Docker IN ACTION

SECOND EDITION

Jeff Nickoloff
Stephen Kuenzli

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Docker in Action
Second Edition
Version 4

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Docker in Action Second Edition*. Quite a bit has changed in the container world since the publication of the first edition. This book is designed as a guide for beginner and intermediate Linux software users or authors. It will guide you through an introduction to Docker, common container concepts, continuous container delivery, and more intermediate topics. This material will help set a foundation for the challenges of a still rapidly changing container and orchestration ecosystem.

In this edition I've brought on Stephen Kuenzli as a co-author. We are both engineering consultants who help organizations and individuals make the shift into containerized software paradigms.

The content in this book presumes an entry-level reader and some sections are included to provide relevant background where it might be helpful. None of us are masters of every topic and the background is often appreciated. Further each chapter is episodic. So, while it is easy to skip chapters, it might be more difficult to read individual sections out of context. More experienced technicians might want to skip around and pick out chapters where they seek specific insight.

Like the first edition, the book is broken up into three parts. Part 1 covers running containers and other basics on a single host. Part 2 digs into building custom images, packaging your software, and publishing that software. Part 3 describes the early steps in multi-container orchestration, clustering with Swarm, and other concepts that you're likely to encounter in real-world service software environments.

This edition has refined the content from the first book. Several chapters will have undergone major changes. New features will be covered and some legacy features - while still available - will have been dropped from the text. In dropping coverage of legacy topics, we hope we're able to deliver a more focused narrative. One that will help people adopt current best-practices as their default. This book is not a complete reference or guide to the full Docker feature set.

All of the source code and images used the book are available on https://github.com/dockerinaction and on https://hub.docker.com/r/dockerinaction.

—Jeff Nickoloff

brief contents

PART 1: PROCESS ISOLATION AND ENVIRONMENT-INDEPENDENT COMPUTING

1 Welcome to Docker

2 Running Software in Containers

3 Software Installation Simplified

4 Working with storage and volumes

5 Single Host Networking

6 Limiting risk with resource controls

PART 2: PACKAGING SOFTWARE FOR DISTRIBUTION

7 Packaging Software in Images

8 Building images automatically with Dockerfiles

9 Public and Private Software Distribution

10 Image Pipelines

PART 3: HIGHER-LEVEL ABSTRACTIONS AND ORCHESTRATION

11 Orchestrate services locally with Docker Compose

12 First-Class Configuration Abstractions

13 Orchestrate Services on a Cluster of Docker Hosts with Swarm Mode

14 Advanced Service Clustering with Swarm Mode

PART 4: DOCKERIZED WORKFLOWS AND OPERATIONS

15 Operating Containers in Production

Appendix A: Clusters with Docker Machine and Swarm v1

Part 1

Process Isolation and Environment-Independent Computing

Isolation is a core concept to so many computing patterns, resource management strategies, and general accounting practices that it is difficult to even begin compiling a list. Someone who learns how Linux containers provide isolation for running programs and how to use Docker to control that isolation can accomplish amazing feats of reuse, resource efficiency, and system simplification.

The most difficult part of learning how to apply containers is in translating the needs of the software you are trying to isolate. Different programs have different requirements. Web services are different from text editors, package managers, compilers, or databases. Containers for each of those programs will need different configurations.

This part covers container configuration and operation fundamentals. It does expand into more detailed container configurations to demonstrate the full spectrum of capabilities. For that reason, I suggest that you try to resist the urge to skip ahead. It may take some time to get to the specific question that is on your mind, but I'm confident that you'll have more than a few revelations along the way.

1

Welcome to Docker

This chapter covers

- What Docker is
- Example: “Hello, World”
- An introduction to containers
- How Docker addresses software problems that most people tolerate
- When, where, and why you should use Docker
- Getting help

A "best practice" is an optional investment in your product, or system that should yield better outcomes in the future. They are things that enhance security, prevent conflicts, improve serviceability, or increase longevity. Best practices often need advocates because it can be difficult to justify the immediate cost. This is especially so when the future of the system or product is uncertain. Docker is a tool that makes adopting software packaging, distribution, and utilization best practices cheap and sensible defaults. It does so by providing a complete vision for process containers and simple tooling for building and working with them.

If you’re on a team that operates service software with dynamic scaling requirements, then deploying software with Docker can help reduce customer impact. Containers come up more quickly and consume fewer resources than using virtual machines.

Teams that use continuous integration and continuous deployment techniques can build more expressive pipelines and create more robust functional testing environments if they use Docker. The containers being tested hold the same software that will go to production. The results are higher production change confidence, tighter production change control, and faster iteration.

If your team uses Docker to model local development environments you will decrease member onboarding time and eliminate the inconsistencies that slow you down. Those same environments can be version controlled with the software and updated as the software requirements change.

Software authors usually know how to install and configure their software with sensible defaults and required dependencies. If you write software, distributing that software with Docker will make it easier for your users to install and run it. They will be able to leverage the default configuration and helper material that you include. If you use Docker you can reduce your product "Installation Guide" to a single command and a single portable dependency.

Where software authors understand dependencies, installation, and packaging it is system administrators who understand the systems where the software will run. Docker provides an expressive language for running software in containers. That language lets system administrators inject environment specific configuration and tightly control access to system resources. That same language, coupled with built-in package management tooling and distribution infrastructure make deployments declarative, repeatable, and trustworthy. It promotes disposable system paradigms, persistent state isolation, and other best practices that help system administrators focus on higher-value activities.

Launched in March 2013, Docker works with your operating system to package, ship, and run software. You can think of Docker as a software logistics provider that will save you time and let you focus on core competencies. You can use Docker with network applications like web servers, databases, and mail servers and with terminal applications like text editors, compilers, network analysis tools, and scripts; in some cases it's even used to run GUI applications like web browsers and productivity software.

Docker runs Linux software on most systems. Docker for Mac and Docker for Windows integrate with common VM technology to create portability with Windows, and MacOS. But it can run native Windows applications on modern Windows server machines.

Docker isn't a programming language, and it isn't a framework for building software. Docker is a tool that helps solve common problems like installing, removing, upgrading, distributing, trusting, and running software. It's open source Linux software, which means that anyone can contribute to it, and it has benefited from a variety of perspectives. It's common for companies to sponsor the development of open source projects. In this case, Docker Inc. is the primary sponsor. You can find out more about Docker Inc. at https://docker.com/company/.

1.1 What is Docker?

If you're picking up this book you have probably already heard of Docker. Docker is an open source project for building, shipping, and running programs. It is a command-line program, a background process, and a set of remote services that take a logistical approach to solving common software problems and simplifying your experience installing, running, publishing,

and removing software. It accomplishes this using an operating system technology called containers.

1.1.1 "Hello, World"

This topic is easier to learn with a concrete example. In keeping with tradition, we'll use "Hello, World." Before you begin, download and install Docker for your system. Detailed instructions are kept up-to-date for every available system at https://docs.docker.com/install/. Once you have Docker installed and an active internet connection, head to your command prompt and type the following:

```
docker run dockerinaction/hello_world
```

After you do so, docker will spring to life. It will start downloading various components and eventually print out "hello world." If you run it again, it will just print out "hello world." Several things are happening in this example and the command itself has a few distinct parts.

First, you use the `docker run` command. This tells docker that you want to trigger the sequence (shown in figure 1.1) that installs and runs a program inside a container.

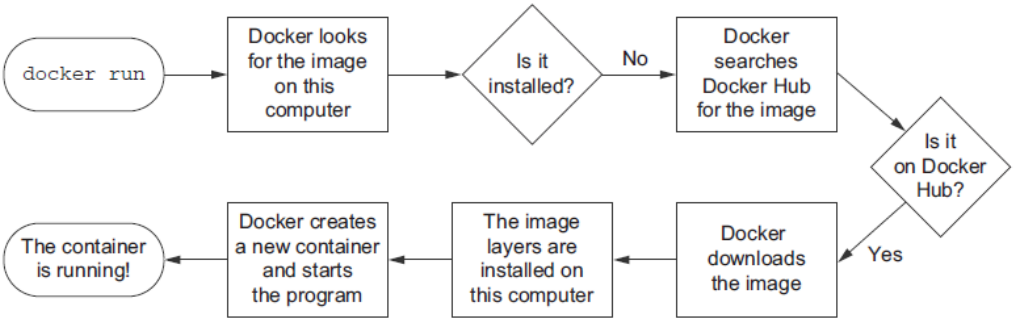


Figure 1.1 What happens after running `docker run`

The second part specifies the program that you want docker to run in a container. In this example that program is `dockerinaction/hello_world`. This is called the image (or repository) name. For now, you can think of the image name as the name of the program you want to install or run. The image itself is a collection of files and metadata. That metadata includes the specific program to execute and other relevant configuration.

> **NOTE** **This repository and several others were created specifically to support the examples in this book. By the end of part 2 you should feel comfortable examining these open source examples.**

The first time you run this command, Docker has to figure out if the `dockerinaction/hello_world` image has already been downloaded. If it's unable to locate it

on your computer (because it's the first thing you do with Docker), Docker makes a call to Docker Hub. Docker Hub is a public registry provided by Docker Inc. Docker Hub replies to Docker running on your computer where the image (`dockerinaction/hello_world`) can be found, and Docker starts the download.

Once installed, Docker creates a new container and runs a single command. In this case, the command is simple:

```
echo "hello world"
```

After the command exists after the program prints "hello world" to the terminal, and the container is marked as stopped. Understand that the running state of a container is directly tied to the state of a single running program inside the container. If a program is running, the container is running. If the program is stopped, the container is stopped. Restarting a container will run the program again.

When you give the command a second time, Docker will check again to see if `dockerinaction/hello_world` is installed. This time it will find the image on the local machine and can build another container and execute it right away. I want to emphasize an important detail. When you use `docker run` the second time, it creates a second container from the same repository (figure 1.2 illustrates this). This means that if you repeatedly use `docker run` and create a bunch of containers, you'll need to get a list of the containers you've created and maybe at some point destroy them. Working with containers is as straightforward as creating them, and both topics are covered in chapter 2.

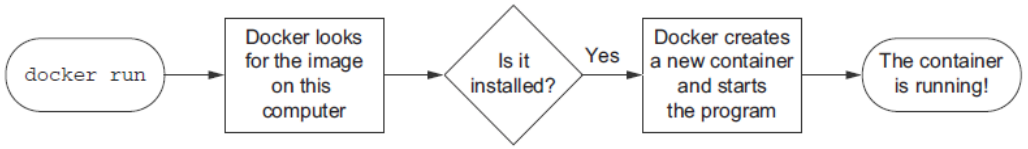


Figure 1.2 Running `docker run` a second time. Because the image is already installed, Docker can start the new container right away.

Congratulations! You're now an official Docker user. Using Docker is just this easy. But it can test your understanding of the application you are running. Consider running a web application in a container. If you did not know that it was a long running application that listened for inbound network communication on TCP port 80, you might not know exactly what docker command should be used to start that container. These are the type of sticking points people encounter as they migrate to containers.

While this book cannot speak to the needs of your specific applications it does identify the common use-cases and help teach most relevant Docker use patterns. By the end of part 1 you should have a strong command of containers with Docker.

1.1.2 Containers

Historically, UNIX-style operating systems have used the term *jail* to describe a modified runtime environment that limits the scope of resources that a *jailed* program can access. Jail features go back to 1979 and have been in evolution ever since. In 2005, with the release of Sun's Solaris 10 and Solaris Containers, *container* has become the preferred term for such a runtime environment. The goal has expanded from limiting file system scope to isolating a process from all resources except where explicitly allowed.

Using containers has been a best practice for a long time. But manually building containers can be challenging and easy to do incorrectly. This challenge has put them out of reach for some. Others using misconfigured containers are lulled into a false sense of security. This was a problem begging to be solved, and Docker helps. Any software run with Docker is run inside a container. Docker uses existing container engines to provide consistent containers built according to best practices. This puts stronger security within reach for everyone.

With Docker, users get containers at a much lower cost. Running the example in section 1.1.1 uses a container and does not require any special knowledge. As Docker and its container engines improve, you get the latest and greatest isolation features. Instead of keeping up with the rapidly evolving and highly technical world of building strong containers, you can let Docker handle the bulk of that for you.

1.1.3 Containers are not virtualization

In this cloud-native era people tend to think about virtual machines as units of deployment where deploying a single process means creating a whole network attached virtual machine. Virtual machines provide virtual hardware (or on which an operating system and other programs can be installed. They take a long time (often minutes) to create and require significant resource overhead because they run a whole operating system in addition to the software you want to use. Virtual machines can be performant once everything is up and running, but the startup delays make virtual machines a poor fit for just-in-time or reactive deployment scenarios.

Unlike virtual machines, Docker containers don't use any hardware virtualization. Programs running inside Docker containers interface directly with the host's Linux kernel. Many programs can run in isolation without running redundant operating systems or suffering the delay of full boot sequences. This is an important distinction. Docker is not a virtualization technology. Instead, it helps you use the container technology already built into your operating system kernel.

Virtual machines provide hardware abstractions so that you can run operating systems. Containers are an operating system feature. So you can always run Docker in a virtual machine if that machine is running a modern Linux. Docker for Mac and Windows users, and almost all cloud compute users will run Docker inside of virtual machines. So these are really complementary technologies.

1.1.4 Running software in containers for isolation

Containers and isolation features have existed for decades. Docker uses Linux namespaces and cgroups, which have been part of Linux since 2007. Docker doesn't provide the container technology, but it specifically makes it simpler to use. To understand what containers look like on a system, let's first establish a baseline. Figure 1.3 shows a basic example running on a simplified computer system architecture.

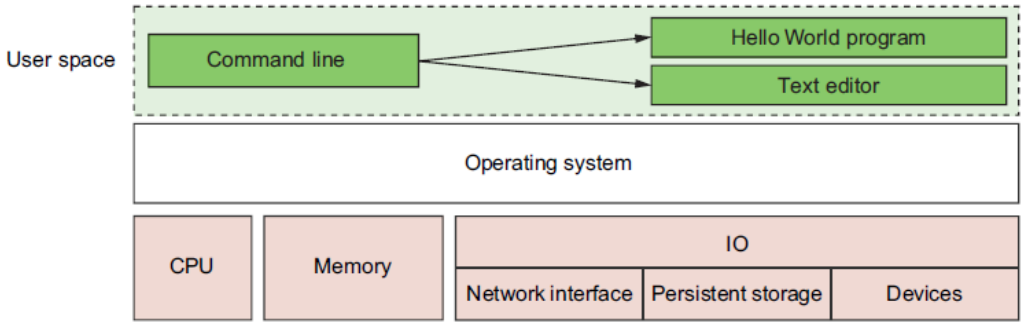


Figure 1.3 A basic computer stack running two programs that were started from the command line

Notice that the command-line interface, or CLI, runs in what is called user space memory just like other programs that run on top of the operating system. Ideally, programs running in user space can't modify kernel space memory. Broadly speaking, the operating system is the interface between all user programs and the hardware that the computer is running on.

You can see in figure 1.4 that running Docker means running two programs in user space. The first is the Docker engine. If installed properly, this process should always be running. The second is the Docker CLI. This is the Docker program that users interact with. If you want to start, stop, or install software, you'll issue a command using the Docker program.

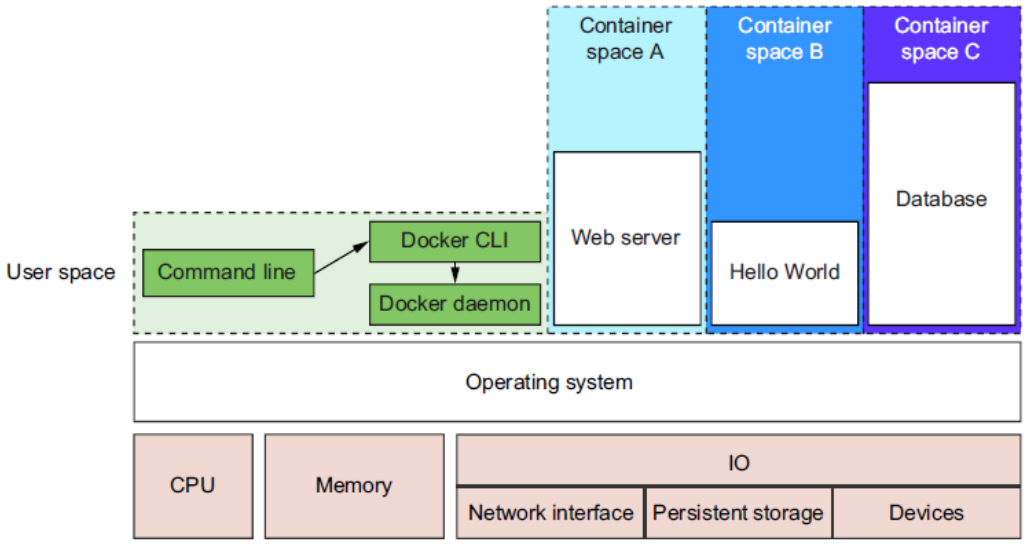


Figure 1.4 Docker running three containers on a basic Linux computer system

Figure 1.4 also shows three running containers. Each is running as a child process of the Docker engine, wrapped with a container, and the delegate process is running in its own memory subspace of the user space. Programs running inside a container can access only their own memory and resources as scoped by the container.

Docker builds containers using ten major system features. Part 1 of this book uses Docker commands to illustrate how these features can be modified to suit the needs of the contained software and to fit the environment where the container will run. The specific aspects are as follows:

- *PID namespace*—Process identifiers and capabilities
- *UTS namespace*—Host and domain name
- *MNT namespace*—File system access and structure
- *IPC namespace*—Process communication over shared memory
- *NET namespace*—Network access and structure
- *USR namespace*—User names and identifiers
- `chroot()`—Controls the location of the file system root
- *cgroups*—Resource protection
- CAP drop—Operating system feature restrictions
- Security Modules—Mandatory access controls

While Docker uses those to build containers at runtime, it uses another set of technologies to package and ship containers.

1.1.5 Shipping containers

You can think of a Docker container as a physical shipping container. It's a box where you store and run an application and all of its dependencies (excluding the running operating system kernel). Just as cranes, trucks, trains, and ships can easily work with shipping containers, so can Docker run, copy, and distribute containers with ease. Docker completes the traditional container metaphor by including a way to package and distribute software. The component that fills the shipping container role is called an *image*.

The example in section 1.1.1 used an image named `dockerinaction/hello_world`. That image contains single file: a small executable Linux program. More generally, a Docker image is a bundled snapshot of all the files that should be available to a program running inside a container. You can create as many containers from an image as you want. But when you do, containers that were started from the same image don't share changes to their file system. When you distribute software with Docker, you distribute these images, and the receiving computers create containers from them. Images are the shippable units in the Docker ecosystem.

Docker provides a set of infrastructure components that simplify distributing Docker images. These components are *registries* and *indexes*. You can use publicly available infrastructure provided by Docker Inc., other hosting companies, or your own registries and indexes.

1.2 What problems does Docker solve?

Using software is complex. Before installation you have to consider what operating system you're using, the resources the software requires, what other software is already installed, and what other software it depends on. You need to decide where it should be installed. Then you need to know how to install it. It's surprising how drastically installation processes vary today. The list of considerations is long and unforgiving. Installing software is at best inconsistent and overcomplicated. The problem is only worsened if you want to make sure that several machines use a consistent set of software over time.

Package managers like apt, brew, yum, npm, etc. attempt to manage this but few of those provide any degree of isolation. Most computers have more than one application installed and running. And most applications have dependencies on other software. What happens when applications you want to use don't play well together? Disaster. Things are only made more complicated when applications share dependencies:

- What happens if one application needs an upgraded dependency but the other does not?
- What happens when you remove an application? Is it really gone?
- Can you remove old dependencies?
- Can you remember all the changes you had to make to install the software you now want to remove?

The truth is that the more software you use, the more difficult it is to manage. Even if you can spend the time and energy required to figure out installing and running applications, how confident can you be about your security? Open and closed source programs release security updates continually, and being aware of all of the issues is often impossible. The more software you run, the greater the risk that it's vulnerable to attack.

Even enterprise grade service software must be deployed with dependencies. It is common for those projects to be shipped with and deployed to machines with hundreds if not thousands of files and other programs. Each of those create a new opportunity for conflict, vulnerability, or licensing liability.

All of these issues can be solved with careful accounting, management of resources, and logistics, but those are mundane and unpleasant things to deal with. Your time would be better spent using the software that you're trying to install, upgrade, or publish. The people who built Docker recognized that, and thanks to their hard work you can breeze through the solutions with minimal effort in almost no time at all.

It's possible that most of these issues seem acceptable today. Maybe they feel trivial because you're used to them. After reading how Docker makes these issues approachable, you may notice a shift in your opinion.

1.2.1 Getting organized

Without Docker, a computer can end up looking like a junk drawer. Applications have all sorts of dependencies. Some applications depend on specific system libraries for common things like sound, networking, graphics, and so on. Others depend on standard libraries for the language they're written in. Some depend on other applications, such as how a Java program depends on the Java Virtual Machine or a web application might depend on a database. It's common for a running program to require exclusive access to some scarce resource such as a network connection or a file.

Today, without Docker, applications are spread all over the file system and end up creating a messy web of interactions. Figure 1.5 illustrates how example applications depend on example libraries without Docker.

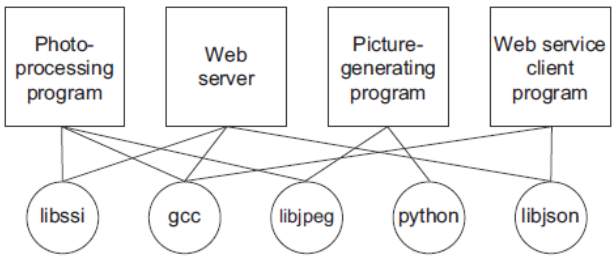


Figure 1.5 Dependency relationships of example programs

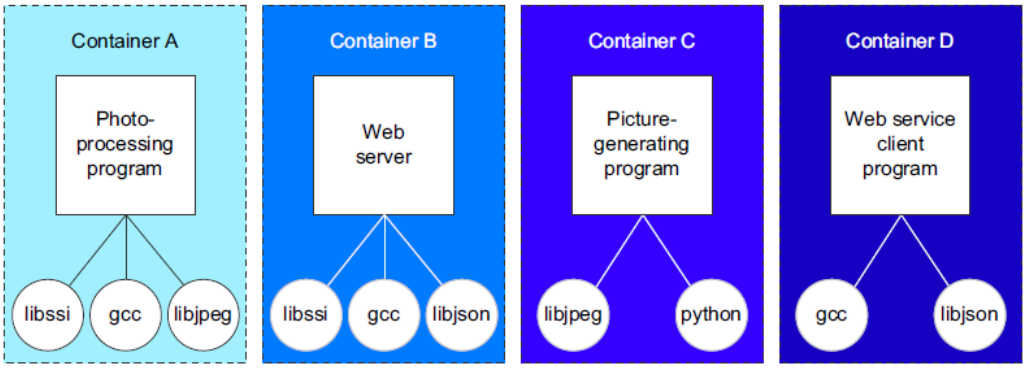


Figure 1.6 Example programs running inside containers with copies of their dependencies

In contrast, the example in section 1.1.1 installed the required software automatically, and that same software can be reliably removed with a single command. Docker keeps things organized by isolating everything with containers and images. Figure 1.6 illustrates these same applications and their dependencies running inside containers. With the links broken and each application neatly contained, understanding the system is an approachable task. At first it seems like this would introduce storage overhead by creating redundant copies of common dependencies like gcc. Chapter 3 describes how the Docker packaging system typically reduces the storage overhead.

1.2.2 Improving portability

Another software problem is that an application's dependencies typically include a specific operating system. Portability between operating systems is a major problem for software users. Although it's possible to have compatibility between Linux software and Mac OS X, using that same software on Windows can be more difficult. Doing so can require building whole ported versions of the software. Even that is only possible if suitable replacement dependencies exist for Windows. This represents a major effort for the maintainers of the application and is frequently skipped. Unfortunately for users, a whole wealth of powerful software is too difficult or impossible to use on their system.

At present, Docker runs natively on Linux and comes with a single virtual machine for OS X and Windows environments. This convergence on Linux means that software running in Docker containers need only be written once against a consistent set of dependencies. You might have just thought to yourself, "Wait a minute. You just finished telling me that Docker is better than virtual machines." That's correct, but they are complementary technologies. Using a virtual machine to contain a single program is wasteful. This is especially so when you're running several virtual machines on the same computer. On OS X and Windows, Docker uses a single, small virtual machine to run all the containers. By taking this approach, the overhead of running a virtual machine is fixed while the number of containers can scale up.

This new portability helps users in a few ways. First, it unlocks a whole world of software that was previously inaccessible. Second, it's now feasible to run the same software—exactly the same software—on any system. That means your desktop, your development environment, your company's server, and your company's cloud can all run the same programs. Running consistent environments is important. Doing so helps minimize any learning curve associated with adopting new technologies. It helps software developers better understand the systems that will be running their programs. It means fewer surprises. Third, when software maintainers can focus on writing their programs for a single platform and one set of dependencies, it's a huge time-saver for them and a great win for their customers.

Without Docker or virtual machines, portability is commonly achieved at an individual program level by basing the software on some common tool. For example, Java lets programmers write a single program that will mostly work on several operating systems because the programs rely on a program called a Java Virtual Machine (JVM). Although this is an adequate approach while writing software, other people, at other companies, wrote most of the software we use. For example, if there is a popular web server that I want to use, but it was not written in Java or another similarly portable language, I doubt that the authors would take time to rewrite it for me. In addition to this shortcoming, language interpreters and software libraries are the very things that create dependency problems. Docker improves the portability of every program regardless of the language it was written in, the operating system it was designed for, or the state of the environment where it's running.

1.2.3 Protecting your computer

Most of what I've mentioned so far have been problems from the perspective of working with software and the benefits of doing so from outside a container. But containers also protect us from the software running inside a container. There are all sorts of ways that a program might misbehave or present a security risk:

- A program might have been written specifically by an attacker.
- Well-meaning developers could write a program with harmful bugs.
- A program could accidentally do the bidding of an attacker through bugs in its input handling.

Any way you cut it, running software puts the security of your computer at risk. Because running software is the whole point of having a computer, it's prudent to apply the practical risk mitigations.

Like physical jail cells, anything inside a container can only access things that are inside it as well. There are exceptions to this rule but only when explicitly created by the user. Containers limit the scope of impact that a program can have on other running programs, the data it can access, and system resources. Figure 1.7 illustrates the difference between running software outside and inside a container.

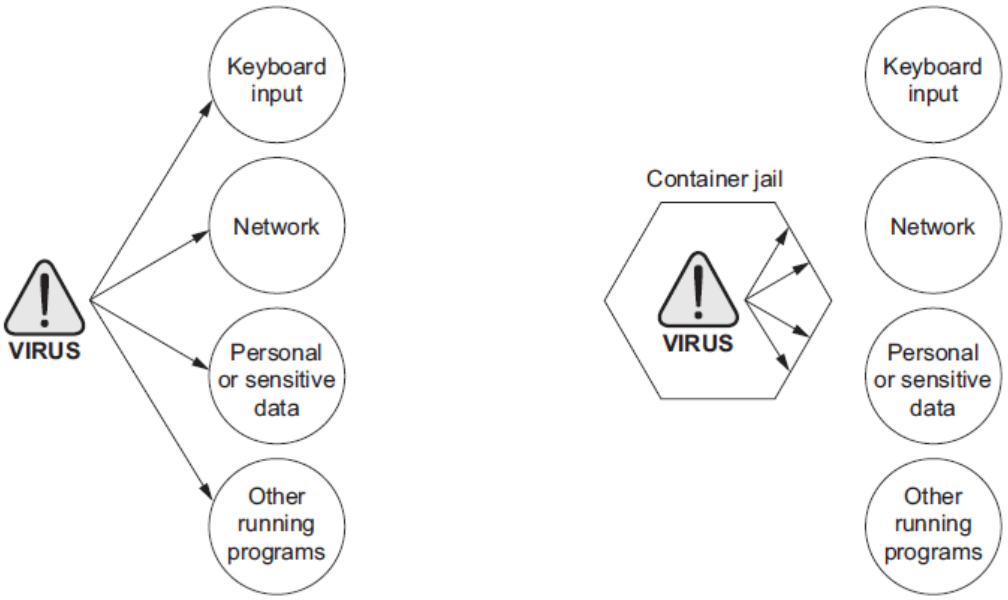


Figure 1.7 Left: a malicious program with direct access to sensitive resources. Right: a malicious program inside a container.

What this means for you or your business is that the scope of any security threat associated with running a particular application is limited to the scope of the application itself. Creating strong application containers is complicated and a critical component of any defense in-depth strategy. It is far too commonly skipped or implemented in a half-hearted manner.

1.3 Why is Docker important?

Docker provides an *abstraction*. Abstractions allow you to work with complicated things in simplified terms. So, in the case of Docker, instead of focusing on all the complexities and specifics associated with installing an application, all we need consider is what software we'd like to install. Like a crane loading a shipping container onto a ship, the process of installing any software with Docker is identical to any other. The shape or size of the thing inside the shipping container may vary, but the way that the crane picks up the container will always be the same. All the tooling is reusable for any shipping container.

This is also the case for application removal. When you want to remove software, you simply tell Docker which software to remove. No lingering artifacts will remain because they were all contained and accounted for by Docker. Your computer will be as clean as it was before you installed the software.

The container abstraction and the tools Docker provides for working with containers has changed the system administration and software development landscape. Docker is important because it makes containers available to everyone. Using it saves time, money, and energy.

The second reason Docker is important is that there is significant push in the software community to adopt containers and Docker. This push is so strong that companies like Amazon, Microsoft, and Google have all worked together to contribute to its development and adopt it in their own cloud offerings. These companies, which are typically at odds, have come together to support an open source project instead of developing and releasing their own solutions.

The third reason Docker is important is that it has accomplished for the computer what app stores did for mobile devices. It has made software installation, compartmentalization, and removal very simple. Better yet, Docker does it in a cross-platform and open way. Imagine if all of the major smartphones shared the same app store. That would be a pretty big deal. It's possible with this technology in place that the lines between operating systems may finally start to blur, and third-party offerings will be less of a factor in choosing an operating system.

Fourth, we're finally starting to see better adoption of some of the more advanced isolation features of operating systems. This may seem minor, but quite a few people are trying to make computers more secure through isolation at the operating system level. It's been a shame that their hard work has taken so long to see mass adoption. Containers have existed for decades in one form or another. It's great that Docker helps us take advantage of those features without all the complexity.

1.4 Where and when to use Docker

Docker can be used on most computers at work and at home. Practically, how far should this be taken?

Docker *can* run almost anywhere, but that doesn't mean you'll want to do so. For example, currently Docker can only run applications that can run on a Linux operating system or Windows applications on Windows Server. This means that if you want to run an OS X or Windows native application on your desktop, you can't yet do so with Docker.

By narrowing the conversation to software that typically runs on a Linux server or desktop, a solid case can be made for running almost any application inside a container. This includes server applications like web servers, mail servers, databases, proxies, and the like. Desktop software like web browsers, word processors, email clients, or other tools are also a great fit. Even trusted programs are as dangerous to run as a program you downloaded from the Internet if they interact with user-provided data or network data. Running these in a container and as a user with reduced privileges will help protect your system from attack.

Beyond the added in-depth benefit of defense, using Docker for day-to-day tasks helps keep your computer clean. Keeping a clean computer will prevent you from running into shared resource issues and ease software installation and removal. That same ease of

installation, removal, and distribution simplifies management of computer fleets and could radically change the way companies think about maintenance.

The most important thing to remember is when containers are inappropriate. Containers won't help much with the security of programs that have to run with full access to the machine. At the time of this writing, doing so is possible but complicated. Containers are not a total solution for security issues, but they can be used to prevent many types of attacks. Remember, you shouldn't use software from untrusted sources. This is especially true if that software requires administrative privileges. That means it's a bad idea to blindly run customer-provided containers in a collocated environment.

1.5 Docker in the Larger Ecosystem

Today the greater container ecosystem is rich with tooling that solves new or higher-level problems. Those problems include: container orchestration, high-availability clustering, microservice life cycle management, and visibility. It can be tricky to navigate that market without depending on keyword association. It is even trickier to understand how Docker and those products work together.

Those products work with Docker in the form of plugins or provide some higher-level functionality and depend on Docker. Some tools use the Docker subcomponents. Those subcomponents are independent projects like runc, libcontainerd, and notary.

Kubernetes is the most notable project in the ecosystem aside from Docker itself. Kubernetes provides an extensible platform for orchestrating services as containers in clustered environments. It is growing into a sort of "datacenter operating system." Like the Linux Kernel, cloud providers and platform companies are packaging Kubernetes. Kubernetes depends on container engines like Docker and so the containers and images you build on your laptop will run in Kubernetes.

There are several tradeoffs to consider when picking up any tool. Kubernetes draws power from its extensibility, but that comes at the expense of learning curve and ongoing support effort. Today building, customizing, or extending Kubernetes clusters is a full-time job. But using existing Kubernetes clusters to deploy your applications is straightforward with minimal research. Most readers looking at Kubernetes should consider adopting a managed offering from a major public cloud provider before building their own. This book focuses on and teaches solutions to higher-level problems using Docker alone. Once you understand what the problems are and how to solve them with one tool you're more likely to succeed in picking up more complicated tooling.

1.6 Getting help with the Docker command line

You'll use the `docker` command-line program throughout the rest of this book. To get you started with that, I want to show you how to get information about commands from the `docker` program itself. This way you'll understand how to use the exact version of Docker on your computer. Open a terminal, or command prompt, and run the following command:

```
docker help
```

Running `docker help` will display information about the basic syntax for using the `docker` command-line program as well as a complete list of commands for your version of the program. Give it a try and take a moment to admire all the neat things you can do.

`docker help` gives you only high-level information about what commands are available. To get detailed information about a specific command, include the command in the `<COMMAND>` argument. For example, you might enter the following command to find out how to copy files from a location inside a container to a location on the host machine:

```
docker help cp
```

That will display a usage pattern for `docker cp`, a general description of what the command does, and a detailed breakdown of its arguments. I'm confident that you'll have a great time working through the commands introduced in the rest of this book now that you know how to find help if you need it.

1.7 Summary

This chapter has been a brief introduction to Docker and the problems it helps system administrators, developers, and other software users solve. In this chapter you learned that:

- Docker takes a logistical approach to solving common software problems and simplifies your experience with installing, running, publishing, and removing software. It's a command-line program, an engine background process, and a set of remote services. It's integrated with community tools provided by Docker Inc.
- The container abstraction is at the core of its logistical approach.
- Working with containers instead of software creates a consistent interface and enables the development of more sophisticated tools.
- Containers help keep your computers tidy because software inside containers can't interact with anything outside those containers, and no shared dependencies can be formed.
- Because Docker is available and supported on Linux, OS X, and Windows, most software packaged in Docker images can be used on any computer.
- Docker doesn't provide container technology; it hides the complexity of working directly with the container software; and turns best practices into reasonable defaults.
- Docker works with the greater container ecosystem; that ecosystem is rich with tooling that solves new and higher-level problems.
- If you need help with a command you can always consult the `docker help` subcommand.

2

Running software in containers

This chapter covers

- Running interactive and daemon terminal programs with containers
- Basic Docker operations and commands
- Containers and the PID namespace
- Container configuration and output
- Running multiple programs in a container
- Injecting configuration into containers
- Durable containers and the container life cycle
- Cleaning up

Before the end of this chapter you'll understand all the basics for working with containers and how to control basic process isolation with Docker. Most examples in this book will use real software. Practical examples will help introduce Docker features and illustrate how you will use them in daily activities. Using off-the-shelf images also reduces the learning curve for new users. If you have software that you want to containerize and you're in a rush then Part 2 will likely answer more of your direct questions. In this chapter, you're going to install a web server called NGINX. Web servers are programs that make website files and programs accessible to web browsers over a network. You're not going to build a website, but you are going to install and start a web server with Docker.

2.1 Controlling containers: building a website monitor

Suppose a new client walks into your office and makes you an outrageous offer to build them a new website. They want a website that's closely monitored. This particular client wants to run their own operations, so they'll want the solution you provide to email their team when

the server is down. They've also heard about this popular web server software called NGINX and have specifically requested that you use it. Having read about the merits of working with Docker, you've decided to use it for this project. Figure 2.1 shows your planned architecture for the project.

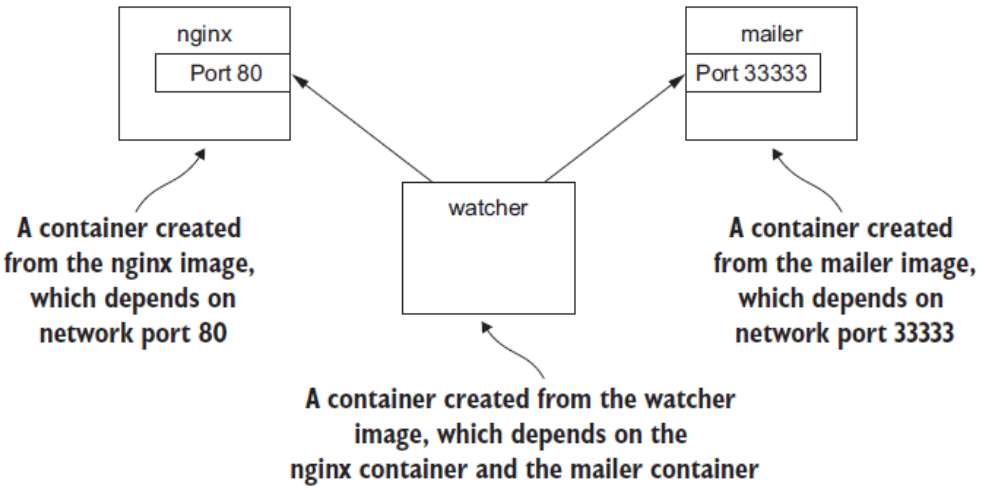


Figure 2.1 The three containers that you'll build in this example

This example uses three containers. The first will run NGINX; the second will run a program called a mailer. Both of these will run as detached containers. *Detached* means that the container will run in the background, without being attached to any input or output stream. A third program, called an agent, will run in an interactive container. Both the mailer and watcher agent are small scripts created for this example. In this section you'll learn how to do the following:

- Create detached and interactive containers
- List containers on your system
- View container logs
- Stop and restart containers
- Reattach a terminal to a container
- Detach from an attached container

Without further delay, let's get started filling your client's order.

2.1.1 Creating and starting a new container

Docker calls the collection of files and instructions needed to run a software program an image. When we install software with Docker, we are really using Docker to download or

create an image. There are different ways to install an image and several sources for images. Images are covered in more detail in chapter 3, but for now you can think of them like the shipping containers used to transport physical goods around the world. Docker images hold everything a computer needs in order to run some software.

In this example we're going to download and install an image for NGINX from Docker Hub. Remember, Docker Hub is the public registry provided by Docker Inc. The NGINX image is from what Docker Inc. calls a trusted repository. Generally, the person or foundation that publishes the software controls the trusted repositories for that software. Running the following command will download, install, and start a container running NGINX:

```
docker run --detach \                    ❶
    --name web nginx:latest
```

❶ **Note the detach flag**

When you run this command, Docker will install `nginx:latest` from the NGINX repository hosted on Docker Hub (covered in chapter 3) and run the software. After Docker has installed and started running NGINX, one line of seemingly random characters will be written to the terminal. It will look something like this:

```
7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

That blob of characters is the unique identifier of the container that was just created to run NGINX. Every time you run `docker run` and create a new container, that new container will get a unique identifier. It's common for users to capture this output with a variable for use with other commands. You don't need to do so for the purposes of this example.

After the identifier is displayed, it might not seem like anything has happened. That's because you used the `--detach` option and started the program in the background. This means that the program started but isn't attached to your terminal. It makes sense to start NGINX this way because we're going to run a few different programs. Server software is generally run in detached containers because it is rare that the software depends on an attached terminal.

Running detached containers is a perfect fit for programs that sit quietly in the background. That type of program is called a *daemon* or a *service*. A daemon generally interacts with other programs or humans over a network or some other communication channel. When you launch a daemon or other program in a container that you want to run in the background, remember to use either the `--detach` flag or its short form, `-d`.

Another daemon that your client needs in this example is a mailer. A mailer waits for connections from a caller and then sends an email. The following command will install and run a mailer that will work for this example:

```
docker run -d \                          ❶
    --name mailer \
          dockerinaction/ch2_mailer
```

❶ Start detached

This command uses the short form of the `--detach` flag to start a new container named mailer in the background. At this point you've run two commands and delivered two-thirds of the system that your client wants. The last component, called the agent, is a good fit for an interactive container.

2.1.2 Running interactive containers

A terminal-based text editor is a great example of a program that requires an attached terminal. It takes input from the user via keyboard (and maybe mouse) and displays output on the terminal. It is interactive over its input and output streams. Running interactive programs in Docker requires that you bind parts of your terminal to the input or output of a running container.

To get started working with interactive containers, run the following command:

```
docker run --interactive --tty \                    ❶
    --link web:web \
    --name web_test \
    busybox:1.29 /bin/sh
```

❶ Create a virtual terminal and bind stdin

The command uses two flags on the `run` command: `--interactive` (or `-i`) and `--tty` (or `-t`). First, the `--interactive` option tells Docker to keep the standard input stream (stdin) open for the container even if no terminal is attached. Second, the `--tty` option tells Docker to allocate a virtual terminal for the container, which will allow you to pass signals to the container. This is usually what you want from an interactive command-line program. You'll usually use both of these when you're running an interactive program like a shell in an interactive container.

Just as important as the interactive flags, when you started this container you specified the program to run inside the container. In this case you ran a shell program called `sh`. You can run any program that's available inside the container.

The command in the interactive container example creates a container, starts a UNIX shell, and is linked to the container that's running NGINX (linking is covered in chapter 5). From this shell you can run a command to verify that your web server is running correctly:

```
wget -O - http://web:80/
```

This uses a program called `wget` to make an HTTP request to the web server (the NGINX server you started earlier in a container) and then display the contents of the web page on your terminal. Among the other lines, there should be a message like "Welcome to nginx!" If you see that message, then everything is working correctly, and you can go ahead and shut down this interactive container by typing `exit`. This will terminate the shell program and stop the container.

It's possible to create an interactive container, manually start a process inside that container, and then detach your terminal. You can do so by holding down the Crtl (or Control) key and pressing P and then Q. This will work only when you've used the `--tty` option.

To finish the work for your client, you need to start an agent. This is a monitoring agent that will test the web server as you did in the last example and send a message with the mailer if the web server stops. This command will start the agent in an interactive container using the short-form flags:

```
docker run -it \                                    (1)
    --name agent \
    --link web:insideweb \
    --link mailer:insidemailer \
    dockerinaction/ch2_agent
```

(1) **Create a virtual terminal and bind stdin**

When running, the container will test the web container every second and print a message like the following:

```
System up.
```

Now that you've seen what it does, detach your terminal from the container. Specifically, when you start the container and it begins writing "System up," hold the Ctrl (or Control) key and then press P and then Q. After doing so you'll be returned to the shell for your host computer. Do not stop the program; otherwise, the monitor will stop checking the web server.

Although you'll usually use detached or daemon containers for software that you deploy to servers on your network, interactive containers are very useful for running software on your desktop or for manual work on a server. At this point you've started all three applications in containers that your client needs. Before you can confidently claim completion, you should test the system.

2.1.3 Listing, stopping, restarting, and viewing output of containers

The first thing you should do to test your current setup is check which containers are currently running by using the `docker ps` command:

```
docker ps
```

Running the command will display the following information about each running container:

- The container ID
- The image used
- The command executed in the container
- The time since the container was created
- The duration that the container has been running
- The network ports exposed by the container
- The name of the container

At this point you should have three running containers with names: web, mailer, and agent. If any is missing but you've followed the example thus far, it may have been mistakenly stopped. This isn't a problem because Docker has a command to restart a container. The next three commands will restart each container using the container name. Choose the appropriate ones to restart the containers that were missing from the list of running containers.

```
docker restart web
docker restart mailer
docker restart agent
```

Now that all three containers are running, you need to test that the system is operating correctly. The best way to do that is to examine the logs for each container. Start with the web container:

```
docker logs web
```

That should display a long log with several lines that contain this substring:

```
"GET / HTTP/1.0" 200
```

This means that the web server is running and that the agent is testing the site. Each time the agent tests the site, one of these lines will be written to the log. The `docker logs` command can be helpful for these cases but is dangerous to rely on. Anything that the program writes to the stdout or stderr output streams will be recorded in this log. The problem with this pattern is that the log is never rotated or truncated by default, so the data written to the log for a container will remain and grow as long as the container exists. That long-term persistence can be a problem for long-lived processes. A better way to work with log data uses volumes and is discussed in chapter 4.

You can tell that the agent is monitoring the web server by examining the logs for web alone. For completeness you should examine the log output for mailer and agent as well:

```
docker logs mailer
docker logs agent
```

The logs for mailer should look something like this:

```
CH2 Example Mailer has started.
```

The logs for agent should contain several lines like the one you watched it write when you started the container:

```
System up.
```

> **TIP** The `docker logs` command has a flag, `--follow` or `-f`, that will display the logs and then continue watching and updating the display with changes to the log as they occur. When you've finished, press Ctrl (or Command) and the C key to interrupt the `logs` command.

Now that you've validated that the containers are running and that the agent can reach the web server, you should test that the agent will notice when the web container stops. When that happens, the agent should trigger a call to the mailer, and the event should be recorded in the logs for both agent and mailer. The `docker stop` command tells the program with PID 1 in the container to halt. Use it in the following commands to test the system:

```
docker stop web                                    ❶
docker logs mailer                                 ❷
```

❶ Stop the web server by stopping the container
❷ Wait a couple seconds and check the mailer logs

Look for

a line at the end of the mailer logs that reads like:

```
"Sending email: To: admin@work  Message: The service is down!"
```

That line means the agent successfully detected that the NGINX server in the container named web had stopped. Congratulations! Your client will be happy, and you've built your first real system with containers and Docker.

Learning the basic Docker features is one thing, but understanding why they're useful and how to use them to customize isolation is another task entirely.

2.2 Solved problems and the PID namespace

Every running program—or process—on a Linux machine has a unique number called a process identifier (PID). A PID namespace is a set of unique numbers that identify processes. Linux provides tools to create multiple PID namespaces. Each namespace has a complete set of possible PIDs. This means that each PID namespace will contain its own PID 1, 2, 3, and so on.

Most programs will not need access to other running processes or to be able to list the other running processes on the system. And so Docker creates a new PID namespace for each container by default. A container's PID namespace isolates processes in that container from processes in other containers.

From the perspective of a process in one container with its own namespace, PID 1 might refer to an init system process like `runit` or `supervisord`. In a different container, PID 1 might refer to a command shell like bash. Run the following to see it in action:

```
docker run -d --name namespaceA \
    busybox:1.29 /bin/sh -c "sleep 30000"
docker run -d --name namespaceB \
    busybox:1.29 /bin/sh -c "nc -l 0.0.0.0 -p 80"

docker exec namespaceA ps           ①
docker exec namespaceB ps           ②
```

Command ① above should generate a process list similar to the following:

```
PID   USER     TIME  COMMAND
  1   root     0:00  sleep 30000
  8   root     0:00  ps
```

Command ② above should generate a slightly different process list:

```
PID   USER     TIME  COMMAND
  1   root     0:00  nc -l 0.0.0.0 -p 80
  9   root     0:00  ps
```

In this example you use the `docker exec` command to run additional processes in a running container. In this case, the command you use is called `ps`, which shows all the running processes and their PID. From the output it's clear to see that each container has a process with PID 1.

Without a PID namespace, the processes running inside a container would share the same ID space as those in other containers or on the host. A process in a container would be able to determine what other processes were running on the host machine. Worse, processes in one container might be able to control processes in other containers. A process that cannot reference any processes outside of its namespace is limited in its ability to perform targeted attacks.

Like most Docker isolation features, you can optionally create containers without their own PID namespace. This is critical if you are using a program to perform system administration task that requires process enumeration from within a container. You can try this yourself by setting the `--pid` flag on `docker create` or `docker run` and setting the value to `host`. Try it yourself with a container running BusyBox Linux and the `ps` Linux command:

```
docker run --pid host busybox:1.29 ps                ❶
```

❶ **Should list all processes running on the computer**

Since containers all have their own PID namespace they both cannot gain meaningful insight from examining it, and can take more static dependencies on it. Suppose a container ran two processes: a server and a local process monitor. That monitor could take a hard dependency on the server's expected PID and use that to monitor and control the server. This is an example of environment independence.

Consider the previous web-monitoring example. Suppose you were not using Docker and were just running NGINX directly on your computer. Now suppose you forgot that you had already started NGINX for another project. When you start NGINX again, the second process won't be able to access the resources it needs because the first process already has them. This is a basic software conflict example. You can see it in action by trying to run two copies of NGINX in the same container:

```
docker run –d --name webConflict nginx:latest
docker logs webConflict                              ❶
docker exec webConflict nginx -g 'daemon off;'       ❷
```

❶ **The output should be empty**

❷ Start a second nginx process in the same container

The last command should display output like:

```
2015/03/29 22:04:35 [emerg] 10#0: bind() to 0.0.0.0:80 failed (98:
Address already in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)
...
```

The second process fails to start properly and reports that the address it needs is already in use. This is called a port conflict, and it's a common issue in real-world systems where several processes are running on the same computer or multiple people contribute to the same environment. It's a great example of a conflict problem that Docker simplifies and solves. Run each in a different container, like this:

```
docker run -d --name webA nginx:latest                ❶

docker logs webA                                      ❷

docker run -d --name webB nginx:latest                ❸

docker logs webB                                      ❹
```

❶ Start the first nginx instance
❷ Verify that it is working, should be empty
❸ Start the second instance
❹ Verify that it is working, should be empty

Environment independence provides the freedom to configure software taking dependencies on scarce system resources without regard for other co-located software with conflicting requirements. Here are some common conflict problems:

- Two programs want to bind to the same network port.
- Two programs use the same temporary filename, and file locks are preventing that.
- Two programs want to use different versions of some globally installed library.
- Two processes want to use the same PID file.
- A second program you installed modified an environment variable that another program uses. Now the first program breaks.
- Multiple processes competing for memory or CPU time.

All these conflicts arise when one or more programs have a common dependency but can't agree to share or have different needs. Like in the earlier port conflict example, Docker solves software conflicts with such tools as Linux namespaces, resource limits, file system roots, and virtualized network components. All these tools are used to isolate software inside a Docker container.

2.3 Eliminating metaconflicts: building a website farm

In the last section you saw how Docker helps you avoid software conflicts with process isolation. But if you're not careful, you can end up building systems that create *metaconflicts,* or conflicts between containers in the Docker layer.

Consider another example where a client has asked you to build a system where you can host a variable number of websites for their customers. They'd also like to employ the same monitoring technology that you built earlier in this chapter. Simply expanding the system you built earlier would be the simplest way to get this job done without customizing the configuration for NGINX. In this example you'll build a system with several containers running web servers and a monitoring agent (agent) for each web server. The system will look like the architecture described in figure 2.2.

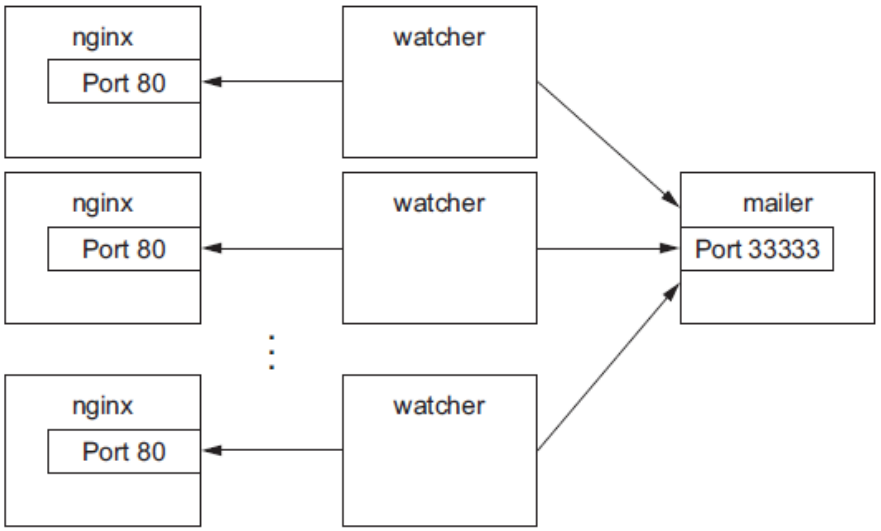


Figure 2.2 A fleet of web server containers and related monitoring agents

One's first instinct might be to simply start more web containers. That's not as simple as it looks. Identifying containers gets complicated as the number of containers increases.

2.3.1 Flexible container identification

The best way to find out why simply creating more copies of the NGINX container you used in the last example is a bad idea is to try it for yourself:

