

Docker

IN ACTION

SECOND EDITION

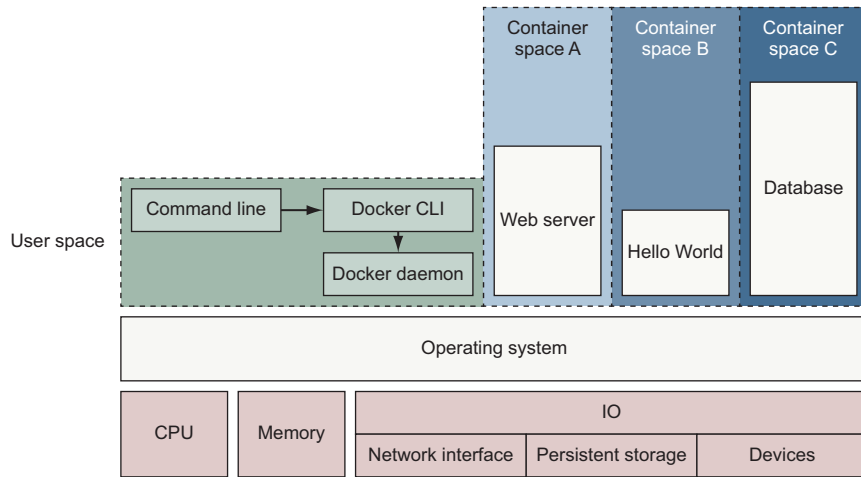
Jeff Nickoloff
Stephen Kuenzli

FOREWORD BY Bret Fisher



MANNING

Docker running three containers on a Linux system



Praise for the first edition

“All there is to know about Docker. Clear, complete, and precise.”

—Jean-Pol Landrain, Agile Partner Luxembourg

“A compelling narrative for real-world Docker solutions. A must-read!”

—John Guthrie, Pivotal, Inc.

“An indispensable guide to understanding Docker and how it fits into your infrastructure.”

—Jeremy Gailor, Gracernote

“Will help you transition quickly to effective Docker use in complex real-world situations.”

—Peter Sellars, Fraedom

“. . . a superlative introduction to, and reference for, the Docker ecosystem.”

—Amazon reader

Docker in Action

SECOND EDITION

JEFF NICKOLOFF
STEPHEN KUENZLI
FOREWORD BY BRET FISHER



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact


Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2019 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Jennifer Stout
Technical development editor: Raphael Villela
Review editor: Aleksandar Dragosavljević
Project editor: Janet Vail
Copy editor: Sharon Wilkey
Proofreader: Keri Hales
Technical proofreader: Niek Palm
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617294761

Printed in the United States of America

For Jarrod Nikoloff and William Kuenzli

contents

<i>foreword</i>	<i>xiii</i>
<i>preface</i>	<i>xv</i>
<i>acknowledgments</i>	<i>xviii</i>
<i>about this book</i>	<i>xx</i>
<i>about the authors</i>	<i>xxii</i>
<i>about the cover illustration</i>	<i>xxiii</i>

1 *Welcome to Docker* 1

- 1.1 What is Docker? 3
 - “Hello, World”* 3
 - *Containers* 5
 - *Containers are not virtualization* 5
 - *Running software in containers for isolation* 6
 - *Shipping containers* 7
- 1.2 What problems does Docker solve? 8
 - Getting organized* 9
 - *Improving portability* 10
 - Protecting your computer* 11
- 1.3 Why is Docker important? 12
- 1.4 Where and when to use Docker 13
- 1.5 Docker in the larger ecosystem 14
- 1.6 Getting help with the Docker command line 14

PART 1 PROCESS ISOLATION AND ENVIRONMENT-INDEPENDENT COMPUTING17

- ## 2 *Running software in containers* 19
- 2.1 Controlling containers: Building a website monitor 20
 - Creating and starting a new container* 21
 - *Running interactive containers* 22
 - *Listing, stopping, restarting, and viewing output of containers* 23
 - 2.2 Solved problems and the PID namespace 25
 - 2.3 Eliminating metaconflicts: Building a website farm 28
 - Flexible container identification* 28
 - *Container state and dependencies* 31
 - 2.4 Building environment-agnostic systems 34
 - Read-only filesystems* 34
 - *Environment variable injection* 37
 - 2.5 Building durable containers 40
 - Automatically restarting containers* 41
 - *Using PID 1 and init systems* 42
 - 2.6 Cleaning up 44
- ## 3 *Software installation simplified* 47
- 3.1 Identifying software 48
 - What is a named repository?* 48
 - *Using tags* 49
 - 3.2 Finding and installing software 50
 - Working with Docker registries from the command line* 50
 - Using alternative registries* 51
 - *Working with images as files* 52
 - Installing from a Dockerfile* 53
 - *Using Docker Hub from the website* 54
 - 3.3 Installation files and isolation 56
 - Image layers in action* 57
 - *Layer relationships* 58
 - Container filesystem abstraction and isolation* 59
 - Benefits of this toolset and filesystem structure* 60
 - Weaknesses of union filesystems* 60
- ## 4 *Working with storage and volumes* 62
- 4.1 File trees and mount points 63
 - 4.2 Bind mounts 64
 - 4.3 In-memory storage 67

- 4.4 Docker volumes 68
 - Volumes provide container-independent data management* 70
 - Using volumes with a NoSQL database* 71
- 4.5 Shared mount points and sharing files 73
 - Anonymous volumes and the volumes-from flag* 74
- 4.6 Cleaning up volumes 77
- 4.7 Advanced storage with volume plugins 78

5 *Single-host networking* 80

- 5.1 Networking background (for beginners) 81
 - Basics: Protocols, interfaces, and ports* 81
 - Bigger picture: Networks, NAT, and port forwarding* 82
- 5.2 Docker container networking 83
 - Creating a user-defined bridge network* 84
 - Exploring a bridge network* 86
 - Beyond bridge networks* 88
- 5.3 Special container networks: host and none 89
- 5.4 Handling inbound traffic with NodePort publishing 91
- 5.5 Container networking caveats and customizations 93
 - No firewalls or network policies* 93
 - Custom DNS configuration* 93
 - Externalizing network management* 97

6 *Limiting risk with resource controls* 99

- 6.1 Setting resource allowances 100
 - Memory limits* 101
 - CPU* 102
 - Access to devices* 105
- 6.2 Sharing memory 105
 - Sharing IPC primitives between containers* 106
- 6.3 Understanding users 107
 - Working with the run-as user* 108
 - Users and volumes* 111
 - Introduction to the Linux user namespace and UID remapping* 113
- 6.4 Adjusting OS feature access with capabilities 114
- 6.5 Running a container with full privileges 116
- 6.6 Strengthening containers with enhanced tools 117
 - Specifying additional security options* 118
- 6.7 Building use-case-appropriate containers 119
 - Applications* 119
 - High-level system services* 120
 - Low-level system services* 120

PART 2 PACKAGING SOFTWARE FOR DISTRIBUTION123

7 *Packaging software in images* 125

- 7.1 Building Docker images from a container 126
 - Packaging “Hello, World”* 126
 - *Preparing packaging for Git* 127
 - *Reviewing filesystem changes* 128
 - *Committing a new image* 129
 - *Configuring image attributes* 130
- 7.2 Going deep on Docker images and layers 131
 - Exploring union filesystems* 131
 - *Reintroducing images, layers, repositories, and tags* 134
 - *Managing image size and layer limits* 137
- 7.3 Exporting and importing flat filesystems 139
- 7.4 Versioning best practices 141

8 *Building images automatically with Dockerfiles* 144

- 8.1 Packaging Git with a Dockerfile 145
- 8.2 A Dockerfile primer 148
 - Metadata instructions* 149
 - *Filesystem instructions* 153
- 8.3 Injecting downstream build-time behavior 156
- 8.4 Creating maintainable Dockerfiles 159
- 8.5 Using startup scripts and multiprocess containers 162
 - Environmental preconditions validation* 163
 - *Initialization processes* 164
 - *The purpose and use of health checks* 166
- 8.6 Building hardened application images 167
 - Content-addressable image identifiers* 168
 - *User permissions* 169
 - *SUID and SGID permissions* 171

9 *Public and private software distribution* 174

- 9.1 Choosing a distribution method 175
 - A distribution spectrum* 175
 - *Selection criteria* 176
- 9.2 Publishing with hosted registries 178
 - Publishing with public repositories: “Hello World!” via Docker Hub* 179
 - *Private hosted repositories* 181
- 9.3 Introducing private registries 183
 - Using the registry image* 186
 - *Consuming images from your registry* 187
- 9.4 Manual image publishing and distribution 188
 - A sample distribution infrastructure using FTP* 190
- 9.5 Image source-distribution workflows 194
 - Distributing a project with Dockerfile on GitHub* 194

10 *Image pipelines* 197

- 10.1 Goals of an image build pipeline 198
- 10.2 Patterns for building images 199
 - All-in-one images* 200
 - *Separate build and runtime images* 201
 - *Variations of runtime image via multi-stage builds* 202
- 10.3 Record metadata at image build time 204
 - Orchestrating the build with make* 205
- 10.4 Testing images in a build pipeline 209
- 10.5 Patterns for tagging images 212
 - Background* 212
 - *Continuous delivery with unique tags* 213
 - Configuration image per deployment stage* 214
 - *Semantic versioning* 215

PART 3 HIGHER-LEVEL ABSTRACTIONS AND ORCHESTRATION.....217

11 *Services with Docker and Compose* 219

- 11.1 A service “Hello World!” 220
 - Automated resurrection and replication* 222
 - *Automated rollout* 224
 - *Service health and rollback* 226
- 11.2 Declarative service environments with Compose V3 229
 - A YAML primer* 231
 - *Collections of services with Compose V3* 233
- 11.3 Stateful services and preserving data 237
- 11.4 Load balancing, service discovery, and networks with Compose 239

12 *First-class configuration abstractions* 244

- 12.1 Configuration distribution and management 245
- 12.2 Separating application and configuration 247
 - Working with the config resource* 249
 - *Deploying the application* 250
 - *Managing config resources directly* 251
- 12.3 Secrets—A special kind of configuration 255
 - Using Docker secrets* 257

13	<i>Orchestrating services on a cluster of Docker hosts with Swarm</i>	264
13.1	Clustering with Docker Swarm	264
	<i>Introducing Docker Swarm mode</i>	265
	<i>Deploying a Swarm cluster</i>	267
13.2	Deploying an application to a Swarm cluster	267
	<i>Introducing Docker Swarm cluster resource types</i>	267
	<i>Defining an application and its dependencies by using Docker services</i>	268
	<i>Deploying the application</i>	273
13.3	Communicating with services running on a Swarm cluster	278
	<i>Routing client requests to services by using the Swarm routing mesh</i>	278
	<i>Working with overlay networks</i>	281
	<i>Discovering services on an overlay network</i>	282
	<i>Isolating service-to-service communication with overlay networks</i>	284
	<i>Load balancing</i>	286
13.4	Placing service tasks on the cluster	287
	<i>Replicating services</i>	288
	<i>Constraining where tasks run</i>	292
	<i>Using global services for one task per node</i>	297
	<i>Deploying real applications onto real clusters</i>	299
	<i>index</i>	301

foreword

Welcome to the container revolution. By reading this book, you're opening your eyes to a new world of tools that are forever changing the way we build, deploy, and run software. Once I discovered Docker in 2014 (the year after it was open-sourced) I did something I had never done in my 20+ year career: I decided to focus exclusively on this single technology. That's how much I believed in what Docker was doing to make our ever-increasing IT world easier to manage.

Fast forward to today, and what's still unique about Docker's way of creating and deploying containers is that it has both developers and operators in mind. You can see this in the user-experience of its command-line tools, and with hundreds of tools in the container ecosystem, I keep coming back to Docker as the easiest and smoothest way to get things done.

Jeff and Stephen know this too about Docker's streamlined approach to containers, which is why this book focuses on the details of the core tools. Docker Engine, Docker Compose, and Docker Swarm are key tools we should all know. They often solve your problems without the need for more complex solutions. This same methodology is how I teach my students and how I guide my clients.

Containers couldn't have come at a better time, taking features of the Linux kernel (and now Windows, ARM, and more) and automating them into accessible one-line commands. Sure, we had container-like features for years in Solaris, FreeBSD, and then Linux, but it was only the bravest sysadmins who got those features to work before Docker.

Containers today are now more than the sum of their parts. The workflow speed and agility that a fully Dockerized software lifecycle gives a team cannot be understated. I'm glad Jeff and Stephen took their battle-hardened experience and updated this already great book with new details and examples, and I'm confident you'll gain benefits by putting their recommendations into practice.

—BRET FISHER, DOCKER CAPTAIN AND CONTAINER CONSULTANT

bretfisher.com

twitter.com/bretfisher

preface

Docker and the container community have come a long way since we started participating in 2013. And Docker has changed in some unexpected ways since 2016, when Jeff released the first edition of this book. Thankfully, most of the user-facing interfaces and core concepts were maintained in a backward-compatible manner. The first two-thirds of the book needed updates only for additional features or closed issues. As anticipated, part 3 of the previous edition needed a full rewrite. Since publication of the previous book, we've seen progress in orchestration, app connectivity, proprietary cloud container offerings, multicontainer app packaging, and function-as-a-service platforms. This edition focuses on the fundamental concepts and practices for using Docker containers and steers clear of rapidly changing technologies that complement Docker.

The biggest change is the development and adoption of several container orchestrators. The primary purpose of a container orchestrator is to run applications modeled as services across a cluster of hosts. Kubernetes, the most famous of these orchestrators, has seen significant adoption and gained support from every major technology vendor. The Cloud Native Computing Foundation was formed around that project, and if you ask them, a “cloud native” app is one designed for deployment on Kubernetes. But it is important not to get too caught up in the marketing or the specific orchestration technology. This book does not cover Kubernetes for two reasons.

While Kubernetes is included with Docker for Desktop, it is massive and in constant flux. It could never be covered at any depth in a handful of chapters or even in a book with fewer than 400 pages. A wealth of excellent resources are available

online as well as wonderful published books on Kubernetes. We wanted to focus on the big idea—service orchestration—in this book without getting too lost in the nuances.

Second, Docker ships with Swarm clustering and orchestration included. That system is more than adequate for smaller clusters, or clusters in edge computing environments. A huge number of organizations are happily using Swarm every day. Swarm is great for people getting started with orchestration and containers at the same time. Most of the tooling and ideas carry over from containers to services with ease. Application developers will likely benefit the most from this approach. System administrators or cluster operations personnel might be disappointed, or might find that Swarm meets their needs. But, we're not sure they'll ever find a long-form written resource that will satisfy their needs.

The next biggest change is that Docker runs everywhere today. Docker for Desktop is well integrated for use on Apple and Microsoft operating systems. It hides the underlying virtual machine from users. For the most part, this is a success; on macOS, the experience is nearly seamless. On Windows, things seem to go well at least for a few moments. Windows users will deal with an intimidating number of configuration variations from corporate firewalls, aggressive antivirus configuration, shell preferences, and several layers of indirection. That variation makes delivering written content for Windows impossible. Any attempt to do so would age out before the material went to production. For that reason, we've again limited the included syntax and system-specific material to Linux and macOS. A reader just might find that all the examples actually run in their environment, but we can't promise that they will or reasonably help guide troubleshooting efforts.

Next, getting an internet-attached virtual machine with Docker installed has become trivial. Every major and minor cloud provider offers as much. For that reason, we've removed material pertaining to Docker Machine and installing Docker. We're confident that our readers will be able to find installation instructions that are most appropriate for the platform of their choice. And today, they might even skip that step and adopt one of the many container-first cloud platforms like AWS ECS. This book won't cover those platforms. They're each unique enough to be difficult to discuss in aggregate. And all of them have put significant effort into their adoption stories and documentation.

Finally, containers and networking have had a complicated history. In the last few years, that story became just a little bit more complicated with the emergence of service mesh platforms and other complementary technologies. A service mesh is a platform of application-aware smart pipes that provide microservice networking best practices out of the box. They use proxies to provide point-to-point encryption, authentication, authorization, circuit-breakers, and advanced request routing. The container networking fundamentals presented in this book should prove useful in understanding and evaluating service mesh technologies.

This book is intended as a deep introduction to the fundamentals of working with Docker. A reader might not learn everything that they need in their daily application of this technology. But they will have the fundamental skillset required to learn advanced topics more quickly and further those pursuits. We wish you the best of luck in those containerized ventures.

acknowledgments

We would like to thank Manning Publications for the opportunity to write this book; the generous help from our editors, particularly Jennifer Stout; and feedback from all of our reviewers: Andy Wiesendanger, Borko Djurkovic, Carlos Curotto, Casey Burnett, Chris Phillips, Christian Kreutzer-Beck, Christopher Phillips, David Knepprath, Dennis Reil, Des Horsley, Ernesto Cárdenas Cangahuala, Ethan Rivett, Georgios Doumas, Gerd Klevesaat, Giuseppe Caruso, Kelly E. Hair, Paul Brown, Reka Horvath, Richard Lebel, Robert Koch, Tim Gallagher, Wendell Beckwith, and Yan Guo. You all helped make this a better book.

Jeff Nickoloff: A second edition is a burden and an opportunity. It is the same burden any SaaS owner feels. People are consuming your work, and, ultimately, you're in some small part responsible for their success or failure. I took on this work knowing that it needed to be done, but also that I would struggle without a coauthor. It is an opportunity to continue sharing what I know with the world, but more importantly an opportunity to introduce and share Stephen Kuenzli's knowledge. He and I have had several opportunities to work together in Phoenix, including co-organizing DevOps-Days, running the Docker PHX meetup, and bouncing a constant stream of ideas off each other.

Since 2013, I've watched and helped countless people and teams work through their container and cloud adoption stories. I learn something new from each encounter, and it is safe to say that I would not be where I am today if it were not for their willingness to include me.

A huge portion of the engineers who shaped my insight into Docker have since moved on to different companies, projects, and passions. I'm thankful for their continued insight into that new and diverse spectrum of challenges and technology.

Portia Dean has been an invaluable partner. Without her willingness to choose the challenging and rewarding paths, I wouldn't have these books, our companies, or the same degree of personal fulfillment. We can accomplish anything together.

Finally, I want to acknowledge my parents, Jeff and Kathy, for their early and ongoing support and encouragement.

Stephen Kuenzli: Writing a book is a great challenge and responsibility. I learned a large portion of my practical professional skills from technical books like this one after graduating with an engineering degree. That knowledge and those skills have been central to my career, and I appreciate that gift of knowledge. When Jeff asked me to help him update *Docker in Action*, I was excited and frightened. Here was an opportunity to expand and improve on a successful work by sharing knowledge I'd gained over the past several years building systems with Docker. My main motivation was to help people along their own development paths. I knew this would be challenging and require tremendous discipline. Indeed, authoring the second edition surpassed my imagined effort, and I am proud of what we have produced.

Every significant work requires assistance and support from people around the creators. I would like to thank the following people who made this book a success:

- My coauthor, Jeff Nickoloff, for the opportunity to collaborate on this work and learn how to write.
- My wife, Jen, for her patience and the quiet time required to actually write this book. Our son, William, for constantly reminding me of the joy in life and inspiring me to do my best.
- Docker, for building a great tool and community.
- Everyone who taught and gave me the opportunities needed to get to this point.

And thanks to all of you reading this book. I hope you find this book useful and that it helps you grow your career.

about this book

Docker in Action's purpose is to introduce developers, system administrators, and other computer users of a mixed skillset to the Docker project and Linux container concepts. Both Docker and Linux are open source projects with a wealth of online documentation, but getting started with either can be a daunting task.

Docker is one of the fastest-growing open source projects ever, and the ecosystem that has grown around it is evolving at a similar pace. For these reasons, this book focuses on the Docker toolset exclusively. This restriction of scope should both help the material age well and help readers understand how to apply Docker features to their specific use-cases. Readers will be prepared to tackle bigger problems and explore the ecosystem once they develop a solid grasp of the fundamentals covered in this book.

Roadmap

This book is split into three parts.

Part 1 introduces Docker and container features. Reading it will help you understand how to install and uninstall software distributed with Docker. You'll learn how to run, manage, and link different kinds of software in different container configurations. Part 1 covers the basic skillset that every Docker user will need.

Part 2 is focused on packaging and distributing software with Docker. It covers the underlying mechanics of Docker images, nuances in file sizes, and a survey of different packaging and distribution methods. This part wraps up with a deep dive into the Docker Distribution project.

Part 3 explores multicontainer projects and multihost environments. This includes coverage of the Docker Compose and Swarm projects. These chapters walk you through building and deploying multiple real world examples that should closely resemble large-scale server software you'd find in the wild.

Code conventions and downloads

This book is about a multipurpose tool, and so there is very little “code” included in the book. In its place are hundreds of shell commands and configuration files. These are typically provided in POSIX-compliant syntax. Notes for Windows users are provided where Docker exposes some Windows-specific features. Care was taken to break up commands into multiple lines in order to improve readability or clarify annotations. Referenced repositories are available on [manning.com](https://www.manning.com/books/docker-in-action-second-edition) at <https://www.manning.com/books/docker-in-action-second-edition> and also on Docker Hub (<https://hub.docker.com/u/dockerinaction/>) with sources hosted on GitHub (<https://github.com/dockerinaction>). No prior knowledge of Docker Hub or GitHub is required to run the examples.

This book uses several open source projects to both demonstrate various features of Docker and help the reader shift software-management paradigms. No single software “stack” or family is highlighted other than Docker itself. Working through the examples, the reader will use tools such as WordPress, Elasticsearch, Postgres, shell scripts, Netcat, Flask, JavaScript, NGINX, and Java. The sole commonality is a dependency on the Linux kernel.

liveBook discussion forum

Purchase of *Docker in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <https://livebook.manning.com/#!/book/docker-in-action-second-edition/discussion>. You can also learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/#!/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors

JEFF NICKOLOFF builds large-scale services, writes about technology, and helps people achieve their product goals. He has done these things at Amazon.com, Limelight Networks, and Arizona State University. After leaving Amazon in 2014, he founded a consulting company and focused on delivering tools, training, and best practices for Fortune 100 companies and startups alike. In 2019, he and Portia Dean founded Topple Inc., where they build productivity software as a service. Topple helps teams address the communication and coordination issues that slow them down, put their business at risk, and generally make work suck. If you'd like to chat or work together, you can find him at <http://allingeek.com>, or on Twitter as @allingeek.

STEPHEN KUENZLI has designed, built, deployed, and operated highly available, scalable software systems in high-tech manufacturing, banking, and e-commerce systems for nearly 20 years. Stephen has a BS in systems engineering and has learned, used, and built many software and infrastructure tools to deliver better systems. He loves working through challenging design problems and building solutions that are safe and enjoyable for customers, users, and stakeholders. Stephen founded and leads QualiMente, which helps businesses migrate and grow on AWS securely. If you would like help adopting secure, modern application delivery processes using technologies such as containers and infrastructure as code, reach out to him at www.qualimente.com.

about the cover illustration

The figure on the cover of *Docker in Action* is captioned “The Angler.” The illustration is taken from a nineteenth-century collection of works by many artists, edited by Louis Curmer and published in Paris in 1841. The title of the collection is *Les Français peints par eux-mêmes*, which translates as *The French People Painted by Themselves*. Each illustration is finely drawn and colored by hand, and the rich variety of drawings in the collection reminds us vividly of how culturally apart the world’s regions, towns, villages, and neighborhoods were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns or regions. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by pictures from collections such as this one.

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

A *best practice* is an optional investment in your product or system that should yield better outcomes in the future. Best practices enhance security, prevent conflicts, improve serviceability, or increase longevity. Best practices often need advocates because justifying the immediate cost can be difficult. 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, deploying software with Docker can help reduce customer impact. Containers come up more quickly and consume fewer resources than 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.

Whereas 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, makes 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 such as web servers, databases, and mail servers, and with terminal applications including text editors, compilers, network analysis tools, and scripts; in some cases, it’s even used to run GUI applications such as web browsers and productivity software.

Docker runs Linux software on most systems. Docker for Mac and Docker for Windows integrate with common virtual machine (VM) technology to create portability with Windows and macOS. But Docker 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 such as installing, removing, upgrading, distributing, trusting, and running software. It’s open source Linux software, which means that anyone can contribute to it and that 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 by 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.

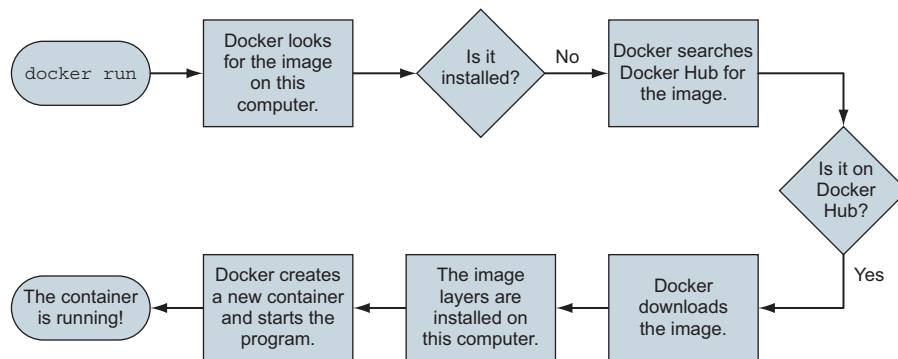


Figure 1.1 What happens after running `docker run`

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 details.

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 whether the `docker-inaction/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 to indicate where the image (`dockerinaction/hello_world`) can be found, and Docker starts the download.

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

```
echo "hello world"
```

After the `echo` command prints "hello world" to the terminal, the program exits, 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 whether `docker-inaction/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. We 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.

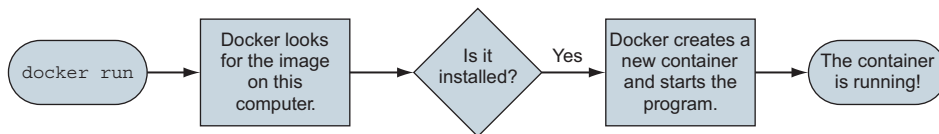


Figure 1.2 Running `docker run` a second time. Because the image is already installed, Docker can start the new container right away.

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 types of sticking points people encounter as they migrate to containers.

Although 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 filesystem 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 hardware 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 perform optimally once everything is up and running, but the startup delays make them 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 hardware virtualization technology. Instead, it helps you use the container technology already built into your operating system kernel.

Virtual machines provide hardware abstractions so 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 kernel. Docker for Mac and Windows users, and almost all cloud computing users, will run Docker inside 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.

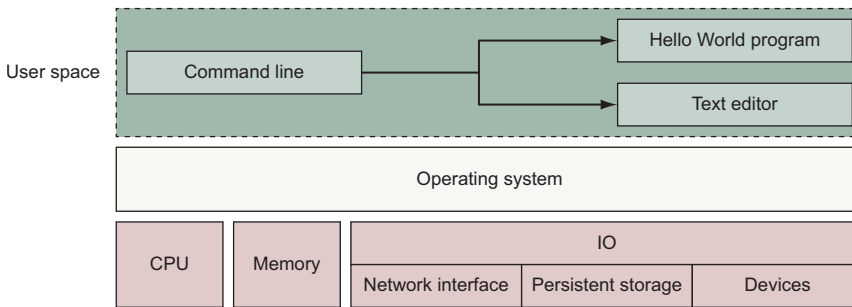


Figure 1.3 A basic computer stack running two programs that were started from the command line

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 by using the Docker program.

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 10 major system features. Part 1 of this book uses Docker commands to illustrate how these features can be modified to suit the needs

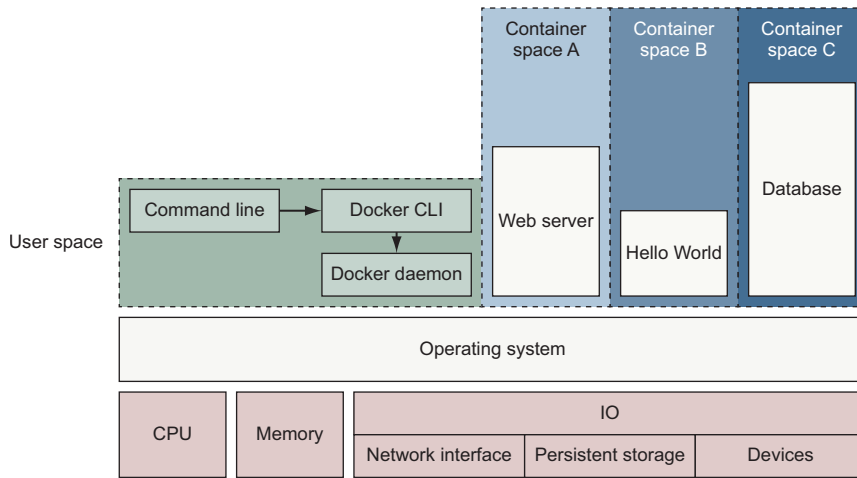


Figure 1.4 Docker running three containers on a basic Linux computer system

of the contained software and to fit the environment where the container will run. The specific features are as follows:

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

Docker uses those to build containers at runtime, but 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 filesystem. 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 the 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 such as APT, Homebrew, YUM, and npm 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 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 creates 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 the way 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 a scarce resource such as a network connection or a file.

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

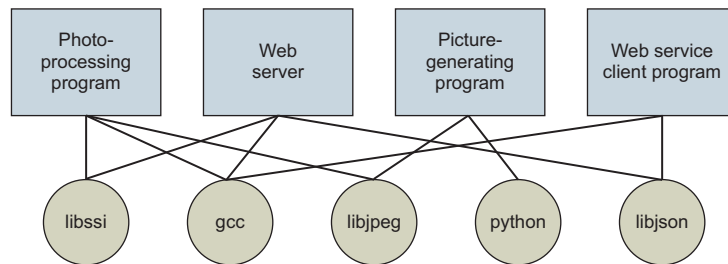


Figure 1.5 Dependency relationships of example programs

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 such as gcc. Chapter 3 describes how the Docker packaging system typically reduces the storage overhead.

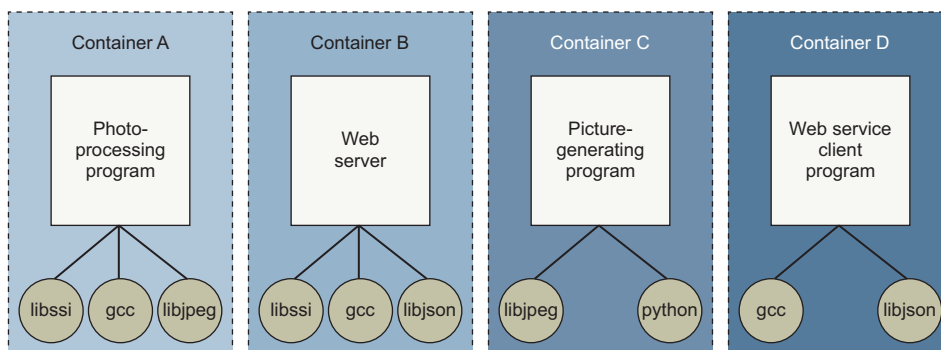


Figure 1.6 Example programs running inside containers with copies of their dependencies

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 macOS, using that same software on Windows can be more difficult. Doing so can require building whole ported versions of the software. Even that is possible only 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 macOS and Windows environments. This convergence on Linux means that software running in Docker containers need be written only 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 macOS 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 a 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 we want to use a popular web server that was not written in Java or another similarly portable language, we doubt that the authors would take time to rewrite it for us. 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 we'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 access only things that are inside it as well. Exceptions to this rule exist, 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.

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 in-depth defense strategy. It is far too commonly skipped or implemented in a half-hearted manner.

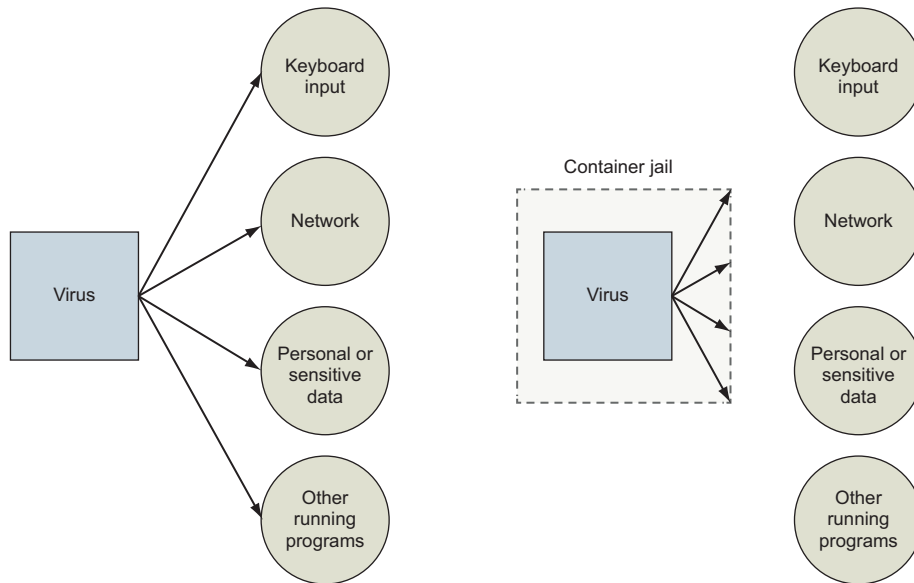


Figure 1.7 Left: A malicious program with direct access to sensitive resources. Right: A malicious program inside a container.

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 to 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 including 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 simple. Better yet, Docker does it in a cross-platform and open way. Imagine if all the major smartphones shared the same app store. That would be a pretty big deal. With this technology in place, it's possible 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 run only applications that can run on a Linux operating system, or Windows applications on Windows Server. If you want to run a macOS 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 such as web servers, mail servers, databases, proxies, and the like. Desktop software such as 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 that sometimes 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 co-located 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 a certain higher-level functionality and depend on Docker. Some tools use the Docker sub-components. Those subcomponents are independent projects such as 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 such as Docker, and so the containers and images you build on your laptop will run in Kubernetes.

You need to consider several trade-offs when picking up any tool. Kubernetes draws power from its extensibility, but that comes at the expense of its 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, we 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. We're 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.

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, macOS, 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.

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 expands into more detailed container configurations to demonstrate the full spectrum of capabilities. For that reason, we 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 we're confident that you'll have more than a few revelations along the way.

Running software in containers

This chapter covers

- Running interactive and daemon terminal programs in containers
- Basic Docker operations and commands
- Isolating programs from each other and injecting configuration
- Running multiple programs in a container
- 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 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'll 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 an outrageous request for you 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.

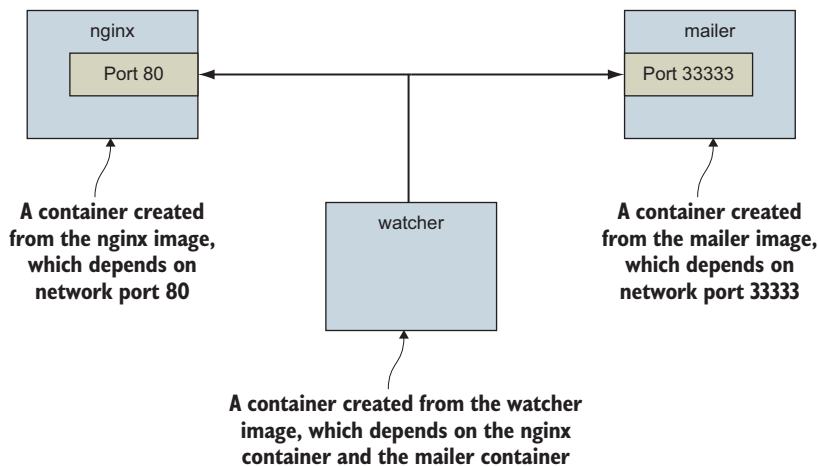


Figure 2.1 The three containers that you'll build in this example

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 named *watcher* will run as a monitoring agent 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 as the shipping containers used to transport physical goods around the world. Docker images hold everything a computer needs in order to run 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 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 \
  --name web nginx:latest
```

← 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 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 installs and run a mailer that will work for this example:

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

← Starts 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 a 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 \
  --link web:web \
  --name web_test \
  busybox:1.29 /bin/sh
```

← Creates a virtual terminal and binds 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 such as 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. 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 `Ctrl` (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 preceding example and send a message with the mailer if the web server stops. This command will start the agent in an interactive container by using the short-form flags:

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

← Creates a virtual terminal and binds 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 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 by 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-C` (or `Command-C`) 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
docker logs mailer
```

Stops the web server by stopping the container

Waits a couple of seconds and checks the mailer logs

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

```
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 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 such as `runit` or `supervisord`. In a different container, PID 1 might refer to a command shell such as `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 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 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, you can clearly 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 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 a 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 ← | Should list all processes
                                     running on the computer
```

Because 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 runs 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
docker exec webConflict nginx -g 'daemon off;'

```

← Output should be empty.

