shown you the tip of the iceberg with Minikube, but if you are interested there are many other commercial and free distributions to explore.

Wrap-Up

Well, in this chapter we've certainly presented you with a lot of options! It's unlikely that you'll ever need to use all of these, since many of them overlap. However, each one has its own perspective on exactly what a production system should look like and what problems are the most important to solve. After exploring all of these tools, you should have a pretty good idea of the wide range of options that you can choose from to build your own production Docker environment.

Underlying all of these tools is the Docker bedrock: its highly portable container format and its ability to abstract away so much of the underlying Linux system makes it easy to move your applications fluidly between your data center and as many cloud providers as you want, or to build your own approximation of a cloud system on your own premises. Now you just have to choose which approach will work best for you and your organization and then implement it.

In the meantime, let's jump into the next chapter and explore some of the most technical topics in the Docker ecosystem, including security, networking, and storage.

[1] At the time of writing, the supported clients included Docker Community Edition for Mac and Windows and Docker Enterprise Edition (EE) for Linux.

Chapter 11. Advanced Topics

In this chapter, we'll do a quick pass through some of the more advanced topics. We're going to assume that you have a pretty good hold on Docker by now and that you've already got it in production or you're at least a regular user. We'll talk about how containers work in detail, and about some aspects of Docker security, Docker networking, Docker plug-ins, swappable runtimes, and other advanced configuration.

Some of this chapter covers configurable changes you can make to your Docker installation. These can be useful, but Docker has good defaults, so as with most software, you should stick to the defaults on your operating system unless you have a good reason to change them and have educated yourself on what those changes mean to you. Getting your installation right for your environment will likely involve some trial and error, tuning, and adjustment over time. However, changing settings from their default before understanding them well is not what we recommend.

Containers in Detail

Though we usually talk about Linux containers as a single entity, they are, in fact, implemented through several separate mechanisms built into the Linux kernel that all work together: control groups (cgroups), namespaces, and SELinux or AppArmor, all of which serve to *contain* the process. cgroups provide for resource limits, namespaces allow for processes to use identically named resources and isolate them from each other's view of the system, and SELinux or AppArmor provides strong security isolation. We'll talk about SELinux and AppArmor in a bit. But what do cgroups and namespaces do for you?

Before we launch into detail, another comparison might be in order to help you understand how each of these subsystems plays into the way that containers work. At the risk of mixing metaphors, we might make a comparison to a hotel. When running Docker, your computer is the hotel. Without Docker, it's more like a hostel with open bunk rooms. In our modern hotel, each container that you launch is an individual room with one or more guests (our processes) in it.

Namespaces make up the walls of the room, and ensure that processes cannot interact with neighboring processes in any ways that they are not specifically allowed to. Control groups are like the floor and ceiling of the room, trying to ensure that the inhabitants have the resources they need to enjoy their stay, without allowing them to use resources or space reserved for others. Imagine the worst kind of noisy hotel neighbors and you can really appreciate good, solid barriers between rooms. Finally, SELinux and AppArmor are a bit like hotel security, ensuring that even if something unexpected or untoward happens, it is unlikely to cause much more than the headache of filling out paperwork and filing an incident report.

cgroups

Traditional distributed system design dictates running each intensive task on its own virtual server. So, for example, you don't run your applications on the database server because they have competing resource demands and their resource usage could grow unbounded and begin to dominate the server, starving the database of performance.

On real hardware systems, this could be quite expensive and so solutions like virtual servers are very appealing, in part because you can share expensive hardware between competing applications, and the virtualization layer will handle your resource partitioning. But while it saves money, this is still a fairly expensive approach if you don't need all the other separation provided by virtualization, because running multiple kernels introduces a reasonable overhead on the applications. Maintaining virtual machines is also not the cheapest solution. All the same, cloud computing has shown that it's immensely powerful and, with the right tooling, incredibly effective.

But if the only kind of isolation you needed was resource partitioning, wouldn't it be great if you could get that on the same kernel? For many years, you could assign a "niceness" value to a process and it would give the scheduler hints about how you wanted this process to be treated in relation to others. But it wasn't possible to impose hard limits like those that you get with virtual machines. And niceness is not at all fine-grained: you can't give something more I/O and less CPU than other processes. This fine-grained control, of course, is one of the promises of Docker and the mechanism that it uses to provide that is cgroups, which predate Docker and were invented to solve just that problem.

Control groups, or *cgroups* for short, allow you to set limits on resources for processes and their children. This is the mechanism that Docker uses to control limits on memory, swap, CPU, and storage and network I/O resources. cgroups are built into the Linux kernel and originally shipped back in 2007 in Linux 2.6.24. The official kernel documentation defines them as "a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behavior." It's important to note that this setting applies to a process and all of the children that descend from it. That's exactly how containers are structured.

Every Docker container is assigned a cgroup that is unique to that container. All of the processes in the container will be in the same group. This means that it’s easy to control resources for each container as a whole without worrying about what might be running. If a container is redeployed with new processes added, you can have Docker assign the same policy and it will apply to all of them.

We talked previously about the cgroups hooks exposed by Docker via the Remote API. This allows you to control memory, swap, and disk usage. But there are lots of other things you can limit with cgroups, including the number of I/O operations per second (iops) a container can have, for example. You might find that in your environment you need to use some of these levers to keep your containers under control, and there are a few ways you can go about doing that. By their nature, cgroups need to do a lot of accounting of resources used by each group. That means that when you’re using them, the kernel has a lot of interesting statistics about how much CPU, RAM, disk I/O, and so on your processes are using. So Docker uses cgroups not just to limit resources but also to report on them. These are many of the metrics you see, for example, in the output of `docker stats`.

THE /SYS FILESYSTEM

The primary way to control cgroups in a fine-grained manner, even if you configured them with Docker, is to manage them yourself. This is the most powerful method because changes don’t just happen at creation time—they can be done on the fly.

On systems with `systemd`, there are command-line tools like `systemctl` that you can use to do this. But since cgroups are built into the kernel, the method that works everywhere is to talk to the kernel directly via the */sys* filesystem. If you’re not familiar with */sys*, it’s a filesystem that directly exposes a number of kernel settings and outputs. You can use it with simple command-line tools to tell the kernel how to behave in a number of ways.

Note that this method of configuring cgroups controls for containers works only directly on the Docker server, so it is not available remotely via any API. If you use this method, you’ll need to figure out how to script this for your own environment.

WARNING

Changing cgroups values yourself, outside of any Docker configuration, breaks some of the repeatability of Docker deployment. Unless you tool changes into your deployment process, settings will revert to their defaults when containers are replaced. Some schedulers take care of this for you, so if you run one in production you might check the documentation to see how to best apply these changes repeatably.

Let’s use an example of changing the CPU cgroups settings for a container we already have running. First we need to get the long ID of the container, and then we need to find it in the */sys* filesystem. Here’s what that looks like:

```
$ docker ps --no-trunc
CONTAINER ID IMAGE        COMMAND        CREATED     STATUS     NAMES
dcbbaa763... 0415448f2cc2 "supervisord" 3 weeks ago Up 2 days romantic_mor
```

Here, we’ve had `docker ps` give us the long ID in the output, and the ID we want is `dcbbaa763daff1dc0a91e7675d3c93895cb6a6d83371e25b7f0bd62803ed8e86`. You can see why Docker normally truncates this. In the examples we’re going to truncate it, too, to make it at least a little readable and fit into the constraints of a standard page. But remember that you will need to use the long one!

Now that we have the ID, we can find our container’s cgroup in the */sys* filesystem. */sys* is laid out so that each type of setting is grouped into a module and that module is exposed at a different place in the */sys* filesystem. So when we look at CPU settings, we won’t see `blkio` settings, for example. You might take a look around in the */sys* to see what else is there. But for now we’re looking at the CPU controller, so let’s inspect what that gives us. You need `root` access on the system to do this because you’re manipulating kernel settings:

```
$ ls /sys/fs/cgroup/cpu/docker/dcbbaa763daf
cgroup.clone_children  cpu.cfs_period_us  cpu.rt_runtime_us  notify_on_rel
cgroup.event_control   cpu.cfs_quota_us   cpu.shares         tasks
cgroup.procs           cpu.rt_period_us   cpu.stat
```

NOTE

The exact path here will change a bit depending on the Linux distribution your Docker server is running on and what the hash of your container is. For example, on CoreOS, the path would look something like:

/sys/fs/cgroup/cpu/system.slice/docker-8122be2d7a67a52e949582f6d5 cb2771a8469ab20ecf7b6915e9217d92ddde98.scope/

You can see that under cgroups, there is a *docker* directory that contains all of the Docker containers that are running on this host. You can't set cgroups for things that aren't running, because they apply only to running processes. This is an important point that you should consider. Docker takes care of reapplying cgroup settings for you when you start and stop containers. Without that mechanism, you are somewhat on your own.

Back to our task. Let's inspect the CPU shares for this container. Remember that we set these earlier via the Docker command-line tool. But for a normal container where no settings were passed, this setting is the default:

```
$ cat /sys/fs/cgroup/cpu/docker/dcbbaa763daf/cpu.shares
1024
```

1024 CPU shares means we are not limited at all. Let's tell the kernel that this container should be limited to half that:

```
$ echo 512 > /sys/fs/cgroup/cpu/docker/dcbbaa763daf/cpu.shares
$ cat /sys/fs/cgroup/cpu/docker/dcbbaa763daf/cpu.shares
512
```

WARNING

In production you should not use this method to adjust cgroups on the fly, but we are demonstrating it here so that you understand the underlying mechanics that make all of this work. Take a look at `docker container update` if you'd like to adjust these on a running container. You might also find the `--cgroup-parent` option to `docker run` interesting.

There you have it. We've changed the container's settings on the fly. This method is very powerful because it allows you to set any cgroups setting for the container. But as we mentioned earlier, it's entirely ephemeral. When the container is stopped and restarted, the setting reverts to the default:

```
$ docker stop dcbbaa763daf
dcbbaa763daf
$ cat /sys/fs/cgroup/cpu/docker/dcbbaa763daf/cpu.shares
cat: /sys/fs/.../cpu.shares: No such file or directory
```

You can see that the directory path doesn't even exist anymore now that the container is stopped. And when we start it back up, the directory comes back but the setting is back to 1024:

```
$ docker start dcbbaa763daf
dcbbaa763daf
$ cat /sys/fs/cgroup/cpu/docker/dcbbaa763daf/cpu.shares
1024
```

If you were to change these kinds of settings in a production system via the */sys* fileystem directly, you'd want to tool that directly. A daemon that watches the `docker events` stream and changes settings at container startup, for example, is a possibility. Currently, the community has not contributed much rich tooling in this area. It's likely that Docker will eventually expand the native driver's functionality to allow this level of configuration.

> **NOTE**
>
> It is possible to create custom cgroups outside of Docker and then attach a new container to that cgroup using the `--cgroup-parent` argument to `docker create`. This mechanism is also used by schedulers that run multiple containers inside the same cgroup (e.g., Kubernetes pods).

Namespaces

Inside each container, you see a filesystem, network interfaces, disks, and other resources that all appear to be unique to the container despite sharing the kernel with all the other processes on the system. The primary network interface on the actual machine, for example, is a single shared resource. But inside your container it will look like it has an entire network interface to itself. This is a really useful abstraction: it's what makes your container feel like a machine all by itself. The way this is implemented in the kernel is with namespaces. Namespaces take a traditionally global resource and present the container with its own unique and unshared version of that resource.

NOTE

Unlike cgroups, namespaces are not something that you can see reflected in the kernel filesystems, like */sys* and */proc*.

Rather than just having a single namespace, however, containers have a namespace on each of the six types of resources that are currently namespaced in the kernel: mounts, UTS, IPC, PID, network, and user namespaces. Essentially when you talk about a container, you're talking about a number of different namespaces that Docker sets up on your behalf. So what do they all do?

Mount namespaces

Docker uses these primarily to make your container look like it has its own entire filesystem namespace. If you've ever used a `chroot` jail, this is its tougher cousin. It looks a lot like a `chroot` jail but goes all the way down to the kernel so that even `mount` and `unmount` system calls are namespaced. If you use `docker exec` or `nsenter` to get into a container, you'll see a filesystem rooted on "/". But we know that this isn't the actual root partition of the system. It's the mount namespace that makes that possible.

UTS namespaces

Named for the kernel structure they namespace, UTS (Unix Timesharing System) namespaces give your container its own hostname and domain name. This is also used by older systems like NIS to identify which domain a host belongs to. When you enter a container and see a hostname that is not the same as the machine on which it runs, it's this namespace that makes that happen.

TIP

To have a container use its host's UTS namespace, you can specify the `--uts=host` option when launching the container with `docker run`. There are similar commands for sharing the other namespaces as well.

IPC namespaces

These isolate your container’s System V IPC and POSIX message queue systems from those of the host. Some IPC mechanisms use filesystem resources like named pipes, and those are covered by the mount namespace. The IPC namespace covers things like shared memory and semaphores that aren’t filesystem resources but which really should not cross the container wall.

PID namespaces

We have already shown that you can see all of the processes in containers in the Linux `ps` output on the host Docker server. But inside the container, processes have a totally different PID. This is the PID namespace in action. A process has a unique PID in each namespace to which it belongs. If you look in */proc* inside a container, or run `ps`, you will only see the processes inside the container’s PID namespace.

Network namespaces

This is what allows your container to have its own network devices, ports, and so on. When you run `docker ps` and see the bound ports for your container, you are seeing ports from both namespaces. Inside the container, your `nginx` might be bound to port 80, but that’s on the namespaced network interface. This namespace makes it possible to have what seems to be a completely separate network stack for your container.

User namespaces

These provide isolation between the user and group IDs inside a container and those on the Docker host. Earlier when we looked at `ps` output outside and then inside the container, we saw different user IDs; this is how that happened. A new user inside a container is not a new user on the Docker host’s main namespace, and vice versa. There are some subtleties here, though. For example, UID 0 (`root`) in a user namespace is not the same thing as UID 0 on the host, although running as `root` inside the container does increase the risk of potential security exploits. Some of this work is reasonably new to the Linux kernel and there are concerns about security leakage, which we’ll talk about in a bit.