```
docker run -d --name webid nginx                    ❶

docker run -d --name webid nginx                    ❷
```

1. Create a container named "webid"
2. Create another container named "webid"

The second command here will fail with a conflict error:

```
FATA[0000] Error response from daemon: Conflict. The name "webid" is
already in use by container 2b5958ba6a00. You have to delete (or rename)
that container to be able to reuse that name.
```

Using fixed container names like *web* is useful for experimentation and documentation, but in a system with multiple containers, using fixed names like that can create conflicts. By default Docker assigns a unique (human-friendly) name to each container it creates. The `--name` flag simply overrides that process with a known value. If a situation arises where the name of a container needs to change, you can always rename the container with the `docker rename` command:

```
docker rename webid webid-old              (1)
docker run -d --name webid nginx           (2)
```

1. Rename the current web container to "webid-old"
2. Create another container named "webid"

Renaming containers can help alleviate one-off naming conflicts but does little to help avoid the problem in the first place. In addition to the name, Docker assigns a unique identifier that was mentioned in the first example. These are hex-encoded 1024-bit numbers and look something like this:

```
7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

When containers are started in detached mode, their identifier will be printed to the terminal. You can use these identifiers in place of the container name with any command that needs to identify a specific container. For example, you could use the previous ID with a `stop` or `exec` command:

```
docker exec \
    7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5 \
ps

docker stop \
    7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

The high probability of uniqueness of the IDs that are generated means that it is unlikely that there will ever be a collision with this ID. To a lesser degree it is also unlikely that there would even be a collision of the first 12 characters of this ID on the same computer. So in most Docker interfaces, you'll see container IDs truncated to their first 12 characters. This makes generated IDs a bit more user friendly. You can use them wherever a container identifier is required. So the previous two commands could be written like this:

```
docker exec 7cb5d2b9a7ea ps
```

```
docker stop 7cb5d2b9a7ea
```

Neither of these IDs is particularly well suited for human use. But they work very well with scripts and automation techniques. Docker has several means of acquiring the ID of a container to make automation possible. In these cases the full or truncated numeric ID will be used.

The first way to get the numeric ID of a container is to simply start or create a new one and assign the result of the command to a shell variable. As you saw earlier, when a new container is started in detached mode, the container ID will be written to the terminal (stdout). You'd be unable to use this with interactive containers if this were the only way to get the container ID at creation time. Luckily you can use another command to create a container without starting it. The `docker create` command is very similar to `docker run`, the primary difference being that the container is created in a stopped state:

```
docker create nginx
```

The result should be a line like:

```
b26a631e536d3caae348e9fd36e7661254a11511eb2274fb55f9f7c788721b0d
```

If you're using a Linux command shell like sh or bash, you can simply assign that result to a shell variable and use it again later:

```
CID=$(docker create nginx:latest)                ❶
echo $CID
```

❶ **This will work on POSIX-compliant shells**

Shell variables create a new opportunity for conflict, but the scope of that conflict is limited to the terminal session or current processing environment that the script was launched in. Those conflicts should be easily avoidable because one use or program is managing that environment. The problem with this approach is that it won't help if multiple users or automated processes need to share that information. In those cases you can use a container ID (CID) file.

Both the `docker run` and `docker create` commands provide another flag to write the ID of a new container to a known file:

```
docker create --cidfile /tmp/web.cid nginx       ❶

cat /tmp/web.cid                                 ❷
```

❶ **Create a new stopped container**
❷ **Inspect the file**

Like the use of shell variables, this feature increases the opportunity for conflict. The name of the CID file (provided after `--cidfile`) must be known or have some known structure. Just like manual container naming, this approach uses known names in a global (Docker-wide)

namespace. The good news is that Docker won't create a new container using the provided CID file if that file already exists. The command will fail just as it does when you create two containers with the same name.

One reason to use CID files instead of names is that CID files can be shared with containers easily and renamed for that container. This uses a Docker feature called volumes, which is covered in chapter 4.

> **TIP** One strategy for dealing with CID file-naming collisions is to partition the namespace by using known or predictable path conventions. For example, in this scenario you might use a path that contains all web containers under a known directory and further partition that directory by the customer ID. This would result in a path like /containers/web/customer1/web.cid or /containers/web/customer8/web.cid.

In other cases, you can use other commands like `docker ps` to get the ID of a container. For example, if you want to get the truncated ID of the last created container, you can use this:

```
CID=$(docker ps --latest --quiet)       ❶
echo $CID

CID=$(docker ps -l –q)                  ❷
echo $CID
```

❶ This will work on POSIX-compliant shells
❷ Run again with the short-form flags

> **TIP** If you want to get the full container ID, you can use the `--no-trunc` option on the `docker ps` command.

Automation cases are covered by the features you've seen so far. But even though truncation helps, these container IDs are rarely easy to read or remember. For this reason, Docker also generates human-readable names for each container.

The naming convention uses a personal adjective, an underscore, and the last name of an influential scientist, engineer, inventor, or other such thought leader. Examples of generated names are compassionate_swartz, hungry_goodall, and distracted_turing. These seem to hit a sweet spot for readability and memory. When you're working with the `docker` tool directly, you can always use `docker ps` to look up the human-friendly names.

Container identification can be tricky, but you can manage the issue by using the ID and name-generation features of Docker.

2.3.2 Container state and dependencies

With this new knowledge, the new system might looks something like this:

```
MAILER_CID=$(docker run -d dockerinaction/ch2_mailer)          ❶
WEB_CID=$(docker create nginx)

AGENT_CID=$(docker create --link $WEB_CID:insideweb \
    --link $MAILER_CID:insidemailer \
```

```
    dockerinaction/ch2_agent)
```

❶ Make sure mailer from first example is running

This snippet could be used to seed a new script that launches a new NGINX and agent instance for each of your client's customers. You can use `docker ps` to see that they've been created:

```
docker ps
```

The reason neither the NGINX nor the agent was included with the output has to do with container state. Docker containers will always be in one of four states and transition via command according to the diagram in figure 2.3.

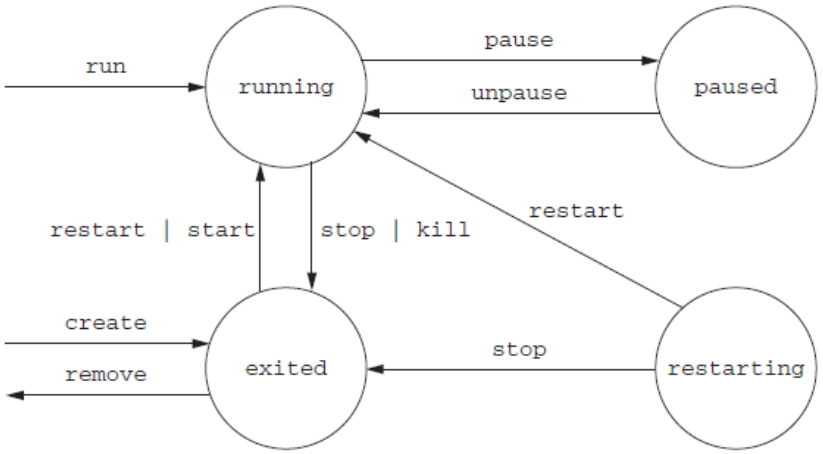


Figure 2.3 The state transition diagram for Docker containers as reported by the status column

Neither of the new containers you started appears in the list of containers because `docker ps` shows only running containers by default. Those containers were specifically created with `docker create` and never started (the exited state). To see all the containers (including those in the exited state), use the `-a` option:

```
docker ps -a
```

Now that you've verified that both of the containers were created, you need to start them. For that you can use the `docker start` command:

```
docker start $AGENT_CID
docker start $WEB_CID
```

Running those commands will result in an error. The containers need to be started in reverse order of their dependency chain. Because you tried to start the agent container before the web container, Docker reported a message like this one:

```
Error response from daemon: Cannot start container
      03e65e3c6ee34e714665a8dc4e33fb19257d11402b151380ed4c0a5e38779d0a: Cannot link
      to a non running container: /clever_wright AS /modest_hopper/insideweb
FATA[0000] Error: failed to start one or more containers
```

In this example, the agent container has a dependency on the web container. You need to start the web container first:

```
docker start $WEB_CID
docker start $AGENT_CID
```

This makes sense when you consider the mechanics at work. The link mechanism injects IP addresses into dependent containers, and containers that aren't running don't have IP addresses. If you tried to start a container that has a dependency on a container that isn't running, Docker wouldn't have an IP address to inject. Container linking is covered in chapter 5, but it's useful to demonstrate this important point in starting containers.

Whether you're using `docker run` or `docker create`, the resulting containers need to be started in the reverse order of their dependency chain. This means that circular dependencies are impossible to build using Docker container relationships.

At this point you can put everything together into one concise script that looks like:

```
MAILER_CID=$(docker run -d dockerinaction/ch2_mailer)

WEB_CID=$(docker run -d nginx)

AGENT_CID=$(docker run -d \
    --link $WEB_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)
```

Now you're confident that this script can be run without exception each time your client needs to provision a new site. Your client has come back and thanked you for the web and monitoring work you've completed so far, but things have changed.

They've decided to focus on building their websites with WordPress (a popular open source content-management and blogging program). Luckily, WordPress is published through Docker Hub in a repository named wordpress:5.0.0-php7.2-apache. All you'll need to deliver is a set of commands to provision a new WordPress website that has the same monitoring and alerting features that you've already delivered.

An interesting thing about content-management systems and other stateful systems is that the data they work with makes each running program specialized. Adam's WordPress blog is different from Betty's WordPress blog, even if they're running the same software. Only the content is different. Even if the content is the same, they're different because they're running on different sites.

If you build systems or software that know too much about their environment—like addresses or fixed locations of dependency services—it's difficult to change that environment or reuse the software. You need to deliver a system that minimizes environment dependence before the contract is complete.

2.4 Building environment-agnostic systems

Much of the work associated with installing software or maintaining a fleet of computers lies in dealing with specializations of the computing environment. These specializations come as global-scoped dependencies (like known host file system locations), hard-coded deployment architectures (environment checks in code or configuration), or data locality (data stored on a particular computer outside the deployment architecture). Knowing this, if your goal is to build low-maintenance systems, you should strive to minimize these things.

Docker has three specific features to help build environment-agnostic systems:

- Read-only file systems
- Environment variable injection
- Volumes

Working with volumes is a big subject and the topic of chapter 4. In order to learn the first two features, consider a requirements change for the example situation used in the rest of this chapter.

WordPress uses a database program called MySQL to store most of its data, so it's a good idea to provide the container running WordPress with a read-only file system to ensure data is only written to the database.

2.4.1 Read-only file systems

Using read-only file systems accomplishes two positive things. First, you can have confidence that the container won't be specialized from changes to the files it contains. Second, you have increased confidence that an attacker can't compromise files in the container.

To get started working on your client's system, create and start a container from the WordPress image using the `--read-only` flag:

```
docker run -d --name wp --read-only \
    wordpress:5.0.0-php7.2-apache
```

When this is finished, check that the container is running. You can do so using any of the methods introduced previously, or you can inspect the container metadata directly. The following command will print `true` if the container named wp is running and `false` otherwise.

```
docker inspect --format "{{.State.Running}}" wp
```

The `docker inspect` command will display all the metadata (a JSON document) that Docker maintains for a container. The format option transforms that metadata, and in this case, it

filters everything except for the field indicating the running state of the container. This command should simply output `false`.

In this case, the container isn't running. To determine why, examine the logs for the container:

```
docker logs wp
```

That command should output something like:

```
WordPress not found in /var/www/html - copying now...
Complete! WordPress has been successfully copied to /var/www/html
... skip output ...
Wed Dec 12 15:17:36 2018 (1): Fatal Error Unable to create lock file: Bad file
      descriptor (9)
```

Interesting, when running WordPress with a read-only filesystem, the Apache webserver process reports that it is unable to create a lock file. Unfortunately, it does not report the location of the files it is trying to create. If we have the locations, we can create exceptions for them. Let's run a WordPress container with a writable filesystem so that Apache is free to write where it wants:

```
docker run -d --name wp_writable wordpress:5.0.0-php7.2-apache
```

Now let's check where Apache changed the container's filesystem with the `docker diff` command:

```
docker container diff wp_writable
C /run
C /run/apache2
A /run/apache2/apache2.pid
```

We will explain the diff command and how Docker knows what changed on the filesystem in more detail in Chapter 3. For now, it's sufficient to know that the output indicates that Apache created the `/run/apache2` directory and added the `apache2.pid` file inside of it.

Since this is an expected part of normal application operation, we will make an exception to the `read-only` filesystem. We will allow the container to write to `/run/apache2` using a writable volume mounted from the host. We will also supply a temporary, in-memory, filesystem to the container at `/tmp` since Apache requires a writable temporary directory, as well:

```
docker run -d --name wp2 \
  --read-only \           ❶
  -v /run/apache2/ \      ❷
  --tmpfs /tmp \          ❸
  wordpress:5.0.0-php7.2-apache
```

❶ **Make container's root filesystem read-only**
❷ **Mount a writable directory from the host**
❸ **Provide container an in-memory temp filesystem**

That command should log successful messages that look like:

```
docker logs wp2
WordPress not found in /var/www/html - copying now...
Complete! WordPress has been successfully copied to /var/www/html
... skip output ...
[Wed Dec 12 16:25:40.776359 2018] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.25
      (Debian) PHP/7.2.13 configured -- resuming normal operations
[Wed Dec 12 16:25:40.776517 2018] [core:notice] [pid 1] AH00094: Command line:
      'apache2 -D FOREGROUND'
```

WordPress also has a dependency on a MySQL database. A database is a program that stores data in such a way that it's retrievable and searchable later. The good news is that you can install MySQL using Docker just like WordPress:

```
docker run -d --name wpdb \
    -e MYSQL_ROOT_PASSWORD=ch2demo \
    mysql:5.7
```

Once that is started, create a different WordPress container that's linked to this new database container (linking is covered in depth in chapter 5):

```
docker run -d --name wp3 \                  (1)
    --link wpdb:mysql \                     (2)
    -p 8000:80 \                            (3)
    --read-only \
    -v /run/apache2/ \
    --tmpfs /tmp \
    wordpress:5.0.0-php7.2-apache
```

(1) **Use a unique name**
(2) **Create a link to the database**
(3) **Direct traffic from host port 8000 to container port 80**

Check one more time that WordPress is running correctly:

```
docker inspect --format "{{.State.Running}}" wp3
```

The output should now be `true`. If you would like to use your new WordPress installation, you can point a web browser to http://127.0.0.1:8000.

An updated version of the script you've been working on should look like this:

```
#!/bin/sh

DB_CID=$(docker create -e MYSQL_ROOT_PASSWORD=ch2demo mysql:5.7)

docker start $DB_CID

MAILER_CID=$(docker create dockerinaction/ch2_mailer)
docker start $MAILER_CID
```

```
WP_CID=$(docker create --link $DB_CID:mysql -p 80 \
    --read-only -v /run/apache2/ --tmpfs /tmp \
    wordpress:5.0.0-php7.2-apache)

docker start $WP_CID

AGENT_CID=$(docker create --link $WP_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)

docker start $AGENT_CID
```

Congratulations, at this point you should have a running WordPress container! By using a read-only file system and linking WordPress to another container running a database, you can be sure that the container running the WordPress image will never change. This means that if there is ever something wrong with the computer running a client's WordPress blog, you should be able to start up another copy of that container elsewhere with no problems.

But there are two problems with this design. First, the database is running in a container on the same computer as the WordPress container. Second, WordPress is using several default values for important settings like database name, administrative user, administrative password, database salt, and so on. To deal with this problem, you could create several versions of the WordPress software, each with a special configuration for the client. Doing so would turn your simple provisioning script into a monster that creates images and writes files. A better way to inject that configuration would be through the use of environment variables.

2.4.2 Environment variable injection

Environment variables are key-value pairs that are made available to programs through their execution context. They let you change a program's configuration without modifying any files or changing the command used to start the program.

Docker uses environment variables to communicate information about dependent containers, the host name of the container, and other convenient information for programs running in containers. Docker also provides a mechanism for a user to inject environment variables into a new container. Programs that know to expect important information through environment variables can be configured at container-creation time. Luckily for you and your client, WordPress is one such program.

Before diving into WordPress specifics, try injecting and viewing environment variables on your own. The UNIX command `env` displays all the environment variables in the current execution context (your terminal). To see environment variable injection in action, use the following command:

```
docker run --env MY_ENVIRONMENT_VAR="this is a test" \      ❶
    busybox:1.29 \
    env                                                     ❷
```

❶ **Inject an environment variable**

❷ **Execute the env command inside the container**

The `--env` flag, or `-e` for short, can be used to inject any environment variable. If the variable is already set by the image or Docker, then the value will be overridden. This way programs running inside containers can rely on the variables always being set. WordPress observes the following environment variables:

```
WORDPRESS_DB_HOST
WORDPRESS_DB_USER
WORDPRESS_DB_PASSWORD
WORDPRESS_DB_NAME
WORDPRESS_AUTH_KEY
WORDPRESS_SECURE_AUTH_KEY
WORDPRESS_LOGGED_IN_KEY
WORDPRESS_NONCE_KEY
WORDPRESS_AUTH_SALT
WORDPRESS_SECURE_AUTH_SALT
WORDPRESS_LOGGED_IN_SALT
WORDPRESS_NONCE_SALT
```

> **TIP** This example neglects the `KEY` and `SALT` variables, but any real production system should absolutely set these values.

To get started, you should address the problem that the database is running in a container on the same computer as the WordPress container. Rather than using linking to satisfy WordPress's database dependency, inject a value for the `WORDPRESS_DB_HOST` variable:

```
docker create --env WORDPRESS_DB_HOST=<my database hostname> \
    wordpress: 5.0.0-php7.2-apache
```

This example would create (not start) a container for WordPress that will try to connect to a MySQL database at whatever you specify at `<my database hostname>`. Because the remote database isn't likely using any default user name or password, you'll have to inject values for those settings as well. Suppose the database administrator is a cat lover and hates strong passwords:

```
docker create \
    --env WORDPRESS_DB_HOST=<my database hostname> \
    --env WORDPRESS_DB_USER=site_admin \
    --env WORDPRESS_DB_PASSWORD=MeowMix42 \
    wordpress:5.0.0-php7.2-apache
```

Using environment variable injection this way will help you separate the physical ties between a WordPress container and a MySQL container. Even in the case where you want to host the database and your customer WordPress sites all on the same machine, you'll still need to fix the second problem mentioned earlier. All the sites are using the same default database name, which means different clients will be sharing a single database. You'll need to use environment variable injection to set the database name for each independent site by specifying the `WORDPRESS_DB_NAME` variable:

```
docker create --link wpdb:mysql \
    -e WORDPRESS_DB_NAME=client_a_wp \  ❶
    wordpress:5.0.0-php7.2-apache

docker create --link wpdb:mysql \
    -e WORDPRESS_DB_NAME=client_b_wp \   ❷
    wordpress:5.0.0-php7.2-apache
```

❶ For client A
❷ For client B

Now that you understand how to inject configuration into the WordPress application and connect it to collaborating processes, let's adapt the provisioning script. First, let's start database and mailer containers that will be shared by our clients and store the container ids in environment variables:

```
export DB_CID=$(docker run -d -e MYSQL_ROOT_PASSWORD=ch2demo mysql:5.7)
export MAILER_CID=$(docker run -d dockerinaction/ch2_mailer)
```

Now update the client site provisioning script to read the database container id, mailer container id, and a new CLIENT_ID from environment variables:

```
#!/bin/sh

if [ ! -n "$CLIENT_ID" ]; then                  ❶
    echo "Client ID not set"
    exit 1
fi

WP_CID=$(docker create \
    --link $DB_CID:mysql \                       ❷
    --name wp_$CLIENT_ID \
    -p 80 \
    --read-only -v /run/apache2/ --tmpfs /tmp \
    -e WORDPRESS_DB_NAME=$CLIENT_ID \
    --read-only wordpress:5.0.0-php7.2-apache)

docker start $WP_CID

AGENT_CID=$(docker create \
        --name agent_$CLIENT_ID \
    --link $WP_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)

docker start $AGENT_CID
```

❶ Assume $CLIENT_ID variable is set as input to script
❷ Create link using DB_CID

If you save this script to a file named `start-wp-for-client.sh`, you can provision WordPress for the `dockerinaction` client using a command like:

```
CLIENT_ID=dockerinaction ./start-wp-multiple-clients.sh
```

This new script will start an instance of WordPress and the monitoring agent for each customer and connect those containers to each other as well as a single mailer program and MySQL database. The WordPress containers can be destroyed, restarted, and upgraded without any worry about loss of data. Figure 2.4 shows this architecture.

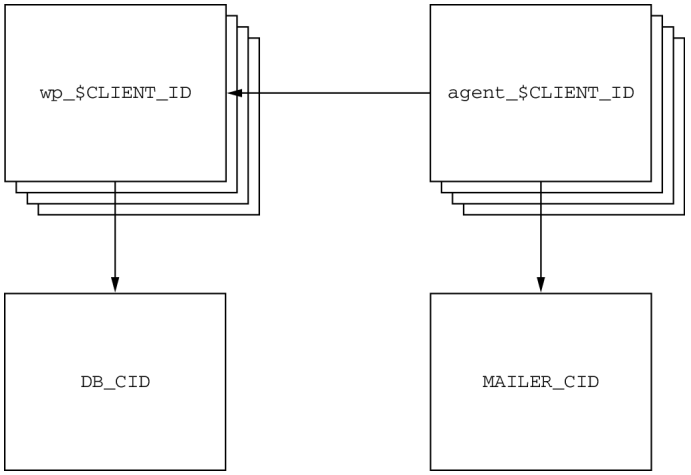


Figure 2.4 Each WordPress and agent container uses the same database and mailer.

The client should be pleased with what is being delivered. But one thing might be bothering you. In earlier testing you found that the monitoring agent correctly notified the mailer when the site was unavailable, but restarting the site and agent required manual work. It would be better if the system tried to automatically recover when a failure was detected. Docker provides restart policies to help deal with that, but you might want something more robust.

2.5 Building durable containers

There are cases where software fails in rare conditions that are temporary in nature. Although it's important to be made aware when these conditions arise, it's usually at least as important to restore the service as quickly as possible. The monitoring system that you built in this chapter is a fine start for keeping system owners aware of problems with a system, but it does nothing to help restore service.

When all the processes in a container have exited, that container will enter the exited state. Remember, a Docker container can be in one of four states:

- Running
- Paused
- Restarting

- Exited (also used if the container has never been started)

A basic strategy for recovering from temporary failures is automatically restarting a process when it exits or fails. Docker provides a few options for monitoring and restarting containers.

2.5.1 Automatically restarting containers

Docker provides this functionality with a restart policy. Using the `--restart` flag at container-creation time, you can tell Docker to do any of the following:

- Never restart (default)
- Attempt to restart when a failure is detected
- Attempt for some predetermined time to restart when a failure is detected
- Always restart the container regardless of the condition

Docker doesn't always attempt to immediately restart a container. If it did, that would cause more problems than it solved. Imagine a container that does nothing but print the time and exit. If that container was configured to always restart and Docker always immediately restarted it, the system would do nothing but restart that container. Instead, Docker uses an exponential backoff strategy for timing restart attempts.

A backoff strategy determines how much time should pass between successive restart attempts. An exponential backoff strategy will do something like double the previous time spent waiting on each successive attempt. For example, if the first time the container needs to be restarted Docker waits 1 second, then on the second attempt it would wait 2 seconds, 4 seconds on the third attempt, 8 on the fourth, and so on. Exponential backoff strategies with low initial wait times are a common service-restoration technique. You can see Docker employ this strategy yourself by building a container that always restarts and simply prints the time:

```
docker run -d --name backoff-detector --restart always busybox:1.29 date
```

Then after a few seconds use the trailing logs feature to watch it back off and restart:

```
docker logs -f backoff-detector
```

The logs will show all the times it has already been restarted and will wait until the next time it is restarted, print the current time, and then exit. Adding this single flag to the monitoring system and the WordPress containers you've been working on would solve the recovery issue.

The only reason you might not want to adopt this directly is that during backoff periods, the container isn't running. Containers waiting to be restarted are in the restarting state. To demonstrate, try to run another process in the backoff-detector container:

```
docker exec backoff-detector echo Just a Test
```

Running that command should result in an error message:

```
Cannot run exec command ... in container ...: No active container exists
with ID ...
```

That means you can't do anything that requires the container to be in a running state, like execute additional commands in the container. That could be a problem if you need to run diagnostic programs in a broken container. A more complete strategy is to use containers that start lightweight init systems.

2.5.2 PID 1 and init systems

An init system is a program that's used to launch and maintain the state of other programs. Any process with PID 1 is treated like an init process by the Linux kernel (even if it is not technically an init system). In addition to other critical functions an init system starts other processes, restarts them in the event that they fail, transforms and forwards signals sent by the operating system, and prevents resource leaks. It is common practice to use real init systems inside containers when that container will run multiple processes or if the program being run uses child processes.

There are several such init systems that might be used inside a container. The most popular include `runit`, `Yelp/dumb-init`, `tini`, `supervisord`, and `tianon/gosu`. Publishing software that uses these programs is covered in chapter 8. For now, take a look at a container that uses `supervisord`.

Docker provides an image that contains a full LAMP (Linux, Apache, MySQL PHP) stack inside a single container. Containers created this way use `supervisord` to make sure that all the related processes are kept running. Start an example container:

```
docker run -d -p 80:80 --name lamp-test tutum/lamp
```

You can see what processes are running inside this container by using the `docker top` command:

```
docker top lamp-test
```

The top subcommand will show the host PID for each of the processes in the container. You'll see `supervisord`, `mysql`, and `apache` included in the list of running programs. Now that the container is running, you can test the `supervisord` restart functionality by manually stopping one of the processes inside the container.

The problem is that to kill a process inside of a container from within that container, you need to know the PID in the container's PID namespace. To get that list, run the following exec subcommand:

```
docker exec lamp-test ps
```

The process list generated will have listed `apache2` in the CMD column:

```
PID TTY          TIME CMD
  1 ?        00:00:00 supervisord
433 ?        00:00:00 mysqld_safe
835 ?        00:00:00 apache2
842 ?        00:00:00 ps
```

The values in the PID column will be different when you run the command. Find the PID on the row for `apache2` and then insert that for `<PID>` in the following command:

```
docker exec lamp-test kill <PID>
```

Running this command will run the Linux `kill` program inside the lamp-test container and tell the `apache2` process to shut down. When `apache2` stops, the `supervisord` process will log the event and restart the process. The container logs will clearly show these events:

```
...
... exited: apache2 (exit status 0; expected)
... spawned: 'apache2' with pid 820
... success: apache2 entered RUNNING state, process has stayed up for >
      than 1 seconds (startsecs)
```

A common alternative to the use of init systems is using a startup script that at least checks the preconditions for successfully starting the contained software. These are sometimes used as the default command for the container. For example, the WordPress containers that you've created start by running a script to validate and set default environment variables before starting the WordPress process. You can view this script by overriding the default command and using a command to view the contents of the startup script:

```
docker run wordpress:5.0.0-php7.2-apache cat /entrypoint.sh
```

Running that command will result in an error messages like:

```
error: missing WORDPRESS_DB_HOST and MYSQL_PORT_3306_TCP environment
variables
...
```

This failed because even though you set the command to run as `cat /entrypoint.sh`, Docker containers run something called an entrypoint before executing the command. Entrypoints are perfect places to put code that validates the preconditions of a container. Although this is discussed in depth in part 2 of this book, you need to know how to override or specifically set the entrypoint of a container on the command line. Try running the last command again but this time using the `--entrypoint` flag to specify the program to run and using the command section to pass arguments:

```
docker run --entrypoint="cat" \                ❶
    wordpress:5.0.0-php7.2-apache \
    /usr/local/bin/docker-entrypoint.sh        ❷
```

❶ **Use "cat" as the entrypoint**
❷ **Pass the full path of the default entrypoint script as an argument to cat**

If you run through the displayed script, you'll see how it validates the environment variables against the dependencies of the software and sets default values. Once the script has validated that WordPress can execute, it will start the requested or default command.

Startup scripts are an important part of building durable containers and can always be combined with Docker restart policies to take advantage of the strengths of each. Because both the MySQL and WordPress containers already use startup scripts, it's appropriate to simply set the restart policy for each in an updated version of the example script.

Running startup scripts as PID 1 is problematic when the script fails to meet the expectations that Linux has for init systems. Depending on your use-case you might find that one approach or a hybrid works best.

With that final modification, you've built a complete WordPress site-provisioning system and learned the basics of container management with Docker. It has taken considerable experimentation. Your computer is likely littered with several containers that you no longer need. To reclaim the resources that those containers are using, you need to stop them and remove them from your system.

2.6 Cleaning up

Ease of cleanup is one of the strongest reasons to use containers and Docker. The isolation that containers provide simplifies any steps that you'd have to take to stop processes and remove files. With Docker, the whole cleanup process is reduced to one of a few simple commands. In any cleanup task, you must first identify the container that you want to stop and/or remove. Remember, to list all of the containers on your computer, use the `docker ps` command:

```
docker ps -a
```

Because the containers you created for the examples in this chapter won't be used again, you should be able to safely stop and remove all the listed containers. Make sure you pay attention to the containers you're cleaning up if there are any that you created for your own activities.

All containers use hard drive space to store logs, container metadata, and files that have been written to the container file system. All containers also consume resources in the global namespace like container names and host port mappings. In most cases, containers that will no longer be used should be removed.

To remove a container from your computer, use the `docker rm` command. For example, to delete the stopped container named wp you'd run:

```
docker rm wp
```

You should go through all the containers in the list you generated by running `docker ps -a` and remove all containers that are in the exited state. If you try to remove a container that's running, paused, or restarting, Docker will display a message like the following:

```
Error response from daemon: Conflict, You cannot remove a running container. Stop the
      container before attempting removal or use -f
FATA[0000] Error: failed to remove one or more containers
```

The processes running in a container should be stopped before the files in the container are removed. You can do this with the `docker stop` command or by using the `-f` flag on `docker rm`. The key difference is that when you stop a process using the `-f` flag, Docker sends a `SIG_KILL` signal, which immediately terminates the receiving process. In contrast, using `docker stop` will send a `SIG_HUP` signal. Recipients of `SIG_HUP` have time to perform finalization and cleanup tasks. The `SIG_KILL` signal makes for no such allowances and can result in file corruption or poor network experiences. You can issue a `SIG_KILL` directly to a container using the `docker kill` command. But you should use `docker kill` or `docker rm -f` only if you must stop the container in less than the standard 30-second maximum stop time.

In the future, if you're experimenting with short-lived containers, you can avoid the cleanup burden by specifying `--rm` on the command. Doing so will automatically remove the container as soon as it enters the exited state. For example, the following command will write a message to the screen in a new BusyBox container, and the container will be removed as soon as it exits:

```
docker run --rm --name auto-exit-test busybox:1.29 echo Hello World
docker ps -a
```

In this case, you could use either `docker stop` or `docker rm` to properly clean up, or it would be appropriate to use the single-step `docker rm -f` command. You should also use the `-v` flag for reasons that will be covered in chapter 4. The docker CLI makes it is easy to compose a quick cleanup command:

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

This concludes the basics of running software in containers. Each chapter in the remainder of part 1 will focus on a specific aspect of working with containers. The next chapter focuses on installing and uninstalling images, how images relate to containers, and working with container file systems.

2.7 Summary

The primary focus of the Docker project is to enable users to run software in containers. This chapter shows how you can use Docker for that purpose. The ideas and features covered include the following:

- Containers can be run with virtual terminals attached to the user's shell or in detached mode.
- By default, every Docker container has its own PID namespace, isolating process information for each container.
- Docker identifies every container by its generated container ID, abbreviated container ID, or its human-friendly name.
- All containers are in any one of four distinct states: running, paused, restarting, or exited.

- The `docker exec` command can be used to run additional processes inside a running container.
- A user can pass input or provide additional configuration to a process in a container by specifying environment variables at container-creation time.
- Using the `--read-only` flag at container-creation time will mount the container file system as read-only and prevent specialization of the container.
- A container restart policy, set with the `--restart` flag at container-creation time, will help your systems automatically recover in the event of a failure.
- Docker makes cleaning up containers with the `docker rm` command as simple as creating them.

3

Software installation simplified

This chapter covers:

- **Identifying software**
- **Finding and installing software with Docker Hub**
- **Installing software from alternative sources**
- **Understanding file system isolation**
- **How images and layers work**
- **Benefits of images with layers**

Chapters 1 and 2 introduce all-new concepts and abstractions provided by Docker. This chapter dives deeper into container file systems and software installation. It breaks down software installation into three steps, as illustrated in figure 3.1.

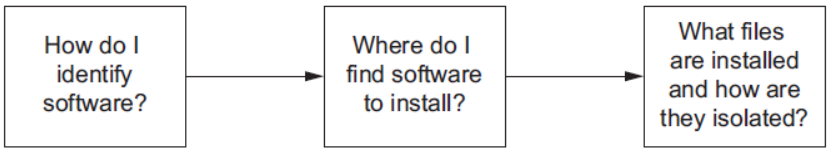


Figure 3.1 Flow of topics covered in this chapter

The first step in installing any software is identifying the software you want to install. You know that software is distributed using images, but you need to know how to tell Docker exactly which image you want to install. I've already mentioned that repositories hold images, but in this chapter I show how repositories and tags are used to identify images in order to install the software you want.

This chapter goes into detail on the three main ways to install Docker images:

- Docker registries
- Using image files with `docker save` and `docker load`
- Building images with Dockerfiles

In the course of reading this material you'll learn how Docker isolates installed software and you'll be exposed to a new term, *layer*. Layers are an important concept when dealing with images and provide multiple important features. This chapter closes with a section about how images work. That knowledge will help you evaluate the image quality and establish a baseline skillset for part 2 of this book.

3.1 Identifying software

Suppose you want to install a program called TotallyAwesomeBlog 2.0. How would you tell Docker what you wanted to install? You would need a way to name the program, specify the version that you want to use, and specify the source that you want to install it from. Learning how to identify specific software is the first step in software installation, as illustrated in figure 3.2.

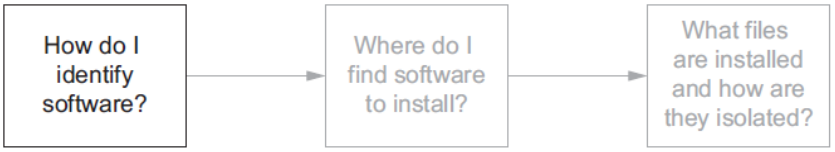


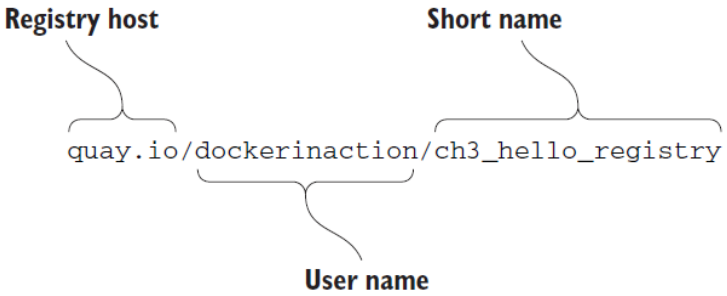
Figure 3.2 Step 1—Software identification

You've learned that Docker creates containers from images. An image is a file. It holds files that will be available to containers created from it and metadata about the image. This metadata contains labels, environment variables, default execution context, the command history for an image, and more.

Every image has a globally unique identifier. You can use that identifier with image and container commands, but in practice it's rare to actually work with raw image identifiers. They are long, unique sequences of letters and numbers. Each time a change is made to an image, the image identifier changes. Image identifiers are difficult to work with because they're unpredictable. Instead, users work with named repositories.

3.1.1 What is a named repository?

A *named repository* is a named bucket of images. The name is similar to a URL. A repository's name is made up of the name of the host where the image is located, the user account that owns the image, and a short name. For example, later in this chapter you will install an image from the repository named registry.dockerinaction.com/dockerinaction/ch3_hello_registry.



Just as there can be several versions of software, a repository can hold several images. Each of the images in a repository is identified uniquely with tags. If I were to release a new version of registry.dockerinaction.com/dockerinaction/ch3_hello_registry, I might tag it "v2" while tagging the old version with "v1." If you wanted to download the old version, you could specifically identify that image by its v1 tag.

In chapter 2 you installed an image from the NGINX repository on Docker Hub that was identified with the "latest" tag. A repository name and tag form a composite key, or a unique reference made up of a combination of non-unique components. In that example, the image was identified by nginx:latest. Although identifiers built in this fashion may occasionally be longer than raw image identifiers, they're predictable and communicate the intention of the image.

3.1.2 Using tags

Tags are both an important way to uniquely identify an image and a convenient way to create useful aliases. Whereas a tag can only be applied to a single image in a repository, a single image can have several tags. This allows repository owners to create useful versioning or feature tags.

For example, the Java repository on Docker Hub maintains the following tags: 7, 7-jdk, 7u71, 7u71-jdk, openjdk-7, and openjdk-7u71. All these tags are applied to the same image. But as the current minor version of Java 7 increases, and they release 7u72, the 7u71 tag will likely go away and be replaced with 7u72. If you care about what minor version of Java 7 you're running, you have to keep up with those tag changes. If you just want to make sure you're always running the most recent version of Java 7, just use the image tagged with 7. It will always be assigned to the newest minor revision of Java 7. These tags give users great flexibility.

It's also common to see different tags for images with different software configurations. For example, I've released two images for an open source program called freegeoip. It's a web application that can be used to get the rough geographical location associated with a network address. One image is configured to use the default configuration for the software. It's meant to run by itself with a direct link to the world. The second is configured to run behind a web

load balancer. Each image has a distinct tag that allows the user to easily identify the image with the features required.

> **TIP** When you're looking for software to install, always pay careful attention to the tags offered in a repository. If you're not sure which one you need, you can download all the tagged images in a repository by simply omitting the tag qualifier when you pull from the repository. I occasionally do this by accident, and it can be annoying. But it's easy to clean up.

This is all there is to identifying software for use with Docker. With this knowledge, you're ready to start looking for and installing software with Docker.

3.2 Finding and installing software

You can identify software by a repository name, but how do you find the repositories that you want to install? Discovering trustworthy software is complex, and it is the second step in learning how to install software with Docker, as shown in figure 3.3.

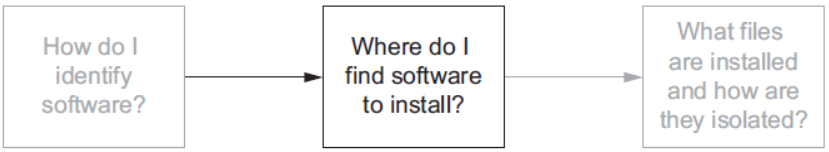


Figure 3.3 Step 2—Locating repositories

The easiest way to find images is to use an index. Indexes are search engines that catalog repositories. There are several public Docker indexes, but by default `docker` is integrated with an index named Docker Hub.

Docker Hub is a registry and index with a web user interface run by Docker Inc. It's the default registry and index used by `docker`. When you issue a `docker pull` or `docker run` command without specifying an alternative registry, Docker will default to looking for the repository on Docker Hub. Docker Hub makes Docker more useful out of the box.

Docker Inc. has made efforts to ensure that Docker is an open ecosystem. It publishes a public image to run your own registry, and the `docker` command-line tool can be easily configured to use alternative registries. Later in this chapter I cover alternative image installation and distribution tools included with Docker. But first, the next section covers how to use Docker Hub so you can get the most from the default toolset.

3.2.1 Working with Docker registries from the command line

There are two ways that an image author can publish their images to a registry like Docker Hub:

- *Use the command line to push images that they built independently and on their own systems.*
- *Make a Dockerfile publicly available and use a continuous build system to publish images.* Dockerfiles are scripts for building images. Images created from these automated builds are preferred because the Dockerfile is available for examination prior to installing the image.

Most registries will require image authors to authenticate before publishing and enforce authorizations checks on the repository they are updating. In these cases, you can use the `docker login` command to log in to specific registry servers like Docker Hub. Once you've logged in, you'll be able to pull from private repositories, push to any repository that you control, and tag images in your repositories. Chapter 7 covers pushing and tagging images.

Running `docker login` will prompt you for your Docker.com credentials. Once you've provided them, your command-line client will be authenticated, and you'll be able to access your private repositories. When you've finished working with your account, you can log out with the `docker logout` command. If you're using a different registry you can specify the server name as an argument to the `docker login` and `docker logout` subcommands.

3.2.2 Using alternative registries

Docker makes the registry software available for anyone to run. Cloud companies like AWS and Google offer private registries and companies that use Docker EE or who use the popular Artifactory project already have private registries. Running a registry from open source components is covered in chapter 8, but it's important that you learn how to use them early.

Using an alternative registry is simple. It requires no additional configuration. All you need is the address of the registry. The following command will download another "Hello World" type example from an alternative registry:

```
docker pull registry.dockerinaction.com/ch3/hello_registry:latest
```

The registry address is part of the full repository specification covered in section 3.1. The full pattern is as follows:

```
[REGISTRYHOST:PORT/][USERNAME/]NAME[:TAG]
```

Docker knows how to talk to Docker registries, so the only difference is that you specify the registry host. In some cases, working with registries will require an authentication step. If you encounter a situation where this is the case, consult the documentation or the group that configured the registry to find out more. When you're finished with the hello-registry image you installed, remove it with the following command:

```
docker rmi registry.dockerinaction.com/ch3/hello_registry
```

Registries are powerful. They enable a user to relinquish control of image storage and transportation. But running your own registry can be complicated and may create a potential

single point of failure for your deployment infrastructure. If running a custom registry sounds a bit complicated for your use case, and third-party distribution tools are out of the question, you might consider loading images directly from a file.

3.2.3 Images as files

Docker provides a command to load images into Docker from a file. With this tool, you can load images that you acquired through other channels. Maybe your company has chosen to distribute images through a central file server or some type of version-control system. Maybe the image is small enough that your friend just sent it to you over email or shared it via flash drive. However you came upon the file, you can load it into Docker with the `docker load` command.

You'll need an image file to load before I can show you the `docker load` command. Because it's unlikely that you have an image file lying around, I'll show you how to save one from a loaded image. For the purposes of this example, you'll pull `busybox:latest`. That image is small and easy to work with. To save that image to a file, use the `docker save` command. Figure 3.5 demonstrates `docker save` by creating a file from BusyBox.

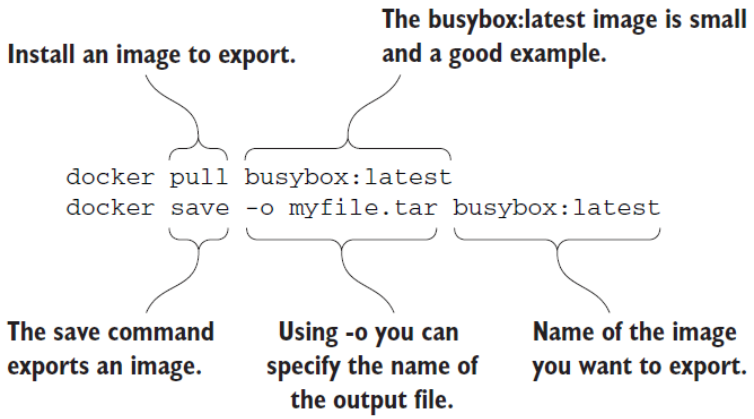


Figure 3.4 Parts of the pull and save subcommands

I used the .tar filename suffix in this example because the `docker save` command creates TAR archive files. You can use any filename you want. If you omit the `-o` flag, the resulting file will be streamed to the terminal.

> **TIP** Other ecosystems that use TAR archives for packing define custom file extensions. For example, Java uses .jar, .war, and .ear. In cases like these, using custom file extensions can help hint at the purpose and content of the archive. Although there are no defaults set by Docker and no official guidance on the matter, you may find using a custom extension useful if you work with these files often.

After running the `save` command, the `docker` program will terminate unceremoniously. Check that it worked by listing the contents of your current working directory. If the specified file is there, use this command to remove the image from Docker:

```
docker rmi busybox
```

After removing the image, load it again from the file you created using the `docker load` command. Like `docker save`, if you run `docker load` without the `-i` command, Docker will use the standard input stream instead of reading the archive from a file:

```
docker load –i myfile.tar
```

Once you've run the `docker load` command, the image should be loaded. You can verify this by running the `docker images` command again. If everything worked correctly, BusyBox should be included in the list.

Working with images as files is as easy as working with registries, but you miss out on all the nice distribution facilities that registries provide. If you want to build your own distribution tools, or you already have something else in place, it should be trivial to integrate with Docker using these commands.

Another popular project distribution pattern uses bundles of files with installation scripts. This approach is popular with open source projects that use public version-control repositories for distribution. In these cases you work with a file, but the file is not an image; it is a Dockerfile.

3.2.4 Installing from a Dockerfile

A Dockerfile is a script that describes steps for Docker to take to build a new image. These files are distributed along with software that the author wants to be put into an image. In this case, you're not technically installing an image. Instead, you're following instructions to build an image. Working with Dockerfiles is covered in depth in chapter 7.

Distributing a Dockerfile is similar to distributing image files. You're left to your own distribution mechanisms. A common pattern is to distribute a Dockerfile with software from common version-control systems like Git or Mercurial. If you have Git installed, you can try this by running an example from a public repository:

```
git clone https://github.com/dockerinaction/ch3_dockerfile.git
docker build -t dia_ch3/dockerfile:latest ch3_dockerfile
```

In this example you copy the project from a public source repository onto your computer and then build and install a Docker image using the Dockerfile included with that project. The value provided to the `-t` option of `docker build` is the repository where you want to install the image. Building images from Dockerfiles is a light way to move projects around that fits into existing workflows. There are two disadvantages to taking this approach. First, depending on the specifics of the project, the build process might take some time. Second, dependencies may drift between the time when the Dockerfile was authored and when an image is built on a

user's computer. These issues make distributing build files less than an ideal experience for a user. But it remains popular in spite of these drawbacks.

When you're finished with this example, make sure to clean up your workspace:

```
docker rmi dia_ch3/dockerfile
rm -rf ch3_dockerfile
```

3.2.5 Docker Hub from the website

If you have yet to stumble upon it while browsing docker.com, you should take a moment to check out https://hub.docker.com. Docker Hub lets you search for repositories, organizations, or specific users. User and organization profile pages list the repositories that the account maintains, recent activity on the account, and the repositories that the account has starred. On repository pages you can see the following:

- General information about the image provided by the image publisher
- A list of the tags available in the repository
- The date the repository was created
- The number of times it has been downloaded
- Comments from registered users

Docker Hub is free to join, and you'll need an account later in this book. When you're signed in, you can star and comment on repositories. You can create and manage your own repositories. We will do that in part 2. For now, just get a feel for the site and what it has to offer.

Activity: a Docker Hub scavenger hunt

It's good to practice finding software on Docker Hub using the skills you learned in chapter 2. This activity is designed to encourage you to use Docker Hub and practice creating containers. You will also be introduced to three new options on the `docker run` command.

In this activity you're going to create containers from two images that are available through Docker Hub. The first is available from the dockerinaction/ch3_ex2_hunt -repository. In that image you'll find a small program that prompts you for a password. You can only find the password by finding and running a container from the second mystery repository on Docker Hub. To use the programs in these images, you'll need to attach your terminal to the containers so that the input and output of your terminal are connected directly to the running container. The following command demonstrates how to do that and run a container that will be removed automatically when stopped:

```
docker run -it --rm dockerinaction/ch3_ex2_hunt
```

When you run this command, the scavenger hunt program will prompt you for the password. If you know the answer already, go ahead and enter it now. If not, just enter anything and it will give you a hint. At this point you should have all the tools you need to complete the activity. Figure 3.4 illustrates what you need to do from this point.

Still stuck? I can give you one more hint. The mystery repository is one that was created for this book. Maybe you should try searching for this book's Docker Hub repositories. Remember, repositories are named with a username/repository pattern.

When you get the answer, pat yourself on the back and remove the images using the `docker rmi` command. Concretely, the commands you run should look something like these:

```
docker rmi dockerinaction/ch3_ex2_hunt
docker rmi <mystery repository>
```

If you were following the examples and using the `--rm` option on your `docker run` commands, you should have no containers to clean up. You've learned a lot in this example. You've found a new image on Docker Hub and used the `docker run` command in a new way. There's a lot to know about running interactive containers. The next section covers that in greater detail.

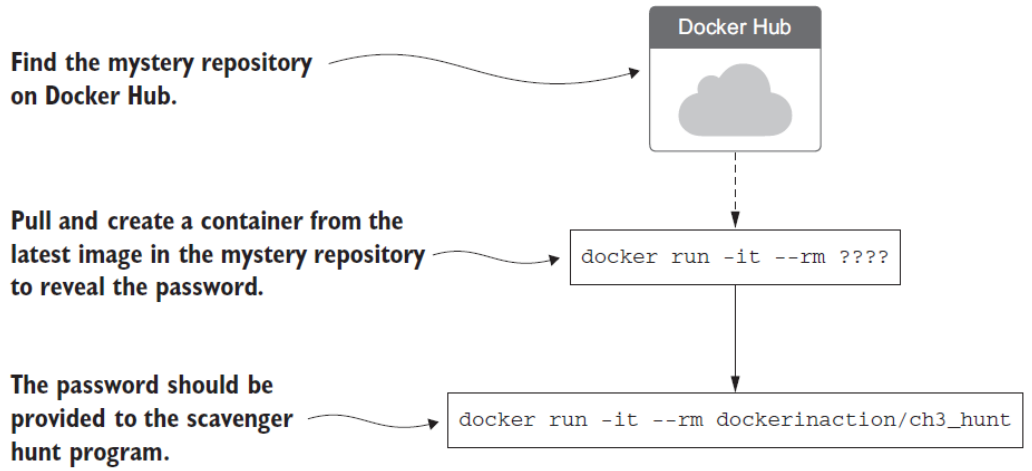


Figure 3.5 The steps required to complete the Docker Hub scavenger hunt. Find the mystery repository on Docker Hub. Install the latest image from that repository and run it interactively to get the password.

Docker Hub is by no means the only source for software. Depending on the goals and perspective of software publishers, Docker Hub may not be an appropriate distribution point. Closed source or proprietary projects may not want to risk publishing their software through a third party. There are three other ways to install software:

- You can use alternative repository registries or run your own registry.
- You can manually load images from a file.
- You can download a project from some other source and build an image using a provided Dockerfile.

All three of these options are viable for private projects or corporate infrastructure. The next few subsections cover how to install software from each alternative source. After reading this section you should have a complete picture of your options to install software with Docker. But

when you install software, you should have an idea about what changes are being made to your computer.

3.3 Installation files and isolation

Understanding how images are identified, discovered, and installed is a minimum proficiency for a Docker user. If you understand what files are actually installed and how those files are built and isolated at runtime, you'll be able to answer more difficult questions that come up with experience, such as these:

- What image properties factor into download and installation speeds?
- What are all these unnamed images that are listed when I use the `docker images` command?
- Why does output from the `docker pull` command include messages about pulling dependent layers?
- Where are the files that I wrote to my container's file system?

Learning this material is the third and final step to understanding software installation with Docker, as illustrated in figure 3.6.

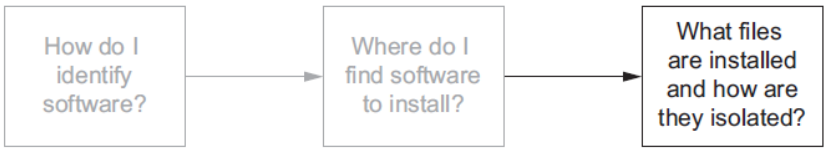


Figure 3.6 Step 3—Understanding how software is installed

So far, when I've written about installing software, I've used the term *image*. This was to infer that the software you were going to use was in a single image and that an image was contained within a single file. Although this may occasionally be accurate, most of the time what I've been calling an image is actually a collection of image layers. A *layer* is an image that's related to at least one other image. It is easier to understand layers when you see them in action.

3.3.1 Image layers in action

In this example you're going to install the two images. Both depend on Java 6. The applications themselves are simple Hello World–style programs. What I want you to keep an eye on is what Docker does when you install each. You should notice how long it takes to install the first compared to the second and read what it's printing to the terminal. When an image is being installed, you can watch Docker determine which dependencies it needs to download and then see the progress of the individual image layer downloads. Java is great for

this example because the layers are quite large, and that will give you a moment to really see Docker in action.

The two images you're going to install are `dockerinaction/ch3_myapp` and `dockerinaction/ch3_myotherapp`. You should just use the `docker pull` command because you only need to see the images install, not start a container from them. Here are the commands you should run:

```
docker pull dockerinaction/ch3_myapp
docker pull dockerinaction/ch3_myotherapp
```

Did you see it? Unless your network connection is far better than mine, or you had already installed Java 6 as a dependency of some other image, the download for - `dockerinaction/ch3_myapp` should have been much slower than `dockerinaction/ch3_myotherapp`.

When you installed `ch3_myapp`, Docker determined that it needed to install the openjdk-6 image because it's the direct dependency (parent layer) of the requested image. When Docker went to install that dependency, it discovered the dependencies of that layer and downloaded those first. Once all the dependencies of a layer are installed, that layer is installed. Finally, openjdk-6 was installed, and then the tiny `ch3_myapp` layer was installed.

When you issued the command to install `ch3_myotherapp`, Docker identified that openjdk-6 was already installed and immediately installed the image for `ch3_myotherapp`. This was simpler, and because less than one megabyte of data was transferred, it was faster. But again, to the user it was an identical process.

From the user perspective this ability is nice to have, but you wouldn't want to have to try to optimize for it. Just take the benefits where they happen to work out. From the perspective of a software or image author, this ability should play a major factor in your image design. I cover that more in chapter 7.

If you run `docker images` now, you'll see the following repositories listed:

- `dockerinaction/ch3_myapp`
- `dockerinaction/ch3_myotherapp`
- `java:6`

By default, the `docker images` command will only show you repositories. Similar to other commands, if you specify the `-a` flag, the list will include every installed intermediate image or layer. Running `docker images -a` will show a list that includes several repositories listed as `<none>`. The only way to refer to these is to use the value in the `IMAGE ID` column.

In this example you installed two images directly, but a third parent repository was installed as well. You'll need to clean up all three. You can do so more easily if you use the condensed `docker rmi` syntax:

```
docker rmi \
    dockerinaction/ch3_myapp \
    dockerinaction/ch3_myotherapp \
    java:6
```

The `docker rmi` command allows you to specify a space-separated list of images to be removed. This comes in handy when you need to remove a small set of images after an example. I'll be using this when appropriate throughout the rest of the examples in this book.

3.3.2 Layer relationships

Images maintain parent/child relationships. In these relationships they build from their parents and form layers. The files available to a container are the union of all of the layers in the lineage of the image the container was created from. Images can have relationships with any other image, including images in different repositories with different owners. The two images in section 3.3.1 use a Java 6 image as their parent. Figure 3.7 illustrates the full image ancestry of both images.

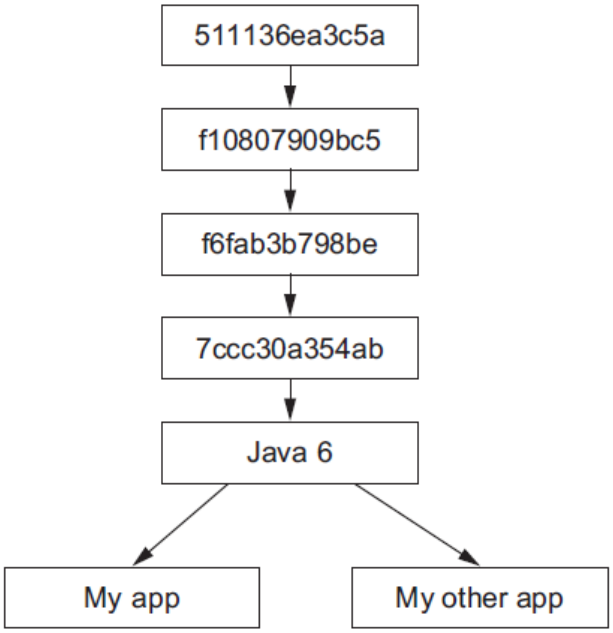


Figure 3.7 The full lineage of the two Docker images used in section 3.3.1

The layers shown in figure 3.7 are a sample of the java:6 image at the time of this writing. An image is named when its author tags and publishes it. A user can create aliases, as you did in chapter 2 using the `docker tag` command. Until an image is tagged, the only way to refer to it is to use its unique identifier (UID) that was generated when the image was built. In figure 3.7, the parents of the common Java 6 image are labeled using the first 12 digits of their UID. These layers contain common libraries and dependencies of the Java 6 software. Docker truncates the UID from 65 (base 16) digits to 12 for the benefit of its human users. Internally

and through API access, Docker uses the full 65. It's important to be aware of this when you've installed images along with similar unnamed images. I wouldn't want you to think something bad happened or some malicious software had made it into your computer when you see these images included when you use the `docker images` command.

The Java images are sizable. At the time of this writing, the openjdk-6 image is 348 MB, and the openjdk-7 image is 590 MB. You get some space savings when you use the runtime-only images, but even openjre-6 is 200 MB. Again, Java was chosen here because its images are particularly large for a common dependency.

3.3.3 Container file system abstraction and isolation

Programs running inside containers know nothing about image layers. From inside a container, the file system operates as though it's not running in a container or operating on an image. From the perspective of the container, it has exclusive copies of the files provided by the image. This is made possible with something called a union file system. Docker uses a variety of union file systems and will select the best fit for your system. The details of how the union file system works are beyond what you need to know to use Docker effectively.

A union file system is part of a critical set of tools that combine to create effective file system isolation. The other tools are MNT namespaces and the `chroot` system call.

The file system is used to create mount points on your host's file system that abstract the use of layers. The layers created are what are bundled into Docker image layers. Likewise, when a Docker image is installed, its layers are unpacked and appropriately configured for use by the specific file system provider chosen for your system.

The Linux kernel provides a namespace for the MNT system. When Docker creates a container, that new container will have its own MNT namespace, and a new mount point will be created for the container to the image.

Lastly, `chroot` is used to make the root of the image file system the root in the container's context. This prevents anything running inside the container from referencing any other part of the host file system.

Using `chroot` and MNT namespaces is common for container technologies. By adding a union file system to the recipe, Docker containers have several benefits.

3.3.4 Benefits of this toolset and file system structure

The first and perhaps most important benefit of this approach is that common layers need to be installed only once. If you install any number of images and they all depend on some common layer, that common layer and all of its parent layers will need to be downloaded or installed only once. This means you might be able to install several specializations of a program without storing redundant files on your computer or downloading redundant layers. By contrast, most virtual machine technologies will store the same files as many times as you have redundant virtual machines on a computer.

Second, layers provide a coarse tool for managing dependencies and separating concerns. This is especially handy for software authors, and chapter 7 talks more about this. From a user perspective, this benefit will help you quickly identify what software you're running by examining which images and layers you're using.

Lastly, it's easy to create software specializations when you can layer minor changes on top of some basic image. That's another subject covered in detail in chapter 7. Providing specialized images helps users get exactly what they need from software with minimal customization. This is one of the best reasons to use Docker.

3.3.5 Weaknesses of union file systems

Docker selects sensible defaults when it is started, but no implementation is perfect for every workload. In fact, there are some specific use cases when you should pause and consider using another Docker feature.

Different file systems have different rules about file attributes, sizes, names, and characters. Union file systems are in a position where they often need to translate between the rules of different file systems. In the best cases they're able to provide acceptable translations. In the worst cases features are omitted. For example, neither btrfs nor OverlayFS provides support for the extended attributes that make SELinux work.

Union file systems use a pattern called copy-on-write, and that makes implementing memory-mapped files (the `mmap()` system call) difficult. Some union file systems provide implementations that work under the right conditions, but it may be a better idea to avoid memory-mapping files from an image.

The backing file system is another pluggable feature of Docker. You can determine which file system your installation is using with the `info` subcommand. If you want to specifically tell Docker which file system to use, do so with the `--storage-driver` or `-s` option when you start the Docker daemon. Most issues that arise with writing to the union file system can be addressed without changing the storage provider. These can be solved with volumes, the subject of chapter 4.

3.4 Summary

The task of installing and managing software on a computer presents a unique set of challenges. This chapter explains how you can use Docker to address them. The core ideas and features covered by this chapter are as follows:

- Human Docker users use repository names to communicate which software they would like Docker to install.
- Docker Hub is the default Docker registry. You can find software on Docker Hub through either the website or the `docker` command-line program.
- The `docker` command-line program makes it simple to install software that's distributed through alternative registries or in other forms.
- The image repository specification includes a registry host field.

- The `docker load` and `docker save` commands can be used to load and save images from TAR archives.
- Distributing a Dockerfile with a project simplifies image builds on user machines.
- Images are usually related to other images in parent/child relationships. These relationships form layers. When we say that we have installed an image, we are saying that we have installed a target image and each image layer in its lineage.
- Structuring images with layers enables layer reuse and saves bandwidth during distribution and storage space on your computer.

4

Working with storage and volumes

This chapter covers:

- Introducing mount points
- How to share data between the host and a container
- How to share data between containers
- Manipulating container mount points
- Bind mounts
- Temporary file system mounts
- Managing data with volumes
- Advanced storage with volume plugins

At this point in the book, you've installed and run a few programs. You've seen a few toy examples but haven't run anything that resembles the real world. The difference between the examples in the first three chapters and the real world is that in the real world, programs work with data. This chapter introduces Docker volumes and strategies that you'll use to manage data with containers.

Consider what it might look like to run a database program inside a container. You would package the software with the image, and when you start the container it might initialize an empty database. When programs connect to the database and enter data, where is that data stored? Is it in a file inside the container? What happens to that data when you stop the container or remove it? How would you move your data if you wanted to upgrade the database program? What happens to that storage on a cloud machine when it is terminated?

Consider another situation where you're running a couple of different web applications inside different containers. Where would you write log files so that they will outlive the

container? How would you get access to those logs to troubleshoot a problem? How can other programs such as log digest tools get access to those files?

The union file system is not appropriate for working with long-lived data or sharing data between containers, or a container and the host. The answer to all these questions involves managing the container file system and mount points.

4.1 File trees and mount points

Unlike other operating systems Linux unifies all storage into a single tree. Storage devices like disk partitions, or USB disk partitions are attached to specific location in that tree. Those locations are called mount points. A mount point defines the location in the tree, the access properties (like writability) to the data at that point, and the source of the data mounted at that point (like a specific hard disk, USB device, or memory backed virtual disk).

Mount points allow software and users to use the file tree in a Linux environment without knowing exactly how that tree is mapped into specific storage devices. This is particularly useful in container environments.

Every container has something called a MNT namespace and a unique file tree root. This is discussed in detail in chapter 6. For now it is enough to understand that the image that a container is created from is mounted at that container's file tree root or at the "/" point and that every container has a different set of mount points.

Logic follows that if different storage devices can be mounted at various points in a file tree then we can mount non-image related storage at other points in a container file tree. That is exactly how containers get access to storage on the host file system and share storage between containers.

The rest of this chapter elaborates on how to manage storage and the mount points in containers. The best place to start is by understanding the three most common types of storage mounted into containers:

1. Bind mounts
2. In-memory storage
3. Docker volumes

All three types of mount points can be created using the `--mount` flag on the `docker run` and `docker create` subcommands.

4.2 Bind mounts

Mount points can be used to re-mount parts of the file tree onto other locations. That is something called a bind mount point. When working with containers bind mounts attach a user-specified location on the host file system to a specific point in a container file tree. Bind mounts are useful when the host provides some file or directory that is needed by a program running in a container, or when that containerized program produces some file or log that is processed by users or programs running outside of containers.

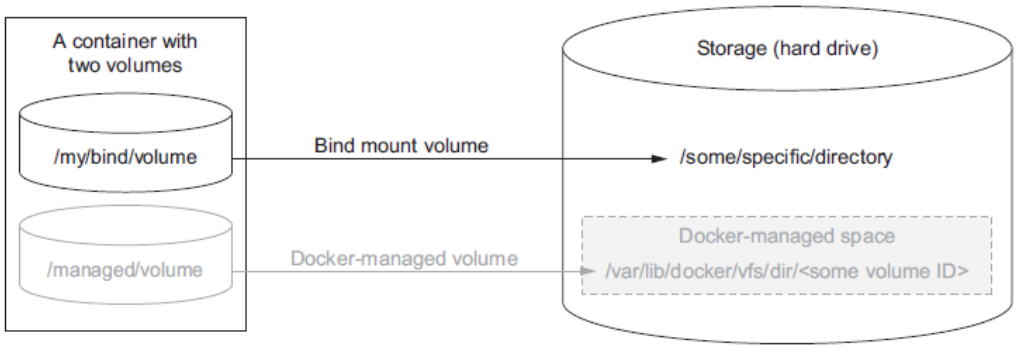


Figure 4.2 A host directory as a bind mount volume

Consider the example shown in figure 4.2. Suppose you're running a web server that depends on sensitive configuration on the host and also emits access logs that need to be forwarded by your log shipping system. You could use Docker to launch the web server in a container and bind mount the configuration location as well as the location where you want the web server to write logs.

You can try this for yourself. Create a placeholder log file and create a special nginx configuration file named `example.conf`. Run the following commands to create and populate the files:

```
touch ~/example.log
cat >~/example.conf <<EOF
server {
  listen 80;
  server_name localhost;
  access_log /var/log/nginx/custom.host.access.log main;
  location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
  }
}
EOF
```

Once a server is started with this configuration file it will offer the nginx default site at http://localhost/ and access logs for that site will be written to a file in the container at /var/log/nginx/custom.host.access.log. The following command will start an nginx HTTP server in a container where your new configuration is bind mounted to the server's configuration root:

```
CONF_SRC=~/example.conf; \
      CONF_DST=/etc/nginx/conf.d/default.conf; \
LOG_SRC=~/example.log; \
      LOG_DST=/var/log/nginx/custom.host.access.log; \
      docker run -d --name diaweb \
  --mount type=bind,src=${CONF_SRC},dst=${CONF_DST} \
  --mount type=bind,src=${LOG_SRC},dst=${LOG_DST} \
  -p 80:80 \
  nginx:latest
```

With this container running, you should be able to point your web browser at http://localhost/ and see the nginx hello-world page, and you will not see any access logs in the container log stream: `docker logs diaweb`. However you will be able to see those logs if you examine the `example.log` file in your home directory: `cat ~/example.log`.

In this example you used the `--mount` option with the `type=bind` option. The other two mount parameters, src and dst define the source location on the host file tree and the destination location on the container file tree. You must specify locations with absolute paths, but in this example we used shell expansion and shell variables to make the command easier to read.

This example touches on an important attribute or feature of volumes. When you mount a volume on a container file system, it replaces the content that the image provides at that location. By default the nginx:latest image provides some default configuration at /etc/nginx/conf.d/default.conf, but when you created the bind mount with a destination at that path, the content provided by the image was overridden by the content on the host. This behavior is the basis for the polymorphic container pattern discussed later in the chapter.

Expanding on this use case, suppose you want to make sure that the nginx web server can't change the contents of the configuration volume. Even the most trusted software can contain vulnerabilities, and it's best to minimize the impact of an attack on your website. Fortunately, Linux provides a mechanism to make mount points read-only. You can do this by adding the readonly=true argument to the mount specification. In the example, you should change the `run` command to something like the following:

```
docker rm -f diaweb

CONF_SRC=~/example.conf; \
      CONF_DST=/etc/nginx/conf.d/default.conf; \
LOG_SRC=~/example.log; \
      LOG_DST=/var/log/nginx/custom.host.access.log; \
      docker run -d --name diaweb \
  --mount type=bind,src=${CONF_SRC},dst=${CONF_DST},readonly=true \ (1)
  --mount type=bind,src=${LOG_SRC},dst=${LOG_DST} \
```

```
  -p 80:80 \
  nginx:latest
```

❶ Note the readonly flag

By creating the read-only mount you can prevent any process inside the container from modifying the content of the volume. You can see this in action by running a quick test:

```
docker exec diaweb \
    sed -i "s/listen 80/listen 8080/" /etc/nginx/conf.d/default.conf
```

This command executes a sed command inside the diaweb container and attempts to modify the configuration file. The command fails because the file is mounted as read-only.

The first problem with bind mounts is that they tie otherwise portable container descriptions to the file system of a specific host. If a container description depends on content at a specific location on the host file system, then that description isn't portable to hosts where the content is unavailable or available in some other location.

The next big problem is that they create an opportunity for conflict with other containers. It would be a bad idea to start multiple instances of Cassandra that all use the same host location as a bind mount for data storage. In that case, each of the instances would compete for the same set of files. Without other tools such as file locks, that would likely result in corruption of the database.

Bind mounts are appropriate tools for workstations, machines with specialized concerns, or in systems combined with more traditional configuration management tooling. It's better to avoid these kinds of specific bindings in generalized platforms or hardware pools.

4.3 In-memory storage

Most service software and web applications have use private key files, database passwords, API key files or other sensitive configuration files, and need upload buffering space. In these cases it is important that you never include those type of files in an image or write them to disk. Instead you should use in-memory storage. You can add in-memory storage to containers with a special type of mount.

Set the type option on the mount flag to `tmpfs`. This is the easiest way to mount memory based files system into a container's file tree. Consider the command:

```
docker run --rm \
    --mount type=tmpfs,dst=/tmp \
    --entrypoint mount \
    alpine:latest -v
```

This command creates an empty tmpfs device and attaches it to the new container's file tree at `/tmp`. Any files created under this file tree will be written to memory instead of disk. More than that, the mount point is created with sensible defaults for generic workloads. Running the command will display a list of all the mount points for the container. The list will include the following line:

```
tmpfs on /tmp type tmpfs (rw,nosuid,nodev,noexec,relatime)
```

This line describes the mount point configuration. From left-to-right it indicates that:

1. a tmpfs device is mounted to the tree at /tmp
2. that the device has a tmpfs file system
3. the tree is read/write capable
4. suid bits will be ignored on all files in this tree
5. no files in this tree will be interpreted as special devices
6. no files in this tree will be executable
7. file access times will be updated if they are older than the current modify or change time

Additionally, the tmpfs device will not have any size limits by default and will be world-writeable (has file permissions 1777 in octal). You can add a size limit and change the file mode with two additional options: tmpfs-size, and tmpfs-mode.

```
docker run --rm \
    --mount type=tmpfs,dst=/tmp,tmpfs-size=16k,tmpfs-mode=1770 \
    --entrypoint mount \
    alpine:latest -v
```

This command limits the tmpfs device mounted at /tmp to 16 kilobytes and is not readable by "other" in-container users.

4.4 Docker volumes

Docker volumes are named file system trees managed by Docker. They can be implemented with disk storage on the host file system, or another more exotic backend like cloud storage. All operations on Docker volumes can be accomplished using the `docker volume` subcommand set. Using volumes is a method of decoupling storage from specialized locations on the file system that you might specify with bind mounts.

You can create and inspect volumes using the `docker volume create` and `docker volume inspect` subcommands. By default Docker creates volumes using the `local` volume plugin. The default behavior will create a directory to store the contents of a volume somewhere in a part of the host file system under control of the Docker engine. For example the following two commands will create a volume named, "location-example" and display the location of the volume host file system tree:

```
docker volume create \                  ❶
    --driver local \                    ❷
    --label example=location \          ❸
    location-example
docker volume inspect \                 ❹
    --format "{{json .Mountpoint}}" \   ❺
    location-example
```

❶ **create the volume**

2. specify the “local” plugin
3. add a volume label
4. inspect the volume
5. select the location on the host

Docker volumes may seem difficult to work with if you’re manually building or linking tools together on your desktop, but in larger systems where specific locality of the data is less important, volumes are a much more effective way to organize your data. Using them decouples volumes from other potential concerns of the system. By using Docker volumes, you’re simply stating, “I need a place to put some data that I'm working with.” This is a requirement that Docker can fill on any machine with Docker installed. Further, when you’re finished with a volume and you ask Docker to clean things up for you, Docker can confidently remove any directories or files that are no longer being used by a container. Using volumes in this way helps manage clutter. As Docker middleware or plugins evolve, volume users will be able to adopt more advanced features.

Sharing access to data is a key feature of volumes. If you have decoupled volumes from known locations on the file system, you need to know how to share volumes between containers without exposing the exact location of managed containers. The next section describes two ways to share data between containers using volumes.

4.4.1 Volumes provide container-independent data management

Semantically, a volume is a tool for segmenting and sharing data that has a scope or life cycle that’s independent of a single container. That makes volumes an important part of any containerized system design that shares or writes files. Examples of data that differs in scope or access from a container include the following:

- Database software versus database data
- Web application versus log data
- Data processing application versus input and output data
- Web server versus static content
- Products versus support tools

Volumes enable separation of concerns and create modularity for architectural components. That modularity helps you understand, build, support, and reuse parts of larger systems more easily.

Think about it this way: images are appropriate for packaging and distributing relatively static files like programs; volumes hold dynamic data or specializations. This distinction makes images reusable and data simple to share. This separation of relatively static and dynamic file space allows application or image authors to implement advanced patterns such as polymorphic and composable tools.

A *polymorphic* tool is one that maintains a consistent interface but might have several implementations that do different things. Consider an application such as a general application server. Apache Tomcat, for example, is an application that provides an HTTP interface on a

network and dispatches any requests it receives to pluggable programs. Tomcat has polymorphic behavior. Using volumes, you can inject behavior into containers without modifying an image. Alternatively, consider a database program like MongoDB or MySQL. The value of a database is defined by the data it contains. A database program always presents the same interface but takes on a wholly different value depending on the data that can be injected with a volume. The polymorphic container pattern is the subject of section 4.5.3.

More fundamentally, volumes enable the separation of application and host concerns. At some point an image will be loaded onto a host and a container created from it. Docker knows little about the host where it's running and can only make assertions about what files should be available to a container. That means Docker alone has no way to take advantage of host-specific facilities like mounted network storage or mixed spinning and solid-state hard drives. But a user with knowledge of the host can use volumes to map directories in a container to appropriate storage on that host.

Now that you're familiar with what volumes are and why they're important, you can get started with them in a real-world example.

4.4.2 Using volumes with a NoSQL database

The Apache Cassandra project provides a column database with built-in clustering, eventual consistency, and linear write scalability. It's a popular choice in modern system designs, and an official image is available on Docker Hub. Cassandra is like other databases in that it stores its data in files on disk. In this section you'll use the official Cassandra image to create a single-node Cassandra cluster, create a keyspace, delete the container, and then recover that keyspace on a new node in another container.

Get started by creating the volume that will store the Cassandra database files. This volume uses disk space on the local machine and in a part of the file system managed by the docker engine.

```
docker volume create \
    --driver local \
    --label example=cassandra \
    cass-shared
```

This volume is not associated with any containers, it is just a named bit of disk that can be accessed by containers. The volume you just created is named, "cass-shared." In this case you added a label to the volume with key, "example" and value, "cassandra." Adding label metadata to your volumes can help you organize and cleanup volumes later. You're going to use this volume when you create a new container running Cassandra:

```
docker run -d \
    --mount cass-shared:/var/lib/cassandra/data \        (1)
    --name cass1 \
    cassandra:2.2
```

(1) **mount the volume into the container**

After Docker pulls the cassandra:2.2 image from Docker Hub, it creates a new container with the `cass-shared` volume mounted at `/var/lib/cassandra/data`. Next, start a container from the cassandra:2.2 image, but run a Cassandra client tool and connect to your running server:

```
docker run –it --rm \
    --link cass1:cass \
    cassandra:2.2 cqlsh cass
```

Now you can inspect or modify your Cassandra database from the CQLSH command line. First, look for a keyspace named docker_hello_world:

```
select *
from system.schema_keyspaces
where keyspace_name = 'docker_hello_world';
```

Cassandra should return an empty list. This means the database hasn't been modified by the example. Next, create that keyspace with the following command:

```
create keyspace docker_hello_world
with replication = {
    'class' : 'SimpleStrategy',
    'replication_factor': 1
};
```

Now that you've modified the database, you should be able to issue the same query again to see the results and verify that your changes were accepted. The following command is the same as the one you ran earlier:

```
select *
from system.schema_keyspaces
where keyspace_name = 'docker_hello_world';
```

This time Cassandra should return a single entry with the properties you specified when you created the keyspace. If you're satisfied that you've connected to and modified your Cassandra node, quit the CQLSH program to stop the client container:

```
# Leave and stop the current container
quit
```

The client container was created with the `--rm` flag and was automatically removed when the command stopped. Continue cleaning up the first part of this example by stopping and removing the Cassandra node you created:

```
docker stop cass1
docker rm -vf cass1
```

Both the Cassandra client and server you created will be deleted after running those commands. If the modifications you made are persisted, the only place they could remain is the `cass-shared` volume. You can test this by repeating these steps. Create a new Cassandra node, attach a client, and query for the keyspace. Figure 4.2 illustrates the system and what you will have built.

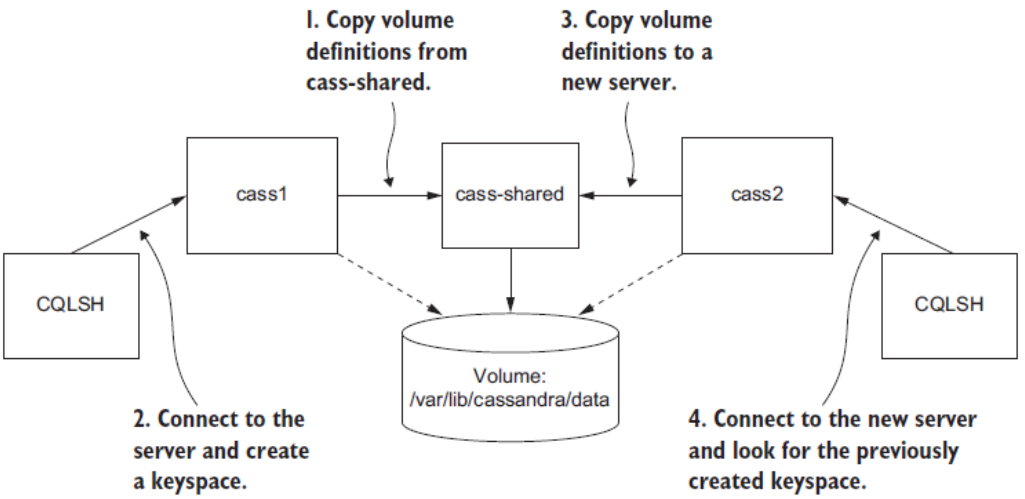


Figure 4.2 Key steps in creating and recovering data persisted to a volume with Cassandra

The next three commands will test recovery of the data:

```
docker run -d \
    --mount cass-shared:/var/lib/cassandra/data \
    --name cass2 \
    cassandra:2.2

docker run –it --rm \
    --link cass2:cass \
    cassandra:2.2 \
    cqlsh cass

select *
from system.schema_keyspaces
where keyspace_name = 'docker_hello_world';
```

The last command in this set returns a single entry, and it matches the keyspace you created in the previous container. This confirms the previous claims and demonstrates how volumes might be used to create durable systems. Before moving on, quit the CQLSH program and clean up your workspace. Make sure to remove that volume container as well:

```
quit

docker rm -vf cass2 cass-shared
```

This example demonstrates one way to use volumes without going into how they work, the patterns in use, or how to manage volume life cycle. The remainder of this chapter dives deeper into each facet of volumes and, starting with the different types available.

4.5 Shared mount points and sharing files

Sharing access to the same set of files between multiple containers is where the value of volumes becomes most obvious. Compare the bind mount and volume based approaches.

Bind mounts are the most obvious way to share some disk space between containers. You can see it in action in the following example:

```
LOG_SRC=~/web-logs-example
mkdir ${LOG_SRC}                                          (1)

docker run --name plath -d \                              (2)
    --mount type=bind,src=${LOG_SRC},dst=/data \          (2)
    registry.dockerinaction.com/ch4/writer_a              (2)

docker run --rm \
    --mount type=bind,src=${LOG_SRC},dst=/data \          (3)
    alpine:latest \                                       (3)
    head /data/logA                                       (3)

cat ${LOG_SRC}/logA                                       (4)

docker rm -f plath                                        (5)
```

(1) Set up a known location
(2) Bind mount the location into a log-writing container
(3) Bind mount the same location into a container for reading
(4) View the logs from the host
(5) Stop the writer

In this example you created two containers: one named plath that writes lines to a file and another that views the top part of the file. These containers share a common bind mount definition. Outside any container you can see the changes by listing the contents of the directory you created or viewing the new file. The major drawback to this approach is that all containers involved must agree on the exact location on the host file path, and they may conflict with other containers that also intend to read or manipulate files at that location.

Now compare that bind mount example with an example that uses volumes. The following commands are equivalent to the prior example but have no host-specific dependencies.

```
docker volume create \                                    (1)
    --driver local \                                      (1)
    logging-example                                       (1)

docker run --name plath -d \                              (2)
    --mount type=volume,src=logging-example,dst=/data \   (2)
    registry.dockerinaction.com/ch4/writer_a              (2)

docker run --rm \
    --mount type=bind,src=logging-example,dst=/data \     (3)
    alpine:latest \                                       (3)
    head /data/logA                                       (3)
```

```
cat "$(docker volume inspect \
    --format "{{json .Mountpoint}}" logging-example)"/logA    4

docker stop plath                                             5
```

1. Set up a named volume
2. Mount the volume into a log-writing container
3. Mount the same volume into a container for reading
4. View the logs from the host
5. Stop the writer

Unlike shared based on bind mounts, named volumes enable containers to share files without any knowledge of the underlying host file system. Unless the volume needs to use some specific settings or plugin it does not have to exist before the first container mounts it. Docker will automatically create volumes named in run or create commands using the defaults. However, it is important to remember that a that volume does exist on the host will be reused and shared by any other containers with the same volume dependency.

This name conflict problem can be solved using anonymous volumes and mount point definition inheritance between containers.

4.5.1 Anonymous volumes and the volumes-from flag

An anonymous volume is created without a name either before use with the `docker volume create` subcommand, or just-in-time with defaults using a `docker run` or `docker create` command. When the volume is created it will be assigned a unique identifier like, "1b3364a8debb5f653d1ecb9b190000622549ee2f812a4fb4ec8a83c43d87531b" instead of a human friendly name. These are more difficult to work with if you are stitching together dependencies manually, but they are useful when you need to eliminate potential volume naming conflicts. The docker command line provides another way to specify mount dependencies instead of referencing volumes by name.

The `docker run` command provides a flag, `--volumes-from` that will copy the mount definitions from one or more containers to the new container. It can be set multiple times to specify multiple source containers. By combining this flag and anonymous volumes you can build rich shared-state relationships in a host-independent way. Consider the following example:

```
docker run --name fowler \
    --mount type=volume,dst=/library/PoEAA \
    --mount type=bind,src=/tmp,dst=/library/DSL \
    alpine:latest \
    echo "Fowler collection created."
docker run --name knuth \
    --mount type=volume,dst=/library/TAoCP.vol1 \
    --mount type=volume,dst=/library/TAoCP.vol2 \
    --mount type=volume,dst=/library/TAoCP.vol3 \
    --mount type=volume,dst=/library/TAoCP.vol4.a \
    alpine:latest \
    echo "Knuth collection created"
```

```
docker run --name reader \                          (1)
    --volumes-from fowler \                         (1)
    --volumes-from knuth \                          (1)
    alpine:latest ls -l /library/

docker inspect --format "{{json .Mounts}}" reader   (2)
```

(1) List all volumes as they were copied into new container
(2) Checkout volume list for reader

In this example you created two containers that defined anonymous volumes as well as a bind mount volume. To share these with a third container without the `--volumes-from` flag, you'd need to inspect the previously created containers and then craft bind mount volumes to the Docker-managed host directories. Docker does all this on your behalf when you use the `--volumes-from` flag. It copies all mount point definitions present on a referenced source container into the new container. In this case, the container named reader copied all the mount points defined by both fowler and knuth.

You can copy volumes directly or transitively. This means that if you're copying the volumes from another container, you'll also copy the volumes that it copied from some other container. Using the containers created in the last example yields the -following:

```
docker run --name aggregator \              (1)
    --volumes-from fowler \
    --volumes-from knuth \
    alpine:latest \
    echo "Collection Created."

docker run --rm \                           (2)
    --volumes-from aggregator \
    alpine:latest \
    ls -l /library/
```

(1) Create an aggregation
(2) Consume volumes from a single source and list them

Copied volumes always have the same mount point. That means that you can't use `--volumes-from` in three situations.

In the first situation, you can't use `--volumes-from` if the container you're building needs a shared volume mounted to a different location. It offers no tooling for remapping mount points. It will only copy and union the mount points specified by the specified containers. For example, if the student in the last example wanted to mount the library to a location like /school/library, they wouldn't be able to do so.

The second situation occurs when the volume sources conflict with each other or a new volume specification. If one or more sources create a managed volume with the same mount point, then a consumer of both will receive only one of the volume -definitions:

```
docker run --name chomsky --volume /library/ss \
    alpine:latest echo "Chomsky collection created."
```

```
docker run --name lamport --volume /library/ss \
    alpine:latest echo "Lamport collection created."

docker run --name student \
    --volumes-from chomsky --volumes-from lamport \
    alpine:latest ls -l /library/

docker inspect -f "{{json .Mounts}}" student
```

When you run the example, the output of `docker inspect` will show that the last container has only a single volume listed at /library/ss and its value is the same as one of the other two. Each source container defines the same mount point, and you create a race condition by copying both to the new container. Only one of the two copy operations can succeed.

A real-world example where this would be limiting is if you were copying the volumes of several web servers into a single container for inspection. If those servers are all running the same software or share common configuration (which is more likely than not in a containerized system), then all those servers might use the same mount points. In that case, the mount points would conflict, and you'd be able to access only a subset of the required data.

The third situation where you can't use `--volumes-from` is if you need to change the write permission of a volume. This is because `--volumes-from` copies the full volumes definition. For example, if your source has a volume mounted with read/write access, and you want to share that with a container that should have only read access, using `--volumes-from` won't work.

Sharing volumes with the `--volumes-from` flag is an important tool for building portable application architectures, but it does introduce some limitations. The more challenging of which are in managing file permissions.

Using volumes decouples containers from the data and file system structure of the host machine, and that's critical for most production environments. The files and directories that Docker creates for managed volumes still need to be accounted for and maintained. The next section will show you how to keep a Docker environment clean.

4.6 Cleaning up volumes

By this point in the chapter you should have quite a few containers and volumes to clean up. You can see all of the volumes present on your system by running the `docker volume list` subcommand. The output will list the type and name of each volume. Any volumes that were created with a name will be listed with that name. Any anonymous volumes will be listed by their identifier.

Anonymous volumes can be cleaned up two ways. First, anonymous volumes are automatically deleted when the container they were created for are automatically cleaned up. This happens when containers are deleted via the `docker run --rm` or `docker rm -v` flags. Second, they can be manually deleted by issuing a `docker volume remove` command.

```
docker volume create --driver=local
```

```
# Outputs:
# 462d0bb7970e47512cd5ebbbb283ed53d5f674b9069b013019ff18ccee37d75d

docker volume remove \
    462d0bb7970e47512cd5ebbbb283ed53d5f674b9069b013019ff18ccee37d75d
# Outputs:
# 462d0bb7970e47512cd5ebbbb283ed53d5f674b9069b013019ff18ccee37d75d
```

Unlike anonymous volumes, named volumes must always be deleted manually. This behavior can be helpful in cases where containers are running jobs that collect partitioned or periodic data. Consider the following:

```
for i in amazon google microsoft; \
do \
docker run --rm \
    --mount type=volume,src=$i,dst=/tmp \
    --entrypoint /bin/sh \
    alpine:latest -c "nslookup $i.com > /tmp/results.txt"; \
done
```

This command performs a DNS lookup on amazon.com, google.com, and microsoft.com in three separate containers and records the results in three different volumes. Those volumes are named amazon, google, and microsoft. Even though the containers are being cleaned up automatically the named volumes will remain. If you ran the command you should be able to see the new volumes when you run `docker volume list`.

The only way to delete these named volumes is by specifying their name in the `docker volume remove` command.

```
docker volume remove \
    amazon google microsoft
```

The remove subcommand supports a list argument for specifying volume names and identifiers. The command above will delete all three named volumes.

There is only one constraint for deleting volumes. No volume that is currently in use can be deleted. Concretely, no volume attached to any container in any state can be deleted. If you attempt to do so the docker command will respond with a message stating, "volume is in use" and display the identifier of the container using the volume.

It can be annoying to determine which volumes are candidates for removal if you simply want to remove all or some of volumes that can be removed. This happens all the time as part of periodic maintenance. The docker volume prune command is built for this case.

Running docker volume prune with no options will prompt you for confirmation and delete all volumes that can be deleted. You can filter that candidate set by providing volume labels:

```
docker volume prune --filter example=cassandra
```

This command would prompt for confirmation and delete the volume you created earlier in the Cassandra example. If you're automating these cleanup procedure then you might want to suppress the confirmation step. In that case use the `--force` or `-f` flag:

```
docker volume prune --filter example=location --force
```

Understanding volumes is critical for working with real-world containers, but in many cases using volumes on local disk can create hard problems. If you're running software on a cluster of machines then the data that software stores in volumes will stay on the disk where it was written. If a container is moved to a different machine then it will lose access to its data in an old volume. You can solve this problem for your organization with volume plugins.

4.7 Advanced storage with volume plugins

Docker provides a volume plugin interface as a means for the community to extend the default engine capabilities. That has been implemented by several storage management tools, and today users are free to leverage all types of backing storage including proprietary cloud block storage, network file system mounts, specialized storage hardware, on-prem cloud solutions like Ceph and vSphere storage.

These community and vendor plugins will help you solve the hard problems associated with writing files to disk on one machine and depending on them from another. They are simple to install, configure, and remove using the appropriate `docker plugin` subcommands.

Docker plugins are not covered in detail in this text. They are always environment specific and difficult to demonstrate without using paid resources or endorsing specific cloud providers. Choosing a plugin depends on the storage infrastructure you want to integrate with, however there are a few projects that simplify this to some degree. Rexray (https://github.com/rexray/rexray) is a popular open source project that provides volumes on several cloud and on-prem storage platforms. If you've come to the point in your container journey that you need more sophisticated volume backends you should look at the latest offerings on Docker Hub and look into the current status of Rexray.

4.8 Summary

One of the first major hurdles in learning how to use Docker is understanding how to work with files that are not part of images and might be shared with other containers or the host. This chapter covers mount points depth, including the following:

- Mount points allow many file systems from many devices to be attached to a single file tree. Every container has its own file tree.
- Containers can use bind mounts to attach parts of the host file system into a container.
- In-memory file systems can be attached to a container file tree so sensitive or temporary data is not written to disk.
- Docker provides anonymous or named storage references called volumes.
- Volumes can be created, listed, and deleted using the appropriate `docker volume` subcommand.
- Volumes are parts of the host file system that Docker mounts into containers at specified locations.

- Volumes have life cycles of their own and might need to be periodically cleaned up.
- Docker can provide volumes backed by network storage or other more sophisticated tools if the appropriate volume plugin is installed.

5

Single Host Networking

This chapter covers

- Networking background
- Creating Docker container networks
- Network-less and host-mode containers
- Publishing services on the ingress network
- Container network caveats

Networking is a whole field of computing and so this chapter can only scratch the surface by covering specific challenges, structure, and tools required for container networks. If you want to run a website, database, email server, or any software that depends on networking, like a web browser inside a Docker container, then you need to understand how to connect that container to the network. After reading this chapter you'll be able to create containers with network exposure appropriate for the application you're running, use network software in one container from another, and understand how containers interact with the host and the host's network.

5.1 Networking background (for beginners)

A quick overview of relevant networking concepts will be helpful for understanding the topics in this chapter. This section includes only high-level detail; so if you're an expert, feel free to skip ahead.

Networking is all about communicating between processes that may or may not share the same local resources. To understand the material in this chapter you only need to consider a few basic network abstractions that are commonly used by processes. The better understanding you have of networking, the more you'll learn about the mechanics at work. But

a deep understanding isn't required to use the tools provided by Docker. If anything, the material contained herein should prompt you to independently research selected topics as they come up. Those basic abstractions used by processes include protocols, network interfaces, and ports.

5.1.1 Basics: protocols, interfaces, and ports

A *protocol* with respect to communication and networking is a sort of language. Two parties that agree on a protocol can understand what each other is communicating. This is key to effective communication. Hypertext Transfer Protocol (HTTP) is one popular network protocol that many people have heard of. It's the protocol that provides the World Wide Web. A huge number of network protocols and several layers of communication are created by those protocols. For now, it's only important that you know what a protocol is so that you can understand network interfaces and ports.

A network *interface* has an address and represents a location. You can think of interfaces as analogous to real-world locations with addresses. A network interface is like a mailbox. Messages are delivered to a mailbox for recipients at that address, and messages are taken from a mailbox to be delivered elsewhere.

Whereas a mailbox has a postal address, a network interface has an *IP address*, which is defined by the Internet Protocol. The details of IP are interesting but outside of the scope of this book. The important thing to know about IP addresses is that they are unique in their network and contain information about their location on their network.

It's common for computers to have two kinds of *interfaces*: an Ethernet interface and a loopback interface. An Ethernet interface is what you're likely most familiar with. It's used to connect to other interfaces and processes. A loopback interface isn't connected to any other interface. At first this might seem useless, but it's often useful to be able to use network protocols to communicate with other programs on the same computer. In those cases a loopback is a great solution.

In keeping with the mailbox metaphor, a *port* is like a recipient or a sender. There might be several people who receive messages at a single address. For example, a single address might receive messages for Wendy Webserver, Deborah Database, and Casey Cache, as illustrated in figure 5.1. Each recipient should only open his or her own messages.

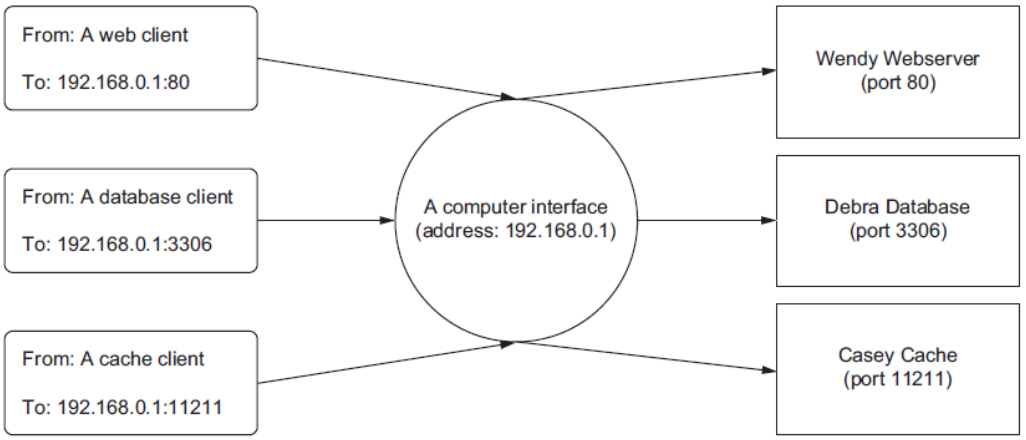


Figure 5.1 Processes use the same interface and are uniquely identified in the same way multiple people might use the same mailbox.

In reality, ports are just numbers and defined as part of the Transmission Control Protocol (TCP). Again the details of the protocol are beyond the scope of this book, but I encourage you to read about it some time. People who created standards for protocols, or companies that own a particular product, decide what port number should be used for specific purposes. For example, web servers provide HTTP on port 80 by default. MySQL, a database product, serves its protocol on port 3306 by default. Memcached, a fast cache technology, provides its protocol on port 11211. Ports are written on TCP messages just like names are written on envelopes.

Interfaces, protocols, and ports are all immediate concerns for software and users. By learning about these things, you develop a better appreciation for the way programs communicate and how your computer fits into the bigger picture.

5.1.2 Bigger picture: networks, NAT, and port forwarding

Interfaces are single points in larger networks. Networks are defined in the way that interfaces are linked together, and that linkage determines an interface's IP address.

Sometimes a message has a recipient that an interface is not directly linked to, so instead it's delivered to an intermediary that knows how to route the message for delivery. Coming back to the mail metaphor, this is similar to how real-world mail carriers operate.

When you place a message in your outbox, a mail carrier picks it up and delivers it to a local routing facility. That facility is itself an interface. It will take the message and send it along to the next stop on the route to a destination. A local routing facility for a mail carrier might forward a message to a regional facility, and then to a local facility for the destination, and finally to the recipient. It's common for network routes to follow a similar pattern. Figure

5.2 illustrates the described route and draws the relationships between physical message routing and network routing.

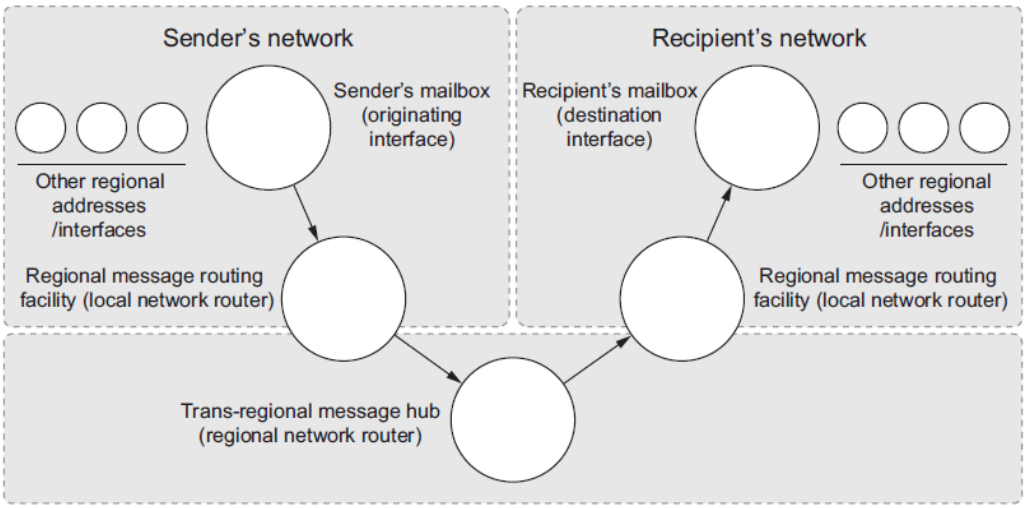


Figure 5.2 The path of a message in a postal system and a computer network

This chapter is concerned with interfaces that exist on a single computer, so the networks and routes we consider won't be anywhere near that complicated. In fact, this chapter is about two specific networks and the way containers are attached to them. The first network is the one that your computer is connected to. The second is a virtual network that Docker creates to connect all of the running containers to the network that the computer is connected to. That second network is called a *bridge*.

A bridge is an interface that connects multiple networks so that they can function as a single network, as shown in figure 5.3. Bridges work by selectively forwarding traffic between the connected networks based on another type of network address. To understand the material in this chapter, you only need to be comfortable with this abstract idea.

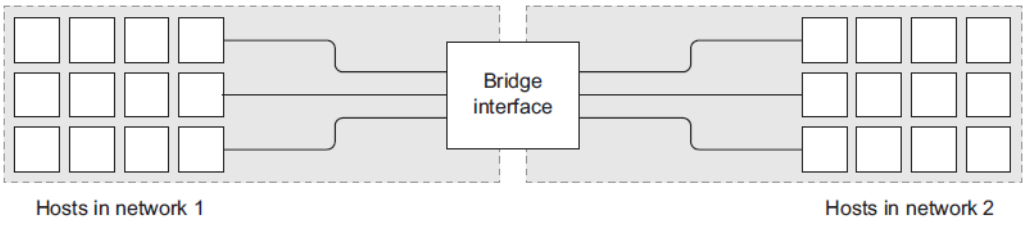


Figure 5.3 A bridge interface connecting two distinct networks

This has been a very rough introduction to some nuanced topics. This has only scratched the surface in order to help you understand how to use Docker and the networking facilities that it simplifies.

5.2 Docker container networking

Docker abstracts the underlying host-attached network from containers. Doing so provides a degree of runtime environment agnosticism for the application, and allows infrastructure managers to adapt the implementation to suit the operating environment. A container attached to a Docker network will get a unique IP address that is routable from other containers attached to the same Docker network.

The main problem with this approach is that there is no easy way for any software running inside a container to determine the IP address of the host where the container is running. This inhibits a container from advertising its service endpoint to other services outside of the container network. There are a few methods for dealing with this edge-case that will be covered in Section 5.5: Service Discovery.

Docker also treats networks as first-class entities. This means that they have their own life cycle and are not bound to any other objects. You can define and manage them directly using the `docker network` subcommands.

To get started with networks in Docker examine the default networks that are available with every Docker installation. Running `docker network ls` will print a table of all the networks to the terminal. The resulting table should look like this:

```
NETWORK ID          NAME                DRIVER              SCOPE
63d93214524b        bridge              bridge              local
6eeb489baff0        host                host                local
3254d02034ed        none                null                local
```

By default Docker includes three networks where each is provided by a different driver. The network named, "bridge" is the default network and provided by a "bridge" driver. The bridge driver provides inter-container connectivity for all containers running on the same machine. The "host" network is provided by the "host" driver, which instructs Docker not to create any special networking namespace or resources for attached containers. Containers on the "host" network interact with the host's network stack like uncontained processes. Finally, the "none" network uses the "null" driver. Containers attached to the "none" network will not have any network connectivity outside of themselves.

The "scope" of a network can take three values, "local," "global," or "swarm." This indicates if the network is constrained to the machine where the network exists (local), should be created on every node in a cluster but not route between them (global), or seamlessly spans all of the hosts participating in a Docker Swarm (multi-host or cluster-wide). As you can see, all of the default networks have the local scope, and will not be able to route traffic between containers running on different machines directly.

The default bridge network maintains compatibility with legacy Docker and cannot take advantage of modern Docker features like service discovery or IPVS based load balancing. Using it is not recommended. So the first thing you should do is create your own bridge network.

5.2.1 Creating a user-defined bridge network

The Docker bridge network driver uses Linux namespaces, virtual Ethernet devices, and the Linux firewall to build a specific and customizable virtual network topology called a bridge. The resulting virtual network is local to the machine where Docker is installed and creates routes between participating containers and the wider network where the host is attached. Figure 5.4 illustrates two containers attached to a bridge network and its components.

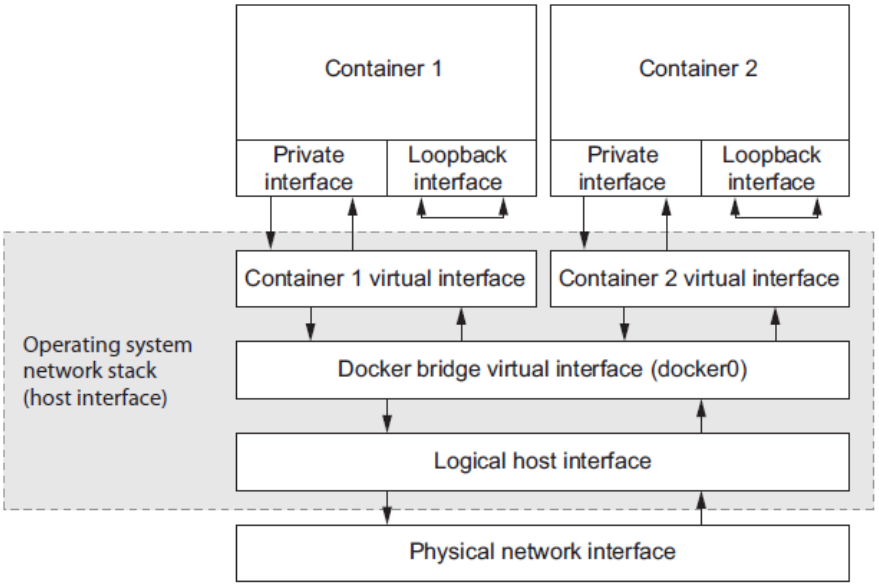


Figure 5.4 The default local Docker network topology and two attached containers

Containers have their own private loopback interface and a separate virtual Ethernet interface linked to another virtual interface in the host's namespace. These two linked interfaces form a link between the host's network and the container. Just like typical home networks, each container is assigned a unique private IP address that's not directly reachable from the external network. Connections are routed through another Docker network that routes traffic between containers and may connect to the host's network to form a bridge.

Build a new network with a single command:

```
docker network create \
--driver bridge \
--label project=dockerinaction \
--label chapter=5 \
--attachable \
--scope local \
--subnet 10.0.42.0/24 \
--ip-range 10.0.42.128/25 \
user-network
```

This command creates a new local bridge network named, "user-network." Adding label metadata to the network will help in identifying the resource later. Marking the new network as "attachable" allows you to attach and detach containers to the network at any time. Here you've manually specified the network scope property and set it to the default value for this driver. Finally, a custom subnet and assignable address range was defined for this network, "10.0.42.0/24" assigning from the upper-half of the last octet (10.0.42.128/25). This means that as you add containers to this network they will receive IP addresses in the range from 10.0.42.128 to 10.0.42.255.

You can inspect networks like other first class Docker entities. The next section demonstrates how to use containers with user networks and inspect the resulting network configuration.

5.2.2 Exploring a bridge network

If you're going to run network software inside a container on a container network then you should have a solid understanding of what that network looks like from within a container. Start exploring your new bridge network by creating a new container attached to that network:

```
docker run -it \
  --network user-network \
  --name network-explorer \
  alpine:3.8 \
    sh
```

Get a list of the IPv4 addresses available in the container from your terminal (which is now attached to the running container) by running the following:

```
ip -f inet -4 -o addr
```

The results should look something like this:

```
1: lo    inet 127.0.0.1/8 scope host lo\ ...
18: eth0    inet 10.0.42.129/24 brd 10.0.42.255 scope global eth0\ ...
```

You can see from this list that the container has two network devices with IPv4 addresses. Those are the loopback interface (or localhost) and eth0 (a virtual Ethernet device) which is connected to the bridge network. Further, you can see that eth0 has an IP address within the

range and subnet specified by the user-network configuration (the range from 10.0.42.128 to 10.0.42.255). That IP address is the one that any other container on this bridge network would use to communicate with services you run in this container. The loopback interface can only be used for communication within the same container.

Next, create another bridge network and attach your running `network-explorer` container to both networks. First, detach your terminal from the running container (press control and P and Q) then create the second bridge network:

```
docker network create \
  --driver bridge \
  --label project=dockerinaction \
  --label chapter=5 \
  --attachable \
  --scope local \
  --subnet 10.0.43.0/24 \
  --ip-range 10.0.43.128/25 \
  user-network2
```

Once the second network has been created you can attach the network-explorer container (still running):

```
docker network connect \
  user-network2 \          ❶
  network-explorer         ❷
```

❶ **Network name (or ID)**
❷ **Target container name (or ID)**

After the container has been attached to the second network, reattach your terminal to continue your exploration:

```
docker attach network-explorer
```

Now back in the container examining the network interface configuration again will show something like:

```
1: lo    inet 127.0.0.1/8 scope host lo\ ...
18: eth0    inet 10.0.42.129/24 brd 10.0.42.255 scope global eth0\ ...
20: eth1    inet 10.0.43.129/24 brd 10.0.43.255 scope global eth1\ ...
```

As you might expect, this output shows that the network-explorer container is attached to both user-defined bridge networks.

Networking is all about communication between multiple parties and examining a network with only one running container can be a bit boring. But is there anything else attached to a bridge network by default? Another tool is needed to continue exploring. Install the nmap package inside your running container using this command:

```
apk update && apk add nmap
```

Nmap is a powerful network inspection tool that can be used to scan network address ranges for running machines, fingerprint those machines, and determine what services they are running. For our purposes we simply want to determine what other containers or other network devices are available on our bridge network. Run the following command to scan the 10.0.42.0/24 subnet that we defined for our bridge network:

```
nmap -sn 10.0.42.* -sn 10.0.43.* -oG /dev/stdout | grep Status
```

The command should output something like this:

```
Host: 10.0.42.128 ()    Status: Up
Host: 10.0.42.129 (7c2c161261cb)        Status: Up
Host: 10.0.43.128 ()    Status: Up
Host: 10.0.43.129 (7c2c161261cb)        Status: Up
```

This shows that there are only two devices attached to each of the bridge networks: the gateway adapters created by the bridge network driver and the currently running container. Create another container on one of the two bridge networks for more interesting results.

Detach from the terminal again (control + P and Q) and start another container attached to user-network2. Run:

```
docker run -d \
  --name lighthouse \
  --network user-network2 \
  alpine:3.8 \
    sleep 1d
```

After the lighthouse container has started reattach to your network-explorer container:

```
docker attach network-explorer
```

And from the shell in that container run the network scan again. The results show that the "lighthouse" container is up and running, and accessible from the "network-explorer" container via its attachment to "user-network2." The output should be similar to this:

```
Host: 10.0.42.128 ()    Status: Up
Host: 10.0.42.129 (7c2c161261cb)        Status: Up
Host: 10.0.43.128 ()    Status: Up
Host: 10.0.43.130 (lighthouse.user-network2)    Status: Up
Host: 10.0.43.129 (7c2c161261cb)        Status: Up
```

Discovering the lighthouse container on the network confirm that the network attachment works as expected, and also demonstrates how the DNS based service discovery system works. When you scanned the network you discovered the new node by its IP address, and nmap was able to resolve that IP address to a name. This means that you (or your code) can discover individual containers on the network based on their name. Try this yourself by running `nslookup lighthouse` inside the container. Container hostnames are based on the container name, or be set manually at container creation time by specifying the `--hostname` flag.

This exploration has demonstrated your ability to shape bridge networks to fit your environment, the ability to attach a running container to more than one network, and what those networks look like to running software inside an attached container. But bridge networks only work on a single machine. They are not cluster aware and the container IP addresses are not routable from outside of that machine.

5.2.3 Beyond bridge networks

Depending on your use-case bridge networks might be enough. For example, bridge networks are typically great for single server deployments like a LAMP stack running a content management system, or most local development tasks. But if you are running a multi-server environment that is designed to tolerate machine failure then you need to be able to seamlessly route traffic between containers on different machines. Bridge networks will not do this.

Docker has a few options to handle this use-case out-of-the-box. The best option depends on the environment where you are building the network. If you are using Docker on Linux hosts and have control of the host network, then you can use underlay networks provided by the macvlan or ipvlan network drivers. Underlay networks create first-class network addresses for each container. Those identities are discoverable and routable from the same network where the host is attached. Each container running on a machine just look like independent nodes on the network.

If you are running Docker-for-Mac or Docker-for-Windows or are running in a managed cloud environment then those options will not work. Further, underlay network configuration is dependent on the host network and so definitions are rarely portable. The more popular multi-host container network option is overlay networks.

The overlay network driver is available on Docker engines where Swarm mode is enabled. Overlay networks are similar in construction to bridge networks, but the logical bridge component is multi-host aware and can route inter-container connections between every node in a Swarm.

Just like on a bridge network, containers on an overlay network are not directly routable from outside of the cluster. But, inter-container communication is simple and network definitions are mostly independent of the host network environment.

In some cases you'll have special network requirements that aren't covered by underlay or overlay networks. Maybe you need to be able to tune the host network configuration or you need to make sure that a container operates with total network isolation. In those cases you should use one of the special container networks.

5.3 Special container networks: host and none

When you list the available networks with `docker network list` the results will include two special entries: host and none. These are not really networks, instead they are network attachment types with special meaning.

When you specify the `--network host` option on a `docker run` command you are telling Docker to create a new container without any special network adapters or network namespace. Whatever software is running inside the resulting container will have the same degree of access to the host network as it would running outside of the container. Since there is no network namespace all of the kernel tools for tuning the network stack are available for modification (as long as the modifying process has access to do so).

Containers running on the host network are able to access host services running on localhost and are able to see and bind to any of the host network interfaces. The following command demonstrates this by listing all of the available network interfaces from inside a container on the host network:

```
docker run --rm \
    --network host \
    alpine:3.8 ip -o addr
```

Running on the host network is useful for system services or other infrastructure components. But it is not appropriate in multi-tenant environments and should be disallowed for third-party containers. Along these lines, there are often cases when you specifically do not want to attach a container to a network. In the spirit of building systems of least privilege you should use the "none" network whenever possible.

Creating a container on the none network instructs Docker not to provision any connected virtual Ethernet adapters for the new container. It will have its own network namespace and so it will be isolated, but without adapters connected across the namespace boundary it will not be able to use the network to communicate outside the container. Containers configured this way will still have their own loopback interface and so multi-process containers can still use connections to localhost for inter-process communication.

You can verify this by inspecting the network configuration yourself. Run the following command to list the available interfaces inside a container on the none network:

```
docker run --rm \
    --network none \
    alpine:3.8 ip -o addr
```

Running this example, you can see that the only network interface available is the loopback interface, bound to the address 127.0.0.1. This configuration means three things:

- Any program running in the container can connect to or wait for connections on that interface.
- Nothing outside the container can connect to that interface.
- No program running inside that container can reach anything outside the container.

That last point is important and easily demonstrable. If you’re connected to the internet, try to reach a popular service that should always be available. In this case, try to reach CloudFlare’s public DNS service:

```
docker run --rm \
```

```
    --network none \                    ❶
    alpine:3.8 \
    ping -w 2 1.1.1.1                   ❷
```

❶ Create a closed container
❷ Ping CloudFlare

In this example you create a network isolated container and try to test the speed between your container and the public DNS server provided by CloudFlare. This attempt should fail with a message like "ping: send-to: Network is unreachable." This makes sense because we know that the container has no route to the larger network.

When to use closed containers

The none network should be used when the need for network isolation is the highest or whenever a program doesn't require network access. For example, running a terminal text editor shouldn't require network access. Running a program to generate a random password should be run inside a container without network access to prevent the theft of that secret.

Containers on the none network are isolated from each other and the rest of the world, but remember that even containers on the bridge network are not directly routable from outside of the host running the Docker engine.

Bridge networks use network address translation (NAT) to make all outbound container traffic with destinations outside of the bridge network look like it is coming from the host itself. This means that the service software you have running in containers is isolated from the rest of the world, and the parts of the network where most of your clients and customers are location. The next section describes how to bridge that gap.

5.4 Handling inbound traffic with NodePort publishing

Docker container networks are all about simple connectivity and routing between containers. Connecting services running in those containers with external network clients requires an extra step. Since container networks are connected to the broader network via network address translation you have to specifically tell Docker how to forward traffic from the external network interfaces. You need to specify a TCP or UDP port on the host interface and a target container and container port, similar to forwarding traffic through a NAT barrier on your home network.

"NodePort publishing," is a term I've used here to match Docker and other ecosystem projects. The "Node" portion is an inference to the host as typically a node in a larger cluster of machines.

Port publication configuration is provided at container creation time and cannot be changed later. The `docker run` and `docker create` commands provide a `-p` or `--publish` list option. Like other options the `-p` option takes a colon-delimited string argument. That argument

specifies the host interface, the port on the host to forward, the target port, and the port protocol. All of the following arguments are equivalent:

- 0.0.0.0:8080:8080/tcp
- 8080:8080/tcp
- 8080:8080

Each of those options will forward TCP port 8080 from all host interfaces to TCP port 8080 in the new container. The first argument is the full form. To put the syntax in a more complete context consider the following example commands:

```
docker run --rm \
  -p 8080 \
  alpine:3.8 echo "forward ephemeral TCP -> container TCP 8080"

docker run --rm \
  -p 8088:8080/udp \
  alpine:3.8 echo "host UDP 8088 -> container UDP 8080"

docker run --rm \
  -p 127.0.0.1:8080:8080/tcp \
  -p 127.0.0.1:3000:3000/tcp \
  alpine: 3.8 echo "forward multiple TCP ports from localhost"
```

These commands all do different things and demonstrate the flexibility of the syntax. The first problem that new users encounter is in presuming that the first example will map 8080 on the host to port 8080 in the container. What actually happens is the host operating system will select a random host port and traffic will be routed to port 8080 in the container. The benefit to this design and default behavior is that ports are scarce resources and choosing a random port allows the software and the tooling to avoid potential conflicts. But programs running inside of a container have no way of knowing that they are running inside a container, that they are bound to a container network, or what port is being forwarded from the host.

Docker provides a mechanism for looking up port mappings. That feature is critical in cases where you let the operating system choose a port. Run the `docker port` subcommand to see the ports forwarded to any given container:

```
docker run -d -p 8080 --name listener alpine:3.8 sleep 300
docker port listener
```

This information is also available in summary form with the `docker ps` subcommand, but picking specific mappings out of the table can be tiresome and does not compose well with other commands. The docker port subcommand also allows you to narrow the lookup query by specifying the container port and protocol. That is particularly useful in cases where multiple ports are published:

```
docker run -d \
  -p 8080 \ ❶
  -p 3000 \ ❶
  -p 7500 \ ❶
```

```
  --name multi-listener \
  alpine:3.8 sleep 300

docker port multi-listener 3000 ❷
```

❶ **Publish multiple ports**
❷ **Lookup the host port mapped to container port 3000**

With the tools covered in this section, you should be able to manage routing any inbound traffic to the correct container running on your host. But, there are several other ways to customize container network configurations and caveats in working with Docker networks. Those are covered in the next section.

5.5 Container Networking Caveats and Customizations

Networking is used by all kinds of applications and in many contexts. Some bring requirements that cannot be fulfilled today, or might require further network customization. This section covers a short list of topics that any user should be familiar with in adopting containers for networked applications.

5.5.1 No firewalls or network policies

Today Docker container networks do not provide any access control or firewall mechanisms between containers. Docker networking was designed to follow the namespace model that is in use in so many other places in Docker. The namespace model solves resource access control problems by transforming them into addressability problems. The thinking is that if software in two containers are in the same container network that they should be able to communicate. In practice this is far from the truth and nothing short of application-level authentication and authorization can protect containers from each other on the same network. Remember different applications carry different vulnerabilities and might be running in containers on different hosts with different security postures. A compromised application does not need to escalate privileges before it opens network connections. The firewall will not protect you.

This design decision impacts the way we have to architect inter-network service dependencies and model common service deployments. In short, always deploy containers with appropriate application-level access control mechanisms because containers on the same container network will have mutual (bi-directional) unrestricted network access.

5.5.2 Custom DNS configuration

Domain Name System (DNS) is a protocol for mapping host names to IP addresses. This mapping enables clients to decouple from a dependency on a specific host IP and instead depend on whatever host is referred to by a known name. One of the most basic ways to change outbound communications is by creating names for IP addresses.

Typically, containers on the bridge network and other computers on your network have private IP addresses that aren't publicly routable. This means that unless you're running your

own DNS server, you can't refer to them by a name. Docker provides different options for customizing the DNS configuration for a new container.

First, the `docker run` command has a `--hostname` flag that you can use to set the host name of a new container. This flag adds an entry to the DNS override system inside the container. The entry maps the provided host name to the container's bridge IP address:

```
docker run --rm \
    --hostname barker \              ❶
    alpine:3.8 \
    nslookup barker                  ❷
```

❶ **Set the container host name**
❷ **Resolve the host name to an IP address**

This example creates a new container with the host name barker and runs a program to look up the IP address for the same name. Running this example will generate output that looks something like the following:

```
Server:    10.0.2.3
Address 1: 10.0.2.3

Name:      barker
Address 1: 172.17.0.22 barker
```

The IP address on the last line is the bridge IP address for the new container. The IP address provided on the line labeled Server is the address of the server that provided the mapping.

Setting the host name of a container is useful when programs running inside a container need to look up their own IP address or must self-identify. Because other containers don't know this hostname, its uses are limited. But if you use an external DNS server, you can share those hostnames.

The second option for customizing the DNS configuration of a container is the ability to specify one or more DNS servers to use. To demonstrate, the following example creates a new container and sets the DNS server for that container to Google's public DNS service:

```
docker run --rm \
    --dns 8.8.8.8 \                  ❶
    alpine:3.8 \
    nslookup docker.com              ❷
```

❶ **Set primary DNS server**
❷ **Resolve IP address of docker.com**

Using a specific DNS server can provide consistency if you're running Docker on a laptop and often move between internet service providers. It's a critical tool for people building services and networks. There are a few important notes on setting your own DNS server:

- *The value must be an IP address.* If you think about it, the reason is obvious; the container needs a DNS server to perform the lookup on a name.
- *The `--dns=[]` flag can be set multiple times to set multiple DNS servers (in case one or*

more are unreachable).

- *The* `--dns=[]` *flag can be set when you start up the Docker engine that runs in the background.* When you do so, those DNS servers will be set on every container by default. But if you stop the engine with containers running and change the default when you restart the engine, the running containers will still have the old DNS settings. You'll need to restart those containers for the change to take effect.

The third DNS-related option, `--dns-search=[]`, allows you to specify a DNS search domain, which is like a default host name suffix. With one set, any host names that don't have a known top-level domain (like .com or .net) will be searched for with the specified suffix appended.

```
docker run --rm \
    --dns-search docker.com \          ❶
    alpine:3.8 \
    nslookup hub                       ❷
```

❶ **Set search domain**
❷ **Look up shortcut for hub.docker.com**

This command will resolve to the IP address of hub.docker.com because the DNS search domain provided will complete the host name. It works by manipulating /etc/resolv.conf, a file used to configure common name resolution libraries. The following command shows how these DNS manipulation options impact the file:

```
docker run --rm \
    --dns-search docker.com \          ❶
    --dns 1.1.1.1 \                    ❷
    alpine:3.8 cat /etc/resolv.conf
# Will display contents that look like:
# search docker.com
# nameserver 1.1.1.1
```

❶ **Set search domain**
❷ **Set primary DNS server**

This feature is most often used for trivialities like shortcut names for internal corporate networks. For example, your company might maintain an internal documentation wiki that you can simply reference at http://wiki/. But this can be much more powerful.

Suppose you maintain a single DNS server for your development and test environments. Rather than building environment-aware software (with hard-coded environment-specific names like myservice.dev.mycompany.com), you might consider using DNS search domains and using environment-unaware names (like myservice):

```
docker run --rm \
    --dns-search dev.mycompany \       ❶
    alpine:3.8 \
    nslookup myservice                 ❷
```

```
docker run --rm \
    --dns-search test.mycompany \          ❸
    alpine:3.8 \
    nslookup myservice                     ❹
```

❶ Note dev prefix
❷ Resolves to myservice.dev.mycompany
❸ Note test prefix
❹ Resolves to myservice.test.mycompany

Using this pattern, the only change is the context in which the program is running. Like providing custom DNS servers, you can provide several custom search domains for the same container. Simply set the flag as many times as you have search domains. For example:

```
docker run --rm \
    --dns-search mycompany \
    --dns-search myothercompany ...
```

This flag can also be set when you start up the Docker engine to provide defaults for every container created. Again, remember that these options are only set for a container when it is created. If you change the defaults when a container is running, that container will maintain the old values.

The last DNS feature to consider provides the ability to override the DNS system. This uses the same system that the `--hostname` flag uses. The `--add-host=[]` flag on the `docker run` command lets you provide a custom mapping for an IP address and host name pair:

```
docker run --rm \
    --add-host test:10.10.10.255 \         ❶
    alpine:3.8 \
    nslookup test                          ❷
```

❶ Add host entry
❷ Resolves to 10.10.10.255

Like `--dns` and `--dns-search`, this option can be specified multiple times. But unlike those other options, this flag can't be set as a default at engine startup.

This feature is a sort of name resolution scalpel. Providing specific name mappings for individual containers is the most fine-grained customization possible. You can use this to effectively block targeted host names by mapping them to a known IP address like 127.0.0.1. You could use it to route traffic for a particular destination through a proxy. This is often used to route unsecure traffic through secure channels like an SSH tunnel. Adding these overrides is a trick that has been used for years by web developers who run their own local copies of a web application. If you spend some time thinking about the interface that name-to-IP address mappings provide, I'm sure you can come up with all sorts of uses.

All the custom mappings live in a file at /etc/hosts inside your container. If you want to see what overrides are in place, all you have to do is inspect that file. Rules for editing and parsing this file can be found online and are a bit beyond the scope of this book:

```
docker run --rm \
    --hostname mycontainer \                  (1)
    --add-host docker.com:127.0.0.1 \         (2)
    --add-host test:10.10.10.2 \              (3)
    alpine:3.8 \
    cat /etc/hosts                            (4)
```

(1) Set host name
(2) Create host entry
(3) Create another host entry
(4) View all entries

This should produce output that looks something like the following:

```
172.17.0.45  mycontainer
127.0.0.1    localhost
::1          localhost ip6-localhost ip6-loopback
fe00::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters
10.10.10.2   test
127.0.0.1    docker.com
```

DNS is a powerful system for changing behavior. The name-to-IP address map provides a simple interface that people and programs can use to decouple themselves from specific network addresses. If DNS is your best tool for changing outbound traffic behavior, then the firewall and network topology is your best tool for controlling inbound traffic.

5.5.3 Externalizing network management

Finally, some organizations, infrastructures, or products require direct management of container network configuration, service discovery, and other network related resources. In those cases you, or the container orchestrator you are using will create containers using the Docker `none` network. Then use some other container-aware tooling to create and manage the container network interfaces, manage nodeport publishing, register containers with service discovery systems, and integrate with upstream load balancing systems.

Kubernetes has a whole ecosystem of networking providers, and depending on how you are consuming Kubernetes (as the project, a productized distribution, or managed service) you may or may not have any say in what provider you use. Entire books could be written about networking options for Kubernetes. I won't do them the disservice of attempting to summarize them here.

Above network provider layer, there is a whole continuum of service discovery tools that leverage different features of Linux and container technology. Service Discovery is not a 'solved' problem so the solution landscape changes quickly. If you find Docker networking constructs insufficient to solve your integration and management problems, then survey the field. Each tool has its own documentation and implementation patterns and you will need to consult those guides to integrate them effectively with Docker.

When you externalize network management Docker is still responsible for creating the network namespace for the container, but it will not create or manage any of the network interfaces. You will not be able to use any of the Docker tooling to inspect the network configuration or port mapping. If you are running a blended environment where some container networking has been externalized then the built-in service discovery mechanisms cannot be used to route traffic from Docker managed containers to externalized containers. Blended environments are rare and should be avoided.

5.6 Summary

Networking is a broad subject that would take several books to properly cover. This chapter should help readers with a basic understanding of network fundamentals adopt the single-host networking facilities provided by Docker. In reading this material, you learned the following:

- Docker networks are first-class entities that can be created, listed, and removed just like containers, volumes, and images.
- Bridge networks are a special kind of network that allows direct inter-container network communication with built-in container name resolution.
- Docker provides two other special networks by default: none and host.
- Networks created with the `none` driver will isolate attached containers from the network.
- A container on a host network will have full access to the network facilities and interfaces on the host.
- Forward network traffic to a host port into a target container and port with NodePort publishing.
- Docker bridge networks do not provide any network firewall or access control functionality.
- The network name resolution stack can be customized for each container. Custom DNS servers, search domains, and static hosts can be defined.
- Network management can be externalized with third-party tooling and by using the Docker `none` network.

6

Limiting risk with resource controls

This chapter covers

- Setting resource limits
- Sharing container memory
- Users, permissions, and administrative privileges
- Granting access to specific Linux features
- Working with enhanced Linux isolation and security tools: SELinux and AppArmor

Containers provide isolated process contexts, not whole system virtualization. The semantic difference may seem subtle, but the impact is drastic. Chapter 1 touches on the differences a bit. Chapters 2 through 5 each cover a different isolation feature set of Docker containers. This chapter covers the remaining four and also includes information about enhancing security on your system.

The features covered in this chapter focus on managing or limiting the risks of running software. These features prevent software misbehaving from a bug or attack from consuming resources that might leave your computer unresponsive. Containers can help ensure that software only uses the computing resources and accesses the data you expect. You will learn how to give containers resource allowances, access to shared memory, run programs as specific users, control the type of changes that a container can make to your computer, and integrate with other Linux isolation tools. Some of these topics involve Linux features that are beyond the scope of this book. In those cases I try to give you an idea about their purpose and some basic usage examples, and you can integrate them with Docker. Figure 6.1 shows the eight namespaces and features that are used to build Docker containers.

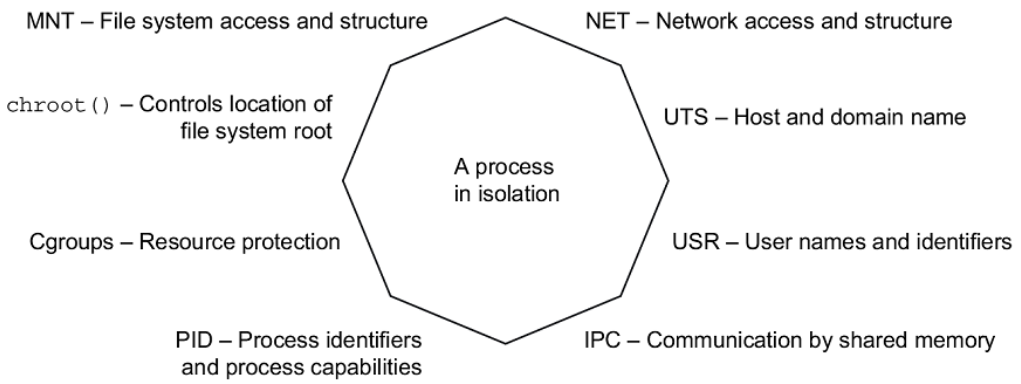


Figure 6.1 Eight-sided containers

One last reminder, Docker and the technology it uses are evolving projects. The examples in this chapter should work for Docker 1.13 and later. Once you learn the tools presented in this chapter, remember to check for developments, enhancements, and new best practices when you go to build something valuable.

6.1 Resource allowances

Physical system resources like memory and time on the CPU are scarce. If the resource consumption of processes on a computer exceeds the available physical resources, the processes will experience performance issues and may stop running. Part of building a system that creates strong isolation includes providing resource allowances on individual containers.

If you want to make sure that a program won't overwhelm other programs on your computer, then the easiest thing to do is set limits on the resources that it can use. You can manage memory, CPU, and device resource allowances with Docker. By default, Docker containers may use unlimited cpu, memory, and device i/o resources. The `docker container create` and `run` commands provide flags for managing resources available to the container.

6.1.1 Memory limits

Memory limits are the most basic restriction you can place on a container. They restrict the amount of memory that processes inside a container can use. Memory limits are useful for ensuring that one container can't allocate all of the system's memory, starving other programs for the memory they need. You can put a limit in place by using the `-m` or `--memory` flag on the `docker container run` or `docker container create` commands. The flag takes a value and a unit. The format is as follows:

```
<number><optional unit> where unit = b, k, m or g
```

In the context of these commands, `b` refers to bytes, `k` to kilobytes, `m` to megabytes, and `g` to gigabytes. Put this new knowledge to use and start up a database application that you'll use in other examples:

```
docker container run -d --name ch6_mariadb \
    --memory 256m \                          (1)
    --cpu-shares 1024 \
    --cap-drop net_raw \
    -e MYSQL_ROOT_PASSWORD=test \
    mariadb:5.5
```

(1) **Set a memory constraint**

With this command you install database software called MariaDB and start a container with a memory limit of 256 megabytes. You might have noticed a few extra flags on this command. This chapter covers each of those, but you may already be able to guess what they do. Something else to note is that you don't expose any ports or bind any ports to the host's interfaces. It will be easiest to connect to this database by linking to it from another container on the host. Before we get to that, I want to make sure you have a full understanding of what happens here and how to use memory limits.

The most important thing to understand about memory limits is that they're not reservations. They don't guarantee that the specified amount of memory will be available. They're only a protection from overconsumption. Additionally, the implementation of the memory accounting and limit enforcement by the Linux kernel is very efficient so you do not need to worry about runtime overhead for this feature.

Before you put a memory allowance in place, you should consider two things. First, can the software you're running operate under the proposed memory allowance? Second, can the system you're running on support the allowance?

The first question is often difficult to answer. It's not common to see minimum requirements published with open source software these days. Even if it were, though, you'd have to understand how the memory requirements of the software scale based on the size of the data you're asking it to handle. For better or worse, people tend to overestimate and adjust based on trial and error. One option is to run the software in a container with real workloads and use the `docker stats` command to see how much memory the container uses in practice. For the mariadb container we just started, `docker stats ch6_mariadb` shows the container is using about 100 megabytes of memory, fitting well inside its 256 megabyte limit. In the case of memory-sensitive tools like databases, skilled professionals such as database administrators can make better-educated estimates and recommendations. Even then, the question is often answered by another: how much memory do you have? And that leads to the second question.

Can the system you're running on support the allowance? It's possible to set a memory allowance that's bigger than the amount of available memory on the system. On hosts that have swap space (virtual memory that extends onto disk), a container may realize the allowance. It is possible to specify an allowance that's greater than any physical memory

resource. In those cases the limitations of the system will always cap the container and runtime behavior will be similar to not having specified an allowance at all.

Finally, understand that there are several ways that software can fail if it exhausts the available memory. Some programs may fail with a memory access fault, whereas others may start writing out-of-memory errors to their logging. Docker neither detects this problem nor attempts to mitigate the issue. The best it can do is apply the restart logic you may have specified using the `--restart` flag described in chapter 2.

6.1.2 CPU

Processing time is just as scarce as memory, but the effect of starvation is performance degradation instead of failure. A paused process that is waiting for time on the CPU is still working correctly. But a slow process may be worse than a failing one if it's running some important latency-sensitive data-processing program, a revenue-generating web application, or a back-end service for your app. Docker lets you limit a container's CPU resources in two ways.

First, you can specify the relative weight of a container to other containers. Linux uses this to determine the percentage of CPU time the container should use relative to other running containers. That percentage is for the sum of the computing cycles of all processors available to the container.

To set the CPU shares of a container and establish its relative weight, both `docker container run` and `docker container create` offer a `--cpu-shares` flag. The value provided should be an integer (which means you shouldn't quote it). Start another container to see how CPU shares work:

```
docker container run -d -P --name ch6_wordpress \
--memory 512m \
--cpu-shares 512 \                              ❶
--cap-drop net_raw \
--link ch6_mariadb:mysql \
-e WORDPRESS_DB_PASSWORD=test \
wordpress:4.1
```

❶ Set a relative process weight

This command will download and start WordPress version 4.1. It's written in PHP and is a great example of software that has been challenged by adapting to security risks. Here we've started it with a few extra precautions. If you'd like to see it running on your computer, use `docker port ch6_wordpress` to get the port number (I'll call it `<port>`) that the service is running on and open http://localhost:<port> in your web browser. Remember, if you're using Docker Machine, you'll need to use `docker-machine ip` to determine the IP address of the virtual machine where Docker is running. When you have that, substitute that value for localhost in the preceding URL.

When you started the MariaDB container, you set its relative weight (`cpu-shares`) to `1024`, and you set the relative weight of WordPress to 512. These settings create a system where

the MariaDB container gets two CPU cycles for every one WordPress cycle. If you started a third container and set its `--cpu-shares` value to `2048`, it would get half of the CPU cycles, and MariaDB and WordPress would split the other half at the same proportions as they were before. Figure 6.2 shows how portions change based on the total weight of the system.

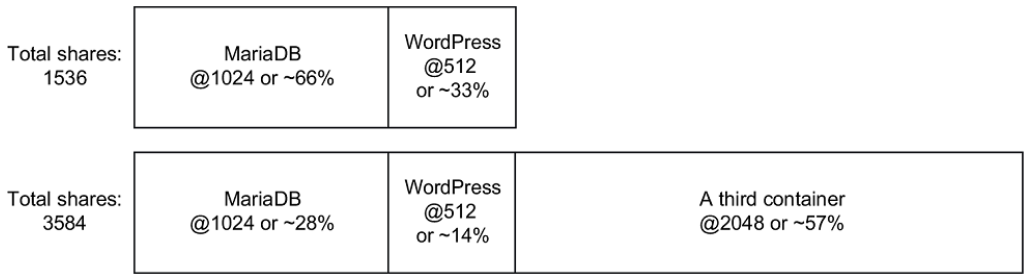


Figure 6.2 Relative weight and CPU shares

CPU shares differ from memory limits in that they're enforced *only* when there is contention for time on the CPU. If other processes and containers are idle, then the container may burst well beyond its limits. This approach ensures sure that CPU time is not wasted and that limited processes will yield if another process needs the CPU. The intent of this tool is to prevent one or a set of processes from overwhelming a computer, not to hinder performance of those processes. The defaults won't limit the container, and it will be able to use 100% of the CPU if the machine is otherwise idle.

Now that we have learned how `cpu-shares` allocates cpu proportionately, we will introduce the `cpus` option which provides a way to limit the total amount of cpu used by a container. The `cpus` option allocates a quota of CPU resources the container may use by configuring Linux' Completely Fair Scheduler (CFS). Docker helpfully allows the quota to be expressed as the number of CPU cores the container should be able to use. The cpu quota is allocated, enforced, and ultimately refreshed every 100ms by default. If a container uses all of its cpu quota, its cpu usage will be throttled until the next measurement period begins. The following command will let the previous wordpress example consume a maximum of 0.75 cpus:

```
docker container run -d -P --name ch6_wordpress \
--memory 512m \
--cpus 0.75 \                                    (1)
--cap-drop net_raw \
--link ch6_mariadb:mysql \
-e WORDPRESS_DB_PASSWORD=test \
wordpress:4.1
```

#Use a maximum of 0.75 cpus

Another feature Docker exposes is the ability to assign a container to a specific CPU set. Most modern hardware uses multi-core CPUs. Roughly speaking, a CPU can process as many

instructions in parallel as it has cores. This is especially useful when you're running many processes on the same computer.

A context switch is the task of changing from executing one process to executing another. Context switching is expensive and may cause a noticeable impact on the performance of your system. In some cases, it makes sense to reduce context switching of critical processes by ensuring they are never executed on the same set of CPU cores. You can use the `--cpuset-cpus` flag on `docker container run` or `docker container create` to limit a container to execute only on a specific set of CPU cores.

You can see the CPU set restrictions in action by stressing one of your machine cores and examining your CPU workload:

```
# Start a container limited to a single CPU and run a load generator
docker container run -d \
    --cpuset-cpus 0 \                                          (1)
    --name ch6_stresser dockerinaction/ch6_stresser

# Start a container to watch the load on the CPU under load
docker container run -it --rm dockerinaction/ch6_htop
```

(1) **Restrict to CPU number 0**

Once you run the second command, you'll see `htop` display the running processes and the workload of the available CPUs. The `ch6_stresser` container will stop running after 30 seconds, so it's important not to delay when you run this experiment. When you finish with `htop`, press Q to quit. Before moving on, remember to shut down and remove the container named `ch6_stresser`:

```
docker rm -vf ch6_stresser
```

I thought this was exciting when I first used it. To get the best appreciation, repeat this experiment a few different times using different values for the `--cpuset-cpus` flag. If you do, you'll see the process assigned to different cores or different sets of cores. The value can be either list or range forms:

- `0,1,2`—A list including the first three cores of the CPU
- `0-2`—A range including the first three cores of the CPU

6.1.3 Access to devices

Devices are the final resource type we will cover. Controlling access to devices differs from memory and CPU limits. Providing access to a host's device inside a container is more like a resource authorization control than a limit.

Linux systems have all sorts of devices, including hard drives, optical drives, USB drives, mouse, keyboard, sound devices, and webcams. Containers have access to some of these devices by default, and Docker creates others for each container (like virtual terminals).

On occasion, it may be important to share other devices between a host and a specific container. Consider a situation where you're running some computer vision software that requires access to a webcam. In that case you'll need to grant access to the container running your software to the webcam device attached to the system; you can use the `--device` flag to specify a set of devices to mount into the new container. The following example would map your webcam at /dev/video0 to the same location within a new container. Running this example will work only if you have a webcam at /dev/video0:

```
docker container run -it --rm \
    --device /dev/video0:/dev/video0 \          (1)
    ubuntu:latest ls -al /dev
```

(1) **Mount video0**

The value provided must be a map between the device file on the host operating system and the location inside the new container. The device flag can be set many times to grant access to different devices.

People in situations with custom hardware or proprietary drivers will find this kind of access to devices useful. It's preferable to resorting to modifying their host operating system.

6.2 Shared memory

Linux provides a few tools for sharing memory between processes running on the same computer. This form of inter-process communication (IPC) performs at memory speeds. It's often used when the latency associated with network or pipe-based IPC drags software performance down below requirements. The best examples of shared memory-based IPC use are in scientific computing and some popular database technologies like PostgreSQL.

Docker creates a unique IPC namespace for each container by default. The Linux IPC namespace partitions share memory primitives such as named shared memory blocks and semaphores, as well as message queues. It's okay if you're not sure what these are. Just know that they're tools used by Linux programs to coordinate processing. The IPC namespace prevents processes in one container from accessing the memory on the host or in other containers.

6.2.1 Sharing IPC primitives between containers

I've created an image named `dockerinactionch6_ipc` that contains both a producer and consumer. They communicate using shared memory. The following will help you understand the problem with running these in separate containers:

```
docker container run -d -u nobody --name ch6_ipc_producer \        (1)
    dockerinaction/ch6_ipc -producer

docker container run -d -u nobody --name ch6_ipc_consumer \        (2)
    dockerinaction/ch6_ipc -consumer
```

1. Start producer
2. Start consumer

These commands start two containers. The first creates a message queue and begins broadcasting messages on it. The second should pull from the message queue and write the messages to the logs. You can see what each is doing by using the following commands to inspect the logs of each:

```
docker logs ch6_ipc_producer

docker logs ch6_ipc_consumer
```

Notice that something is wrong with the containers you started. The consumer never sees any messages on the queue. Each process used the same key to identify the shared memory resource, but they referred to different memory. The reason is that each container has its own shared memory namespace.

If you need to run programs that communicate with shared memory in different containers, then you'll need to join their IPC namespaces with the `--ipc` flag. The `--ipc` flag has a container mode that will create a new container in the same IPC namespace as another target container. This is just like the `--net` flag covered in chapter 5. Figure 6.3 illustrates the relationship between containers and their namespaced shared memory pools.

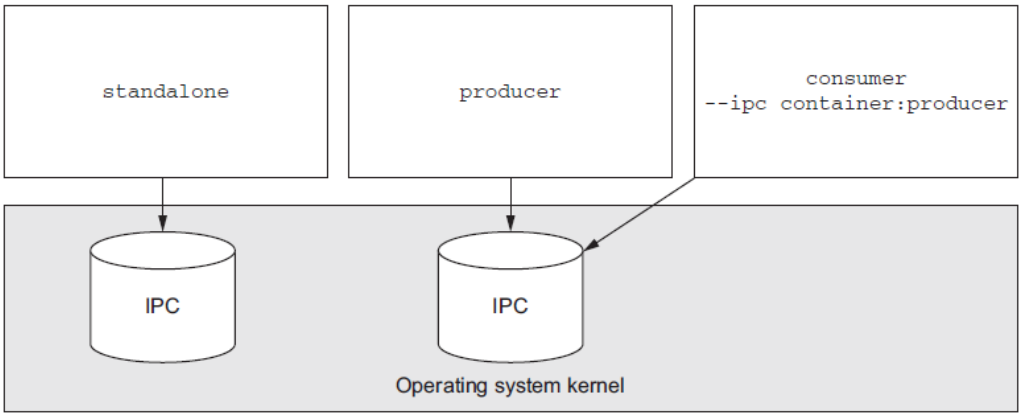


Figure 6.3 Three containers and their shared memory pools. Producer and consumer share a single pool.

Use the following commands to test joined IPC namespaces for yourself:

```
docker container rm -v ch6_ipc_consumer                    (1)

docker container run -d --name ch6_ipc_consumer \          (2)
    --ipc container:ch6_ipc_producer \                     (3)
    dockerinaction/ch6_ipc -consumer
```

1. Remove original consumer
2. Start new consumer
3. Join IPC namespace

These commands rebuild the consumer container and reuse the IPC namespace of the `ch6_ipc_producer` container. This time the consumer should be able to access the same memory location where the server is writing. You can see this working by using the following commands to inspect the logs of each:

```
docker logs ch6_ipc_producer

docker logs ch6_ipc_consumer
```

Remember to clean up your running containers before moving on:

- The `v` option will clean up volumes.
- The `f` option will kill the container if it is running.
- The `rm` command takes a list of containers.

```
docker rm -vf ch6_ipc_producer ch6_ipc_consumer
```

There are obvious security implications to reusing the shared memory namespaces of containers. But this option is available if you need it. Sharing memory between containers is a safer alternative than sharing memory with the host. Sharing memory with the host is possible using the `--ipc=host` option. However, sharing host memory is difficult in modern Docker distributions because it contradicts Docker's secure by default posture for containers.

Feel free to check out the source code for this example. It's an ugly but simple C program. You can find it by checking out the source repository linked to from the image's page on Docker Hub.

6.3 Understanding users

Docker starts containers as the user specified by the image metadata by default, which is often the root user. The root user has almost full privileged access to the state of the container. Any processes running as that user inherit those permissions. It follows that if there's a bug in one of those processes, they might damage the container. There are ways to limit the damage, but the most effective way to prevent these types of issues is not to use the root user.

There are reasonable exceptions when using the root user is the best if not only available option. You use the root user for building images and at runtime when there's no other option. There are other similar situations when you want to run system administration software inside a container. In those cases the process needs privileged access not only to the container but also to the host operating system. This section covers the range of solutions to these problems.

6.3.1 Working with the run-as user

Before you create a container, it would be nice to be able to tell what username (and user ID) is going to be used by default. The default is specified by the image. There's currently no way to examine an image to discover attributes like the default user in Docker Hub. You can inspect image metadata using the `docker inspect` command. If you missed it in chapter 2, the `inspect` subcommand displays the metadata of a specific container or image. Once you've pulled or created an image, you can get the default username that the container is using with the following commands:

```
docker pull busybox:latest
docker inspect busybox:latest                                    (1)
docker inspect --format "{{.Config.User}}" busybox:latest        (2)
```

(1) Display all of busybox's metadata
(2) Show only the run-as user defined by the busybox image

If the result is blank, the container will default to running as the root user. If it isn't blank, either the image author specifically named a default run-as user or you set a specific run-as user when you created the container. The `--format` or `-f` option used in the second command allows you to specify a template to render the output. In this case you've selected the `User` field of the `Config` property of the document. The value can be any valid GoLang template, so if you're feeling up to it, you can get creative with the results.

There is a problem with this approach. The run-as user might be changed by the entrypoint or command the image uses to start up. These are sometimes referred to as boot, or init, scripts. The metadata returned by `docker inspect` includes only the configuration that the container will start with. So if the user changes, it won't be reflected there.

Currently, the only way to fix this problem is to look inside the image. You could expand the image files after you download them and examine the metadata and init scripts by hand, but doing so is time-consuming and easy to get wrong. For the time being, it may be better to run a simple experiment to determine the default user. This will solve the first problem but not the second:

```
docker container run --rm --entrypoint "" busybox:latest whoami    (1)

docker container run --rm --entrypoint "" busybox:latest id        (2)
```

(1) Outputs: root
(2) Outputs: uid=0(root) gid=0(root) groups=10(wheel)

This demonstrates two commands that you might use to determine the default user of an image (in this case, busybox:latest). Both the `whoami` and `id` commands are common among Linux distributions, and so they're likely to be available in any given image. The second command is superior because it shows both the name and ID details for the run-as user. Both these commands are careful to unset the `entrypoint` of the container. This will make sure that the command specified after the image name is the command that is executed by the

container. These are poor substitutes for a first-class image metadata tool, but they get the job done. Consider the brief exchange between two root users in figure 6.4.

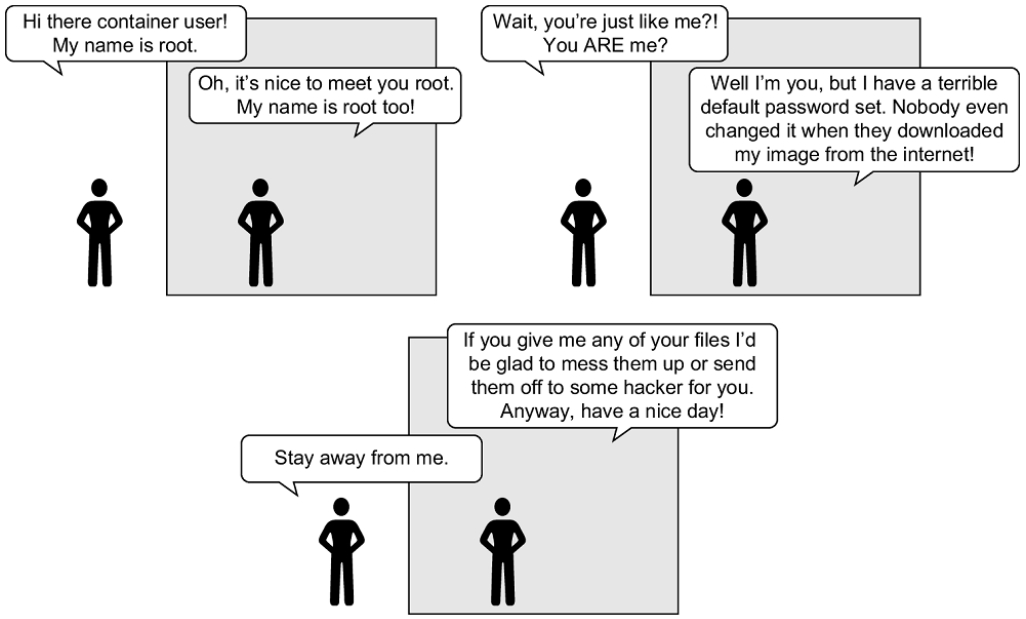


Figure 6.4 Root vs. root—a security drama

You can entirely avoid the default user problem if you change the run-as user when you create the container. The quirk with using this is that the username must exist on the image you're using. Different Linux distributions ship with different users predefined, and some image authors reduce or augment that set. You can get a list of available users in an image with the following command:

```
docker container run --rm busybox:latest awk -F: '$0=$1' /etc/passwd
```

I won't go into much detail here, but the Linux user database is stored in a file located at /etc/passwd. This command will read that file and pull a list of the usernames. Once you've identified the user you want to use, you can create a new container with a specific run-as user. Docker provides the `--user` or `-u` flag on `docker container run` and `docker container create` for setting the user. This will set the user to "nobody":

```
docker container run --rm \
    --user nobody \              ❶
    busybox:latest id            ❷
```

❶ Set run-as user to nobody

❷ Outputs: uid=99(nobody) gid=99(nobody)

This command used the "nobody" user. That user is very common and intended for use in restricted-privileges scenarios like running applications. That was just one example. You can use any username defined by the image here, including root. This only scratches the surface of what you can do with the `-u` or `--user` flag. The value can accept any user or group pair. It can also accept user and group names or IDs. When you use IDs instead of names, the options start to open up:

```
docker container run --rm \
    -u nobody:default \       ❶
    busybox:latest id         ❷

docker container run --rm \
    -u 10000:20000 \          ❸
    busybox:latest id         ❹
```

❶ Set run-as user to nobody and group to default
❷ Outputs: uid=99(nobody) gid=1000(default)
❸ Set UID and GID
❹ Outputs: uid=10000 gid=20000

The second command starts a new container that sets the run-as user and group to a user and group that do not exist in the container. When that happens, the IDs won't resolve to a user or group name, but all file permissions will work as if the user and group did exist. Depending on how the software packaged in the container is configured, changing the run-as user may cause problems. Otherwise, this is a powerful feature that can make file-permission issues simple to resolve.

The best way to be confident in your runtime configuration is to pull images from trusted sources or build your own. As with any standard Linux distribution, it's possible to do malicious things like turning a default non-root user into the root user using an suid-enabled program (making your attempt at safety a trap) or opening up access to the root account without authentication. Images should be analyzed and secured like a full Linux host using principle of least privilege. Fortunately, Docker images can be purpose-built to support the application that needs to be run with everything else left out. Chapter 7 covers this topic briefly.

6.3.2 Users and volumes

Now that you've learned how users inside containers share the same user ID space as the users on your host system, you need to learn how those two might interact. The main reason for that interaction is the file permissions on files in volumes. For example, if you're running a Linux terminal, you should be able to use these commands directly; otherwise, you'll need to use the `docker-machine ssh` command to get a shell in your Docker Machine virtual machine:

```
echo "e=mc^2" > garbage          ❶
```

```
chmod 600 garbage                    ❷

sudo chown root:root garbage         ❸

docker container run --rm -v "$(pwd)"/garbage:/test/garbage \
    -u nobody \
    ubuntu:latest cat /test/garbage                      ❹


docker container run --rm -v "$(pwd)"/garbage:/test/garbage \
    -u root ubuntu:latest cat /test/garbage              ❺
# Outputs: "e=mc^2"

# cleanup that garbage
sudo rm -f garbage
```

❶ Create new file on your host
❷ Make file readable only by its owner
❸ Make file owned by root (assuming you have sudo access)
❹ Try to read file as nobody
❺ Try to read file as "container root"

The second-to-last `docker` command should fail with an error message like "Permission denied." But the last `docker` command should succeed and show you the contents of the file you created in the first command. This means that file permissions on files in volumes are respected inside the container. But this also reflects that the user ID space is shared. Both root on the host and root in the container have user ID 0. So, although the container's nobody user with ID 65534 can't access a file owned by root on the host, the container's root user can.

Unless you want a file to be accessible to a container, don't mount it into that container with a volume.

The good news about this example is that you've seen how file permissions are respected and can solve some more mundane—but practical—operational issues. For example, how do you handle a log file written to a volume?

The preferred way is with volume containers, as described in chapter 4. But even then you need to consider file ownership and permission issues. If logs are written to a volume by a process running as user 1001 and another container tries to access that file as user 1002, then file permissions might prevent the operation.

One way to overcome this obstacle would be to specifically manage the user ID of the running user. You can either edit the image ahead of time by setting the user ID of the user you're going to run the container with, or you can use the desired user and group ID:

```
mkdir logFiles

sudo chown 2000:2000 logFiles                          ❶

docker container run --rm -v "$(pwd)"/logFiles:/logFiles \      ❷
    -u 2000:2000 ubuntu:latest \                       ❸
    /bin/bash -c "echo This is important info > /logFiles/important.log"
```

```
docker container run --rm -v "$(pwd)"/logFiles:/logFiles \      ❹
    -u 2000:2000 ubuntu:latest \                                ❺
    /bin/bash -c "echo More info >> /logFiles/important.log"

sudo rm –r logFiles
```

❶ Set ownership of directory to desired user and group
❷ Write important log file
❸ Set UID:GID to 2000:2000
❹ Append to log from another container
❺ Also set UID:GID to 2000:2000

After running this example, you'll see that the file could be written to the directory that's owned by user 2000. Not only that, but any container that uses a user or group with write access to the directory could write a file in that directory or to the same file if the permissions allow. This trick works for reading, writing, and executing files.

One uid and filesystem interaction bears special mention. By default, the Docker daemon API is accessible via a Unix domain socket located on the host at `/var/run/docker.sock`. The domain socket is protected with filesystem permissions ensuring only the `root` user and members of the `docker` group may send commands or retrieve data from the Docker daemon. Some programs are built to interact directly with the Docker daemon API and know how to send commands to inspect or run containers.

The Power of the Docker API

The `docker` command-line program interacts with the docker daemon almost entirely via the API, which should give you a sense of how powerful the API is. Any program that can read and write to the Docker API can do anything `docker` can do, subject to Docker's Authorization plugin system.

Programs that manage or monitor containers often require the ability to read or even write to the Docker daemon's endpoint. The ability to read or write to Docker's API is often provided by running the management program as a user or group that has permission to read or write to docker.sock and mounting /var/run/docker.sock into the container:

```
docker container run --rm -it
  –v /var/run/docker.sock:/var/run/docker.sock:ro \    ❶
    -u root        monitoringtool                      ❷
```

❶ Bind docker.sock from host into container as a read-only file
❷ Container runs as root user, aligning with file permissions on host

The preceding example illustrates a relatively common request by authors of privileged programs. You should be very careful about which users or programs on your systems can control your Docker daemon. If a user or program controls your Docker daemon, it effectively controls the root account on your host and can run any program or delete any file.

6.3.3 Introduction to the Linux user namespace and uid remapping

Linux's user (USR) namespace maps users in one namespace to users in another. The user namespace operates like the process identifier (PID) namespace with container uids and gids partitioned from the host's default identities.

By default, Docker containers do not use the USR namespace. This means that a container running with a user ID (number, not name) that's the same as a user on the host machine has the same host file permissions as that user. This isn't a problem. The file system available inside a container has been mounted so that changes made inside that container will stay inside that container's file system. But this does impact volumes where files are shared between containers or with the host.

When a user namespace is enabled for a container, the container's uids are mapped to a range of unprivileged uids on the host. Operators activate user namespace remapping by defining a map of subuid and subguid maps for the host in Linux and configuring the Docker daemon's `userns-remap` option. The mappings determine how user IDs on the host correspond to user IDs in a container namespace. For example, uid remapping could be configured to map container uids to the host starting with host uid 5000 and a range of 1000 uids. The result is that uid 0 in containers would be mapped to host uid 5000, container uid 1 to host uid 5001, and so on for 1000 uids. Since uid 5000 is an unprivileged user from Linux' perspective and doesn't have permissions to modify the host system files, the risk of running with 'uid=0' in the container is greatly reduced. Even if a containerized process gets ahold of a file or other resource from the host, the containerized process will be running as a remapped uid without privileges to do anything with that resource unless an operator specifically gave it permissions to do so.

User namespace remapping is particularly useful for resolving file permissions issues in cases like reading and writing to volumes. Let's step through an example of sharing a filesystem between containers whose process run as uid 0 in the container with user namespacing enabled. In our example, we will assume Docker is using:

- the (default) `dockremap` user for remapping container uid and gid ranges
- an entry in /etc/subuid of `dockremap:5000:10000`, providing a range of 10,000 uids starting at 5000
- an entry in /etc/subgid of `dockremap:5000:10000`, providing a range of 10,000 gids starting at 5000

First, let's check the user and group id of the dockeremap user on the host. Then, we will create a shared directory owned by the remapped container uid 0, host uid 5000.