← Starts a second NGINX process in the same container

The last command should display output like this:

```

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. Called a *port conflict*, this is a common issue in real-world systems in which 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
docker logs webA

```

← Starts the first NGINX instance

← Verifies that it is working; should be empty.

```

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

```

← Starts the second instance

← Verifies 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 a 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 are 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. As in the earlier port conflict example, Docker solves software conflicts with such tools as Linux namespaces, resource limits, filesystem 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 preceding 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: a client has asked you to build a system on which 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. 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 watcher for each web server. The system will look like the architecture described in figure 2.2.

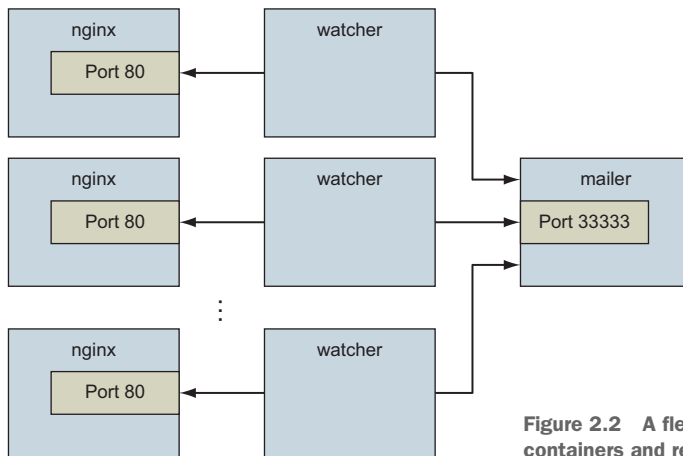


Figure 2.2 A fleet of web server containers and related monitoring agents

Your first instinct might be to simply start more web containers. But 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 previous example is a bad idea is to try it for yourself:

```
docker run -d --name webid nginx ← Creates a container named "webid"
docker run -d --name webid nginx ← Creates 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 such as `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 overrides that process with a known value. If a situation arises in which the name of a container needs to change, you can always rename the container with the `docker rename` command:

```
docker rename webid webid-old
docker run -d --name webid nginx
```

Renames the current web container to "webid-old"

Creates 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 \
  echo hello

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. 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 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 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 this:

```
b26a631e536d3caae348e9fd36e7661254a11511eb2274fb55f9f7c788721b0d
```

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

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

← 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 in which the script was launched. Those conflicts should be easily avoidable because one user 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
cat /tmp/web.cid
```

← Inspects the file

← Creates a new
stopped container

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 by 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 such as `/containers/web/customer1/web.cid` or `/containers/web/customer8/web.cid`.

In other cases, you can use other commands such as `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)
echo $CID
```

← This will work on POSIX-compliant shells.

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

← 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 look something like this:

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

← Make sure mailer from first example is running.

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

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 be in one of the states shown in figure 2.3. The Docker container management commands to move between states annotate each transition.

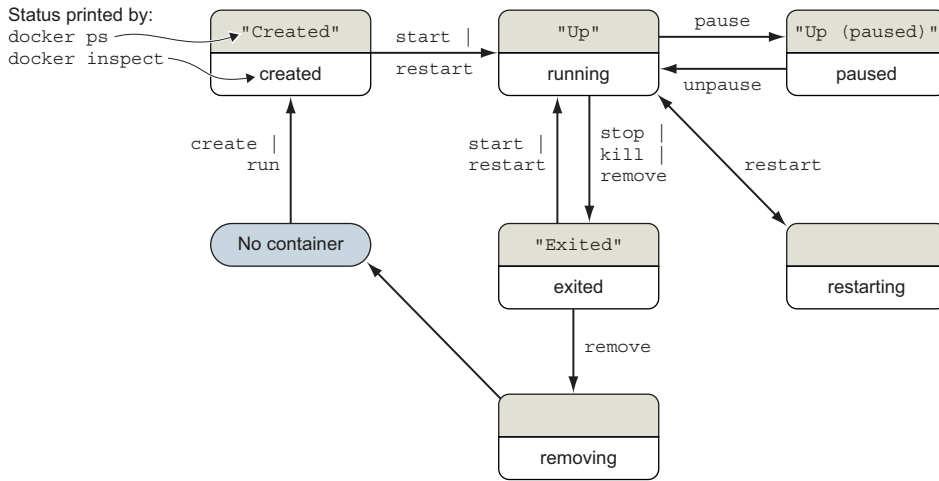


Figure 2.3 The state transition diagram for Docker containers

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 created state). To see all the containers (including those in the created state), use the `-a` option:

```
docker ps -a
```

The status of the new containers should be "Created". The `docker ps` command displays the container state using the “friendly” names shown in gray in figure 2.3. The `docker inspect` command uses the names shown in the bottom half of each state (for example, `created`). The `restarting`, `removing`, and `dead` (not illustrated) states are internal to Docker and are used to track transitions between the states visible in `docker ps`.

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. In chapter 5, you'll learn to connect containers with user-defined bridge networks to avoid this particular dependency problem. The key point here is that Docker will try to resolve a container's dependencies before creating or starting a container to avoid application runtime failures.

The legacy of container network linking

You may notice that the Docker documentation describes network links as a legacy feature. Network links were an early and popular way to connect containers. Links create a *unidirectional* network connection from one container to other containers on the same host. Significant portions of the container ecosystem asked for fully peered, *bidirectional* connections between containers. Docker provides this with the user-defined networks described in chapter 5. These networks can also extend across a cluster of hosts as described in chapter 13. Network links and user-defined networks are not equivalent, but Docker recommends migrating to user-defined networks.

It is uncertain whether the container network linking feature will ever be removed. Numerous useful tools and unidirectional communication patterns depend on linking, as illustrated by the containers used to inspect and watch the web and mailer components in this section.

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 this:

```
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 returned 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`. 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—for example, 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 (for example, known host filesystem locations), hardcoded 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 filesystems
- Environment variable injection
- Volumes

Working with volumes is a big subject and the topic of chapter 4. To learn the first two features, consider a requirement 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 filesystem to ensure data is written only to the database.

2.4.1 **Read-only filesystems**

Using read-only filesystems 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 by 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 this:

```
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)
```

When running WordPress with a read-only filesystem, the Apache web server 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 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` by 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 \
  -v /run/apache2/ \
  --tmpfs /tmp \
  wordpress:5.0.0-php7.2-apache
```

Makes container's root filesystem read-only

Mounts a writable directory from the host

Provides container an in-memory temp filesystem

That command should log successful messages that look like this:

```
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 by 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:

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

Uses a unique name

Creates a link to the database

Directs 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 filesystem 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 this design has two problems. 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 such as 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 hostname 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" \
  busybox:1.29 \
  env
```

← Injects an environment variable

← Executes 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, 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 username 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 when 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 \ ← For client A
  wordpress:5.0.0-php7.2-apache

docker create --link wpdb:mysql \
  -e WORDPRESS_DB_NAME=client_b_wp \ ← For client B
  wordpress:5.0.0-php7.2-apache
```

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
  echo "Client ID not set"
  exit 1
fi

WP_CID=$(docker create \
  --link $DB_CID:mysql \
  --name wp_$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 \
```

← Assumes `$CLIENT_ID` variable is set as input to script

← Creates link using `DB_CID`

```
--link $MAILER_CID:insidemailer \
dockerinaction/ch2_agent)
```

```
docker start $AGENT_CID
```

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

```
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.

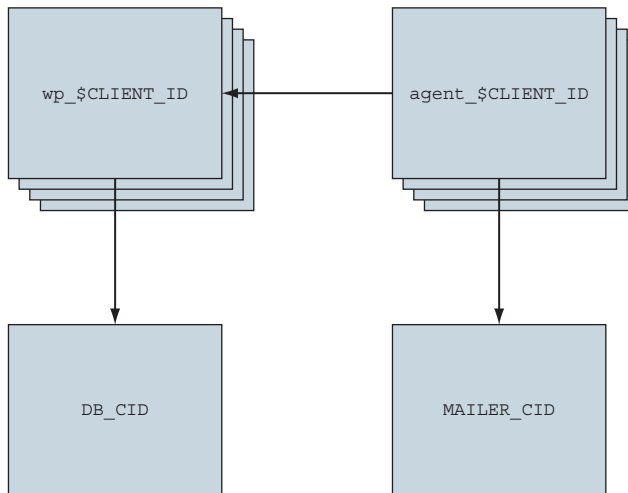


Figure 2.4 Each WordPress and agent container uses the same database and mailer.

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*

Software may fail 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 six states:

- Created
- Running
- Restarting
- Paused
- Removing
- 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 the amount of time that should pass between successive restart attempts. An exponential back-off 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:

```
Container <ID> is restarting, wait until the container is running
```

That means you can't do anything that requires the container to be in a running state, such as 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 *Using 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.

Several such init systems could 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 which 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 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 list `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 with one to view the contents of the startup script:

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

Uses "cat" as the entrypoint

Passes 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 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 filesystem. All containers also consume resources in the global namespace such as 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 this:

```
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 by 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 by 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 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, understanding how images relate to containers, and working with container filesystems.

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 six distinct states: created, running, restarting, paused, removing, 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 filesystem 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.

Software installation simplified

This chapter covers

- Identifying software
- Finding and installing software with Docker Hub
- Installing software from alternative sources
- Understanding filesystem isolation
- Working with images and layers

Chapters 1 and 2 introduced new concepts and abstractions provided by Docker. This chapter dives deeper into container filesystems and software installation. It breaks software installation into three steps, as illustrated in figure 3.1.

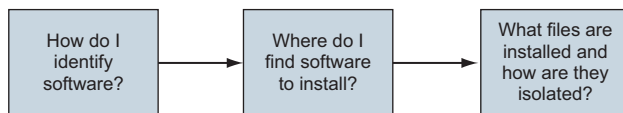


Figure 3.1 Flow of topics covered in this chapter

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. We've already mentioned that repositories hold images, but this chapter shows how repositories and tags are used to identify images in order to install the software you want.

This chapter details the three main ways to install Docker images:

- Using 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, an important concept when dealing with images, provide multiple important features. This chapter closes with a section about how images work. That knowledge will help you evaluate 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 want 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.

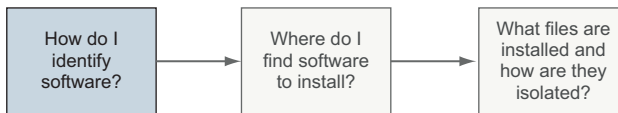


Figure 3.2 Step 1—software identification

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, as shown in figure 3.3. For example, later in this chapter you will install an image from the repository named `docker.io/dockerinaction/ch3_hello_registry`.

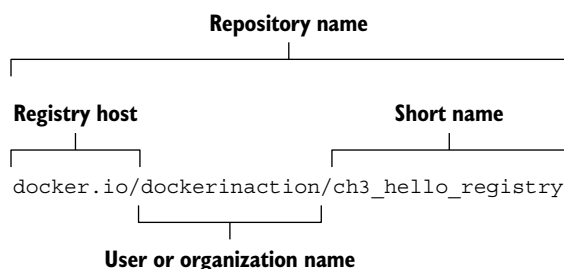


Figure 3.3 The Docker image repository name

Just as there can be several versions of software, a repository can hold several images. Each image in a repository is identified uniquely with tags. If you were to release a new version of `docker.io/dockerinaction/ch3_hello_registry`, you 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 nonunique 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 be applied to only 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: `11-stretch`, `11-jdk-stretch`, `11.0-stretch`, `11.0-jdk-stretch`, `11.0.4-stretch`, and `11.0.4-jdk-stretch`. All these tags are applied to the same image. This image is built by installing the current Java 11 Development Kit (JDK) into a Debian Stretch base image. As the current patch version of Java 11 increases, and the maintainers release Java 11.0.5, the `11.0.4` tag will be replaced in this set with `11.0.5`. If you care about which minor or patch version of Java 11 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 11, use the image tagged with `11-stretch`. It should always be assigned to the newest release of Java 11. These tags give users great flexibility.

It’s also common to see different tags for images with different software configurations. For example, we’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

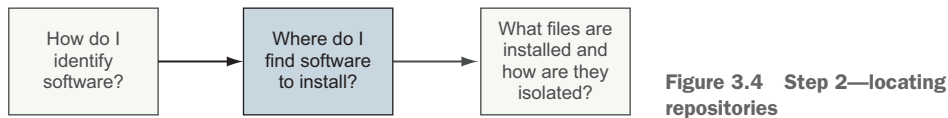
has a distinct tag that allows the user to easily identify the image with the features required.

TIP When you're looking for software to install, always pay careful attention to the tags offered in a repository. Many repositories publish multiple releases of their software, sometimes on multiple operating systems or in *full* or *slim* versions to support different use cases. Consult the repository's documentation for specifics of what the repository's tags mean and the image release practices.

That 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 3.4.



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 and is located at the host `docker.io`. 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, we 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*

An image author can publish images to a registry such as Docker Hub in two ways:

- *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 authorization checks on the repository they are updating. In these cases, you can use the `docker login` command to log in to specific registry servers such as Docker Hub. Once you've logged in, you'll be able to pull from private repositories, tag images in your repositories, and push to any repository that you control. Chapter 7 covers tagging and pushing 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 including AWS and Google offer private registries, and companies that use Docker EE or that 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 of example from an alternative registry:

```
docker pull quay.io/dockerinaction/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 quay.io/dockerinaction/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 Working with 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 we can show you the `docker load` command. Because it's unlikely that you have an image file lying around, we'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.

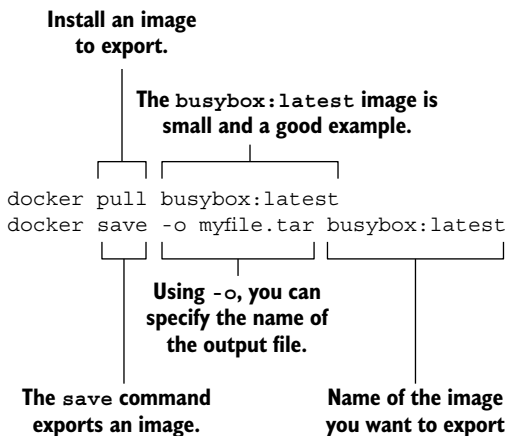


Figure 3.5 Parts of the `pull` and `save` subcommands

We 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 by using the docker load command. As with 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 by 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 by 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.

This approach has two disadvantages. 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 Using Docker Hub from the website

If you have yet to stumble upon it while browsing the Docker website, 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.

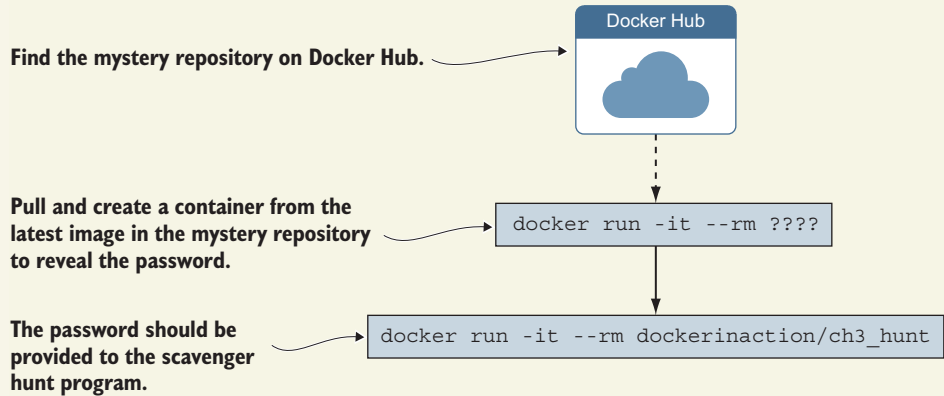
Activity: A Docker Hub scavenger hunt

It's good to practice finding software on Docker Hub by 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'll 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 find the password only by finding and running a container from the second mystery repository on Docker Hub. To use the programs in these images, you 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. The following diagram illustrates what you need to do from this point.



Still stuck? We 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 by using the `docker rmi` command. 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 learned a lot in this example. You 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.

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. You can install software in three other ways:

- 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 by using a provided Dockerfile.

All three options are viable for private projects or corporate infrastructure. The next few subsections cover how to install software from each alternative source. Chapter 9 covers Docker image distribution in detail. After reading this section, you should have a complete picture of your options to install software with Docker. When you install software, you should have an idea about what is in the software package and the changes 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 filesystem?

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

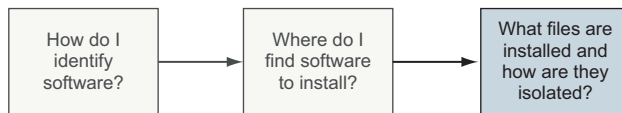


Figure 3.6 Step 3—understanding how software is installed

So far, when we've written about installing software, we'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 we've been calling an image is actually a collection of image layers.

A *layer* is set of files and file metadata that is packaged and distributed as an atomic unit. Internally, Docker treats each layer like an image, and layers are often called *intermediate images*. You can even promote a layer to an image by tagging it. Most layers build upon a parent layer by applying filesystem changes to the parent. For example, a layer might update the software in an image with a package manager by using, for example, Debian's `apt-get update`. The resulting image contains the combined set of files from the parent and the layer that was added. It is easier to understand layers when you see them in action.

3.3.1 Image layers in action

In this example, you'll install the two images. Both depend on Java 11. The applications themselves are simple “Hello, World”-style programs. We want you to keep an eye on 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 the `docker pull` command prints 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 need to only 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 OpenJDK 11.0.4 (slim) as a dependency of some other image, the download of `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:11.0.4-jdk-slim` 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:11.0.4-jdk-slim` was installed, and then the tiny `ch3_myapp` layer was installed.

When you issued the command to install `ch3_myotherapp`, Docker identified that `openjdk:11.0.4-jdk-slim` was already installed and immediately installed the image for `ch3_myotherapp`. Since the second application shared almost all of its image layers with the first application, Docker had much less to do. Installing the unique layer of `ch3_myotherapp` was very fast because less than one megabyte of data was transferred. 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. Chapter 7 covers this in more detail.

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

- `dockerinaction/ch3_myapp`
- `dockerinaction/ch3_myotherapp`

By default, the `docker images` command will show only repositories. As with 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 and may also include some listed as `<none>`. Unnamed images can exist for several reasons, such as building an image without tagging it. The only way to refer to these is to use the value in the `IMAGE ID` column.

In this example, you installed two images. Let's clean them up now. You can do so more easily if you use the condensed `docker rmi` syntax:

```
docker rmi \
  dockerinaction/ch3_myapp \
  dockerinaction/ch3_myotherapp
```

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. We'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 layers in the lineage of the image that the container was created from. Images can have relationships with any other image, including images in different repositories with different owners. The two application images in section 3.3.1 use an OpenJDK 11.0.4 image as their parent. The OpenJDK image's parent is a minimal version of the Debian Linux Buster operating system release. Figure 3.7 illustrates the full image ancestry of both images and the layers contained in each image.

The images and layers in figure 3.7 show that the application images will inherit three layers from `openjdk:11.0.4-jdk-slim` and one more layer from `debian:buster-slim`. The three layers from OpenJDK contain common libraries and dependencies of the Java 11 software, and the Debian image contributes a minimal operating system toolchain.

An image is named when its author tags and publishes it. A user can create aliases, as you did in chapter 2 by using the `docker tag` command. Until an image is tagged, the only way to refer to it is to use its unique identifier (ID) that was generated when the image was built. In figure 3.7, the parent image of the OpenJDK 11.0.4 image is the Debian Buster OS whose ID is `83ed3c583403`. The Debian image authors tagged and published this image as `debian:buster-slim`. The figure labels these images with the first 12 digits of their image ID, a common convention. Docker truncates the ID from 65 (base 16) digits to 12 in output of common commands for the benefit of its human users. Internally and through API access, Docker uses the full 65.

Even these “slim” Java images are sizable and were chosen to illustrate a point. At the time of this writing, the `openjdk:11.0.4-jdk-slim` image is 401 MB. You get some space savings when you use the runtime-only images, but even `openjdk:11.0.4-jre-slim-buster` is 204 MB. Because Docker uniquely identifies images and layers, it is able to recognize shared image dependencies between applications and avoid

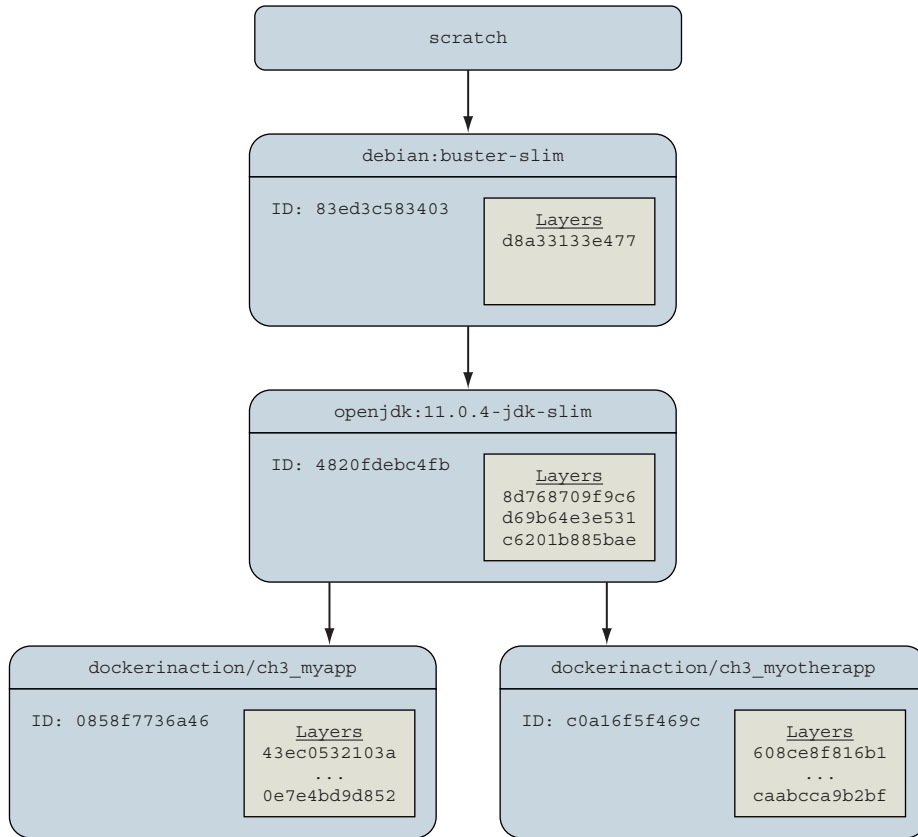


Figure 3.7 The full lineage of the two Docker images used in section 3.3.1

downloading those dependencies again. This is done without requiring any coordination between applications at runtime, only build time. Chapter 10 discusses image build pipelines in depth. Let’s continue by examining container filesystems.

3.3.3 Container filesystem abstraction and isolation

Programs running inside containers know nothing about image layers. From inside a container, the filesystem 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 filesystem (UFS)*.

Docker uses a variety of union filesystems and will select the best fit for your system. The details of how the union filesystem works are beyond what you need to know to use Docker effectively. A union filesystem is part of a critical set of tools that combine to create effective filesystem isolation. The other tools are MNT namespaces and the `chroot` system call.

The filesystem is used to create mount points on your host's filesystem that abstract the use of layers. The layers created 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 filesystem 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 filesystem the root in the container's context. This prevents anything running inside the container from referencing any other part of the host filesystem.

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

3.3.4 Benefits of this toolset and filesystem 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 a 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 a 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 filesystems

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

Different filesystems have different rules about file attributes, sizes, names, and characters. Union filesystems are in a position where they often need to translate between the rules of different filesystems. 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 filesystems use a pattern called *copy-on-write*, and that makes implementing memory-mapped files (the `mmap` system call) difficult. Some union filesystems 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 filesystem is another pluggable feature of Docker. You can determine which filesystem your installation is using with the `info` subcommand. If you want to specifically tell Docker which filesystem 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 filesystem can be addressed without changing the storage provider. These can be solved with volumes, the subject of chapter 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 users of Docker 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 and image distribution servers.

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
- Using temporary, in-memory filesystems
- 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 filesystem 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 filesystem and mount points.

4.1 File trees and mount points

Unlike other operating systems, Linux unifies all storage into a single tree. Storage devices such as disk partitions or USB disk partitions are attached to specific locations in that tree. Those locations are called *mount points*. A mount point defines the location in the tree, the access properties to the data at that point (for example, writability), and the source of the data mounted at that point (for example, a specific hard disk, USB device, or memory-backed virtual disk). Figure 4.1 depicts a filesystem constructed from multiple storage devices, with each device mounted to a specific location and level of access.

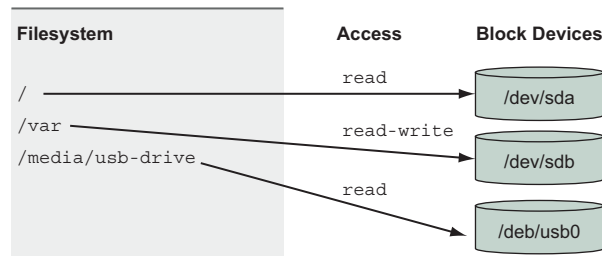


Figure 4.1 Storage devices attached to a filesystem tree at their mount point

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, we can mount nonimage-related storage at other points in a container file tree. That is exactly how containers get access to storage on the host filesystem 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:

- Bind mounts
- In-memory storage
- Docker volumes

These storage types can be used in many ways. Figure 4.2 shows an example of a container filesystem that starts with the files from the image, adds an in-memory `tmpfs` at `/tmp`, bind-mounts a configuration file from the host, and writes logs into a Docker volume on the host.

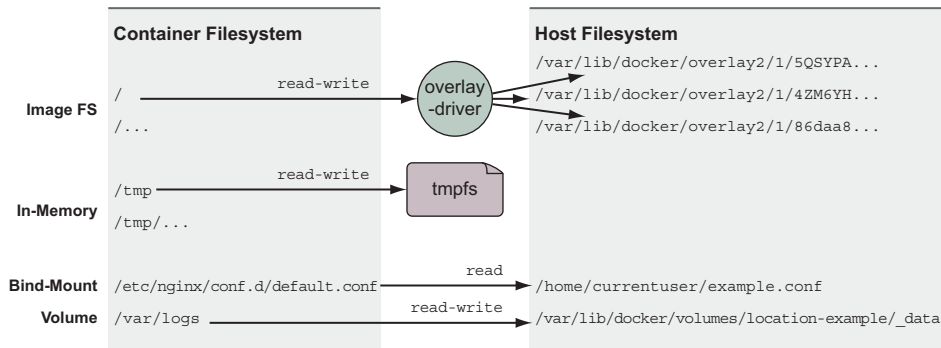


Figure 4.2 Example of common container storage mounts

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

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

Consider the example in figure 4.3. Suppose you're running a web server that depends on sensitive configuration on the host and 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.

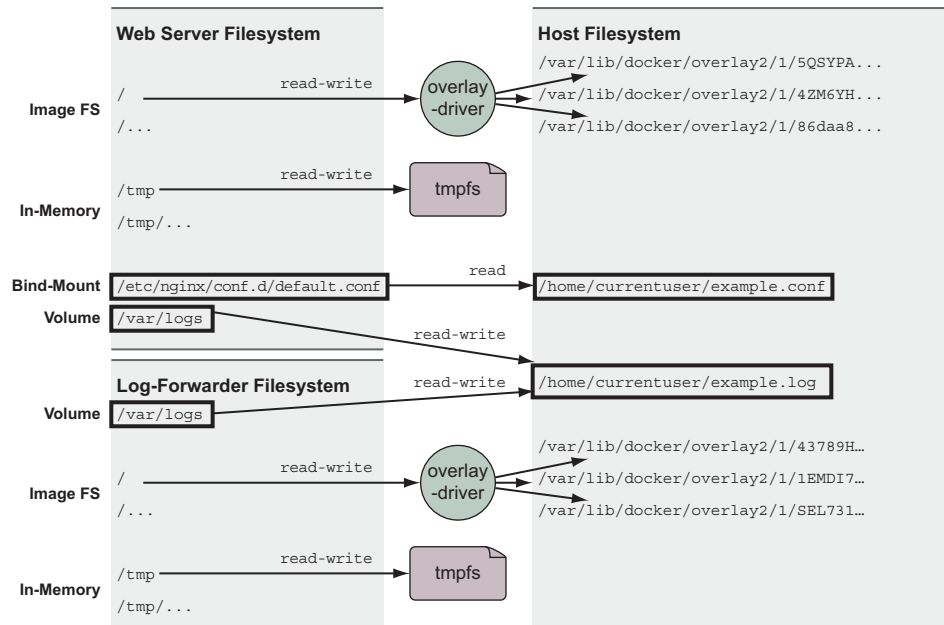


Figure 4.3 Host files shared as a bind-mount volumes

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 feature of volumes. When you mount a volume on a container filesystem, 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 \
  --mount type=bind,src=${LOG_SRC},dst=${LOG_DST} \
  -p 80:80 \
  nginx:latest
```

**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 filesystem of a specific host. If a container description depends on content at a specific location on the host filesystem, 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 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 types 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 a memory-based filesystem into a container's file tree. Consider this 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 the following:

- A `tmpfs` device is mounted to the tree at `/tmp`.
- The device has a `tmpfs` filesystem.
- The tree is read/write capable.

- `suid` bits will be ignored on all files in this tree.
- No files in this tree will be interpreted as special devices.
- No files in this tree will be executable.
- 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-writable (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 KB and is not readable by other in-container users.

4.4 Docker volumes

Docker volumes are named filesystem *trees* managed by Docker. They can be implemented with disk storage on the host filesystem, or another more exotic backend such as 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 filesystem that you might specify with bind mounts.

If you were to convert the web server and log-forwarding container example from section 4.2 to use a volume for sharing access to the logs, the pair could run on any machine without considering other software that might have a conflict with static locations on disk. That example would look like figure 4.4, and the containers would read and write logs through the `location-example` volume.

You can create and inspect volumes by using the `docker volume create` and `docker volume inspect` subcommands. By default, Docker creates volumes by 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 filesystem 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 filesystem tree:

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

Creates the volume

Specifies the “local” plugin

Adds a volume label

Inspects the volume

Selects the location on the host

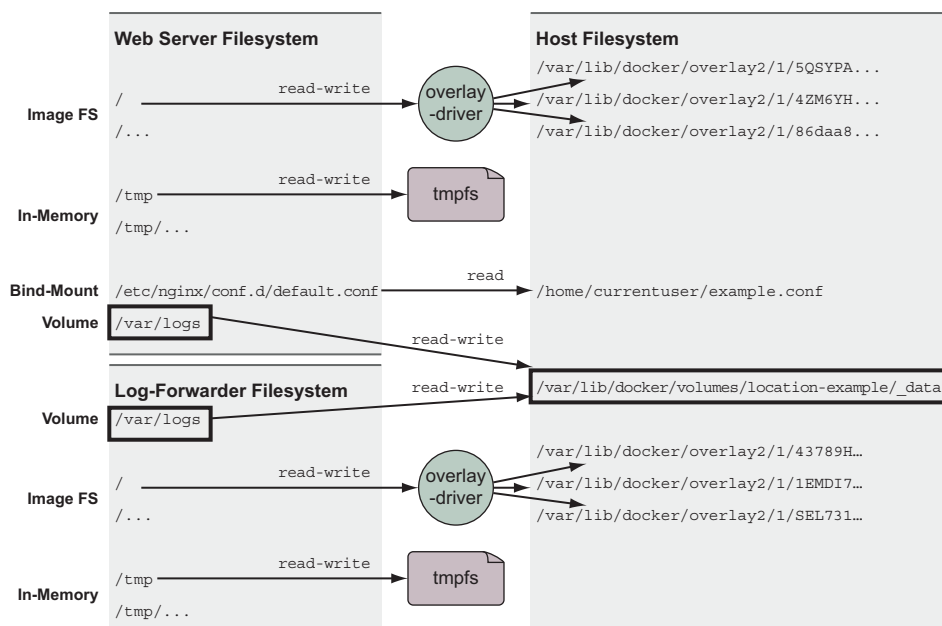


Figure 4.4 Sharing files between containers with a volume

Docker volumes may seem difficult to work with if you're manually building or linking tools together on your desktop, but for larger systems in which 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 filesystem, 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 by 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 such as 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 4.5.1.

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 make assertions only about what files should be available to a container. 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 filesystem 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 the key `example` and the value `cassandra`. Adding label metadata to your volumes can help you organize and clean up volumes later. You'll use this volume when you create a new container running Cassandra:

```
docker run -d \
  --volume cass-shared:/var/lib/cassandra/data \
  --name cass1 \
  cassandra:2.2
```

← Mounts 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 (CQLSH) 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 4.5 illustrates the system and what you will have built.

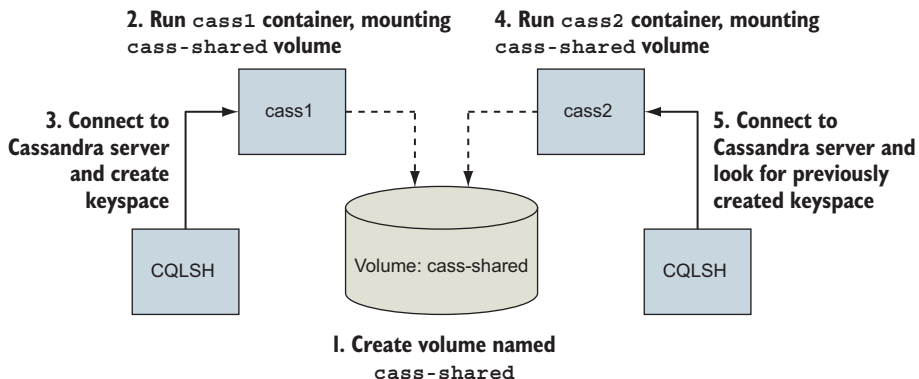


Figure 4.5 Key steps in creating and recovering data persisted to a volume with Cassandra

The next three commands test recovery of the data:

```
docker run -d \
  --volume 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, starting with the 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 disk space between containers. You can see it in action in the following example:

```
LOG_SRC=~/.web-logs-example
mkdir ${LOG_SRC}

docker run --name plath -d \
  --mount type=bind,src=${LOG_SRC},dst=/data \
  dockerinaction/ch4_writer_a

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

cat ${LOG_SRC}/logA

docker rm -f plath
```

Sets up a known location

Bind-mounts the location into a log-writing container

Bind-mounts the same location into a container for reading

Views the logs from the host

Stops 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 \
  --driver local \
  logging-example
```

| **Sets up a named volume**

```
docker run --name plath -d \
  --mount type=volume,src=logging-example,dst=/data \
  dockerinaction/ch4_writer_a
```

| **Mounts the volume into a log-writing container**

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

| **Mounts the same volume into a container for reading**

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

← **Views the logs from the host**