So all of the namespaces combined together provide the visual and, in many cases, the functional isolation that makes a container look like a virtual machine even though it’s on the same kernel. Let’s explore what some of that namespacing that we just described actually looks like.

NOTE

There is a lot of work going into making containers more secure. The community is actively looking into ways to support rootless containers that would enable regular users to create, run, and manage containers locally without needing special privileges. New container runtimes like Google gVisor are also trying to explore better ways to create much more secure container sandboxes without losing most of the advantages of containerized workflows.

EXPLORING NAMESPACES

One of the easiest to demonstrate namespaces is the UTS namespace, so let's use `docker exec` to get a shell in a container and take a look. From within the Docker server, run the following:

```
$ hostname
docker2

$ docker exec -i -t 28970c706db0 /bin/bash -l
# hostname
28970c706db0
```

NOTE

Although `docker exec` will work from a remote system, here we `ssh` into the Docker server itself in order to demonstrate that the hostname of the server is different from inside the container. There are also easier ways to obtain some of the information we're getting here. But the idea is to explore how namespaces work, not to pick the most ideal path to gather the information.

That `docker exec` command line gets us an interactive process (`-i`), allocates a pseudo-TTY (`-t`), and then executes `/bin/bash` while executing all the normal login process in the `bash` shell (`-l`). Once we have a terminal open inside the container's namespace, we ask for the hostname and get back the container ID. That's the default hostname for a Docker container unless you tell Docker to name it otherwise. This is a pretty simple example, but it should clearly show that we're not in the same namespace as the host.

Another example that's easy to understand and demonstrate involves PID namespaces. Let's log into the Docker server again and look at the process list of a new container:

```
$ docker run -d --rm --name pstest spkane/train-os sleep 240
6e005f895e259ed03c4386b5aeb03e0a50368cc173078007b6d1beaa8cd7dded

$ docker exec -ti pstest /bin/bash -l
[root@6e005f895e25 /]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 16:35 ?        00:00:00 sleep 240
root         7     0  0 16:36 pts/0    00:00:00 /bin/bash -l
root        27     7  0 16:36 pts/0    00:00:00 ps -ef
[root@6e005f895e25 /]# exit
logout
```

and then let's get the same list from the Docker server:

```
$ ps axlf
...
 899  root       3:29 /usr/bin/containerd
 1088 root       7:12 /usr/local/bin/dockerd -H unix:///var/run/docker.soc
 1158 root       4:59 docker-containerd --config ...
 4946 root       0:00 docker-containerd-shim -namespace moby -workdir ...
 4966 root       0:00 sleep 240
...

$ pstree -p 1088
dockerd(1088)---docker-containe(1158)-+-docker-containe(4946)---sleep(4966
```

What we can see here is that from inside our container, the original command run by Docker from the CMD in our Dockerfile is `sleep 240` and it has been assigned PID 1 inside the container. You might recall that this is the PID normally used by the `init` process on Unix systems. In this case, the `sleep 240` command that we started the container with is the first process, so it gets PID 1. But in the Docker server's main namespace, we do a little work to find our container's processes and we see the PID there is not 1, but 4966, and it's a child of the `dockerd` process, which is PID 1088.

We can go ahead and remove the container we started in the last example, by running:

```
 $ docker rm -f pstest
```

The other namespaces work in essentially the same manner, and you probably get the idea by now. It's worth pointing out here that when we were working with `nsenter` back in Chapter 4, we had to pass a pretty arcane (at that point) set of arguments to the command when we ran it to enter a container from the Docker server. Let's look at that command line now:

```
$ sudo nsenter --target $PID --mount --uts --ipc --net --pid
root@3c4f916619a5:/#
```

Now that we've explained namespaces in detail, this probably makes a lot more sense to you. You're telling `nsenter` exactly which of the namespaces you want to enter. It can also be educational to use `nsenter` to only enter parts of the namespace of a throwaway container to see what you get. In the preceding example, we enter all of the namespaces we just talked about.

When it comes down to it, namespaces are the primary thing that make a container look like a container. Combine them with cgroups, and you have reasonably robust isolation between processes on the same kernel.

Security

We've spent a good bit of space now talking about how Docker contains applications, allows you to constrain resources, and uses namespaces to give the container its own unique view of the world. We have also briefly mentioned the need for technologies like SELinux and AppArmor. One of the advantages of containers is the ability to replace virtual machines in a number of cases. So let's take a look at what isolation we really get, and what we don't.

You are undoubtedly aware by now that the isolation you get from a container is not as strong as that from a virtual machine. We've been reinforcing the idea from the start of this book that containers are just processes running on the Docker server. Despite the isolation provided by namespaces, containers are not as secure as you might imagine, especially if you are still mentally comparing them to lightweight virtual machines.

One of the big boosts in performance for containers, and one of the things that makes them lightweight, is that they share the kernel of the Docker server. This is also the source of the greatest security concern around Docker containers. The main reason for this concern is that not everything in the kernel is namespaced. We have talked about all of the namespaces that exist and how the container's view of the world is constrained by the namespaces it runs in. However, there are still lots of places in the kernel where no real isolation exists and namespaces constrain the container only if it does not have the power to tell the kernel to give it access to a different namespace.

Containerized applications are more secure than noncontainerized applications because cgroups and standard namespaces provide some important isolation from the host's core resources. But you should not think of containers as a substitute for good security practices. If you think about how you would run an application on a production system, that is really how you should run all your containers. If your application would traditionally run as a nonprivileged user on a server, then it should be run in the same manner inside the container. It is very easy to tell Docker to run your whole container as a nonprivileged user, and in almost all cases, this is what you should be doing.

> **TIP**
>
> The `--userns-remap` argument to the `dockerd` command makes it possible to force all containers to run within a user and group context that is unprivileged on the host system. This protects the host from various potential security exploits. For more information about this topic, read through the official userns-remap and docker daemon documentation.

Let’s look at some common security risks and controls.

UID 0

The first and most overarching security risk in a container is that, unless you are using the `userns-remap` functionality in the Docker daemon, the `root` user in the container is actually the `root` user on the system. There are extra constraints on `root` in a container, and namespaces do a good job of isolating `root` in the container from the most dangerous parts of the */proc* and */sys* filesystems. But generally speaking, by default, you have `root` access, so if you somehow get access to protected resources on a file mount or outside of your namespace, then the kernel will treat you as `root`, and therefore give you access to the resource. Unless otherwise configured, Docker starts all services in containers as `root`, which means you are responsible for managing privileges in your applications just like you are on any Linux system. Let’s explore some of the limits on `root` access and look at some obvious holes. This is not intended to be an exhaustive statement on container security, but rather an attempt to give you a healthy understanding of some of the classes of security risks.

First, let’s fire up a container and get a `bash` shell using the public Ubuntu image shown in the following code. Then we’ll see what kinds of access we have, after installing some tools we want to run.

```
$ docker run -t -i ubuntu /bin/bash
```

```
root@808a2b8426d1:/# apt-get update
...
root@808a2b8426d1:/# apt-get install kmod
...
root@808a2b8426d1:/# lsmod
Module                  Size  Used by
xt_nat                 12726  2
xt_tcpudp              12603  8
veth                   13244  0
xt_addrtype            12713  2
xt_conntrack           12760  1
iptable_filter         12810  1
acpiphp                24119  0
ipt_MASQUERADE         12759  4
aufs                  191008  14
iptable_nat            12909  1
nf_conntrack_ipv4      14538  2
nf_defrag_ipv4         12729  1 nf_conntrack_ipv4
nf_nat_ipv4            13316  1 iptable_nat
nf_nat                 26158  4 ipt_MASQUERADE,nf_nat_ipv4
nf_conntrack           83996  6 ipt_MASQUERADE,nf_nat
ip_tables              27473  2 iptable_filter,iptable_nat
x_tables               29938  7 ip_tables,xt_tcpudp
bridge                101039  0
floppy                 70206  0
...
```

In Docker Community Edition you may see only two modules in the list, but on a normal Linux system this list can be very long, so we've cut the output down quite a bit in this example. What we're looking at here is a new container that we started. Using `lsmod`, we've just asked the kernel to tell us what modules are loaded. It is not that surprising that we get this list, since a normal user can always do this. If you run this listing on the Docker server itself, it will be identical, which reinforces the fact that the container is talking to the same Linux kernel that is running on the server. So we can see the kernel modules; what happens if we try to unload the `floppy` module?

```
root@808a2b8426d1:/# rmmod floppy

rmmod: ERROR: ... kmod_module_remove_module() could not remove 'floppy': .
rmmod: ERROR: could not remove module floppy: Operation not permitted
```

That's the same error message we would get if we were a nonprivileged user trying to tell the kernel to remove a module. This should give you a good sense that the kernel is doing its best to prevent us from doing things we shouldn't. And because we're in a limited namespace, we can't get the kernel to give us access to the top-level namespace either. We are essentially relying on the hope that there are no bugs in the kernel that allow us to escalate our privileges inside the container. Because if we do manage to do that, we are `root`, which means that we will be able to make changes, if the kernel allows us to.

We can contrive a simple example of how things can go wrong by starting a `bash` shell in a container that has had the Docker server's */etc* bind-mounted into the container's namespace. Keep in mind that anyone who can start a container on your Docker server can do what we're about to do any time they like because you can't configure Docker to prevent it, so you must instead rely on external tools like SELinux to avoid exploits like this:

```
$ docker run -i -t -v /etc:/host_etc ubuntu /bin/bash

root@e674eb96bb74:/# more /host_etc/shadow
root:!:16230:0:99999:7:::
daemon:*:16230:0:99999:7:::
bin:*:16230:0:99999:7:::
sys:*:16230:0:99999:7:::
...
irc:*:16230:0:99999:7:::
nobody:*:16230:0:99999:7:::
libuuid:!:16230:0:99999:7:::
syslog:*:16230:0:99999:7:::
messagebus:*:16230:0:99999:7:::
kmatthias:$1$aTAYQT.j$3xamPL3dHGow4ITBdRh1:16230:0:99999:7:::
sshd:*:16230:0:99999:7:::
lxc-dnsmasq:!:16458:0:99999:7:::
```

Here we've used the `-v` switch to Docker to tell it to mount a host path into the container. The one we've chosen is */etc*, which is a dangerous thing to do. But it serves to prove a point: we are `root` in the container, and `root` has file permissions in this path. So we can look at the real */etc/shadow* file any time we like. There are plenty of other things you could do here, but the point is that by default you're only partly constrained.

WARNING

It is a bad idea to run your container processes with UID 0. This is because any exploit that allows the process to somehow escape its namespaces will expose your host system to a fully privileged process. You should always run your standard containers with a nonprivileged UID. This is not a theoretical problem: it has been demonstrated publicly with several exploits.

The easiest way to deal with the potential problems surrounding the use of UID 0 inside containers is to always tell Docker to use a different UID for your container.

You can do this by passing the `-u` argument to `docker run`. In the next example we run the `whoami` command to show that we are `root` by default and that we can read the */etc/shadow* file that is inside this container.

```
$ docker run spkane/train-os:latest whoami
root

$ docker run spkane/train-os:latest cat /etc/shadow
root:locked::0:99999:7:::
bin:*:16489:0:99999:7:::
daemon:*:16489:0:99999:7:::
adm:*:16489:0:99999:7:::
lp:*:16489:0:99999:7:::
...
```

In this example, when you add `-u 500`, you will see that we become a new, unprivileged user and can no longer read the same */etc/shadow* file.

```
$ docker run -u 500 spkane/train-os:latest whoami
user500

$ docker run -u 500 spkane/train-os:latest cat /etc/shadow
cat: /etc/shadow: Permission denied
```

Privileged Containers

There are times when you need your container to have special kernel capabilities that would normally be denied to the container. These could include mounting a USB drive, modifying the network configuration, or creating a new Unix device.

In the following code, we try to change the MAC address of our container:

```
$ docker run --rm -ti ubuntu /bin/bash

root@b328e3449da8:/# ip link ls
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
9: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state ...
    link/ether 02:42:0a:00:00:04 brd ff:ff:ff:ff:ff:ff
root@b328e3449da8:/# ip link set eth0 address 02:0a:03:0b:04:0c
RTNETLINK answers: Operation not permitted
root@b328e3449da8:/# exit
```

As you can see, it doesn't work. This is because the underlying Linux kernel blocks the nonprivileged container from doing this, which is exactly what we'd normally want. However, assuming that we need this functionality for our container to work as intended, the easiest way to significantly expand a container's privileges is by launching it with the `--privileged=true` argument.

WARNING

We don't recommend actually running the `ip link set eth0 address` command in the next example, since this will change the MAC address on the container's network interface. We show it to help you understand the mechanism. Try it at your own risk.

```
$ docker run -ti --rm --privileged=true ubuntu /bin/bash

root@88d9d17dc13c:/# ip link ls
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
9: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state ...
    link/ether 02:42:0a:00:00:04 brd ff:ff:ff:ff:ff:ff
root@88d9d17dc13c:/# ip link set eth0 address 02:0a:03:0b:04:0c
root@88d9d17dc13c:/# ip link ls
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
9: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state ...
    link/ether 02:0a:03:0b:04:0c brd ff:ff:ff:ff:ff:ff
root@88d9d17dc13c:/# exit
```

In the preceding output, you will notice that we no longer get the error and the *link/ether* entry for `eth0` has been changed.

The problem with using the `--privileged=true` argument is that you are giving your container very broad privileges, and in most cases you likely need only one or two kernel capabilities to get the job done.