```
# id dockremap  (1)
uid=997(dockremap) gid=993(dockremap) groups=993(dockremap)
# cat /etc/subuid
dockremap:5000:10000
# cat /etc/subgid
dockremap:5000:10000
# mkdir /tmp/shared
```

```
# chown -R 5000:5000 /tmp/shared ❷
```

❶ inspect user and group id of dockremap user on host
❷ change ownership of 'shared' directory to uid used for remapped container uid 0

Now run a container as the container's 'root' user:

```
# docker run -it --rm --user root -v /tmp/shared:/shared alpine ash
/ # touch /host/afile ❸
touch: /host/afile: Permission denied
/ # echo "hello from $(id) in $(hostname)" >> /shared/afile
/ # exit
# back in the host shell
# ls -la /tmp/shared/afile
-rw-r--r--. 1 5000 5000 157 Apr 16 00:13 /tmp/shared/afile
# cat /tmp/shared/afile ❹
hello from uid=0(root) gid=0(root)
      groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(
      dialout),26(tape),27(video) in d3b497ac0d34 ❺
```

❸ The /host mount is owned by host's uid and gid: 0:0, so disallowed
❹ /tmp/shared is owned by host's non-privileged uid and gid: 5000:5000, so write allowed
❺ uid for 'root' in container was 0

This example demonstrates the implications on filesystem access when using user namespaces with containers. User namespaces can be very useful in tightening security of applications that run or share data between containers as a privileged user. User namespace remapping can be disabled on a per container basis when creating or running the container, making it easier to make it the default execution mode. Note that user namespaces are incompatible with some optional features such as SELinux or using a privileged container. Consult the Security documentation on the Docker website for further details in designing and implementing a Docker configuration leveraging user namespace remapping that supports your use cases.

6.4 Adjusting OS feature access with capabilities

Docker can adjust a container's authorization to use individual operating system features. In Linux these feature authorizations are called *capabilities*, but as native support expands to other operating systems, other back-end implementations would need to be provided. Whenever a process attempts to make a gated system call such as opening a network socket, the capabilities of that process are checked for the required capability. The call will succeed if the process has the required capability and fail otherwise.

When you create a new container, Docker drops all capabilities except for a whitelist of capabilities necessary and safe to run most applications. This is done to further isolate the running process from the administrative functions of the operating system. A sample of the 37 dropped capabilities follows, and you might be able to guess at the reason for their removal:

- *SYS_MODULE*—Insert/remove kernel modules
- *SYS_RAWIO.*—Modify kernel memory
- *SYS_NICE.*—Modify priority of processes
- *SYS_RESOURCE.*—Override resource limits
- *SYS_TIME.*—Modify the system clock
- *AUDIT_CONTROL.*—Configure audit subsystem
- *MAC_ADMIN.*—Configure MAC configuration
- *SYSLOG.*—Modify kernel print behavior
- *NET_ADMIN.*—Configure the network
- *SYS_ADMIN.*—Catchall for administrative functions

The default set of capabilities provided to Docker containers provides a reasonable feature reduction, but there will be times when you need to add or reduce this set further. For example, the capability NET_RAW can be dangerous. If you wanted to be a bit more careful than the default configuration, you could drop NET_RAW from the list of capabilities. You can drop capabilities from a container using the `--cap-drop` flag. on `docker container create` or `.docker container run`.

```
docker container run --rm -u nobody \
    ubuntu:latest \
    /bin/bash -c "capsh --print | grep net_raw"

docker container run --rm -u nobody \
    --cap-drop net_raw \                              ❶
    ubuntu:latest \
    /bin/bash -c "capsh --print | grep net_raw"
```

❶ **Drop NET_RAW capability**

In Linux documentation you'll often see capabilities named in all uppercase and prefixed with CAP_, but that prefix won't work if provided to the capability-management options. Use unprefixed and lowercase names for the best results.

Similar to the `--cap-drop` flag, the `--cap-add` flag will add capabilities. If you needed to add the SYS_ADMIN capability for some reason, you'd use a command like the following:

```
docker container run --rm -u nobody \
    ubuntu:latest \
    /bin/bash –c "capsh --print | grep sys_admin"     ❶

docker container run --rm -u nobody \
    --cap-add sys_admin \                             ❷
    ubuntu:latest \
    /bin/bash -c "capsh --print | grep sys_admin"
```

❶ **SYS_ADMIN is not included**
❷ **Add SYS_ADMIN**

Like other container-creation options, both `--cap-add` and `--cap-drop` can be specified multiple times to add or drop multiple capabilities. These flags can be used to build containers that will let a process perform exactly and only what is required for proper operation. For example, you might be able to run a network management daemon as the nobody user and give it the NET_ADMIN capability instead of running it as root directly on the host or as a privileged container. If you are wondering if any capabilities were added or dropped from a container, you can inspect the container and print the `.HostConfig.CapAdd` and `.HostConfig.CapDrop` members of the output.

6.5 Running a container with full privileges

In those cases when you need to run a system administration task inside a container, you can grant that container privileged access to your computer. Privileged containers maintain their file system and network isolation but have full access to shared memory and devices and possess full system capabilities. You can perform several interesting tasks, like running Docker inside a container, with privileged containers.

The bulk of the uses for privileged containers is administrative. Take, for example, an environment where the root file system is read-only, or installing software outside a container has been disallowed, or you have no direct access to a shell on the host. If you wanted to run a program to tune the operating system (for something like load balancing) and you had access to run a container on that host, then you could simply run that program in a privileged container.

If you find a situation that can be solved only with the reduced isolation of a privileged container, use the `--privileged` flag on `docker container create` or `docker container run` to enable this mode:

```
docker container run --rm \
    --privileged \
    ubuntu:latest id                  ❶


docker container run --rm \
    --privileged \
    ubuntu:latest capsh —print        ❷

docker container run --rm \
    --privileged \
    ubuntu:latest ls /dev             ❸

docker container run --rm \
    --privileged \
    ubuntu:latest ifconfig            ❹
```

❶ **Check out our IDs**
❷ **Check out our Linux capabilities**
❸ **Check out list of mounted devices**
❹ **Examine network configuration**

Privileged containers are still partially isolated. For example, the network namespace will still be in effect. If you need to tear down that namespace, you'll need to combine this with `--net host` as well.

6.6 Stronger containers with enhanced tools

Docker uses reasonable defaults and a "batteries included" toolset to ease adoption and promote best practices. Most modern Linux kernels enable seccomp and Docker's default seccomp profile blocks over 40 syscalls that most programs do not need. You can enhance the containers Docker builds if you bring additional tools. Tools you can use to harden your containers include custom seccomp profiles, AppArmor, and SELinux.

Whole books have been written about each of these tools. They bring their own nuances, benefits, and required skillsets. Their use can be more than worth the effort. Support for each varies by Linux distribution, so you may be in for a bit of work. But once you've adjusted your host configuration, the Docker integration is simpler.

Security research

The information security space is very complicated and constantly evolving. It's easy to feel overwhelmed when reading through open conversations between InfoSec -professionals. These are often highly skilled people with long memories and very -different contexts from developers or general users. If you can take any one thing away from open InfoSec conversations, it is that balancing system security with user needs -is complex.

One of the best things you can do if you're new to this space is start with articles, papers, blogs, and books before you jump into conversations. This will give you an opportunity to digest one perspective and gain some deeper insight before switching to thought from a different perspective. When you've had an opportunity to form your own insight and opinions, these conversations become much more valuable.

It's very difficult to read one paper or learn one thing and know the best possible way to build a hardened solution. Whatever your situation, the system will evolve to include improvements from several sources. So the best thing you can do is take each tool and learn it by itself. Don't be intimidated by the depth some tools require for a strong understanding. The effort will be worth the result, and you'll understand the systems you use much better for it.

Docker isn't a perfect solution. Some would argue that it's not even a security tool. But what improvements it provides are far better than the alternative where people forego any isolation due to perceived cost. If you've read this far, maybe you'd be willing to go further with these auxiliary topics.

6.6.1 Specifying additional security options

Docker provides a single `--security-opt` flag for specifying options that configure Linux's seccomp and Linux Security Modules (LSM) features. Security options can be provided to the `docker container run` and `docker container create` commands. This flag can be set multiple times to pass multiple values.

Seccomp configures which Linux system calls a process may invoke. Docker's default seccomp profile blocks blocks all syscalls by default and then whitelists 260+ syscalls safe for use by most programs. The 44 blocked system calls are unneeded or are unsafe for normal

programs (e.g. `unshare`, used in creating new namespaces) or cannot be namespaced (e.g. `clock_settime`, sets the machine's time). Changing Docker's default seccomp profile is not recommended. If the default seccomp profile is too restrictive or permissive, a custom profile can be specified as a security option:

```
docker container run --rm -it \
    --security-opt seccomp=<FULL_PATH_TO_PROFILE> \
    ubuntu:latest sh
```

<FULL_PATH_TO_PROFILE> is the full path to a seccomp profile defining the allowed syscalls for the container. The Moby project on GitHub contains docker's default seccomp profile at `profiles/seccomp/default.json` that can be used as a starting point for a custom profile. Use the special value `unconfined` to disable use of seccomp for the container.

Linux Security Modules is a framework Linux adopted to act as an interface layer between the operating system and security providers. AppArmor and SELinux are both LSM providers. They both provide mandatory access control (MAC—the system defines access rules) and replace the standard Linux discretionary access control (file owners define access rules).

The LSM security option values are specified in one of six formats:

- To set a SELinux user label, use the form `label=user:<USERNAME>` where `<USERNAME>` is the name of the user you want to use for the label.
- To set a SELinux role label, use the form `label=role:<ROLE>` where `<ROLE>` is the name of the role you want to apply to processes in the container.
- To set a SELinux type label, use the form `label=type:<TYPE>` where `<TYPE>` is the type name of the processes in the container.
- To set a SELinux level label, use the form `label:level:<LEVEL>` where `<LEVEL>` is the level where processes in the container should run. Levels are specified as low-high pairs. Where abbreviated to the low level only, SELinux will interpret the range as single level.
- To disable SELinux label confinement for a container, use the form `label=disable`.
- To apply an AppArmor profile on the container, use the form `apparmor=<PROFILE>` where `<PROFILE>` is the name of the AppArmor profile to use.

As you can guess from these options, SELinux is a labeling system. A set of labels, called a *context*, is applied to every file and system object. A similar set of labels is applied to every user and process. At runtime when a process attempts to interact with a file or system resource, the sets of labels are evaluated against a set of allowed rules. The result of that evaluation determines whether the interaction is allowed or blocked.

The last option will set an AppArmor profile. AppArmor is frequently substituted for SELinux because it works with file paths instead of labels and has a training mode that you can use to passively build profiles based on observed application behavior. These differences are often cited as reasons why AppArmor is easier to adopt and maintain.

Free and commercial tools that monitor a program's execution and generate custom profiles tailored for applications are available. These tools help operators use information from actual program behavior in test and production environments to create a profile that works.

6.7 Build use-case-appropriate containers

Containers are a cross-cutting concern. There are more reasons and ways that people could use them than I could ever enumerate. So it's important, when you use Docker to build containers to serve your own purposes, that you take the time to do so in a way that's appropriate for the software you're running.

The most secure tactic for doing so would be to start with the most isolated container you can build and justify reasons for weakening those restrictions. In reality, people tend to be a bit more reactive than proactive. For that reason I think Docker hits a sweet spot with the default container construction. It provides reasonable defaults without hindering the productivity of users.

Docker containers are not the most isolated by default. Docker does not require that you enhance those defaults. It will let you do silly things in production if you want to. This makes Docker seem much more like a tool than a burden and something people generally want to use rather than feel like they have to use. For those who would rather not do silly things in production, Docker provides a simple interface to enhance container isolation.

6.7.1 Applications

Applications are the whole reason we use computers. Most applications are programs that other people wrote and work with potentially malicious data. Consider your web browser.

A web browser is a type of application that's installed on almost every computer. It interacts with web pages, images, scripts, embedded video, Flash documents, Java applications, and anything else out there. You certainly didn't create all that content, and most people were not contributors on web browser projects. How can you trust your web browser to handle all that content correctly?

Some more cavalier readers might just ignore the problem. After all, what's the worst thing that could happen? Well, if an attacker gains control of your web browser (or other application), they will gain all the capabilities of that application and the permissions of the user it's running as. They could trash your computer, delete your files, install other malware, or even launch attacks against other computers from yours. So, this isn't a good thing to ignore. The question remains: how do you protect yourself when this is a risk you need to take?

The best approach is to isolate the risk of running the program. First, make sure the application is running as a user with limited permissions. That way, if there's a problem, it won't be able to change the files on your computer. Second, limit the system capabilities of the browser. In doing so, you make sure your system configuration is safer. Third, set limits on how much of the CPU and memory the application can use. Limits help reserve resources to

keep the system responsive. Finally, it's a good idea to specifically whitelist devices that it can access. That will keep snoops off your webcam, USB, and the like.

6.7.2 High-level system services

High-level system services are a bit different from applications. They're not part of the operating system, but your computer makes sure they're started and kept running. These tools typically sit alongside applications outside the operating system, but they often require privileged access to the operating system to operate correctly. They provide important functionality to users and other software on a system. Examples include `cron`, `syslogd`, `dnsmasq`, `sshd`, and `docker`.

If you're unfamiliar with these tools (hopefully not all of them), it's all right. They do things like keep system logs, run scheduled commands, and provide a way to get a secure shell on the system from the network, and `docker` manages containers.

Although running services as root is common, few of them actually need full privileged access. Consider containerizing services and use capabilities to tune their access for the specific features they need.

6.7.3 Low-level system services

Low-level services control things like devices or the system's network stack. They require privileged access to the components of the system they provide (for example, firewall software needs administrative access to the network stack).

It's rare to see these run inside containers. Tasks such as file-system management, device management, and network management are core host concerns. Most software run in containers is expected to be portable. So machine-specific tasks like these are a poor fit for general container use cases.

The best exceptions are short-running configuration containers. For example, in an environment where all deployments happen with Docker images and containers, you'd want to push network stack changes in the same way you push software. In this case, you might push an image with the configuration to the host and make the changes with a privileged container. The risk in this case is reduced because you authored the configuration to be pushed, the container is not long running, and changes like these are simple to audit.

6.8 Summary

This chapter introduced the isolation features provided by Linux and talked about how Docker uses those to build configurable containers. With this knowledge, you will be able to customize that container isolation and use Docker for any use case. The following points were covered in this chapter:

- Docker uses cgroups, which let a user set memory limits, CPU weight, limits, and core restrictions as well as restrict access to specific devices.

- Docker containers each have their own IPC namespace that can be shared with other containers or the host in order to facilitate communication over shared memory.
- Docker supports isolating the USR namespace. By default, user and group IDs inside a container are equivalent to the same IDs on the host machine. When the user namespace is enabled user and group IDs in the container are remapped to IDs that do not exist on the host.
- You can and should use the `-u` option on `docker container run` and `docker container create` to run containers as non-root users.
- Avoid running containers in privileged mode whenever possible.
- Linux capabilities provide operating system feature authorization. Docker drops certain capabilities in order to provide reasonably isolating defaults.
- The capabilities granted to any container can be set with the `--cap-add` and `--cap-drop` flags.
- Docker provides tooling for integrating easily with enhanced isolation technologies like seccomp, SELinux, and AppArmor. These are powerful tools that security conscious Docker adopters should investigate.

Part 2

Packaging Software for Distribution

Inevitably a Docker user will need to create an image. There are times when the software you need is not packaged in an image. Other times you will need a feature that has not been enabled in an available image. The four chapters in this part will help you understand how to originate, customize, and specialize the images you intend to deploy or share using Docker.

7

Packaging software in images

This chapter covers

- Manual image construction and practices
- Images from a packaging perspective
- Working with flat images
- Image versioning best practices

The goal of this chapter is to help you understand the concerns of image design, learn the tools for building images, and discover advanced image patterns. You will accomplish these things by working through a thorough real-world example. Before getting started, you should have a firm grasp on the concepts in part 1 of this book.

You can create a Docker image by either modifying an existing image inside a container or defining and executing a build script called a Dockerfile. This chapter focuses on the process of manually changing an image, the fundamental mechanics of image manipulation, and the artifacts that are produced. Dockerfiles and build automation are covered in chapter 8.

7.1 Building Docker images from a container

It's easy to get started building images if you're already familiar with using containers. Remember, a union file system (UFS) mount provides a container's file system. Any changes that you make to the file system inside a container will be written as new layers that are owned by the container that created them.

Before you work with real software, the next section details the typical workflow with a Hello World example.

7.1.1 Packaging Hello World

The basic workflow for building an image from a container includes three steps. First, you need to create a container from an existing image. You will choose the image based on what you want to be included with the new finished image and the tools you will need to make the changes.

The second step is to modify the file system of the container. These changes will be written to a new layer on the union file system for the container. We'll revisit the relationship between images, layers, and repositories later in this chapter.

Once the changes have been made, the last step is to commit those changes. Once the changes are committed, you'll be able to create new containers from the resulting image. Figure 7.1 illustrates this workflow.

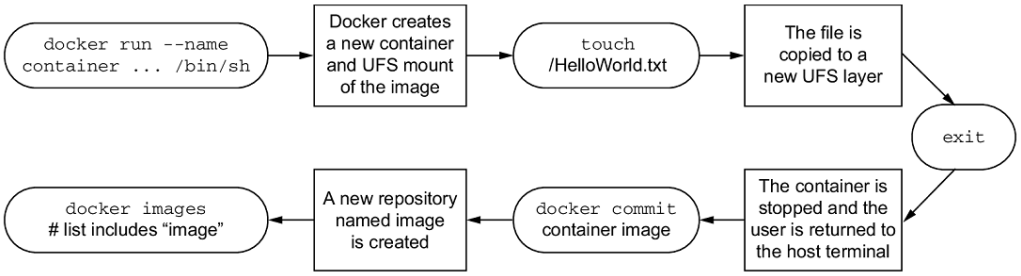


Figure 7.1 Building an image from a container

With these steps in mind, work through the following commands to create a new image named hw_image.

```
docker container run --name hw_container \
    ubuntu:latest \
    touch /HelloWorld                    ❶

docker container commit hw_container hw_image      ❷

docker container rm -vf hw_container               ❸

docker container run --rm \
    hw_image \
    ls -l /HelloWorld                    ❹
```

❶ Modify file in container
❷ Commit change to new image
❸ Remove changed container
❹ Examine file in new container

If that seems stunningly simple, you should know that it does become a bit more nuanced as the images you produce become more sophisticated, but the basic steps will always be the same.

Now that you have an idea of the workflow, you should try to build a new image with real software. In this case, you'll be packaging a program called Git.

7.1.2 Preparing packaging for Git

Git is a popular, distributed version-control tool. Whole books have been written about the topic. If you're unfamiliar with it, I recommend that you spend some time learning how to use Git. At the moment, though, you only need to know that it's a program you're going to install onto an Ubuntu image.

To get started building your own image, the first thing you'll need is a container created from an appropriate base image:

```
docker container run -it --name image-dev ubuntu:latest /bin/bash
```

This will start a new container running the bash shell. From this prompt, you can issue commands to customize your container. Ubuntu ships with a Linux tool for software installation called `apt-get`. This will come in handy for acquiring the software that you want to package in a Docker image. You should now have an interactive shell running with your container. Next, you need to install Git in the container. Do that by running the following commands:

```
apt-get update
apt-get –y install git
```

This will tell APT to download and install Git and all its dependencies on the container's file system. When it's finished, you can test the installation by running the `git` program:

```
git version
# Output something like:
# git version 2.7.4
```

Package tools like `apt-get` make installing and uninstalling software easier than if you had to do everything by hand. But they provide no isolation to that software and dependency conflicts often occur. You can be sure that other software you install outside this container won't impact the version of Git you have installed in this container.

Now that Git has been installed on your Ubuntu container, you can simply exit the container:

```
exit
```

The container should be stopped but still present on your computer. Git has been installed in a new layer on top of the ubuntu:latest image. If you were to walk away from this example right now and return a few days later, how would you know exactly what changes were made?

When you're packaging software, it's often useful to review the list of files that have been modified in a container, and Docker has a command for that.

7.1.3 Reviewing file system changes

Docker has a command that shows you all the file-system changes that have been made inside a container. These changes include added, changed, or deleted files and directories. To review the changes that you made when you used APT to install Git, run the `diff` subcommand:

```
docker container diff image-dev          ❶
```

❶ # Outputs a LONG list of file changes…

Lines that start with an `A` are files that were added. Those starting with a `C` were changed. Finally, those with a `D` were deleted. Installing Git with APT in this way made several changes. For that reason, it might be better to see this at work with a few specific examples:

```
docker container run --name tweak-a busybox:latest touch /HelloWorld    ❶
docker container diff tweak-a
# Output:
#    A /HelloWorld

docker container run --name tweak-d busybox:latest rm /bin/vi           ❷
docker container diff tweak-d
# Output:
#    C /bin
#    D /bin/vi

docker container run --name tweak-c busybox:latest touch /bin/vi        ❸
docker container diff tweak-c
# Output:
#    C /bin
#    C /bin/busybox
```

❶ Add new file to busybox
❷ Remove existing file from busybox
❸ Change existing file in busybox

Always remember to clean up your workspace, like this:

```
docker container rm -vf tweak-a
docker container rm -vf tweak-d
docker container rm -vf tweak-c
```

Now that you've seen the changes you've made to the file system, you're ready to commit the changes to a new image. As with most other things, this involves a single -command that does several things.

7.1.4 Committing a new image

You use the `docker container commit` command to create an image from a modified container. It's a best practice to use the `-a` flag that signs the image with an author string. You should also always use the `-m` flag, which sets a commit message. Create and sign a new image that you'll name ubuntu-git from the `image-dev` container where you installed Git:

```
docker container commit -a "@dockerinaction" -m "Added git" image-dev ubuntu-git
# Outputs a new unique image identifier like:
# bbf1d5d430cdf541a72ad74dfa54f6faec41d2c1e4200778e9d4302035e5d143
```

Once you've committed the image, it should show up in the list of images installed on your computer. Running `docker images` should include a line like this:

```
REPOSITORY    TAG       IMAGE ID      CREATED         VIRTUAL SIZE
ubuntu-git    latest    bbf1d5d430cd  5 seconds ago   248 MB
```

Make sure it works by testing Git in a container created from that image:

```
docker container run --rm ubuntu-git git version
```

Now you've created a new image based on an Ubuntu image and installed Git. That's a great start, but what do you think will happen if you omit the command override? Try it to find out:

```
docker container run --rm ubuntu-git
```

Nothing appears to happen when you run that command. That's because the command you started the original container with was committed with the new image. The command you used to start the container that the image was created by was `/bin/bash`. When you create a container from this image using the default command, it will start a shell and immediately exit. That's not a terribly useful default command.

I doubt that any users of an image named ubuntu-git would expect that they'd need to manually invoke Git each time. It would be better to set an entrypoint on the image to `git`. An entrypoint is the program that will be executed when the container starts. If the entrypoint isn't set, the default command will be executed directly. If the entrypoint is set, the default command and its arguments will be passed to the entrypoint as arguments.

To set the entrypoint, you'll need to create a new container with the `--entrypoint` flag set and create a new image from that container:

```
docker container run --name cmd-git --entrypoint git ubuntu-git      (1)

docker container commit -m "Set CMD git" \
    -a "@dockerinaction" cmd-git ubuntu-git                (2)

docker container rm -vf cmd-git                                       (3)

docker container run --name cmd-git ubuntu-git version                (4)
```

(1) Show standard git help and exit
(2) Commit new image to same name

❸ Cleanup
❹ Test

Now that the entrypoint has been set to `git`, users no longer need to type the command at the end. This might seem like a marginal savings with this example, but many tools people use are not as succinct. Setting the entrypoint is just one thing you can do to make images easier for people to use and integrate into their projects.

7.1.5 Configurable image attributes

When you use `docker container commit`, you commit a new layer to an image. The file-system snapshot isn't the only thing included with this commit. Each layer also includes metadata describing the execution context. Of the parameters that can be set when a container is created, all the following will carry forward with an image created from the container:

- All environment variables
- The working directory
- The set of exposed ports
- All volume definitions
- The container entrypoint
- Command and arguments

If these values weren't specifically set for the container, the values will be inherited from the original image. Part 1 of this book covers each of these, so I won't reintroduce them here. But it may be valuable to examine two detailed examples. First, consider a container that introduces two environment variable specializations:

```
docker container run --name rich-image-example \
    -e ENV_EXAMPLE1=Rich -e ENV_EXAMPLE2=Example \      ❶
    busybox:latest

docker container commit rich-image-example rie                    ❷

docker container run --rm rie \
    /bin/sh -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2"     ❸
```

❶ Create environment variable specialization
❷ Commit image
❸ Outputs: Rich Example

Next, consider a container that introduces an entrypoint and command specialization as a new layer on top of the previous example:

```
docker container run --name rich-image-example-2 \
    --entrypoint "/bin/sh" \                        ❶
    rie \
    -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2"         ❷
```

```
docker container commit rich-image-example-2 rie          ❸

docker container run --rm rie                             ❹
```

❶ Set default entrypoint
❷ Set default command
❸ Commit image
❹ Different command with same output

This example builds two additional layers on top of BusyBox. In neither case are files changed, but the behavior changes because the context metadata has been altered. These changes include two new environment variables in the first new layer. Those environment variables are clearly inherited by the second new layer, which sets the entrypoint and default command to display their values. The last command uses the final image without specifying any alternative behavior, but it's clear that the previous defined behavior has been inherited.

Now that you understand how to modify an image, take the time to dive deeper into the mechanics of images and layers. Doing so will help you produce high-quality images in real-world situations.

7.2 Going deep on Docker images and layers

By this point in the chapter, you've built a few images. In those examples you started by creating a container from an image like ubuntu:latest or busybox:latest. Then you made changes to the file system or context within that container. Finally, everything seemed to just work when you used the `docker container commit` command to create a new image. Understanding how the container's file system works and what the `docker container commit` command actually does will help you become a better image author. This section dives into that subject and demonstrates the impact to authors.

7.2.1 An exploration of union file systems

Understanding the details of union file systems (UFS) is important for image authors for two reasons:

- Authors need to know the impact that adding, changing, and deleting files have on resulting images.
- Authors need have a solid understanding of the relationship between layers and how layers relate to images, repositories, and tags.

Start by considering a simple example. Suppose you want to make a single change to an existing image. In this case the image is ubuntu:latest, and you want to add a file named mychange to the root directory. You should use the following command to do this:

```
docker container run --name mod_ubuntu ubuntu:latest touch /mychange
```

The resulting container (named `mod_ubuntu`) will be stopped but will have written that single change to its file system. As discussed in chapters 3 and 4, the root file system is provided by the image that the container was started from. That file system is implemented with something called a union file system.

A union file system is made up of layers. Each time a change is made to a union file system, that change is recorded on a new layer on top of all of the others. The "union" of all of those layers, or top-down view, is what the container (and user) sees when accessing the file system. Figure 7.2 illustrates the two perspectives for this example.

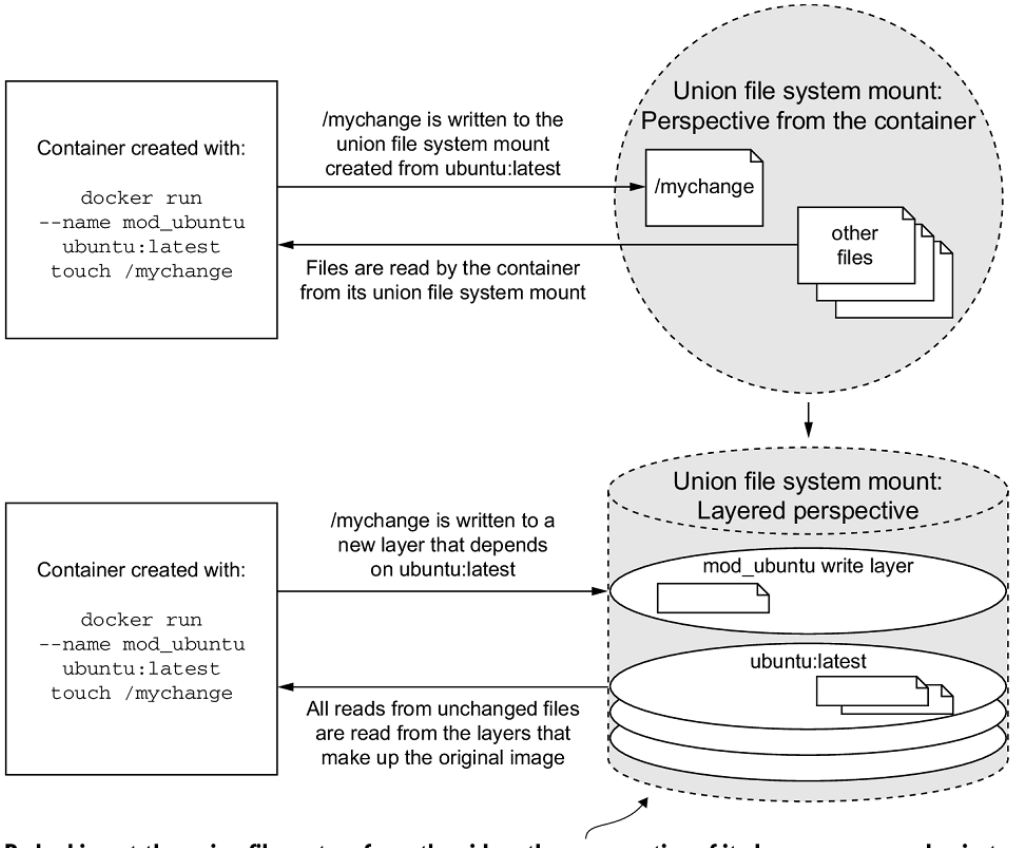


Figure 7.2 A simple file write example on a union file system from two perspectives

When you read a file from a union file system, that file will be read from the top-most layer where it exists. If a file was not created or changed on the top layer, the read will fall through the layers until it reaches a layer where that file does exist. This is illustrated in figure 7.3.

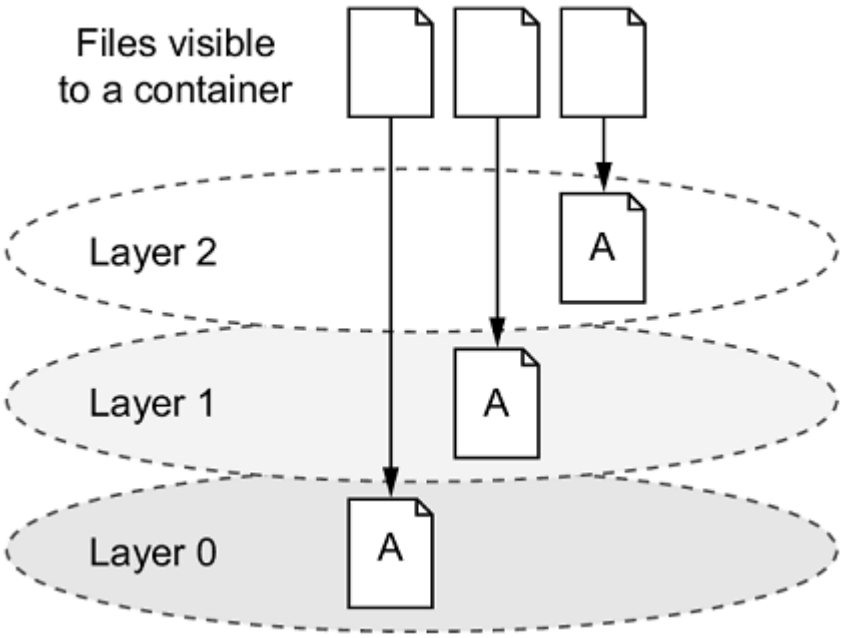


Figure 7.3 Reading files that are located on different layers

All this layer functionality is hidden by the union file system. No special actions need to be taken by the software running in a container to take advantage of these features. Understanding layers where files were added covers one of three types of file system writes. The other two are deletions and file changes.

Like additions, both file changes and deletions work by modifying the top layer. When a file is deleted, a delete record is written to the top layer, which hides any versions of that file on lower layers. When a file is changed, that change is written to the top layer, which again hides any versions of that file on lower layers. The changes made to the file system of a container are listed with the `docker container diff` command you used earlier in the chapter:

```
docker container diff mod_ubuntu
```

This command will produce the output:

```
A /mychange
```

The `A` in this case indicates that the file was added. Run the next two commands to see how a file deletion is recorded:

```
docker container run --name mod_busybox_delete busybox:latest rm /etc/passwd
docker container diff mod_busybox_delete
```

This time the output will have two rows:

```
C /etc
D /etc/passwd
```

The `D` indicates a deletion, but this time the parent folder of the file was also included. The `C` indicates that it was changed. The next two commands demonstrate a file change:

```
docker container run --name mod_busybox_change busybox:latest touch /etc/passwd
docker container diff mod_busybox_change
```

The diff subcommand will show two changes:

```
C /etc
C /etc/passwd
```

Again, the `C` indicates a change, and the two items are the file and the folder where it's located. If a file nested five levels deep were changed, there would be a line for each level of the tree. Changes to filesystem attributes such as file ownership and permissions are recorded in the same way as changes to files. Be careful when modifying file system attributes on large numbers of files as those files will likely be copied into the layer performing the change. File-change mechanics are the most important thing to understand about union file systems and we will examine that a little deeper next.

Most union file systems use something called copy-on-write, which is easier to understand if you think of it as copy-on-change. When a file in a read-only layer (not the top layer) is modified, the whole file is first copied from the read-only layer into the writable layer before the change is made. This has a negative impact on runtime performance and image size. Section 7.2.3 covers the way this should influence your image design.

Take a moment to solidify your understanding of the system by examining how the more comprehensive set of scenarios is illustrated in figure 7.4. In this illustration files are added, changed, deleted, and added again over a range of three layers.

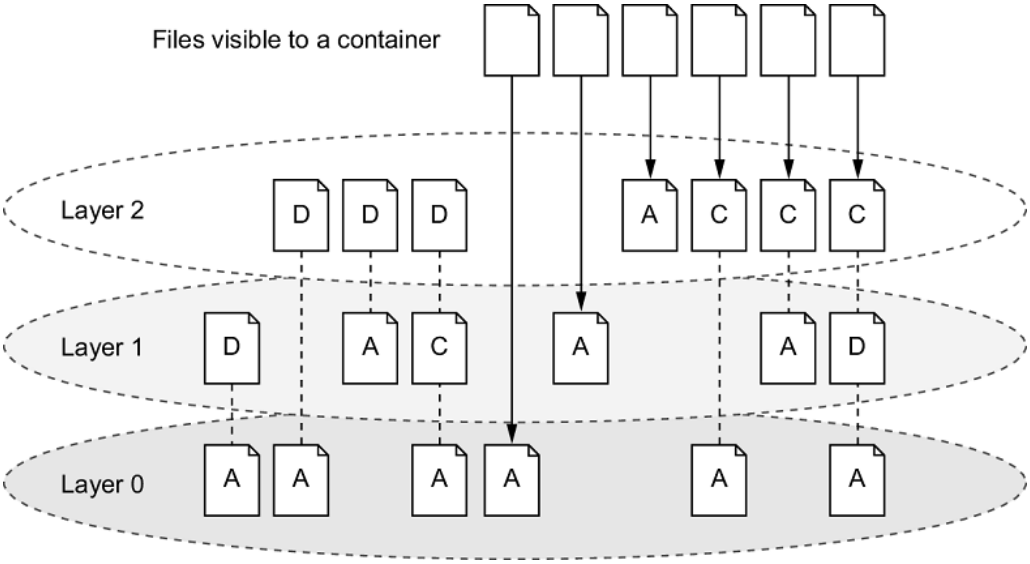


Figure 7.4 Various file addition, change, and deletion combinations over a three-layered image

Knowing how file system changes are recorded, you can begin to understand what happens when you use the `docker container commit` command to create a new image.

7.2.2 Reintroducing images, layers, repositories, and tags

You've created an image using the `docker container commit` command, and you understand that it commits the top-layer changes to an image. But we've yet to define *commit*.

Remember, a union file system is made up of a stack of layers where new layers are added to the top of the stack. Those layers are stored separately as collections of the changes made in that layer and metadata for that layer. When you commit a container's changes to its file system, you're saving a copy of that top layer in an identifiable way.

When you commit the layer, a new ID is generated for it, and copies of all the file changes are saved. Exactly how this happens depends on the storage engine that's being used on your system. It's less important for you to understand the details than it is for you to understand the general approach. The metadata for a layer includes that generated identifier, the identifier of the layer below it (parent), and the execution context of the container that the layer was created from. Layer identities and metadata form the graph that Docker and the UFS use to construct images.

An image is the stack of layers that you get by starting with a given top layer and then following all the links defined by the parent ID in each layer's metadata, as shown in figure 7.5.

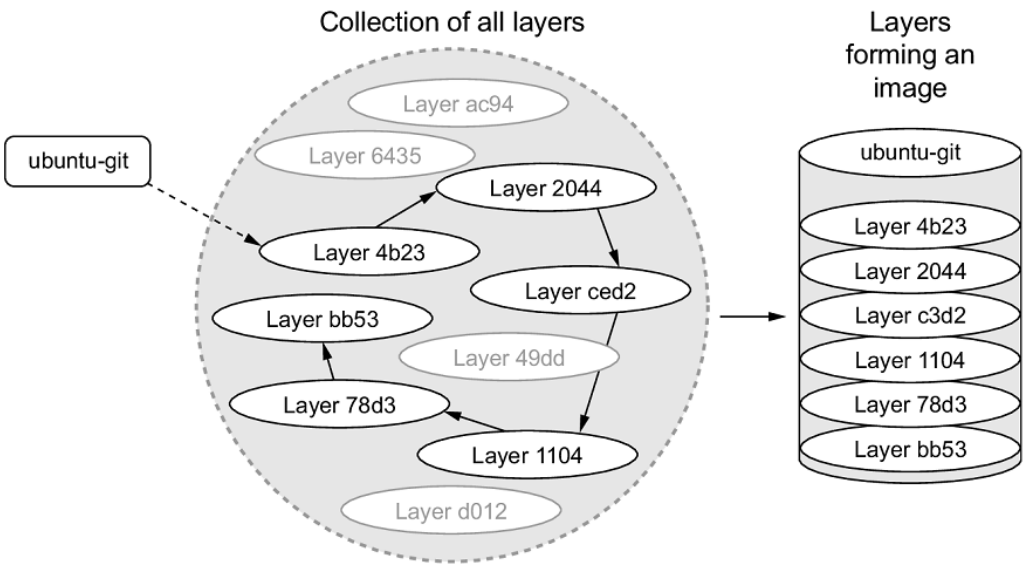


Figure 7.5 An image is the collection of layers produced by traversing the parent graph from a top layer.

Images are stacks of layers constructed by traversing the layer dependency graph from some starting layer. The layer that the traversal starts from is the top of the stack. This means that a layer's ID is also the ID of the image that it and its dependencies form. Take a moment to see this in action by committing the `mod_ubuntu` container you created earlier:

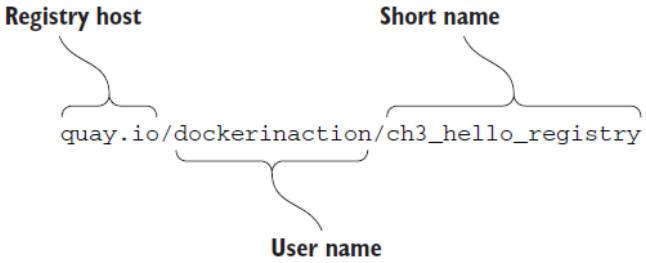
```
docker container commit mod_ubuntu
```

That commit subcommand will generate output that includes a new image ID like this:

```
6528255cda2f9774a11a6b82be46c86a66b5feff913f5bb3e09536a54b08234d
```

You can create a new container from this image using the image ID as it's presented to you. Like containers, layer IDs are large hexadecimal numbers that can be difficult for a person to work with directly. For that reason, Docker provides repositories.

In chapter 3, a *repository* is roughly defined as a named bucket of images. More specifically, repositories are location/name pairs that point to a set of specific layer -identifiers. Each repository contains at least one tag that points to a specific layer identifier and thus the image definition. Let's revisit the example used in chapter 3:



This repository is located in the registry hosted at quay.io. It's named for the user (dockerinaction) and a unique short name (ch3_hello_registry). Pulling this repository would pull all the images defined for each tag in the repository. In this example, there's only one tag, `latest`. That tag points to a layer with the short form ID 07c0f84777ef, as illustrated in figure 7.6.

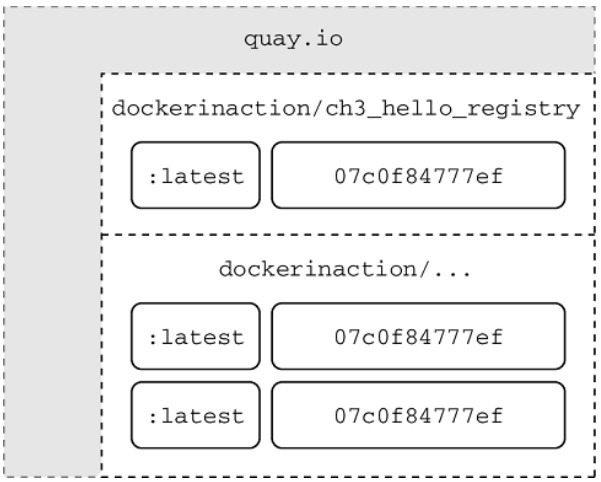


Figure 7.6 A visual representation of repositories

Repositories and tags are created with the `docker tag`, `docker container commit`, or `docker build` commands. Revisit the `mod_ubuntu` container again and put it into a repository with a tag:

```
docker container commit mod_ubuntu myuser/myfirstrepo:mytag
# Outputs:
# 82ec7d2c57952bf57ab1ffdf40d5374c4c68228e3e923633734e68a11f9a2b59
```

The generated ID that's displayed will be different because another copy of the layer was created. With this new friendly name, creating containers from your images requires little effort. If you want to copy an image, you only need to create a new tag or repository from the

existing one. You can do that with the `docker tag` command. Every repository contains a "latest" tag by default. That will be used if the tag is omitted like in the previous command:

```
docker tag myuser/myfirstrepo:mytag myuser/mod_ubuntu
```

By this point you should have a strong understanding of basic UFS fundamentals as well as how Docker creates and manages layers, images, and repositories. With these in mind, let's consider how they might impact image design.

All layers below the writable layer created for a container are immutable, meaning they can never be modified. This property makes it possible to share access to images instead of creating independent copies for every container. It also makes individual layers highly reusable. The other side of this property is that anytime you make changes to an image, you need to add a new layer, and old layers are never removed. Knowing that images will inevitably need to change, you need to be aware of any image limitations and keep in mind how changes impact image size.

7.2.3 Managing image size and layer limits

If images evolved in the same way that most people manage their file systems, Docker images would quickly become unusable. For example, suppose you wanted to make a different version of the ubuntu-git image you created earlier in this chapter. It may seem natural to modify that ubuntu-git image. Before you do, create a new tag for your ubuntu-git image. You'll be reassigning the latest tag:

```
docker image tag ubuntu-git:latest ubuntu-git:2.7        (1)
```

(1) Create new tag: 2.7

The first thing you'll do in building your new image is remove the version of Git you installed:

```
docker container run --name image-dev2 \
        --entrypoint /bin/bash \                         (1)
    ubuntu-git:latest -c "apt-get remove -y git"         (2)

docker container commit image-dev2 ubuntu-git:removed    (3)

docker image tag ubuntu-git:removed ubuntu-git:latest    (4)

docker images                                            (5)
```

(1) Execute bash command
(2) Remove Git
(3) Commit image
(4) Reassign latest tag
(5) Examine image sizes

The image list and sizes reported will look something like the following:

```
REPOSITORY   TAG        IMAGE ID        CREATED           VIRTUAL SIZE
ubuntu-git   latest     826c66145a59    10 seconds ago    226.6 MB
```

```
ubuntu-git   removed    826c66145a59    10 seconds ago    226.6 MB
ubuntu-git   2.7        3e356394c14e    41 hours ago      226 MB
...
```

Notice that even though you removed Git, the image actually increased in size. Although you could examine the specific changes with `docker container diff`, you should be quick to realize that the reason for the increase has to do with the union file system.

Remember, UFS will mark a file as deleted by actually adding a file to the top layer. The original file and any copies that existed in other layers will still be present in the image. It's important to minimize image size for the sake of the people and systems that will be consuming your images. If you can avoid causing long download times and significant disk usage with smart image creation, then your consumers will benefit. In the early days of Docker, image authors sometimes minimized the number of layers in an image due to limits of image storage drivers. Modern Docker image storage drivers do not have image layer limits normal users will encounter, so design for other attributes like size and cacheability.

You can examine all the layers in an image using the `docker image history` command. It will display the following:

- Abbreviated layer ID
- Age of the layer
- Initial command of the creating container
- Total file size of that layer

By examining the history of the ubuntu-git:removed image, you can see that three layers have already been added on the top of the original ubuntu:latest image:

```
docker image history ubuntu-git:removed
```

Outputs are something like:

```
IMAGE          CREATED          CREATED BY                      SIZE
826c66145a59   24 minutes ago   /bin/bash -c apt-get remove     662 kB
3e356394c14e   42 hours ago     git                             0 B
bbf1d5d430cd   42 hours ago     /bin/bash                       37.68 MB
b39b81afc8ca   3 months ago     /bin/sh -c #(nop) CMD [/bin     0 B
615c102e2290   3 months ago     /bin/sh -c sed -i 's/^#\s*\     1.895 kB
837339b91538   3 months ago     /bin/sh -c echo '#!/bin/sh'     194.5 kB
53f858aaaf03   3 months ago     /bin/sh -c #(nop) ADD file:     188.1 MB
511136ea3c5a   22 months ago                                    0 B
```

You can flatten images if you export them and then reimport them with Docker. But that's a bad idea because you lose the change history as well as any savings customers might get when they download images with the same lower levels. Flattening images defeats the purpose. The smarter thing to do in this case is to create a branch.

Instead of fighting the layer system, you can solve both the size and layer growth problems by using the layer system to create branches. The layer system makes it trivial to go back in the history of an image and make a new branch. You are potentially creating a new branch every time you create a container from the same image.

In reconsidering your strategy for your new ubuntu-git image, you should simply start from ubuntu:latest again. With a fresh container from ubuntu:latest, you could install whatever version of Git you want. The result would be that both the original ubuntu-git image you created and the new one would share the same parent, and the new image wouldn't have any of the baggage of unrelated changes.

Branching increases the likelihood that you'll need to repeat steps that were accomplished in peer branches. Doing that work by hand is prone to error. Automating image builds with Dockerfiles is a better idea.

Occasionally the need arises to build a full image from scratch. Docker provides special handling for the `scratch` image that tells the build process to make the next command the first layer of the resulting image. This practice can be beneficial if your goal is to keep images small and if you're working with technologies that have few dependencies such as the Go or Rust programming languages. Other times you may want to flatten an image to trim an image's history. In either case, you need a way to import and export full file systems.

7.3 Exporting and importing flat file systems

On some occasions it's advantageous to build images by working with the files destined for an image outside the context of the union file system or a container. To fill this need, Docker provides two commands for exporting and importing archives of files.

The `docker container export` command will stream the full contents of the flattened union file system to stdout or an output file as a tarball. The result is a tarball that contains all the files from the container perspective. This can be useful if you need to use the file system that was shipped with an image outside the context of a container. You can use the `docker cp` command for this purpose, but if you need several files, exporting the full file system may be more direct.

Create a new container and use the `export` subcommand to get a flattened copy of its filesystem:

```
docker container create --name export-test \
   dockerinaction/ch7_packed:latest ./echo For Export    ❶

docker container export --output contents.tar export-test

docker container rm export-test

tar -tf contents.tar                                     ❷
```

❶ Export file system contents
❷ Show archive contents

This will produce a file in the current directory named contents.tar. That file should contain two files. At this point you could extract, examine, or change those files to whatever end. If you had omitted the `--output` (or `-o` for short), then the contents of the file system would be

streamed in tarball format to stdout. Streaming the contents to stdout makes the `export` command useful for chaining with other shell programs that work with tarballs.

The `docker import` command will stream the content of a tarball into a new image. The `import` command recognizes several compressed and uncompressed forms of tarballs. An optional Dockerfile instruction can also be applied during file-system import. Importing file systems is a simple way to get a complete minimum set of files into an image.

To see how useful this is, consider a statically linked Go version of Hello World. Create an empty folder and copy the following code into a new file named helloworld.go:

```
package main
import "fmt"
func main() {
        fmt.Println("hello, world!")
}
```

You may not have Go installed on your computer, but that's no problem for a Docker user. By running the next command, Docker will pull an image containing the Go compiler, compile and statically link the code (which means it can run all by itself), and place that program back into your folder:

```
docker container run --rm -v "$(pwd)":/usr/src/hello \
    -w /usr/src/hello golang:1.9 go build -v
```

If everything works correctly, you should have an executable program (binary file) in the same folder, named hello. Statically linked programs have no external file dependencies at runtime. That means this statically linked version of Hello World can run in a container with no other files. The next step is to put that program in a tarball:

```
tar -cf static_hello.tar hello
```

Now that the program has been packaged in a tarball, you can import it using the `docker import` command:

```
docker import -c "ENTRYPOINT [\"/hello\"]" - \
    dockerinaction/ch7_static < static_hello.tar      (1)
```

(1) **Tar file streamed via UNIX pipe**

In this command you use the `-c` flag to specify a Dockerfile command. The command you use sets the entrypoint for the new image. The exact syntax of the Dockerfile command is covered in chapter 8. The more interesting argument on this command is the hyphen (`-`) at the end of the first line. This hyphen indicates that the contents of the tarball will be streamed through stdin. You can specify a URL at this position if you're fetching the file from a remote web server instead of from your local file system.

You tagged the resulting image as the dockerinaction/ch7_static repository. Take a moment to explore the results:

```
docker container run dockerinaction/ch7_static                    (1)
```

```
docker history dockerinaction/ch7_static
```

❶ Outputs: hello, world!

You'll notice that the history for this image has only a single entry (and layer):