```
docker stop plath
```

← **Stops the writer**

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

This name conflict problem can be solved by 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 is assigned a unique identifier such as `1b3364a8debb5f653d1ecb9b190000622549ee2f812a4fb4ec8a83c43d87531b` instead of a human-friendly name. These are more difficult to work with if you are manually stitching together dependencies, 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 \
  --volumes-from fowler \
  --volumes-from knuth \
  alpine:latest ls -l /library/

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

Lists all volumes as they were copied into new container

Checks out 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 preceding example yields the following:

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

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

Creates an aggregation

Consumes volumes from a single source and lists 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 indicated by the specified containers. For example, if the student in the previous 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, 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 lampport --volume /library/ss \
  alpine:latest echo "Lampport collection created."

docker run --name student \
  --volumes-from chomsky --volumes-from lampport \
  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 of this limitation may occur if you copy 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), 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 in which you can't use `--volumes-from` occurs when you need to change the write permission of a volume. This is because `--volumes-from` copies the full volume 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 these are in managing file permissions.

Using volumes decouples containers from the data and filesystem 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 shows 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 in 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 when 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 run 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 preceding command will delete all three named volumes.

There is only one constraint for deleting volumes: no volume that is currently in use can be deleted. More specifically, 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 the 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 procedures, 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 disks can create problems. If you're running software on a cluster of machines, 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, 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 use all types of backing storage including proprietary cloud block storage, network filesystem mounts, specialized storage hardware, and on-premises cloud solutions such as 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, although a few projects simplify this to some degree. REX-Ray (<https://github.com/rexray/rexray>) is a popular open source project that provides volumes on several cloud and on-premises 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 REX-Ray.

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 in depth, including the following:

- Mount points allow many filesystems 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 filesystem into a container.
- In-memory filesystems can be attached to a container file tree so that 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 filesystem 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, structures, and tools required for container networks. If you want to run a website, database, email server, or any software that depends on networking, such as a web browser inside a Docker container, 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; 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 need to consider only 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 the 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. Several people might 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 open only their own messages.

In reality, ports are just numbers and defined as part of the Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). Again, the details of the protocol

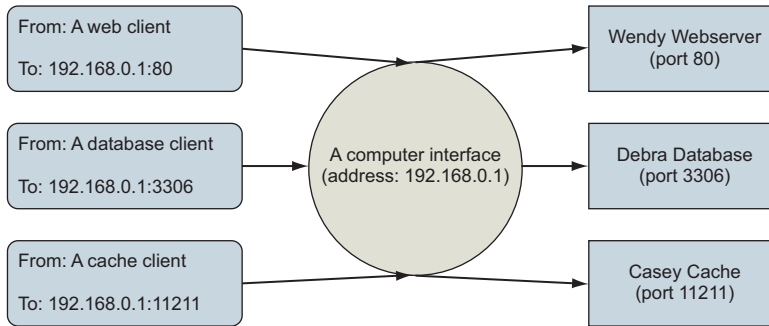


Figure 5.1 Processes use the same interface and are uniquely identified in the same way that multiple people might use the same mailbox.

are beyond the scope of this book, but we encourage you to read about it sometime. People who created standards for protocols, or companies that own a particular product, decide which 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 as 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 as well as the way 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 the way 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.

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

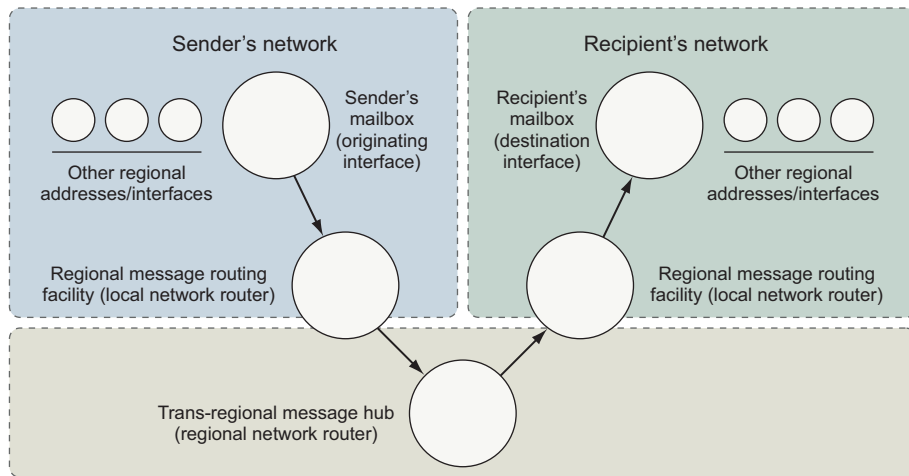


Figure 5.2 The path of a message in a postal system and a computer network

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 need to be comfortable with only this abstract idea.

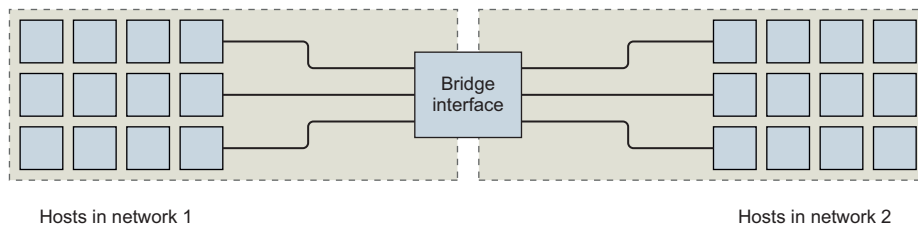


Figure 5.3 A bridge interface connecting two distinct networks

This has been a very rough introduction to some nuanced topics. The explanation 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 the container network. Section 5.5 covers a few methods for dealing with this edge case.

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 by 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, and 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 intercontainer 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 themselves.

The *scope* of a network can take three values: `local`, `global`, or `swarm`. This indicates whether 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 directly route traffic between containers running on different machines.

The default `bridge` network maintains compatibility with legacy Docker and cannot take advantage of modern Docker features including service discovery or 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

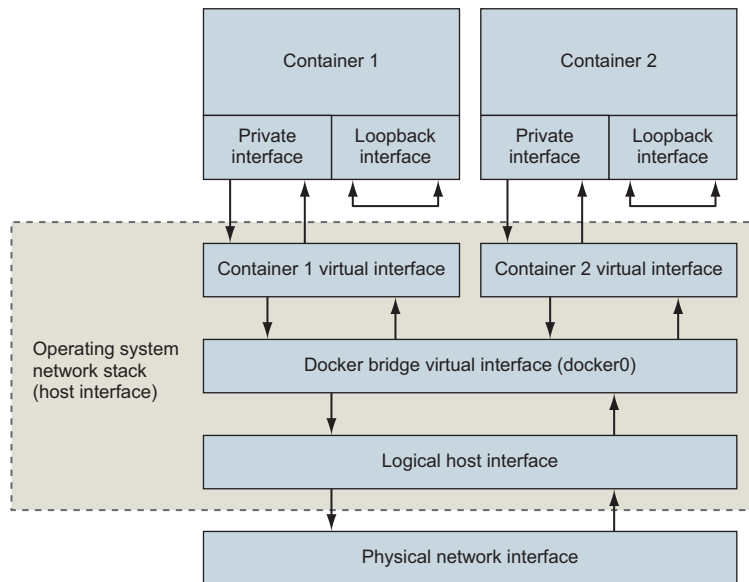


Figure 5.4 The default local Docker network topology and two attached containers

where the host is attached. Figure 5.4 illustrates two containers attached to a bridge network and its components.

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, 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 be used only 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 Ctrl-P and then Ctrl-Q) and 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 \
  network-explorer
```

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 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\ ...
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 by 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 only two devices are 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 (Ctrl-P, Ctrl-Q) and start another container attached to `user-network2`. Run the following:

```
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 confirms that the network attachment works as expected, and 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 can 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 work on only a single machine. They are not cluster-aware, and the container IP addresses are not routable from outside 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 such as a LAMP stack running a content management system, or most local development tasks. But if you are running a multiserver environment that is designed to tolerate machine failure, 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, 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 looks like an independent node on the network.

If you are running Docker for Mac or Docker for Windows or are running in a managed cloud environment, those options will not work. Further, underlay network configuration is dependent on the host network, and so definitions are rarely portable. The more popular multihost 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 multihost-aware and can route intercontainer connections between every node in a swarm.

Just like on a bridge network, containers on an overlay network are not directly routable from outside the cluster. But intercontainer 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 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 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 multitenant environments and should be disallowed for third-party containers. Along these lines, you'll often want to not 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 multiprocess containers can still use connections to localhost for interprocess 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:

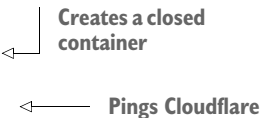
```
docker run --rm \
  --network none \
  alpine:3.8 ip -o addr
```

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 demonstrated. 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 \
  alpine:3.8 \
  ping -w 2 1.1.1.1
```



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 the host running the Docker engine.

Bridge networks use network address translation (NAT) to make all outbound container traffic with destinations outside 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 located. 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 we'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 a container have no way of knowing that they are running inside a container, that they are bound to a container network, or which port is being forwarded from the host.

Docker provides a mechanism for looking up port mappings. That feature is critical when 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 when multiple ports are published:

```

docker run -d \
  -p 8080 \
  -p 3000 \
  -p 7500 \
  --name multi-listener \
  alpine:3.8 sleep 300
docker port multi-listener 3000

```

Publishes
multiple ports

Looks up 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 software that's in two containers in the same container network 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 internetwork 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 (bidirectional) unrestricted network access.

5.5.2 Custom DNS configuration

Domain Name System (DNS) is a protocol for mapping hostnames 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 hostname of a new container. This flag adds an entry to the DNS override system inside the container. The entry maps the provided hostname to the container's bridge IP address:

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

← Sets the container hostname

← Resolves the hostname to an IP address

This example creates a new container with the hostname `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 hostname 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 \
  alpine:3.8 \
  nslookup 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 hostname suffix. With one set, any hostnames that

don't have a known top-level domain (for example, .com or .net) will be searched for with the specified suffix appended:

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

← Sets search domain
← Looks 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 hostname. 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 \
  --dns 1.1.1.1 \
  alpine:3.8 cat /etc/resolv.conf
```

Will display contents that look like:
search docker.com
nameserver 1.1.1.1

← Sets search domain
← Sets primary DNS server

This feature is most often used for trivialities such as 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 hardcoded environment-specific names such as `myservice.dev.mycompany.com`), you might consider using DNS search domains and using environment-unaware names (for example, `myservice`):

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

← Note dev prefix.
← Resolves to **myservice.dev.mycompany**

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

← Note test prefix.
← Resolves to **myservice.test.mycompany**

Using this pattern, the only change is the context in which the program is running. As with 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 set for a container only 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 hostname pair:

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

← Adds host entry

← 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 hostnames by mapping them to a known IP address such as 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 such as 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, we're 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 \
  --add-host docker.com:127.0.0.1 \
  --add-host test:10.10.10.2 \
  alpine:3.8 \
  cat /etc/hosts
```

← Sets hostname

← Creates host entry

← Creates another
host entry

← Views 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 by 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 which 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 the network provider layer, a whole continuum of service-discovery tools use various 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, 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 in which some container networking has been externalized, 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.

Summary

Networking is a broad subject that would take several books to cover properly. 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 intercontainer network communication with built-in container name resolution.

- Docker provides two other special networks by default: host and none.
- 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.

Limiting risk with resource controls

This chapter covers

- Setting resource limits
- Sharing container memory
- Setting users, permissions, and administrative privileges
- Granting access to specific Linux features
- Working with 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 touched on the differences a bit. Chapters 2 through 5 each covered a different isolation feature set of Docker containers. This chapter covers the remaining four and 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 from misbehaving because of 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 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, we try to give you an idea about their purpose and basic usage examples, and how you can integrate them with Docker. Figure 6.1 shows the eight namespaces and features that are used to build Docker containers.

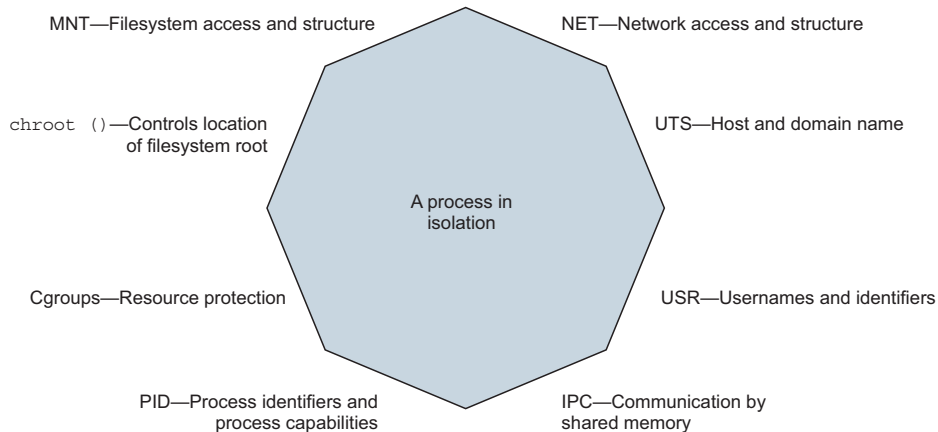


Figure 6.1 Eight-sided containers

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 **Setting resource allowances**

Physical system resources such as 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, 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 \
  --cpu-shares 1024 \
  --cap-drop net_raw \
  -e MYSQL_ROOT_PASSWORD=test \
  mariadb:5.5
```

← Sets 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, we 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 don't 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 that 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 an important latency-sensitive data-processing program, a revenue-generating web application, or a backend 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 \
--cap-drop net_raw \
--link ch6_mariadb:mysql \
-e WORDPRESS_DB_PASSWORD=test \
wordpress:5.0.0-php7.2-apache
```

← Sets a relative process weight

This command will download and start WordPress version 5.0. 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 (we'll call it `<port>`) that the service is running on and open `http://localhost:<port>` in your web browser. 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 in which 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.

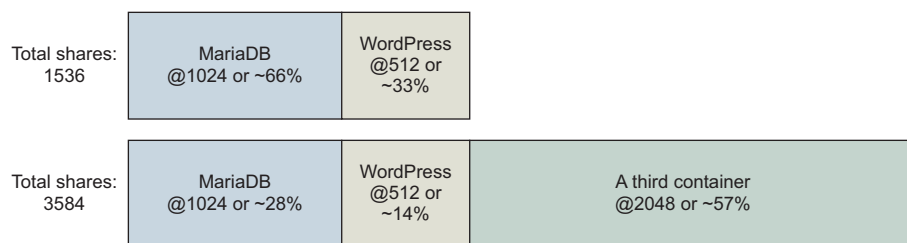


Figure 6.2 Relative weight and CPU shares

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, the container may burst well beyond its limits. This approach ensures 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 you 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 the 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 CPU cores:

```
docker container run -d -P --name ch6_wordpress \
--memory 512m \
--cpus 0.75 \
--cap-drop net_raw \
--link ch6_mariadb:mysql \
-e WORDPRESS_DB_PASSWORD=test \
wordpress:5.0.0-php7.2-apache
```

← Uses 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 multicore 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 \
  --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
```

← Restricts 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
```

We thought this was exciting when we first used it. To get the best appreciation, repeat this experiment a few times by 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 a list or range:

- 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 the host's devices by default, and Docker creates other devices specifically for each container. This works similarly to how a virtual terminal provides dedicated input and output devices to the user.

On occasion, it may be important to share other devices between a host and a specific container. Say you're running computer vision software that requires access to a webcam, for example. 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 \
  ubuntu:16.04 ls -al /dev
```

← Mounts 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 Sharing memory

Linux provides a few tools for sharing memory between processes running on the same computer. This form of interprocess 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 such as PostgreSQL.

Docker creates a unique IPC namespace for each container by default. The Linux IPC namespace partitions shared memory primitives such as named shared memory blocks and semaphores, as well as message queues. It's OK 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*

We'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 \ ← Starts producer
  --ipc shareable \
  dockerinaction/ch6_ipc -producer

docker container run -d -u nobody --name ch6_ipc_consumer \ ← Starts consumer
  dockerinaction/ch6_ipc -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 uses the same key to identify the shared memory resource, but they refer 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 works like the `--network` flag covered in chapter 5. Figure 6.3 illustrates the relationship between containers and their namespace shared memory pools.

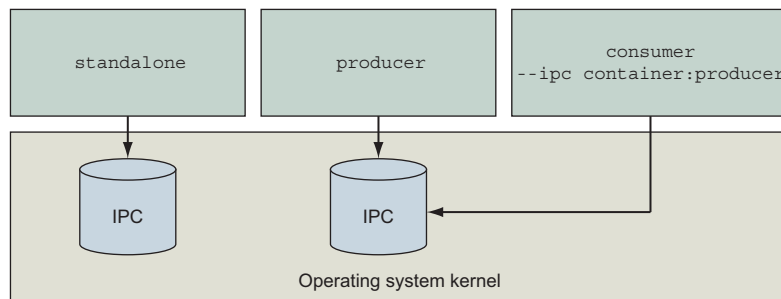


Figure 6.3 Three containers and their shared memory pools; producer and consumer share a single pool.

Use the following commands to test joined IPC namespaces for yourself:

```
docker container rm -v ch6_ipc_consumer
```

← Removes original consumer

```
docker container run -d --name ch6_ipc_consumer \
  --ipc container:ch6_ipc_producer \
  dockerinaction/ch6_ipc -consumer
```

← Starts new consumer

← Joins 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:

```
docker rm -vf ch6_ipc_producer ch6_ipc_consumer
```

- 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.

Reusing the shared memory namespaces of containers has obvious security implications. 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 from the image's page on Docker Hub.

6.3 Understanding users

Docker starts containers as the user that is 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, it 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.

Reasonable exceptions exist; sometimes using the root user is the best or only available option. You use the root user for building images, and at runtime when there's no other option. Similarly, at times you might 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 know 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 such as the default user in Docker Hub. You can inspect image metadata by 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 image pull busybox:1.29
docker image inspect busybox:1.29
docker inspect --format "{{.Config.User}}" busybox:1.29
```

← Displays all of busybox's metadata

← Shows 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.

This approach has a problem. 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:1.29 whoami
docker container run --rm --entrypoint "" busybox:1.29 id
```

← Outputs: root

← 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:1.29`). 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.

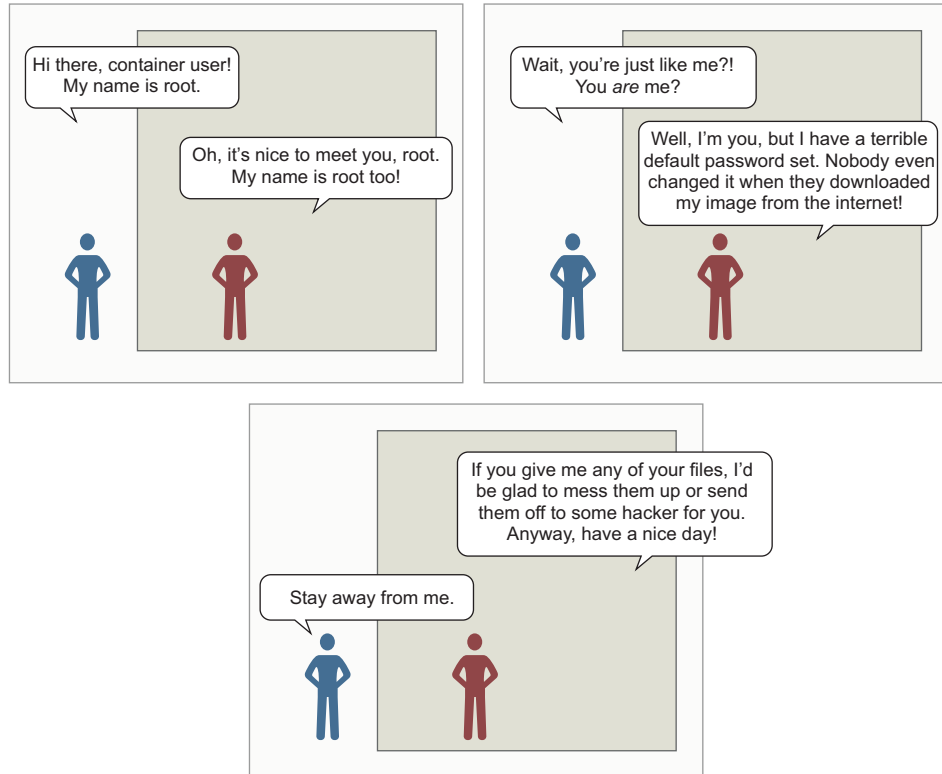


Figure 6.4 Root versus root—a security drama

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:1.29 awk -F: '$0=$1' /etc/passwd
```

We 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 from the container filesystem and pull the list of 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 \
  busybox:1.29 id
```

← **Sets run-as user to nobody**

← **Outputs: uid=65534(nobody)
gid=65534(nogroup)**

This command used the `nobody` user. That user is common and intended for use in restricted-privileges scenarios such as running applications. That is 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. When you specify a user by name, that name is resolved to the user ID (UID) specified in the container's `passwd` file. Then the command is run with that UID. This leads to another feature. The `--user` flag also accepts 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:nogroup \
  busybox:1.29 id
```

← **Sets run-as user to nobody and group to nogroup**

← **Outputs: uid=65534(nobody)
gid=65534(nogroup)**

```
docker container run --rm \
  -u 10000:20000 \
  busybox:1.29 id
```

← **Sets UID and GID**

← **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 simplify running applications with limited privileges and solving file-permission problems.

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 such as turning a default nonroot user into the root user by using an `suid`-enabled program or opening up access to the root account without authentication. The threat of the `suid` example can be mitigated by using the custom container security options described in section 6.6, specifically the `--security-opt no-new-privileges` option. However, that's late in the delivery process to address that problem. Like a full Linux host, images should be analyzed and secured using the 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. Chapters 7, 8, and 10 cover how to create minimal application images.

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
chmod 600 garbage
sudo chown root garbage
docker container run --rm -v "$(pwd)"/garbage:/test/garbage \
  -u nobody \
  ubuntu:16.04 cat /test/garbage
docker container run --rm -v "$(pwd)"/garbage:/test/garbage \
  -u root ubuntu:16.04 cat /test/garbage
# Outputs: "e=mc^2"
# cleanup that garbage
sudo rm -f garbage

```

Creates new file on your host

Makes file readable only by its owner

Makes file owned by root (assuming you have sudo access)

Tries to read file as nobody

Tries 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 volumes, 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 (GID):

```
mkdir logFiles

sudo chown 2000:2000 logFiles

docker container run --rm -v "$(pwd)"/logFiles:/logFiles \
  -u 2000:2000 ubuntu:16.04 \
  /bin/bash -c "echo This is important info > /logFiles/important.log"

docker container run --rm -v "$(pwd)"/logFiles:/logFiles \
  -u 2000:2000 ubuntu:16.04 \
  /bin/bash -c "echo More info >> /logFiles/important.log"

sudo rm -r logFiles
```

Sets ownership of directory to desired user and group

Writes important log file

Sets UID:GID to 2000:2000

Appends to log from another container

Also sets 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 that 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 \
  -u root monitoringtool
```

Binds docker.sock from host into container as a read-only file

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 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 filesystem available inside a container has been mounted so that changes made inside that container will stay inside that container's filesystem. But this does impact volumes in which 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 `subuid` and `subgid` 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 permission issues in cases like reading and writing to volumes. Let's step through an example of sharing a filesystem between containers whose process runs as UID 0 in the container with user namespacing enabled. In our example, we will assume Docker is using the following:

- 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 `dockremap` user on the host. Then, we will create a shared directory owned by the remapped container UID 0, host UID 5000.

```
# id dockremap
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
```

← **Inspects user and group ID of dockremap user on host**

← **Changes 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 -v /:/host alpine ash
/ # touch /host/afile
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
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
```

← **The /host mount is owned by host's UID and GID: 0:0, so write disallowed**

← **UID for root in container was 0**

← **/tmp/shared is owned by host's nonprivileged UID and GID: 5000:5000, so write allowed**

This example demonstrates the implications on filesystem access when using user namespaces with containers. User namespaces can be 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 backend 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 an explicit list of capabilities that are necessary and safe to run most applications. This further isolates 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 reasons 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 at times you'll 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 by using the `--cap-drop` flag on `docker container create` or `docker container run`. First, print the default capabilities of a containerized process running on your machine and notice that `net_raw` is in the list of capabilities:

```
docker container run --rm -u nobody \
  ubuntu:16.04 \
  /bin/bash -c "capsh --print | grep net_raw"
```

Now, drop the `net_raw` capability when starting the container. Grep cannot find the string `net_raw` because the capability has been dropped and so there is no visible output:

```
docker container run --rm -u nobody \
  --cap-drop net_raw \
  ubuntu:16.04 \
  /bin/bash -c "capsh --print | grep net_raw"
```

← | **Drops 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 unprefix 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:16.04 \
  /bin/bash -c "capsh --print | grep sys_admin"

```

SYS_ADMIN is
not included.

```

docker container run --rm -u nobody \
  --cap-add sys_admin \
  ubuntu:16.04 \
  /bin/bash -c "capsh --print | grep sys_admin"

```

Adds SYS_ADMIN

Like other container-creation options, both `--cap-add` and `--cap-drop` can be specified multiple times to add or drop multiple capabilities, respectively. 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 whether 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

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 filesystem and network isolation but have full access to shared memory and devices and possess full system capabilities. You can perform several interesting tasks, including running Docker inside a container, with privileged containers.

The bulk of the uses for privileged containers is administrative. Take, for example, an environment in which the root filesystem 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:16.04 id

```

Checks
out IDs

```

docker container run --rm \
  --privileged \
  ubuntu:16.04 capsh --print

```

Checks out Linux
capabilities

```

docker container run --rm \
  --privileged \
  ubuntu:16.04 ls /dev

```

Checks out list of
mounted devices

```
docker container run --rm \
  --privileged \
  ubuntu:16.04 networkctl
```

← Examines 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 Strengthening 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 kernel system calls (syscalls) that most programs don't 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 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 away any one thing 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 deeper insight before switching to thoughts 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 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. 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 the improvements it provides are far better than the alternative of forgoing any isolation because of 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 all syscalls by default and then explicitly permits more than 260 syscalls as safe for use by most programs. The 44 blocked system calls are unneeded or are unsafe for normal programs (for example, `unshare`, used in creating new namespaces) or cannot be namespaced (for example, `clock_settime`, which 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:16.04 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 LSM providers. Both provide mandatory access control, or 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 seven formats:

- To prevent a container from gaining new privileges after it starts, use `no-new-privileges`.
- 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 at which 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 Building use-case-appropriate containers

Containers are a cross-cutting concern. There are more reasons and ways that people could use them than we 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, we 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 that 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 snoopers 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 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 filesystem 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.

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 UFS 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 nonroot 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 such as 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. Sometimes 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.



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.


```

docker container rm -vf hw_container
docker container run --rm \
  hw_image \
  ls -l /HelloWorld

```

← Removes changed container

← Examines 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, we recommend that you spend some time learning how to use Git. At the moment, though, you need to know only 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 filesystem. 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 filesystem changes*

Docker has a command that shows you all the filesystem 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
```

← **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
docker container diff tweak-a
# Output:
#   A /HelloWorld
```

← **Adds new file to busybox**

```
docker container run --name tweak-d busybox:latest rm /bin/vi
docker container diff tweak-d
# Output:
#   C /bin
#   D /bin/vi
```

← **Removes existing file from busybox**

```
docker container run --name tweak-c busybox:latest touch /bin/vi
docker container diff tweak-c
# Output:
#   C /bin
#   C /bin/busybox
```

← **Changes 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 filesystem, 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 by 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 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
docker container commit -m "Set CMD git" \
  -a "@dockerinaction" cmd-git ubuntu-git
```

← Shows standard git help and exit

← Commits new image to same name

```
docker container rm -vf cmd-git      ↵ Cleanup
docker container run --name cmd-git ubuntu-git version  ← 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 that 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 *Configuring image attributes*

When you use `docker container commit`, you commit a new layer to an image. The filesystem 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 we 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 \
  busybox:latest      ↵ Creates environment
                    variable specialization
docker container commit rich-image-example rie      ↵ Commits
                                                    image
docker container run --rm rie \
  /bin/sh -c "echo \${ENV_EXAMPLE1} \${ENV_EXAMPLE2}"      ↵ 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" \
  rie \
  -c "echo \${ENV_EXAMPLE1} \${ENV_EXAMPLE2}"      ↵ Sets default
                                                    entrypoint
                                                    ↵ Sets default
                                                    command
docker container commit rich-image-example-2 rie      ← Commits image
docker container run --rm rie      ↵ 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 such as `ubuntu:latest` or `busybox:latest`. Then you made changes to the filesystem 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 filesystem 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 Exploring union filesystems

Understanding the details of union filesystems is important for image authors for two reasons:

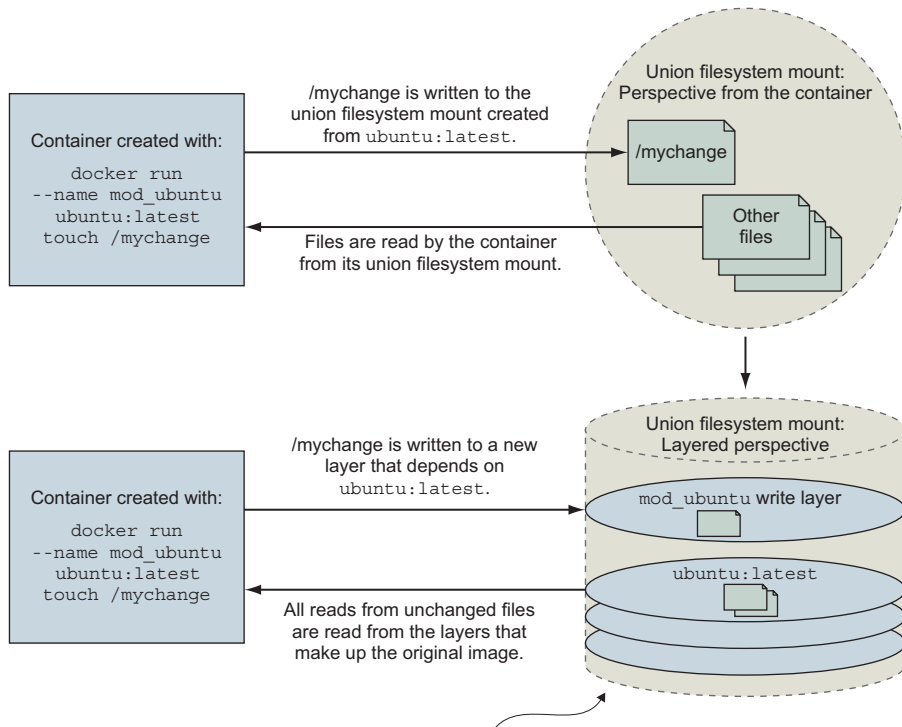
- Authors need to know the impact that adding, changing, and deleting files have on resulting images.
- Authors need have a solid understanding of the relationship between layers and how layers relate to images, repositories, and tags.

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 filesystem. As discussed in chapters 3 and 4, the root filesystem is provided by the image that the container was started from. That filesystem is implemented with a union filesystem.

A union filesystem is made up of layers. Each time a change is made to a union filesystem, 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 filesystem. Figure 7.2 illustrates the two perspectives for this example.



By looking at the union filesystem from the side—the perspective of its layers—you can begin to understand the relationship between different images and how file changes impact image size.

Figure 7.2 A simple file write example on a union filesystem from two perspectives

When you read a file from a union filesystem, that file will be read from the topmost 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.

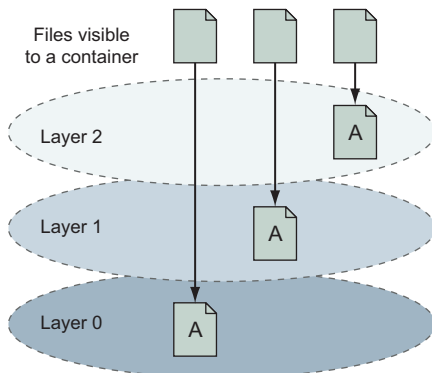


Figure 7.3 Reading files that are located on different layers

All this layer functionality is hidden by the union filesystem. 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 filesystem 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 filesystem 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 is 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 filesystem 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 filesystems, and we will examine that a little deeper next.

Most union filesystems 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.

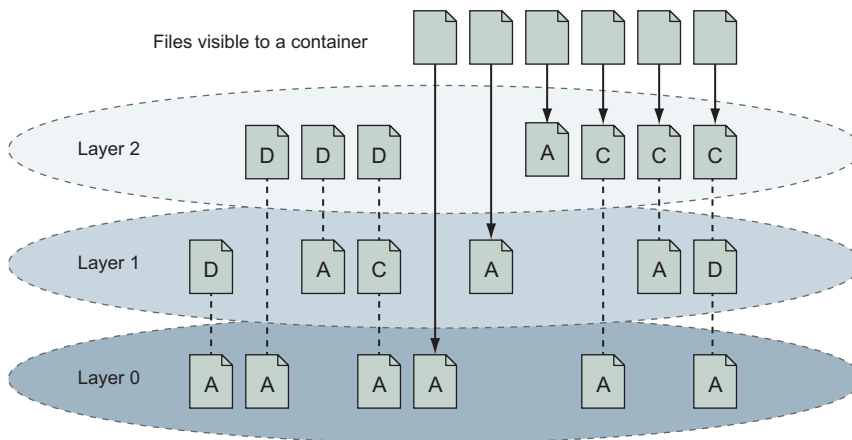


Figure 7.4 Various file addition, change, and deletion combinations over a three-layered image

Knowing how filesystem 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 by 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 filesystem is made up of a stack of layers, and 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 filesystem, 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.

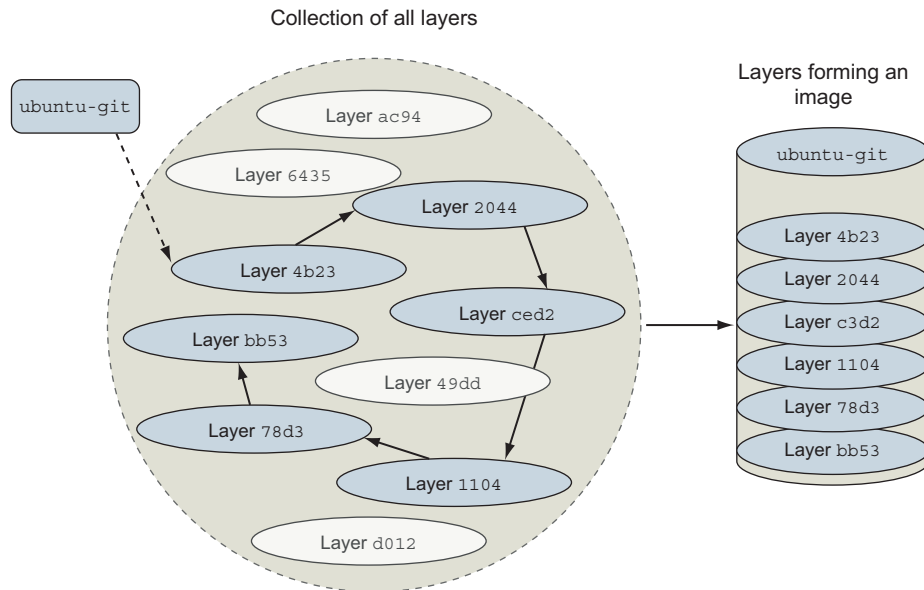


Figure 7.5 An image is the collection of layers produced by traversing the parent graph from a top layer.

Images are stacks of layers constructed by traversing the layer dependency graph from a 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:

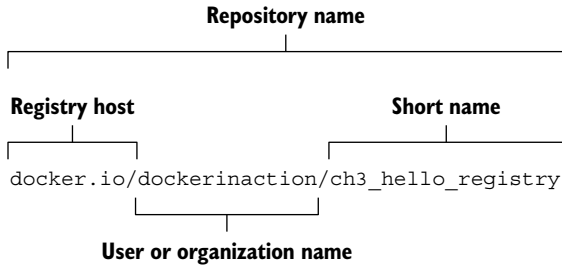
```
docker container commit mod_ubuntu
```

That `commit` subcommand will generate output that includes a new image ID like this:

```
6528255cda2f9774a11a6b82be46c86a66b5feff913f5bb3e09536a54b08234d
```

You can create a new container from this image by 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 Docker Hub registry, but we have used the fully qualified registry hostname, `docker.io`. It's named for the user (`dockerinaction`) and a unique short name (`ch3_hello_registry`). If you pull this repository without specifying a tag, Docker will try to pull an image tagged with `latest`. You can pull all tagged images in a repository by adding the `--all-tags` option to your pull command. In this example, there's only one tag: `latest`. That tag points to a layer with the short form ID `4203899414c0`, as illustrated in figure 7.6.

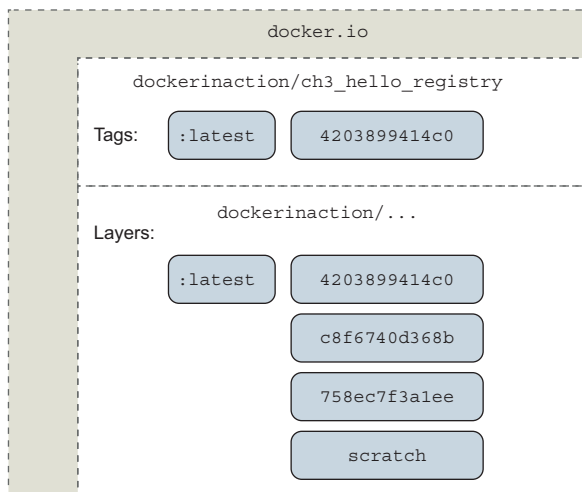


Figure 7.6 A visual representation of repositories

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 need only 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, as 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 any time 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 filesystems, Docker images would quickly become unusable. For example, suppose you want 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 ← Creates 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 \
  ubuntu-git:latest -c "apt-get remove -y git"
  ↳ Executes bash command
  ↳ Removes Git
```

```
docker container commit image-dev2 ubuntu-git:removed ← Commits image
```

```
docker image tag ubuntu-git:removed ubuntu-git:latest
  ↳ Reassigns latest tag
```

```
docker image ls
  ↳ Examines 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 filesystem.

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, your consumers will benefit. In the early days of Docker, image authors sometimes minimized the number of layers in an image because of the limits of image storage drivers. Modern Docker image storage drivers do not have image layer limits that normal users will encounter, so design for other attributes such as size and cacheability.

You can examine all the layers in an image by 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 this:

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 by saving the image to a TAR file with `docker image save`, and then importing the contents of that filesystem back into Docker with `docker image`

import. But that's a bad idea, because you lose the original image's metadata, its change history, and any savings customers might get when they download images with the same lower levels. 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 filesystems.

7.3 Exporting and importing flat filesystems

On some occasions, it's advantageous to build images by working with the files destined for an image outside the context of the union filesystem 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 filesystem 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 filesystem 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 filesystem 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
docker container export --output contents.tar export-test
docker container rm export-test
tar -tf contents.tar
```

Exports filesystem contents

Shows archive contents

This will produce a file in the current directory named `contents.tar`. That file should contain two files from the `ch7_packed` image: `message.txt` and `folder/message.txt`. At this point, you could extract, examine, or change those files to whatever end. The archive will also contain some zero-byte files related to devices and files that Docker manages for every container such as `/etc/resolv.conf`. You can ignore these. If you had omitted the `--output` (or `-o` for short), then the contents of the filesystem 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 filesystem import. Importing filesystems 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 by using the `docker import` command:

```
docker import -c "ENTRYPOINT [\"/hello\"]" - \
    dockerinaction/ch7_static < static_hello.tar
```

← 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 filesystem.