If we explore our privileged container some more, we will discover that we have capabilities that have nothing to do with changing the MAC address. We can even do things that could cause issues with both Docker and the host system. In the following code, we are going to create a memory swapfile and enable it:[1]

```
$ docker run -ti --rm --privileged=true ubuntu /bin/bash

root@0ffcdd8f7535:/# dd if=/dev/zero of=/swapfile1 bs=1024 count=100
100+0 records in
100+0 records out
102400 bytes (102 kB) copied, 0.00046004 s, 223 MB/s
root@0ffcdd8f7535:/# mkswap /swapfile1
Setting up swapspace version 1, size = 96 KiB
no label, UUID=fc3d6118-83df-436e-867f-87e9fbce7692
root@0ffcdd8f7535:/# swapon /swapfile1
root@0ffcdd8f7535:/# swapoff /swapfile1
root@0ffcdd8f7535:/# exit
exit
```

WARNING

In the previous example, if you do not disable the swapfile before exiting your container, you will leave your Docker host in a bad state where Docker can't destroy the container because your host is accessing a swapfile that is inside the container's filesystem.

In that case, the error message will look something like this:

```
FATAL [0049] Error response from daemon:
Cannot destroy container 0f...70:
 Driver overlay failed to remove root filesystem 0f...70:
 remove /var/lib/docker/overlay/0f...70/upper/swapfile1:
 operation not permitted
```

In this example, you can fix this from the Docker server by running:

```
$ sudo swapoff \
    /var/lib/docker/overlay/0f...70/upper/swapfile1
```

So, as we've seen, it is possible for people to do malicious things in a fully privileged container.

To change the MAC address, the only kernel capability we actually need is `CAP_NET_ADMIN`. Instead of giving our container the full set of privileges, we can give it this one privilege by launching our Docker container with the `--cap-add` argument, as shown here:

```
$ docker run -ti --rm --cap-add=NET_ADMIN ubuntu /bin/bash

root@852d18f5c38d:/# ip link set eth0 address 02:0a:03:0b:04:0c
root@852d18f5c38d:/# ip link ls
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
9: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state ...
    link/ether 02:0a:03:0b:04:0c brd ff:ff:ff:ff:ff:ff
root@852d18f5c38d:/# exit
```

You should also notice that although we can change the MAC address, we can no longer use the `swapon` command inside our container.

```
$ docker run -ti --rm --cap-add=NET_ADMIN ubuntu /bin/bash

root@848aa7924594:/# dd if=/dev/zero of=/swapfile1 bs=1024 count=100
100+0 records in
100+0 records out
102400 bytes (102 kB) copied, 0.000575541 s, 178 MB/s
root@848aa7924594:/# mkswap /swapfile1
Setting up swapspace version 1, size = 96 KiB
no label, UUID=3b365d90-8116-46ad-80c5-24341299dc41
root@848aa7924594:/# swapon /swapfile1
swapon: /swapfile1: swapon failed: Operation not permitted
root@848aa7924594:/# exit
```

It is also possible to remove specific capabilities from a container. I imagine for a moment that your security team requires that `tcpdump` be disabled in all containers, and when you test some of your containers, you find that `tcpdump` is installed and can easily be run.

```
$ docker run -ti --rm spkane/train-os:latest tcpdump -i eth0

tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
26:52:10.182287 IP6 :: > ff03::1:ff3e:f0e4: ICMP6, neighbor solicitation,
26:52:10.199257 ARP, Request who-has 1.qarestr.sub-172-14-0.myvzw.com
26:52:10.199312 ARP, Reply 1.qarestr.sub-172-14-0.myvzw.com is-at
26:52:10.199328 IP 204117a59e09.38725 > 192.168.62.4.domain: 52244+ PTR?
26:52:11.223184 IP6 fe83::8c11:30ff:fe5e:f0e4 > ff02::16: HBH ICMP6,
...
```

You could remove `tcpdump` from your images, but there is very little preventing someone from reinstalling it. The most effective way to solve this problem is to determine what capability `tcpdump` needs to operate and remove that from the container. In this case, you can do so by adding `--cap-drop=NET_RAW` to your `docker run` command.

```
$ docker run -ti --rm --cap-drop=NET_RAW

spkane/train-os:latest tcpdump -i eth0
tcpdump: eth0: You don't have permission to capture on that device
(socket: Operation not permitted)
```

By using both the `--cap-add` and `--cap-drop` arguments to `docker run`, you can finely control your container's Linux kernel capabilities.

Secure Computing Mode

When Linux kernel version 2.6.12 was released in 2005, it included a new security feature called Secure Computing Mode, or `seccomp` for short. This feature enables a process to make a one-way transition into a special state, where it will only be allowed to make the system calls `exit()`, `sigreturn()`, and `read()` or `write()` to already-open file descriptors.

An extension to `seccomp`, called `seccomp-bpf`, utilizes the Linux version of Berkeley Packet Filter (`bpf`) rules to allow you to create a policy that will provide an explicit list of system calls that a process can utilize while running under Secure Computing Mode. The Docker support for Secure Computing Mode utilizes `seccomp-bpf` so that users can create profiles that give them very fine-grained control of what kernel system calls their containerized processes are allowed to make.

NOTE

By default, all containers use Secure Computing Mode and have the default profile attached to them. You can read more about Secure Computing Mode and what system calls the default profile blocks in the documentation. You can examine the default policy's JSON file to see what a policy looks like and understand exactly what it defines.

To see how you could use this, let's use the program `strace` to trace the system calls that a process is making.

If you try to run `strace` inside a container for debugging, you'll quickly realize that it will not work when using the default `seccomp` profile.

```
$ docker run -ti --rm spkane/train-os:latest whoami
```

```
root

$ docker run -ti --rm spkane/train-os:latest strace whoami
strace: ptrace(PTRACE_TRACEME, ...): Operation not permitted
+++ exited with 1 +++
```

You could potentially fix this by giving your container the process-tracing-related capabilities, like this:

```
$ docker run -ti --rm --cap-add=SYS_PTRACE spkane/train-os:latest \
    strace whoami

execve("/usr/bin/whoami", ["whoami"], [/* 4 vars */]) = 0
brk(NULL)                               = 0x136f000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, ...
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file ...
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
...
write(1, "root\n", 5root
)                   = 5
close(1)                                = 0
munmap(0x7f854286b000, 4096)            = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

But you can also solve this problem by using a `seccomp` profile. Unlike `seccomp`, `--cap-add` will enable a whole set of system calls, and you may not need them all. With a `seccomp` profile, however, you can be very specific about exactly what you want enabled or disabled.

If we take a look at the default `seccomp` profile, we'll see something like this:

```
{
        "defaultAction": "SCMP_ACT_ERRNO",
        "archMap": [
                {
                        "architecture": "SCMP_ARCH_X86_64",
                        "subArchitectures": [
                                "SCMP_ARCH_X86",
                                "SCMP_ARCH_X32"
                        ]
                },
...
        ],
        "syscalls": [
                {
                        "names": [
                                "accept",
                                "accept4",
...
                                "write",
                                "writev"
                        ],
                        "action": "SCMP_ACT_ALLOW",
                        "args": [],
                        "comment": "",
                        "includes": {},
                        "excludes": {}
                },
                {
                        "names": [
                                "personality"
                        ],
                        "action": "SCMP_ACT_ALLOW",
                        "args": [
                                {
                                        "index": 0,
                                        "value": 0,
                                        "valueTwo": 0,
                                        "op": "SCMP_CMP_EQ"
                                }
                        ],
                        "comment": "",
                        "includes": {},
                        "excludes": {}
                },
...
                {
                        "names": [
                                "vhangup"
                        ],
                        "action": "SCMP_ACT_ALLOW",
                        "args": [],
                        "comment": "",
                        "includes": {
                                "caps": [
                                        "CAP_SYS_TTY_CONFIG"
                                ]
                        },
                        "excludes": {}
                }
        ]
}
```

This JSON file provides a list of supported architectures, a default rule, and a specific setting for each system call that doesn’t fall under the default rule. In this case, the default action is `SCMP_ACT_ERRNO`, and will generate an error if an unspecified call is attempted.

If you look closely at the default profile, you’ll notice that `CAP_SYS_PTRACE` includes four system calls: kcmp, process_vm_readv, process_vm_writev, and ptrace.

```
{
                        "names": [
                                "kcmp",
                                "process_vm_readv",
                                "process_vm_writev",
                                "ptrace"
                        ],
                        "action": "SCMP_ACT_ALLOW",
                        "args": [],
                        "comment": "",
                        "includes": {
                                "caps": [
                                        "CAP_SYS_PTRACE"
                                ]
                        },
```

In the current use case, the container actually needs only one of those syscalls to function properly. This means that you are giving your container more capabilities than it requires when you use --cap-add=SYS_PTRACE. To ensure that you are adding only the one additional system call that you need, you can create your own Secure Computing Mode policy, based on the default policy that Docker provides.

First, pull down the default policy and make a copy of it.

```
$ wget https://raw.githubusercontent.com/moby/moby/master/\
profiles/seccomp/default.json
$ cp default.json strace.json
```

NOTE

The URL above has been continued on the following line so that it fits in the margins. You may find that you need to re-assemble the URL and remove the back slashes for the command to work properly in your environment.

Then edit the file and insert the line containing the word ptrace into the syscalls.names section of JSON:

```
...
  "syscalls": [
    {
      "names": [
...
        "pwritev2",
        "ptrace",
        "read",
...
```

Once you have saved your changes, you can verify them using a command like `diff` (Unix) or `compare-object` (PowerShell).

```
$ diff -u default.json strace.json
```

```
--- default.json        2018-02-19 11:47:07.000000000 -0800
+++ strace.json 2018-02-19 12:12:21.000000000 -0800
@@ -229,6 +229,7 @@
                "pwrite64",
                "pwritev",
                "pwritev2",
+               "ptrace",
                "read",
                "readahead",
                "readlink",
```

You are now ready to test your new finely tuned `seccomp` profile.

```
$ docker run -ti --rm --security-opt=seccomp:strace.json \
    spkane/train-os:latest strace whoami

execve("/usr/bin/whoami", ["whoami"], [/* 4 vars */]) = 0
brk(NULL)                               = 0x944000
...
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

If everything went according to plan, your `strace` of the `whoami` program should have run perfectly, and now you can rest comfortably knowing that your container has been given only the exact privileges it needs to get the job done, and nothing more.

WARNING

You could completely disable the default Secure Computing Mode profile by setting `--securityopt=seccomp:unconfined`, however running a container unconfined is a very bad idea in general, and is probably only useful when you are trying to figure out exactly what system calls you need to define in your profile.

The strength of Secure Computing Mode is that it allows users to be much more selective about what a container can and can’t do with the underlying Linux kernel. Custom profiles are not required for most containers, but they are an incredibly handy tool when you need to carefully craft a powerful container and ensure that you maintain the overall security of the system.

SElinux and AppArmor

Earlier we talked about how containers are a combination of two or three things: cgroups, namespaces, and SELinux or AppArmor. We’re going to discuss the latter two systems now. They allow you to apply security controls that extend beyond those normally supported by Unix systems. SELinux (Security-Enhanced Linux) originated in the US National Security Agency, was strongly adopted by Red Hat, and supports very fine-grained control. AppArmor is an effort to achieve many of the same goals without the level of complication involved in SELinux. AppArmor actually predates the open source release of SELinux by two years, having first appeared in 1998 in the Immunix Linux distribution. Novell, SuSE, and Canonical have been some of AppArmor’s recent champions.

Docker ships by default with reasonable profiles enabled on platforms that support them. This means that on Ubuntu systems, AppArmor is enabled and configured, and on Red Hat systems, SELinux is set up. You can further configure these profiles to enable or prevent all sorts of features, and if you’re running Docker in production, you should do a risk analysis to determine if there are additional considerations that you should be aware of. We’ll give a quick outline of the benefits you are getting from these systems.

Both systems provide *Mandatory Access Control*, a class of security system where a systemwide security policy grants users (or “initiators”) access to a resource (or “target”). This allows you to prevent anyone, including `root`, from accessing a part of the system that they should not have access to. You can apply the policy to a whole container so that all processes are constrained. Many chapters would be required to provide a clear and detailed overview of how to configure these systems. The default profiles are performing tasks like blocking access to parts of the */proc* and */sys* filesystems that would be dangerous to expose in the container, even though they show up in the container's namespace. The default profiles also provide more narrowly scoped mount access to prevent containers from getting hold of mount points they should not see.

If you are considering using Docker containers in production, make certain that the systems you are running have AppArmor or SELinux enabled and running. For the most part, both systems are reasonably equivalent. But in the Docker context, one notable limitation of SELinux is that it only works fully on systems that support filesystem metadata, which means that it won't work with all Docker storage drivers. AppArmor, on the other hand, does not use filesystem metadata and therefore works on all of the Docker backends. Which one you use is somewhat distribution-centric, so you may be forced to choose a filesystem backend that also supports the security system that you use.

The Docker Daemon

From a security standpoint, the Docker daemon and its components are the only completely new risk you are introducing to your infrastructure. Your containerized applications are not any less secure and are, at least, a little more secure than they would be if deployed outside of containers. But without the containers, you would not be running `dockerd`, the Docker daemon. You can run Docker such that it doesn't expose any ports on the network. In this model, you'd need to do something like set up an SSH tunnel to each Docker server or run a specialized agent if you wanted to control the containers. That's often not ideal, so many people end up exposing at least one Docker port on the local network.

The default configuration for Docker, on most distributions, leaves Docker isolated from the network with only a local Unix socket exposed. Since you cannot remotely administer Docker when it is set up this way, it is not uncommon to see people simply add the nonencrypted port *2375* to the configuration. This may be great for getting started with Docker, but it is not what you should do in any environment where you care about the security of your servers. In fact, you should not open Docker up to the outside world at all unless you have a very good reason to do it. If you do, you should also commit to properly securing it. Most scheduler systems run their own software on each node and expect to talk to Docker over the Unix domain socket.