```
IMAGE           CREATED         CREATED BY     SIZE
edafbd4a0ac5    11 minutes ago                 1.824 MB
```

In this case, the image we produced was small for two reasons. First, the program we produced was only just over 1.8 MB, and we included no operating system files or support programs. This is a minimalistic image. Second, there's only one layer. There are no deleted or unused files carried with the image in lower layers. The downside to using single-layer (or flat) images is that your system won't benefit from layer reuse. That might not be a problem if all your images are small enough. But the overhead may be significant if you use larger stacks or languages that don't offer static linking.

There are trade-offs to every image design decision, including whether or not to use flat images. Regardless of the mechanism you use to build images, your users need a consistent and predictable way to identify different versions.

7.4 Versioning best practices

Pragmatic versioning practices help users make the best use of images. The goal of an effective versioning scheme is to communicate clearly and provide flexibility to image users.

It's generally insufficient to build or maintain only a single version of your software unless it's your first. If you're releasing the first version of your software, you should be mindful of your users' adoption experience from the beginning. Versions are important because they identify contracts your adopters depend on. Unexpected software changes cause problems for adopters and versions are one of the primary ways to signal software changes.

With Docker, the key to maintaining multiple versions of the same software is proper repository tagging. The understanding that every repository contains multiple tags and that multiple tags can reference the same image is at the core of a pragmatic tagging scheme.

The `docker image tag` command is unlike the other two commands that can be used to create tags. It's the only one that's applied to existing images. To understand how to use tags and how they impact the user adoption experience, consider the two tagging schemes for a repository shown in figure 7.7.

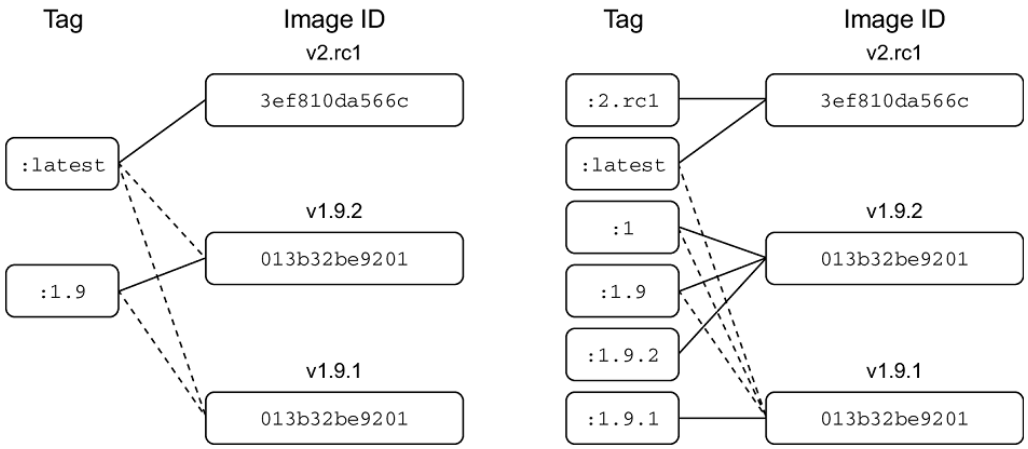


Figure 7.7 Two different tagging schemes (left and right) for the same repository with three images. Dotted lines represent old relationships between a tag and an image.

There are two problems with the tagging scheme on the left side of figure 7.7. First, it provides poor adoption flexibility. A user can choose to declare a dependency on `1.9` or `latest`. When a user adopts version 1.9 and that implementation is actually 1.9.1, they may develop dependencies on behavior defined by that build version. Without a way to explicitly depend on that build version, they will experience pain when 1.9 is updated to point to 1.9.2.

The best way to eliminate this problem is to define and tag versions at a level where users can depend on consistent contracts. This is not advocating a three-tiered versioning system. It means only that the smallest unit of the versioning system you use captures the smallest unit of contract iteration. By providing multiple tags at this level, you can let users decide how much version drift they want to accept.

Consider the right side of figure 7.7. A user who adopts version 1 will always use the highest minor and build version under that major version. Adopting 1.9 will always use the highest build version for that minor version. Adopters who need to carefully migrate between versions of their dependencies can do so with control and at times of their choosing.

The second problem is related to the `latest` tag. On the left, `latest` currently points to an image that's not otherwise tagged, and so an adopter has no way of knowing what version of the software that is. In this case, it's referring to a release candidate for the next major version of the software. An unsuspecting user may adopt the `-latest` tag with the impression that it's referring to the latest build of an otherwise tagged version.

There are other problems with the `latest` tag. It's adopted more frequently than it should be. This happens because it's the default tag. The impact is that a responsible repository maintainer should always make sure that its repository's `latest` refers to the latest stable build of its software instead of the true latest.

The last thing to keep in mind is that in the context of containers, you're versioning not only your software but also a snapshot of all of your software's packaged dependencies. For example, if you package software with a particular distribution of Linux, like Debian, then those additional packages become part of your image's interface contract. Your users will build tooling around your images and in some cases may come to depend on the presence of a particular shell or script in your image. If you suddenly rebase your software on something like CentOS but leave your software otherwise unchanged, your users will experience pain.

In situations where the software dependencies change, or the software needs to be distributed on top of multiple bases, then those dependencies should be included with your tagging scheme.

The Docker official repositories are ideal examples to follow. Consider this abbreviated tag list for the official golang repository, where each row represents a distinct image:

```
1.9,             1.9-stretch, 1.9.6
1.9-alpine
1,               1.10,        1.10.2,          latest,    stretch
1.10-alpine,     alpine
```

Users can determine that the latest version of Golang 1, 1.x, and 1.10 all currently point to version 1.10.2. A Golang user can select a tag that meets their needs for tracking changes in Golang or platform. If an adopter needs the latest image built on the debian:stretch platform, they can use the `stretch` tag. This scheme puts the control and responsibility for upgrades in the hands of your adopters.

7.5 Summary

This is the first chapter to cover the creation of Docker images, tag management, and other distribution concerns such as image size. Learning this material will help you build images and become a better consumer of images. The following are the key points in the chapter:

- New images are created when changes to a container are committed using the `docker container commit` command.
- When a container is committed, the configuration it was started with will be encoded into the configuration for the resulting image.
- An image is a stack of layers that's identified by its top layer.
- An image's size on disk is the sum of the sizes of its component layers.
- Images can be exported to and imported from a flat tarball representation using the `docker container export` and `docker image import` commands.
- The `docker image tag` command can be used to assign several tags to a single repository.
- Repository maintainers should keep pragmatic tags to ease user adoption and migration control.
- Tag your latest *stable* build with the `latest` tag.
- Provide fine-grained and overlapping tags so that adopters have control of the scope of

their dependency version creep.

8

Building images automatically with Dockerfiles

This chapter covers

- Automated packaging with Dockerfile
- Metadata instructions
- File system instructions
- Build arguments
- Multi-stage builds
- Packaging for multiprocess and durable containers
- Trusted base images
- Working with users
- Reducing the image attack surface

A Dockerfile is a text file that contains instructions for building an image. The Docker image builder executes the Dockerfile from top to bottom and the instructions can configure or change anything about an image. Building images from Dockerfiles makes tasks like adding files to a container from your computer simple one-line instructions. Dockerfiles are the most common way to describe how to build a Docker image. This section covers the basics of working with Dockerfile builds and the best reasons to use them, a lean overview of the instructions, and how to add future build behavior. We'll get started with a familiar example that shows how you can automate the process of building images with code instead of creating them manually. Once an image's build is defined in code, it is simple to track changes in version control, share with team members, optimize, and secure.

8.1 Packaging Git with a Dockerfile

Let's start by revisiting the Git example image we built by hand in Chapter 7. You should recognize many of the details and advantages of working with a Dockerfile as we translate the image build process from manual operations to code.

First, create a new directory and from that directory create a new file with your favorite text editor. Name the new file Dockerfile. Write the following five lines and then save the file:

```
# An example Dockerfile for installing Git on Ubuntu
FROM ubuntu:latest
LABEL maintainer="dia@allingeek.com"
RUN apt-get update && apt-get install -y git
ENTRYPOINT ["git"]
```

Before dissecting this example, build a new image from it with the `docker image build` command from the same directory containing the Dockerfile and tag the image with `auto`:

```
docker image build --tag ubuntu-git:auto .
```

Outputs several lines about steps and output from apt-get and will finally display a message like this:

```
Successfully built cc63aeb7a5a2
Successfully tagged ubuntu-git:auto
```

Running this command starts the build process. When it's completed, you should have a brand-new image that you can test. View the list of all your ubuntu-git images and test the newest one with this command:

```
docker image ls
```

The new build tagged "auto" should now appear in the list:

```
REPOSITORY   TAG        IMAGE ID        CREATED             VIRTUAL SIZE
ubuntu-git   auto       cc63aeb7a5a2     2 minutes ago      219MB
ubuntu-git   latest     826c66145a59    10 minutes ago      249MB
ubuntu-git   removed    826c66145a59    10 minutes ago      249MB
ubuntu-git   1.9        3e356394c14e    41 hours ago        249MB
...
```

Now you can run a Git command using the new image:

```
docker container run --rm ubuntu-git:auto
```

These commands demonstrate that the image you built with the Dockerfile works and is functionally equivalent to the one you built by hand. Examine what you did to accomplish this:

First, you created a Dockerfile with four instructions:

- `FROM ubuntu:latest`—Tells Docker to start from the latest Ubuntu image just as you did when creating the image manually.
- `LABEL maintainer`—Sets the maintainer name and email for the image. Providing this

information helps people know whom to contact if there's a problem with the image. This was accomplished earlier when you invoked `commit`.

- `RUN apt-get update && apt-get install -y git`—Tells the builder to run the provided commands to install Git.
- `ENTRYPOINT ["git"]`—Sets the entrypoint for the image to `git`.

Dockerfiles, like most scripts, can include comments. Any line beginning with a `#` will be ignored by the builder. It's important for Dockerfiles of any complexity to be well documented. In addition to improving Dockerfile maintainability, comments help people audit images that they're considering for adoption and spread best practices.

The only special rule about Dockerfiles is that the first instruction must be `FROM`. If you're starting from an empty image and your software has no dependencies, or you'll provide all the dependencies, then you can start from a special empty repository named `scratch`.

After you saved the Dockerfile, you started the build process by invoking the `docker image build` command. The command had one flag set and one argument. The `--tag` flag (or `-t` for short) specifies the full repository designation that you want to use for the resulting image. In this case you used `ubuntu-git:auto`. The argument that you included at the end was a single period. That argument told the builder the location of the Dockerfile. The period told it to look for the file in the current directory.

The `docker image build` command has another flag, `--file` (or `-f` for short), that lets you set the name of the Dockerfile. `Dockerfile` is the default, but with this flag you could tell the builder to look for a file named `BuildScript` or `release-image.df`. This flag sets only the name of the file, not the location of the file. That must always be specified in the location argument.

The builder works by automating the same tasks that you'd use to create images by hand. Each instruction triggers the creation of a new container with the specified modification. After the modification has been made, the builder commits the layer and moves on to the next instruction and container created from the fresh layer.

The builder validated that the image specified by the `FROM` instruction was installed as the first step of the build. If it were not, Docker would have automatically tried to pull the image. Take a look at the output from the `build` command that you ran:

```
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM ubuntu:latest
 ---> 452a96d81c30
```

You can see that in this case the base image specified by the `FROM` instruction is `ubuntu:latest`, which should have already been installed on your machine. The abbreviated image ID of the base image is included in the output.

The next instruction sets the maintainer information on the image. This creates a new container and then commits the resulting layer. You can see the result of this operation in the output for step 1:

```
Step 2/4 : LABEL maintainer="dia@allingeek.com"
```

```
 ---> Running in 11140b391074
Removing intermediate container 11140b391074
```

The output includes the ID of the container that was created and the ID of the committed layer. That layer will be used as the top of the image for the next instruction, `RUN`. The output for the `RUN` instruction was clouded with all the output for the command `apt-get update && apt-get install -y git`. If you're not interested in this output, you can invoke the `docker image build` command with the `--quiet` or `-q` flag. Running in quiet mode will suppress all output from the build process and management of intermediate containers. The only output of the build process in quiet mode is the resulting image id which looks like this:

```
sha256:e397ecfd576c83a1e49875477dcac50071e1c71f76f1d0c8d371ac74d97bbc90
```

Although this third step to install git usually takes much longer to complete, you can see the instruction and input as well as the ID of the container where the command was run and the ID of the resulting layer. Finally, the `ENTRYPOINT` instruction performs all the same steps, and the output is similarly unsurprising:

```
Step 4/4 : ENTRYPOINT ["git"]
 ---> Running in 6151803c388a
Removing intermediate container 6151803c388a
 ---> e397ecfd576c
Successfully built e397ecfd576c
Successfully tagged ubuntu-git:auto
```

A new layer is being added to the resulting image after each step in the build. Although this means you could potentially branch on any of these steps, the more important implication is that the builder can aggressively cache the results of each step. If a problem with the build script occurs after several other steps, the builder can restart from the same position after the problem has been fixed. You can see this in action by breaking your Dockerfile.

Add this line to the end of your Dockerfile:

```
RUN This will not work
```

Then run the build again:

```
docker image build --tag ubuntu-git:auto .
```

The output will show which steps the builder was able to skip in favor of cached results:

```
Sending build context to Docker daemon  2.048kB
Step 1/5 : FROM ubuntu:latest
 ---> 452a96d81c30
Step 2/5 : LABEL maintainer="dia@allingeek.com"
 ---> Using cache
 ---> 83da14c85b5a
Step 3/5 : RUN apt-get update && apt-get install -y git
 ---> Using cache
 ---> 795a6e5d560d
Step 4/5 : ENTRYPOINT ["git"]
 ---> Using cache
```

```
 ---> 89da8ffa57c7
Step 5/5 : RUN This will not work
 ---> Running in 2104ec7bc170
/bin/sh: 1: This: not found
The command '/bin/sh -c This will not work' returned a non-zero code: 127
```

❶ Note use of cache

Steps 1 through 4 were skipped because they were already built during your last build. Step 5 failed because there's no program with the name `This` in the container. The container output was valuable in this case because the error message informs you about the specific problem with the Dockerfile. If you fix the problem, the same steps will be skipped again, and the build will succeed, resulting in output like `Successfully built d7a8ee0cebd4`.

The use of caching during the build can save time if the build includes downloading material, compiling programs, or anything else that is time-intensive. If you need a full rebuild, you can use the `--no-cache` flag on `docker image build` to disable the use of the cache. Make sure you're disabling the cache only when required because it will place much more strain on upstream source systems and image building systems.

This short example uses 4 of the 18 Dockerfile instructions. The example was limited in that all the files that were added to the image were downloaded from the network; it modified the environment in a very limited way and provided a very general tool. The next example with a more specific purpose and local code will provide a more complete Dockerfile primer.

8.2 A Dockerfile primer

Dockerfiles are expressive and easy to understand due to their terse syntax that allows for comments. You can keep track of changes to Dockerfiles with any version-control system. Maintaining multiple versions of an image is as simple as maintaining multiple Dockerfiles. The Dockerfile build process itself uses extensive caching to aid rapid development and iteration. The builds are traceable and reproducible. They integrate easily with existing build systems and many continuous integration tools. With all these reasons to prefer Dockerfile builds to hand-made images, it's important to learn how to write them.

The examples in this section cover the core Dockerfile instructions used in most images. The following sections show how to create downstream behavior and more maintainable Dockerfiles. Every instruction is covered here at an introductory level. For deep coverage of each instruction, the best reference will always be the Docker documentation online at https://docs.docker.com/reference/builder/. The Docker builder reference also provides examples of good Dockerfiles and a best practices guide.

8.2.1 Metadata instructions

The first example builds a base image and two other images with distinct versions of the mailer program you used in chapter 2. The purpose of the program is to listen for messages on a TCP port and then send those messages to their intended recipients. The first version of

the mailer will listen for messages but only log those messages. The second will send the message as an `HTTP POST` to the defined URL.

One of the best reasons to use Dockerfile builds is that they simplify copying files from your computer into an image. But it's not always appropriate for certain files to be copied to images. The first thing to do when starting a new project is to define which files should never be copied into any images. You can do this in a file called .dockerignore. In this example you'll be creating three Dockerfiles, and none needs to be copied into the resulting images.

Use your favorite text editor to create a new file named .dockerignore and copy in the following lines:

```
.dockerignore
mailer-base.df
mailer-logging.df
mailer-live.df
```

Save and close the file when you've finished. This will prevent the .dockerignore file, or files named mailer-base.df, mailer-log.df, or mailer-live.df, from ever being copied into an image during a build. With that bit of accounting finished, you can begin working on the base image.

Building a base image helps create common layers. Each of the different versions of the mailer will be built on top of an image called `mailer-base`. When you create a Dockerfile, you need to keep in mind that each Dockerfile instruction will result in a new layer being created. Instructions should be combined whenever possible because the builder won't perform any optimization. Putting this in practice, create a new file named mailer-base.df and add the following lines:

```
FROM debian:stretch
LABEL maintainer="dia@allingeek.com"
RUN groupadd -r -g 2200 example && \
    useradd -rM -g example -u 2200 example
ENV APPROOT="/app" \
    APP="mailer.sh" \
    VERSION="0.6"
LABEL base.name="Mailer Archetype" \
      base.version="${VERSION}"
WORKDIR $APPROOT
ADD . $APPROOT
ENTRYPOINT ["/app/mailer.sh"]                          (1)
EXPOSE 33333
# Do not set the default user in the base otherwise
# implementations will not be able to update the image
# USER example:example
```

(1) **This file does not exist yet**

Put it all together by running the `docker image build` command from the directory where the mailer-base file is located. The `-f` flag tells the builder which filename to use as input:

```
docker image build -t dockerinaction/mailer-base:0.6 -f mailer-base.df .
```

Naming Dockerfiles

The default and most common name for a Dockerfile is `Dockerfile`. However, Dockerfiles can be named anything because they are simple text files and the build command accepts any filename you tell it. Some people name their Dockerfiles with an extension such as `df` so that they can easily define builds for multiple images in a single project directory, e.g. app-build.df, app-runtime.df, and app-debug-tools.df. A file extension also makes it easy to activate Dockerfile support in editors.

Five new instructions are introduced in this Dockerfile. The first new instruction is `ENV`. `ENV` sets environment variables for an image similar to the `--env` flag on `docker container run` or `docker container create`. In this case, a single `ENV` instruction is used to set three distinct environment variables. That could have been accomplished with three subsequent `ENV` instructions, though doing so would result in the creation of three layers. You can keep instructions easy to read by using a backslash to escape the newline character (just like shell scripting):

```
Step 4/9 : ENV APPROOT="/app"     APP="mailer.sh"     VERSION="0.6"
 ---> Running in c525f774240f
Removing intermediate container c525f774240f
```

Environment variables declared in the Dockerfile are made available to the resulting image but can be used in other Dockerfile instructions as substitutions. In this Dockerfile the environment variable `VERSION` was used as a substitution in the next new instruction, `LABEL`:

```
Step 5/9 : LABEL base.name="Mailer Archetype"       base.version="${VERSION}"
 ---> Running in 33d8f4d45042
Removing intermediate container 33d8f4d45042
 ---> 20441d0f588e
```

The `LABEL` instruction is used to define key/value pairs that are recorded as additional metadata for an image or container. This mirrors the `--label` flag on `docker run` and `docker create`. Like the `ENV` instruction before it, multiple labels can and should be set with a single instruction. In this case, the value of the `VERSION` environment variable was substituted for the value of the `base.version` label. By using an environment variable in this way, the value of `VERSION` will be available to processes running inside a container as well as recorded to an appropriate label. This increases maintainability of the Dockerfile because it's more difficult to make inconsistent changes when the value is set in a single location.

Organizing Metadata with Labels

Docker, Inc recommends recording metadata with labels to help organize images, networks, containers, and other objects. Each label key should be prefixed with the reverse DNS notation of a domain controlled or collaborating with the author, such as `com.<your company>.some-label`. Labels are flexible, extensible, and lightweight, but the lack of structure makes leveraging the information difficult. The Label Schema project (http://label-schema.org/) is a community effort to standardize label names and promote compatible tooling. The schema covers many important attributes of an image such as build date, name, and description. For example, when using the label schema

namespace, the key for 'build date' is named `org.label-schema.build-date` **and should have a value in RFC 3339 format such as** `2018-07-12T16:20:50.52Z`.

The next two instructions are `WORKDIR` and `EXPOSE`. These are similar in operation to their corresponding flags on the `docker run` and `docker create` commands. An environment variable was substituted for the argument to the `WORKDIR` command:

```
Step 6/9 : WORKDIR $APPROOT
Removing intermediate container c2cb1fc7bf4f
 ---> cb7953a10e42
```

The result of the `WORKDIR` instruction will be an image with the default working directory set to `/app`. Setting `WORKDIR` to a location that doesn't exist will create that location just like the command-line option. Last, the `EXPOSE` command creates a layer that opens TCP port 33333:

```
Step 9/9 : EXPOSE 33333
 ---> Running in cfb2afea5ada
Removing intermediate container cfb2afea5ada
 ---> 38a4767b8df4
```

The parts of this Dockerfile that you should recognize are the `FROM`, `LABEL`, and `ENTRYPOINT` instructions. In brief, the `FROM` instruction sets the layer stack to start from the `debian:stretch` image. Any new layers built will be placed on top of that image. The `LABEL` instruction adds key/value pairs to the image's metadata. The `ENTRYPOINT` instruction sets the executable to run at container startup. Here, it's setting the instruction to `exec ./mailer.sh` and using the shell form of the instruction.

The `ENTRYPOINT` instruction has two forms: the shell form and an exec form. The shell form looks like a shell command with whitespace-delimited arguments. The exec form is a string array where the first value is the command to execute and the remaining values are arguments. A command specified using the shell form would be executed as an argument to the default shell. Specifically, the command used in this Dockerfile will be executed as `/bin/sh -c 'exec ./mailer.sh'` at runtime. Most importantly, if the shell form is used for `ENTRYPOINT`, then all other arguments provided by the `CMD` instruction or at runtime as extra arguments to `docker container run` will be ignored. This makes the shell form of `ENTRYPOINT` less flexible.

You can see from the build output that the `ENV` and `LABEL` instructions each resulted in a single step and layer. But the output doesn't show that the environment variable values were substituted correctly. To verify that, you'll need to inspect the image:

```
docker inspect dockerinaction/mailer-base:0.6
```

TIP **Remember, the** `docker inspect` **command can be used to view the metadata of either a container or an image. In this case, you used it to inspect an image.**

The relevant lines are these:

```
"Env": [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
    "APPROOT=/app",
    "APP=mailer.sh",
    "VERSION=0.6"
],
...
"Labels": {
    "base.name": "Mailer Archetype",
    "base.version": "0.6",
    "maintainer": "dia@allingeek.com"
},
...
"WorkingDir": "/app"
```

The metadata makes it clear that the environment variable substitution works. You can use this form of substitution in the `ENV`, `ADD`, `COPY`, `LABEL`, `WORKDIR`, `VOLUME`, `EXPOSE`, and `USER` instructions.

The last commented line is a metadata instruction `USER`. It sets the user and group for all further build steps and containers created from the image. In this case, setting it in a base image would prevent any downstream Dockerfiles from installing software. That would mean that those Dockerfiles would need to flip the default back and forth for permission. Doing so would create at least two additional layers. The better approach would be to set up the user and group accounts in the base image and let the implementations set the default user when they've finished building.

The most curious thing about this Dockerfile is that the `ENTRYPOINT` is set to a file that doesn't exist. The entrypoint will fail when you try to run a container from this base image. But now that the entrypoint is set in the base image, that's one less layer that will need to be duplicated for specific implementations of the mailer. The next two Dockerfiles build different mailer.sh implementations.

8.2.2 File system instructions

Images that include custom functionality will need to modify the file system. A Dockerfile defines three instructions that modify the file system: `COPY`, `VOLUME`, and `ADD`. The Dockerfile for the first implementation should be placed in a file named mailer-logging.df:

```
FROM dockerinaction/mailer-base:0.6
COPY ["./log-impl", "${APPROOT}"]
RUN chmod a+x ${APPROOT}/${APP} && \
    chown example:example /var/log
USER example:example
VOLUME ["/var/log"]
CMD ["/var/log/mailer.log"]
```

In this Dockerfile you used the image generated from mailer-base as the starting point. The three new instructions are `COPY`, `VOLUME`, and `CMD`. The `COPY` instruction will copy files from the

file system where the image is being built into the build container. The `COPY` instruction takes at least two arguments. The last argument is the destination, and all other arguments are source files. This instruction has only one unexpected feature: any files copied will be copied with file ownership set to root. This is the case regardless of how the default user is set before the `COPY` instruction. It's better to delay any `RUN` instructions to change file ownership until all the files that you need to update have been copied into the image.

The `COPY` instruction will honor both shell style and exec style arguments, just like `ENTRYPOINT` and other instructions. But if any of the arguments contains whitespace, then you'll need to use the exec form.

> **TIP** Using the exec (or string array) form wherever possible is the best practice. At a minimum, a Dockerfile should be consistent and avoid mixing styles. This will make your Dockerfiles more readable and ensure that instructions behave as you'd expect without detailed understanding of their nuances.

The second new instruction is `VOLUME`. This behaves exactly as you'd expect if you understand what the `--volume` flag does on a call to `docker run` or `docker create`. Each value in the string array argument will be created as a new volume definition in the resulting layer. Defining volumes at image build time is more limiting than at runtime. You have no way to specify a bind-mount volume or read-only volume at image build time. This instruction will only create the defined location in the file system and then add a volume definition to the image metadata.

The last instruction in this Dockerfile is `CMD`. `CMD` is closely related to the `ENTRYPOINT` instruction. They both take either shell or exec forms and are both used to start a process within a container. But there are a few important differences.

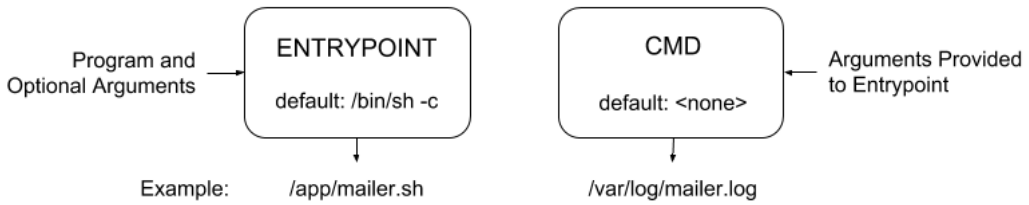


Figure 8.1 Relationship between ENTRYPOINT and CMD

The `CMD` command represents an argument list for the entrypoint. The default entrypoint for a container is /bin/sh. If no entrypoint is set for a container, then the values are passed, because the command will be wrapped by the default entrypoint. But if the entrypoint is set and is declared using the exec form, then you use `CMD` to set default arguments. The base for this Dockerfile defines the `ENTRYPOINT` as the mailer command. This Dockerfile injects an implementation of mailer.sh and defines a default argument. The argument used is the location that should be used for the log file.

Before building the image, you'll need to create the logging version of the mailer program. Create a directory at ./log-impl. Inside that directory create a file named mailer.sh and copy the following script into the file:

```
#!/bin/sh
printf "Logging Mailer has started.\n"
while true
do
    MESSAGE=$(nc -l -p 33333)
    printf "[Message]: %s\n" "$MESSAGE" > $1
    sleep 1
done
```

The structural specifics of this script are unimportant. All you need to know is that this script will start a mailer daemon on port 33333 and write each message that it receives to the file specified in the first argument to the program. Use the following command to build the `mailer-logging` image from the directory containing mailer-logging.df:

```
docker image build -t dockerinaction/mailer-logging -f mailer-logging.df .
```

The results of this image build should be anti-climactic. Go ahead and start up a named container from this new image:

```
docker run -d --name logging-mailer dockerinaction/mailer-logging
```

The logging mailer should now be built and running. Containers that link to this implementation will have their messages logged to /var/log/mailer.log. That's not very interesting or useful in a real-world situation, but it might be handy for testing. An implementation that sends email would be better for operational monitoring.

The next implementation example uses the Simple Email Service provided by Amazon Web Services to send email. Get started with another Dockerfile. Name this file mailer-live.df:

```
FROM dockerinaction/mailer-base:0.6
ADD ["./live-impl", "${APPROOT}"]
RUN apt-get update && \
    apt-get install -y curl python && \
    curl "https://bootstrap.pypa.io/get-pip.py" -o "get-pip.py" && \
    python get-pip.py && \
    pip install awscli && \
    rm get-pip.py && \
    chmod a+x "${APPROOT}/${APP}"
RUN apt-get install -y netcat
USER example:example
CMD ["mailer@dockerinaction.com", "pager@dockerinaction.com"]
```

This Dockerfile includes one new instruction, `ADD`. The `ADD` instruction operates similarly to the `COPY` instruction with two important differences. The `ADD` instruction will

- Fetch remote source files if a URL is specified
- Extract the files of any source determined to be an archive file

The auto-extraction of archive files is the more useful of the two. Using the remote fetch feature of the `ADD` instruction isn't good practice. The reason is that although the feature is convenient, it provides no mechanism for cleaning up unused files and results in additional layers. Instead, you should use a chained `RUN` instruction like the third instruction of mailer-live.df.

The other instruction to note in this Dockerfile is the `CMD` instruction, where two arguments are passed. Here you're specifying the From and To fields on any emails that are sent. This differs from mailer-logging.df, which specifies only one argument.

Next, create a new subdirectory named live-impl under the location containing mailer-live.df. Add the following script to a file in that directory named mailer.sh:

```
#!/bin/sh
printf "Live Mailer has started.\n"
while true
do
  MESSAGE=$(nc -l -p 33333)
  aws ses send-email --from $1 \
    --destination {\"ToAddresses\":[\"$2\"]} \
    --message "{\"Subject\":{\"Data\":\"Mailer Alert\"},\
                \"Body\":{\"Text\":{\"Data\":\"$MESSAGE}\"}}}"
  sleep 1
done
```

The key takeaway from this script is that, like the other mailer implementation, it will wait for connections on port 33333, take action on any received messages, and then sleep for a moment before waiting for another message. This time, though, the script will send an email using the Simple Email Service command-line tool. Build and start a container with these two commands:

```
docker image build -t dockerinaction/mailer-live -f mailer-live.df .
docker run -d --name live-mailer dockerinaction/mailer-live
```

If you link a watcher to these, you'll find that the logging mailer works as advertised. But the live mailer seems to be having difficulty connecting to the Simple Email Service to send the message. With a bit of investigation, you'll eventually realize that the container is misconfigured. The `aws` program requires certain environment variables to be set.

You'll need to set `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION` in order to get this example working. Discovering execution preconditions this way can be frustrating for users. Section 8.5.1 details an image design pattern that reduces this friction and helps adopters.

Before you get to design patterns, you need to learn about the final Dockerfile instruction. Remember, not all images contain applications. Some are built as platforms for downstream images. Those cases specifically benefit from the ability to inject downstream build-time behavior.

8.3 Injecting downstream build-time behavior

A Dockerfile instruction that is important for authors of base images is `ONBUILD`. The `ONBUILD` instruction defines instructions to execute if the resulting image is used as a base for another build. For example, you could use `ONBUILD` instructions to compile a program that's provided by a downstream layer. The upstream Dockerfile copies the contents of the build directory into a known location and then compiles the code at that location. The upstream Dockerfile would use a set of instructions like this:

```
ONBUILD COPY [".", "/var/myapp"]
ONBUILD RUN go build /var/myapp
```

The instructions following `ONBUILD` instructions aren't executed when their containing Dockerfile is built. Instead, those instructions are recorded in the resulting image's metadata under ContainerConfig.OnBuild. The previous instructions would result in the following metadata inclusions:

```
...
"ContainerConfig": {
...
    "OnBuild": [
        "COPY [\".\", \"/var/myapp\"]",
        "RUN go build /var/myapp"
    ],
    ...
```

This metadata is carried forward until the resulting image is used as the base for another Dockerfile build. When a downstream Dockerfile uses the upstream image (the one with the `ONBUILD` instructions) in a `FROM` instruction, those `ONBUILD` instructions are executed after the `FROM` instruction and before the next instruction in a Dockerfile.

Consider the following example to see exactly when `ONBUILD` steps are injected into a build. You need to create two Dockerfiles and execute two build commands to get the full experience. First, create an upstream Dockerfile that defines the `ONBUILD` instructions. Name the file base.df and add the following instructions:

```
FROM busybox:latest
WORKDIR /app
RUN touch /app/base-evidence
ONBUILD RUN ls -al /app
```

You can see that the image resulting from building base.df will add an empty file named base-evidence to the /app directory. The `ONBUILD` instruction will list the contents of the /app directory at build time, so it's important that you not run the build in quiet mode if you want to see exactly when changes are made to the file system.

The next file to create is the downstream Dockerfile. When this is built, you will be able to see exactly when the changes are made to the resulting image. Name the file downstream.df and include the following contents:

```
FROM dockerinaction/ch8_onbuild
RUN touch downstream-evidence
RUN ls -al .
```

This Dockerfile will use an image named `dockerinaction/ch8_onbuild` as a base, so that's the repository name you'll want to use when you build the base. Then you can see that the downstream build will create a second file and then list the contents of /app again.

With these two files in place, you're ready to start building. Run the following to create the upstream image:

```
docker image build -t dockerinaction/ch8_onbuild -f base.df .
```

The output of the build should look like this:

```
Sending build context to Docker daemon  3.072kB
Step 1/4 : FROM busybox:latest
 ---> 6ad733544a63
Step 2/4 : WORKDIR /app
Removing intermediate container dfc7a2022b01
 ---> 9bc8aeafdec1
Step 3/4 : RUN touch /app/base-evidence
 ---> Running in d20474e07e45
Removing intermediate container d20474e07e45
 ---> 5d4ca3516e28
Step 4/4 : ONBUILD RUN ls -al /app
 ---> Running in fce3732daa59
Removing intermediate container fce3732daa59
 ---> 6ff141f94502
Successfully built 6ff141f94502
Successfully tagged dockerinaction/ch8_onbuild:latest
```

Then build the downstream image with this command:

```
docker image build -t dockerinaction/ch8_onbuild_down -f downstream.df .
```

The results clearly show when the `ONBUILD` instruction (from the base image) is executed:

```
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM dockerinaction/ch8_onbuild
# Executing 1 build trigger
 ---> Running in 591f13f7a0e7
total 8
drwxr-xr-x    1 root     root          4096 Jun 18 03:12 .
drwxr-xr-x    1 root     root          4096 Jun 18 03:13 ..
-rw-r--r--    1 root     root             0 Jun 18 03:12 base-evidence
Removing intermediate container 591f13f7a0e7
 ---> 5b434b4be9d8
Step 2/3 : RUN touch downstream-evidence
 ---> Running in a42c0044d14d
Removing intermediate container a42c0044d14d
 ---> e48a5ea7b66f
Step 3/3 : RUN ls -al .
 ---> Running in 7fc9c2d3b3a2
total 8
drwxr-xr-x    1 root     root          4096 Jun 18 03:13 .
```

```
drwxr-xr-x    1 root     root          4096 Jun 18 03:13 ..
-rw-r--r--    1 root     root             0 Jun 18 03:12 base-evidence
-rw-r--r--    1 root     root             0 Jun 18 03:13 downstream-evidence
Removing intermediate container 7fc9c2d3b3a2
 ---> 46955a546cd3
Successfully built 46955a546cd3
Successfully tagged dockerinaction/ch8_onbuild_down:latest
```

You can see the builder registering the `ONBUILD` instruction with the container metadata in step 4 of the base build. Later, the output of the downstream image build shows which triggers (`ONBUILD` instructions) it has inherited from the base image. The builder discovers and processes the trigger immediately after step 0, the `FROM` instruction. The output then includes the result of the `RUN` instruction specified by the trigger. The output shows that only evidence of the base build is present. Later, when the builder moves on to instructions from the downstream Dockerfile, it lists the contents of the /app directory again. The evidence of both changes is listed.

That example is more illustrative than it is useful. You should consider browsing Docker Hub and looking for images tagged with `onbuild` suffixes to get an idea about how this is used in the wild. Here are a couple of my favorites:

- https://hub.docker.com/r/_/python/
- https://hub.docker.com/r/_/node/

8.4 Creating maintainable Dockerfiles

Dockerfile has features that make maintaining closely related images easier. These features help authors share metadata and data between images at build time. Let's work through a couple Dockerfile implementations and use these features to make them more concise and maintainable.

As you were writing the mailer application's Dockerfiles, you may have noticed a few repeated bits that need to change for every update. The `VERSION` variable is the best example of repetition. The version metadata goes into the image tag, environment variable, and label metadata. There's another issue, too. Build systems often derive version metadata from the application's version control system. We would prefer not to hardcode it in our Dockerfiles or scripts. Dockerfile's `ARG` instruction provides a solution to these problems. `ARG` defines a variable that users can provide to Docker when building an image. Docker interpolates the argument value into the Dockerfile, allowing creation of parametrized Dockerfiles. You provide build arguments to the `docker image build` command using one or more `--build-arg <varname>=<value>` options.

Let's introduce the `ARG VERSION` instruction into `mailer-base.df` on line 2:

```
FROM debian:stretch
ARG VERSION=unknown                          (1)
LABEL maintainer="dia@allingeek.com"
RUN groupadd -r -g 2200 example && \
    useradd -rM -g example -u 2200 example
```

```
ENV APPROOT="/app" \
    APP="mailer.sh" \
    VERSION="${VERSION}"
LABEL base.name="Mailer Archetype" \
      base.version="${VERSION}"
WORKDIR $APPROOT
ADD . $APPROOT
ENTRYPOINT ["/app/mailer.sh"]
EXPOSE 33333
```

❶ **Define the VERSION build arg with default value 'unknown'**

Now the version can be defined once as a shell variable and passed on the command-line as both the image tag and a build argument for use within the image:

```
version=0.6; docker image build -t dockerinaction/mailer-base:${version} -f mailer-
      base.df --build-arg VERSION=${version} .
```

Let's use `docker inspect` to verify the `VERSION` was substituted all the way down to the `base.version` label:

```
docker inspect --format '{{ json .Config.Labels }}' dockerinaction/mailer-base:0.6
```

The inspect command should produce json output that looks like:

```
{
  "base.name": "Mailer Archetype",
  "base.version": "0.6",
  "maintainer": "dia@allingeek.com"
}
```

If we had not specified `VERSION` as a build argument, then the default value of `unknown` would be used and a warning printed during the build process.

Let's turn our attention to multi-stage builds, which can help manage important concerns by distinguishing between phases of an image build. Multi-stage builds can help solve a few common problems. The primary uses are: reusing parts of another image, separating the build of an application from the build of an application runtime image, and enhancing an application's runtime image with specialized test or debug tools. The example that follows will demonstrate reusing parts of another image as well as separating an application's build and runtime concerns. First, let's learn about Dockerfile's multi-stage feature.

A multi-stage Dockerfile is a Dockerfile that has multiple `FROM` instructions. Each `FROM` instruction marks a new build stage whose final layer may be referenced in a downstream stage. A build stage is named by appending `AS <name>` to the FROM instruction, where name is an identifier you specify such as *builder*. The name can be used in subsequent `FROM` and `COPY --from=<name|index>` instructions, providing a convenient way identify the source layer for files brought into the image build. When you build a Dockerfile with multiple stages, the build process still produces a single Docker image. The image is produced from the final stage executed in the Dockerfile. Let's demonstrate the use of multi-stage builds with an example that uses two stages and a bit of composition. An easy way to follow along with this example

is to clone the git repository at `git@github.com:dockerinaction/ch8_multi_stage_build.git`

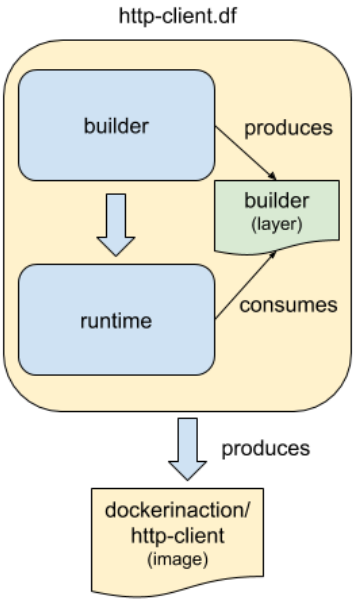


Figure 8.2 Multi-stage Docker Builds

This Dockerfile defines two stages: builder and runtime. The builder stage of the creates a stage that gathers dependencies and builds the example program. The runtime stage copies the Certificate Authority and program files into the runtime image for execution. The source of the http-client.df Dockerfile is:

```
################################################
# Define a Builder stage and build app inside it
FROM golang:1-alpine as builder

# Install CA Certificates
RUN apk update && apk add ca-certificates

# Copy source into Builder
ENV HTTP_CLIENT_SRC=$GOPATH/src/dia/http-client/
COPY . $HTTP_CLIENT_SRC
WORKDIR $HTTP_CLIENT_SRC

# Build HTTP Client
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -v -o /go/bin/http-client

################################################
# Define a stage to build a runtime image.
FROM scratch as runtime
```

```
ENV PATH="/bin"
# Copy CA certificates and application binary from builder stage
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/ca-
      certificates.crt
COPY --from=builder /go/bin/http-client /http-client
ENTRYPOINT ["/http-client"]
```

Let's examine the image build in detail. The `FROM golang:1-alpine as builder` instruction declares that the first stage will be based on golang's alpine image variation and aliased to `builder` for easy referencing by later stages. First, the builder installs Certificate Authority files used to establish TLS connections supporting https. These CA files aren't used in this stage, but will be stored for composition by the runtime image. Next, the `builder` stage copies the http-client source code into the container and builds the http-client golang program into a static binary. The http-client program is stored in the builder container at `/go/bin/http-client`.

The http-client program is very simple. `http-client` makes an http request to retrieve its own source code from GitHub:

```
package main

import (
      "net/http"
)
import "io/ioutil"
import "fmt"

func main() {
      resp, err :=
      http.Get("https://raw.githubusercontent.com/dockerinaction/ch8_multi_stage_bui
      ld/master/http-client.go")
      if err != nil {
        panic(err)
      }
      defer resp.Body.Close()
      body, err := ioutil.ReadAll(resp.Body)
      fmt.Println("response:\n", string(body))
}
```

The `runtime` stage is based on `scratch`. When you build an image `FROM scratch`, the filesystem begins empty and the image will contain only what is `COPY`'d there. Notice that the http.Get statement retrieves the file using the https protocol. This means the program will need a set of valid TLS certificate authorities. CA authorities are available from the `builder` stage because we installed them previously. The `runtime` stage copies both the `ca-certificates.crt` and `http-client` files from the builder stage into the `runtime` stage with:

```
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/ca-
      certificates.crt
COPY --from=builder /go/bin/http-client /http-client
```

The runtime stage finishes by setting the ENTRYPOINT of the image to `/http-client`, which will be invoked when the container starts. The final image will contain just two files. You can build the image with a command like:

```
docker build -t dockerinaction/http-client -f http-client.df .
```

The image can be executed with:

```
docker container run --rm -it dockerinaction/http-client:latest
```

When the http-client image runs successfully, it will output the `http-client.go` sourcecode listed above. To recap, the `http-client.df` Dockerfile uses a builder stage to retrieve runtime dependencies and build the http-client program. The runtime stage then copies http-client and its dependencies from the builder stage onto the minimal scratch base and configures it for execution. The resulting image contains only what is needed to run the program and is just over 6MiB in size. In the next section, we'll work through a different style of application delivery using defensive startup scripts.

8.5 Using startup scripts and multiprocess containers

Whatever tooling you choose to use, you'll always need to consider a few image design aspects. You'll need to ask yourself whether the software running in your container requires any startup assistance, supervision, monitoring, or coordination with other in-container processes. If so, then you'll need to include a startup script or initialization program with the image and install it as the entrypoint.

8.5.1 Environmental preconditions validation

Failure modes are difficult to communicate and can catch someone off guard if they occur at arbitrary times. If container configuration problems always cause failures at startup time for an image, users can be confident that a started container will keep running.

In software design, failing fast and precondition validation are best practices. It makes sense that the same should hold true for image design. The preconditions that should be evaluated are assumptions about the context.

Docker containers have no control over the environment where they're created. They do, however, have control of their own execution. An image author can solidify the user experience of their image by introducing environment and dependency validation prior to execution of the main task. A container user will be better informed about the requirements of an image if containers built from that image fail fast and display descriptive error messages.

For example, WordPress requires certain environment variables to be set or container links to be defined. Without that context, WordPress would be unable to connect to the database where the blog data is stored. It would make no sense to start WordPress in a container without access to the data it's supposed to serve. WordPress images use a script as the container entrypoint. That script validates that the container context is set in a way that's

compatible with the contained version of WordPress. If any required condition is unmet (a link is undefined or a variable is unset), then the script will exit before starting WordPress, and the container will stop unexpectedly.

Validating the preconditions for a program startup is generally use-case specific. If you're packaging software into an image, you'll usually need to write a script or carefully configure the tool used to startup the program yourself. The startup process should validate as much of the assumed context as possible. This should include the following:

- Presumed links (and aliases)
- Environment variables
- Secrets
- Network access
- Network port availability
- Root file system mount parameters (read-write or read-only)
- Volumes
- Current user

You can use whichever scripting or programming language you want to accomplish this task. In the spirit of building minimal images, it's a good idea to use a language or scripting tool that's already included with the image. Most base images ship with a shell like /bin/sh or /bin/bash. Shell scripts are the most common because shell programs are commonly available and they adapt to program and environment-specific requirements easily. When building an image from `scratch` for a single binary such as the http-client example from Section 8.4, the program is responsible for validating its own preconditions as no other programs will exist in the container.

Consider the following shell script that might accompany a program that depends on a web server. At container startup, this script enforces that either another container has been linked to the web alias and has exposed port 80 or the `WEB_HOST` environment variable has been defined:

```
#!/bin/bash
set -e

if [ -n "$WEB_PORT_80_TCP" ]; then
  if [ -z "$WEB_HOST" ]; then
    WEB_HOST='web'
  else
    echo >&2 '[WARN]: Linked container, "web" overridden by $WEB_HOST.'
    echo >&2 "===> Connecting to WEB_HOST ($WEB_HOST)"
  fi
fi

if [ -z "$WEB_HOST" ]; then
  echo >&2 '[ERROR]: specify a linked container, "web" or WEB_HOST environment
      variable'
  exit 1
fi
exec "$@" # run the default command
```

If you're unfamiliar with shell scripting, this is an appropriate time to learn it. The topic is approachable, and there are several excellent resources for self-directed learning. This specific script uses a pattern where both an environment variable and a container link are tested. If the environment variable is set, the container link will be ignored. Finally, the default command is executed.

Images that use a startup script to validate configuration should fail fast if someone uses them incorrectly, but those same containers may fail later for other reasons. You can combine startup scripts with container restart policies to make reliable containers. But container restart policies are not perfect solutions. Containers that have failed and are waiting to be restarted aren't running. This means that an operator won't be able to execute another process within a container that's in the middle of a backoff window. The solution to this problem involves making sure the container never stops.

8.5.2 Initialization processes

UNIX-based computers usually start an initialization (init) process first. That init process is responsible for starting all the other system services, keeping them running, and shutting them down. It's often appropriate to use an init-style system to launch, manage, restart, and shut down container processes with a similar tool.

Init processes typically use a file or set of files to describe the ideal state of the initialized system. These files describe what programs to start, when to start them, and what actions to take when they stop. Using an init process is the best way to launch multiple programs, clean up orphaned processes, monitor processes, and automatically restart any failed processes.

If you decide to adopt this pattern, you should use the init process as the entrypoint of your application-oriented Docker container. Depending on the init program you use, you may need to prepare the environment beforehand with a startup script.

For example, the runit program doesn't pass environment variables to the programs it launches. If your service uses a startup script to validate the environment, it won't have access to the environment variables it needs. The best way to fix that problem might be to use a startup script for the runit program. That script might write the environment variables to some file so the startup script for your application can access them.

There are several open source init programs. Full-featured Linux distributions ship with heavyweight and full-featured init systems like SysV, Upstart, and systemd. Linux Docker images like Ubuntu, Debian, and CentOS typically have their init programs installed but nonfunctioning out of the box. These can be complex to configure and may have hard dependencies on resources that require root access. For that reason, the community has tended toward the use of lighter-weight init programs.

Popular options include runit, tini, BusyBox init, Supervisord, and DAEMON Tools. These all attempt to solve similar problems, but each has its benefits and costs. Using an init process is a best practice for application containers, but there's no perfect init program for every use case. When evaluating any init program for use in a container, consider these factors:

- Additional dependencies the program brings into the image
- File sizes
- How the program passes signals to its child processes (or if it does at all)
- Required user access
- Monitoring and restart functionality (backoff-on-restart features are a bonus)
- Zombie process cleanup features

Init processes are so important that Docker provides an `--init` option to run an init process inside the container to manage the program being executed. The `--init` option can be used to 'add' an init process to an existing image. For example, we can run netcat using the alpine:3.6 image and manage it with an init process:

```
docker container run -it --init alpine:3.6 nc -l -p 3000
```

If you inspect the host's processes with `ps -ef`, you will see Docker ran `/dev/init -- nc -l -p 3000` inside the container instead of just `nc`. Docker uses the `tini` program as an init process by default, though you may specify another init process instead.

Whichever init program you decide on, make sure your image uses it to boost adopter confidence in containers created from your image. If the container needs to fail fast to communicate a configuration problem, make sure the init program won't hide that failure. Now that we have a solid foundation for running and signaling processes inside containers, let's see how to communicate the health of containerized processes to collaborators.

8.5.3 The Purpose and Use of Health Checks

Health checks are used to determine if the application running inside the container is ready and able to perform its function. Engineers define application-specific health checks for containers to detect conditions when the application is running, but is 'stuck' or has broken dependencies.

Docker runs a single command inside the container to determine if the application is healthy. There are two ways to specify the health check command:

- Use a `HEALTHCHECK` instruction when defining the image
- On the command-line when running a container

This Dockerfile defines a healthcheck for the nginx webserver:

```
FROM nginx:1.13-alpine

HEALTHCHECK --interval=5s --retries=2 \
  CMD nc -vz -w 2 localhost 80 || exit 1
```

The health check command should be reliable, lightweight, and not interfere with the operation of the main application because it will be executed frequently. The command's exit status will be used to determine the container's health. Docker has defined the following exit statuses:

- 0: success - the container is healthy and ready for use
- 1: unhealthy - the container is not working correctly
- 2: reserved - do not use this exit code

Most programs in the Unix world exit with a 0 status when things went as expected and a non-zero status otherwise. The `|| exit 1` is a bit of shell trickery that means 'or exit 1'. This means whenever `nc` exits with any non-zero status, `nc`'s status will be converted to 1 so that Docker knows the container is unhealthy. Conversion of non-zero exit statuses to 1 is a common pattern since Docker does not define the behavior of all non-zero health check status, only 1 and 2. As of this writing, use of an exit code whose behavior is not defined will result in an unhealthy status.

Let's build and run the nginx example with:

```
docker image build -t dockerinaction/healthcheck .
docker container run –name healthcheck_ex -d dockerinaction/healthcheck
```

Now that a container with a healthcheck is running, we can inspect the container's health status with `docker ps`. When a health check is defined, the `docker ps` command reports the container's current health status in the STATUS column. Docker `ps` output can be a bit unwieldy, so we will use a custom format that prints the container name, image name, and status in a table:

```
docker ps --format 'table {{.Names}}\t{{.Image}}\t{{.Status}}'
NAMES               IMAGE                          STATUS
healthcheck_ex      dockerinaction/healthcheck   Up 3 minutes (healthy)
```

By default, the health check command will be run every 30 seconds and three failed checks are required before transitioning the container's `health_status` to `unhealthy`. The health check interval and number of consecutive failures before reporting a container as unhealthy can be adjusted in the HEALTHCHECK instruction or when running the container.

The health check facility also supports options for:

- timeout - a timeout for the health check command to run and exit
- start period - a grace period at the start of a container to *not* count health check failures towards the health status; once the health check command returns healthy, the container is considered started and subsequent failures count towards the health status

Image authors should define a useful health check in images where possible. Usually this means exercising the application in some way or checking an internal application health status indicator such as a `/health` endpoint on a web server. However, sometimes it is impractical to define a `HEALTHCHECK` instruction because not enough is known about how the image will run ahead of time. To address this problem, Docker provides the `--health-cmd` to define a health check when running a container.

Let's take the previous `HEALTHCHECK` example and specify the health check when running the container instead:

```
docker container run --name=healthcheck_ex -d \
  --health-cmd='nc -vz -w 2 localhost 80 || exit 1' \
  nginx:1.13-alpine
```

Defining a health check at runtime overrides the health check defined in the image where one exists. This is very useful for integrating a 3rd-party image because you can account for requirements specific to your environment.

These are the tools at your disposal to build images that result in durable containers. Durability is not security, and although adopters of your durable images might trust that they will keep running as long as they can, they shouldn't trust your images until they've been hardened.

8.6 Building hardened application images

As an image author, it's difficult to anticipate all the scenarios where your work will be used. For that reason, harden the images you produce whenever possible. *Hardening an image* is the process of shaping it in a way that will reduce the attack surface inside any Docker containers based on it.

A general strategy for hardening an application image is to minimize the software included with it. Naturally, including fewer components reduces the number of potential vulnerabilities. Further, building minimal images keeps image download times short and helps adopters deploy and build containers more rapidly.

There are three things that you can do to harden an image beyond that general strategy. First, you can enforce that your images are built from a specific image. Second, you can make sure that regardless of how containers are built from your image, they will have a sensible default user. Last, you should eliminate a common path for root user escalation from programs with setuid or setgid attributes set.

8.6.1 Content addressable image identifiers

The image identifiers discussed so far in this book are all designed to allow an author to update images in a transparent way to adopters. An image author chooses what image their work will be built on top of, but that layer of transparency makes it difficult to trust that the base hasn't changed since it was vetted for security problems. Since Docker 1.6, the image identifier has included an optional digest component.

An image ID that includes the digest component is called a content addressable image identifier (CAIID). This refers to a specific layer containing specific content, instead of simply referring to a particular and potentially changing layer.

Now image authors can enforce a build from a specific and unchanging starting point as long as that image is in a version 2 repository. Append an `@` symbol followed by the digest in place of the standard tag position.

Use `docker image pull` and observe the line labeled `Digest` in the output to discover the digest of an image from a remote repository. Once you have the digest, you can use it as the

identifier to `FROM` instructions in a Dockerfile. For example, consider the following, which uses a specific snapshot of `debian:stable` as a base:

```
docker pull debian:stable
stable: Pulling from library/debian
31c6765cabf1: Pull complete
Digest: sha256:6aedee3ef827...