You tagged the resulting image as the `dockerinaction/ch7_static` repository. Take a moment to explore the results:

```
docker container run dockerinaction/ch7_static
docker history dockerinaction/ch7_static
```

← 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 you produced was small for two reasons. First, the program you produced was only just over 1.8 MB, and you 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.

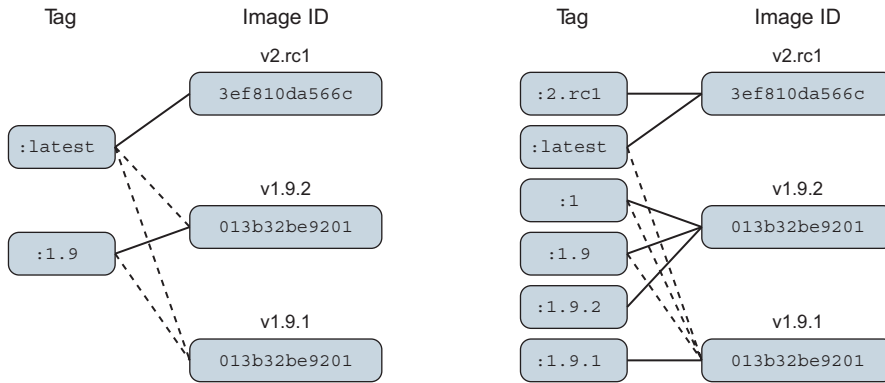


Figure 7.7 Two tagging schemes (left and right) for the same repository with three images. Dotted lines represent old relationships between a tag and an image.

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, 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.

The latest tag has other problems. 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, such as 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.

When 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 the base operating system. 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.

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 by 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.

Building images automatically with Dockerfiles

This chapter covers

- Automated image packaging with Dockerfiles
- Metadata and filesystem instructions
- Creating maintainable image builds with arguments and multiple stages
- Packaging for multiprocess and durable containers
- Reducing the image attack surface and building trust

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 chapter 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 to 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 .
```

This 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. 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 `RUN` instruction executes the program with the arguments you specify on top of a new image layer. Then Docker commits the filesystem changes to the layer so they are available for the next Dockerfile instruction. In this case, the output for the `RUN` instruction was clouded with all the output for the command `apt-get update && apt-get install -y git`. Installing software packages is one of the most common use cases for the `RUN` instruction. You should explicitly install each software package needed by your container to ensure that it is available when needed.

If you're not interested in reams of build process 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

```

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 is limited in that all the files added to the image were downloaded from the network; the example modifies the environment in a limited way and provides a general tool. The next example, which has a more specific purpose and local code, provides a more complete Dockerfile primer.

8.2 *A Dockerfile primer*

Dockerfiles are expressive and easy to understand because of 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 handmade 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/engine/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 create 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're finished. This will prevent the `.dockerignore` file, or files named `mailer-base.df`, `mailer-logging.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 version 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:buster-20190910
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"]
EXPOSE 33333
# Do not set the default user in the base otherwise
# implementations will not be able to update the image
# USER example:example

```

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 (for example, `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 as in 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 values are 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 that is 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 the 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 as it would with 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:buster-20190910 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 in which 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, 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 Filesystem instructions

Images that include custom functionality will need to modify the filesystem. A Dockerfile defines three instructions that modify the filesystem: `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
RUN apt-get update && \
    apt-get install -y netcat
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 use 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 filesystem 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, just like `ENTRYPOINT` and other instructions, will honor both shell style and exec style arguments. But if any of the arguments contains whitespace, you 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. The `VOLUME` instruction will do only two things: create the defined location in the image filesystem 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, as shown in figure 8.1. They both take either shell or exec forms and are both used to start a process within a container. But a few important differences exist.

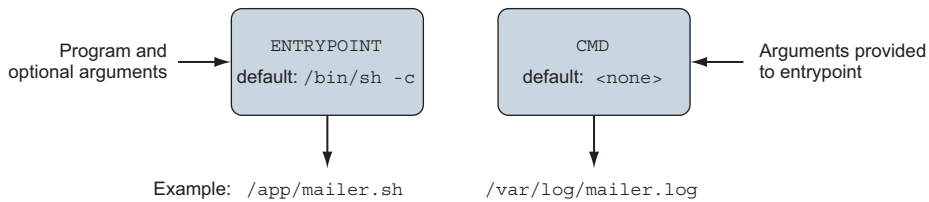


Figure 8.1 Relationship between `ENTRYPOINT` and `CMD`

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, 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, 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 `mailler.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 need to create the logging version of the mailer program. Create a directory at `./log-impl`. Inside that directory, create a file named `mailler.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 `mailler-logging` image from the directory containing `mailler-logging.df`:

```
docker image build -t dockerinaction/mailler-logging -f mailler-logging.df .
```

The results of this image build should be anticlimactic. Go ahead and start up a named container from this new image:

```
docker run -d --name logging-mailer dockerinaction/mailler-logging
```

The logging mailer should now be built and running. Containers that link to this implementation will have their messages logged to `/var/log/mailler.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 `mailler-live.df`:

```
FROM dockerinaction/mailler-base:0.6
ADD ["/live-impl", "${APPROOT}"]
RUN apt-get update && \
    apt-get install -y curl netcat python && \
    curl "https://bootstrap.pypa.io/get-pip.py" -o "get-pip.py" && \
    python get-pip.py && \
    pip install awscli && \
    rm get-pip.py && \
    chmod a+x "${APPROOT}/${APP}"
USER example:example
CMD ["mailler@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; 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 `mailler-live.df`.

The other instruction to note in this Dockerfile is `CMD`, where two arguments are passed. Here you're specifying the `From` and `To` fields on any emails that are sent. This differs from `mailler-logging.df`, which specifies only one argument.

Next, create a new subdirectory named `live-impl` under the location containing `mailler-live.df`. Add the following script to a file in that directory named `mailler.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/mailler-live -f mailler-live.df .
docker run -d --name live-mailer dockerinaction/mailler-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 need to set `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION` in order to get this example working. These environment variables define the AWS cloud credentials and location to use for this example. Discovering execution pre-conditions one by one as programs need them 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 other 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` 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 filesystem.

Next, create 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 our 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 of 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.

The 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 by 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:buster-20190910
ARG VERSION=unknown
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
```

← Defines 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 image inspect` to verify that the `VERSION` was substituted all the way down to the `base.version` label:

```
docker image inspect --format '{{ json .Config.Labels }}' \
  dockerinaction/mailler-base:0.6
```

The `inspect` command should produce JSON output that looks like this:

```
{
  "base.name": "Mailer Archetype",
  "base.version": "0.6",
  "maintainer": "dia@allingeek.com"
}
```

If you had not specified `VERSION` as a build argument, the default value of `unknown` would be used and a warning printed during the build process.

Let's turn our attention to multistage builds, which can help manage important concerns by distinguishing between phases of an image build. Multistage 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 demonstrates reusing parts of another image as well as separating an application's build and runtime concerns. First, let's learn about Dockerfile's multistage feature.

A *multistage 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. The 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 to 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 multistage builds with an example that uses two stages and a bit of composition; see figure 8.2. 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`.

This Dockerfile defines two stages: `builder` and `runtime`. The `builder` stage gathers dependencies and builds the example program. The `runtime` stage copies the

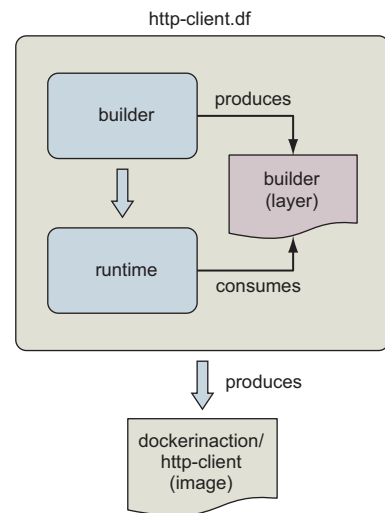


Figure 8.2 Multistage Docker builds

certificate authority (CA) and program files into the runtime image for execution. The source of the `http-client.df` Dockerfile is as follows:

```
#####
# 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, `builder` installs certificate authority files used to establish Transport Layer Security (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 simple. It makes an HTTP request to retrieve its own source code from GitHub:

```
package main

import (
    "net/http"
)
import "io/ioutil"
import "fmt"

func main() {
    url := "https://raw.githubusercontent.com/" +
```

```

    "dockerinaction/ch8_multi_stage_build/master/http-client.go"
    resp, err := http.Get(url)

    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 by 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 you 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 the following:

```

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 this:

```
docker build -t dockerinaction/http-client -f http-client.df .
```

The image can be executed as follows:

```
docker container run --rm -it dockerinaction/http-client:latest
```

When the `http-client` image runs successfully, it will output the `http-client.go` source code listed previously. 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 6 MB 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, 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), 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 start 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 filesystem 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 such as `/bin/sh` or `/bin/bash`. Shell scripts are the most common because shell programs are commonly available and they easily adapt to program and environment-specific requirements. 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 "$WEB_HOST" ]; then
  echo >&2 '[ERROR]: specify container to link; "web" or WEB_HOST env var'
  exit 1
fi
exec "$@" # run the default command
```

If you're unfamiliar with shell scripting, this is an appropriate time to learn it. The topic is approachable, and several excellent resources are available for self-directed learning. This specific script uses a pattern in which 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 entry-point 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 a file so the startup script for your application can access them.

Several open source init programs exist. Full-featured Linux distributions ship with heavyweight and full-featured init systems such as SysV, Upstart, and systemd. Linux Docker images such as 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, Supervisor, 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, you 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 you 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 whether 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 whether 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 health check for the NGINX web server:

```
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 nonzero status otherwise. The `|| exit 1` is a bit of shell trickery that means *or exit 1*. This means whenever `nc` exits with any nonzero status, `nc`'s status will be converted to 1 so that Docker knows the container is unhealthy. Conversion of nonzero exit statuses to 1 is a common pattern because Docker does not define the behavior of all nonzero health check statuses, 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:

```
docker image build -t dockerinaction/healthcheck .
docker container run --name healthcheck_ex -d dockerinaction/healthcheck
```

Now that a container with a health check is running, you 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 you 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 the following:

- *Time-out*—A time-out 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 toward the health status; once the health check command returns `healthy`, the container is considered started, and subsequent failures count toward 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 if one exists. This is useful for integrating a third-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.

You can do three things 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, each build will use the content identified with that CAIID as its base image. 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 USER 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 creating 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 to create other nonroot users and establish a nonroot 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, 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 `UserPermission-Denied.df`:

```
FROM busybox:1.29
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
```

The container should print an error message:

```
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 `usersns-remap` feature described in chapter 6 is enabled, 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 `usersns-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 when doing so can be avoided. With that in mind, using raw UID/GID numbers is still burdensome. 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 PostgreSQL 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 PostgreSQL containers require elevated privileges during startup. Instead, this particular image uses a `su` or `sudo`-like

program called `gosu` to start the PostgreSQL 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 specific application code, 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 `setuid` (SUID) or `setgid` (SGID) permissions. The well-known filesystem 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 nonroot user such as `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.

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 including the default user, exposed ports, default command, and entrypoint.
- Other Dockerfile instructions copy files from the local filesystem 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 multistage 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 nonroot default user, and disabling or removing any executable with `SUID` or `SGID` permissions.

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.

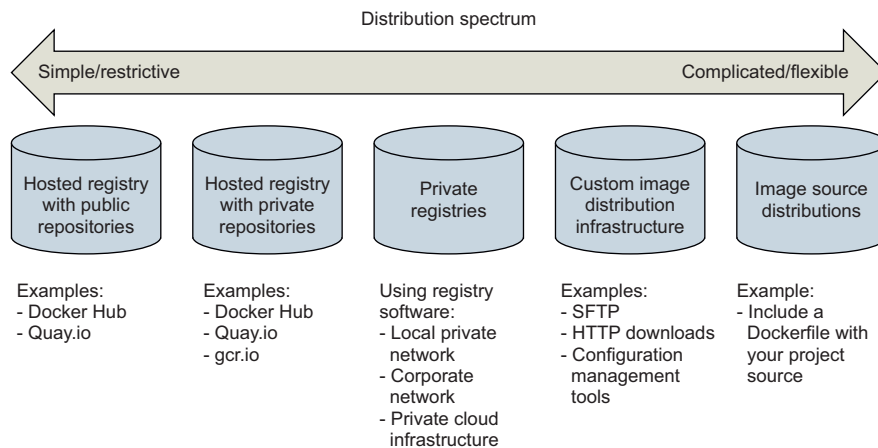


Figure 9.1 The image distribution spectrum

The methods included in the spectrum range from hosted registries such as Docker Hub to totally custom distribution architectures or source-distribution methods. We cover some of these subjects in more detail than others. We also place particular focus 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, longevity control is important. Docker makes it simple to switch between methods, and other criteria such as requisite expertise or cost may 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 a service-level agreement (SLA) 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. Varying degrees of access control are available. 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, in which 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, 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.

EXPERTISE

The last thing to consider when choosing a distribution method is the level of *expertise* required. Using hosted methods can be 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 various distribution methods. The following sections evaluate these methods against the criteria by 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.

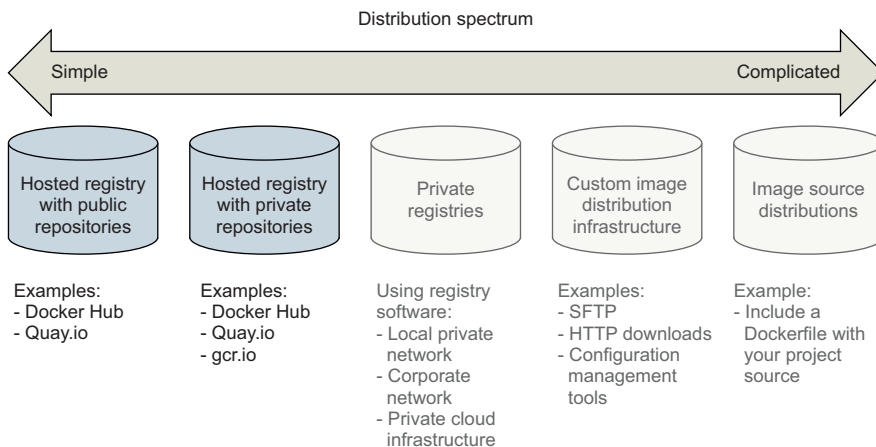


Figure 9.2 The simplest side of the distribution spectrum and the topic of this section

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 by 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
CMD echo 'Hello World!'
```

← 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 \
  -f HelloWorld.df \
  .
```

← 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 after you’ve logged in. Use the `docker push` command to do so:

```
docker image push <insert Docker Hub username>/hello-dockerfile
```

← 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
```

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 such as 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.
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.
Availability control	Worst	You have no availability control over hosted registries.

Table 9.1 Performance of public hosted repositories (continued)

Criteria	Rating	Notes
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 in 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.
Confidentiality	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, 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 authenticate with the registry where the repository is hosted. To do so, you 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	Good	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, 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. They are great tools for organizations that don't want to incur the overhead associated with running their own registry or for small private projects. Private repositories are poor candidates for publicizing availability of commercial software or distributing open source images.
Transport speed/size	Better	Any hosted registry such as 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 nonregistry solutions.
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.

Table 9.2 Performance of private hosted repositories (*continued*)

Criteria	Rating	Notes
Availability control	Worst/OK	No hosted registry provides any availability control. Unlike using public repositories, however, using private repositories will make you a paying customer. Paying customers may have stronger SLA guarantees or access to support personnel.
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.
Confidentiality	Worst	Despite the privacy provided by these repositories, they are never suitable for storing cleartext secrets or trade-secret code. Although the registries require user authentication and authorization to requested resources, these mechanisms have several potential problems. 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	As with 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.

Many free and commercially supported software packages are available for running a Docker image registry. If your organization has a commercial artifact repository for operating system or application software packages, it probably supports the Docker image registry API. A simple option for running a nonproduction image registry is to use Docker's registry software. The Docker registry, 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. Figure 9.3 illustrates that private registries fall in the middle of the distribution spectrum.

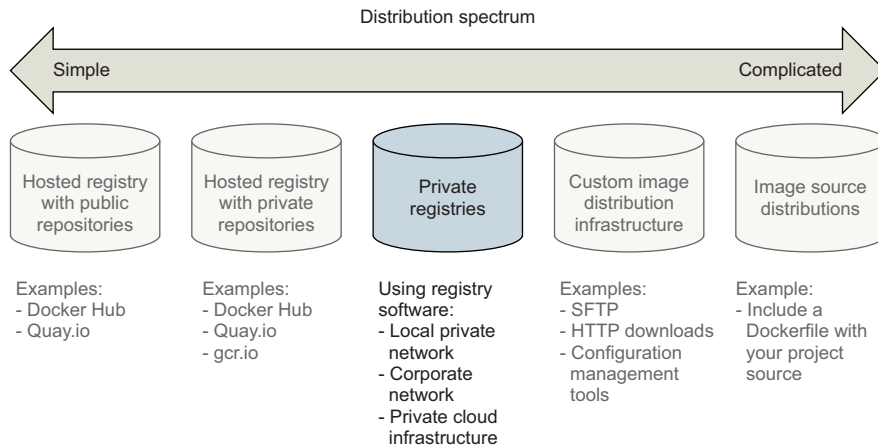


Figure 9.3 Private registries in the image distribution spectrum

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 a certain 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 such as Docker Hub.

Table 9.3 Performance of private registries (continued)

Criteria	Rating	Notes
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 because of 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.
Longevity control	Best	You have full control over solution longevity as the registry owner.
Availability control	Best	You have full control over availability 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 backend. For additional integrity protections, you can force the use of TLS over the network and use backend storage with encryption at rest.
Confidentiality	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. Many commercial product solutions are 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 when 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 in order 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 9.4.

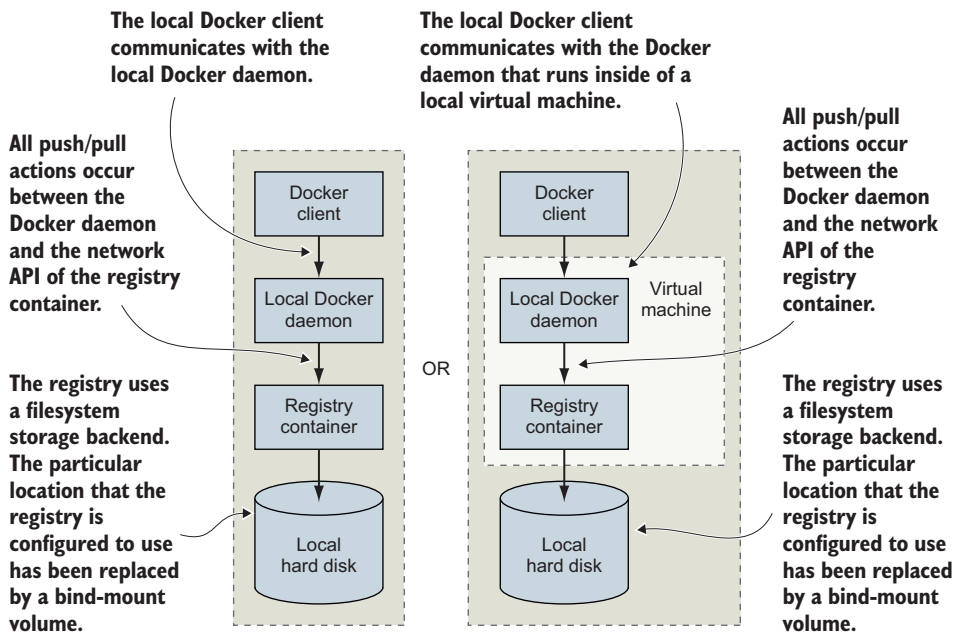


Figure 9.4 Interactions between the Docker client, daemon, local registry container, and local storage

Companies that want tight version control on their external image dependencies will pull images from external sources such as Docker Hub and copy them into their own registry. You might do this to ensure that 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
docker image ls -f "label=dia_excercise=ch9_registry_bound"
docker image tag dockerinaction/ch9_registry_bound \
  localhost:5000/dockerinaction/ch9_registry_bound
docker image push localhost:5000/dockerinaction/ch9_registry_bound

```

← Pulls demo image from Docker Hub

← Verifies image is discoverable with label filter

← Pushes demo image into registry

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 from which 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
docker image ls -f "label=dia_excercise=ch9_registry_bound"
docker image pull localhost:5000/dockerinaction/ch9_registry_bound
docker image ls -f "label=dia_excercise=ch9_registry_bound"
docker container rm -vf local-registry

```

← Removes tagged reference

← Pulls from registry again

← Demonstrates that image is back

← Cleans 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.

This is the most flexible distribution method that involves Docker registries. If you need greater control over 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 a USB drive 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 9.5.

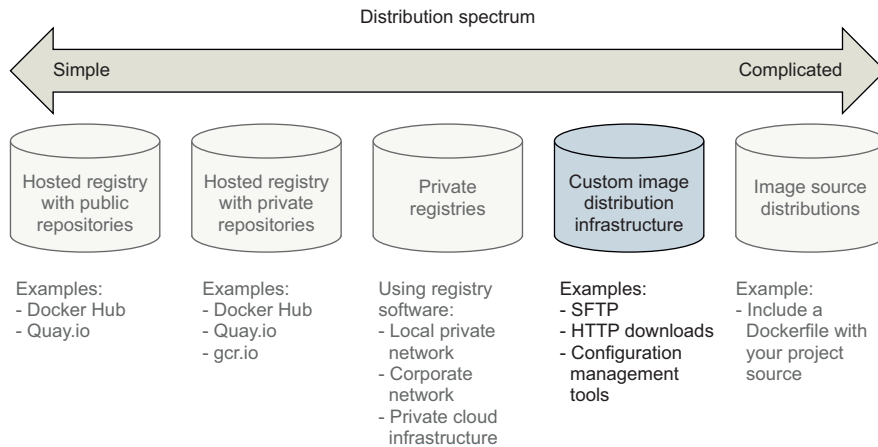


Figure 9.5 Docker image distribution over custom infrastructure

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 filesystems as flattened images. These techniques are the foundation for building distribution workflows like the one shown in figure 9.6.

This workflow 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 after you have an image in file form. One custom component not shown in figure 9.6 is the mechanism that uploads an image to the

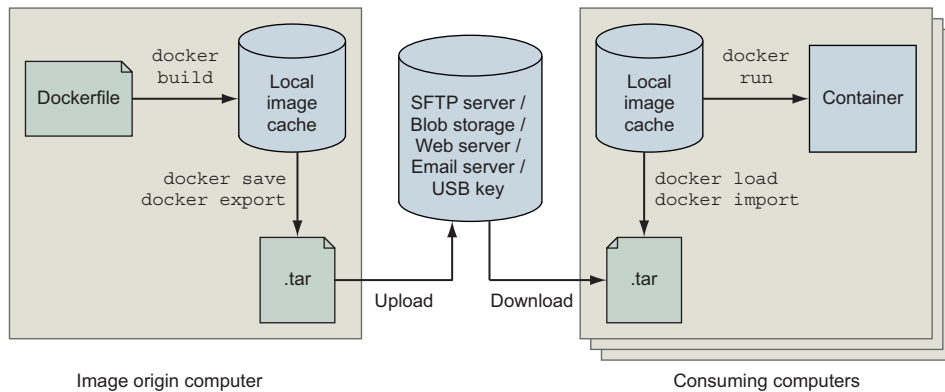


Figure 9.6 A typical manual distribution workflow with producer, transport, and consumers

transport. That mechanism may be a folder that is watched by a file-sharing tool such as 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. Clients require a process or mechanism to learn where the image is located and then acquire the image from the 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 such as 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.

Table 9.4 Performance of custom image distribution infrastructure (*continued*)

Criteria	Rating	Notes
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 filesystem.
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 such as 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.
Availability control	Best	If availability control is an important factor for your case, you can use a transport mechanism that you own.
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 <code>docker image save</code> and <code>load</code> .
Confidentiality	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 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 as easily use simple tools that you own 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 FTP*

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 9.7 illustrates what you'll build.

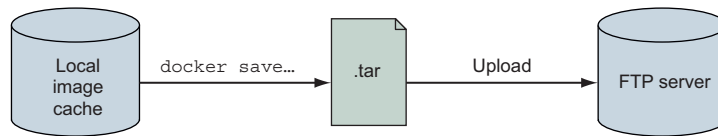


Figure 9.7 An FTP publishing infrastructure

The example in this section uses two existing images. The first, `dockerinaction/ch9_ftpd`, is a specialization of the `centos:6` image; `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 you 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 starts 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.

Next, 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 exports the `registry:2` image as a structured image file in your current directory. The file retains all the metadata and history associated with the image. At this point, you could inject all sorts of phases, such as 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
```

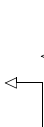
The registry image is now available for download to any FTP client that knows about the server and can access it over the network. But that file may never be overridden in the current FTP server configuration. You'd need to come up with your own versioning scheme if you were going to use a similar tool in production.

Advertising the availability of the image in this scenario requires clients to periodically poll the server by 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
docker image ls registry
```


Need to remove any registry containers first
Confirms the registry image has been removed

Then download the image file from your FTP server by 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
```

You can confirm that the image has been loaded from the archive by listing images for the registry repository again with `docker image ls registry`.

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 happens between containers on the same computer, so all the commands finish quickly. If a client connects to your FTP service over the network, 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.
Longevity control	Best	You can use the FTP server created for this example as long as you want.
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.
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 no integrity protections exist between file creation and upload or between download and import.
Confidentiality	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 need to familiarize yourself with <code>vsftpd</code> 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 such as Git on GitHub will have very different traits from using a file backup tool such as `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 9.8 shows image source distribution on the most complicated end of the spectrum.

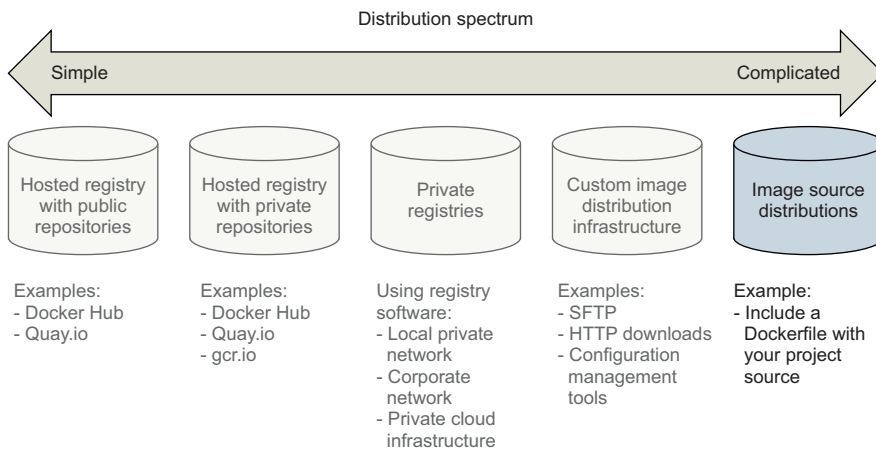


Figure 9.8 Using existing infrastructure to distribute image sources

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 don't need an account on Docker Hub or another Docker registry to publish an image.

Supposing a producer has an existing project, Dockerfile, and GitHub repository, their distribution workflow will 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.
Longevity control	Bad	Although Git is a popular tool and should be around for a while, you forgo any longevity control by integrating with GitHub or other hosted version-control providers.
Availability control	Worst	Relying on GitHub or other hosted version-control providers eliminates any availability control.
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.
Confidentiality	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.

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 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 and using metadata to help consumers use your image
- Common approaches for testing that images are configured correctly and secure
- Patterns for tagging images so they can be identified and delivered to consumers
- Patterns for publishing images to runtime environments and registries

In chapter 8, you learned how to build Docker images automatically by using Dockerfiles 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 that the image works under the expected operating conditions. Confidence in the correctness of the image artifact grows as it passes those tests. Next, 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. Pipelines help software authors quickly publish updates and efficiently deliver new features and fixes to consumers.

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 (CI) practices and is not specific to Docker images.

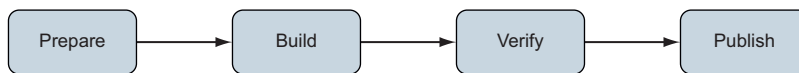


Figure 10.1 Generic artifact build pipeline

People often automate build pipelines with continuous integration systems such as Jenkins, Travis CI, 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 CI process for a Docker image is similar to other software artifacts and looks like figure 10.2.

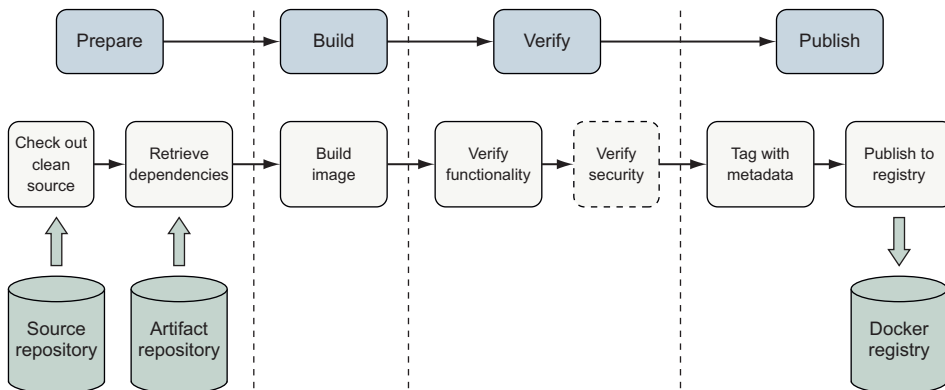


Figure 10.2 A Docker image build pipeline

When building a Docker image, this process includes the following steps:

- 1 Check out 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 by using a Dockerfile.
- 4 Verify that the image is structured and functions as intended.
- 5 (Optional) Verify that 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 CI systems operate. The exercises in this chapter will show how to build applications by using containers and how to package those application artifacts into a Docker image that runs the application. We will implement the build process by 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

Several patterns exist for building applications and images using containers. We will discuss three of the most popular patterns here:

- *All-in-One*—You use an all-in-one image to build and run the application.
- *Build Plus Runtime*—You use a build image with a separate, slimmer runtime image to build a containerized application.
- *Build Plus Multiple Runtimes*—You use a slim runtime image with variations for debugging and other supplemental use cases in a multi-stage build.

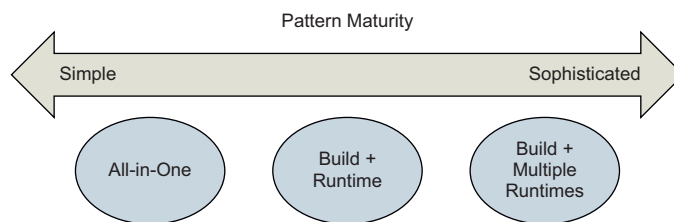


Figure 10.3 Image build pattern maturity

Multiple build patterns have evolved to produce images appropriate for particular consumption use cases. In this context, *maturity* refers to the 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 Build Plus Runtime 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*

All-in-one images include all the tools required to build and run an application. These tools might include software development kits (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 use the All-in-One pattern to build a simple Java web server using the popular Spring Boot framework. Here is an all-in-one Dockerfile that builds the application into an image along with the build tools:

```
FROM maven:3.6-jdk-11

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 as follows:

```
docker image build -t dockerinaction/ch10:all-in-one \
  --file all-in-one.df .
```

In this Dockerfile, the source image is the community Maven 3.6 image, which also includes OpenJDK 11. 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!”

All-in-one images have downsides. Because they 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. In addition, all-in-one images will be large, often 500 MB or more. The `maven:3.6-jdk-11` base image used in the example is 614 MB to start with, and the final image is 708 MB. 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 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 contain only what the application requires at runtime.

You can build the application with a Maven container:

```
docker container run -it --rm \
  -v "$(pwd)":/project/ \
  -w /project/ \
  maven:3.6-jdk-11 \
  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 16142344 Jul  2 15:17 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 simple Dockerfile that `COPYs` the application artifact into an image based on OpenJDK 10:

```
FROM openjdk:11-jdk-slim

COPY target/ch10-0.1.0.jar /app.jar

ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Build the runtime image:

```
docker image build -t dockerinaction/ch10:simple-runtime \
  --file simple-runtime.df .
```

Now run the web server image:

```
docker container run --rm -it -p 8080:8080 dockerinaction/ch10:simple-runtime
```

The application runs just as 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 (401 MB versus 708 MB!) and has a smaller attack surface.

This pattern is now supported and encouraged by many CI tools. The support usually comes in the ability to specify a Docker image to use as a hygienic execution environment for a step or the ability to run containerized build agents and assign steps to them.

10.2.3 Variations of runtime image via 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 adding specialized tools or changing 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 Build Plus Multiple Runtimes pattern of creating specialized images by 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.

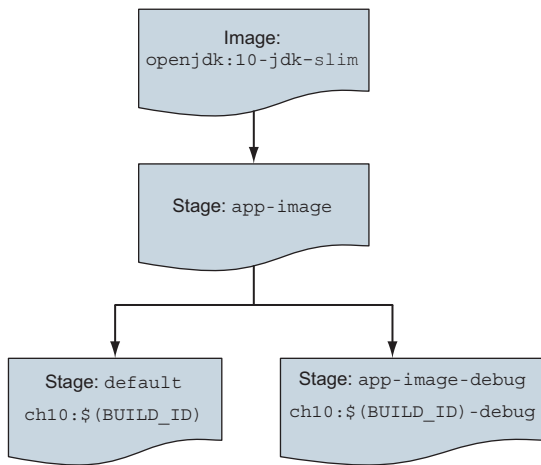


Figure 10.4 Image hierarchy for multi-stage build example

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:11-jdk-slim as app-image

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}" \

```

← **app-image build stage starts from openjdk.**

```

    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
#COPY needed debugging tools into image
ENTRYPOINT ["sh"]

FROM app-image as default

```

app-image-debug stage inherits and adds to the app-image.

default stage ensures app-image is produced by default.

The build stage of the main application image is declared as starting from `openjdk:11-jdk-slim` and named `app-image`:

```

# The app-image build target defines the application image
FROM openjdk:11-jdk-slim 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
#COPY needed debugging tools into image
ENTRYPOINT ["sh"]

```

Uses 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 dockerinaction/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:

```
# 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 `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 via 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 the build time or version-control system (VCS) commit identifier. Including the source Dockerfile in the image can be a quick reference for people debugging a problem to navigate within a container. Orchestrators and security tools may find other metadata describing the image's architectural role or security profile useful in deciding where the container should run or what it is permitted to do.