If you do need to expose the daemon to the network, you can do a few things to tighten Docker down in a way that makes sense in most production environments. But no matter what you do, you are relying on the Docker daemon itself to be resilient against threats like buffer overflows and race conditions, two of the more common classes of security vulnerabilities. This is true of any network service. The risk is perhaps a little higher from Docker because it has the ability to control most of your applications, and because of the privileges the daemon requires, it has to be run as `root`.

The basics of locking Docker down are common with many other network daemons: encrypt your traffic and authenticate users. The first is reasonably easy to set up on Docker; the second is not as easy. If you have SSL certificates you can use for protecting HTTP traffic to your hosts, such as a wildcard certificate for your domain, you can turn on TLS support to encrypt all of the traffic to your Docker servers, using port 2376. This is a good first step. The Docker documentation will walk you through doing this.

Authenticating users is more complicated. Docker does not provide any kind of fine-grained authorization: you either have access or you don't. But the authentication control it does provide—signed certificates—is reasonably strong. Unfortunately this also means that you don't get a cheap step from no authentication to some authentication without also having to set up your own certificate authority in most cases. If your organization already has one, then you are in luck. Certificate management needs to be implemented carefully in any organization, both to keep certificates secure and to distribute them efficiently. So, given that, here are the basic steps:

1. Set up a method of generating and signing certificates.

2. Generate certificates for the server and clients.

3. Configure Docker to require certificates with `--tlsverify`.

Detailed instructions on getting a server and client set up, as well as a simple certificate authority, are included in the Docker documentation.

> **WARNING**
>
> Because it's a daemon that runs with privilege, and because it has direct control of your applications, it is a bad idea to expose Docker directly on the internet. If you need to talk to your Docker hosts from outside your network, consider something like a VPN or an SSH tunnel to a secure jump host.

Advanced Configuration

Docker has a very clean external interface and on the surface it looks pretty monolithic. But there's actually a lot going on under the covers that is configurable, and the logging backends we described in "Logging" are a good example. You can also do things like change out the storage backend for container images for the whole daemon, use a completely different runtime, or configure individual containers to run on a totally different network configuration. Those are powerful switches and you'll want to know what they do before turning them on. First we'll talk about the network configuration, then we'll cover the storage backends, and finally we'll try out a completely different container runtime to replace the default `runc` supplied with Docker.

Networking

Early on we described the layers of networking between a Docker container and the real live network. Let's take a closer look at how that works. Docker supports a rich set of network configurations, but let's start out with the default setup. Figure 11-1 shows a drawing of a typical Docker server, where there are three containers running on their private network, shown on the right. One of them has a public port (TCP port 10520) that is exposed on the Docker server. We'll track how an inbound request gets to the Docker container and also how a Docker container can make an outbound connection to the external network.

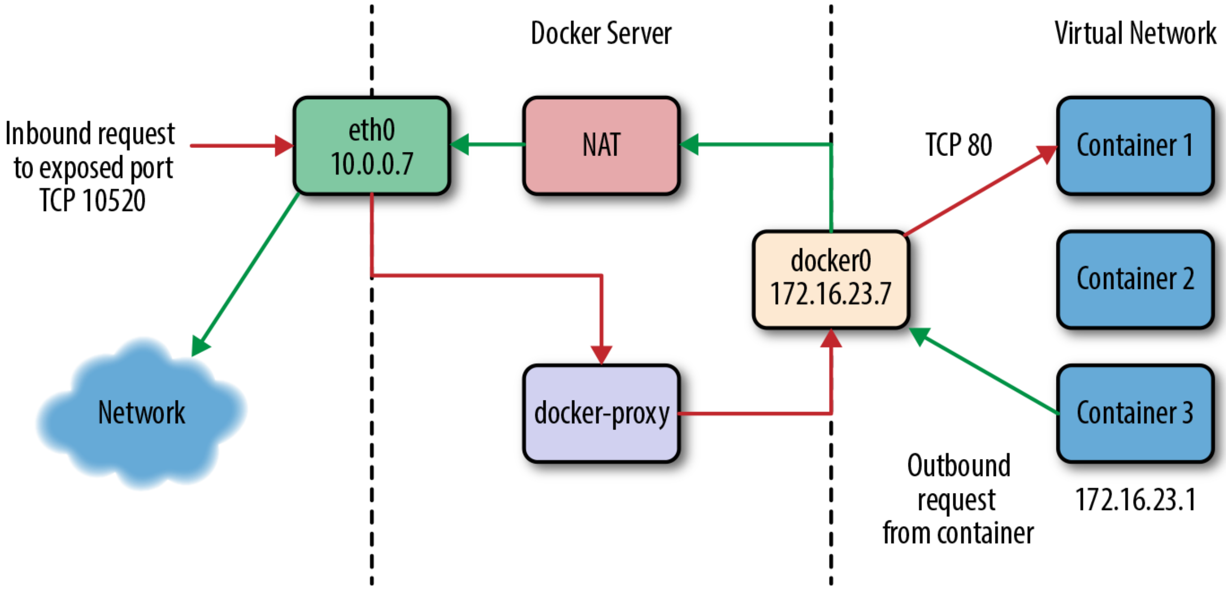


Figure 11-1. The network on a typical Docker server

If we have a client somewhere on the network that wants to talk to the `nginx` server running on TCP port 80 inside Container 1, the request will come into the `eth0` interface on the Docker server. Because Docker knows this is a public port, it has spun up an instance of `docker-proxy` to listen on port 10520. So our request is passed to the `docker-proxy` process, which then forwards the request to the correct container address and port on the private network. Return traffic from the request flows through the same route.

Outbound traffic from the container follows a different route in which the `docker-proxy` is not involved at all. In this case, Container 3 wants to contact a server on the public internet. It has an address on the private network of 172.16.23.1 and its default route is the `docker0` interface 172.16.23.7. So it sends the traffic there. The Docker server now sees that this traffic is outbound and it has traffic forwarding enabled. And since the virtual network is private, it wants to send the traffic from its own public address instead. So the request is passed through the kernel's network address translation (NAT) layer and put onto the external network via the `eth0` interface on the server. Return traffic passes through the same route. Note that the NAT is one-way, so containers on the virtual network will see real network addresses in response packets.

You've probably noticed that it's not a simple configuration. It's a fair amount of complexity, but it makes Docker seem pretty transparent. It also contributes to the security posture of the Docker stack because the containers are namespaced into their own network namespace, are on their own private network, and don't have access to things like the main system's DBus or IPTables.

Let's examine what's happening at a more detailed level. The interfaces that show up in `ifconfig` or `ip addr show` in the Docker container are actually virtual Ethernet interfaces on the Docker server's kernel. They are then mapped into the container's network namespace and given the names that you see inside the container. Let's take a look at what we see when running `ip addr show` on a Docker server. We'll shorten the output a little for clarity and spaces, as shown here:

```
$ ip addr show

1: lo: <LOOPBACK,UP,LOWER_UP>
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP>
    link/ether 00:0c:29:b2:2a:21 brd ff:ff:ff:ff:ff:ff
    inet 172.16.168.178/24 brd 172.16.168.255 scope global eth0
    inet6 fe80::20c:29ff:feb2:2a21/64 scope link
       valid_lft forever preferred_lft forever

4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP>
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever

112: vethf3e8733: <BROADCAST,UP,LOWER_UP>
    link/ether b6:3e:7a:ba:5e:1c brd ff:ff:ff:ff:ff:ff
    inet6 fe80::b43e:7aff:feba:5e1c/64 scope link
       valid_lft forever preferred_lft forever
```

What this tells us is that we have the normal loopback interface, our real Ethernet interface `eth0`, and then the Docker bridge interface, `docker0`, that we described earlier. This is where all the traffic from the Docker containers is picked up to be routed outside the virtual network. The surprising thing in this output is the `vethf3e8733` interface. When Docker creates a container, it creates two virtual interfaces, one of which sits on the server side and is attached to the `docker0` bridge, and one that is exposed into the container's namespace. What we're seeing here is the server-side interface. Did you notice how it doesn't show up as having an IP address assigned to it? That's because this interface is just joined to the bridge. This interface will have a totally different name in the container's namespace as well.

As with so many pieces of Docker, you can replace the proxy with a different implementation. To do so, you would use the `--userland-proxy-path=<path>` setting, but there are probably not that many good reasons to do this unless you have a very specialized network. However, the `--userland-proxy=false` flag to `dockerd` will completely disable the `userland-proxy` and instead rely on hairpin NAT functionality to route traffic between local containers. This performs a lot better than the `userland-proxy` and will likely become the preferred approach. Docker documentation currently recommends it as the best approach, but it is not yet the default. If you need higher-throughput services, this might be right for you.

HOST NETWORKING

As we've noted, there is a lot of complexity involved in the default implementation. You can, however, run a container without the whole networking configuration that Docker puts in place for you. And the `docker-proxy` can also limit throughput for very high-volume data services, by requiring all the network traffic to pass through the docker-proxy process before being received by the container. So what does it look like if we turn off the Docker network layer? Since the beginning, Docker has let you do this on a per-container basis with the `--net=host` command-line switch. There are times, like when you want to run high-throughput applications, where you might want to do this. But you lose some of Docker's flexibility when you do it. Even if you never need or want to do this, it's useful to expose how the mechanism works.

> **WARNING**
>
> Like others we discuss in this chapter, this is not a setting you should take lightly. It has operational and security implications that might be outside your tolerance level. It can be the right thing to do, but you should consider the repercussions.

Let’s start a container with `--net=host` and see what happens.

```
$ docker run -i -t --net=host ubuntu /bin/bash

root@782d18f5c38a:/# apt update
...
root@782d18f5c38a:/# apt install -y iproute2 net-tools
...
root@782d18f5c38a:/# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP>
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP>
    link/ether 00:0c:29:b2:2a:21 brd ff:ff:ff:ff:ff:ff
    inet 172.16.168.178/24 brd 172.16.168.255 scope global eth0
    inet6 fe80::20c:29ff:feb2:2a21/64 scope link
       valid_lft forever preferred_lft forever

3: lxcbr0: <BROADCAST,MULTICAST,UP,LOWER_UP>
    link/ether fe:59:0b:11:c2:76 brd ff:ff:ff:ff:ff:ff
    inet 10.0.3.1/24 brd 10.0.3.255 scope global lxcbr0
    inet6 fe80::fc59:bff:fe11:c276/64 scope link
       valid_lft forever preferred_lft forever

4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP>
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever

112: vethf3e8733: <BROADCAST,UP,LOWER_UP>
    link/ether b6:3e:7a:ba:5e:1c brd ff:ff:ff:ff:ff:ff
    inet6 fe80::b43e:7aff:feba:5e1c/64 scope link
       valid_lft forever preferred_lft forever
```

That should look pretty familiar. That’s because when we run a container with the host networking option, we’re just in the host’s network namespace. Note that we’re *also* in the server’s UTS namespace. Our server’s hostname is `docker2`, so let’s see what the container’s hostname is:

```
root@852d18f5c38d:/# hostname
docker2
```

If we do a `mount` to see what’s mounted, though, we see that Docker is still maintaining our */etc/resolv.conf*, */etc/hosts*, and */etc/hostname* directories. Interestingly, the */etc/hostname* directory simply contains the server’s hostname. Just to prove that we can see all the normal networking on the Docker server, let’s look at `netstat -an` to check if we can see the `docker` daemon running:

```
root@852d18f5c38d:/# netstat -an | grep 2375

tcp6       0      0 :::2375                 :::*                    LISTEN
```

NOTE

This `netstat` example will work as shown only if the Docker daemon is actually listening on the unencrypted TCP port (2375). You can try to look for another server port that you know your server is using.

So we are indeed in the server’s network namespace. What all of this means is that if we were to launch a high-throughput network service, we could expect network performance from it that is essentially native. But it also means we could try to bind to ports that would collide with those on the server, so if you do this you should be careful about how you allocate port assignments.

CONFIGURING NETWORKS

There is more to networking than just the default network or host networking, however. The `docker network` command lets you create multiple networks backed by different drivers. It also allows you to view and manipulate the Docker network layers and how they are attached to containers that are running on the system.

Listing the networks available from Docker’s perspective is easily accomplished with the following command:

```
$ docker network ls

NETWORK ID          NAME                DRIVER
c37e1476e9c1        none                null
d7f88e502765        host                host
15dd2b3b16b1        bridge              bridge
```

You can then find out more details about any individual network by using the `docker network inspect` command along with the network ID:

```
$ docker network inspect 15dd2b3b16b1
```

```
[
    {
        "Name": "bridge",
        "Id": ...,
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.18.0.0/16"
                }
            ]
        },
        "Containers": {
            "...": {
                "EndpointID": "...",
                "MacAddress": "04:42:ab:26:03:52",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        }
    }
]
```

Docker networks can be created and removed, as well as attached and detached from individual containers, with the `network` subcommand.

So far, we've set up a bridged network, no Docker network, and a bridged network with hairpin NAT. There are a few other drivers that you can use to create different topologies using Docker as well, with the `overlay` and `macvlan` drivers being the most common. Let's take a brief look at what these can do for you.

`overlay`

> This driver is used in Swarm mode to generate a network overlay between the Docker hosts, creating a private network between all the containers that runs on top of the real network.

This is useful for Swarm, but not scoped for general use with non-Swarm containers.

`macvlan`

This a driver creates a real MAC address for each of your containers and then exposes them on the network via the interface of your choice. This requires that you switch gears to support more than one MAC address per physical port on the switch. The end result is that all the containers appear directly on the underlying network. When you're moving from a legacy system to a container-native one, this can be a really useful step. There are drawbacks here, such as making it harder when debugging to identify which host traffic is really coming from, overflowing the MAC tables in your network switches, excessive ARPing by container hosts, and other underlying network issues. For that reason the `macvlan` driver is not recommended unless you have a good understanding of your underlying network and can manage it effectively.

There are a few sets of configurations that are possible here, but the basic setup is easy to configure:

```
$ docker network create -d macvlan \
    --subnet=172.16.16.0/24 \
    --gateway=172.16.16.1  \
    -o parent=eth0 ourvlan
```

TIP

You can prevent Docker from allocating specific addresses by specifying them as named auxiliary addresses `--aux-address="my-router=172.16.16.129"`.