# Dockerfile:
FROM debian@sha256:6aedee3ef827...
...
```

Regardless of when or how many times the Dockerfile is used to build an image, they will all use the content identified with that CAIID as their base. This is particularly useful for incorporating known updates to a base into your images and identifying the exact build of the software running on your computer.

Although this doesn't directly limit the attack surface of your images, using CAIIDs will prevent it from changing without your knowledge. The next two practices do address the attack surface of an image.

8.6.2 User permissions

The known container breakout tactics all rely on having system administrator privileges inside the container. Chapter 6 covers the tools used to harden containers. That chapter includes a deep dive into user management and a discussion of the USR Linux namespace. This section covers standard practices for establishing reasonable user defaults for images.

First, please understand that a Docker user can always override image defaults when they create a container. For that reason, there's no way for an image to prevent containers from running as the root user. The best things an image author can do are create other non-root users and establish a non-root default user and group.

Dockerfile includes a `USER` instruction that sets the user and group in the same way you would with the `docker container run` or `docker container create` command. The instruction itself was covered in the Dockerfile primer. This section is about considerations and best practices.

The best practice and general guidance is to drop privileges as soon as possible. You can do this with the `USER` instruction before any containers are ever created or with a startup script that's run at container boot time. The challenge for an image author is to determine the earliest appropriate time.

If you drop privileges too early, the active user may not have permission to complete the instructions in a Dockerfile. For example, this Dockerfile won't build correctly:

```
FROM busybox:latest
USER 1000:1000
RUN touch /bin/busybox
```

Building that Dockerfile would result in step 2 failing with a message like `touch: /bin/busybox: Permission denied`. File access is obviously impacted by user changes. In this case UID 1000 doesn't have permission to change the ownership of the file /bin/busybox. That file is currently owned by root. Reversing the second and third lines would fix the build.

The second timing consideration is the permissions and capabilities needed at runtime. If the image starts a process that requires administrative access at runtime, then it would make no sense to drop user access to a non-root user before that point. For example, any process that needs access to the system port range (1–1024) will need to be started by a user with administrative (at the very least CAP_NET_ADMIN) privileges. Consider what happens when you try to bind to port 80 as a non-root user with Netcat. Place the following Dockerfile in a file named UserPermissionDenied.df:

```
FROM busybox:latest
USER 1000:1000
ENTRYPOINT ["nc"]
CMD ["-l", "-p", "80", "0.0.0.0"]
```

Build the Dockerfile and run the resulting image in a container. In this case the user (UID 1000) will lack the required privileges, and the command will fail:

```
docker image build \
    -t dockerinaction/ch8_perm_denied \
    -f UserPermissionDenied.df \
    .
docker container run dockerinaction/ch8_perm_denied
# Output:
# nc: bind: Permission denied
```

In cases like these, you may see no benefit in changing the default user. Instead, any startup scripts that you build should take on the responsibility of dropping permissions as soon as possible. The last question is which user should be dropped into?

In the default Docker configuration, containers use the same Linux USR namespace as the host. This means that UID 1000 in the container is UID 1000 on the host machine. All other aspects apart from the UID and GID are segregated, just as they would be between computers. For example, UID 1000 on your laptop might be your username, but the username associated with UID 1000 inside a BusyBox container could be `default`, `busyuser`, or whatever the BusyBox image maintainer finds convenient. When the Docker `userns-remap` feature described in Chapter 6 is enabled, then UIDs in the container are mapped to unprivileged UIDs on the host. USR namespace remapping provides full UID and GID segregation, even for root. But can you depend on `userns-remap` being in effect?

Image authors often do not know the Docker daemon configuration where their images will run. Even if Docker adopted USR namespace remapping in a default configuration, it will be difficult for image authors to know which UID/GID is appropriate to use. The only thing we can be sure of is that it's inappropriate to use common or system-level UID/GIDs where doing so can be avoided. With that in mind, it's still burdensome to use raw UID/GID numbers. Doing

so makes scripts and Dockerfiles less readable. For that reason, it's typical for image authors to include `RUN` instructions that create users and groups used by the image. The following is the second instruction in a Postgres Dockerfile:

```
# add our user and group first to make sure their IDs get assigned
# consistently, regardless of whatever dependencies get added
RUN groupadd -r postgres && useradd -r -g postgres postgres
```

This instruction simply creates a postgres user and group with automatically assigned UID and GID. The instruction is placed early in the Dockerfile so that it will always be cached between rebuilds, and the IDs remain consistent regardless of other users that are added as part of the build. This user and group could then be used in a `USER` instruction. That would make for a safer default. But Postgres containers require elevated privileges during startup. Instead, this particular image uses a `su` or `sudo`-like program called `gosu` to start the Postgres process as the postgres user. Doing so makes sure that the process runs without administrative access in the container.

User permissions are one of the more nuanced aspects of building Docker images. The general rule you should follow is that if the image you're building is designed to run some specific application code, then the default execution should drop user permissions as soon as possible.

A properly functioning system should be reasonably secure with reasonable defaults in place. Remember, though, an application or arbitrary code is rarely perfect and could be intentionally malicious. For that reason, you should take additional steps to reduce the attack surface of your images.

8.6.3 SUID and SGID permissions

The last hardening action to cover is the mitigation of SUID or SGID permissions. The well-known file system permissions (read, write, execute) are only a portion of the set defined by Linux. In addition to those, two are of particular interest: SUID and SGID.

These two are similar in nature. An executable file with the SUID bit set will always execute as its owner. Consider a program like /usr/bin/passwd, which is owned by the root user and has the SUID permission set. If a non-root user like bob executes passwd, he will execute that program as the root user. You can see this in action by building an image from the following Dockerfile:

```
FROM ubuntu:latest
# Set the SUID bit on whoami
RUN chmod u+s /usr/bin/whoami
# Create an example user and set it as the default
RUN adduser --system --no-create-home --disabled-password --disabled-login \
    --shell /bin/sh example
USER example
# Set the default to compare the container user and
# the effective user for whoami
CMD  printf "Container running as:          %s\n" $(id -u -n) && \
     printf "Effectively running whoami as: %s\n" $(whoami)
```

Once you've created the Dockerfile, you need to build an image and run the default command in a container:

```
docker image build -t dockerinaction/ch8_whoami .
docker run dockerinaction/ch8_whoami
```

Doing so prints results like these to the terminal:

```
Container running as:          example
Effectively running whoami as: root
```

The output of the default command shows that even though you've executed the `whoami` command as the example user, it's running from the context of the root user. The SGID works similarly. The difference is that the execution will be from the owning group's context, not the owning user.

Running a quick search on your base image will give you an idea of how many and which files have these permissions:

```
docker run --rm debian:stretch find / -perm /u=s -type f
```

It will display a list like this:

```
/bin/umount
/bin/ping
/bin/su
/bin/mount
/usr/bin/chfn
/usr/bin/passwd
/usr/bin/newgrp
/usr/bin/gpasswd
/usr/bin/chsh
```

This command will find all of the SGID files:

```
docker container run --rm debian:stretch find / -perm /g=s -type f
```

The resulting list is much shorter:

```
/sbin/unix_chkpwd
/usr/bin/chage
/usr/bin/expiry
/usr/bin/wall
```

Each of the listed files in this particular image has the SUID or SGID permission, and a bug in any of them could be used to compromise the root account inside a container. The good news is that files that have either of these permissions set are typically useful during image builds but rarely required for application use cases. If your image is going to be running software that's arbitrary or externally sourced, it's a best practice to mitigate this risk of escalation.

Fix this problem and either delete all these files or unset their SUID and SGID permissions. Taking either action would reduce the image's attack surface. The following Dockerfile instruction will unset the SUID and GUID permissions on all files currently in the image:

```
RUN for i in $(find / -type f \( -perm /u=s -o -perm /g=s \)); \
    do chmod ug-s $i; done
```

Hardening images will help users build hardened containers. Although it's true that no hardening measures will protect users from intentionally building weak containers, those measures will help the more unsuspecting and most common type of user.

8.7 Summary

Most Docker images are built automatically from Dockerfiles. This chapter covers the build automation provided by Docker and Dockerfile best practices. Before moving on, make sure that you've understood these key points:

- Docker provides an automated image builder that reads instructions from Dockerfiles.
- Each Dockerfile instruction results in the creation of a single image layer.
- Merge instructions to minimize the size of images and layer count when possible.
- Dockerfiles include instructions to set image metadata like the default user, exposed ports, default command, and entrypoint.
- Other Dockerfile instructions copy files from the local file system or a remote location into the produced images.
- Downstream builds inherit build triggers that are set with `ONBUILD` instructions in an upstream Dockerfile.
- Dockerfile maintenance can be improved with multi-stage builds and the ARG instruction.
- Startup scripts should be used to validate the execution context of a container before launching the primary application.
- A valid execution context should have appropriate environment variables set, network dependencies available, and an appropriate user configuration.
- Init programs can be used to launch multiple processes, monitor those processes, reap orphaned child processes, and forward signals to child processes.
- Images should be hardened by building from content addressable image identifiers, creating a non-root default user, and disabling or removing any executable with SUID or SGID permissions.

9

Public and private software distribution

This chapter covers

- Choosing a project distribution method
- Using hosted infrastructure
- Running and using your own registry
- Understanding manual image distribution workflows
- Distributing image sources

You have your own images of software you've written, customized, or just pulled from the internet. But what good is an image if nobody can install it? Docker is different from other container management tools because it provides image distribution features.

There are several ways to get your images out to the world. This chapter explores those distribution paradigms and provides a framework for making or choosing one or more for your own projects.

Hosted registries offer both public and private repositories with automated build tools. By contrast, running a private registry lets you hide and customize your image distribution infrastructure. Heavier customization of a distribution workflow might require you to abandon the Docker image distribution facilities and build your own. Some systems might abandon the image as the distribution unit altogether and distribute the source files for images instead.

This chapter will teach you how to select and use a method for distributing your images to the world or just at work.

9.1 Choosing a distribution method

The most difficult thing about choosing a distribution method is choosing the appropriate method for your situation. To help with this problem, each method presented in this chapter is examined on the same set of selection criteria.

The first thing to recognize about distributing software with Docker is that there's no universal solution. Distribution requirements vary for many reasons, and several methods are available. Every method has Docker tools at its core, so it's always possible to migrate from one to another with minimal effort. The best way to start is by examining the full spectrum of options at a high level.

9.1.1 A distribution spectrum

The image distribution spectrum offers many methods with differing levels of flexibility and complexity. The methods that provide the most flexibility can be the most complicated to use, whereas those that are the simplest to use are generally the most restrictive. Figure 9.1 shows the full spectrum.

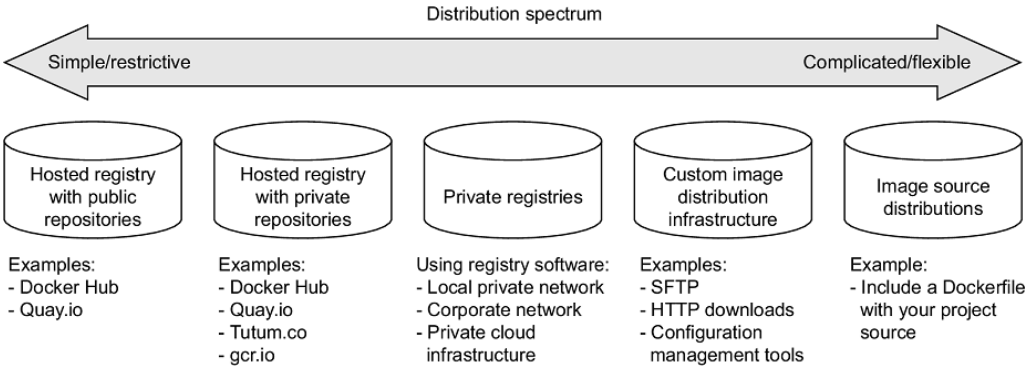


Figure 9.1 The image distribution spectrum

The methods included in the spectrum range from hosted registries like Docker Hub to totally custom distribution architectures or source-distribution methods. Some of these subjects will be covered in more detail than others. Particular focus is placed on private registries because they provide the most balance between the two concerns.

Having a spectrum of choices illustrates your range of options, but you need a consistent set of selection criteria in order to determine which you should use.

9.1.2 Selection criteria

Choosing the best distribution method for your needs may seem daunting with this many options. In situations like these you should take the time to understand the options, identify criteria for making a selection, and avoid the urge to make a quick decision or settle.

The following identified selection criteria are based on differences across the spectrum and on common business concerns. When making a decision, consider how important each of these is in your situation:

- Cost
- Visibility
- Transport speed or bandwidth overhead
- Longevity control
- Availability control
- Access control
- Artifact integrity
- Artifact confidentiality
- Requisite expertise

How each distribution method stacks up against these criteria is covered in the relevant sections over the rest of this chapter.

COST

Cost is the most obvious criterion, and the distribution spectrum ranges in cost from free to very expensive and "it's complicated." Lower cost is generally better, but cost is typically the most flexible criterion. For example, most people will trade cost for artifact confidentiality if the situation calls for it.

VISIBILITY

Visibility is the next most obvious criterion for a distribution method. Secret projects or internal tools should be difficult if not impossible for unauthorized people to discover. In another case, public works or open source projects should be as visible as possible to promote adoption.

TRANSPORTATION

Transportation speed and bandwidth overhead are the next most flexible criteria. File sizes and image installation speed will vary between methods that leverage image layers, concurrent downloads, and prebuilt images and those that use flat image files or rely on deployment time image builds. High transportation speeds or low installation latency is critical for systems that use just-in-time deployment to service synchronous requests. The opposite is true in development environments or asynchronous processing systems.

LONGEVITY

Longevity control is a business concern more than a technical concern. Hosted distribution methods are subject to other people's or companies' business concerns. An executive faced with the option of using a hosted registry might ask, "What happens if they go out of business or pivot away from repository hosting?" The question reduces to, "Will the business needs of the third party change before ours?" If this is a concern for you, then longevity control is important. Docker makes it simple to switch between methods, and other criteria like requisite expertise or cost may actually trump this concern. For those reasons, longevity control is another of the more flexible criteria.

AVAILABILITY

Availability control is the ability to control the resolution of availability issues with your repositories. Hosted solutions provide no availability control. Businesses typically provide some service-level agreement on availability if you're a paying customer, but there's nothing you can do to directly resolve an issue. On the other end of the spectrum, private registries or custom solutions put both the control and responsibility in your hands.

ACCESS CONTROL

Access control protects your images from modification or access by unauthorized parties. There are varying degrees of access control. Some systems provide only access control of modifications to a specific repository, whereas others provide course control of entire registries. Still other systems may include pay walls or digital rights management controls. Projects typically have specific access control needs dictated by the product or business. This makes access control requirements one of the least flexible and most important to consider.

INTEGRITY

Artifact integrity and confidentiality both fall in the less-flexible and more-technical end of the spectrum. Artifact integrity is trustworthiness and consistency of your files and images. Violations of integrity may include man-in-the-middle attacks, where an attacker intercepts your image downloads and replaces the content with their own. They might also include malicious or hacked registries that lie about the payloads they return.

CONFIDENTIALITY

Artifact confidentiality is a common requirement for companies developing trade secrets or proprietary software. For example, if you use Docker to distribute cryptographic material, then confidentiality will be a major concern. Artifact integrity and confidentiality features vary across the spectrum. Overall, the out-of-the-box distribution security features won't provide the tightest confidentiality or integrity. If that's one of your needs, an information security professional will need to implement and review a solution.

The last thing to consider when choosing a distribution method is the level of expertise required. Using hosted methods can be very simple and requires little more than a mechanical understanding of the tools. Building custom image or image source distribution pipelines requires expertise with a suite of related technologies. If you don't have that expertise or don't have access to someone who does, using more complicated solutions will be a challenge. In that case, you may be able to reconcile the gap at additional cost.

With this strong set of selection criteria, you can begin learning about and evaluating different distribution methods. The following sections will evaluate these methods against the criteria using ratings of: Worst, Bad, Good, Better, and Best. The best place to start is on the far left of the spectrum with hosted registries.

9.2 Publishing with hosted registries

As a reminder, Docker registries are services that make repositories accessible to Docker pull commands. A registry hosts repositories. The simplest way to distribute your images is by using hosted registries.

A hosted registry is a Docker registry service that's owned and operated by a third-party vendor. Docker Hub, Quay.io, and Google Container Registry are all examples of hosted registry providers. By default, Docker publishes to Docker Hub. Docker Hub and most other hosted registries provide both public and private registries, as shown in figure 9.2.

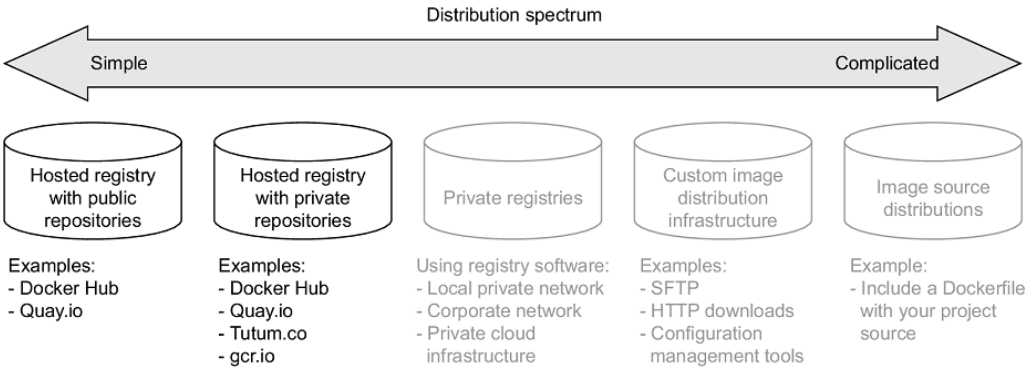


Figure 9.2 The simplest side of the distribution spectrum and the topic of this section

The example images used in this book are distributed with public repositories hosted on Docker Hub and Quay.io. By the end of this section you'll understand how to publish your own images using hosted registries and how hosted registries measure up to the selection criteria.

9.2.1 Publishing with public repositories: Hello World via Docker Hub

The simplest way to get started with public repositories on hosted registries is to push a repository that you own to Docker Hub. To do so, all you need is a Docker Hub account and an image to publish. If you haven't done so already, sign up for a Docker Hub account now.

Once you have your account, you need to create an image to publish. Create a new Dockerfile named HelloWorld.df and add the following instructions:

```
FROM busybox:latest                    (1)
CMD echo Hello World
```

(1) From HelloWorld.df

Chapter 8 covers Dockerfile instructions. As a reminder, the `FROM` instruction tells the Docker image builder which existing image to start the new image from. The `CMD` instruction sets the default command for the new image. Containers created from this image will display "Hello World" and exit. Build your new image with the following command:

```
docker image build \
    -t <insert Docker Hub username>/hello-dockerfile \          (1)
    -f HelloWorld.df \
    .
```

(1) Insert your username

Be sure to substitute your Docker Hub username in that command. Authorization to access and modify repositories is based on the username portion of the repository name on Docker Hub. If you create a repository with a username other than your own, you won't be able to publish it.

Publishing images on Docker Hub with the `docker` command-line tool requires that you establish an authenticated session with that client. You can do that with the `login` command:

```
docker login
```

This command will prompt you for your username, email address, and password. Each of those can be passed to the command as arguments using the `--username`, `--email`, and `--password` flags. When you log in, the `docker` client maintains a map of your credentials for the different registries that you authenticate with in a file. It will specifically store your username and an authentication token, not your password.

You will be able to push your repository to the hosted registry once you've logged in. Use the `docker push` command to do so:

```
docker image push <insert Docker Hub username>/hello-dockerfile      (1)
```

(1) Insert your username

Running that command should create output like the following:

```
The push refers to a repository
[dockerinaction/hello-dockerfile] (len: 1)
7f6d4eb1f937: Image already exists
8c2e06607696: Image successfully pushed
6ce2e90b0bc7: Image successfully pushed
cf2616975b4a: Image successfully pushed
Digest:
  sha256:ef18de4b0ddf9ebd1cf5805fae1743181cbf3642f942cae8de7c5d4e375b1f20
```

The command output includes upload statuses and the resulting repository content digest. The push operation will create the repository on the remote registry, upload each of the new layers, and then create the appropriate tags.

Your public repository will be available to the world as soon as the push operation is completed. Verify that this is the case by searching for your username and your new repository. For example, use the following command to find the example owned by the dockerinaction user:

```
docker search dockerinaction/hello-dockerfile
```

Replace the dockerinaction username with your own to find your new repository on Docker Hub. You can also log in to the Docker Hub website and view your repositories to find and modify your new repository.

Having distributed your first image with Docker Hub, you should consider how this method measures up to the selection criteria; see table 9.1.

Table 9.1 Performance of public hosted repositories

Criteria	Rating	Notes
Cost	Best	Public repositories on hosted registries are almost always free. That price is difficult to beat. These are especially helpful when you're getting started with Docker or publishing open source software.
Visibility	Best	Hosted registries are well-known hubs for software distribution. A public repository on a hosted registry is an obvious distribution choice if you want your project to be well known and visible to the public.
Transport speed/size	Better	Hosted registries like Docker Hub are layer-aware and will work with Docker clients to transfer only the layers that the client doesn't already have. Further, pull operations that require multiple repositories to be transferred will perform those transfers in parallel. For those reasons, distributing an image from a hosted repository is fast, and the payloads are minimal.
Availability control	Worst	You have no availability control over hosted registries.
Longevity control	Good	You have no longevity control over hosted registries. But registries will all conform to the Docker registry API, and migrating from one host to another should be a low-cost exercise.

Access control	Better	Public repositories are open to the public for read access. Write access is still controlled by whatever mechanisms the host has put in place. Write access to public repositories on Docker Hub is controlled two ways. First, repositories owned by an individual may be written to only by that individual account. Second, repositories owned by organizations may be written to by any user who is part of that organization.
Artifact integrity	Best	The current version of the Docker registry API, V2, provides content-addressable images. The V2 API lets you request an image with a specific cryptographic signature. The Docker client will validate the integrity of the returned image by recalculating the signature and comparing it to the one requested. Old versions of Docker that are unaware of the V2 registry API don't support this feature and use V1 instead. In those cases and for other cases where signatures are unknown, a high degree of trust is put into the authorization and at-rest security features provided by the host.
Secrecy	Worst	Hosted registries and public repositories are never appropriate for storing and distributing cleartext secrets or sensitive code. Remember, secrets include passwords, API keys, certificates and more. Anyone can access these secrets.
Requisite experience	Best	Using public repositories on hosted registries requires only that you be minimally familiar with Docker and capable of setting up an account through a website. This solution is within reach for any Docker user.

Public repositories on hosted registries are the best choice for owners of open source projects or people who are just getting started with Docker. People should still be skeptical of software that they download and run from the internet, and so public repositories that don't expose their sources can be difficult for some users to trust. Hosted (trusted) builds solve this problem to a certain extent.

9.2.2 Private hosted repositories

Private repositories are similar to public repositories from an operational and product perspective. Most registry providers offer both options, and any differences in provisioning through their websites will be minimal. Because the Docker registry API makes no distinction between the two types of repositories, registry providers that offer both generally require you to provision private registries through their website, app, or API.

The tools for working with private repositories are identical to those for working with public repositories, with one exception. Before you can use `docker image pull` or `docker container run` to install an image from a private repository, you need to have authenticated with the registry where the repository is hosted. To do so, you will use the `docker login` command just as you would if you were using `docker image push` to upload an image.

The following commands prompt you to authenticate with the registries provided by Docker Hub and quay.io. After creating accounts and authenticating, you'll have full access to

your public and private repositories on all three registries. The `login` subcommand takes an optional server argument:

```
docker login
# Username: dockerinaction
# Password:
# Email: book@dockerinaction.com
# WARNING: login credentials saved in /Users/xxx/.dockercfg.
# Login Succeeded

docker login quay.io
# Username: dockerinaction
# Password:
# Email: book@dockerinaction.com
# WARNING: login credentials saved in /Users/xxx/.dockercfg.
# Login Succeeded
```

Before you decide that private hosted repositories are the distribution solution for you, consider how they might fulfill your selection criteria; see table 9.2

Table 9.2 Performance of private hosted repositories

Criteria	Rating	Notes
Cost	Best	The cost of private repositories typically scales with the number of repositories that you need. Plans usually range from a few dollars per month for 5 repositories up to around $50 for 50 repositories. Price pressure of storage and monthly virtual server hosting is a driving factor here. Users or organizations that require more than 50 repositories may find it more appropriate to run their own private registry.
Visibility	Best	Private repositories are by definition private. These are typically excluded from indexes and should require authentication before a registry acknowledges the repository's existence. Private repositories are poor candidates for publicizing availability of some software or distributing open source images. Instead they're great tools for small private projects or organizations that don't want to incur the overhead associated with running their own registry.
Transport speed/size	Better	Any hosted registry like Docker Hub will minimize the bandwidth used to transfer an image and enable clients to transfer an image's layers in parallel. Ignoring potential latency introduced by transferring files over the internet, hosted registries should always perform well against other non-registry solutions.
Availability control	Worst	No hosted registry provides any availability control. Unlike public repositories, however, using private repositories will make you a paying customer. Paying customers may have stronger SLA guarantees or access to support personnel.
Longevity control	Good	You have no longevity control over hosted registries. But registries will all conform to the Docker registry API, and migrating from one host to another should be a low-cost exercise.

Access control	Better	Both read and write access to private repositories is restricted to users with authorization.
Artifact integrity	Best	It's reasonable to expect all hosted registries to support the V2 registry API and content-addressable images.
Secrecy	Worst	Despite the privacy provided by these repositories, these are never suitable for storing clear-text secrets or trade-secret code. Although the registries require user authentication and authorization to requested resources, there are several potential problems with these mechanisms. The provider may use weak credential storage, have weak or lost certificates, or leave your artifacts unencrypted at rest. Finally, your secret material should not be accessible to employees of the registry provider.
Requisite experience	Best	Just like public repositories, using private repositories on hosted registries requires only that you be minimally familiar with Docker and capable of setting up an account through a website. This solution is within reach for any Docker user.

Individuals and small teams will find the most utility in private hosted repositories. Their low cost and basic authorization features are friendly to low-budget projects or private projects with minimal security requirements. Large companies or projects that need a higher degree of secrecy and have a suitable budget may find their needs better met by running their own private registry.

9.3 Introducing private registries

When you have a hard requirement on availability control, longevity control, or secrecy, then running a private registry may be your best option. In doing so, you gain control without sacrificing interoperability with Docker pull and push mechanisms or adding to the learning curve for your environment. People can interact with a private registry exactly as they would with a hosted registry.

The Docker registry software (called Distribution) is open source software and distributed under the Apache 2 license. The availability of this software and permissive license keep the engineering cost of running your own registry low. It’s available through Docker Hub and is simple to use for non-production purposes. Figure 9.4 illustrates that private registries fall in the middle of the distribution spectrum.

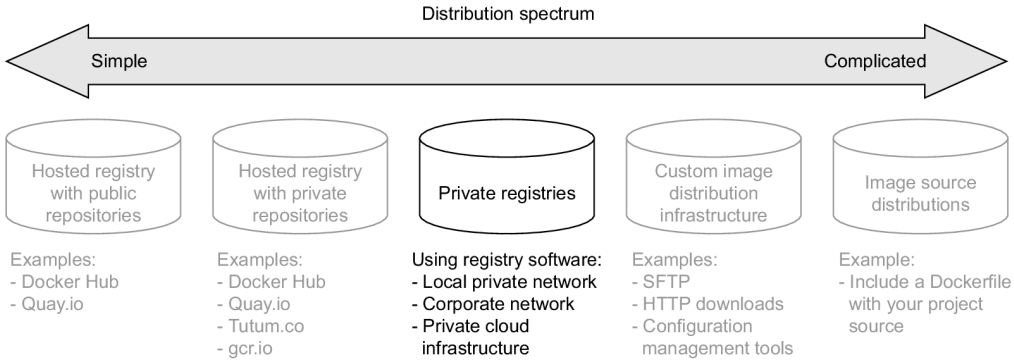


Figure 9.3 Private registries in the image distribution spectrum

Running a private registry is a great distribution method if you have special infrastructure use cases like the following:

- Regional image caches
- Team-specific image distribution for locality or visibility
- Environment or deployment stage-specific image pools
- Corporate processes for approving images
- Longevity control of external images

Before deciding that this is the best choice for you, consider the costs detailed in the selection criteria, shown in table 9.3.

Table 9.3 Performance of private registries

Criteria	Rating	Notes
Cost	Good	At a minimum, a private registry adds to hardware overhead (virtual or otherwise), support expense, and risk of failure. But the community has already invested the bulk of the engineering effort required to deploy a private registry by building the open source software. Cost will scale on different dimensions than hosted registries. Whereas the cost of hosted repositories scales with raw repository count, the cost of private registries scales with transaction rates and storage usage. If you build a system with high transaction rates, you'll need to scale up the number of registry hosts so that you can handle the demand. Likewise, registries that serve some number of small images will have lower storage costs than those serving the same number of large images.
Visibility	Good	Private registries are as visible as you decide to make them. But even a registry that you own and open up to the world will be less visible than advertised popular

		registries like Docker Hub.
Transport speed/size	Best	Latency of operations between any client and any registry will vary based on network performance between those two nodes and load on the registry. Private registries may be faster or slower than hosted registries due to these variables. Most people operating large-scale deployments or internal infrastructure will find private registries appealing. Private registries eliminate a dependency on the internet or inter-datacenter networking and will improve latency proportionate to the external network constraint. Because this solution uses a Docker registry, it shares the same parallelism gains as hosted registry solutions.
Availability control	Best	You have full control over availability as the registry owner.
Longevity control	Best	You have full control over solution longevity as the registry owner.
Access control	Good	The registry software doesn't include any authentication or authorization features out of the box. But implementing those features can be achieved with a minimal engineering exercise.
Artifact integrity	Best	Version 2 of the registry API supports content-addressable images, and the open source software supports a pluggable storage back end. For additional integrity protections, you can force the use of TLS over the network and use back-end storage with encryption at rest.
Secrecy	Good	Private registries are the first solution on the spectrum appropriate for storage of trade secrets or secret material. You control the authentication and authorization mechanisms. You also control the network and in-transit security mechanisms. Most importantly, you control the at-rest storage. It's in your power to ensure that the system is configured in such a way that your secrets stay secret.
Requisite experience	Good	Getting started and running a local registry requires only basic Docker experience. But running and maintaining a highly available production private registry requires experience with several technologies. The specific set depends on what features you want to take advantage of. Generally, you'll want to be familiar with NGINX to build a proxy, LDAP or Kerberos to provide authentication, and Redis for caching. There are many commercial product solutions available for running a private Docker registry, ranging from traditional artifact repositories such as Artifactory and Nexus to software delivery systems like GitLab.

The biggest trade-off going from hosted registries to private registries is gaining flexibility and control while requiring greater depth and breadth of engineering experience to build and maintain the solution. Docker image registries often consume large amounts of storage, so be sure to account for that in your analysis. The remainder of this section covers what you need to implement all but the most complicated registry deployment designs and highlights opportunities for customization in your environment.

9.3.1 Using the registry image

Whatever your reasons for doing so, getting started with the Docker registry software is easy. The Distribution software is available on Docker Hub in a repository named registry. Starting a local registry in a container can be done with a single command:

```
docker run -d -p 5000:5000 \
        -v "$(pwd)"/data:/tmp/registry-dev \
        --restart=always --name local-registry registry:2
```

The image that's distributed through Docker Hub is configured for insecure access from the machine running a client's Docker daemon. When you've started the registry, you can use it like any other registry with `docker pull`, `run`, `tag`, and `push` commands. In this case, the registry location is localhost:5000. The architecture of your system should now match that described in figure 9.5

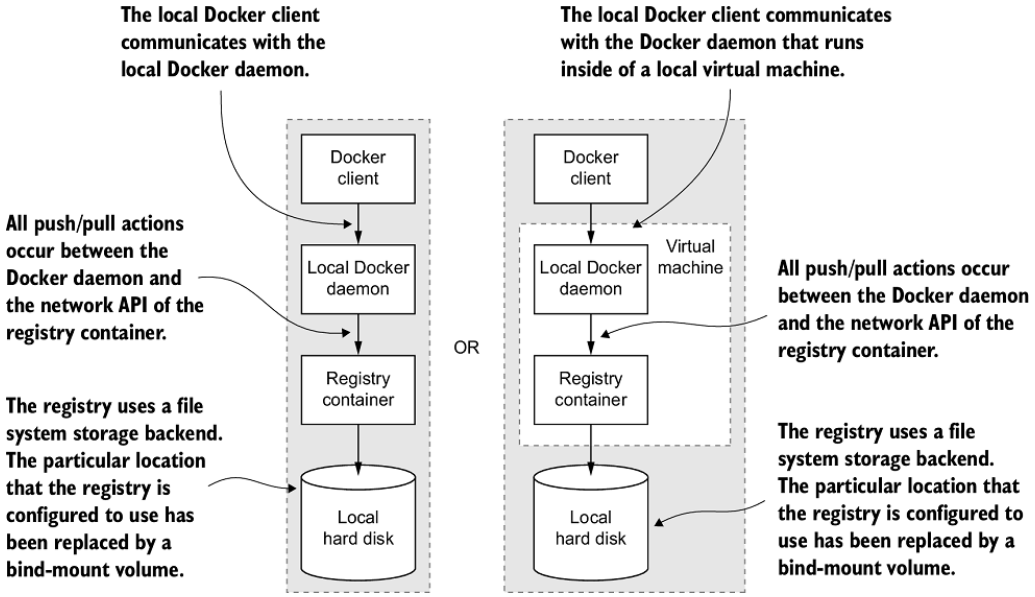


Figure 9.4 Interactions between the `docker` client, daemon, local registry container, and local storage

Companies that want tight version control on their external image dependencies will pull images from external sources like Docker Hub and copy them into their own registry. You might do this to ensure an important image does not change or disappear unexpectedly when the author updates or removes the source image. To get an idea of what it's like working with your registry, consider a workflow for copying images from Docker Hub into your new registry:

```
docker image pull dockerinaction/ch9_registry_bound                    ❶
```

```
docker image ls -f "label=dia_excercise=ch9_registry_bound"          ❷

docker image tag dockerinaction/ch9_registry_bound \                 ❸
    localhost:5000/dockerinaction/ch9_registry_bound

docker image push localhost:5000/dockerinaction/ch9_registry_bound
```

❶ Pull demo image from Docker Hub
❷ Verify image is discoverable with label filter
❸ Push demo image into

In running these four commands, you copy an example repository from Docker Hub into your local repository. If you execute these commands from the same location as where you started the registry, you'll find that the newly created data subdirectory contains new registry data.

9.3.2 Consuming images from your registry

The tight integration you get with the Docker ecosystem can make it feel like you're working with software that's already installed on your computer. When internet latency has been eliminated, such as when you're working with a local registry, it can feel even less like you're working with distributed components. For that reason, the exercise of pushing data into a local repository isn't very exciting on its own.

The next set of commands should impress on you that you're working with a real registry. These commands will remove the example repositories from the local cache for your Docker daemon, demonstrate that they're gone, and then reinstall them from your personal registry:

```
docker image rm \
    dockerinaction/ch9_registry_bound \
    localhost:5000/dockerinaction/ch9_registry_bound          ❶

docker image ls -f "label=dia_excercise=ch9_registry_bound"   ❷

docker image pull localhost:5000/dockerinaction/ch9_registry_bound

docker image ls -f "label=dia_excercise=ch9_registry_bound"   ❸

docker container rm -vf local-registry                        ❹
```

❶ Remove tagged reference
❷ Pull from registry again
❸ Demonstrate that image is back
❹ Clean up local registry

You can work with this registry locally as much as you want, but the insecure default configuration will prevent remote Docker clients from using your registry (unless they specifically allow insecure access). This is one of the few issues that you'll need to address before deploying a registry in a production environment. Chapter 10 covers the registry software in depth.

This is the most flexible distribution method that involves Docker registries. If you need greater control over the transport, storage, and artifact management, you should consider working directly with images in a manual distribution system.

9.4 Manual image publishing and distribution

Images are files, and you can distribute them as you would any other file. It's common to see software available for download on websites, File Transport Protocol (FTP) servers, corporate storage networks, or via peer-to-peer networks. You could use any of these distribution channels for image distribution. You can even use email or USB key in cases where you know your image recipients. Manual image distribution methods provide the ultimate in flexibility, enabling varied use cases such as distributing images to many people at an event simultaneously or to a secure air-gapped network.

When you work with images as files, you use Docker only to manage local images and create files. All other concerns are left for you to implement. That void of functionality makes manual image publishing and distribution the second-most flexible but complicated distribution method. This section covers custom image distribution infrastructure, shown on the spectrum in figure 9.6.

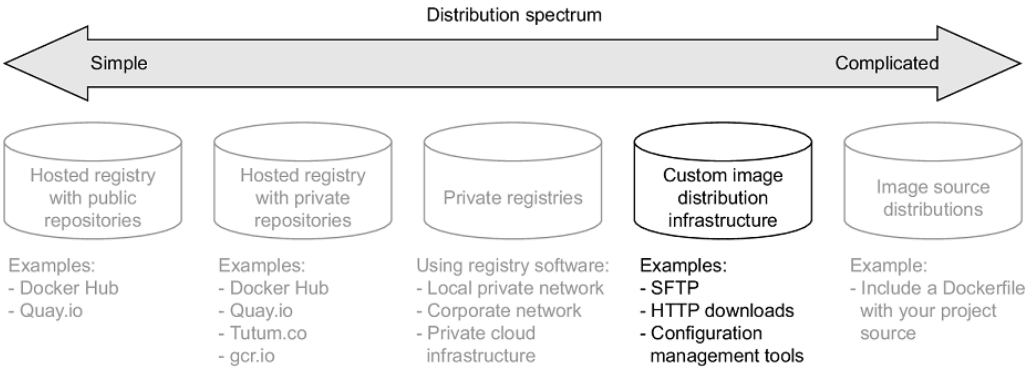


Figure 9.5 Docker image distribution over custom infrastructure

We've already covered all the methods for working with images as files. Chapter 3 covers loading images into Docker and saving images to your hard drive. Chapter 7 covers exporting and importing full file systems as flattened images. These techniques are the foundation for building distribution workflows like the one shown in figure 9.7.

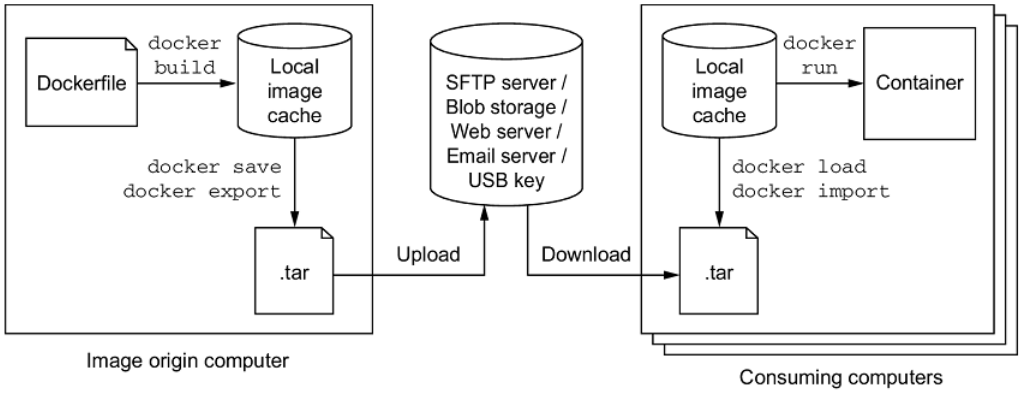


Figure 9.6 A typical manual distribution workflow with producer, transport, and consumers

The workflow illustrated in figure 9.7 is a generalization of how you'd use Docker to create an image and prepare it for distribution. You should be familiar with using `docker image build` to create an image and `docker image save` or `docker container export` to create an image file. You can perform each of these operations with a single command.

You can use any file transport once you have an image in file form. One custom component not show in figure 9.7 is the mechanism that uploads an image to the transport. That mechanism may be a folder that is watched by a file-sharing tool like Dropbox. It could also be a piece of custom code that runs periodically, or in response to a new file, and uses FTP or HTTP to push the file to a remote server. Whatever the mechanism, this general component will require some effort to integrate.

The figure also shows how a client would ingest the image and use it to build a container after the image has been distributed. Similar to image origins, clients require some process or mechanism to acquire the image from a remote source. Once clients have the image file, they can use the `docker image load` or `import` commands to complete the transfer.

Manual image distribution methods are difficult to measure against the selection criteria without knowing the specifics of the distribution problem. Using a non-Docker distribution channel gives you full control, making it possible to handle unusual requirements. It will be up to you to determine how your options measure against the selection criteria. Table 9.4 explores how manual image distribution methods rate against the selection criteria.

Table 9.4 Performance of custom image distribution infrastructure.

Criteria	Rating	Notes
Cost	Good	Distribution costs are driven by bandwidth, storage, and hardware needs. Hosted distribution solutions like cloud storage will bundle these costs and generally scale down price per unit as your usage increases. But hosted solutions bundle in the cost

		of personnel and several other benefits that you may not need, driving up the price compared to a mechanism that you own.
Visibility	Bad	Most manual distribution methods are special and will take more effort to advertise and use than public or private registries. Examples might include using popular websites or other well-known file distribution hubs.
Transport speed/size	Good	Whereas transport speed depends on the transport, file sizes are dependent on your choice of using layered images or flattened images. Remember, layered images maintain the history of the image, container-creation metadata, and old files that might have been deleted or overridden. Flattened images contain only the current set of files on the file system.
Availability control	Best	If availability control is an important factor for your case, you can use a transport mechanism that you own.
Longevity control	Bad	Using proprietary protocols, tools, or other technology that is neither open nor under your control will impact longevity control. For example, distributing image files with a hosted file-sharing service like Dropbox will give you no longevity control. On the other hand, swapping USB drives with your friend will last as long as the two of you decide to use USB drives.
Access control	Bad	You could use a transport with the access control features you need or use file encryption. If you built a system that encrypted your image files with a specific key, you could be sure that only a person or people with the correct key could access the image.
Artifact integrity	Bad	Integrity validation is a more expensive feature to implement for broad distribution. At a minimum, you'd need a trusted communication channel for advertising cryptographic file signatures and creating archives that maintain image and layer signatures by using `docker image save` and `load`.
Secrecy	Good	You can implement content secrecy with cheap encryption tools. If you need meta-secrecy (where the exchange itself is secret) as well as content secrecy, then you should avoid hosted tools and make sure that the transport that you use provides secrecy (HTTPS, SFTP, SSH, or offline).
Requisite experience	Good	Hosted tools will typically be designed for ease of use and require a lesser degree of experience to integrate with your workflow. But you can use simple tools that you own as easily in most cases.

All the same criteria apply to manual distribution, but it's difficult to discuss them without the context of a specific transportation method.

9.4.1 A sample distribution infrastructure using the File Transfer Protocol

Building a fully functioning example will help you understand exactly what goes into a manual distribution infrastructure. This section will help you build an infrastructure with the File Transfer Protocol.

FTP is less popular than it used to be. The protocol provides no secrecy and requires credentials to be transmitted over the wire for authentication. But the software is freely available and clients have been written for most platforms. That makes FTP a great tool for building your own distribution infrastructure. Figure 9.8 illustrates what you'll build.

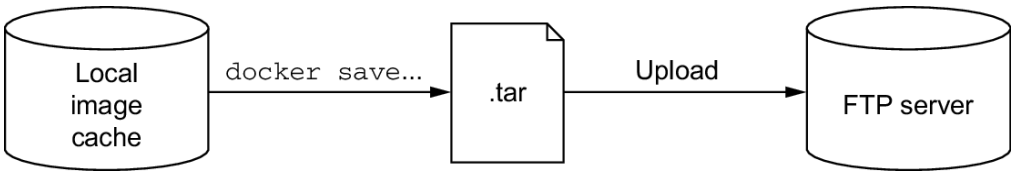


Figure 9.7 An FTP publishing infrastructure

The example in this section uses two existing images. The first, `dockerinaction/ch9_ftpd`, is a specialization of the `centos:6` image where `vsftpd` (an FTP daemon) has been installed and configured for anonymous write access. The second image, `dockerinaction/ch9_ftp_client`, is a specialization of a popular minimal Alpine Linux image. An FTP client named `LFTP` has been installed and set as the entrypoint for the image.

To prepare for the experiment, pull a known image from Docker Hub that you want to distribute. In the example, the `registry:2` image is used:

```
docker image pull registry:2
```

Once you have an image to distribute, you can begin. The first step is building your image distribution infrastructure. In this case, that means running an FTP server, which we will do on a dedicated network:

```
docker network create ch9_ftp
docker container run -d --name ftp-server --network=ch9_ftp -p 21:21 \
    dockerinaction/ch9_ftpd
```

This command will start an FTP server that accepts FTP connections on TCP port 21 (the default port). Don't use this image in any production capacity. The server is configured to allow anonymous connections write access under the pub/incoming folder. Your distribution infrastructure will use that folder as an image distribution point.

The next thing you need to do is export an image to the file format. You can use the following command to do so:

```
docker image save -o ./registry.2.tar registry:2
```

Running this command will export the `registry:2` image as a structured image file in your current directory. The file will retain all the metadata and history associated with the image. At this point, you could inject all sorts of phases like checksum generation or file encryption. This infrastructure has no such requirements, and you should move along to distribution.

The `dockerinaction/ch9_ftp_client` image has an FTP client installed and can be used to upload your new image file to your FTP server. Remember, you started the FTP server in a container named `ftp-server`. The `ftp-server` container is attached to a user-defined bridge network (see chapter 5) named ch9_ftp and other containers attached to the ch9_ftp network will be able to connect to `ftp-server`. Let's upload the registry image archive with an ftp client:

```
docker container run --rm -it --network ch9_ftp \
    -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e 'cd pub/incoming; put registry.2.tar; exit' ftp-server
```

This command creates a container with a volume bound to your local directory and joined to the `ch9_ftp` network where the FTP server container is listening. The command uses LFTP to upload a file named registry.2.tar to the server located at ftp_server. You can verify that you uploaded the image by listing the contents of the FTP server's folder:

```
docker run --rm -it --network ch9_ftp \
    -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e "cd pub/incoming; ls; exit" ftp-server
```

The registry image is now available for download to any FTP client that knows about the server and can access it over the network. But that file may never be overridden in the current FTP server configuration. You'd need to come up with your own versioning scheme if you were going to use a similar tool in production.

Advertising the availability of the image in this scenario requires clients to periodically poll the server using the last command you ran to list files. Alternatively, you could build a website or send an email notifying clients about the image, but that all happens outside the standard FTP transfer workflow.

Before moving on to evaluating this distribution method against the selection criteria, consume the registry image from your FTP server to get an idea of how clients would need to integrate.

First, eliminate the registry image from your local image cache and the file from your local directory:

```
rm registry.2.tar
docker image rm registry:2                    (1)
```

(1) Need to remove any registry containers first

Then download the image file from your FTP server using the ftp client:

```
docker container run --rm -it --network ch9_ftp \
    -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e 'cd pub/incoming; get registry.2.tar; exit' ftp-server
```

At this point you should once again have the registry.2.tar file in your local directory. You can reload that image into your local cache with the `docker load` command:

```
docker image load -i registry.2.tar
```

This is a minimal example of how a manual image publishing and distribution infrastructure might be built. With a bit of extension you could build a production-quality, FTP-based distribution hub. In its current configuration this example matches against the selection criteria, as shown in table 9.5.

Table 9.5 Performance of a sample FTP-based distribution infrastructure

Criteria	Rating	Notes
Cost	Good	This is a low-cost transport. All the related software is free. Bandwidth and storage costs should scale linearly with the number of images hosted and the number of clients.
Visibility	Worst	The FTP server is running in an unadvertised location with a non-standard integration workflow. The visibility of this configuration is very low.
Transport speed/size	Bad	In this example, all the transport happened between containers on the same computer, so all the commands finished quickly. If a client connects to your FTP service over the network, then speeds are directly impacted by your upload speeds. This distribution method will download redundant artifacts and won't download components of the image in parallel. Overall, this method isn't bandwidth-efficient.
Availability control	Best	You have full availability control of the FTP server. If it becomes unavailable, you're the only person who can restore service.
Longevity control	Best	You can use the FTP server created for this example as long as you want.
Access control	Worst	This configuration provides no access control.
Artifact integrity	Worst	The network transportation layer does provide file integrity between endpoints. But it's susceptible to interception attacks, and there are no integrity protections between file creation and upload or between download and import.
Secrecy	Worst	This configuration provides no secrecy.
Requisite experience	Good	All requisite experience for implementing this solution has been provided here. If you're interested in extending the example for production, you'll need to familiarize yourself with `vsftpd` configuration options and SFTP.

In short, there's almost no real scenario where this transport configuration is appropriate. But it helps illustrate the different concerns and basic workflows that you can create when you work with image as files. Try to imagine how replacing FTP with `scp` or `rsync` tooling using the SSH protocol would improve the system's performance for artifact integrity and secrecy. The final image distribution method we will consider distributes image sources and is both more flexible and potentially complicated.

9.5 Image source distribution workflows

When you distribute image sources instead of images, you cut out all the Docker distribution workflow and rely solely on the Docker image builder. As with manual image publishing and distribution, source-distribution workflows should be evaluated against the selection criteria in the context of a particular implementation.

Using a hosted source control system like Git on GitHub will have very different traits from using a file backup tool like `rsync`. In a way, source-distribution workflows have a superset of the concerns of manual image publishing and distribution workflows. You'll have to build your workflow but without the help of the `docker save`, `load`, `export`, or `import` commands. Producers need to determine how they will package their sources, and consumers need to understand how those sources are packaged as well as how to build an image from them. That expanded interface makes source-distribution workflows the most flexible and potentially complicated distribution method. Figure 9.9 shows image source distribution on the most complicated end of the spectrum.

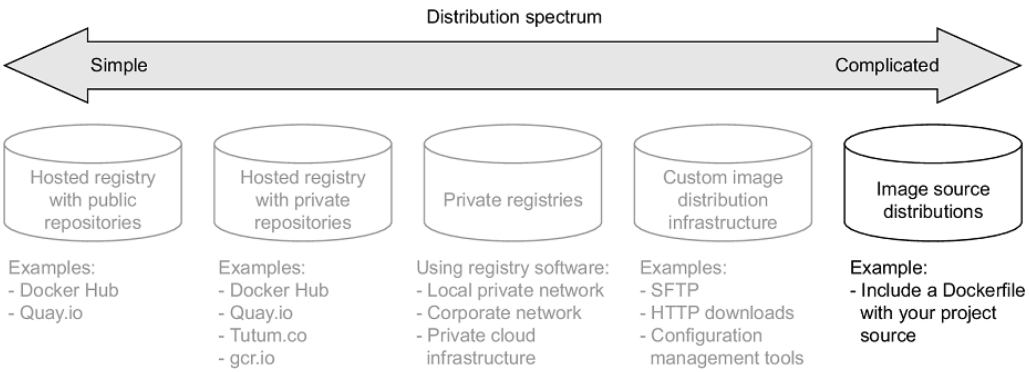


Figure 9.8 Using existing infrastructure to distribute image sources

Image source distribution is one of the most common methods, despite having the most potential for complication. Popular version-control software handles many of the complications of source distribution's expanded interface.

9.5.1 Distributing a project with Dockerfile on GitHub

When you use Dockerfile and GitHub to distribute image sources, image consumers clone your GitHub repository directly and use `docker image build` to build your image locally. With source distribution, publishers do not need an account on Docker Hub or another Docker registry to publish an image.

Supposing a producer had an existing project, Dockerfile, and GitHub repository, their distribution workflow would look like this:

```
git init
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
git add Dockerfile
# git add *whatever other files you need for the image*
git commit -m "first commit"
git remote add origin https://github.com/<your username>/<your repo>.git
git push -u origin master
```

Meanwhile, a consumer would use a general command set that looks like this:

```
git clone https://github.com/<your username>/<your repo>.git
cd <your-repo>
docker image build -t <your username>/<your repo> .
```

These are all steps that a regular Git or GitHub user is familiar with, as shown in table 9.6.

Table 9.6 Performance of image source distribution via GitHub

Criteria	Rating	Notes
Cost	Best	There's no cost if you're using a public GitHub repository.
Visibility	Best	GitHub is a highly visible location for open source tools. It provides excellent social and search components, making project discovery simple.
Transport speed/size	Good	By distributing image sources, you can leverage other registries for base layers. Doing so will reduce the transportation and storage burden. GitHub also provides a content delivery network (CDN). That CDN is used to make sure clients around the world can access projects on GitHub with low network latency.
Availability control	Worst	Relying on GitHub or other hosted version-control providers eliminates any availability control.
Longevity control	Bad	Although Git is a popular tool and should be around for a while, you forego any longevity control by integrating with GitHub or other hosted version-control providers.
Access control	Good	GitHub or other hosted version-control providers do provide access control

		tools for private repositories.
Artifact integrity	Good	This solution provides no integrity for the images produced as part of the build process, or of the sources after they have been cloned to the client machine. But integrity is the whole point of version-control systems. Any integrity problems should be apparent and easily recoverable through standard Git processes.
Secrecy	Worst	Public projects provide no source secrecy.
Requisite Experience	Good	Image producers and consumers need to be familiar with Dockerfile, the Docker builder, and the Git tooling.

Image source distribution is divorced from all Docker distribution tools. By relying only on the image builder, you're free to adopt any distribution toolset available. If you're locked into a particular toolset for distribution or source control, this may be the only option that meets your criteria.

9.6 Summary

This chapter covers various software distribution mechanisms and the value contributed by Docker in each. A reader that has recently implemented a distribution channel, or is currently doing so, might take away additional insights into their solution. Others will learn more about available choices. In either case, it is important to make sure that you have gained the following insights before moving on:

- Having a spectrum of choices illustrates your range of options.
- You should use a consistent set of selection criteria in order to evaluate your distribution options and determine which method you should use.
- Hosted public repositories provide excellent project visibility, are free, and require very little experience to adopt.
- Consumers will have a higher degree of trust in images generated by automated builds because a trusted third party builds them.
- Hosted private repositories are cost-effective for small teams and provide satisfactory access control.
- Running your own registry enables you to build infrastructure suitable for special use cases without abandoning the Docker distribution facilities.
- Distributing images as files can be accomplished with any file-sharing system.
- Image source distribution is flexible but only as complicated as you make it. Using popular source-distribution tools and patterns will keep things simple.

10

Image Pipelines

This chapter covers:

- The goals of Docker image pipelines
- Patterns for building images
- Common approaches for testing that images are configured correctly and secure
- Techniques for including image metadata to help consumers use your image
- Patterns for tagging images so they can be identified and delivered to consumers
- Patterns for publishing images to runtime environments and registries

Pipelines help software authors publish updates quickly and deliver new features and fixes to consumers efficiently. In Chapter 8, you learned how to build Docker images automatically using Dockerfile and the `docker build` command. However, building the image is merely one critical step in a longer process for delivering functioning and trustworthy images. Image publishers should perform tests to verify the image works under the expected operating conditions. Confidence in the correctness of the image artifact grows as it passes tests in the pipeline. Once an image passes its tests, it can finally be tagged and published to a registry for consumption. Consumers can deploy these images with confidence knowing that many important requirements have already been verified. These steps, preparing image material, building an image, testing, and finally publishing images to registries are together called an image build pipeline.

10.1 Goals of an image build pipeline

In this context, pipelines automate the process for building, testing, and publishing artifacts so they can be deployed to a runtime environment. Figure 10.1 illustrates the high-level process

for building software or other artifacts in a pipeline. This process should be familiar to anyone using continuous integration practices and is not specific to Docker images.

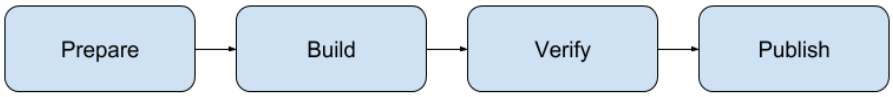


Figure 10.1 Generic artifact build pipeline

People often automate build pipelines with continuous integration tools like Jenkins, TravisCI, or Drone. Regardless of the specific pipeline modeling technology, the goal of a build pipeline is to apply a consistent set of rigorous practices in creating deployable artifacts from source definitions. Differences between the specific tools employed in a pipeline are simply an implementation detail. A continuous integration process for a Docker image is similar to other software artifacts and will look like:

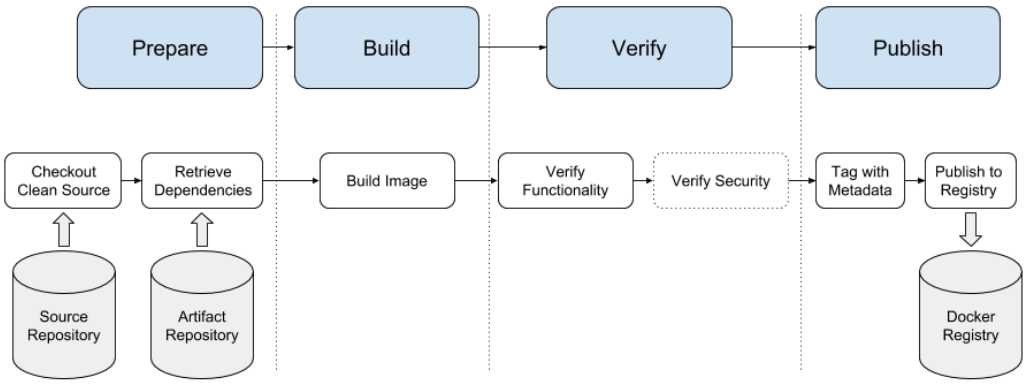


Figure 10.2 A Docker image build pipeline

When building a Docker image this process includes the following steps:

1. Checkout a clean copy of the source code defining the image and build scripts so the origin and process used to build the image is known
2. Retrieve or generate artifacts that will be included in the image such as the application package and runtime libraries
3. Build the image using a Dockerfile
4. Verify the image is structured and functions as intended
5. (Optional) Verify the image does not contain known vulnerabilities
6. Tag the image so that it can be consumed easily
7. Publish the image to a registry or another distribution channel

Application artifacts are the runtime scripts, binaries (exe, tgz, zip), and configuration files produced by software authors. This image build process assumes the application artifacts have already been built, tested, and published to an artifact repository for inclusion in an image. The application artifact may be built inside a container and this is how many modern continuous integration systems operate. The exercises in this chapter will show how to build applications using containers and how to package those application artifacts into a Docker image that runs application. We will implement the build process using a small and common set of tools available in Unix-like environments. This pipeline's concepts and basic commands should be easily transferrable into your organization's own tooling.

10.2 Patterns for building images

There are several patterns for building applications and images in using containers and we will discuss three of the most popular patterns here:

- All-in-one image to build and run the application
- Containerized application build using a build image with a separate, slimmer runtime image
- Slim runtime image with variations for debugging and other supplemental use cases using a multi-stage build

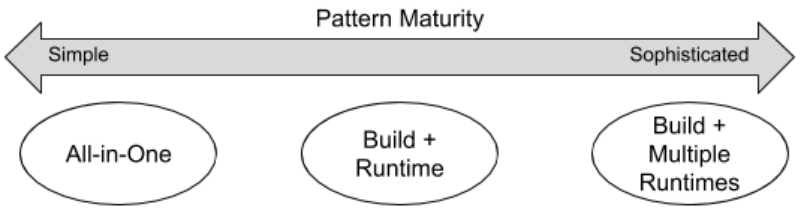


Figure 10.3 Image build pattern maturity

Multiple build patterns have evolved to produce images appropriate for particular consumption use cases. In this context, "maturity" refers to the of the design and process used to build the image, not the organization applying the pattern. When an image will be used for internal experimentation or as a portable development environment, the "all-in-one" pattern might be most appropriate. By contrast, when distributing a server that will be licensed and supported commercially, the "separate build plus multiple runtime images" pattern will probably be most appropriate. A single software publishing organization will often use multiple patterns for building images that they use and distribute. Apply and modify the patterns described here to solve your own image build and delivery problems.

10.2.1 All-in-one images

The tools might include SDKs, package managers, shared libraries, language-specific build tooling, or other binary tools. This type of image will also commonly include default

application runtime configuration. All-in-one images are the simplest way to get started containerizing an application. They are especially useful when containerizing a development environment or "legacy" application that has many dependencies.

Let's build a simple Java web server using the popular Spring Boot framework using this pattern. Here is an all-in-one Dockerfile that builds the application into an image along with the build tools:

```
FROM maven:3.5-jdk-10

ENV WORKDIR=/project
RUN mkdir -p ${WORKDIR}
COPY . ${WORKDIR}
WORKDIR ${WORKDIR}
RUN mvn -f pom.xml clean verify
RUN cp ${WORKDIR}/target/ch10-0.1.0.jar /app.jar

ENTRYPOINT ["java","-jar","/app.jar"]
```

Clone the https://github.com/dockerinaction/ch10_patterns-for-building-images.git repo and build the project with:

```
docker image build -t dockerinaction/ch10:all-in-one \
    --file all-in-one.df .
```

In this Dockerfile, the source image is the community Maven 3.5 image, which also includes OpenJDK 10. The Dockerfile builds a simple Java web server and the application artifact is added to the image. The image definition finishes with an ENTRYPOINT that runs the service by invoking `java` with the application artifact built in the image. This is about the simplest thing that could possibly work and a great approach for demonstrating "look, we can containerize our application!"

There are downsides to all-in-one images. Since all-in-one images contain more tools than are necessary to run the application, attackers have more options to exploit an application and images may need to update more frequently to accommodate change from a broad set of development and operational requirements. All-in-one images will be large, often 500MiB or more. The maven:3.5-jdk-10 base image used in the example is 987MB to start with and the final image is 1.06GB. Large images put more stress on image distribution mechanisms, though this problem is relatively innocuous until you get to large scale or very frequent releases.

This approach is good for creating a portable application image or development environment with little effort. The next pattern will show how to improve many characteristics of the runtime image by separating application build and runtime concerns.

10.2.2 Separate build and runtime images

The all-in-one pattern can be improved on by creating separate build and runtime images. Specifically, in this approach all of the application build and test tooling will be included in one image, and the other will only contain what the application requires at runtime.

We can build the application with a Maven container like:

```
docker container run -it --rm \
  -v "$(pwd)":/project/ \
  -w /project/ \
  maven:3.5-jdk-10 \
  mvn clean verify
```

Maven compiles and packages the application artifact into the project's `target` directory:

```
$ ls -la target/ch10-0.1.0.jar
-rw-r--r--  1 user  group  16142348 Nov 16 18:28 target/ch10-0.1.0.jar
```

In this approach, the application is built using a container created from the public Maven image. The application artifact is output to the host filesystem via the volume mount instead of storing it in the build image as in the all-in-one pattern. The runtime image is created using a very simple Dockerfile that COPYs the application artifact into an image based on OpenJDK 10:

```
FROM openjdk:10-jdk-slim

COPY target/ch10-0.1.0.jar /app.jar

ENTRYPOINT ["java","-jar","/app.jar"]
```

Build the runtime image with:

```
docker image build -t dockerinaction/ch10:simple-runtime \
    --file simple-runtime.df .
```

Run the web server image with:

```
docker container run --rm -it dockerinaction/ch10:simple-runtime
```

The application runs just like it did in the previous all-in-one example. With this approach, the build-specific tools such as Maven and intermediate artifacts are no longer included in the runtime image. The runtime image is now much smaller (515MiB vs 1GiB!) and has a smaller attack surface.

10.2.3 Create variations of application runtime image using multi-stage builds

As your build and operational experience matures, you may find it useful to create small variations of an application image to support use cases such as debugging, specialized testing, or profiling. These use cases often require addition of specialized tools or changes to the application's image. Multi-stage builds can be used to keep the specialized image

synchronized with the application image and avoid duplication of image definitions. In this section, we will focus on the pattern of creating specialized images using the multi-stage features of the `FROM` instruction introduced in Chapter 8.

Let's build a 'debug' variation of our app-image based on our application image. The hierarchy will look like:

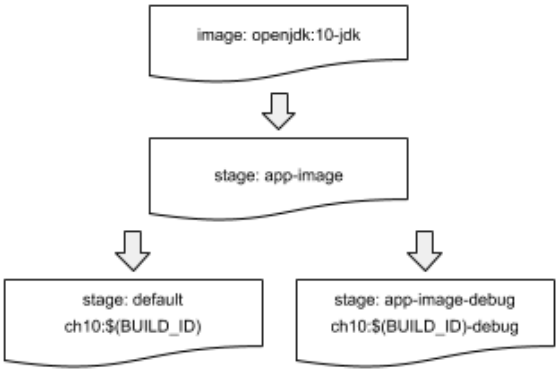


Figure 10.4 Image Hierarchy for Multi-Stage Build Example

The multi-stage-runtime.df in the chapter's example repository implements this hierarchy:

```
# The app-image build target defines the application image
FROM openjdk:10-jdk-slim as app-image                    (1)

ARG BUILD_ID=unknown
ARG BUILD_DATE=unknown
ARG VCS_REF=unknown

LABEL org.label-schema.version="${BUILD_ID}" \
      org.label-schema.build-date="${BUILD_DATE}" \
      org.label-schema.vcs-ref="${VCS_REF}" \
      org.label-schema.name="ch10" \
      org.label-schema.schema-version="1.0rc1"

COPY multi-stage-runtime.df /Dockerfile

COPY target/ch10-0.1.0.jar /app.jar

ENTRYPOINT ["java","-jar","/app.jar"]

FROM app-image as app-image-debug                        (2)
#COPY needed debugging tools into image
ENTRYPOINT ["sh"]

FROM app-image as default                                (3)
```

(1) app-image build stage starts from openjdk
(2) app-image-debug stage inherits and adds to the app-image

❸ default stage ensures app-image is produced by default

The build stage of the main application image is declared as starting from `openjdk:10-jdk` and named `app-image`:

```
# The app-image build target defines the application image
FROM openjdk:10-jdk as app-image
…
```

Naming a build stage serves two important purposes. First, the stage name enables other build stages within the Dockerfile to use another stage easily. Second, build processes can build a stage by specifying that name as a build target. Build stage names are localized to the context of a Dockerfile and do not affect image tagging.

Let's define a variation of the application image by adding a build stage to the Dockerfile that supports debugging:

```
FROM app-image as app-image-debug  ❶
#COPY needed debugging tools into image
ENTRYPOINT ["sh"]
```

❶ Use app-image as the base for the debug image

The debug application image definition specifies app-image as its base image and demonstrates making minor changes. In this case, the only change is to reconfigure the image's entrypoint to be a shell instead of running the application. The debug image is otherwise identical to the main application image.

The `docker image build` command produces a single image regardless of how many stages are defined in the Dockerfile. You can use the build command's `--target` option to select the stage to build the image. When you define multiple build stages in a Dockerfile, it is best to be explicit about what image you want to build. To build the debug image, invoke docker build and target the `app-image-debug` stage:

```
docker image build -t ch10_multi-stage-runtime:debug \
       -f multi-stage-runtime.df \
       --target=app-image-debug
```

The build process will execute the app-image-debug stage as well as the app-image stage it depends on to produce the debug image.

Note that when you build an image from a Dockerfile that defines multiple stages and do *not* specify a build target, Docker will build an image from the last stage defined in the Dockerfile. You can build an image for the main build stage defined in your Dockerfile by adding a trivial build stage at the end of the Dockerfile like:

```
# Ensure app-image is the default image built with this Dockerfile
FROM app-image as default
```

This `FROM` statement defines a new build stage named `default` that is based on the `app-image`. The default stage makes no additions to the last layer produced by app-image and is thus identical.

Now that we have covered several patterns for producing an image or family of closely related images, let's discuss what metadata we should capture along with our images to facilitate delivery and operational processes.

10.3 Record metadata at image build time

As described in Chapter 8, images can be annotated with metadata that is useful to consumers and operators using the `LABEL` instruction. You should use labels to capture at least the following data in your images:

- Application name
- Application version
- Build date and time
- Version control commit identifier

In addition to image labels, consider adding the Dockerfile used to build the image and software package manifests to the image filesystem.

All this information is highly valuable when orchestrating deployments and debugging problems. Orchestrators can provide traceability by logging metadata to audit logs. Deployment tools can visualize the composition of a service deployment using build time or VCS commit. Including the source Dockerfile in the image can be a quick reference for people debugging a problem to navigate within a container. Orchestrators may find other metadata describing the image's architectural role or security profile useful in deciding where the container should run.

The Docker community Label Schema project has defined commonly used labels at http://label-schema.org/. Representing the recommended metadata using the label schema and build arguments in Dockerfile looks like:

```
FROM openjdk:10-jdk

ARG BUILD_ID=unknown
ARG BUILD_DATE=unknown
ARG VCS_REF=unknown

LABEL org.label-schema.version="${BUILD_ID}" \
      org.label-schema.build-date="${BUILD_DATE}" \
      org.label-schema.vcs-ref="${VCS_REF}" \
      org.label-schema.name="ch10" \
      org.label-schema.schema-version="1.0rc1"

COPY multi-stage-runtime.df /Dockerfile

COPY target/ch10-0.1.0.jar /app.jar

ENTRYPOINT ["java","-jar","/app.jar"]
```

Our build process is more complicated now that we have more steps: gather metadata, build application artifacts, build image. Let's orchestrate the build process with a time-tested build tool, `make`.

10.3.1 Orchestrating the build with make

`make` is a tool used to build programs that understands dependencies between steps of a build process. Build process authors describe each step in a `Makefile` that `make` interprets and executes to complete the build. The `make` tool provides a flexible shell-like execution environment, so you can implement virtually any kind of build step. The primary advantage of `make` over a standard shell script is that users declare dependencies between steps rather than implementing the flow of control between steps directly. These steps are called rules and each rule is identified by a target name. The general form of a `make` rule is:

```
target … : prerequisites …  (1) (2)
        recipe command 1    (3)
        recipe command 2
        …
```

(1) target identifies the rule with a logical name or file name produced by the rule
(2) prerequisites is an optional list of targets to build before this target
(3) The recipe section contains the list of commands used to build the target

When you run the `make` command, `make` constructs a dependency graph from the prerequisites declared for each rule. `make` calculates the sequence of steps to build a specified target using the dependency graph. `make` has many features and quirks that we will not describe here, but you can read more about at https://www.gnu.org/software/make/manual/. One item of note is that `make` is famous for its sensitivity to whitespace characters, particularly tabs for indentation and spaces around variable declarations. You will probably find it easiest to use the `Makefile` provided in this chapter's source repository (https://github.com/dockerinaction/ch10_patterns-for-building-images.git) instead of typing them in yourself. With our `make` primer complete, let's return to building our Docker images.

Building on Windows

If you are using Windows, you will probably find that `make` and several other commands used in this example are not available in your environment. The easiest solution will probably be to use a Linux virtual machine either locally or in the Cloud. If you plan to develop software using Docker on Windows, you should also investigate using the Windows Subsystem for Linux (WSL) with Docker for Windows.

Here is a `Makefile` that will gather metadata, build, test, and tag the application artifact and images:

```
# if BUILD_ID is unset, compute metadata that will be used in builds
ifeq ($(strip $(BUILD_ID)),)
    VCS_REF := $(shell git rev-parse --short HEAD)
```

```
    BUILD_TIME_EPOCH := $(shell date +"%s")
    BUILD_TIME_RFC_3339 := $(shell date -u -r $(BUILD_TIME_EPOCH) '+%Y-%m-
      %dT%I:%M:%SZ')
    BUILD_TIME_UTC := $(shell date -u -r $(BUILD_TIME_EPOCH) +'%Y%m%d-%H%M%S')
    BUILD_ID := $(BUILD_TIME_UTC)-$(VCS_REF)
endif

ifeq ($(strip $(TAG)),)
    TAG := unknown
endif

.PHONY: metadata
metadata:
    @echo "Gathering Metadata"
    @echo BUILD_TIME_EPOCH IS $(BUILD_TIME_EPOCH)
    @echo BUILD_TIME_RFC_3339 IS $(BUILD_TIME_RFC_3339)
    @echo BUILD_TIME_UTC IS $(BUILD_TIME_UTC)
    @echo BUILD_ID IS $(BUILD_ID)

target/ch10-0.1.0.jar:
    @echo "Building App Artifacts"
    docker run -it --rm  -v "$(shell pwd)":/project/ -w /project/ \
    maven:3.5-jdk-10 \
    mvn clean verify

.PHONY: app-artifacts
app-artifacts: target/ch10-0.1.0.jar

.PHONY: lint-dockerfile
lint-dockerfile:
    @set -e
    @echo "Linting Dockerfile"
    docker container run --rm -i hadolint/hadolint:v1.15.0 < multi-stage-runtime.df

.PHONY: app-image
app-image: app-artifacts metadata lint-dockerfile                ❶
    @echo "Building App Image"
    docker image build -t dockerinaction/ch10:$(BUILD_ID) \
    -f multi-stage-runtime.df \
    --build-arg BUILD_ID='$(BUILD_ID)' \
    --build-arg BUILD_DATE='$(BUILD_TIME_RFC_3339)' \
    --build-arg VCS_REF='$(VCS_REF)' \
    .
.PHONY: app-image-debug
app-image-debug: app-image
    @echo "Building Debug App Image"
    docker image build -t dockerinaction/ch10:$(BUILD_ID)-debug \
    -f multi-stage-runtime.df \
    --target=app-image-debug \
    --build-arg BUILD_ID='$(BUILD_ID)' \
    --build-arg BUILD_DATE='$(BUILD_TIME_RFC_3339)' \
    --build-arg VCS_REF='$(VCS_REF)' \
    .

.PHONY: image-tests
image-tests:
    @echo "Testing image structure"
```

```
    docker container run --rm -it \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v $(shell pwd)/structure-tests.yaml:/structure-tests.yaml \
    gcr.io/gcp-runtimes/container-structure-test:v1.6.0 test \
    --image dockerinaction/ch10:$(BUILD_ID) \
    --config /structure-tests.yaml

.PHONY: inspect-image-labels
inspect-image-labels:
    docker image inspect --format '{{ json .Config.Labels }}'
      dockerinaction/ch10:$(BUILD_ID) | jq

.PHONY: tag
tag:
    @echo "Tagging Image
    docker image tag dockerinaction/ch10:$(BUILD_ID) dockerinaction/ch10:$(TAG)

.PHONY: all
all: app-artifacts app-image image-tests                    (2)
```

(1) The app-image target requires building the app-artifacts, metadata, and linting target
(2) You can build everything with `make all`

This `Makefile` defines a target for each of the build steps we have discussed: gathering metadata, building the application, and building, testing, and tagging the image. Targets like `app-image` have dependencies on other targets to ensure that steps execute in the right order. Since build metadata is essential for all steps, it is generated automatically unless a BUILD_ID is provided. The `Makefile` implements an image pipeline that you can run locally or use within a CI/CD system. You can build the application artifacts and image by making the `app-image` target:

```
make app-image
```

Making the application artifacts will produce a lot of output as dependencies are retrieved and then code compiled. However, the application build should indicate success with a message like:

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Immediately following that you should see a `Gathering Metadata` message followed by metadata for this build:

```
BUILD_TIME_EPOCH IS 1546193546
BUILD_TIME_RFC_3339 IS 2018-12-30T06:12:26Z
BUILD_TIME_UTC IS 20181230-181226
BUILD_ID IS 20181230-181226-61ceb6d
```

The next step in the build process is the first quality assurance step for our image. You should see a message like:

```
Linting Dockerfile
docker container run --rm -i hadolint/hadolint:v1.15.0 < multi-stage-runtime.df
```

Before building the image, the Dockerfile is analyzed by a linting tool named `hadolint` (https://github.com/hadolint/hadolint). The linter checks Dockerfiles to verify they follow best practices and identify common mistakes. Hadolint is one of several linters available for Dockerfiles. Because hadolint parses the Dockerfile into an abstract syntax tree, it is able to perform deeper and more complex analysis than approaches based on regular expressions. Hadolint identifies incorrectly specified or deprecated Dockerfile instructions, omitting a tag in the FROM image instruction, common mistakes when using `apt`, `apk`, `pip`, and `npm` package managers, and other commands specified in `RUN` instructions.

Once the Dockefile has been linted, the app-image target executes and builds the application image. You should see indication of success without from the docker image build command similar to:

```
Successfully built a6ba2274b5c2
Successfully tagged dockerinaction/ch10:20181230-181226-61ceb6d
```

In this build process, each application image is tagged with a BUILD_ID computed from the time of the build and the current git commit hash. The fresh Docker image is tagged with the repository and BUILD_ID, `20181230-181226-61ceb6d` in this case. The `20181230-181226-61ceb6d` tag now identifies Docker image id `a6ba2274b5c2` in the `dockerinaction/ch10` image repository. This style of BUILD_ID identifies the image with a high degree of precision in both wall clock time and version history. Capturing the time of an image build is an important practice because people understand time well and many image builds will perform software package manager updates or other operations that may not produce the same result from build to build. Including the version control id, `7c5fd3d`, provides a convenient pointer back to the original source material used to build the image.

You can inspect the metadata that was added to the image by inspecting the LABELs using the command:

```
make inspect-image-labels BUILD_ID=20181230-181226-61ceb6d
```

This command uses `docker image inspect` to show the image's labels:

```
docker image inspect --format '{{ json .Config.Labels }}'
{
  "org.label-schema.build-date": "2018-12-30T06:12:26Z",
  "org.label-schema.name": "ch10",
  "org.label-schema.schema-version": "1.0rc1",
  "org.label-schema.vcs-ref": "61ceb6d",
  "org.label-schema.version": "20181230-181226-61ceb6d"
}
```

The application image is now ready for further testing and tagging prior to release. The image has a unique BUILD_ID tag that will conveniently identify the image through the rest of the

delivery process. In the next section, we will examine some ways to test that an image has been constructed correctly and is ready for deployment.

10.4 Testing images in a build pipeline

Image publishers can use several techniques in their build pipelines to build confidence in the produced artifacts. The Dockerfile linting step described in the previous section is one quality assurance technique, but we can go further. One of the principal advantages of the Docker image format is that image metadata and the filesystem can be analyzed by tools easily. For example, the image can be tested to verify it contains files required by the application, those files have appropriate permissions, and by executing key programs to verify they run correctly. Docker images can be inspected to verify traceabilty and deployment metadata has been added. Security conscious users can scan the image for vulnerabilities. Publishers can stop the image delivery process if any of these steps fail and together these steps raise the quality of published images significantly.

One popular tool for verifying the construction of a Docker image is the Container Structure Test tool from Google (https://github.com/GoogleContainerTools/container-structure-test). With the Container Structure Test tool (CST), authors can verify an image (or image tarball) contains files with desired file permissions and ownership, commands execute with expected output, and the image contains particular metadata such as a label or command. Many of these inspections could be done by a traditional system configuration inspection tool such as Inspec or ServerSpec. However, Container Structure Test's approach is more appropriate for containers as the tool operates on arbitrary images without requiring any tooling or libraries to be included inside the image. Let's verify the application artifact has the proper permissions and the proper version of Java is installed by executing CST with the following configuration:

```
schemaVersion: "2.0.0"

# Verify the expected version of Java is available and executable
commandTests:
  - name: "java version"
    command: "java"
    args: ["-version"]
    exitCode: 0
    # OpenJDK java -version stderr will include a line like:
    # OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
    expectedError: ["OpenJDK Runtime Environment.* 10.*"]

# Verify the application archive is readable and owned by root
fileExistenceTests:
- name: 'application archive'
  path: '/app.jar'
  shouldExist: true
  permissions: '-rw-r--r--'
  uid: 0
  gid: 0
```

First, this configuration tells CST to invoke Java and output the version information. The OpenJDK Java runtime prints its version information to stderr, so CST is configured to match that string against the `OpenJDK Runtime Environment.* 10.*` regular expression. If we needed to ensure the application ran against a specific version of Java, the regex could be made more specific and the base image updated to match. Second, CST will verify that the application archive is at /app.jar, owned by root, and readable by everyone. Verifying file ownership and permissions might seem basic but helps prevent basic problems with 'invisible' bugs due to programs not being executable, readable, or in the executable PATH. Execute the image tests against the image you built earlier with a command like:

```
make image-tests BUILD_ID=20181230-181226-61ceb6d
```

This command should produce a successful result that looks like:

```
Testing image structure
docker container run --rm -it \
      -v /var/run/docker.sock:/var/run/docker.sock \
      -v /Users/dia/structure-tests.yaml:/structure-tests.yaml \
      gcr.io/gcp-runtimes/container-structure-test:v1.6.0 test \
      --image dockerinaction/ch10:20181230-181226-61ceb6d \
      --config /structure-tests.yaml

============================================
====== Test file: structure-tests.yaml ======
============================================

INFO: stderr: openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Debian-2, mixed mode)

=== RUN: Command Test: java version
--- PASS
stderr: openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Debian-2, mixed mode)
INFO: File Existence Test: application archive
=== RUN: File Existence Test: application archive
--- PASS

============================================
================= RESULTS ==================
============================================
Passes:      2
Failures:    0
Total tests: 2

PASS
```

Many image authors want to scan images for vulnerabilities prior to publishing them and halt the delivery process when a significant vulnerability exists. We will give a quick overview of how these systems work and how they are typically integrated into an image build pipeline. There are a number of image vulnerability scanning solutions available from both commercial

and community sources. In general, image vulnerability scanning solutions rely on a lightweight scanning client program that runs in the image build pipeline. The scanning client examines the contents of the image and compares the software package metadata and filesystem contents to vulnerability data retrieved from a centralized vulnerability database or API. Most of these scanning systems require registration with the vendor to use the service so we will not integrate any of the tools into this image build workflow. After choosing an image scanning tool, it should be easy to add another target to the build process.

Features of a General Vulnerability Scanning and Remediation Workflow

Using a scanner to identify vulnerabilities in a single image is the first step and most critical step in publishing images without vulnerabilities. The leading container security systems cover a wider set of scanning and remediation use cases than discussed in the image build pipeline example. These systems incorporate vulnerability feeds with low-false positive rates, integrate with the organization's Docker registries to identify issues in images that have already been published or were built by an external source, and notify maintainers of the base image or layer with a vulnerability to speed remediation. When evaluating container security systems, pay special attention to these features and how each solution will integrate with your delivery and operational processes.

10.5 Patterns for tagging images

Once an image has been tested and is deemed ready for deployment in the next stage of delivery, the image should be tagged so that it is easy for consumers to find and use it. There are several schemes for tagging images and some are better for certain consumption patterns than others. The most important image tagging features to understand are:

- tags are human-readable strings that point to a particular content addressable image id
- multiple tags may point to a single image id
- tags are *mutable* and may be moved between images in a repository or removed entirely

You can use all of these features to construct a scheme that works for an organization, but there is not a single scheme in use or only a single way to do it. Certain tagging schemes will work well for certain consumption patterns and not others.

10.5.1 Background

Docker image tags are mutable. An image repository owner can remove a tag from an image id or move it from one id to another. Image tag mutation is commonly used to identify the latest image in a series. The `latest` tag is extensively within the Docker community to identify the most recent build of an image repository. However, the `latest` tag causes a lot of confusion because there is no real agreement on what `latest` means. Depending on the image repository or organization, any of the following are valid answers to what does the `latest` tag identify?

- The most recent image built by the continuous integration system, regardless of source

control branch

- The most recent image built by the continuous integration system, from the main release branch
- The most recent image built from the stable release branch that has passed all the author's tests
- The most recent image built from an active development branch that has passed all the author's tests
- Nothing! Because the author has never pushed an image tagged latest or has not done so recently

Even trying to define `latest` prompts many questions. When adopting an image release tagging scheme, be sure to specify what the tag does and does not mean in your own context. Since tags can be mutated, you will also need to decide if and when consumers should pull images to receive updates to an image tag that already exists on the machine.

Common tagging & deploy schemes:

- Continuous Delivery with Unique Tags: Pipelines promote a single image with a unique tag through delivery stages
- Continuous Delivery with Environment Specific Artifacts: Pipelines produce environment-specific artifacts and promote them through dev, stage, prod
- Semantic Versioning: Tag and publish images with a Major.Minor.Patch scheme that communicates level of change in a release

10.5.2 Continuous Delivery with Unique Tags

The 'unique tag' scheme is a common and simple way to support continuous delivery of an application. In this scheme, the image is built and deployed into an environment using the unique BUILD_ID tag. When people or automation decide that this version of the application is ready for promotion to the next environment, they run a deployment to that environment with the unique tag.

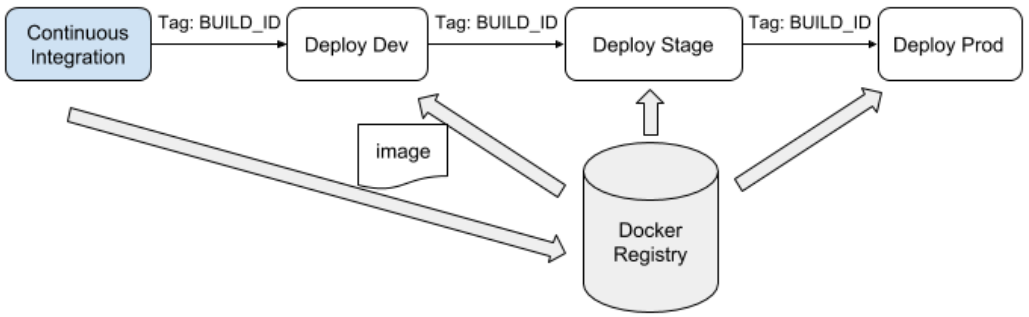


Figure 10.5 Continuous Delivery with Unique Tags

This scheme is simple to implement and supports continuous delivery of applications that use a linear release model without branching. The main disadvantage of this scheme is that people must deal with precise build identifiers instead of being able to say 'latest' or a 'dev' tag. Since an image may be tagged multiple times, many teams apply and publish additional tags such as 'latest' to provide a convenient way to consume the most recent image.

10.5.3 Configuration image per deployment stage

Some organizations package software releases into a distinct artifact for each stage of deployment. These packages are then deployed to dedicated internal environments for integration testing and have names like 'dev' and 'stage.' Once the software has been tested in internal environments, the production package is deployed to the production environment. We could create a Docker image for each environment. Each image would include both the application artifact and environment-specific configuration. However, this is an anti-pattern because the main deployment artifact is built multiple times and usually not tested prior to production.

A better way to support deployment to multiple environments is to create two kinds of images:

1. A generic, environment-agnostic application image
2. A set of environment-specific configuration images, with each image containing the environment-specific configuration files for that environment

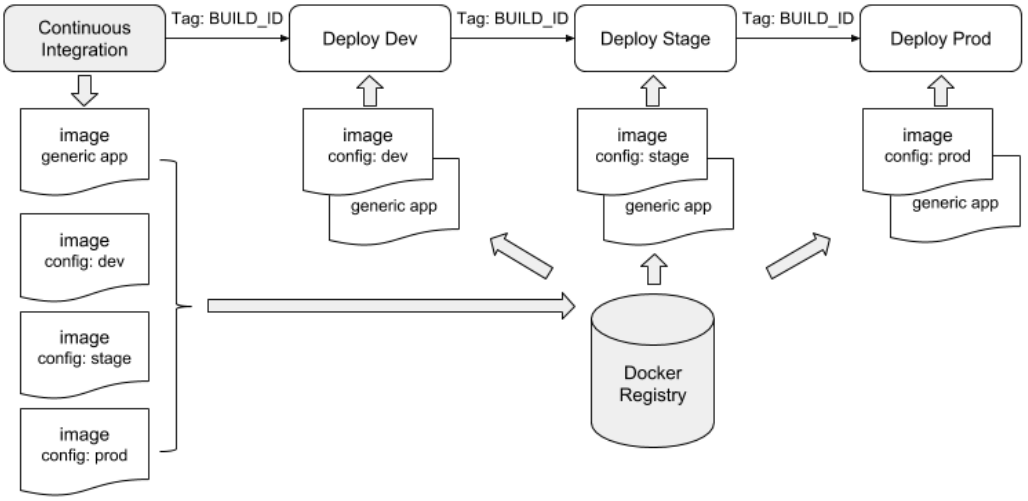


Figure 10.6 Configuration image per environment

The generic application and environment-specific configuration images should be built at the same time and tagged with the same BUILD_ID. The deployment process identifies the software and configuration for deployment using the BUILD_ID as described in the continuous delivery case. At deployment time, two containers are created. First, a configuration container is created from the environment-specific configuration image. Second, the application container is created from the generic application image and that container mounts the config container's filesystem as a volume. Consuming environment-specific files from a config container's filesystem is a popular application orchestration pattern and a variation of 12-factor application principles (https://12factor.net/). In Chapter 12, you will see how Docker Swarm supports environment-specific configuration of Docker services as a first-class feature of orchestration without using a secondary image.

This approach enables software authors and operators to support environment specific variation while maintaining traceability back to the originating sources and preserving a simple deployment workflow.

10.5.4 Semantic Versioning

Semantic versioning (https://semver.org/) is a popular approach to versioning artifacts with a version number of the form `MAJOR.MINOR.PATCH`. The semantic versioning specification defines that as software changes, authors should increment the:

1. Major version when making incompatible API changes,
2. Minor version when adding functionality in a backwards-compatible manner
3. Patch version when making backwards-compatible bug fixes.

Semantic versioning helps both publishers and consumers manage expectations for what kind of changes a consumer is getting when updating an image dependency. Authors who publish images to a large number of consumers or who must maintain several release streams for a long time often find semantic versioning or similar scheme attractive. Semantic versioning is a good choice for images that many people depend on as a base Operating System, language runtime, or database.

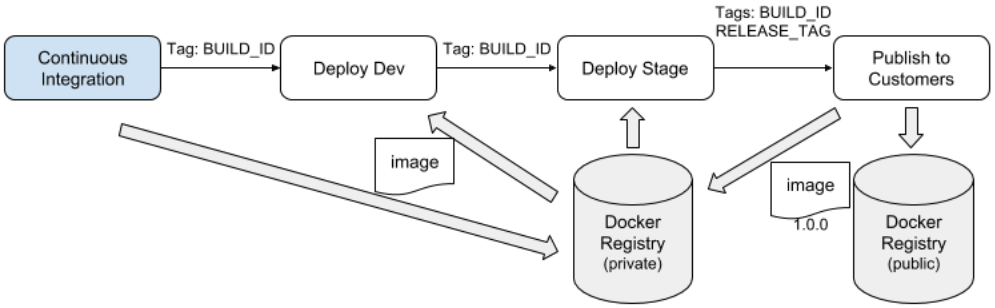


Figure 10.7 Tag and Publish Image Release with Semantic Versioning

Suppose that after testing our image in dev and stage that we want to release our recent build of the example app as version 1.0.0 to our customers. We can use the BUILD_ID to identify our image and tag it with 1.0.0 using:

```
make tag BUILD_ID=20181230-181226-61ceb6d TAG=1.0.0
```

Tagging the image as version `1.0.0` signals that we are ready to maintain backwards compatibility in the operation of the software. Now that you have tagged the image, you can push it to a registry for distribution. You may even choose to publish to multiple registries. Use multiple registries to keep images intended for internal use private and only publish official releases to the public registry for consumption by customers.

No matter what the scheme for identifying an image to promote, once the decision to promote the image is made, a promotion pipeline should resolve semantic tags (latest, dev, 7) to a unique tag or content addressable identifier and deploy *that* image. This ensures that if the tag being promoted is moved to another image in the meantime, the image that people decided to promote is deployed instead of merely whatever image the tag is associated with at the time of deployment.

10.6 Summary

This chapter covered common goals, patterns, and techniques used to build and publish applications in Docker images. The options described in this chapter illustrate the range of options available when creating image delivery processes. With this foundation, you should be able to navigate, select, and customize options that are appropriate for delivering your own applications as Docker images. The key points to understand from this chapter are:

- Pipelines for building images have the same structure and goals for assuring quality of Docker images as other software and infrastructure build pipelines.
- Tools for detecting bugs, security problems, and other image construction problems exist and can be incorporated into image build pipelines easily.

- Codify the image build process using a build tool such as `make` and use that process in local development and CI/CD processes.
- There are several patterns for organizing Docker image definitions and these patterns provide tradeoffs in managing application build and deployment concerns such as attack surface and image size versus sophistication.
- Information about the source and build process of an image should be recorded as image metadata to support traceability, debugging, and orchestration activities when deploying images.
- Docker image tags provide a foundation for delivering software to consumers using styles ranging from continuous delivery in a private service deployment to publishing long-lived releases using semantic versioning to the public.

Index

- (hyphen) character[(hyphen) character], 136

.

.dockerignore file[dockerignore file], 23, 25, 146

A

access control, distribution method and, 173
ADD instruction, 149, 151
Apache Cassandra project, 67, 69
apache2, 41
AppArmor, 115
apt-get tool, 122
automation. See build automatoin, 169
availability, distribution method and, 173
aws program, 152

B

bind mount volumes, 62
boot2docker ip, 99
boot2docker ssh command, 108
bridged containers
  custom name resolution, 90, 94
bridges, 80
build automation, 141, 169
  Dockerfile, 145, 152
    file system instructions, 149, 152
    metadata instructions, 145, 149
    packaging Git with, 142, 145
  hardened application images, 164, 169
    content addressable image identifiers, 164, 165
    SUID and SGID permissions, 167, 169
    user permissions, 165, 167
  injecting downstream build-time behavior, 155
  using startup scripts and multiprocess containers, 159, 164
    environmental preconditions validation, 159, 161

initialization processes, 161, 164

C

-c flag[c flag], 136
CAIID (content addressable image identifier), 164
capabilities, feature access using, 113
--cap-add flag[cap-add flag], 112
Cassandra project, 67, 69
CMD instruction, 149, 150, 152, 175
confidentiality, distribution method and, 173, 174
containers, 17, 44
  bridged
    custom name resolution, 90, 94
  building environment-agnostic systems, 32, 38
    environment variable injection, 35, 38
    read-only file systems, 32, 35
  building images from, 120, 126
    committing new image, 124, 125
    configurable image attributes, 125, 126
    packaging Hello World, 121, 122
    preparing packaging for Git, 122, 123
    reviewing file system changes, 123
  cleaning up, 42
  container file system abstraction and isolation, 57
  container-independent data management, 66, 67
  creating, 20
  durable, building, 38, 42
    automatically restarting containers, 39, 40
    keeping containers running with supervisor and startup processes, 40, 42
  eliminating metaconflicts, 26, 32
    container state and dependencies, 29, 32
    flexible container identification, 26, 29
  interactive, running, 21
  inter-container dependencies, 95
    link nature and shortcomings, 95
  multiprocess, 159, 164
  networking
    local Docker network topology, 82
  output of, 21
  PID namespace and, 23, 25
  running with full privileges, 113, 114
  sharing IPC primitives between, 104

use-case-appropriate containers, 116, 118
applications, 116, 117
high-level system services, 117
low-level system services, 117, 118
with enhanced tools, 114
specifying additional security options, 115, 116
content addressable image identifier. See CAIID, 164
COPY instruction, 149, 151
CPU, limits on resources, 99, 101
--cpuset-cpus flag[cpuset-cpus flag], 101
--cpu-shares flag[cpu-shares flag], 99
cron tool, 117

D

daemons, 19
dbus tool, 117
dependencies
container state and, 29, 32
inter-container
link nature and shortcomings, 95
--detach flag[detach flag], 20
--device flag[device flag], 102
distribution
choosing method of, 171, 174
distribution spectrum, 171
selection criteria, 172, 174
image source distribution workflows, 190, 192
manual image publishing and distribution, 184
private registries, 179, 184
consuming images from, 183, 184
using registry image, 182, 183
using hosted registries, 174, 179
private hosted repositories, 177, 179
public repositories, 175, 177
DNS (Domain Name System), 90
Docker
containers and, 6
overview, 3
problems solved by, 9, 13
getting organized, 10
protecting computer, 13
use of, where/when, 15

docker build command, 132, 143, 146, 185, 191
docker command-line tool, 15
docker commit command, 124, 126, 130, 132
docker cp command, 135
docker create command, 97
  --cpu-shares flag, 99
  --privileged flag, 113
  --security-opt flag, 115
  --volume flag, 150
docker diff command, 128
docker exec command, 24
docker export command, 135, 185
docker history command, 134
Docker Hub, 5, 19, 54, 174
docker images command, 54, 55, 124
docker import command, 136, 185
docker inspect command, 32, 73, 105, 148
docker kill command, 43
docker load command, 51, 185, 189
docker login command, 175, 177
docker logs command, 22
docker ps -a, 42
docker ps command, 21, 30, 42
docker pull command, 48, 54, 55
docker push command, 175
docker rename command, 27
docker rm command, 43
docker rm -f command, 43
docker rmi command, 56
docker run command, 48, 97
  --cpu-shares flag, 99
  flags of, 91, 93
  --privileged flag, 113
  --security-opt flag, 115
  --volume flag, 150
docker save command, 185
docker start command, 30
docker stop command, 23, 43
docker tag command, 56, 132, 133, 137
docker top command, 40
Dockerfile, 145, 152
  file system instructions, 149, 152
  installing software from, 54

metadata instructions, 145, 149
packaging Git with, 142, 145
Domain Name System. See DNS, 90
downstream build-time behavior, 155

E

--email flag[email flag], 175
--entrypoint flag[entrypoint flag], 41, 124
ENTRYPOINT instruction, 144, 148, 149, 150
entrypoints, 41
--env (-e) flag[env (-e) flag], 36, 147
ENV instruction, 147, 148, 149
environment-agnostic systems, building, 32, 38
environment variable injection, 35, 38
read-only file systems, 32, 35
Ethernet interface, 78
exporting, flat file systems, 135, 137
EXPOSE command, 148, 149

F

-f flag[f flag], 43
f option, 104
features, 113
--file (-f) flag[file (-f) flag], 143
file systems
changes to, reviewing, 123
changes to, reviewing, 123
flat, importing and exporting, 135, 137
structure of, 57, 58
flat file systems, 135, 137
--format (-f) option[format (-f) option], 105
FROM instruction, 148, 153, 155, 165, 175
FTP (File Transfer Protocol), sample distribution infrastructure using, 187
ftp-transport container, 188
full privileges, running containers with, 113, 114

G

Git
packaging with Dockerfile, 142, 145
preparing packaging for, 122, 123
gosu program, 167

H

hardened application images, building, 164, 169
  content addressable image identifiers, 164, 165
  SUID and SGID permissions, 167, 169
  user permissions, 165, 167
Hello World, packaging, 121, 122
high-level system services, 117
hosted registries, distribution, 174, 179
  private hosted repositories, 177, 179
  public repositories, 175, 177
--hostname flag[hostname flag], 91, 93
HTTP (Hypertext Transfer Protocol), 78
Hypertext Transfer Protocol See HTTP, 78
hyphen (-) character, 136

I

id command, 105
image layers, 54, 56
image-dev container, 124
images, 133, 139
  building from containers, 120, 126
    committing new image, 124, 125
    configurable image attributes, 125, 126
    packaging Hello World, 121, 122
    preparing packaging for Git, 122, 123
    reviewing file system changes, 123
  consuming from private registries, 183, 184
  exporting and importing flat file systems, 135, 137
  hardened application images, 169
    content addressable image identifiers, 164, 165
    SUID and SGID permissions, 167, 169
    user permissions, 165, 167
  loading as files, 50, 51
  size of, 133, 135
  union file systems, 126, 130
  versioning best practices, 137
importing, flat file systems, 135, 137
info subcommand, 58
InfoSec conversations, 114
inspect subcommand, 105
installing software
  from Dockerfile, 54

installation files and isolation, 54
container file system abstraction and isolation, 57
file system structure, 57, 58
image layers, 54, 56
layer relationships, 56, 57
loading images as files, 50, 51
searching Docker Hub for repositories, 54
using alternative registries, 50
integrity, distribution method and, 173
--interactive (-i) flag[interactive (-i) flag], 20
interactive containers, running, 20, 21
inter-container dependencies, 95
link nature and shortcomings, 95
interfaces, 78, 79
IP (Internet Protocol), 78
--ipc flag[ipc flag], 103
isolation, 118
containers with enhanced tools, 114
specifying additional security options, 115, 116
feature access, 113
resource allowances, 97
CPU, 99, 101
memory limits, 97, 99
running container with full privileges, 113, 114
shared memory
sharing IPC primitives between containers, 104
use-case-appropriate containers, 116, 118
applications, 116, 117
high-level system services, 117
low-level system services, 117, 118
users, 104
run-as user, 105

K

kill program, 41

L

--label flag[label flag], 147
LABEL instruction, 147, 148
LAMP (Linux, Apache, MySQL PHP) stack, 40
latest tag, 132, 138
layer relationships, 56, 57

layers, 46, 133, 135
layers, 133
links
  nature of, 95
  shortcomings of, 95
Linux Security Modules. See LSM, 115
Linux USR namespace, 166
longevity, distribution method and, 173
loopback interface, 78
low-level system services, 117, 118
LSM (Linux Security Modules), 115

M

-m (--memory) flag[m (memory) flag], 97, 124
mailer-base image, 146
MAINTAINER instruction, 148
Memcached technology, 79
memory
  limits on, 97, 99
  shared
    sharing IPC primitives between containers, 104
metaconflicts, eliminating, 26, 32
  container state and dependencies, 29, 32
  flexible container identification, 26, 29
metadata, Dockerfile and, 145, 149
mmap() function, 58
mod_ubuntu container, 127, 131, 132
multiprocess containers, 159, 164

N

--name flag[name flag], 27
NAT, 79, 81
--net flag[net flag], 103
networks, 95
  bridged containers
    custom name resolution, 90, 94
  container networking
    local Docker network topology, 82
  inter-container dependencies, 95
    link nature and shortcomings, 95
  interfaces, 78, 79
  NAT, 79, 81

overview, 77, 81
port forwarding, 79, 81
ports, 78, 79
protocols, 78, 79
--no-cache flag[no-cache flag], 145
NoSQL database, using volumes with, 67, 69

O

ONBUILD instruction, 153, 154, 155
--output (-o)[output (-o)], 135

P

--password flag[password flag], 175
permissions, 167, 169
permissions, 165, 167
polymorphic tools, 66
ports, forwarding, 79, 81
ports, forwarding, 78, 79
private hosted repositories, 177, 179
private registries, 179, 184
consuming images from, 183, 184
using registry image, 182, 183
--privileged flag[privileged flag], 113
problems solved by Docker
getting organized, 10
protecting computer, 13
protocols, 78, 79
ps command, 24
public and private software distribution, 170
public repositories, 175, 177

Q

--quiet (-q) flag[quiet (-q) flag], 144

R

read-only file systems, 32, 35
registries
alternative, 50
hosted, 174, 179
private hosted repositories, 177, 179
public repositories, 175, 177

private, 179, 184
consuming images from, 183, 184
using registry image, 182, 183
repositories, 46, 47, 133
private hosted, 177, 179
public, 175, 177
searching Docker Hub for, 54
resource allowances, 97
CPU, 99, 101
memory limits, 97, 99
--restart flag[restart flag], 39, 99
restarting containers, 39, 40
rm command, 104
--rm flag[rm flag], 68
rsync tool, 190
RUN instruction, 144, 150, 152, 167
run-as user, 105

S

-s option[s option], 58
security, InfoSec conversations, 13, 114
--security-opt flag[security-opt flag], 115
SELinux, 115
SGID permission set, 167, 169
shared memory
sharing IPC primitives between containers, 104
sharing volumes
generalized sharing and volumes-from flag, 71
SIG_HUP signal, 43
SIG_KILL signal, 43
Simple Email Service example, 151, 152
solutions provided by Docker
getting organized, 10
protecting computer, 13
sshd tool, 117
startup process, 40, 42
startup scripts, 159, 164
environmental preconditions validation, 159, 161
initialization processes, 161, 164
--storage-driver option[storage-driver option], 58
SUID permission set, 167, 169
supervisor process, 40, 42

supervisord program, 40, 41
syslogd tool, 117

T

--tag (-t) flag[tag (-t) flag], 143
tags, 47, 133
TCP (Transmission Control Protocol), 79
Transmission Control Protocol See TCP, 79
transportation, distribution method and, 172
–tty (–t) flag, 20

U

UFS (union file system), 120
use-case-appropriate containers, 116, 118
  applications, 116, 117
  high-level system services, 117
  low-level system services, 117, 118
--user (-u) flag[user (-u) flag], 107
USER instruction, 149, 165, 167
user permissions, 165, 167
--username flag[username flag], 175
users, 104
  run-as user, 105

V

-v (--volume) option[v (volume) option], 63
-v flag[v flag], 43
v option, 104
VERSION variable, 147
versioning, best practices, 137
visibility, distribution method and, 172
--volume flag[volume flag], 150
VOLUME instruction, 149, 150
volumes
  container-independent data management using, 66, 67
  managed volume life cycle, 73
  overview, 69
  sharing
    generalized sharing and volumes-from flag, 71
  types of, 66
    bind mount volumes, 62

Docker managed volumes, 65, 66
using with NoSQL database, 67, 69
--volumes-from flag[volumes-from flag], 73

W

WEB_HOST environment variable, 160
wget program, 20
wheezy tag, 139
whoami command, 105, 168
WORDPRESS_DB_HOST variable, 36
WORKDIR instruction, 148, 149