The Docker community Label Schema project has defined commonly used labels at <http://label-schema.org/>. Representing the recommended metadata by using the label schema and build arguments in Dockerfile looks like the following:

```
FROM openjdk:11-jdk-slim

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`, a widely available tool used to build programs, understands dependencies between the 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 directly implementing the flow of control between steps. These steps are called *rules*, and each rule is identified by a target name. Here is the general form of a `make` rule:

```

target ... : prerequisites ...
    recipe command 1
    recipe command 2
    ...

```

target identifies the rule with a logical name or filename produced by the rule.

prerequisites is an optional list of targets to build before this target.

The recipe section contains the list of commands used to build the target.

When you run the `make` command, it constructs a dependency graph from the prerequisites declared for each rule. The command uses this graph to calculate the sequence of steps to build a specified target. `make` has many features and quirks that we will not describe here, but you can read more about it 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 by using Docker on Windows, you should also investigate using the Windows Subsystem for Linux (WSL or WSL2) with Docker for Windows.

Here is a Makefile that will gather metadata, and then 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: clean
clean:
    @echo "Cleaning"
    rm -rf target

.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:3.6-jdk-11 \
        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
    @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)' \
```

The app-image target
requires building the
app-artifacts, metadata,
and linting target.


```

--build-arg VCS_REF='${VCS_REF}' \
.
@echo "Built App Image. BUILD_ID: $(BUILD_ID)"

.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}' \
.
@echo "Built Debug App Image. BUILD_ID: $(BUILD_ID)"

.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

```

You can build everything
with “make all”.

This Makefile defines a target for each build step we have discussed: gathering metadata, building the application, and building, testing, and tagging the image. Targets such as `app-image` have dependencies on other targets to ensure that steps execute in the right order. Because 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 continuous integration or continuous delivery (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 this:

```
[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 1562106748
BUILD_TIME_RFC_3339 IS 2019-07-02T10:32:28Z
BUILD_TIME_UTC IS 20190702-223228
BUILD_ID IS 20190702-223228-ade3d65
```

The next step in the build process is the first quality-assurance step for our image. You should see a message like this:

```
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 that they follow best practices and identify common mistakes. As with other quality-assurance practices, you may choose to stop the image build pipeline when a linter reports a problem. Hadolint is one of several linters available for Dockerfiles. Because it parses the Dockerfile into an abstract syntax tree, it's 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 Dockerfile has been linted, the app-image target executes and builds the application image. The docker image build command should indicate success with output similar to the following:

```
Successfully built 79b61fb87b96
Successfully tagged dockerinaction/ch10:20190702-223619-ade3d65
Built App Image. BUILD_ID: 20190702-223619-ade3d65
```

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, 20190702-223619-ade3d65 in this case. The 20190702-223619-ade3d65 tag now identifies Docker image ID 79b61fb87b96 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.

The steps that follow will make use of the `BUILD_ID`. You can make the `BUILD_ID` easily accessible by copying it from the last line of the `app-image` build step output in your terminal and exporting it as a variable in your shell:

```
export BUILD_ID=20190702-223619-ade3d65
```

You can inspect the metadata that was added to the image by inspecting the labels via this command:

```
make inspect-image-labels BUILD_ID=20190702-223619-ade3d65
```

Or you can use the following if you exported the `BUILD_ID` tag:

```
make inspect-image-labels BUILD_ID=$BUILD_ID
```

This command uses `docker image inspect` to show the image's labels:

```
{
  "org.label-schema.build-date": "2019-07-02T10:36:19Z",
  "org.label-schema.name": "ch10",
  "org.label-schema.schema-version": "1.0rc1",
  "org.label-schema.vcs-ref": "ade3d65",
  "org.label-schema.version": "20190702-223619-ade3d65"
}
```

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 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 easily analyzed by tools. 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 that they run correctly. Docker images can be inspected to verify that traceability 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 (CST) from Google (<https://github.com/GoogleContainerTools/container-structure-test>). With this tool, authors can verify that 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 Chef Inspec or Serverspec. However, CST'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 that the application artifact has the proper permissions and that 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 18.9 (build 11.0.3+7)
    expectedError: ["OpenJDK Runtime Environment.*build 11\\.\\.\\."]

# 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.*build 11\\.\\.\\.*` regular expression. If you needed to ensure that the application runs 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 problems with bugs that are “invisible” because programs aren't executable, readable, or in the executable `PATH`. Execute the image tests against the image you built earlier with the following command:

```
make image-tests BUILD_ID=$BUILD_ID
```

This command should produce a successful result:

```

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 "11.0.3" 2019-04-16
OpenJDK Runtime Environment 18.9 (build 11.0.3+7)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.3+7, mixed mode)

=== RUN: Command Test: java version
--- PASS
stderr: openjdk version "11.0.3" 2019-04-16
OpenJDK Runtime Environment 18.9 (build 11.0.3+7)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.3+7, 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. Several image vulnerability scanning solutions are 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 file-system 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 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 the way 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. Several schemes for tagging images exist, and some are better for certain consumption patterns than others. The most important image-tagging features to understand are as follows:

- 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 used 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 it 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 CI system, regardless of source control branch
- The most recent image built by the CI 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. Because 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 and deployment schemes include the following:

- *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 development, stage, and production.
- *Semantic versioning*—Tag and publish images with a Major.Minor.Patch scheme that communicates the level of change in a release.

10.5.2 Continuous delivery with unique tags

The Unique Tags scheme, illustrated in figure 10.5, 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 by 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.

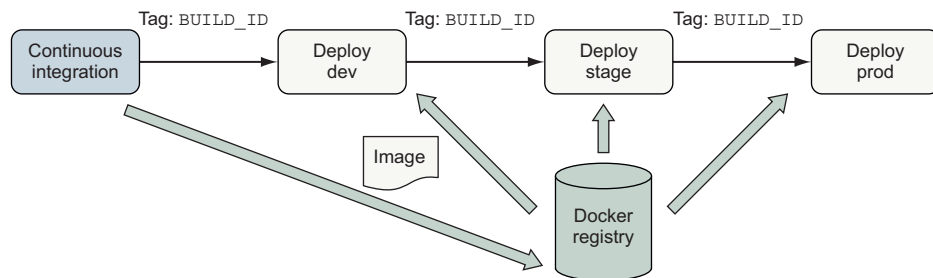


Figure 10.5 Continuous delivery with unique tags

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 use a latest or a dev tag. Because 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 such as `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 antipattern because the main deployment artifact is built multiple times and is usually not tested prior to production.

A better way to support deployment to multiple environments is to create two kinds of images:

- A generic, environment-agnostic application image
- A set of environment-specific configuration images, with each image containing the environment-specific configuration files for that environment

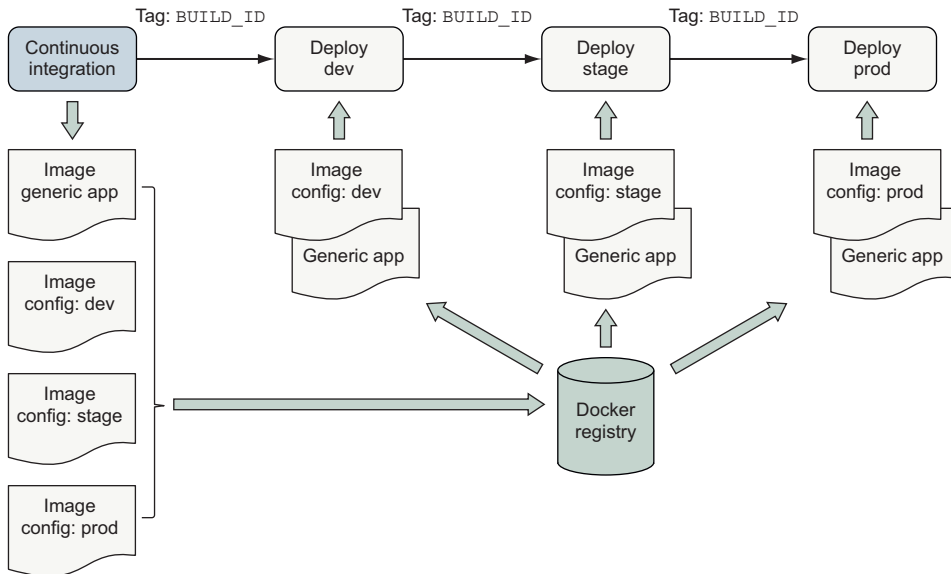


Figure 10.6 Configuration image per environment

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 by 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 supports environment-specific configuration of 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 following:

- 1 Major version when making incompatible API changes
- 2 Minor version when adding functionality in a backward-compatible manner
- 3 Patch version when making backward-compatible bug fixes

Semantic versioning helps both publishers and consumers manage expectations for the 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 a 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.

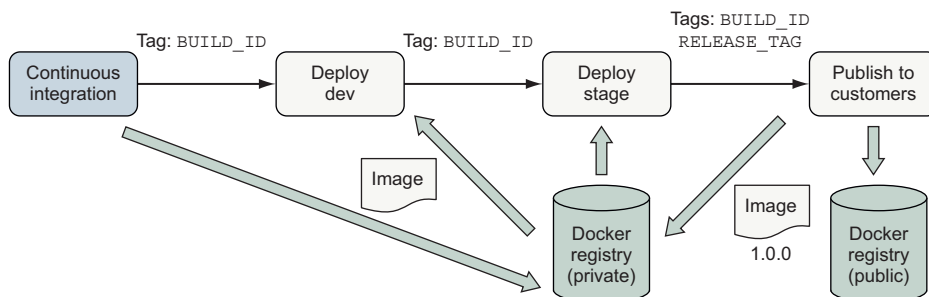


Figure 10.7 Tag and publish image release with semantic versioning

Suppose that after testing your image in dev and stage that you want to release your recent build of the example app as version 1.0.0 to your customers. You can use BUILD_ID to identify the image and tag it with 1.0.0:

```
make tag BUILD_ID=$BUILD_ID TAG=1.0.0
```

Tagging the image as version 1.0.0 signals that you are ready to maintain backward 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 publish only 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.

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 ensuring 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 easily be incorporated into image build pipelines.
- Codify the image build process by using a build tool such as make and use that process in local development and CI/CD processes.
- Several patterns exist for organizing Docker image definitions. These patterns provide trade-offs 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 by using styles ranging from continuous delivery in a private service deployment to publishing long-lived releases via semantic versioning to the public.

Higher-level abstractions and orchestration

This part focuses on managing systems of components by using containers. Most valuable systems have of two or more components. Managing systems with a small number of components is manageable without much automation. But achieving consistency and repeatability is difficult without automation as the system grows.

Managing modern service architectures is complex and requires automated tooling. This part dives into higher-level abstractions such as service, environment, and configuration. You will learn how to use Docker to provide that automated tooling.

11

Services with Docker and Compose

This chapter covers

- Understanding services and how they relate to containers
- Basic service administration with Docker Swarm
- Building declarative environments with Docker Compose and YAML
- Iterating projects with Compose and the `deploy` command
- Scaling services and cleaning up

Today most of the software we run is designed to interact with other programs, not human users. The resulting webs of interdependent processes serve a collective purpose such as processing payments, running games, facilitating global communications, or delivering content. When you look closely at that web, you'll find individual running processes that might be running in containers. Those processes are allocated memory and given time on the CPU. They are bound to a network and listen for requests from other programs on specific ports. Their network interface and port are registered with a naming system so they are discoverable on that network. But as you expand your view and examine more processes, you'll notice that most of them share common characteristics and goals.

Any processes, functionality, or data that must be discoverable and available over a network is called a *service*. That name, *service*, is an abstraction. By encoding those goals into an abstract term, we simplify how we talk about the things that use this pattern. When we talk about a specific service, we do not need to expressly state that the name should be discoverable via DNS or the environment-appropriate service-discovery mechanism. We do not need to state that the service should be running when a client needs to use it. Those expectations are already communicated by common understanding of the service abstraction. The abstraction lets us focus on the things that make any specific service special.

We can reflect those same benefits in our tooling. Docker already does this for containers. Containers were described in chapter 6 as processes that were started using specific Linux namespaces, with specific filesystem views, and resource allotments. We don't have to describe those specifics each time we talk about a container, nor do we have to do the work of creating those namespaces ourselves. Docker does this for us. Docker provides tooling for other abstractions as well, including *service*.

This chapter introduces the basic tooling that Docker provides for working with services in swarm mode. It covers the service life cycle, the role of an orchestrator, and how to interact with that orchestrator to deploy and manage services on your machine. You will use the tools described in this chapter throughout the remainder of the book. The same concepts, problems, and fundamental tooling are provided by all of the container orchestration systems including Kubernetes. The material that follows will be helpful in understanding whichever orchestrators you use in your daily job.

11.1 A service “Hello World!”

Getting started with services is as easy as getting started with containers. In this case, you can start a “Hello World!” web server locally by running these two commands:

```
docker swarm init
docker service create \
  --publish 8080:80 \
  --name hello-world \
  dockerinaction/ch11_service_hw:v1
```

← Enables the service abstraction
← Starts the server on localhost:8080

Unlike containers, Docker services are available only when Docker is running in swarm mode. Initializing swarm mode starts an internal database as well as a long-running loop in the Docker Engine that performs service orchestration; see figure 11.1. Swarm provides several other features that will be covered in the rest of the book.

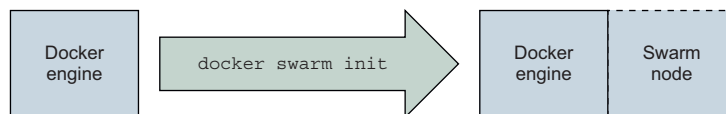


Figure 11.1 Initializing a swarm node

Running the `init` command in the preceding code enables the service management subcommands.

The `service create` subcommand defines a service named `hello-world` that should be available on port 8080, and uses the image, `dockerinaction/ch11_service_hw:v1` as shown in figure 11.2.

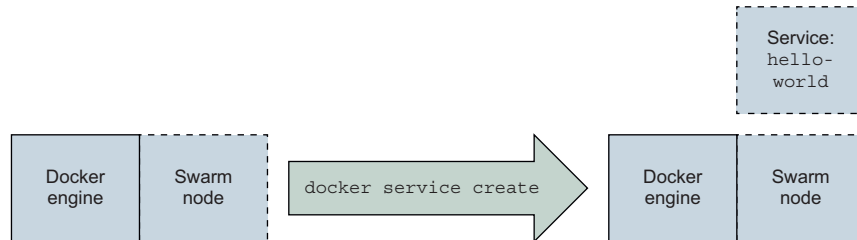


Figure 11.2 Creating your first service

After you run these two commands, you should see a progress bar describing the state of the service. Once the progress is complete, it will be labeled `Running`, and the command will exit. At this point, you should be able to open `http://localhost:8080` and see a nice little message, `Hello, World! --ServiceV1`. It will also display the task ID (container ID) that served the request.

A *task* is a swarm concept that represents a unit of work. Each task has one associated container. Although there could be other types of tasks that don’t use containers, those are not the subject of this chapter. Swarm works only with tasks. The underlying components transform task definitions into containers. This book is not concerned with Docker internals. For our purposes, you can consider the terms *task* and *container* roughly interchangeable.

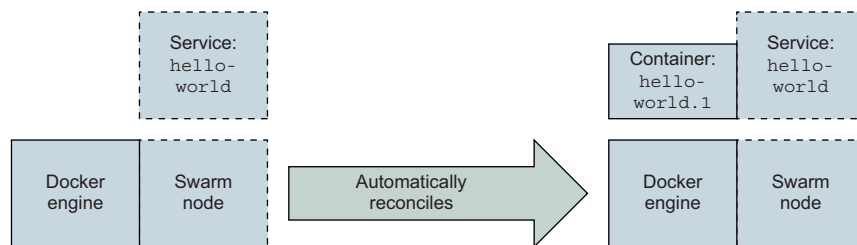


Figure 11.3 The swarm node automatically creates a container to run the service software.

This should feel just like running a similar container example. Rather than focusing on how services and containers are similar, it is more enlightening to focus on how they differ. First, recognize that service workloads are implemented with containers. With the service running, run `docker container ps` to discover that there is a container

running with a name, such as `hello-world.1.pqamgg6b15eh6p8j4fj503kur`. There is nothing special about this container. Inspecting the container does not yield any particularly interesting results. You might notice a few Swarm-specific labels, but that is it. However, if you remove the container, something interesting happens. The remainder of this section describes the higher-level properties, the service life cycle, and how Swarm uses those to perform small automated miracles such as service resurrection.

11.1.1 Automated resurrection and replication

Bringing a service back to life is something most developers and operators are a bit too familiar with. For those people, manually killing the sole process running a service might feel like tempting fate. Like kicking an early model robot with artificial intelligence—it just feels like a bigger risk than we’d prefer to take. But with the right tools (and validation of those tools), we can be comfortable knowing that taking the risk probably won’t result in any Armageddon-like business scenario.

If you remove the sole container that is powering the `hello-world` service (from the previous section), the container will be stopped and removed, but after a few moments it will be back. Or at least another similarly configured container will be there in its place. Try this yourself: find the ID of the container running the server (using `docker ps`); use `docker container rm -f` to remove it; then issue a few `docker container ps` commands to verify that it has been removed, and watch a replacement appear. Next, dig into this resurrection by using the `service` subcommands shown in figure 11.4.

First, run `docker service ls` to list the running services. The list will include the `hello-world` service and show that one replica is running, indicated by `1/1` in the `REPLICAS` column of the command output. This is evidenced by the container that

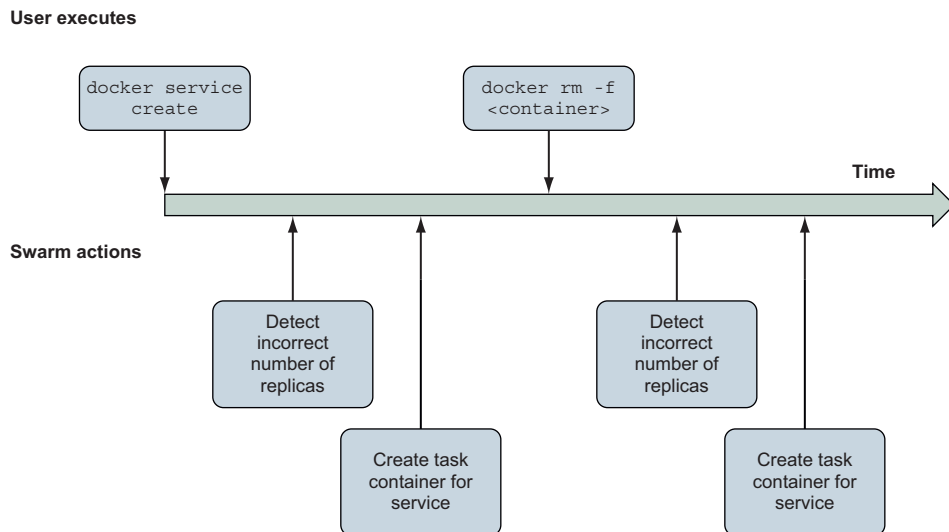


Figure 11.4 Timeline of Swarm reactions to changes in service specification and state

came back to life. Next, run `docker service ps hello-world` to list the containers associated with a specific service (`hello-world` in this case). The list includes two entries. The list will show the first entry with a Desired State of “Running,” and a Current State of “Running x minutes ago.” The second entry will be listed as Shutdown and Failed, respectively. These two columns hint at critical ideas, so let’s unpack them now. Consider this excerpt from `docker service ps` output:

NAME	DESIRED STATE	CURRENT STATE
hello-world.1	Running	Running less than a second ago
_ hello-world.1	Shutdown	Failed 16 seconds ago

Autonomous orchestrators—such as the Swarm components in Docker—track two things: desired state and the current state. The *desired state* is what the user wants the system to be doing, or what it is supposed to be doing. The *current state* describes what the system is actually doing. Orchestrators track these two descriptions of state and reconcile the two by changing the system.

In this example, the swarm orchestrator notices that the container for the `hello-world` service has failed. It doesn’t matter that you killed the process in this case. All the orchestrator knows is that the process has failed and that the desired state of the service is `Running`. Swarm knows how to make a process run: start a container for that service. And that is exactly what it does.

Using higher-level abstractions with autonomous orchestrators is more like a partnership than using a tool. Orchestrators remember how a system should be operating and manipulate it without being asked to do so by a user. So, in order to use orchestrators effectively, you need to understand how to describe systems and their operation. You can learn quite a bit about managing services by inspecting the `hello-world` service.

When you run `docker service inspect hello-world`, Docker will output the current desired state definition for the service. The resulting JSON document includes the following:

- Name of the service
- Service ID
- Versioning and timestamps
- A template for the container workloads
- A replication mode
- Rollout parameters
- Similar rollback parameters
- A description of the service endpoint

The first few items identify the service and its change history. It starts to get interesting with replication mode, and rollout and rollback parameters. Recall our definition of a service: *any processes, functionality, or data that must be discoverable and available over a network*. The difficulty of running a service is, by definition, more about managing

availability of something on a network. So it shouldn't be too surprising that a service definition is predominantly about how to run replicas, manage changes to the software, and route requests to the service endpoint to that software. These are the higher-level properties uniquely associated with the service abstraction. Let's examine these in more depth.

A replication mode tells Swarm how to run replicas of the workload. Today there are two modes: replicated and global. A service in *replicated mode* will create and maintain a fixed number of replicas. This is the default mode, and you can experiment with that now by using the `docker service scale` command. Tell your swarm to run three replicas of the `hello-world` service by running the following:

```
docker service scale hello-world=3
```

After the containers start up, you can verify the work by using either `docker container ps` or `docker service ps hello-world` to list the individual containers (there should be three now). You should notice that the container-naming convention encodes the service replica number. For example, you should see a container with a name such as `hello-world.3.pqamgg6b15eh6p8j4fj503kur`. If you scale the service back down, you'll also notice that higher-numbered containers are removed first. So if you run `docker service scale hello-world=2`, the container named `hello-world.3.pqamgg6b15eh6p8j4fj503kur` will be removed. But `hello-world.1` and `hello-world.2` will remain untouched.

The second mode, *global*, tells Docker to run one replica on each node in the swarm cluster. This mode is more difficult to experiment with at this point because you're running only a single-node cluster (unless you're skipping ahead). Services in global mode are useful for maintaining a common set of infrastructure services that must be available locally on each node in a cluster.

It isn't important that you have a deep understanding of how replication works in a docker swarm at this point. But it is critical to understand that maintaining high service availability requires running replicas of that service software. Using replicas allows you to replace or change portions of the replica set, or survive failures without impacting the service availability. When you have replicas of software, certain operational stories become more complicated. For example, upgrading software is not as simple as stopping the old version and starting the new one. A few properties impact change management and deployment processes.

11.1.2 Automated rollout

Rolling out a new version of replicated service software is not particularly complicated, but you should consider a few important parameters when automating the process. You have to describe the characteristics of the deployment. Those include order, batch sizes, and delays. This is done by specifying constraints and parameters to the swarm orchestrator that it will respect during the deployment process. An

illustration of a Docker swarm’s actions when deploying an update is shown in figure 11.5.

User executes

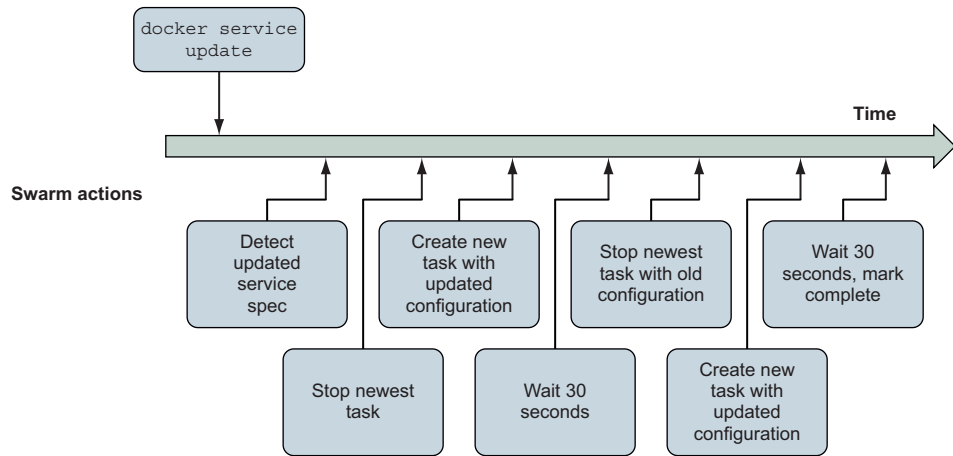


Figure 11.5 Timeline of automated deployment of an updated service definition to a Docker swarm

Consider this command to update the `hello-world` service created earlier in this chapter:

```
docker service update \
  --image dockerinaction/ch11_service_hw:v2 \
  --update-order stop-first \
  --update-parallelism 1 \
  --update-delay 30s \
  hello-world
```

← The new image

← The name of the service to update

This command tells Docker to change the `hello-world` service to use the image tagged `v2`. It further qualifies the deployment characteristics: only one replica should be updated at a time; wait for 30 seconds between updating each batch of replicas; and each replica should be stopped before its replacement is started. When you execute the command, Docker will report the deployment progress for each replica and the change overall. If everything goes well, it will close by stating that the Service converged. That is a particularly robotic way of telling the user that the command was successful. *Converged* is a technical way to say that current state of the service is the same as the desired state described by the command.

After you’ve updated the service, you should be able to reload the application in your browser and see that it has been signed `--ServiceV2`. This is not a particularly interesting example because everything just works. Things aren’t so simple in the real world. We use parallelism to balance the time it takes to update our service with

protection for our users from failed transitions. We introduce delay between update batches to allow new service instances to become stable before starting (and to make sure that the underlying platform remains stable). In reality, 30 seconds might not be enough time. It depends on the application.

Docker and its Swarm components are application agnostic. They could never anticipate the behavior of all the applications that might be deployed or how they might fail at runtime. Instead, the Docker command line and API provide ways for the user to specify the methods for discovering problems and validating success, and the behavior for managing failed deployments.

11.1.3 *Service health and rollback*

A successful partnership with an orchestrator means clearly communicating the expected requirements and behavior of the workload you're asking it to orchestrate. Although Docker can be sure that a stopped container is unhealthy, there is no universal and accurate definition of service health. This limits the safe assumptions that any orchestrator can make about service health, and deployment success or failure. Determining workload health or startup behavior expectations might be more nuanced than you'd expect unless you're an experienced service owner.

Before getting too far into the details, start with a simple example. In the most obvious case of service health problems, the new containers might fail to start:

```
docker service update \  
  --image dockerinaction/ch11_service_hw:start-failure \  
  hello-world
```

When you execute this command, Docker will start a deployment on the `hello-world` service. Unlike the others, this change will fail. By default, Docker will pause the deployment after the first replica fails to start. The command will exit, but it will continue to attempt to start that container. If you run `docker service ps hello-world`, you will see that two of the replicas remain on the old version of the service, and the other replica keeps cycling through starting and failed states.

In this scenario, the deployment cannot proceed. The new version will never start. And as a result, the service is stuck at reduced capacity and will require human intervention to repair. Fix the immediate issue by using the `--rollback` flag on the `update` command:

```
docker service update \  
  --rollback \  
  hello-world
```

This command will ask Docker to reconcile the current state with the previous desired state. Docker will determine that it needs to change only one of the three replicas (the one that failed to start). It knows that the service is currently in a paused deployment and that only one of the replicas was transitioned to the current desired state. The other replicas will continue operating.

Knowing that rollback is appropriate for this service (no risk of incompatible application state changes), you can automate rollback when the deployment fails. Use the `--update-failure-action` flag to tell Swarm that failed deployments should roll back. But you should also explicitly tell Swarm which conditions should be considered a failure.

Suppose you’re running 100 replicas of a service and those are to be run on a large cluster of machines. Chances are that a certain set of conditions might prevent a replica from starting correctly. In that case, it might be appropriate to continue a deployment as long as a critical threshold of replicas are operational. In this next deployment, tell Swarm to tolerate start failures as long as one-third of the fleet is operational for illustrative purposes. You’ll use the `--update-max-failure-ratio` flag and specify 0.6:

```
docker service update \
  --update-failure-action rollback \
  --update-max-failure-ratio 0.6 \
  --image dockerinaction/ch11_service_hw:start-failure \
  hello-world
```

When you run this example, you’ll watch Docker attempt to deploy updated replicas one at a time. The first one will retry a few times before the delay expires and the next replica deployment starts. Immediately after the second replica fails, the whole deployment will be marked as failed and a rollback will be initiated. The output will look something like this:

```
hello-world
overall progress: rolling back update: 2 out of 3 tasks
1/3: running [> ]
2/3: starting [====> ]
3/3: running [> ]
rollback: update rolled back due to failure or early termination of ↴
task tdpv6fud16e4nbg3tx2jpkah
service rolled back: rollback completed
```

After the command finishes, the service will be in the same state as it was prior to the update. You can verify this by running `docker service ps hello-world`. Note that one replica was untouched, while the other two were started much more recently and at nearly the same time. At this point, all the replicas will be running from the `dockerinaction/ch11_service_hw:v2` image.

As we mentioned earlier, *running* is not the same thing as service health. There are plenty of ways that a program might be running, but not correctly. Like other orchestrators, Docker models health separately from process status and provides a few configuration points for specifying how to determine service health.

Docker is application agnostic. Rather than making assumptions about how to determine whether a specific task is healthy, it lets you specify a health check command. That command will be executed from within the container for each service

replica. Docker will execute that command periodically on the specified schedule from within the task containers themselves. This behaves just like issuing a `docker exec` command. Health checks and related parameters can be specified at service creation time, changed or set on service updates, or even specified as image metadata by using the `HEALTHCHECK` Dockerfile directive.

Each service replica container (task) will inherit the definition of health and health check configuration from the service. When you want to manually inspect the health of a specific task in Docker, you need to inspect the container, not the service itself. Both the `v1` and `v2` versions of the service you've been using have health checks specified in the image. The images include a small custom program called `httping`. It verifies that the service is responsive on localhost and that requests to `/` result in an HTTP 200 response code. The Dockerfiles include the following directive:

```
HEALTHCHECK --interval=10s CMD ["/bin/httping"]
```

Run `docker container ps` to see that each `hello-world` replica is marked as healthy in the status column. You can further inspect the configuration by inspecting either the image or container.

Including some default health check configuration is a good idea for images containing service software, but it is not always available. Consider `dockerinaction/ch11_service_hw:no-health`. This image is essentially the same as the `v1` and `v2` images, but it does not include any health check metadata. Update the `hello-world` service to use this version now:

```
docker service update \
  --update-failure-action rollback \
  --image dockerinaction/ch11_service_hw:no-health \
  hello-world
```

After deploying this version, you should be able to run `docker container ps` again and see that the containers are no longer marked healthy. Docker cannot determine whether the service is healthy without health check metadata. It knows only whether the software is running. Next, update the service to add the health check metadata from the command line:

```
docker service update \
  --health-cmd /bin/httping \
  --health-interval 10s \
  hello-world
```

Health monitoring requires continuous evaluation. The interval specified here tells Docker how often to check the health of each service instance. When you run this command, you can verify that the service replicas are marked as healthy once again.

Today you can also tell Docker how many times to retry a health check before reporting an unhealthy status, a startup delay, and a time-out for running a health check command. These parameters help tune the behavior to accommodate most

cases. Sometimes a default or current health check for a service is inappropriate for the way you are using it. In those cases, you can create or update a service with health checks disabled by using the `--no-healthcheck` flag.

During a deployment, a new container might not start. Or it might start but not quite work correctly (be unhealthy). But how do you define service health? Timing issues might blur those definitions. How long should you wait for an instance to become healthy? Some but not all of the service replicas might fail or be otherwise unhealthy. How many or what portion of replica deployment failure can your service tolerate? Once you can answer these questions, you can tell Docker about those thresholds and tune the signal of health from your application to the orchestrator. In the meantime, you're free to delete the `hello-world` service:

```
docker service rm hello-world
```

Setting all of these parameters when you're managing a service from the command line is a mess. When you're managing several services, it can get even worse. In the next section, you'll learn how to use the declarative tooling provided by Docker and make things more manageable.

11.2 Declarative service environments with Compose V3

Until this point in the book, you've used the Docker command line to individually create, change, remove, or interact with containers, images, networks, and volumes. We say that systems like this follow an *imperative pattern*. Imperative-style tools carry out commands issued by a user. The commands might retrieve specific information or describe a specific change. Programming languages and command-line tools follow an imperative pattern.

The benefit of imperative tools is that they give a user the ability to use primitive commands to describe much more complex flows and systems. But the commands must be followed exactly, in precise order, so that they have exclusive control of the working state. If another user or process is changing the state of the system at the same time, the two users might make undetectable conflicting changes.

Imperative systems have a few problems. It is burdensome for users to carefully plan and sequence all of the commands required to achieve a goal. Those plans are often challenging to audit or test. Timing or shared state issues are incredibly difficult to discover and test before deployment. And as most programmers will tell you, small or innocuous mistakes might radically change the outcome.

Imagine you're responsible for building and maintaining a system that comprises 10, 100, or 1000 logical services, each with its own state, network attachment, and resource requirements. Now imagine that you are using raw containers to manage replicas of those services. It would be much more difficult to manage the raw containers than Docker services.

Docker services are declarative abstractions, as illustrated in figure 11.6. When we create a service, we *declare* that we want a certain number of replicas of that service,

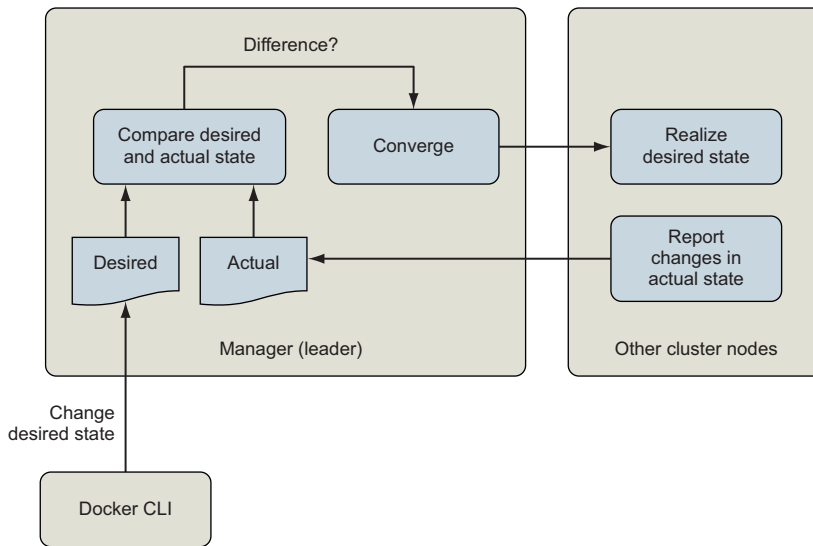


Figure 11.6 Declarative processing loop

and Docker takes care of the individual commands required to maintain them. Declarative tools enable users to describe the new state of a system, rather than the steps required to change from the current state to the new state.

The Swarm orchestration system is a state reconciliation loop that continuously compares the declared state of the system that the user desires with the current state of the system. When it detects a difference, it uses a simple set of rules to change the system so that it matches the desired state.

Declarative tools close the issues with the imperative pattern. Declarative interfaces simplify the system by imposing constraints on how the system operates. This enables the declared policy to be implemented by one or a few well-tested engines that can converge the system to the declared state. They are easier to write, audit, and comprehend. Declarative statements or documents are wonderfully paired with version-control systems because they allow us to effectively version-control the state of the systems that they describe.

Imperative and declarative tools are not in competition. You will almost never use one without the other. For example, when we create or update a Docker service, we are using an imperative tool to describe a change to the system. Issuing the `docker service create` command is imperative, but it creates a declarative abstraction. It rolls up a whole world of lower-level management commands for creating and removing containers, evaluating health, pulling images, managing service discovery, and network routing.

As you build more complex systems of services, volumes, networks, and configuration, the number of imperative commands required to achieve your goals will become

a new burden. When that happens, it is time to adopt a higher-level declarative abstraction. In this case, that abstraction is a stack or complete environment as described with Docker Compose.

When you need to model a whole environment of services, you should use a Docker stack. A *stack* describes collections of services, volumes, networks, and other configuration abstractions. The `docker` command line provides imperative commands for deploying, removing, and inspecting stacks. Stacks are created from a declarative description of an entire environment. These environments are described using the Docker Compose V3 file format. A Compose file that describes an environment with the `hello-world` service from earlier might look like the following:

```
version: "3.7"
services:
  hello-world:
    image: dockerinaction/ch11_service_hw:v1
    ports:
      - 8080:80
    deploy:
      replicas: 3
```

Compose files use Yet Another Markup Language (YAML). Not everyone is familiar with YAML, and that can be a hurdle to adopting several tools in this generation of infrastructure and workload management tooling. The good news is that people rarely use exotic YAML features. Most people stick to the basics.