There is a lot more you can configure with the Docker network layer. However, the defaults, host networking, and userland-proxyless mode are the ones that you're most likely to use or encounter in the wild. Some of the other options you can configure include the container's DNS nameservers, resolver options, and default gateways, among other things. The networking section of the Docker documentation gives an overview of how to do some of this configuration.

NOTE

For advanced network configuration of Docker, check out Weave—a well-supported overlay network tool for spanning containers across multiple Docker hosts, similar to the `overlay` driver but much more configurable, and without the Swarm requirement. Another offering, supported directly by Docker Enterprise Edition, is Project Calico. If you are running Kubernetes, which has its own networking configuration, you might also take a look at CoreOS's flannel, which is an `etcd`-backed network fabric for containers.

Storage

Backing all of the images and containers on your Docker server is a storage backend that handles reading and writing all of that data. Docker has some strenuous requirements on its storage backend: it has to support layering, the mechanism by which Docker tracks changes and reduces both how much disk a container occupies and how much is shipped over the wire to deploy new images. Using a copy-on-write strategy, Docker can start up a new container from an existing image without having to copy the whole image. The storage backend supports that. The storage backend is what makes it possible to export images as groups of changes in layers, and also lets you save the state of a running container. In most cases, you need the kernel's help in doing this efficiently. That's because the filesystem view in your container is generally a union of all of the layers below it, which are not actually copied into your container. Instead, they are made visible to your container, and only when you make changes does anything get written to your container's own filesystem. One place this layering mechanism is exposed to you is when you upload or download a new image from a registry like Docker Hub. The Docker daemon will push or pull each layer at a time, and if some of the layers are the same as others it has already stored, it will use the cached layer instead. In the case of a push to a registry, it will sometimes even tell you which image they are mounted from.

Docker relies on an array of possible kernel drivers to handle the layering. The Docker codebase contains code that can handle interacting with all of these backends, and you can configure the decision about which to use on daemon restart. So let's look at what is available and some of the pluses and minuses of each.

Various backends have different limitations that may or may not make them your best option. In some cases, your choices of which backend to use are limited by what your distribution of Linux actually supports. Using the drivers that are built into the kernel shipped with your distribution will always be the easiest approach. It's generally best to stay near the tested path here as well.

We’ve seen all manner of oddities from various backends since Docker’s release. And, as usual, the common case is always the best-supported one. Different backends also report different statistics up through the Docker Remote API (*/info* endpoint). This is potentially useful for monitoring your Docker systems. However, not all backends are created equal, so let’s see how they differ.

Overlay

Overlay (formerly OverlayFS) is a union filesystem where multiple layers are mounted together so that they appear as a single filesystem. This is the most recommended driver these days and works on most major distributions. If you are running on a Linux kernel older than 4.0 (or 3.10.0-693 for CentOS/RHEL), then you won’t be able to take advantage of this backend. The reliability and performance are good enough that it might be worth updating your OS for Docker hosts in order to support it, even if your company standard is an older distribution. Overlay is part of the mainline Linux kernel and has become increasingly stable over time. Being in the mainline means that long-term support is virtually guaranteed, which is another nice advantage. Docker supports two versions of the Overlay backend, `overlay` and `overlay2`. As you might expect, you are strongly advised to use `overlay2` as it is faster, more efficient with inode usage, and more robust.

> **NOTE**
>
> The Docker community is frequently improving support for a variety of filesystem backends. For more details about the supported filesystems, take a look at the official documentation.

AuFS

Although at the time of this writing it is no longer recommended, `aufs` is the original backend. AuFS is a union filesystem driver with reasonable support on various popular Linux distributions. It was never accepted into the mainline kernel, however, and this has limited its availability on various distributions. It is not supported on recent versions of Red Hat, Fedora, or CentOS, for example. It is not shipped in the standard Ubuntu distribution, but is in the Ubuntu `linux-image-extra` package.

Its status as a second-class citizen in the kernel has led to the development of many of the other backends now available. If you are running an older distribution that supports AuFS, you might consider it for stability but you should really upgrade to a kernel version that natively supports Overlay or Btrfs (discussed next).

Btrfs

Btrfs is fundamentally a copy-on-write filesystem, which means it's a pretty good fit for the Docker image model. Like `aufs` and unlike `devicemapper`, Docker is using the backend in the way it was intended. That means it's both pretty stable in production and also a good performer. It scales reasonably to thousands of containers on the same system. A drawback for Red Hat–based systems is that Btrfs does not support SELinux. If you can use the `btrfs` backend, we currently recommend it as one of the most stable backends for production, after the `overlay2` driver. One popular way to run `btrfs` backends for Docker containers without having to give over a whole volume to this somewhat prerelease filesystem is to make a Btrfs filesystem in a file and loopback-mount it with something like `mount -o loop file.btrs /mnt`. Using this method, you could build a 50 GB Docker container storage filesystem even on cloud-based systems without having to give over all your precious local storage to Btrfs.

Device Mapper

Originally written by Red Hat to support their distributions, which lacked AuFS in Docker's early days, Device Mapper has become the default backend on all Red Hat–based distributions of Linux. If you run Docker Enterprise Edition or the commercially supported release on a Red Hat OS, this is your only option. Device Mapper itself has been built into the Linux kernel for ages and is very stable. The way the Docker daemon uses it is a bit unconventional, though, and in the past this backend was not that stable. This checkered past means that we prefer to pick a different backend when possible. If your distribution supports only the `devicemapper` driver, then you will likely be fine. But it's worth considering using `overlay2` or `btrfs`. By default, `devicemapper` utilizes the `loop-lvm` mode, which has zero configuration, but it is very slow and useful only for development. If you decide to use the `devicemapper` driver, it is very important that you make sure it is configured to use `direct-lvm` mode for all nondevelopment environments.

NOTE

You can find out more about using the various `devicemapper` modes with Docker in the official documentation. A 2014 blog article also provides some interesting history about the various Docker storage backends.

VFS

The Virtual File System (vfs) driver is the simplest, and slowest, to start up of the supported drivers. It doesn't really support copy-on-write. Instead, it makes a new directory and copies over all of the existing data. It was originally intended for use in tests and for mounting host volumes. The vfs driver is very slow to create new containers, but runtime performance is native, which is a real benefit. Its mechanism is very simple, which means there is less to go wrong. Docker, Inc., does not recommend it for production use, so proceed with caution if you think it's the right solution for your production environment.

ZFS

ZFS, originally created by Sun Microsystems, is the most advanced open source filesystem available on Linux. Due to licensing restrictions, it does not ship in mainline Linux. However, the ZFS On Linux project has made it pretty easy to install. Docker can then run on top of the ZFS filesystem and use its advanced copy-on-write facilities to implement layering. Given that ZFS is not in the mainline kernel and not available off the shelf in the major commercial distributions, however, going this route requires some extended effort. If you are already running ZFS in production, however, then this is probably your very best option.

WARNING

Storage backends can have a big impact on the performance of your containers. And if you swap the backend on your Docker server, all of your existing images will disappear. They are not gone, but they will not be visible until you switch the driver back. Caution is advised.

You can use `docker info` to see which storage backend your system is running:

```
$ docker info
...
Storage Driver: overlay2
 Backing Filesystem: extfs
 Supports d_type: true
 Native Overlay Diff: true
...
```

As you can see, Docker will also tell you what the underlying or "backing" filesystem is in cases where there is one. Since we’re running `overlay2` here, we can see it’s backed by an `ext` filesystem. In some cases, like with `devicemapper` on raw partitions or with `btrfs`, there won’t be a different underlying filesystem.

Storage backends can be swapped via command-line arguments to `docker` on startup. If we wanted to switch our Ubuntu system from `aufs` to `devicemapper`, we would do so like this:

```
$ dockerd --storage-driver=devicemapper
```

That will work on pretty much any Linux system that can support Docker because `devicemapper` is almost always present. The same holds true for `overlay2` on modern Linux kernels. However, you will need to have the actual underlying dependencies in place for the other drivers. For example, without `aufs` in the kernel—usually via a kernel module—Docker will not start up with `aufs` set as the storage driver, and the same holds true for Btrfs or ZFS.

Getting the appropriate storage driver for your systems and deployment needs is one of the more important technical points to get right when you’re taking Docker to production. Be conservative; make sure the path you choose is well supported in your kernel and distribution. Historically this was a pain point, but most of the drivers have reached reasonable maturity. Remain cautious for any newly appearing backends, however, as this space continues to change. Getting new backend drivers to work reliably for production systems takes quite some time, in our experience.

The Structure of Docker

What we think of as Docker is made of five major server-side components that present a common front via the API. These parts are `dockerd`, `containerd`, `runc`, `docker-containerd-shim`, and the `docker-proxy` we described in “Networking”. We’ve spent a lot of time interacting with `dockerd` and the API it presents. It is, in fact, responsible for orchestrating the whole set of components that make up Docker. But when it starts a container, Docker relies on `containerd` to handle instantiating the container. All of this used to be handled in the `dockerd` process itself, but there were a number of shortcomings to that design:

- `dockerd` had a huge number of jobs.

- A monolithic runtime prevented any of the components from being swapped out easily.

- `dockerd` had to supervise the lifecycle of the containers themselves and it couldn’t be restarted or upgraded without losing all the running containers.

Another one of the major motivations for `containerd` was that, as we've just shown, containers are not just a single abstraction. On the Linux platform, they are a process involving namespaces, cgroups, and security rules in AppArmor or SELinux. But Docker runs on Windows and will likely work on other platforms in the future. The idea of `containerd` is to present a standard layer to the outside world where, regardless of implementation, developers can think about the higher-level concepts of containers, tasks, and snapshots rather than worrying about specific Linux system calls. This simplifies the Docker daemon quite a lot, and enables platforms like Kubernetes to integrate directly into `containerd` rather than using the Docker API. Kubernetes currently supports both methods, though it normally talks to Docker directly. In theory, you could replace `containerd` with an API-compatible layer, something that doesn't talk to containers at all, but at this time there aren't any alternatives that we know of.

Let's take a look at the components (shown in Figure 11-2) and see what each of them does:

`dockerd`

One per server. Serves the API, builds container images, and does high-level network management including volumes, logging, statistics reporting, and more.

`containerd`

One per server. Manages lifecycle, execution, copy-on-write filesystem, and low-level networking drivers.

`docker-containerd-shim`

One per container. Handles file descriptors passed to the container (e.g., `stdin/out`) and reports exit status.

`runc`

Constructs the container and executes it, gathers statistics, and reports events on lifecycle.

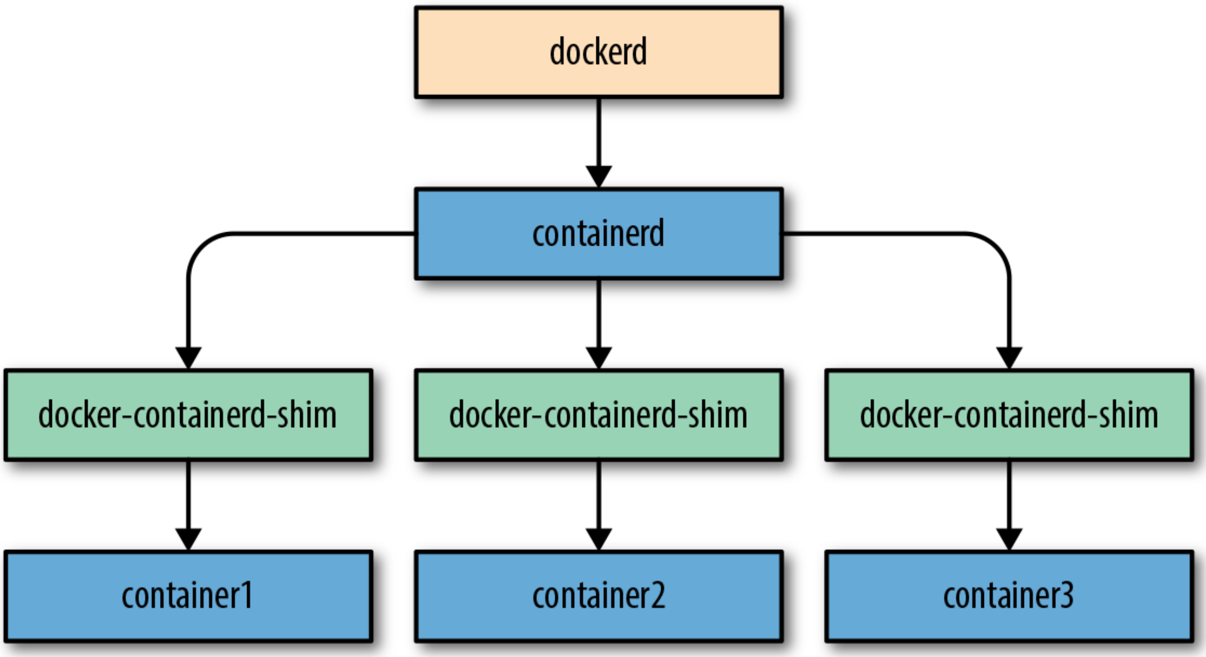


Figure 11-2. Structure of Docker

`dockerd` and `containerd` speak to each other over a socket, usually a Unix socket, using a gRPC API. `dockerd` is the client in this case, and `containerd` is the server! `runc` is a CLI tool that reads configuration from JSON on disk and is executed by `containerd`.

When we start a new container, `dockerd` will handle making sure that the image is present or will pull it from the repository specified in the image name. (In the future this responsibility may shift to `containerd`, which already supports image pulls.) The Docker daemon also does most of the rest of the setup around the container, like setting up any logging drivers and volumes or volume drivers. It then talks to `containerd` and asks it to run the container. `containerd` will take the image and apply the container configuration passed in from `dockerd` to generate an OCI (Open Container Initiative) bundle that `runc` can execute.[2] It will then execute `docker-containerd-shim` to start the container. This will in turn execute `runc` to actually construct and start the container. However, `runc` will not stay running, and the `docker-containerd-shim` will be the actual parent process of the new container process.