This chapter is not an exhaustive survey of Compose or the properties you will use to manage services. The official Docker documentation should serve that purpose. Expect to find every command-line feature mirrored in Compose. The next section is a brief primer on YAML and Compose files.

11.2.1 A YAML primer

YAML is used to describe structured documents, which are made up of structures, lists, maps, and scalar values. Those features are defined as continuous blocks, where substructures are defined with nested block definitions in a style that will feel familiar to most high-level language programmers.

The default scope of a YAML document is a single file or stream. YAML provides a mechanism to specify multiple documents in the same file, but Docker will use only the first document it encounters in a Compose file. The standard filename for Compose files is `docker-compose.yml`.

Comment support is one of the most popular reasons to adopt YAML instead of JSON today. A YAML document can include a comment at the end of any line. Comments are marked by a space followed by a hash sign (`#`). Any characters that follow until the end of the line are ignored by the parser. Empty lines between elements have no impact on the document structure.

YAML uses three types of data and two styles of describing that data, block and flow. *Flow* collections are specified similarly to collection literals in JavaScript and other languages. For example, the following is a list of strings in the flow style:

```
["PersonA", "PersonB"]
```

The *block* style is more common and will be used in this primer except where noted. The three types of data are maps, lists, and scalar values.

Maps are defined by a set of unique properties in the form of key/value pairs that are delimited by a colon and space (:). Whereas property names must be string values, property values can be any of the YAML data types except documents. A single structure cannot have multiple definitions for the same property. Consider this block style example:

```
image: "alpine"
command: echo hello world
```

This document contains a single map with two properties: `image` and `command`. The `image` property has a scalar string value, "alpine". The `command` property has a scalar string value, `echo hello world`. A scalar is a single value. The preceding example demonstrates two of the three flow scalar styles.

The value for `image` in the preceding example is specified in *double-quote style*, which is capable of expressing arbitrary strings, by using `\` escape sequences. Most programmers are familiar with this string style.

The value for the `command` is written in *plain style*. The plain (unquoted) style has no identifying indicators and provides no form of escaping. It is therefore the most readable, most limited, and most context-sensitive style. A bunch of rules are used for plain style scalars. Plain scalars

- Must not be empty
- Must not contain leading or trailing whitespace characters
- Must not begin with an indicator character (for example, `-` or `:`) in places where doing so would cause an ambiguity
- Must never contain character combinations using a colon (`:`) and hash sign (`#`)

Lists (or block sequences) are series of nodes in which each element is denoted by a leading hyphen (`-`) indicator. For example:

```
- item 1
- item 2
- item 3
- # an empty item
- item 4
```

Finally, YAML uses indentation to indicate content scope. Scope determines which block each element belongs to. There are a few rules:

- Only spaces can be used for indentation.
- The amount of indentation does not matter as long as
 - All peer elements (in the same scope) have the same amount of indentation.
 - Any child elements are further indented.

These documents are equivalent:

```
top-level:
  second-level:          # three spaces
    third-level:        # two more spaces
      - "list item"      # single additional indent on items in this list
    another-third-level: # a third-level peer with the same two spaces
      fourth-level: "string scalar" # 6 more spaces
  another-second-level: # a 2nd level peer with three spaces
    - a list item # list items in this scope have
                  # 15 total leading spaces
    - a peer item # A peer list item with a gap in the list
---
# every scope level adds exactly 1 space
top-level:
  second-level:
    third-level:
      - "list item"
    another-third-level:
      fourth-level: "string scalar"
  another-second-level:
    - a list item
    - a peer item
```

The full YAML 1.2 specification is available at <http://yaml.org/spec/1.2/2009-07-21/spec.html> and is quite readable. Armed with a basic understanding of YAML, you're ready to jump into basic environment modeling with Compose.

11.2.2 Collections of services with Compose V3

Compose files describe every first-class Docker resource type: services, volumes, networks, secrets, and configs. Consider a collection of three services: a PostgreSQL database, a MariaDB database, and a web administrative interface for managing those databases. You might represent those services with this Compose file:

```
version: "3.7"
services:
  postgres:
    image: dockerinaction/postgres:11-alpine
    environment:
      POSTGRES_PASSWORD: example
```

```

mariadb:
  image: dockerinaction/mariadb:10-bionic
  environment:
    MYSQL_ROOT_PASSWORD: example

adminer:
  image: dockerinaction/adminer:4
  ports:
    - 8080:8080

```

Keep in mind that this document is in YAML, and all of the properties with the same indentation belong to the same map. This Compose file has two top-level properties: `version` and `services`. The `version` property tells the Compose interpreter which fields and structure to anticipate. The `services` property is a map of service names to service definitions. Each service definition is a map of properties.

In this case, the `services` map has three entries with keys: `postgres`, `mariadb`, and `adminer`. Each of those entries defines a service by using a small set of service properties such as `image`, `environment`, or `ports`. Declarative documents make it simple to concretely specify a full service definition. Doing so will reduce implicit dependencies on default values and reduce the educational overhead for your fellow team members. Omitting a property will use the default values (just as when using the command-line interface). These services each define the container `image`. The `postgres` and `mariadb` services specify environment variables. The `adminer` service uses `ports` to route requests to port 8080 on the host to port 8080 in the service container.

CREATING AND UPDATING A STACK

Now use this Compose file to create a stack. Remember, a Docker stack is a named collection of services, volumes, networks, secrets, and configs. The `docker stack` subcommands manage stacks. Create a new file called `databases.yml` in an empty directory. Edit the file and add the preceding Compose file contents. Create a new stack and deploy the services it describes by using the following command:

```
docker stack deploy -c databases.yml my-databases
```

When you run this command, Docker will display output like this:

```

Creating network my-databases_default
Creating service my-databases_postgres
Creating service my-databases_mariadb
Creating service my-databases_adminer

```

At this point, you can test the services by using your browser to navigate to `http://localhost:8080`. Services are discoverable by their names within a Docker network. You'll need to keep that in mind when you use the `adminer` interface to connect to your `postgres` and `mariadb` services. The `docker stack deploy` subcommand is used to create and update stacks. It always takes the Compose file that represents the desired state of the stack. Whenever the Compose file that you use differs from the definitions used

in the current stack, Docker will determine how the two differ and make the appropriate changes.

You can try this yourself. Tell Docker to create three replicas of the adminer service. Specify the `replicas` property under the `deploy` property of the adminer service. Your `databases.yml` file should look like this:

```
version: "3.7"
services:
  postgres:
    image: dockerinaction/postgres:11-alpine
    environment:
      POSTGRES_PASSWORD: example

  mariadb:
    image: dockerinaction/mariadb:10-bionic
    environment:
      MYSQL_ROOT_PASSWORD: example

  adminer:
    image: dockerinaction/adminer:4
    ports:
      - 8080:8080
    deploy:
      replicas: 3
```

After you've updated your Compose file, repeat the earlier docker stack deploy command:

```
docker stack deploy -c databases.yml my-databases
```

This time, the command will display a message noting that services are being updated, not created:

```
Updating service my-databases_mariadb (id: lpvun5ncnleb6mhqj8bbphsf6)
Updating service my-databases_adminer (id: i2gatqudz9pdsaoux7auaiicm)
Updating service my-databases_postgres (id: eejvkaqgbb135glatt977m65a)
```

The message appears to indicate that all of the services are being changed. But that is not the case. You can use `docker stack ps` to list all of the tasks and their ages:

```
docker stack ps \
  --format '{{.Name}}\t{{.CurrentState}}' \
  my-databases
```

Specifies which
stack to list

This command should filter for the interesting columns and report something like the following:

```
my-databases_mariadb.1 Running 3 minutes ago
my-databases_postgres.1 Running 3 minutes ago
my-databases_adminer.1 Running 3 minutes ago
my-databases_adminer.2 Running about a minute ago
my-databases_adminer.3 Running about a minute ago
```

This view hints that none of the original service containers were touched during the new deployment. The only new tasks are those additional replicas of the `adminer` service that were required to bring the system into the state described by the current version of the `databases.yml` file. You should note that the `adminer` service really doesn't work well when multiple replicas are running. We're using it here for illustrative purposes only.

SCALING DOWN AND REMOVING SERVICES

When you're using Docker and Compose, you never need to tear down or otherwise remove a whole stack when you're redeploying or making changes. Let Docker figure it out and handle the change for you. Only one case is tricky to deal with when you're working with declarative representations such as `Compose: deletion`.

Docker will delete service replicas automatically when you scale down. It always chooses the highest numbered replicas for removal. For example, if you have three replicas of the `adminer` service running, they will be named `my-databases_adminer.1`, `my-databases_adminer.2`, and `my-databases_adminer.3`. If you scale down to two replicas, Docker will delete the replica named `my-databases_adminer.3`. Things get weird when you try to delete a whole service.

Edit the `databases.yml` file to delete the `mariadb` service definition and set the `adminer` service to two replicas. The file should look like this:

```
version: "3.7"
services:
  postgres:
    image: dockerinaction/postgres:11-alpine
    environment:
      POSTGRES_PASSWORD: example

  adminer:
    image: dockerinaction/adminer:4
    ports:
      - 8080:8080
    deploy:
      replicas: 2
```

Now when you run `docker stack deploy -c databases.yml my-databases`, the command will generate output like this:

```
Updating service my-databases_postgres (id: lpvun5ncnleb6mhqj8bbphsf6)
Updating service my-databases_adminer (id: i2gatqudz9pdsaoux7auaiicm)
```

The Compose file you provided to the `stack deploy` command did not have any reference to the `mariadb` service, so Docker did not make any changes to that service. When you list the tasks in your stack again, you'll notice that the `mariadb` service is still running:

```
docker stack ps \
  --format '{{.Name}}\t{{.CurrentState}}' \
  my-databases
```

← Specifies which stack to list

Running this command results in this output:

```
my-databases_mariadb.1 Running 7 minutes ago
my-databases_postgres.1 Running 7 minutes ago
my-databases_adminer.1 Running 7 minutes ago
my-databases_adminer.2 Running 5 minutes ago
```

You can see that the third replica of the adminer service has been removed, but the mariadb service is still running. This works as intended. Docker stacks can be created and managed with several Compose files. But doing so creates several error opportunities and is not recommended. There are two ways to delete services or other objects.

You can manually remove a service by using `docker service remove`. This works, but does not get any of the benefits of working with declarative representations. If this change is made manually and not reflected in your Compose files, the next `docker stack deploy` operation will create the service again. The cleanest way to remove services in a stack is by removing the service definition from your Compose file and then executing `docker stack deploy` with the `--prune` flag. Without further altering your `databases.yml` file, run the following:

```
docker stack deploy \
  -c databases.yml \
  --prune \
  my-databases
```

This command will report that the services described by `databases.yml` have been updated, but will also report that `my-databases_mariadb` has been removed. When you list the tasks again, you will see that this is the case:

```
my-databases_postgres.1 Running 8 minutes ago
my-databases_adminer.1 Running 8 minutes ago
my-databases_adminer.2 Running 6 minutes ago
```

The `--prune` flag will clean up any resource in the stack that isn't explicitly referenced in the Compose file used for the deploy operation. For that reason, it is important to keep a Compose file that represents the entire environment. Otherwise, you might accidentally delete absent services or volumes, networks, secrets, and configs.

11.3 Stateful services and preserving data

The Docker stack that you've been working with includes a database service. In chapter 4, you learned how to use volumes to separate a container life cycle from the life cycle for the data it uses. This is especially important for databases. As specified earlier, the stack will create a new volume for the `postgres` service each time the container is replaced (for whatever reason), and a new volume will be created for each replica. This would cause problems in a real-world system for which stored data is an important part of service identity.

The best way to begin addressing this problem is by modeling volumes in Compose. Compose files use another top-level property named `volumes`. Like `services`,

volumes is a map of volume definitions; the key is the name of the volume, and the value is a structure defining the volume properties. You do not need to specify values for every volume property. Docker will use defaults for omitted property values. The top-level property defines the volumes that can be used by services within the file. Using a volume in a service requires that you specify the dependency from the service that needs it.

A Compose service definition can include a volumes property. That property is a list of short or long volume specifications. Those correspond to the volumes and mount syntax supported by the Docker command line, respectively. We'll use the long form to enhance databases.yml and add a volume to store the postgres data:

```
version: "3.7"
volumes:
  pgdata: # empty definition uses volume defaults
services:
  postgres:
    image: dockerinaction/postgres:11-alpine
    volumes:
      - type: volume
        source: pgdata # The named volume above
        target: /var/lib/postgresql/data
    environment:
      POSTGRES_PASSWORD: example
  adminer:
    image: dockerinaction/adminer:4
    ports:
      - 8080:8080
    deploy:
      replicas: 1 # Scale down to 1 replica so you can test
```

In this example, the file defines a volume named pgdata, and the postgres service mounts that volume at /var/lib/postgresql/data. That location is where the PostgreSQL software will store any database schema or data. Deploy the stack and inspect the results:

```
docker stack deploy \
  -c databases.yml \
  --prune \
  my-databases
```

Run `docker volume ls` after the changes have been applied to verify that the operation was successful:

```
DRIVER          VOLUME NAME
local          my-databases_pgdata
```

The name of the stack prefixes any resources created for it such as services or volumes. In this case, Docker created the volume you named pgdata with the prefix my-databases. You could spend time inspecting the service configuration or container further, but it is more interesting to perform a functional test.

Open `http://localhost:8080` and use the adminer interface to manage the postgres database. Select the postgresql driver, use postgres as the hostname, postgres as the username, and example as the password. After you've logged in, create a few tables or insert some data. When you're done, remove the postgres service:

```
docker service remove my-databases_postgres
```

Then restore the service by using the Compose file:

```
docker stack deploy \
  -c databases.yml \
  --prune \
  my-databases
```

Because the data is stored in a volume, Docker is able to attach the new database replica to the original pg-data volume. If the data wasn't stored in the volume and existed in only the original replica, it would have been lost when the service was removed. Log into the database by using the adminer interface again (remember that the username is postgres, and password is example as specified in the Compose file). Examine the database and look for the changes you made. If you've followed these steps correctly, those changes and the data will be available.

This example uses two levels of naming indirection that make it a bit complicated to follow. Your browser is pointed at localhost, which loads the adminer service, but you're telling adminer to access the database by using the postgres hostname. The next section explains that indirection, and describes the built-in Docker network enhancements and how to use them with services.

11.4 Load balancing, service discovery, and networks with Compose

In accessing the adminer interface from a web browser, you're accessing the published port on the adminer service. Port publishing for a service is different from publishing a port on a container. Whereas containers directly map the port on the host interface to an interface for a specific container, services might be made up of many replica containers.

Container network DNS faces a similar challenge. When you're resolving the network address of a named container, you'll get the address of that container. But services might have replicas.

Docker accommodates services by creating virtual IP (VIP) addresses and balancing requests for a specific service between all of the associated replicas. When a program attached to a Docker network looks up the name of another service attached to that network, Docker's built-in DNS resolver will respond with the virtual IP for that service's location on the network. Figure 11.7 illustrates the logical flow of service name and IP resolution.

Likewise, when a request comes into the host interface for a published port or from an internal service, it will be routed to the target service's virtual IP address.

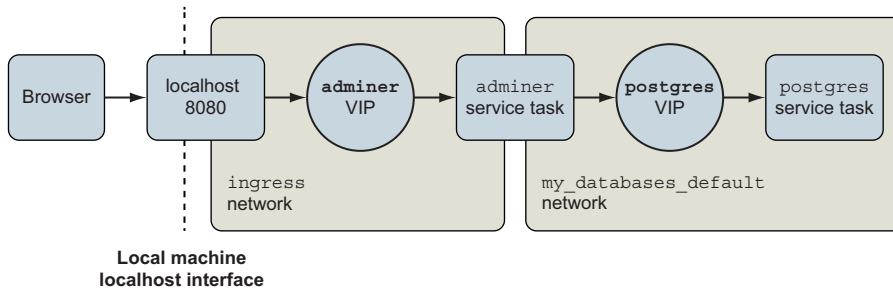


Figure 11.7 Docker network topology, service virtual IP addresses, and load balancing

From there, it is forwarded to one of the service replicas. This means that there are a few more things to understand about Docker networks. When you're using services, you're using at least two Docker networks.

The first network is named `ingress` and handles all port forwarding from the host interface to services. It is created when you initialize Docker in swarm mode. In this stack, there is only one service with forwarded ports, `adminer`. You can plainly see the associated interface in the `ingress` network upon inspection:

```
"Containers": {
  "6f64f8aec8c2...": {
    "Name": "my-databases_adminer.1.leijm5mpoz8o3lf4yxd7khnqn",
    "EndpointID": "9401eca40941...",
    "MacAddress": "02:42:0a:ff:00:22",
    "IPv4Address": "10.255.0.34/16",
    "IPv6Address": ""
  },
  "ingress-sbox": {
    "Name": "ingress-endpoint",
    "EndpointID": "36c9b1b2d807...",
    "MacAddress": "02:42:0a:ff:00:02",
    "IPv4Address": "10.255.0.2/16",
    "IPv6Address": ""
  }
}
```

Every service that uses port forwarding will have an interface in this network; `ingress` is critical to the Docker service functionality.

The second network is shared between all of the services in your stack. The Compose file you used to create the `my-databases` stack does not define any networks, but if you watch closely during initial deployment, you'll see that Docker will create a network for your stack named `default`. All services in your stack will be attached to this network by default, and all service-to-service communication will go through this network. When you inspect that network, you will see three entries like the following:

```
"Containers": {
  "3e57151c76bd...": {
```

```

    "Name": "my-databases_postgres.1.44phu5vee71bu3r3ao38ffqfu",
    "EndpointID": "375felbf3bc8...",
    "MacAddress": "02:42:0a:00:05:0e",
    "IPv4Address": "10.0.5.14/24",
    "IPv6Address": ""
  },
  "6f64f8aec8c2...": {
    "Name": "my-databases_adminer.1.leijm5mpoz8o3lf4yxd7khnqn",
    "EndpointID": "7009ae008702...",
    "MacAddress": "02:42:0a:00:05:0f",
    "IPv4Address": "10.0.5.15/24",
    "IPv6Address": ""
  },
  "lb-my-databases_default": {
    "Name": "my-databases_default-endpoint",
    "EndpointID": "8b94baa16c94...",
    "MacAddress": "02:42:0a:00:05:04",
    "IPv4Address": "10.0.5.4/24",
    "IPv6Address": ""
  }
}

```

The top two interfaces are used by the individual containers running the `postgres` and `adminer` services. If you were to resolve the `postgres` service name to an IP address, you might expect the address `10.0.5.14` in response. But what you would get is something else not listed here. The address listed here is the container address, or the address that the internal load balancer would forward requests onto. The address you would get is listed under `endpoints` in the `postgres` service spec. When you run `docker service inspect my-databases_postgres`, part of the result will look like this:

```

"Endpoint": {
  "Spec": {
    "Mode": "vip"
  },
  "VirtualIPs": [
    {
      "NetworkID": "2wvn2x73bx55lrr0w08xk5am9",
      "Addr": "10.0.5.11/24"
    }
  ]
}

```

That virtual IP address is handled by the Docker internal load balancer. Connections to that address will be forwarded to one of the `postgres` service replicas.

You can change service network attachment or the networks Docker creates for a stack with Compose. Compose can create networks with specific names, types, driver options, or other properties. Working with networks is similar to volumes. There are two parts: a top-level `networks` property that includes network definitions, and a `networks` property of services, where you describe attachments. Consider this final example:

```

version: "3.7"
networks:
  foo:
    driver: overlay
volumes:
  pgdata: # empty definition uses volume defaults
services:
  postgres:
    image: dockerinaction/postgres:11-alpine
    volumes:
      - type: volume
        source: pgdata # The named volume above
        target: /var/lib/postgresql/data
    networks:
      - foo
    environment:
      POSTGRES_PASSWORD: example
  adminer:
    image: dockerinaction/adminer:4
    networks:
      - foo
    ports:
      - 8080:8080
    deploy:
      replicas: 1

```

This example replaces the `my-databases_default` network with a network called `foo`. The two configurations are functionally equivalent.

There are several occasions to model networks with Compose. For example, if you manage multiple stacks and want to communicate on a shared network, you would declare that network as shown previously, but instead of specifying the driver, you would use the `external: true` property and the network name. Or suppose you have multiple groups of related services, but those services should operate in isolation. You would define the networks at the top level and use different network attachments to isolate the groups.

Summary

This chapter introduces higher-level Docker abstractions and working with the Docker declarative tooling, and using Compose to model the desired state for multi-service applications. Compose and declarative tooling relieve much of the tedium associated with command-line management of containers. The chapter covers the following:

- A service is any process, functionality, or data that must be discoverable and available over a network.
- Orchestrators such as Swarm track and automatically reconcile user-provided desired state and the current state of Docker objects including services, volumes, and networks.

- Orchestrators automate service replication, resurrection, deployments, health checking, and rollback.
- The desired state is what the user wants the system to be doing, or what it is supposed to be doing. People can describe desired state in a declarative style by using Compose files.
- Compose files are structured documents represented in YAML.
- Declarative environment descriptions with Compose enable environment versioning, sharing, iteration, and consistency.
- Compose can model services, volumes, and networks. Consult the official Compose file reference material for a full description of its capabilities.

12

First-class configuration abstractions

This chapter covers

- The problems configuration and secrets solve and the forms those solutions take
- Modeling and solving configuration problems for Docker services
- The challenge of delivering secrets to applications
- Modeling and delivering secrets to Docker services
- Approaches for using configurations and secrets in Docker services

Applications often run in multiple environments and must adapt to different conditions in those environments. You may run an application locally, in a test environment integrated with collaborating applications and data sources, and finally in production. Perhaps you deploy an application instance for each customer in order to isolate or specialize each customer's experience from that of others. The adaptations and specializations for each deployment are usually expressed via configuration. *Configuration* is data interpreted by an application to adapt its behavior to support a use case.

Common examples of configuration data include the following:

- Features to enable or disable
- Locations of services the application depends on
- Limits for internal application resources and activities such as database connection pool sizes and connection time-outs

This chapter will show you how to use Docker's configuration and secret resources to adapt Docker service deployments according to various deployment needs. Docker's first-class resources for modeling configuration and secrets will be used to deploy a service with different behavior and features, depending on which environment it is deployed to. You will see how the naive approach for naming configuration resources is troublesome and learn about a pattern for solving that problem. Finally, you will learn how to safely manage and use secrets with Docker services. With this knowledge, you will deploy the example web application with an HTTPS listener that uses a managed TLS certificate.

12.1 Configuration distribution and management

Most application authors do not want to change the program's source code and rebuild the application each time they need to vary the behavior of an application. Instead, they program the application to read configuration data on startup and adjust its behavior accordingly at runtime.

In the beginning, it may be feasible to express this variation via command-line flags or environment variables. As the program's configuration needs grow, many implementations move on to a file-based solution. These configuration files help you express both a greater number of configurations and more complex structures. The application may read its configuration from a file formatted in the ini, properties, JSON, YAML, TOML, or other format. Applications may also read configurations from a configuration server available somewhere on the network. Many applications use multiple strategies for reading configuration. For example, an application may read a file first and then merge the values of certain environment variables into a combined configuration.

Configuring applications is a long-standing problem that has many solutions. Docker directly supports several of these application configuration patterns. Before we discuss that, we will explore how the configuration change life cycle fits within the application's change life cycle. Configurations control a wide set of application behaviors and so may change for many reasons, as illustrated in figure 12.1.

Let's examine a few events that drive configuration changes. Configurations may change in conjunction with an application enhancement. For example, when developers add a feature, they may control access to that feature with a feature flag. An application deployment may need to update in response to a change outside the application's scope. For example, the hostname configured for a service dependency may need to update from `cluster-blue` to `cluster-green`. And of course,

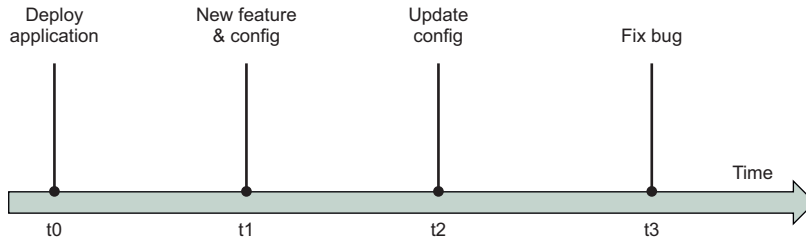


Figure 12.1 Timeline of application changes

applications may change code without changing configuration. These changes may occur for reasons that are internal or external to an application. The application’s delivery process must merge and deploy these changes safely regardless of the reason for the change.

A Docker service depends on configuration resources much in the same way it depends on the Docker image containing the application, as illustrated in figure 12.2. If a configuration or secret is missing, the application will probably not start or behave properly. Also, once an application expresses a dependency on a configuration resource, the existence of that dependency must be stable. Applications will likely break or behave in unexpected ways if the configuration they depend on disappears when the application restarts. The application might also break if the *values* inside the configuration change unexpectedly. For example, renaming or removing entries inside a configuration file will break an application that does not know how to read that format. So the configuration lifecycle must adopt a scheme that preserves backward compatibility for existing deployments.

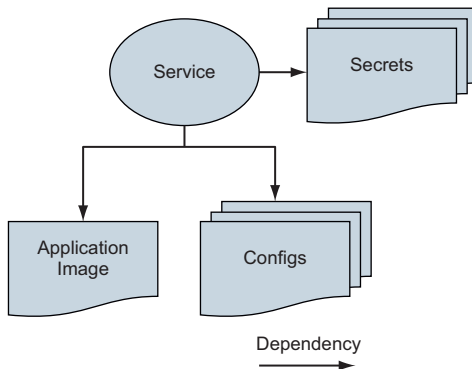


Figure 12.2 Applications depend on configurations.

If you separate change for software and configuration into separate pipelines, tension will exist between those pipelines. Application delivery pipelines are usually modeled in an application-centric way, with an assumption that all changes will run through the application’s source repository. Because configurations can change for reasons that

are external to the application, *and* applications are generally not built to handle changes that break backward compatibility in the configuration model, we need to model, integrate, and sequence configuration changes to avoid breaking applications. In the next section, we will separate configuration from an application to solve deployment variation problems. Then we will join the correct configuration with the service at deployment time.

12.2 Separating application and configuration

Let's work through a problem of needing to vary configuration of a Docker service deployed to multiple environments. Our example application is a `greetings` service that says "Hello World!" in different languages. The developers of this service want the service to return a greeting when a user requests one. When a native speaker verifies that a greeting translation is accurate, it is added to the list of greetings in the `config.common.yml` file:

```
greetings:
  - 'Hello World!'
  - 'Hola Mundo!'
  - 'Hallo Welt!'
```

The application image build uses the Dockerfile `COPY` instruction to populate the `greetings` application image with this common configuration resource. This is appropriate because that file has no deployment-time variation or sensitive data.

The service also supports loading environment-specific greetings in addition to the standard ones. This allows the development team to vary and test the greetings shown in each of three environments: `dev`, `stage`, and `prod`. The environment-specific greetings will be configured in a file named after the environment; for example, `config.dev.yml`:

```
# config.dev.yml
greetings:
  - 'Orbis Terrarum salve!'
  - 'Bonjour le monde!'
```

Both the common and environment-specific configuration files must be available as files on the `greetings` service container's filesystem, as shown in figure 12.3.

So the immediate problem we need to solve is how to get the environment-specific configuration file into the container by using only the deployment descriptor. Follow along with this example by cloning and reading the files in the https://github.com/dockerinaction/ch12_greetings.git source repository.

The `greetings` application deployment is defined using the Docker Compose application format and deployed to Docker Swarm as a stack. These concepts were introduced in chapter 11. In this application, there are three Compose files. The deployment configuration that is common across all environments is contained in

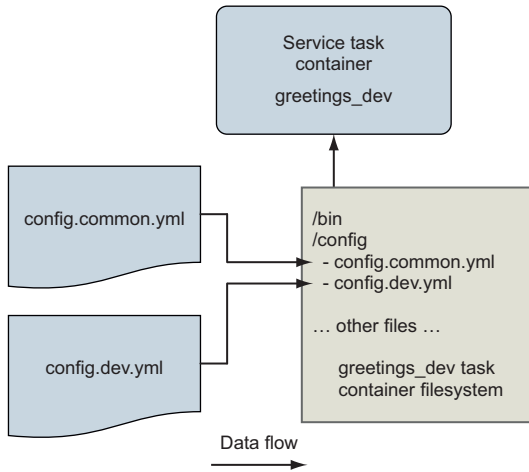


Figure 12.3 The `greetings` service supports common and environment-specific configuration via files.

`docker-compose.yml`. There are also environment-specific Compose files for both of the stage and prod environments (for example, `docker-compose.prod.yml`). The environment-specific Compose files define additional configuration and secret resources that the service uses in those deployments.

Here is the shared deployment descriptor, `docker-compose.yml`:

```
version: '3.7'

configs:
  env_specific_config:
    file: ./api/config/config.${DEPLOY_ENV:-prod}.yaml ← Defines a config resource using the contents of the env-specific configuration file

services:

  api:
    image: ${IMAGE_REPOSITORY:-dockerinaction/ch12_greetings}:api
    ports:
      - '8080:8080'
      - '8443:8443'
    user: '1000'
    configs:
      - source: env_specific_config
        target: /config/config.${DEPLOY_ENV:-prod}.yaml ← Maps the env_specific_config resource to a file in the container
    uid: '1000'
    gid: '1000'
    mode: 0400 #default is 0444 - readonly for all users
    secrets: []
    environment:
      DEPLOY_ENV: ${DEPLOY_ENV:-prod}
```

This Compose file loads the `greetings` application’s environment-specific configuration files into the `api` service’s containers. This supplements the common configuration

file built into the application image. The `DEPLOY_ENV` environment variable parameterizes this deployment definition. This environment variable is used in two ways.

First, the deployment descriptor will produce different deployment definitions when Docker interpolates `DEPLOY_ENV`. For example, when `DEPLOY_ENV` is set to `dev`, Docker will reference and load `config.dev.yml`.

Second, the deployment descriptor's value of the `DEPLOY_ENV` variable is passed to the `greetings` service via an environment variable definition. This environment variable signals to the service which environment it is running in, enabling it to do things such as load configuration files named after the environment. Now let's examine Docker config resources and how the environment-specific configuration files are managed.

12.2.1 Working with the config resource

A Docker *config resource* is a Swarm cluster object that deployment authors can use to store runtime data needed by applications. Each config resource has a cluster-unique name and a value of up to 500 KB. When a Docker service uses a config resource, Swarm mounts a file inside the service's container filesystems populated with the config resource's contents.

The top-level `configs` key defines the Docker config resources that are specific to this application deployment. This `configs` key defines a map with one config resource, `env_specific_config`:

```
configs:
  env_specific_config:
    file: ./api/config/config.${DEPLOY_ENV:-prod}.yml
```

When this stack is deployed, Docker will interpolate the filename with the value of the `DEPLOY_ENV` variable, read that file, and store it in a config resource named `env_specific_config` inside the Swarm cluster.

Defining a config in a deployment does not automatically give services access to it. To give a service access to a config, the deployment definition must map it under the service's own `configs` key. The config mapping may customize the location, ownership, and permissions of the resulting file on the service container's filesystem:

```
# ...snip...
services:
  api:
    # ...snip...
    user: '1000'
    configs:
      - source: env_specific_config
        target: /config/config.${DEPLOY_ENV:-prod}.yml
        uid: '1000'
        gid: '1000'
        mode: 0400
```

Overrides default file mode of 0444, read-only for all users →

Overrides default uid and gid of 0, to match service user with uid 1000 ←

Overrides default target file path is /env_specific_config ←

In this example, the `env_specific_config` resource is mapped into the `greetings` service container with several adjustments. By default, config resources are mounted into the container filesystem at `/<config_name>`; for example, `/env_specific_config`. This example maps `env_specific_config` to the target location `/config/config.${DEPLOY_ENV:-prod}.yaml`. Thus, for a deployment in the `dev` environment, the environment-specific config file will appear at `/config/config.dev.yaml`. The ownership of this configuration file is set to `userid=1000` and `groupid=1000`. By default, files for config resources are owned by the user ID and group ID 0. The file permissions are also narrowed to a mode of `0400`. This means the file is readable by only the file owner, whereas the default is readable by the owner, group, and others (`0444`).

These changes are not strictly necessary for this application because it is under our control. The application could be implemented to work with Docker's defaults instead. However, other applications are not as flexible and may have startup scripts that work in a specific way that you cannot change. In particular, you may need to control the configuration filename and ownership in order to accommodate a program that wants to run as a particular user and read configuration files from predetermined locations. Docker's service config resource mapping allows you to accommodate these demands. You can even map a single config resource into multiple service definitions differently, if needed.

With the config resources set along with the service definition, let's deploy the application.

12.2.2 *Deploying the application*

Deploy the `greetings` application with the `dev` environment configuration by running the following command:

```
DEPLOY_ENV=dev docker stack deploy \
  --compose-file docker-compose.yml greetings_dev
```

After the stack deploys, you can point a web browser to the service at `http://localhost:8080/`, and should see a welcome message like this:

```
Welcome to the Greetings API Server!
Container with id 642abc384de5 responded at 2019-04-16 00:24:37.0958326 +0000 UTC
DEPLOY_ENV: dev
```

← The value "dev" is read from environment variable.

When you run `docker stack deploy`, Docker reads the application's environment-specific configuration file and stores it as a config resource inside the Swarm cluster. Then when the `api` service starts, Swarm creates a copy of those files on a temporary, read-only filesystem. Even if you set the file mode as writable (for example, `rw-rw-rw-`), it will be ignored. Docker mounts these files at the target location specified in the config. The config file's target location can be pretty much anywhere, even inside a directory that contains regular files from the application image. For example, the `greetings` service's common config files (COPY'd into app image) and environment-specific config

file (a Docker config resource) are both available in the `/config` directory. The application container can read these files when it starts up, and those files are available for the life of the container.

On startup, the `greetings` application uses the `DEPLOY_ENV` environment variable to calculate the name of the environment-specific config file; for example, `/config/config.dev.yml`. The application then reads both of its config files and merges the list of greetings. You can see how the `greetings` service does this by reading the `api/main.go` file in the source repository. Now, navigate to the `http://localhost:8080/greeting` endpoint and make several requests. You should see a mix of greetings from the common and environment-specific config. For example:

```
Hello World!  
Orbis Terrarum salve!  
Hello World!  
Hallo Welt!  
Hola Mundo!
```

Config resources vs. config images

You may recall the Configuration Image per Deployment Stage pattern described in chapter 10. In that pattern, environment-specific configurations are built into an image that runs as a container, and that filesystem is mounted into the “real” service container’s filesystem at runtime. The Docker config resource automates most of this pattern. Using config resources results in a file being mounted into the service task container and does so without needing to create and track additional images. The Docker config resource also allows you to easily mount a single config file into an arbitrary location on the filesystem. With the container image pattern, it’s best to mount an entire directory in order to avoid confusion about what file came from which image.

In both approaches, you’ll want to use uniquely identified config or image names. However, it is convenient that image names can use variable substitutions in Docker Compose application deployment descriptors, avoiding the resource-naming problems that will be discussed in the next section.

So far, we have managed config resources through the convenience of a Docker Compose deployment definition. In the next section, we will step down an abstraction level and use the `docker` command-line tool to directly inspect and manage config resources.

12.2.3 Managing config resources directly

The `docker config` command provides another way to manage config resources. The `config` command has several subcommands for creating, inspecting, listing, and removing config resources: `create`, `inspect`, `ls`, and `rm`, respectively. You can use these commands to directly manage a Docker Swarm cluster’s config resources. Let’s do that now.

Inspect the greetings service's `env_specific_config` resource:

```
docker config inspect greetings_dev_env_specific_config
```

Docker automatically prefixes the config resource with the `greetings_dev` stack name.

This should produce output similar to the following:

```
[
  {
    "ID": "bconclhuvlzoix3z5xj0j16u1",
    "Version": {
      "Index": 2066
    },
    "CreatedAt": "2019-04-12T23:39:30.6328889Z",
    "UpdatedAt": "2019-04-12T23:39:30.6328889Z",
    "Spec": {
      "Name": "greetings_dev_env_specific_config",
      "Labels": {
        "com.docker.stack.namespace": "greetings"
      },
      "Data":
        "Z3JlZXRpbmdzOgogIC0gJ09yYmlzIFRlcnJhcnVtIHhnbHJlIiScKICAtICdCb25qb3VyIGx1IG1vbmRlIiScK"
    }
  }
]
```

The `inspect` command reports metadata associated with the config resource and the config's value. The config's value is returned in the `Data` field as a Base64-encoded string. This data is not encrypted, so no confidentiality is provided here. The Base64 encoding only facilitates transportation and storage within the Docker Swarm cluster. When a service references a config resource, Swarm will retrieve this data from the cluster's central store and place it in a file on the service task's filesystem.

Docker config resources are immutable and cannot be updated after they are created. The `docker config` command supports only `create` and `rm` operations to manage the cluster's config resources. If you try to create a config resource multiple times by using the same name, Docker will return an error saying the resource already exists:

```
$ docker config create greetings_dev_env_specific_config \
  api/config/config.dev.yml
Error response from daemon: rpc error: code = AlreadyExists ↵
  desc = config greetings_dev_env_specific_config already exists
```

Similarly, if you change the source configuration file and try to redeploy the stack by using the same config resource name, Docker will also respond with an error:

```
$ DEPLOY_ENV=dev docker stack deploy \
  --compose-file docker-compose.yml greetings_dev
failed to update config greetings_dev_env_specific_config: ↵
```

```
Error response from daemon: rpc error: code = InvalidArgument ↵
desc = only updates to Labels are allowed
```

You can visualize the relationships between a Docker service and its dependencies as a directed graph, like the simple one shown in figure 12.4:

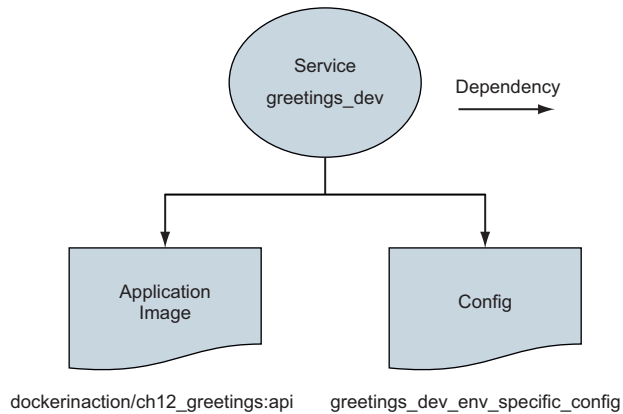


Figure 12.4 Docker services depend on config resources.

Docker is trying to maintain a stable set of dependencies between Docker services and the config resources they depend on. If the `greetings_dev_env_specific_config` resource were to change or be removed, new tasks for the `greetings_dev` service may not start. Let's see how Docker tracks these relationships.

Config resources are each identified by a unique ConfigID. In this example, the `greetings_dev_env_specific_config` is identified by `bconclhuvlzoix3z5xj0j16u1`, which is visible in the `docker config inspect` command's `greetings_dev_env_specific_config` output. This same config resource is referenced by its ConfigID in the `greetings` service definition.

Let's verify that now by using the `docker service inspect` command. This inspection command prints only the `greetings` service references to config resources:

```
docker service inspect \
  --format '{{ json .Spec.TaskTemplate.ContainerSpec.Configs }}' \
  greetings_dev_api
```

For this service instantiation, the command produces the following:

```
[
  {
    "File": {
      "Name": "/config/config.dev.yml",
      "UID": "1000",
      "GID": "1000",
      "Mode": 256
    },
    "ConfigID": "bconclhuvlzoix3z5xj0j16u1",
```

```

    "ConfigName": "greetings_dev_env_specific_config"
  }
]

```

There are a few important things to call out. First, the `ConfigID` references the `greetings_dev_env_specific_config` config resource's unique config ID, `bconclhuvlzoix3z5xj0j16u1`. Second, the service-specific target file configurations have been included in the service's definition. Also note that you cannot create a config if the name already exists or remove a config while it is in use. Recall that the `docker config` command does not offer an update subcommand. This may leave you wondering how you update configurations. Figure 12.5 illustrates a solution to the problem.

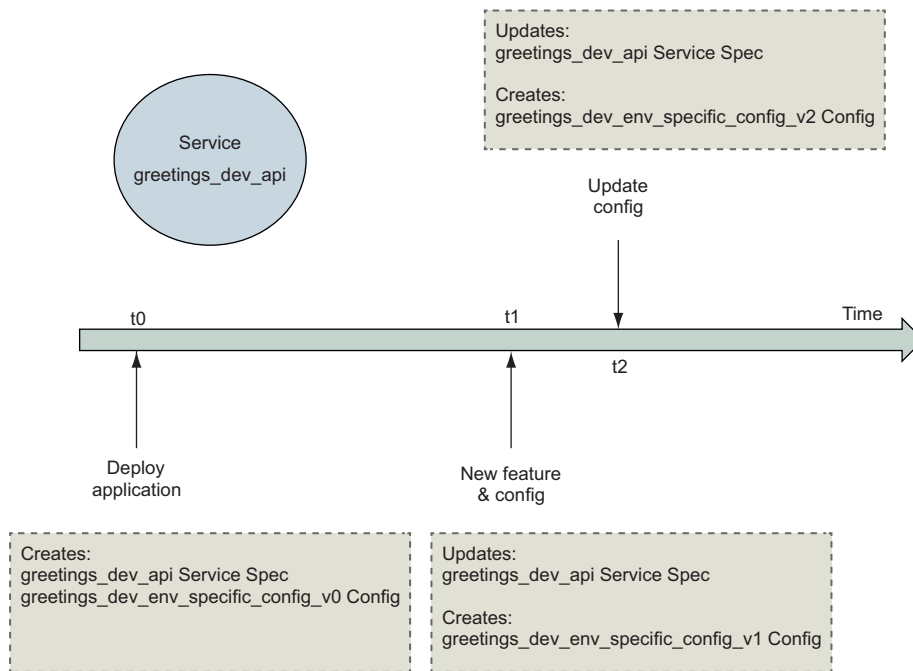


Figure 12.5 Copy on deploy

The answer is that you *don't* update Docker config resources. Instead, when a configuration file changes, the deployment process should create a new resource with a different name and then reference that name in service deployments. The common convention is to append a version number to the configuration resource's name. The `greetings` application's deployment definition could define an `env_specific_config_v1` resource. When the configuration changes, that configuration could be stored in a new config resource named `env_specific_config_v2`. Services can adopt this new config by updating configuration references to this new config resource name.

This implementation of immutable config resources creates challenges for automated deployment pipelines. The issue is discussed in detail on GitHub issue [moby/moby 35048](#). The main challenge is that the name of a config resource cannot be parameterized directly in the YAML Docker Compose deployment definition format. Automated deployment processes can work around this by using a custom script that substitutes a unique version into the Compose file prior to running the deployment.

For example, say the deployment descriptor defines a config `env_specific_config_vNNN`. An automated build process could search for the `_vNNN` character sequence and replace it with a unique deployment identifier. The identifier could be the deployment job's ID or the application's version from a version-control system. A deploy job with ID 3 could rewrite all instances of `env_specific_config_vNNN` to `env_specific_config_v3`.

Try this config resource versioning scheme. Start by adding some greetings to the `config.dev.yml` file. Then rename the `env_specific_config` resource in `docker-compose.yml` to `env_specific_config_v2`. Be sure to update the key names in both the top-level config map as well as the `api` service's list of configs. Now update the application by deploying the stack again. Docker should print a message saying that it is creating `env_specific_config_v2` and updating the service. Now when you make requests to the greeting endpoint, you should see the greetings you added mixed into the responses.

This approach may be acceptable to some but has a few drawbacks. First, deploying resources from files that don't match version control may be a nonstarter for some people. This issue can be mitigated by archiving a copy of the files used for deployment. A second issue is that this approach will create a set of config resources for each deployment, and the old resources will need to be cleaned up by another process. That process could periodically inspect each config resource to determine whether it is in use and remove it if it is not.

We are done with the dev deployment of the `greetings` application. Clean up those resources and avoid conflicts with later examples by removing the stack:

```
docker stack rm greetings_dev
```

That completes the introduction of Docker config and its integration into delivery pipelines. Next, we'll examine Docker's support for a special kind of configuration: secrets.

12.3 Secrets—A special kind of configuration

Secrets look a lot like configuration with one important difference. The *value* of a secret is important and often highly valuable because it authenticates an identity or protects data. A secret may take the form of a password, API key, or private encryption key. If these secrets leak, someone may be able to perform actions or access data they are not authorized for.

A further complication exists. Distributing secrets in artifacts such as Docker images or configuration files makes controlling access to those secrets a wide and

difficult problem. Every point in the distribution chain needs to have robust and effective access controls to prevent leakage.

Most organizations give up on trying to deliver secrets without exposing them through normal application delivery channels. This is because delivery pipelines often have many points of access, and those points may not have been designed or configured to ensure confidentiality of data. Organizations avoid those problems by storing secrets in a secure vault and injecting them right at the final moment of application delivery using specialized tooling. These tools enable applications to access their secrets *only* in the runtime environment.

Figure 12.6 illustrates the flow of application configuration and secret data through application artifacts.

If an application is started without secret information such as a password credential to authenticate to the secret vault, how can the vault authorize access to the application's secrets? It can't. This is the *First Secret Problem*. The application needs help bootstrapping the chain of trust that will allow it to retrieve its secrets. Fortunately, Docker's design for clustering, services, and secret management solves this problem, as illustrated in figure 12.7.

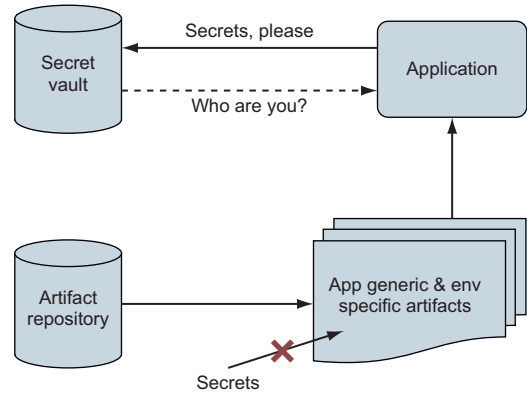


Figure 12.6 The first secret problem

Service specs and secrets are modifiable by only authorized docker users.

Service's task template references specific SecretIDs and ConfigIDs.

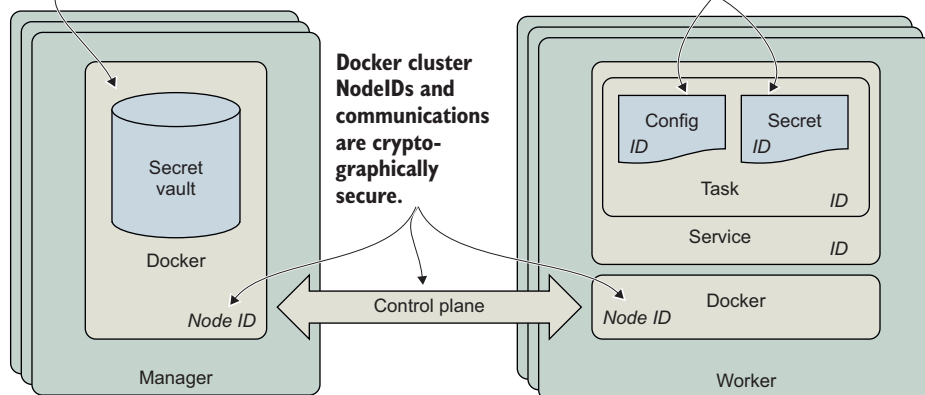


Figure 12.7 The Docker swarm cluster's chain of trust

The core of the First Secret Problem is one of identity. In order for a secret vault to authorize access to a given secret, it must first authenticate the identity of the requestor. Fortunately, Docker Swarm includes a secure secret vault and solves the trust bootstrapping problem for you. The Swarm secret vault is tightly integrated with the cluster's identity and service management functions that are using secure communication channels. The Docker service ID serves as the application's identity. Docker Swarm uses the service ID to determine which secrets the service's tasks should have access to. When you manage application secrets with Swarm's vault, you can be confident that only a person or process with administrative control of the Swarm cluster can provision access to a secret.

Docker's solution for deploying and operating services is built on a strong and cryptographically sound foundation. Each Docker service has an identity, and so does each task container running in support of that service. All of these tasks run on top of Swarm nodes that also have unique identities. The Docker secret management functionality is built on top of this foundation of identity. Every task has an identity that is associated with a service. The service definition references the secrets that the service and task needs. Since service definitions can be modified by only an authorized user of Docker on the manager node, Swarm knows which secrets a service is authorized to use. Swarm can then deliver those secrets to nodes that will run the service's tasks.

NOTE The Docker swarm clustering technology implements an advanced, secure design to maintain a secure, highly available, and optimally performing control plane. You can learn more about this design and how Docker implements it at <https://www.docker.com/blog/least-privilege-container-orchestration/>.

Docker services solve the First Secret Problem by using Swarm's built-in identity management features to establish trust rather than relying on a secret passed via another channel to authenticate the application's access to its secrets.

12.3.1 Using Docker secrets

Using Docker secret resources is similar to using Docker config resources, with a few adjustments.

Again, Docker provides secrets to applications as files mounted to a container-specific, in-memory, read-only `tmpfs` filesystem. By default, secrets will be placed into the container's filesystem in the `/run/secrets` directory. This method of delivering secrets to applications avoids several leakage problems inherent with providing secrets to applications as environment variables.

Now let's examine how to tell an application where a secret or configuration file has been placed into a container; see figure 12.8.

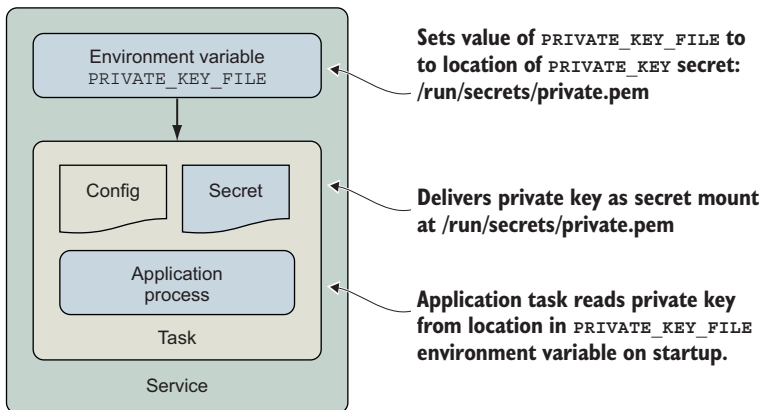


Figure 12.8 Provide location of secret file to read as environment variable

Problems with secrets as environment variables

The most important and common problems with using environment variables as secret transfer mechanisms are as follows:

- You can't assign access-control mechanisms to an environment variable.
- This means any process executed by the application will likely have access to those env vars. To illustrate this, think about what it might mean for an application that does image resizing via ImageMagick to execute resizing operations with untrusted input in the environment containing the parent application's secrets. If the environment contains API keys in well-known locations, as is common with cloud providers, those secrets could be stolen easily. Some languages and libraries will help you prepare a safe process execution environment, but your mileage will vary.
- Many applications will print all of their environment variables to standard out when issued a debugging command or when they crash. This means you may expose secrets in your logs on a regular basis.

When applications read secrets from files, we often need to specify the location of that file at startup. A common pattern for solving this problem is to pass the location of the file containing a secret, such as a password to the application, as an environment variable. The location of the secret in the container is not sensitive information, and only processes running inside the container will have access to that file, file permissions permitting. The application can read the file specified by the environment variable to load the secret. This pattern is also useful for communicating the location of configuration files.

Let's work through an example; we'll provide the greetings service with a TLS certificate and private key so that it can start a secure HTTPS listener. We will store the certificate's private key as a Docker secret and the public key as a config. Then we will provide those resources to the greeting service's production service configuration. Finally, we will specify the location of the files to the greetings service via an environment variable so that it knows where to load those files from.

Now we will deploy a new instance of the greetings service with the stack configured for production. The deployment command is similar to the one you ran previously. However, the production deployment includes an additional `--compose-file` option to incorporate the deployment configuration in `docker-compose.prod.yml`. The second change is to deploy the stack by using the name `greetings_prod` instead of `greetings_dev`.

Run this docker stack deploy command now:

```
DEPLOY_ENV=prod docker stack deploy --compose-file docker-compose.yml \
  --compose-file docker-compose.prod.yml \
  greetings_prod
```

You should see some output:

```
Creating network greetings_prod_default
Creating config greetings_prod_env_specific_config
service api: secret not found: ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1
```

The deployment fails because the `ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1` secret resource is not found. Let's examine `docker-compose.prod.yml` and determine why this is happening. Here are the contents of that file:

```
version: '3.7'
configs:
  ch12_greetings_svc-prod-TLS_CERT_V1:
    external: true

secrets:
  ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1:
    external: true

services:

  api:
    environment:
      CERT_PRIVATE_KEY_FILE: '/run/secrets/cert_private_key.pem'
      CERT_FILE: '/config/svc.crt'
    configs:
      - source: ch12_greetings_svc-prod-TLS_CERT_V1
        target: /config/svc.crt
        uid: '1000'
        gid: '1000'
        mode: 0400
```

```

secrets:
  - source: ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1
    target: cert_private_key.pem
    uid: '1000'
    gid: '1000'
    mode: 0400

```

There is a top-level `secrets` key that defines a secret named `ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1`, the same as what was reported in the error. The secret definition has a key we have not seen before, `external: true`. This means that the value of the secret is not defined by this deployment definition, which is prone to leakage. Instead, the `ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1` secret must be created by a Swarm cluster administrator using the `docker CLI`. Once the secret is defined in the cluster, this application deployment can reference it.

Let's define the secret now by running the following command:

```

$ cat api/config/insecure.key | \
  docker secret create ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1 -
vnyy0gr1a09be0vcfvvqogeoj

```

The `docker secret create` command requires two arguments: the name of the secret and the value of that secret. The value may be specified either by providing the location of a file, or by using the `-` (hyphen) character to indicate that the value will be provided via standard input. This shell command demonstrates the latter form by printing the contents of the example TLS certificate's private key, `insecure.key`, into the `docker secret create` command. The command completes successfully and prints the ID of the secret: `vnyy0gr1a09be0vcfvvqogeoj`.

WARNING Do *not* use this certificate and private key for anything but working through these examples. The private key has not been kept confidential and thus cannot protect your data effectively.

Use the `docker secret inspect` command to view details about the secret resource that Docker created:

```

$ docker secret inspect ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1
[
  {
    "ID": "vnyy0gr1a09be0vcfvvqogeoj",
    "Version": {
      "Index": 2172
    },
    "CreatedAt": "2019-04-17T22:04:19.3078685Z",
    "UpdatedAt": "2019-04-17T22:04:19.3078685Z",
    "Spec": {
      "Name": "ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1",
      "Labels": {}
    }
  }
]

```

Notice that there is no `Data` field as there was with a `config` resource. The secret's value is not available via the Docker tools or Docker Engine API. The secret's value is guarded closely by the Docker Swarm control plane. Once you load a secret into Swarm, you cannot retrieve it by using the `docker` CLI. The secret is available only to services that use it. You may also notice that the secret's spec does not contain any labels, as it is managed outside the scope of a stack.

When Docker creates containers for the `greetings` service, the secret will be mapped into the container in a way that is almost identical to the process we already described for `config` resources. Here is the relevant section from the `docker-compose.prod.yml` file:

```
services:
  api:
    environment:
      CERT_PRIVATE_KEY_FILE: '/run/secrets/cert_private_key.pem'
      CERT_FILE: '/config/svc.crt'
    secrets:
      - source: ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1
        target: cert_private_key.pem
        uid: '1000'
        gid: '1000'
        mode: 0400
    # ... snip ...
```

The `ch12_greetings-svc-prod-TLS_PRIVATE_KEY_V1` secret will be mapped into the container, in the file `cert_private_key.pem`. The default location for secret files is `/run/secrets/`. This application looks for the location of its private key and certificate in environment variables, so those are also defined with fully qualified paths to the files. For example, the `CERT_PRIVATE_KEY_FILE` environment variable's value is set to `/run/secrets/cert_private_key.pem`.

The production `greetings` application also depends on a `ch12_greetings_svc-prod-TLS_CERT_V1` `config` resource. This `config` resource contains the public, non-sensitive, x.509 certificate the `greetings` application will use to offer HTTPS services. The private and public keys of an x.509 certificate change together, which is why these secret and `config` resources are created as a pair. Define the certificate's `config` resource now by running the following command:

```
$ docker config create \
  ch12_greetings_svc-prod-TLS_CERT_V1 api/config/insecure.crt
5a1lybiyjnaseg0j1wj2s1v5m
```

The `docker config create` command works like the secret creation command. In particular, a `config` resource can be created by specifying the path to a file, as we have done here with `api/config/insecure.crt`. The command completed successfully and printed the new `config` resource's unique ID, `5a1lybiyjnaseg0j1wj2s1v5m`.

Now, rerun the `deploy` command:

```
$ DEPLOY_ENV=prod docker stack deploy \
  --compose-file docker-compose.yml \
```

```
--compose-file docker-compose.prod.yml \  
greetings_prod  
Creating service greetings_prod_api
```

This attempt should succeed. Run `docker service ps greetings_prod_api` and verify that the service has a single task in the running state:

ID	NAME	IMAGE	NODE
DESIRED STATE	CURRENT STATE	ERROR	PORTS
93fgzy5lmarp	greetings_prod_api.1	dockerinaction/ch12_greetings:api	
docker-desktop	Running	Running 2 minutes ago	

Now that the production stack is deployed, we can check the service's logs to see whether it found the TLS certificate and private key:

```
docker service logs --since 1m greetings_prod_api
```

That command will print the greetings service application logs, which should look like this:

```
Initializing greetings api server for deployment environment prod  
Will read TLS certificate private key from  
  '/run/secrets/cert_private_key.pem'  
chars in certificate private key 3272  
Will read TLS certificate from '/config/svc.crt'  
chars in TLS certificate 1960  
Loading env-specific configurations from /config/config.common.yml  
Loading env-specific configurations from /config/config.prod.yml  
Greetings: [Hello World! Hola Mundo! Hallo Welt!]  
Initialization complete  
Starting https listener on :8443
```

Indeed, the greetings application found the private key at `/run/secrets/cert_private_key.pem` and reported that the file has 3,272 characters in it. Similarly, the certificate has 1,960 characters. Finally, the greetings application reported that it is starting a listener for HTTPS traffic on port 8443 inside the container.

Use a web browser to open `https://localhost:8443`. The example certificate is not issued by a trusted certificate authority, so you will receive a warning. If you proceed through that warning, you should see a response from the greetings service:

```
Welcome to the Greetings API Server!  
Container with id 583a5837d629 responded at 2019-04-17 22:35:07.3391735 +0000 UTC  
DEPLOY_ENV: prod
```

Woo-hoo! The greetings service is now serving traffic over HTTPS using TLS certificates delivered by Docker's secret management facilities. You can request greetings from the service at `https://localhost:8443/greeting` as you did before. Notice that only the three greetings from the common config are served. This is because the application's environment-specific configuration file for prod, `config.prod.yml`, does not add any greetings.

The greetings service is now using every form of configuration supported by Docker: files included in the application image, environment variables, config resources, and secret resources. You've also seen how to combine the usage of all these approaches to vary application behavior in a secure manner across several environments.

Summary

This chapter described the core challenges of varying application behavior at deployment time instead of build time. We explored how you can model this variation with Docker's configuration abstractions. The example application demonstrated using Docker's config and secret resources to vary its behavior across environments. This culminated in a Docker Service serving traffic over https with an environment-specific dataset. The key points to understand from this chapter are:

- Applications often must adapt their behavior to the environment they are deployed into.
- Docker config and secret resources can be used to model and adapt application behavior to various deployment needs.
- Secrets are a special kind of configuration data that is challenging to handle safely.
- Docker Swarm establishes a chain of trust and uses Docker service identities to ensure that secrets are delivered correctly and securely to applications.
- Docker provides config and secrets to services as files on a container-specific `tmpfs` filesystem that applications can read at startup.
- Deployment processes must use a naming scheme for config and secret resources that enables automation to update services.

Orchestrating services on a cluster of Docker hosts with Swarm

This chapter covers

- How Docker application deployments work and options
- Deploying a multitier application to Docker Swarm
- How Swarm attempts to converge the Docker application deployment to the desired state declared by operators
- How Swarm ensures the desired number of replicas are running around the cluster within the declared placement and resource constraints
- Routing of request traffic from a cluster node to network service instances and how collaborating services reach each other using Docker networks
- Controlling placement of Docker Service containers within the cluster

13.1 Clustering with Docker Swarm

Application developers and operators frequently deploy services onto multiple hosts to achieve greater availability and scalability. When an application is deployed across multiple hosts, the redundancy in the application's deployment provides capacity that can serve requests when a host fails or is removed from service.

Deploying across multiple hosts also permits the application to use more compute resources than any single host can provide.

For example, say you run an e-commerce site that usually performs well on a single host, but is slow during big promotions that drive peak load twice as high as normal. This site might benefit from being redeployed onto three hosts. Then you should have enough capacity to handle peak traffic even if one host fails or is out of service for an upgrade. In this chapter, we will show you how to model and deploy a web API across a cluster of hosts managed with Swarm.

Docker Swarm provides a sophisticated platform for deploying and operating a containerized application across a set of Docker hosts. Docker's deployment tooling automates the process for deploying a new Docker service to the cluster or changes to an existing service. Service configuration changes may include anything declared in the service definition (`docker-compose.yml`) such as image, container command, resource limits, exposed ports, mounts, and consumed secrets. Once deployed, Swarm supervises the application so that problems are detected and repaired. Additionally, Swarm routes requests from the application's users to the service's containers.

In this chapter, we will examine how Docker Swarm supports each of these functions. We will build on the service, configuration, and secret resources explored in chapters 11 and 12. We will also leverage your fundamental knowledge of Docker containers (chapter 2), resource limits (chapter 6), and networking (chapter 5).

13.1.1 Introducing Docker Swarm mode

Docker Swarm is a clustering technology that connects a set of hosts running Docker and lets you run applications built using Docker services across those machines. Swarm orchestrates the deployment and operation of Docker services across the collection of machines. Swarm schedules tasks according to the application's resource requirements and machine capabilities. The Swarm clustering software is included in the Docker Engine and command-line tool. You can enable Swarm mode and start using Swarm without installing any additional components. Figure 13.1 shows how the components of a Docker Swarm deployment relate to each other and how the machines of the cluster collaborate to run applications.

When you join a Docker Engine to a Swarm cluster, you specify whether that machine should be a manager or a worker. Managers listen for instructions to create, change, or remove definitions for entities such as Docker services, configuration, and secrets. Managers instruct worker nodes to create containers and volumes that implement Docker service instances. Managers continuously converge the cluster to the state you have declared it should be in. The control plane connecting the cluster's Docker Engines depicts the communication of the desired cluster state and events related to realizing that state. Clients of a Docker service may send requests to any node of the cluster on the port published for that service. Swarm's network mesh will route the request from whichever node received the request to a healthy service container that can handle it. Swarm deploys and manages lightweight, dedicated load-balancer

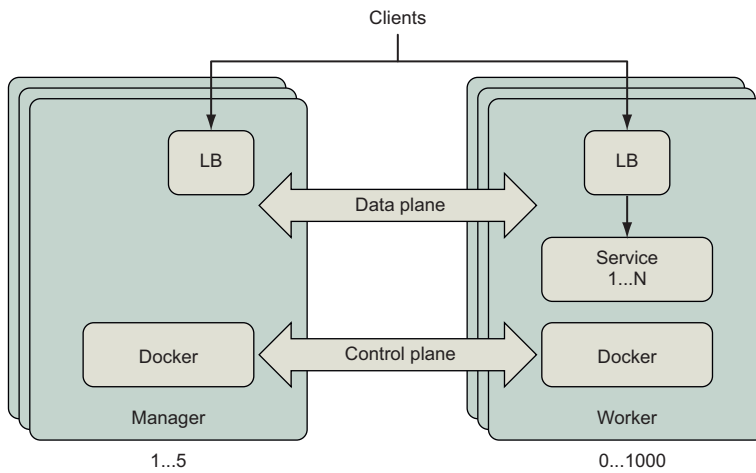


Figure 13.1 Swarm cluster deployment

and network routing components to receive and transport network traffic for each published port. Section 13.3.1 explains the Swarm network mesh in detail. Let's deploy a cluster to work through the examples in this chapter.

Swarm clusters can be deployed in many topologies. Each cluster has at least one manager to safeguard cluster state and orchestrate services across workers. Swarm managers require a majority of the managers to be available in order to coordinate and record a change to the cluster. Most production Swarm deployments should have three or five nodes in the manager role. Increasing the number of managers will improve availability of the Swarm control plane, but will also increase the time it takes for managers to acknowledge a change to the cluster. See the Swarm Admin Guide for a detailed explanation of the trade-offs (https://docs.docker.com/engine/swarm/admin_guide/). Swarm clusters can scale reliably to hundreds of worker nodes. The community has demonstrated tests of a single Swarm with thousands of worker nodes (see the Swarm3K project at <https://dzone.com/articles/docker-swarm-lessons-from-swarm3k>).

Swarm, the native clustered application deployment option provided by Docker, supports the Docker application model well. Many people will find Swarm simpler to deploy, use, and manage than other container clustering technologies. You may find it useful to deploy small Swarm clusters for an individual team or project. A large Swarm cluster can be partitioned into multiple zones by using labels, and then you can place service instances into the proper zone by using scheduling constraints. You can label cluster resources with metadata meaningful to your organization such as `environment=dev` or `zone=private` so the cluster's actual management model matches your own terminology.

13.1.2 Deploying a Swarm cluster

You have many options for building a swarm from a cluster of nodes. The examples in this chapter use a Swarm cluster with five nodes, though most of the examples work on a single node, such as Docker for Mac. You may provision a Swarm cluster however you like. Because of the wide variety of provisioning options and the rate of change, we recommend you follow an up-to-date guide to provision a Swarm cluster on your favorite infrastructure provider. Many people deploy test clusters with `docker-machine` on cloud providers such as DigitalOcean and Amazon Web Services.

The examples in this chapter were created and tested using Play with Docker (<https://labs.play-with-docker.com/>). On the Play with Docker site, you can experiment with Docker and learn about it for free. The cluster was created using the Play with Docker template that provisions three manager and two worker nodes. You will need at least two workers to complete all of the exercises in this chapter.

The general process for deploying a Swarm cluster is as follows:

- 1 Deploy at least three nodes with Docker Engine installed and running, preferably five.
- 2 Ensure that network traffic is permitted between the machines on the following ports and protocols:
 - a TCP port 2377 for cluster management communications
 - b TCP and UDP port 7946 for communication among nodes
 - c UDP port 4789 for overlay network traffic
- 3 Initialize a Swarm cluster by running `docker swarm init` on a manager.
- 4 Record the Swarm cluster join tokens or display them again with `docker swarm join-token`.
- 5 Join the manager and then worker nodes to the cluster with `docker swarm join`.

13.2 Deploying an application to a Swarm cluster

In this section, we will deploy an example web application with a common three-tier architecture. The application features a stateless API server connected to a PostgreSQL relational database. Both the API server and database will be managed as Docker services. The database will use a Docker volume to persist data across restarts. The API servers will communicate with the database over a private, secure network. This application will demonstrate how the Docker resources you have learned about in previous chapters translate to a deployment spanning multiple nodes.

13.2.1 Introducing Docker Swarm cluster resource types

Docker Swarm supports nearly all of the concepts discussed in this book, as illustrated in figure 13.2. When using Swarm, these resources are defined and managed at the cluster level.

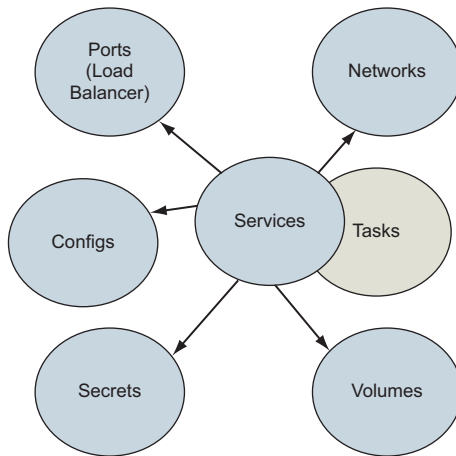


Figure 13.2 Docker Swarm resource types

The key Docker Swarm resource types are as follows:

- *Services*—A Docker service defines the application processes that run on the Swarm cluster’s nodes. Swarm managers interpret the service definition and create tasks that are executed on the cluster’s manager and worker nodes. Services are introduced in chapter 11.
- *Tasks*—Tasks define a containerized process that Swarm will schedule and run once until completion. A task that exits may be *replaced* by a new task, depending on the restart policy defined by the service. Tasks also specify dependencies on other cluster resources such as networks and secrets.
- *Networks*—Applications can use Docker overlay networks for traffic between services. Docker networks have low overhead, so you can create network topologies that suit your desired security model. Section 13.3.2 describes overlay networks.
- *Volumes*—Volumes provide persistent storage to service tasks. These volumes are bound to a single node. Volumes and mounts are described in chapter 4.
- *Configs and secrets*—Configurations and secrets (chapter 12) provide environment-specific configurations to services deployed on the cluster.

The example application uses each of these Docker resource types.

13.2.2 Defining an application and its dependencies by using Docker services

The example application we will work with in this chapter is a simple web application with three tiers: a load balancer, API server, and PostgreSQL database. We will model this application with Docker and then deploy it to our Swarm cluster. Logically, the application deployment will look like figure 13.3.

The application has an API server with two endpoints: `/` and `/counter`. The API service publishes a port to the cluster’s edge that is implemented by Swarm’s built-in

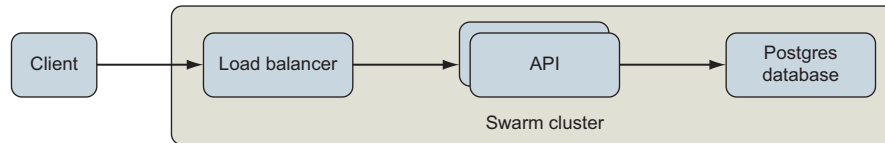


Figure 13.3 Logical architecture of example application

load balancer. Requests to the `/` endpoint will return information about the container that handled the request. The `/counter` endpoint will increment an integer with each request. The value of the counter is stored in a PostgreSQL database.

Let's define the application a piece at a time by using the Docker Compose version 3 format and synthesize the concepts covered in previous chapters. After that, we will deploy it with the `docker stack` command. This application definition is available in full at https://github.com/dockerinaction/ch13_multi_tier_app.git. Clone that repo to follow along as the application is explained piece by piece.

The application uses two networks, a *public* network handling requests coming from external clients, and a *private* network that is more trusted. These networks are described in the `docker-compose.yml` application descriptor as follows:

```

version: '3.7'

networks:
  public:
    driver: overlay
    driver_opts:
      encrypted: 'true'
  private:
    driver: overlay
    driver_opts:
      encrypted: 'true'
    attachable: true

```

NOTE The value `true` is quoted for `driver_opts` because Docker requires a string or number. The value of `true` is unquoted for `attachable` because Docker requires a boolean.

Both networks are defined by adding named entries to a top-level `networks` key in the application descriptor. The team that built this application has an end-to-end encryption requirement. The application definition satisfies a large portion of that requirement by encrypting all of the traffic on the networks used by the application. The only remaining work is to secure communications on the service's published port by using TLS. Section 13.3 explains why applications should secure the published ports, and chapter 12's `greetings` application showed one way to do this. This highlights an interesting feature of Swarm: it is easy to satisfy many transport encryption requirements by using a relatively simple, auditable configuration in the deployment descriptor.

Next, the database needs persistent storage for its data. We define a Docker volume under a top-level volumes key:

```
volumes:
  db-data:
```

Notice that no options are defined for this volume. Swarm will use Docker's built-in local volume driver to create it. The volume will be local to that Swarm node and not replicated, backed up, or shared elsewhere. Some Docker volume plugins can create and manage volumes that persist and share data across nodes; the Docker Cloudstor and REX-Ray plugins are good examples.