If we look at the output of `ps axlf` we can see the parent/child relationship between the processes:

```
...36   1  ... /usr/bin/dockerd -D --default-runtime=runc -H fd:// -H 0.0.0
...41  36  ... \_ docker-containerd -l unix:...containerd/docker-containerd
...04  41  ...   \_ docker-containerd-shim 487c7e3065e... /var/run/do
...21  04  ...     \_ nginx: master process nginx -g daemon off;
...45  21  ...       \_ nginx: worker process
```

So what happened to `runc`? Its job is to construct the container and start it running, and then it leaves and its children are inherited by its parent, the `docker-containerd-shim`. This leaves the minimal amount of code in memory necessary to manage the file descriptors and exit status for `containerd`.

To help you understand what's going on here, let's look at what happens under the hood when we start a container. We'll just use `nginx` for this since it's very lightweight and the container stays running when backgrounded.

```
$ docker run -d -P nginx

487c7e3065e5e4959e3ae860cd6689e62d20aa36b9c86304331d2381c1cc3634
```

Once that's running, let's use the `runc` runtime CLI tool to take a look at its view of the system. We could see a similar view from `containerd-ctr`, the CLI client for `containerd`, but `runc` is nicer to work with and it's at the lowest level.

> **NOTE**
>
> Docker ships with its own copy of `runc`, so we'll just use that. While you can install `runc` separately, using the one that ships with your Docker installation will guarantee compatibility.

```
$ sudo docker-runc list

ID                PID      STATUS    BUNDLE
487c7e3065e...   123309   running   /run/docker/libcontainerd/487c7e3065e5e..
```

Note that we normally need root privileges to run this command. Unlike with the Docker CLI, we can't rely on the Docker daemon's permissions to let us access lower-level functionality. With `runc` we need direct access to these privileges. What we can see in the output from `docker-runc` is our container! This is the actual OCI runtime bundle that represents our container, with which it shares an ID. Notice that it also gives us the PID of the container; that's the PID on the host of the application running inside the container:

```
$ ps -edaf | grep 123309
```

```
root      123309 123292  0 12:41 ?  00:00:00 nginx: master process nginx
systemd+ 123329 123309  0 12:41 ?  00:00:00 nginx: worker process
```

If we look in the bundle, we'll see a set of named pipes and the *config.json* file for our container:

```
$ sudo ls -la /run/docker/libcontainerd/487c7e3065e5e...

total 20
drwx------ 2 root root   120 Feb 27 12:42 .
drwx------ 6 root root   180 Feb 27 12:42 ..
-rw-r--r-- 1 root root 20131 Feb 27 12:42 config.json
prwx------ 1 root root     0 Feb 27 12:42 init-stderr
prwx------ 1 root root     0 Feb 27 12:42 init-stdin
prwx------ 1 root root     0 Feb 27 12:42 init-stdout
```

The *config.json* file is a very verbose equivalent of what Docker shows in `docker inspect`. We are not going to reproduce it here due to size, but we encourage you to dig around and see what's in the config. You may, for example, note all the entries for the "Secure Computing Mode" that are present in it.

If you want to explore `runc` some more, you can experiment with the CLI tool. Most of this is available already in Docker, usually on a higher and more useful level than the one available in `runc`. But it's useful to explore in order to understand how containers and the Docker stack are put together. It's also interesting to watch the events that `runc` reports about a running container. We can hook into those with the `runc events` command. During normal operation of a running container, there is not a lot of activity in the events stream. But `runc` regularly reports runtime statistics, which we can see in JSON format:

```
$ sudo docker-runc events 487c7e3065e5e4959...

{"type":"stats","id":"487c7e3065e5e4959...","data":{"cpu":{"usage":{".
.."}}}}
```

To conserve space, we have removed much of the output from the previous command, but this might look familiar to you now that we've spent some time looking at `docker stats`. Guess where Docker gets those statistics by default. That's right, `runc`.

Swapping Runtimes

As we mentioned in Chapter 2, there are a few other OCI-compliant runtimes that can be substituted in place of `runc`. There is Oracle's Railcar, which is a reimplementation of `runc` and aims to be an exact drop-in replacement. Its main goal is to provide an alternative to validate that the spec properly covers all aspects of the runtime's requirements. CoreOS has stated its intention of making `rkt` compliant with the OCI standard, although at the time of this writing, this is not currently the case. Next, we'll talk about two different runtimes: Intel's Clear Containers, which has now become a part of the Kata Containers project from the OpenStack Foundation, and gVisor, Google's user space containerizer.

CLEAR CONTAINERS/KATA CONTAINERS

The Clear Containers runtime is quite different from `runc` and Railcar. It actually uses hardware virtualization to provide much harder isolation between containers than you get with namespaces and cgroups. Rather than being a process on the main kernel, each container (or Kubernetes pod) runs inside a lightweight virtual machine and has its own kernel. From the outside it looks just like a normal Docker container, but the isolation is much stronger than cgroups and namespaces can provide. As a result of the move to Kata Containers, the project is in flux at this time and installation instructions are likely to change a lot in the near term. If this piques your interest and you want to try it out, the project provides installation guides for multiple Linux distributions.

Rather than walking through the installation, we'll just show you here what Docker looks like with this runtime installed and how `containerd` and Docker can manage more than one runtime. For the following, let's assume that the installation has been completed. We can see that Docker knows about the new runtime with the `docker info` command:

```
$ docker info

Containers: 6
 Running: 4
 Paused: 0
 Stopped: 2
Images: 119
...
Runtimes: cc-runtime runc
Default Runtime: runc
...
```

Notice how we have both `runc` and `cc-runtime` installed. We can choose at execution time which of these our container will use by leveraging the `--runtime` argument to `docker run`:

```
$ docker run --runtime cc-runtime -d -P nginx
```

```
88b0d2a2c3f36f67251d1c0b94c3d940ee2ed940b0c6b964f1db16a36140d88f
```

On this machine we have our `nginx` container still running from earlier, and we’ve now started one with the Clear Containers runtime as well. Despite being on totally different runtimes, the output of `docker ps` looks identical:

```
$ docker ps

CONTAINER ID  IMAGE  COMMAND                   CREATED         STATUS
88b0d2a2c3f3  nginx  "nginx -g 'daemon ..."  10 seconds ago  Up 8 seconds
487c7e3065e5  nginx  "nginx -g 'daemon ..."  2 hours ago     Up 55 seconds
```

The first container is our new container that we launched using `cc-runtime` and the second is our container that we launched using the usual `runc` runtime. We can take a closer look at the new container with the `cc-runtime` command itself:

```
$ sudo cc-runtime list

ID                   PID     STATUS   BUNDLE
88b0d2a2c3f36f6...  124379  running  /run/docker/libcontainerd/88b0d2a2c3f
```

This is exactly like the output from `runc`, but notice that we don’t see the other container because it’s on a different runtime. If we take a look at the process tree for our container process on the host, we see that we have a few more processes wrapping our container too:

```
$ ps axlf

...
...122836      1 ... /usr/bin/dockerd -D --add-runtime cc-runtime=/usr/bin
...122841 122836 ...  \_ docker-containerd -l unix:///var/run/docker/libco
...124228 122841 ...      \_ docker-containerd-shim 487c7e3065e5e4959e3ae8
...124245 124228 ...      |   \_ nginx: master process nginx -g daemon off
...124266 124245 ...      |       \_ nginx: worker process
...124333 122841 ...      \_ docker-containerd-shim 88b0d2a2c3f36f67251d1c
...124363 124333 ...          \_ /usr/bin/qemu-lite-system-x86_64 -name po
...124375 124333 ...          \_ /usr/libexec/clear-containers/cc-proxy -u
...124379 124333 ...          \_ /usr/libexec/clear-containers/cc-shim -c
...124380 124379 ...              \_ /usr/libexec/clear-containers/cc-shim
```

The `dockerd` here is running with the `--add-runtime` command pointing to our Clear

Containers runtime. We also see the original container, just like before on the `runc` runtime. The last container in the tree is the new one. Underneath the `docker-containerd-shim`, we have a copy of the QEMU virtual machine running, with a Pod under it that has the same ID as our container. We also have two pieces of the Clear Containers runtime, a `cc-proxy` and a `cc-shim`. Clear Containers can actually run more than one container inside the same virtual machine so that systems like Kubernetes that have a Pod abstraction can allow containers to share namespaces and cgroups. With a virtual machine, you can't do that because of its much harder isolation, so any resource sharing has to be in the same virtual machine.

The `cc-proxy` is responsible for managing any communication from the host across the barrier of the virtual machine in order to enable Docker to control the lifecycle of containers inside the virtual machine. Since multiple containers may be running in the same virtual machine (i.e., they are in the same Pod), there is only one proxy per virtual machine. On the other hand, there is one instance of `cc-shim` per container. It sits between the `docker-containerd-shim` and the `cc-proxy` and passes messages back and forth for the container it is responsible for managing.

As you might expect, Clear Containers has its own configuration as well. We can take a peek at it with the runtime CLI itself. Again, we need `root` access:

```
$ sudo cc-runtime cc-env
```

```
[Meta]
  Version = "1.0.6"
...
[Hypervisor]
  MachineType = "pc"
  Version = "QEMU emulator version 2.7.1(2.7.1+git.d4a337fe91-9.cc)..."
  Path = "/usr/bin/qemu-lite-system-x86_64"
  Debug = false

[Image]
  Path = "/usr/share/clear-containers/clear-19490-containers.img"

[Kernel]
  Path = "/usr/share/clear-containers/vmlinuz-4.9.60-80.container"
  Parameters = ""
...
```

This shows us the version we're running and tells us that it's backed by QEMU. We can see the binary it will use to start the containers. You can see that you could change the processor architecture here by swapping which binary it uses, and actually run, for example, Docker containers intended for Linux on the ARM architecture on your AMD64 Linux machine. We've removed a bunch of content here, but left the links to the image and kernel that are used. The image is actually the startup disk the virtual machine will use on startup. It contains a whole disk, including the master boot record and partition table. The kernel will be executed for our container inside the virtual machine because, unlike on Docker containers, our Clear Containers will not share the same kernel as the system. This makes sense because of how they are implemented, but this might not be obvious when you're coming from a normal Docker install. In this case, the kernel is 4.9.60-80 and our Ubuntu host is actually running 4.4.0-112!

In the end, Clear Containers work and behave more or less just like other Docker containers, but have much stronger isolation and may be a good alternative for very security-sensitive environments. The startup overhead is surprisingly small considering a whole virtual machine is being started under the covers. In most cases, you wouldn't even notice the overhead. It's a testament to how well the Docker and `containerd` abstractions work that this much more alien system can work so well with Docker.

gVisor

In mid-2018, Google released gVisor, which is a completely new take on a runtime. It's OCI compliant and so also runs underneath Docker. But it runs in user space and isolates the application by implementing system calls there rather than relying on Kernel isolation mechanisms. It doesn't redirect the calls to the kernel; rather, it actually implements them itself using kernel calls. The most obvious win from this approach is security isolation since gVisor itself is running in user space and thus isolated from the kernel. Any security issues are still trapped in user space and all of the kernel security controls we've mentioned still apply. The downside is that it performs much worse than Kernel or VM-based solutions. Given the track record of Google's past projects, it's likely gVisor will get much faster over time.

If you have processes that do not require massive scaling but do require highly secure isolation, gVisor may be an ideal solution for you. A common use case for gVisor is when your containers will be running code provided by your end users and you cannot guarantee that the code is benign. Let's run a quick demo so you can see how gVisor works.

Installation is covered on the gVisor GitHub page. It is written in Go and so is delivered as a single executable with no packages required. You then add it to the `systemd` unit file you installed previously for Clear Containers. Once that's installed, you can start containers with the `runsc` runtime. To demonstrate the different isolation level offered by gVisor, we'll run a shell using it and compare that to one using a standard container.

First, let's start a shell on gVisor and look around a bit:

```
$ docker run --runtime=runsc -i -t alpine /bin/sh
```

That will drop us into a shell running in an Alpine Linux container. One very revealing difference is apparent when you look at the output of the `mount` command:

```
/ # mount
none on / type overlayfs (rw)
none on /proc type proc (rw)
none on /sys type overlayfs (rw)
none on /etc/resolv.conf type 9p (rw)
none on /etc/hostname type 9p (rw)
none on /etc/hosts type 9p (rw)
none on /dev type devtmpfs (rw)
none on /tmp type tmpfs (rw)
```

There is not very much in there! Compare that with the output from a cgroups and namespaces-based traditional container:

```
$ docker run -i -t alpine /bin/sh
/ # mount
overlay on / type overlay (rw,relatime,...)
...
cgroup on /sys/fs/cgroup/devices type cgroup (ro,nosuid,nodev,noexec,...)
cgroup on /sys/fs/cgroup/blkio type cgroup (ro,nosuid,nodev,noexec,...)
...
/dev/sda1 on /etc/resolv.conf type ext4 (rw,relatime,errors=remount-ro,...
/dev/sda1 on /etc/hostname type ext4 (rw,relatime,errors=remount-ro,...)
...
proc on /proc/sys type proc (ro,relatime)
proc on /proc/sysrq-trigger type proc (ro,relatime)
tmpfs on /proc/kcore type tmpfs (rw,nosuid,size=65536k,mode=755)
...
tmpfs on /sys/firmware type tmpfs (ro,relatime)
```

This output was 35 lines long, so we truncated it a lot. It should be pretty clear that there is a lot of system detail here. That detail represents the kernel footprint exposed to the container in one way or another. The contrast with the very short output from gVisor should give you an idea of the differing level of isolation. We won't spend a lot more time on it, but it's also worth looking at the output of `ip addr show` as well. On gVisor:

```
/ # ip addr show
1: lo: <>
    link/generic
    inet 127.0.0.1/32 scope global dynamic
```

```
2: eth0: <>
    link/generic
    inet 172.17.0.2/32 scope global dynamic
```

And in a normal Docker container:

```
/ # ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
216: eth0@if217: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN>
    mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
```

Even the Linux */proc* filesystem exposes a lot less in the gVisor container:

```
/ # ls /proc
1            cpuinfo      loadavg      mounts       sys          version
9            filesystems  meminfo      stat         uptime
```

Once more comparing to a normal Docker container:

```
/ # ls /proc
1                  fb                 locks              softirqs
10                 filesystems        mdstat             stat
acpi               fs                 meminfo            swaps
asound             interrupts         misc               sys
buddyinfo          iomem              modules            sysrq-trigger
bus                ioports            mounts             sysvipc
cgroups            irq                mpt                thread-self
cmdline            kallsyms           mtrr               timer_list
consoles           kcore              net                timer_stats
cpuinfo            key-users          pagetypeinfo       tty
crypto             keys               partitions         uptime
devices            kmsg               sched_debug        version
diskstats          kpagecgroup        schedstat          version_signature
dma                kpagecount         scsi               vmallocinfo
driver             kpageflags         self               vmstat
execdomains        loadavg            slabinfo           zoneinfo
```

Aside from being more isolated, the experience inside the gVisor container is interesting because it looks a lot more like what you might expect to see in an isolated environment. There is a lot of potential here for a new runtime to handle secure applications.

Wrap-Up

That’s a quick tour of some of the more advanced concepts of Docker. Hopefully it has expanded your knowledge of what is happening under the covers and has opened up some avenues for you to continue your exploration. As you build and maintain a production platform, this background should provide you with a broad enough perspective of Docker to know where to start when you need to customize the system.

---

[1] Swapfiles are used to virtually extend your system’s memory capacity by giving it access to a large file for additional storage space.

[2] To quote the OCI website: “The Open Container Initiative (OCI) is a lightweight, open governance structure (project), formed under the auspices of the Linux Foundation, for the express purpose of creating open industry standards around container formats and runtime. The OCI was launched on June 22nd 2015 by Docker, CoreOS and other leaders in the container industry.”

Chapter 12. Container Platform Design

When implementing any technology in production, you'll often gain the most mileage by designing a resilient platform that can withstand the unexpected issues that will inevitably occur. Docker can be a powerful tool but requires attention to detail to get the whole platform right around it. As a technology that is going through very rapid growth, it is bound to produce frustrating bugs that crop up between the various components that make up your container platform.

If, instead of simply deploying Docker into your existing environment, you take the time to build a well-designed container platform on top of Docker, you can enjoy the many benefits of a Docker-based workflow while simultaneously protecting yourself from some of the sharper edges that can exist in such high-velocity projects.

Like all other technology, Docker doesn't magically solve all your problems. To reach its true potential, organizations must make very conscious decisions about why and how to use it. For small projects, it is possible to use Docker in a simple manner; however, if you plan to support a large project that can scale with demand, it's crucial that you design your applications and the platform very deliberately. This ensures that you can maximize the return on your investment in the technology. Taking the time to intentionally design your platform will also make it much easier to modify your production workflow over time. A well-designed Docker platform will ensure that your software is running on a dynamic foundation that can easily be upgraded as technology and company processes develop.

In this chapter, we will explore two open documents, "The Twelve-Factor App" and the "The Reactive Manifesto" and discuss how they relate to Docker and to building robust container platforms. Both documents contain a lot of ideas that should help guide you through the design and implementation of your container platform and ensure more resiliency and supportability across the board.

The Twelve-Factor App

In November of 2011, well before the release of Docker, Heroku cofounder Adam Wiggins and his colleagues released an article called "The Twelve-Factor App." This document describes a series of 12 practices, distilled from the experiences of the Heroku engineers, for designing applications that will thrive and grow in a modern container-based Software-as-a-Service (SaaS) environment.

Although not required, applications built with these 12 steps in mind are ideal candidates for the Docker workflow. Next we will explore each of these steps and explain why they can, in numerous ways, help improve your development cycle.

Codebase

One codebase tracked in revision control.

Many instances of your application will be running at any given time, but they should all come from the same code repository. Each and every Docker image for a given application should be built from a single source code repository that contains all the code required to build the Docker container. This ensures that the code can easily be rebuilt, and that all third-party requirements are well defined within the repository, if not actually directly included with the source code.

What this means is that building your application shouldn't require stitching together code from multiple source repositories. That is not to say that you can't have a dependency on an artifact from another repo. But it does mean that there should be a clear mechanism for determining which pieces of code were shipped when you built your application. Docker's ability to simplify dependency management is much less useful if building your application requires pulling down multiple source code repositories and stitching pieces together. It also is not very repeatable if you must know a magic incantation to get the build to work correctly.

A good test might be to give a new developer in your company a clean laptop and a paragraph of directions and then see if they can successfully build your application in under an hour. If they can't, then the process probably needs to be refined and simplified.

Dependencies

Explicitly declare and isolate dependencies.

Never rely on the belief that a dependency will be made available via some other avenue, like the operating system install. Any dependencies that your application requires should be well defined in the codebase and pulled in by the build process. This will help ensure that your application will run when deployed, without relying on libraries being installed by other people or processes. This is particularly important within a container since the container's processes are isolated from the rest of the host operating system and will usually not have access to anything outside of the host's kernel and the container image's filesystem.

The Dockerfile and language-dependent configuration files like Node's *package.json* or Ruby's Gemfile should define every nonexternal dependency required by your application. This ensures that your image will run correctly on any system to which it is deployed. Gone will be the days when you try to deploy and run your application in production only to find out that important libraries are missing or installed with the wrong version. This pattern has huge reliability and repeatability advantages, and very positive ramifications for system security. If, to fix a security issue, you update the OpenSSL or `libyaml` libraries that your Dockerized application uses, then you can be assured that it will always be running with that version wherever you deploy that particular application.

It is also important to note that most Docker base images are actually much larger than they need to be. Remember that your application process will be running on a shared kernel, and the only files that you actually need inside your base image are the ones that the process will require to run. It's good that base images are so readily available, but they can sometimes mask hidden dependencies. Although people often start with a minimal install of Alpine, Ubuntu, or Fedora, these images still contain a lot of operating system files that your process almost certainly does not need, or possibly some that you rely on without realizing it. You need to be fully aware of your dependencies, even when containerizing your application.

A good way to shed light on this is to compare a "small" base image with an image for a statically linked program written in a language like Go or C. These applications can be designed to run directly on the Linux kernel without any additional libraries or files.

To help drive this point home, it might be useful to review the exercises in "Keeping Images Small", where we explored one of these ultra-light containers, `adejonge/helloworld`, and then dived into the underlying filesystem a bit and compared it with the popular `alpine` base image.

In addition to being conscientious about how you manage the filesystem layers in your images, keeping your images stripped down to the bare necessities is another great way to keep everything streamlined and your `docker pull` commands fast. It's much harder to do with interpreted languages living in a container because of the large runtimes and dependency graphs you often need to install. But the point is that you should try to keep as minimal a base layer as needed so that you can reason about your dependencies. Docker helps you package them up, but you still need to be in charge of them.

Config

Store configuration in environment variables, not in files checked into the codebase.

This makes it simple to deploy the exact same codebase to different environments, like staging and production, without maintaining complicated configuration in code or rebuilding your container for each environment. This keeps your codebase much cleaner by keeping environment-specific information like database names and passwords out of your source code repository. More importantly, though, it means that you don’t bake deployment environment assumptions into the repository, and thus it is extremely easy to deploy your applications anywhere that it might be useful. You also want to be able to test the exact same image you will ship to production. You can’t do that if you have to build an image for each environment with all of its configuration already baked in.

As discussed in Chapter 4, you can achieve this by launching `docker run` commands that leverage the `-e` command-line argument. Using `-e APP_ENV=production` tells Docker to set the environment variable `APP_ENV` to the value “production” within the newly launched container.

For a real-world example, let’s assume we pulled the image for the chat robot `hubot` with the HipChat adapter installed. We’d issue something like the following command to get it running:

```
docker run \
  -e BIND_ADDRESS="0.0.0.0"
  -e ENVIRONMENT="development" \
  -e SERVICE_NAME="hubot" \
  -e SERVICE_ENV="development" \
  -e EXPRESS_USER="hubot" \
  -e EXPRESS_PASSWORD="Chd273gdExAmPl3wlkjdf" \
  -e PORT="8080" \
  -e HUBOT_ADAPTER="hipchat" \
  -e HUBOT_ALIAS="/" \
  -e HUBOT_NAME="hubot" \
  -e HUBOT_HIPCHAT_JID="someroom@chat.hipchat.com" \
  -e HUBOT_HIPCHAT_PASSWORD='SOMEEXAMPLE' \
  -e HUBOT_HIPCHAT_NAME="hubot" \
  -e HUBOT_HIPCHAT_ROOMS="anotherroom@conf.hipchat.com" \
  -e HUBOT_HIPCHAT_JOIN_ROOMS_ON_INVITE="true" \
  -e REDIS_URL="redis://redis:6379" \
  -d --restart="always" --name hubot hubot:latest
```

Here, we are passing a whole set of environment variables into the container when it is created. When the process is launched in the container, it will have access to these environment variables so that it can properly configure itself at runtime. These configuration items are now an external dependency that we can inject at runtime.

NOTE

There are many other ways to provide this data to a container, including using key/value stores like `etcd` and `consul`. Environment variables are simply a universal option that acts as a very good starting point for most projects. They are the easy path for Docker configuration because they are well supported by the platform. They all aid in observability of your applications because the configuration can be inspected with `docker inspect`.

In the case of a Node.js application like `hubot`, you could then write the following code to make decisions based on these environment variables:

```
switch(process.env.ENVIRONMENT){
        case 'development':
            console.log('Running in development');

        case 'staging':
            console.log('Running in staging');

        case 'production':
            console.log('Running in production');

        default:
            console.log('Assuming that I am running in development');
    }
```

NOTE

The exact method used to pass this configuration data into your container will vary depending on the specific tooling that you’ve chosen for your projects, but almost all of them will make it easy to ensure that every deployment contains the proper settings for that environment.

Keeping specific configuration information out of your source code makes it very easy to deploy the exact same container to multiple environments, with no changes and no sensitive information committed into your source code repository. Crucially, it supports testing your container images thoroughly before deploying to production by allowing the same image to be used in all environments.

TIP

If you need a process for managing secrets that need to be provided to your containers, you might want to look into the documentation for the `docker secret` command and HashiCorp’s Vault.

Backing Services

Treat backing services as attached resources.

Local databases are no more reliable than third-party services, and should be treated as such. Applications should handle the loss of an attached resource gracefully. By implementing graceful degradation in your application and never assuming that any resource, including filesystem space, is available, you ensure that your application will continue to perform as many of its functions as it can, even when external resources are unavailable.

This isn’t something that Docker helps you with directly, and although it is always a good idea to write robust services, it is even more important when you are using containers. When using containers, you achieve high availability most often through horizontal scaling and rolling deployments, instead of relying on the live migration of long-running process, like on traditional virtual machines. This means that specific instances of a service will often come and go over time and your service should be able to handle this gracefully.

Additionally, because Docker containers have limited filesystem resources, you can’t simply rely on having some local disk available. You need to plan that into your application’s dependencies and handle it explicitly.

Build, Release, Run

Strictly separate build and run stages.

Build the code, release it with the proper configuration, and then deploy it. This ensures that you maintain control of the process and can perform any single step without triggering the whole workflow. By ensuring that each of these steps is self-contained in a distinct process, you can tighten the feedback loop and react more quickly to any problems within the deployment flow.

As you design your Docker workflow, then, you want to clearly separate each step in the deployment process. It is perfectly fine to have a single button that builds a container, tests it, and then deploys it, assuming that you trust your testing processes—but you don’t want to be forced to rebuild a container simply in order to deploy it to another environment.

Docker supports the 12-factor ideal well in this area because there is a clean hand-off point between building an image and shipping it to production: the registry. If your build process generates images and pushes them to the registry, then deployment can simply be pulling the image down to servers and running it.

Processes

Execute the app as one or more stateless processes.

All shared data must be accessed via a stateful backing store so that application instances can easily be redeployed without losing any important session data. You don't want to keep critical state on disk in your ephemeral container, nor in the memory of one of its processes. Containerized applications should always be considered ephemeral. A truly dynamic container environment requires the ability to destroy and recreate containers at a moment's notice. This flexibility helps enable the rapid deployment cycle and outage recovery demanded by modern, Agile workflows.

As much as possible, it is preferable to write applications that do not need to keep state longer than the time required to process and respond to a single request. This ensures that the impact of stopping any given container in your application pool is very minimal. When you must maintain state, the best approach is to use a remote datastore like Redis, PostgreSQL, Memcache, or even Amazon S3, depending on your resiliency needs.

Port Binding

Export services via port binding.

Your application needs to be addressable by a port specific to itself. Applications should bind directly to a port to expose the service and should not rely on an external daemon like `inetd` to handle that for them. You should be certain that when you're talking to that port, you're talking to your application. Most modern web platforms are quite capable of directly binding to a port and servicing their own requests.

To expose a port from your container, as discussed in Chapter 4, you can launch `docker run` commands that use the `-p` command-line argument. Using `-p 80:8080`, for example, would tell Docker to proxy the container's port 8080 on the host's port 80.

The statically linked Go Hello World container that we discussed in "Keeping Images Small" is a great example of this, because the container contains nothing but our application to serve its content to a web browser. We did not need to include any additional web servers, which would require further configuration, introduce additional complexity, and increase the number of potential failure points in our system.

Concurrency

Scale out via the process model.

Design for concurrency and horizontal scaling within your applications. Increasing the resources of an existing instance can be difficult and hard to reverse. Adding and removing instances as scale fluctuates is much easier and helps maintain flexibility in the infrastructure. Launching another container on a new server is incredibly inexpensive compared to the effort and expense required to add resources to an underlying virtual or physical system. Designing for horizontal scaling allows the platform to react much faster to changes in resource requirements.

As an example, in Chapter 10, you saw how easy a service could be scaled using Docker Swarm mode by simply running a command like this:

```
docker service scale myservice=8
```

This is where tools like Docker Swarm mode, Mesos, and Kubernetes really begin to shine. Once you have implemented a Docker cluster with a dynamic scheduler, it is very easy to add three more instances of a container to the cluster as load increases, and then to later remove two instances of your application from the cluster as load starts to decrease again.

Disposability

Maximize robustness with fast startup and graceful shutdown.

Services should be designed to be ephemeral. We already talked a little bit about this when discussing external state with containers. Responding well to dynamic horizontal scaling, rolling deploys, and unexpected problems requires applications that can quickly and easily be started or shut down. Services should respond gracefully to a `SIGTERM` signal from the operating system and even handle hard failures with aplomb. Most importantly, we shouldn’t care if any given container for our application is up and running. As long as requests are being served, the developer should be freed of concerns about the health of any single component within the system. If an individual node is behaving poorly, turning it off or redeploying it should be an easy decision that doesn’t entail long planning sessions and concerns about the health of the rest of the cluster.

As discussed in Chapter 7, Docker sends standard Unix signals to containers when it is stopping or killing them; therefore, it is possible for any containerized application to detect these signals and take the appropriate steps to shut down gracefully.

Development/Production Parity

Keep development, staging, and production as similar as possible.

The same processes and artifacts should be used to build, test, and deploy services into all environments. The same people should do the work in all environments, and the physical nature of the environments should be as similar as reasonably possible. Repeatability is incredibly important. Almost any issue discovered in production points to a failure in the process. Every area where production diverges from staging is an area where risk is being introduced into the system. These inconsistencies blind you to certain types of issues that could occur in your production environment until it is too late to proactively deal with them.

In many ways, this advice essentially repeats a few of the early recommendations. However, the specific point here is that any environment divergence introduces risks, and although these differences are common in many organizations, they are much less necessary in a containerized environment. Docker servers can normally be created so that they are identical in all of your environments, and environment-based configuration changes should typically impact only which endpoints your service connects to without specifically changing the application's behavior.

Logs

Treat logs as event streams.

Services should not concern themselves with routing or storing logs. Instead, events should be streamed, unbuffered, to `STDOUT` and `STDERR` for handling by the hosting process. In development, `STDOUT` and `STDERR` can be easily viewed, whereas in staging and production, the streams can be routed to anything, including a central logging service. Different environments have different exceptions for log handling. This logic should never be hardcoded into the application. Streaming everything to `STDOUT` and `STDERR` enables the top-level process manager to handle the logs via whatever method is best for the environment, allowing the application developer to focus on core functionality.

In Chapter 6, we discussed the `docker logs` command, which collects the output from your container's `STDOUT` and `STDERR` and records it as logs. If you write logs to random files within the container's filesystem, you will not have easy access to them. It is also possible to send logs to a local or remote logging system using tools like `rsyslog`, `journald`, or `fluentd`.

If you use a process manager or initialization system on your servers, like `upstart` or `systemd`, it is usually very easy to direct all process output to `STDOUT` and `STDERR` and then have your process monitor capture them and send them to a remote logging host.

Admin Processes

Run admin/management tasks as one-off processes.

One-off administration tasks should be run via the exact same codebase and configuration that the application uses. This helps avoid problems with synchronization and code/schema drift problems. Oftentimes, management tools exist as one-off scripts or live in a completely different codebase. It is much safer to build management tools within the application's codebase, and utilize the same libraries and functions to perform the required work. This can significantly improve the reliability of these tools by ensuring that they leverage the same codepaths that the application relies on to perform its core functionality.

What this means is that you should never rely on random `cron`-like scripts to perform administrative and maintenance functions. Instead, include all of these scripts and functionality in your application codebase. Assuming that these don't need to be run on every instance of your application, you can launch a special short-lived container, or use `docker exec` with the existing container, whenever you need to run a maintenance job. This command can trigger the required job, report its status somewhere, and then exit.

Twelve-Factor Wrap-Up

While "The Twelve-Factor App" wasn't written as a Docker-specific manifesto, almost all of this advice can be applied to writing and deploying applications on a Docker platform. This is in part because the article heavily influenced Docker's design, and in part because the manifesto itself codified many of the best practices promoted by modern software architects.

The Reactive Manifesto

Riding alongside "The Twelve-Factor App," another pertinent document was released in July of 2013 by Typesafe cofounder and CTO Jonas Bonér, entitled: "The Reactive Manifesto." Jonas originally worked with a small group of contributors to solidify a manifesto that discusses how the expectations for application resiliency have evolved over the last few years, and how applications should be engineered to react in a predictable manner to various forms of interaction, including events, users, load, and failures.

"The Reactive Manifesto" states that "reactive systems" are responsive, resilient, elastic, and message-driven.

Responsive

The system responds in a timely manner if at all possible.

In general, this means that the application should respond to requests very quickly. Users simply don’t want to wait, and there is almost never a good reason to make them. If you have a containerized service that renders large PDF files, design it so that it immediately responds with a *job submitted* message so that users can go about their day, and then provide a message or banner that informs them when the job is finished and where they can download the resulting PDF.

Resilient

The system stays responsive in the face of failure.

When your application fails for any reason, the situation will always be worse if it becomes unresponsive. It is much better to handle the failure gracefully, and dynamically reduce the application’s functionality or even display a simple but clear problem message to the user while reporting the issue internally.

Elastic

The system stays responsive under varying workload.

With Docker, you achieve this by dynamically deploying and decommissioning containers as requirements and load fluctuate so that your application is always able to handle server requests quickly, without deploying a lot of underutilized resources.

Message-Driven

Reactive systems rely on asynchronous message passing to establish a boundary between components that ensures loose coupling, isolation, and location transparency.

Although not directly addressed by Docker, the idea here is that there are times when an application can become busy or unavailable. If you utilize asynchronous message passing between your services, you can help ensure that your services will not lose requests and that they will be processed as soon as possible.

Wrap-Up

All four of the design features in “The Reactive Manifesto” require application developers to design graceful degradation and a clear separation of responsibilities into their applications. By treating all dependencies as properly designed, attached resources, dynamic container environments allow you to easily maintain *N*+2 status across your application stack, reliably scale individual services in your environment, and quickly replace unhealthy nodes.

A service is only as reliable as its least reliable dependency, so it is vital to incorporate these ideas into every component of your platform.

The core ideas in “The Reactive Manifesto” merge very nicely with “The Twelve-Factor App” and the Docker workflow. These documents successfully summarize many of the most important discussions about the way you need to think and work if you want to be successful in meeting new expectations in the industry. The Docker workflow provides a practical way to implement many of these ideas in any organization in a completely approachable manner.

Chapter 13. Conclusion

At this point you have had a solid tour through the Docker ecosystem, and have seen many examples of how Docker can benefit you and your organization. We have tried to map out some of the common pitfalls and impart some of the wisdom that we have picked up over the many years that we've run Docker in production. Our experience has shown that the promise of Docker is quite achievable, and we've seen significant benefits in our organizations as a result. Like other powerful technologies, Docker is not without its compromises, but the net result has been a big positive for us, our teams, and our organizations. If you implement the Docker workflow and integrate it into the processes you already have in your organization, there is every reason to believe that you can significantly benefit from it as well. So let's quickly review the problems that Docker is designed to help you solve and some of the power it brings to the table.

The Challenges Docker Addresses

In traditional deployment workflows, there are often a multitude of required steps that significantly contribute to the overall pain felt by teams. Every step you add to the deployment process for an application increases the risk inherent in shipping it to production. Docker combines a workflow with a simple toolset that is directly targeted at addressing these concerns. Along the way, it squarely aims your development processes toward some of the industry's best practices, and its opinionated approach often leads to better communication and more robustly crafted applications.

Some of the specific problems that Docker can help mitigate include:

- Large divergence between deployment environments

- Requiring application developers to recreate configuration and logging logic in applications

- Outdated build and release processes that require multiple levels of handoff between development and operations teams

- The complex and fragile build and deploy processes
- Divergent dependency versions required by applications that need to share the same hardware, and the related software needed to manage those dependencies
- Managing multiple Linux distributions in the same organization
- Building one-off deployment processes for each application you put into production
- The constant need to audit and patch dependencies for security vulnerabilities while running your application in production
- And much more

By using the registry as a handoff point, Docker eases and simplifies communication between operations and development teams, or between multiple development teams on the same project. By bundling all of the dependencies for an application into one shipping artifact, Docker eliminates concerns about which Linux distribution developers want to work on, which versions of libraries they need to use, and how they compile their assets or bundle their software. It isolates operations teams from the build process and puts developers in charge of their dependencies.

The Docker Workflow

Docker's workflow helps organizations tackle really hard problems—some of the same problems that DevOps processes are aimed at solving. A major problem in incorporating DevOps successfully into a company's processes is that many people have no idea where to start. Tools are often incorrectly presented as the solution to what are fundamentally process problems. Adding virtualization, automated testing, deployment tools, or configuration management suites to the environment often just changes the nature of the problem without delivering a resolution.

It would be easy to dismiss Docker as just another tool making unfulfillable promises about fixing your business processes, but that would be selling it short. Docker's power in the way that its natural workflow allows applications to travel through their whole lifecycle, from conception to retirement, within one ecosystem. Unlike other tools that often target only a single aspect of the DevOps pipeline, Docker significantly improves almost every step of the process. That workflow is often opinionated, but it simplifies the adoption of some of the core principles of DevOps. It encourages development teams to understand the whole lifecycle of their application, and allows operations teams to support a much wider variety of applications on the same runtime environment. And that delivers value across the board.

Minimizing Deployment Artifacts

Docker alleviates the pain that is often induced by sprawling deployment artifacts. It does this by defining the result of a build as a single artifact, the Docker image, which contains everything your Linux application requires to run, and it executes it within a protected runtime environment. Containers can then be easily deployed on modern Linux distributions. But because of the clean split between the Docker client and server, developers can build their applications on non-Linux systems and still participate in the Linux container environment remotely.

Leveraging Docker allows software developers to create Docker images that, starting with the very first proof of concept, can be run locally, tested with automated tools, and deployed into integration or production environments without ever having to be rebuilt. This ensures that the application that is launched in production is exactly the same as what was tested. Nothing needs to be recompiled or repackaged during the deployment workflow, which significantly lowers the risks normally inherent in most deployment processes. It also means that a single build step replaces a typically error-prone process that involves compiling and packaging multiple complex components for distribution.

Docker images also simplify the installation and configuration of an application. Every single piece of software that an application requires to run on a modern Linux kernel is contained in the image, and the dependency conflicts you might find in a traditional environment are eliminated. This makes it trivial to run multiple applications that rely on different versions of core system software on the exact same server.

Optimizing Storage and Retrieval

Docker leverages filesystem layers to allow containers to be built from a composite of multiple images. This shaves a vast amount of time and effort off of many deployment processes by shipping only significant changes across the wire. It also saves considerable disk space by allowing multiple containers to be based on the same lower-level base image, and then utilizing a copy-on-write process to write new or modified files into a top layer. This also helps in scaling an application by allowing more copies of an application to be started on the same servers without the need to push the binaries across the wire for each new instance.

To support image retrieval, Docker leverages the image registry for hosting images. While not revolutionary on the face of it, the registry actually helps split team responsibilities clearly along the lines embraced by DevOps principles. Developers can build their application, test it, ship the final image to the registry, and deploy the image to the production environment, while the operations team can focus on building excellent deployment and cluster management tooling that pulls from the registry, runs reliably, and ensures environmental health. Operations teams can provide feedback to developers and see the results of all the test runs at build time rather than waiting to find problems when the application is shipped to production. This enables both teams to focus on what they do best without a multiphase handoff process.

The Payoff

As teams become more confident with Docker and its workflow, the realization often dawns that containers create a powerful abstraction layer between all of their software components and the underlying operating system. Organizations can begin to move away from having to create custom physical servers or virtual machines for most applications, and instead deploy fleets of identical Docker hosts that can be used as a large pool of resources to dynamically deploy their applications to, with an ease that was unheard of previously.

When these process changes are successful, the cultural impact within a software engineering organization can be dramatic. Developers gain more ownership of their complete application stack, including many of the smallest details, which would typically be handled by a completely different group. Operations teams are simultaneously freed from trying to package and deploy complicated dependency trees with little or no detailed knowledge of the application.

In a well-designed Docker workflow, developers compile and package the application, which makes it much easier for them to focus on ensuring that the application is running properly in all environments, without worrying about significant changes introduced to the application environment by the operations teams. At the same time, operations teams are freed from spending most of their time supporting the application and can focus on creating a robust and stable platform for the application to run on. This dynamic creates a very healthy environment in which teams have clearer ownership and responsibilities in the application delivery process, and friction between them is significantly decreased.

Getting the process right has a huge benefit to both the company and the customers as well. With organizational friction removed, software quality is improved, processes are streamlined, and code ships to production faster. This all helps free the organization to spend more time providing a satisfying customer experience and delivering directly to the broader business objectives. A well-implemented Docker-based workflow can greatly help organizations achieve those goals.

The Final Word

You are now armed with knowledge that we hope can help you make the transition to a modern, container-based build and deployment process. We encourage you to experiment with Docker on a small scale on your laptop or in a virtual machine to further your understanding of how all of the pieces fit together, and then consider how you might begin to implement it for your organization. Every company or individual developer will follow a different path determined by their own needs and competencies. If you’re looking for guidance on how to start, we’ve found success in tackling the deployment problem first with simpler tools, and then moving on to tasks like service discovery and distributed scheduling. Docker can be made as complicated as you like, but as with anything, starting simple usually pays off.

We hope you can now go forth with the knowledge we’ve imparted and make good on some of the promises for yourself.