Before we move on to the service definition, create a reference to the password that will be used by the API to access the PostgreSQL database. The password will be configured in the Swarm cluster as one of the first steps of the deployment process. Add a top-level secrets key that instructs the password to be retrieved from the cluster's secret resources:

```
secrets
  ch13_multi_tier_app-POSTGRES_PASSWORD:
    external: true
```

Retrieves from cluster's
secret resources

Now we are ready to define the application's services. Let's start with the database by defining a postgres service under a top-level services key:

```
services:
  postgres:
    image: postgres:9.6.6
    networks:
      - private
    volumes:
      - db-data:/var/lib/postgresql/data
    secrets:
      - source: ch13_multi_tier_app-POSTGRES_PASSWORD
        target: POSTGRES_PASSWORD
    uid: '999'
    gid: '999'
    mode: 0400
    environment:
      POSTGRES_USER: 'exercise'
      POSTGRES_PASSWORD_FILE: '/run/secrets/POSTGRES_PASSWORD'
      POSTGRES_DB: 'exercise'
    deploy:
      replicas: 1
      update_config:
        order: 'stop-first'
      rollback_config:
        order: 'stop-first'
      resources:
        limits:
          cpus: '1.00'
          memory: 50M
```

Injects PostgreSQL
password from a
cluster-managed secret

The postgres user (uid: 999)
managed by the container
needs to read the file.

Ensures at-most one
instance of PostgreSQL by
limiting replicas to 1 and
stopping after first update
or rollback failure


```

reservations:
  cpus: '0.25'
  memory: 50M

```

This database service will use the official PostgreSQL image to start a database. That PostgreSQL container will attach to (only) the private network, mount the db-data volume, and use the `POSTGRES_*` environment variables to initialize the database. The `POSTGRES_DB` and `POSTGRES_USER` environment variables determine the name of the database and the user we will use to access the database, respectively. However, you should avoid providing secrets such as passwords to processes via environment variables because they are leaked easily.

A better way is to read that secret from a file that is managed safely. Docker supports this directly with its secrets functionality. The PostgreSQL image also has support for reading sensitive data such as the `POSTGRES_PASSWORD` from a file. For this stack definition, Docker will retrieve the PostgreSQL password from the cluster's `ch13_multi_tier_app-POSTGRES_PASSWORD` secret resource definition. Swarm places the secret's value in a file mounted into the container at `/run/secrets/POSTGRES_PASSWORD`. The PostgreSQL process switches to a user with user ID 999 when it starts up, so the secret file's owner is configured to be readable by that user.

NOTE All processes that execute inside a Docker container can access all the environment variables of that container. However, access to data in files is controlled by file permissions. So nobody can read a `$SECRET` environment variable, but not the `/run/secrets/SECRET` file unless file ownership and permissions permit reading by nobody. For details, see chapter 12, which explores Docker configurations and secrets in detail.

Does it look like anything is missing from the `postgres` service definition? One thing that is not clear is how clients will connect to the database.

When using a Docker overlay network, applications connected to a given network will be able to communicate with each other on *any* port. No firewalls exist between applications attached to a Docker network. Because PostgreSQL listens on port 5432 by default and no firewall is present, other applications that are also attached to that private network will be able to connect to the `postgres` service on that port.

Now let's add a service definition for the API under the `services` key:

```

api:
  image: ${IMAGE_REPOSITORY:-dockerinaction/ch13_multi_tier_app}:api
  networks:
    - public
    - private
  ports:
    - '8080:80'
  secrets:
    - source: ch13_multi_tier_app-POSTGRES_PASSWORD
      target: POSTGRES_PASSWORD
      mode: 0400

```

```

environment:
  POSTGRES_HOST: 'postgres'
  POSTGRES_PORT: '5432'
  POSTGRES_USER: 'exercise'
  POSTGRES_DB: 'exercise'
  POSTGRES_PASSWORD_FILE: '/run/secrets/POSTGRES_PASSWORD'
depends_on:
  - postgres
deploy:
  replicas: 2
  restart_policy:
    condition: on-failure
    max_attempts: 10
    delay: 5s
  update_config:
    parallelism: 1
    delay: 5s
  resources:
    limits:
      cpus: '0.50'
      memory: 15M
    reservations:
      cpus: '0.25'
      memory: 15M

```

The API servers are attached to both the public and private network. Clients of the API server issue requests to port 8080 of the cluster. The Swarm network routing mesh will forward client requests from the edge of the network to a task and ultimately into an API server container on port 80. The API servers connect to PostgreSQL, which is attached to *only* the private network. The API servers are configured to connect to PostgreSQL by using the information defined in the `POSTGRES_*` environment variables.

Notice that the PostgreSQL user's password is also provided to the API server via a Docker secret. As with the `postgres` service, the secret is mounted into each API service container as a file. Though the API service uses an image built from scratch and includes only a static Golang binary, the secret mount still works because Docker manages the underlying `tmpfs` filesystem mount for you. Docker goes to great lengths to help you manage and use secrets safely.

The rest of the API service definition manages the specifics of how Swarm should deploy the service. The `depends_on` key contains a list of other services that the API server depends on—in this case, `postgres`. When we deploy the stack, Swarm will start the `postgres` service before `api`. The `deploy` key declares how Swarm should deploy the `api` service across the cluster.

In this configuration, Swarm will deploy two replicas to the cluster and try to keep that many tasks running to support the service. The `restart_policy` determines how Swarm handles a service task exiting or entering a failed state, according to its health check.

Here, Swarm will restart the task when it fails to start. *Restart* is a misnomer, as Swarm will actually start a new container rather than restart the failed container. Swarm

restarts service tasks an infinite number of times by default. The API service’s configuration restarts tasks up to 10 times with a 5-second delay between each restart.

Service authors should think through their restart strategies carefully to determine how long and how many attempts Swarm should make to start a service. First, it’s rarely useful to try indefinitely. Second, infinite retry processes could exhaust cluster resources that are consumed when new containers start, but aren’t cleaned up quickly enough.

The API service uses a simple `update_config` that limits the rollout of updates to the service to one task at a time. In this configuration, Swarm will update the service by shutting down a task with the old configuration, start one with the new configuration, and wait until the new task is healthy prior to moving on to replacing the next task in the service. The delay configuration introduces an interval between task replacement actions to keep the cluster and traffic to the service stable during the rollout.

Many configuration options exist for restart, update, and rollback configurations that were discussed in chapter 11. You can fine-tune these to complement the application’s behavior and create a robust deployment process.

13.2.3 Deploying the application

In this section, we will deploy the application we’ve defined to a Swarm cluster. We will use the `docker stack` command introduced in chapter 11 to do that. Figure 13.4 shows how this command will be communicated to the cluster.

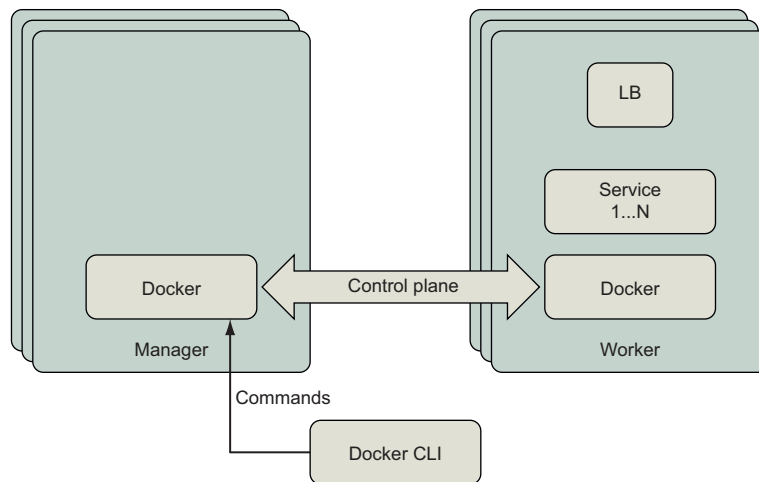


Figure 13.4 Communication path for Docker control plane

Docker services, networks, and other swarm resources are managed by issuing the appropriate `docker` command to a manager node of the swarm cluster. When you issue a command with the `docker` CLI, it will connect to the Docker Engine API and

request updates to the swarm cluster's state. The leader of the swarm will orchestrate the changes required to converge the actual application resources on the cluster to the desired state.

If you issue Docker commands to manage the cluster or its resources to a worker node, you will receive an error:

```
[worker1] $ docker node ls
Error response from daemon: This node is not a swarm manager. Worker
nodes can't be used to view or modify cluster state. Please run this
command on a manager node or promote the current node to a manager.
```

Open a command shell to any of the manager nodes in your cluster. List the cluster's nodes with `docker node ls`:

```
[manager1] $ docker node ls
ID                                HOSTNAME          STATUS
AVAILABILITY                      MANAGER STATUS  ENGINE VERSION
7baqi6gedujmycxwufj939r44 *    manager1        Ready
Active                             Reachable       18.06.1-ce
bbqicrevqkfu8w4f9wli1tjcr      manager2        Ready
Active                             Leader          18.06.1-ce
hdpskn4q93f5oulwhw9ht8y01      manager3        Ready
Active                             Reachable       18.06.1-ce
x1e0g72ydvj24sf40vnaw08n0      worker1         Ready
Active                             18.06.1-ce
l6fkyyzqglocnwc0y4va2anfho     worker2        Ready
Active                             18.06.1-ce
[manager1] $
```

In the preceding output, notice that the command was executed from the node named `manager1`. This node is operating in a manager role but is not currently the leader of the cluster. When a cluster management command is issued to this node, it will be forwarded to the leader for processing.

Use Git to clone the application to the manager node and change into the `ch13_multi_tier_app` directory:

```
git clone https://github.com/dockerinaction/ch13_multi_tier_app.git
cd ch13_multi_tier_app
```

We are now ready to deploy the application with `docker stack`. The `stack` subcommand can deploy applications defined in two formats. The first format is the Docker Compose format we will be using. The second is the older and less popular Distributed Application Bundle (DAB) format. Because we are using the Docker Compose format, we will specify the path(s) to the compose file with `--compose-file`. Let's deploy our Compose application to Swarm now:

```
docker stack deploy --compose-file docker-compose.yml multi-tier-app
```

The application deployment should fail with an error indicating that `ch13_multi_tier_app-POSTGRES_PASSWORD` was not found:

```
$ docker stack deploy --compose-file docker-compose.yml multi-tier-app
Creating network multi-tier-app_private
Creating network multi-tier-app_public
service postgres: secret not found: ch13_multi_tier_app-POSTGRES_PASSWORD
```

The docker command output shows Swarm was able to create the networks, but not the services. Swarm requires that all cluster-level resources that a service depends on exist prior to proceeding with the deployment. So Docker halted the application deployment when it determined a resource dependency was missing. The resources that were created have been left as is and can be used in subsequent deployment attempts. These predeployment checks help build robust application delivery processes. The fail-fast deployment behavior helped us quickly discover a missing dependency.

The missing cluster-level resource that this application depends on is the `ch13_multi_tier_app-POSTGRES_PASSWORD` secret. Recall that the application's reference to that secret said that it was defined externally:

```
secrets:
  ch13_multi_tier_app-POSTGRES_PASSWORD:
    external: true                # Retrieve from cluster's secret resources
```

In this context, `external` means defined outside the application deployment definition and provided by Swarm. Let's store the application's database password as a Docker secret in the Swarm cluster now:

```
echo 'mydbpass72' | docker secret create \
  ch13_multi_tier_app-POSTGRES_PASSWORD -
```

NOTE The only place this password is defined is in this Docker secret managed by Swarm. You can use any valid PostgreSQL password you want. Feel free to change it. This demonstrates how easy it is to safely handle secrets in distributed applications with Swarm.

The `docker secret` command should succeed and print the random identifier that Docker assigned to manage the secret. You can verify that the secret was created by listing the secrets in your cluster:

```
docker secret ls --format "table {{.ID}} {{.Name}} {{.CreatedAt}}"
```

←
Formats output as a table with secret identifier, name, and time since creation (optional)

The listing should show the secret was created recently:

```
ID NAME CREATED
<random id> ch13_multi_tier_app-POSTGRES_PASSWORD 6 seconds ago
```

Now let's try deploying the stack again:

```
[manager1] $ docker stack deploy \
  --compose-file docker-compose.yml multi-tier-app
Creating service multi-tier-app_postgres
Creating service multi-tier-app_api
```

The `docker stack` command should report that it has created two Docker services for the multitier app: `multi-tier-app_postgres` and `multi-tier-app_api`. List the services and check their status:

```
docker service ls \
  --format "table {{.Name}} {{.Mode}} {{.Replicas}}"
```

Formats output as a table with service name, mode, and replicas (optional) ←

That command will produce output like this:

```
NAME MODE REPLICAS
multi-tier-app_api replicated 2/2
multi-tier-app_postgres replicated 1/1
```

Each of the services has the expected number of replicas. There is one task for PostgreSQL and two tasks for the API shown in the `REPLICAS` column.

You can check that the `api` service started up correctly by inspecting logs:

```
docker service logs --follow multi-tier-app_api
```

Each `api` task should log a message saying that it is initializing, reading the PostgreSQL password from a file, and listening for requests. For example:

```
$ docker service logs --no-task-ids multi-tier-app_api
multi-tier-app_api.1@worker1 | 2019/02/02 21:25:22 \
  Initializing api server
multi-tier-app_api.1@worker1 | 2019/02/02 21:25:22 \
  Will read postgres password from '/run/secrets/POSTGRES_PASSWORD'
multi-tier-app_api.1@worker1 | 2019/02/02 21:25:22 \
  dial tcp: lookup postgres on 127.0.0.11:53: no such host
multi-tier-app_api.1@worker1 | 2019/02/02 21:25:23 \
  dial tcp 10.0.0.12:5432: connect: connection refused
multi-tier-app_api.1@worker1 | 2019/02/02 21:25:25 \
  Initialization complete, starting http service
multi-tier-app_api.2@manager1 | 2019/02/02 21:25:22 \
  Initializing api server
multi-tier-app_api.2@manager1 | 2019/02/02 21:25:22 \
  Will read postgres password from '/run/secrets/POSTGRES_PASSWORD'
multi-tier-app_api.2@manager1 | 2019/02/02 21:25:22 \
  dial tcp: lookup postgres on 127.0.0.11:53: no such host
multi-tier-app_api.2@manager1 | 2019/02/02 21:25:23 \
  dial tcp: lookup postgres on 127.0.0.11:53: no such host
multi-tier-app_api.2@manager1 | 2019/02/02 21:25:25 \
  Initialization complete, starting http service
```

The `docker service logs <service name>` command streams log messages from the node where service tasks are deployed to your terminal. You may view the logs of any service by issuing this command to Docker Engine on a manager node, but not a worker. When you view service logs, Docker Engine connects to the engines in the cluster where its tasks have run, retrieves the logs, and returns them to you.

From the log messages, we can see that these `api` tasks appear to be running on the `worker1` and `manager1` nodes. Your service tasks may have started on different nodes. We can verify this with the `docker service ps` command, which lists a service's tasks. Run this:

```
docker service ps \
  --format "table {{.ID}} {{.Name}} {{.Node}} {{.CurrentState}}" \
  multi-tier-app_api
```

←
**Formats output as a table with
essential task data (optional)**

This command will produce output like this:

```
ID NAME NODE CURRENT STATE
5jk32y4agzst multi-tier-app_api.1 worker1 Running 16 minutes ago
nh5trkrpojlc multi-tier-app_api.2 manager1 Running 16 minutes ago
```

The `docker service ps` command reports that two tasks are running for the `api` service, as expected. Notice that the tasks are named in the form `<stack name>_<service name>.<replica slot number>`; for example, `multi-tier-app_api.1`. Each task also gets a unique ID. The `docker service ps` command lists the tasks and their status for a service no matter where they are running on the cluster.

By contrast, when running `docker container ps` on the `manager1` node, it shows only the single container running on that node:

```
$ docker container ps --format "table {{.ID}} {{.Names}} {{.Status}}"
CONTAINER ID NAMES STATUS
4a95fa59a7f8 multi-tier-app_api.2.nh5trkrpojlc3knysxza3sffl \
  Up 27 minutes (healthy)
```

The container name for a service task is constructed from the task name and unique task ID. Both `ps` commands report that this task is running and healthy. The `api` server's image defines a `HEALTHCHECK`, so we can be confident this is true.

Great—our application deployed successfully, and everything looks healthy!

Open a web browser and point it to port 8080 on *any* node of your cluster. Play with Docker users should have an 8080 hyperlink at the top of the web console. You can also use a `curl` command to issue an HTTP request from one of the cluster nodes to port 8080:

```
curl http://localhost:8080
```

The `api` server should respond with a simple message similar to the following:

```
Welcome to the API Server!
Container id 256e1c4fb6cb responded at 2019-02-03 00:31:23.0915026 +0000 UTC
```

TIP If you are using Play with Docker, the detail page for each cluster node will have a link to ports published on that node. You can open that link or use it with `curl`.

When you make that request several times, you should see different container IDs serving your requests. This shell script will issue four HTTP requests and produce the output that follows:

```

Bash shell commands to issue four requests to the application; you
  can replace "localhost" with the hostname of any cluster node.
$ for i in `seq 1 4`; do curl http://localhost:8080; sleep 1; done;
Welcome to the API Server!
Server 9c2eea9f140c responded at 2019-02-05 17:51:41.2050856 +0000 UTC
Welcome to the API Server!
Server 81fbc94415e3 responded at 2019-02-05 17:51:42.1957773 +0000 UTC
Welcome to the API Server!
Server 9c2eea9f140c responded at 2019-02-05 17:51:43.2172085 +0000 UTC
Welcome to the API Server!
Server 81fbc94415e3 responded at 2019-02-05 17:51:44.241654 +0000 UTC

```

← **Output from each HTTP request**

Here, the `curl` program issues an HTTP GET request to a cluster node. In the preceding example, the `curl` program runs on one of the cluster's nodes and sends the request to that node, `localhost`, on port 8080. As there are no firewalls preventing `curl` from opening a socket to that network location, Docker Swarm's service mesh will handle the connection to port 8080 and route the request to a live container. We will investigate how requests are routed to Docker services in more detail next.

13.3 **Communicating with services running on a Swarm cluster**

Docker makes it easy for clients outside a Swarm cluster to connect to services running in the cluster. Swarm also helps services running within the cluster to find and contact each other when they share a Docker network. In this section, we will first explore how Docker exposes services to the world outside the cluster. Then we will look at how Docker services communicate with each other by using Swarm's service discovery and overlay networking features.

13.3.1 **Routing client requests to services by using the Swarm routing mesh**

The Swarm *routing mesh* provides a simple way to expose a service running on a container cluster with the outside world, which is one of Swarm's most compelling features. The routing mesh combines several sophisticated network building blocks to publish a service port. Figure 13.5 depicts the logical network topology Swarm creates for the example application.

Swarm sets up a listener on each node of the cluster for each published service port. You can configure the port to listen for TCP, UDP, or both kinds of traffic. Client applications can connect to this port on any cluster node and issue requests.

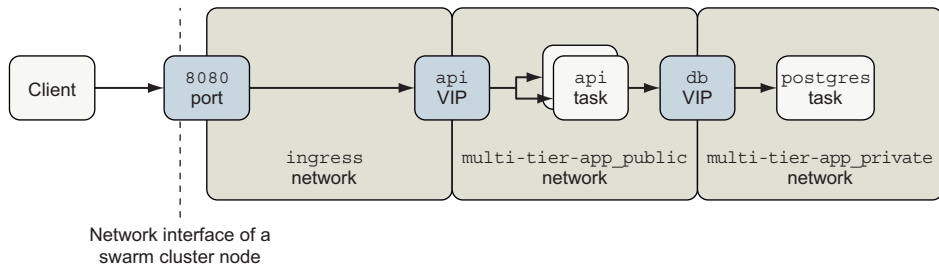


Figure 13.5 Swarm network components for example app

Swarm implements this listener with a combination of Linux `iptables` and `ipvs` features. An `iptables` rule redirects traffic to a dedicated virtual IP (VIP) allocated for the service. The service’s dedicated VIP is made available across the swarm cluster by using a Linux kernel feature called *IP Virtual Server*, `ipvs`. `IPVS` is a transport-layer load balancer that forwards requests for TCP or UDP services to their real endpoints. `IPVS` is not an application-layer load balancer for protocols such as HTTP. Swarm creates a VIP for each published Service port using `ipvs`. It then attaches the VIP to the ingress network, which is available across the Swarm cluster.

Returning to our example application, when traffic reaches a cluster node on TCP port 8080, `iptables` reroutes that traffic to the `api` service VIP attached to the ingress network. `IPVS` forwards traffic from the VIP to the ultimate endpoints, which are Docker service tasks.

Swarm’s routing mesh will handle the connection from the client, connect to a healthy service task, and forward the client’s request data to the service task. Figure 13.6 shows how Swarm routes `curl`’s HTTP request to an API service task and back.

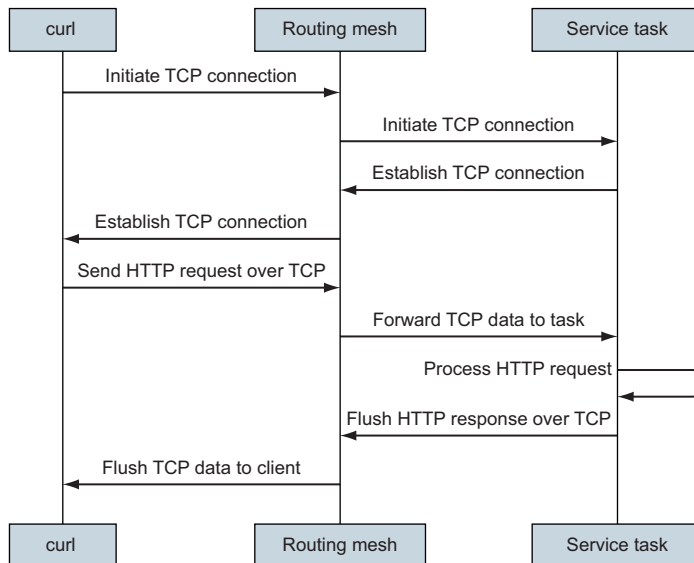


Figure 13.6 Routing an HTTP request to a service task

When a program connects to a published port, Swarm will attempt to connect to a healthy task for that service. If the service has been scaled to zero replicas or no healthy tasks exist, the routing mesh will refuse to initiate the network connection. Once the TCP connection is established, the client may move on to the next stage of transmission. In the case of the API service, the client writes an HTTP GET request onto the TCP socket connection. The routing mesh receives that data and sends it to the task handling this connection.

It's important to note that a service task does not need to be running on the node that handles the client's connection. Publishing a port establishes a stable ingress point for a Docker service that is independent of the transient locations of that service's tasks within the Swarm cluster. You can inspect the ports published by a service with `docker service inspect`:

```
$ docker service inspect --format="{json .Endpoint.Spec.Ports}" \
  multi-tier-app_api
[
  {
    "Protocol": "tcp",
    "TargetPort": 80,
    "PublishedPort": 8080,
    "PublishMode": "ingress"
  }
]
```

This output indicates that `multi-tier-app_api` has a listener attached to the `ingress` network on TCP port 8080, and that traffic will be routed into service tasks on port 80.

Bypassing the routing mesh

An alternate `PublishMode` called `host` bypasses the routing mesh and attachment to the `ingress` network. When using this mode, clients connect directly to the service task on a given host. If a task is deployed there, it can handle the connection; otherwise, the connection attempt will fail.

This `PublishMode` is likely most appropriate for services that are deployed in `global` mode so that there is one, and only one, task for a particular service on a cluster node. This ensures that a task is available to handle requests and avoids port collisions. Global services are explained in more detail in section 13.4.3.

Clients interact with the example application's API service by using HTTP. HTTP is an application protocol (layer 7) that is transported over the TCP/IP (layer 4) networking protocol. Docker also supports services that listen on UDP/IP (layer 4). The Swarm routing mesh relies on IPVS, which routes and balances network traffic at layer 4.

The distinction between routing at layer 4 versus layer 7 is important. Because Swarm routes and load-balances connections at the IP layer, it means client *connections* will be balanced across backend service tasks, not *HTTP requests*. When one client

issues many requests over a single connection, all of those requests will go to a single task, and will not be distributed across all backend service tasks as you might expect. Note that Docker Enterprise Edition supports load balancing of the HTTP protocol (layer 7), and third-party solutions exist as well.

13.3.2 Working with overlay networks

Docker Swarm offers a type of network resource called an *overlay network*, illustrated in figure 13.7. This network, whose traffic is logically segmented from other networks, runs on top of another network. The Docker Engines of a Swarm cluster can create overlay networks that connect containers running on different Docker hosts. In a Docker overlay network, only the containers attached to that network can communicate with other containers on that network. An overlay network isolates the communication between containers attached to that network from other networks.

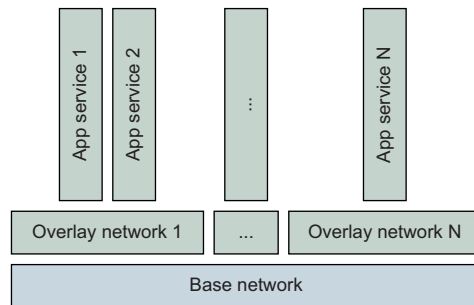


Figure 13.7 Layered view of overlay networks

One way to think about an overlay network is that it enhances the user-defined bridge networks described in chapter 5 to span Docker hosts. Just as with a user-defined bridge network, all containers attached to an overlay network can communicate with each other directly as peers. A special example of an overlay network is the ingress network.

The ingress network is a special-purpose overlay network that it is created by Docker when you initialize a swarm. The ingress network's only responsibility is to route traffic from external clients connected to ports published by Docker services within the cluster. This network is managed by Swarm, and only Swarm can attach containers to the ingress network. You should be aware that the default configuration of the ingress network is not encrypted.

If your application needs end-to-end encryption, all services that publish ports should terminate their connections with TLS. TLS certificates can be stored as Docker secrets and retrieved by services on startup, just as we have demonstrated with passwords in this chapter and TLS certificates in chapter 12.

Next, we will explore how Docker helps services discover and connect to each other on a shared network.

13.3.3 Discovering services on an overlay network

Docker services use the Domain Name System (DNS) to discover the location of other Docker services on a Docker network that they share. A program can connect to a Docker service if it knows the name of that service. In our example application, the `api` server is configured with the name of the database service via the `POSTGRES_HOST` environment variable:

```
api:
  # ... snip ...
  environment:
    POSTGRES_HOST: 'postgres'
```

When an `api` task creates a connection to the PostgreSQL database, it will resolve the `postgres` name to an IP by using DNS. Containers attached to a Docker overlay network are automatically configured by Docker to perform DNS lookups via a special resolver, 127.0.0.11. This is also true for user-defined bridge and MACVLAN networks. The Docker Engine handles DNS lookups made to 127.0.0.1. If the name resolution request is for a Docker service that is present on that network, Docker will respond with the location of that service's virtual IP. If the lookup is for another name, Docker will forward the request on to the normal DNS resolver for that container host.

In our example application, that means when the `api` service looks up `postgres`, the Docker Engine on that host will respond with the virtual IP of the `postgres` service endpoint; for example, 10.0.27.2. The `api` database connection driver can establish a connection to this virtual IP, and Swarm will route the connection to the `postgres` service task, which could be at 10.0.27.3. You may have expected this convenient name resolution and network routing functionality to exist, but not all container orchestrators work this way.

If you recall figure 13.5 shown previously, you may also have an explanation for something that looked unusual. Figure 13.8 reproduces that diagram here.

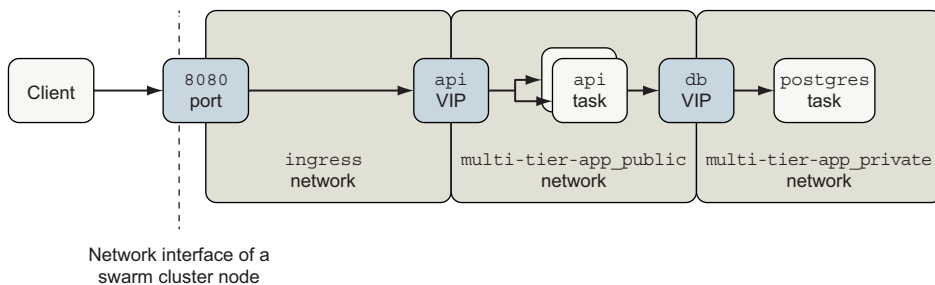


Figure 13.8 Swarm network components for example app

The `api` service has three virtual IPs establishing its presence on each of three overlay networks it is attached to: `ingress`, `multi-tier-app_public`, and `multi-tier-app_private`.

If you inspect the api service's endpoints, you should see output that verifies this with VirtualIPs on those three networks:

```
docker service inspect --format '{{ json .Endpoint.VirtualIPs }}' \
  multi-tier-app_api
[
  {
    "NetworkID": "5oruhwaq4996xfpdp194k82td", ← ingress network
    "Addr": "10.255.0.8/16"
  },
  {
    "NetworkID": "rah2lj4tw67lgn87of6n5nihc", ← multi-tier-app_private
    "Addr": "10.0.2.2/24"
  },
  {
    "NetworkID": "vc12njqthcq1shhqt4eph697", ← multi-tier-app_public
    "Addr": "10.0.3.2/24"
  }
]
```

Follow along with a little experiment that demonstrates the discoverability of services attached to a network and even the containers behind them. Start a shell and attach it to the multi-tier-app_private network:

```
docker container run --rm -it --network multi-tier-app_private \
  alpine:3.8 sh
```

We can attach our shell container to the application's private network because it was defined as attachable:

```
private:
  driver: overlay
  driver_opts:
    encrypted: "true"
  attachable: true
```

By default, only Swarm can attach containers for service tasks to a network. This private network was made attachable specifically for this service discovery exercise.

Ping the postgres service once. You should see output like this:

```
/ # ping -c 1 postgres
PING postgres (10.0.2.6): 56 data bytes
64 bytes from 10.0.2.6: seq=0 ttl=64 time=0.110 ms

--- postgres ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.110/0.110/0.110 ms
```

Now ping the api service:

```
/ # ping -c 1 api
PING api (10.0.2.2): 56 data bytes
64 bytes from 10.0.2.2: seq=0 ttl=64 time=0.082 ms
```

```
--- api ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.082/0.082/0.082 ms
```

Let's use Netcat to issue a request manually from your shell on the private network to the api service:

```
$ printf 'GET / HTTP/1.0\nHost: api\n\n' | nc api 80
```

← Creates an HTTP request and pipes through Netcat to the API

You should see output similar to that in the previous section:

```
HTTP/1.0 200 OK
Connection: close
Content-Type: text/plain; charset=utf-8
Date: Wed, 13 Feb 2019 05:21:43 GMT
Content-Length: 98

Welcome to the API Server!
Server 82f4ab268c2a responded at 2019-02-13 05:21:43.3537073 +0000 UTC
```

We successfully issued a request to the api service from a shell attached to the private network. This works because the api service is attached to the private network in addition to the public and ingress networks. You can also connect to the PostgreSQL DB from your shell:

```
/ # nc -vz postgres 5432
postgres (10.0.2.6:5432) open
```

This Netcat command opens a socket to the postgres hostname on port 5432 and then closes it right away. Netcat's output indicates that it succeeded in connecting to the postgres VIP, 10.0.2.6. This might surprise you. After all, if you review the postgres service definition, you can confirm that we never published or exposed any ports. What's going on here?

Communication between containers attached to a given Docker network is *completely* open. There are *no* firewalls between containers on a Docker overlay network. Because the PostgreSQL server is listening on port 5432 and is attached to the private network, any other container attached to that network can connect to it.

This behavior might be convenient in some cases. However, you may need to approach access control between connected services differently than you are accustomed to. We will discuss some ideas for isolating service-to-service communications next.

13.3.4 *Isolating service-to-service communication with overlay networks*

Many people control access to a service by restricting the network connections that can be made to that service. For example, it is common to use a firewall that permits traffic to flow from service A to service B, but not permit traffic in the reverse direction from B to A. This approach does not translate well to Docker overlay networks because there are no firewalls between peers connected to a given network. Traffic

flows freely in both directions. The only access-control mechanism available for an overlay network is attachment (or not) to the network.

However, you can achieve substantial isolation of application traffic flows with Docker overlay networks. Overlay networks are lightweight and easy to create with Swarm so they can be used as a design tool to create secure application communication topologies. You can use fine-grained, application-specific networks for your application deployments and avoid sharing services to achieve isolation. The example application demonstrates this approach, with the exception of making the private network attachable.

The key point to remember is that while traffic flows on a tightly scoped network may be isolated to a few containers, there is no such thing as using a network identity to authenticate and authorize traffic. When an application needs to control access to its functionality, the application *must* verify the identity and authorization of clients at the application level. The example application controls access to the postgres database by using the PostgreSQL user and password. This ensures that only the api service can interact with the database in our deployment. The api service is meant to be used anonymously, so it does not implement authentication, but it certainly could.

One challenge you may run into is integrating centralized, shared services such as a logging service. Suppose an application such as our example and a centralized logging service are attached to a shared network. Docker networks would enable the logging service to contact the api or postgres service if it (or an attacker) chooses to do so.

The solution to this problem is to deploy the centralized logging service or other shared services as a Docker service that publishes a port. Swarm will set up a listener for the logging service on the ingress network. Clients running inside the cluster can connect to this service like any other published service. Connections from tasks and containers running inside the cluster will be routed to the logging service as described in section 13.3.1. Because the logging service's listener will be available on every node of the Swarm cluster, the logging service should authenticate its clients.

Let's demonstrate this idea with a simple echo service that replies with whatever input you send it. First create the service:

```
docker service create --name echo --publish '8000:8' busybox:1.29 \
  nc -v -lk -p 8 -e /bin/cat
```

If you send data to the echo service using port 8000 of a cluster node using Netcat (nc):

```
echo "Hello netcat my old friend, I've come to test connections again." \
| nc -v -w 3 192.168.1.26 8000
```

← Replace 192.168.1.26 with the IP of one of your cluster's nodes or use \$(hostname -i) to substitute the current host IP on Linux.

Netcat should print a response similar to this:

```
192.168.1.26 (192.168.1.26:8000) open
Hello netcat my old friend, I've come to test connections again.
```

Clients should connect to shared services by using a port published by that service. Switch to or reopen the shell we created in the previous section so we can verify a few things:

```
docker container run --rm -it --network multi-tier-app_private \
  alpine:3.8 sh
```

Then, if you try to ping the echo service, the ping will report an error:

```
/ $ ping -c 1 echo
ping: bad address 'echo'
```

The same occurs with nslookup when trying to resolve the hostname echo:

```
/ $ nslookup echo
nslookup: can't resolve '(null)': Name does not resolve
```

The echo service's name doesn't resolve when attached to the `multi-tier-app_private` network. The `api` service needs to connect to the port published by the echo service at the cluster's edge, just like processes running outside the Swarm cluster. The only route to the echo service is through the `ingress` network.

We can say a few good things about this design. First, all clients reach the echo service in a uniform way, through a published port. Second, because we didn't join the echo service to any networks (besides the implicit `ingress` network join), it is isolated and cannot connect to other services, except for those that are published. Third, Swarm has pushed application authentication responsibilities into the application layer, where they belong.

One of the main implications with this design is that an application described with Docker Compose may rely on two sets of names for services and their locations. First, some services are scoped to and defined within the application's deployment (for example, `api` depends on `postgres`). Second, there are services such as the echo service that an application may depend on, but that are managed with a different deployment life cycle and have a different scope. These latter services may be shared by many applications. This second kind of service needs to be registered with a registry such as the corporate-wide DNS so that applications can discover its location. Next we will examine how client connections are balanced behind a service VIP after its location has been discovered.

13.3.5 *Load balancing*

Let's explore how Docker client connections are balanced across a Docker service's tasks. Clients usually connect to Docker services through a virtual IP. Docker services have a property called `endpoint-mode` that defaults to `vip`. We have been using this default `vip` endpoint mode for all of our examples so far. When a service uses the `vip` endpoint mode, clients will access the service through the VIP. Connections to that VIP will be load-balanced automatically by Docker.

For example, in section 13.3.3, we attached a shell to the `multi-tier-app_private` network and used Netcat to issue an HTTP request to the `api`. When Netcat resolved the `api` hostname to an IP, Docker's internal DNS replied with the VIP of the `api` service. In that case, more than one healthy service task was available. Docker's network routing implementation is responsible for distributing connections to the service VIP equally between the healthy tasks behind the VIP.

Docker's network-based load-balancing implementation is used for all traffic routed through a VIP endpoint. That traffic could be from an internal overlay network or come in through a port published to the `ingress` network.

Docker does not guarantee which service task will handle a client's request. Even when a client is running on the same node as a healthy service task, the client's request may go to a healthy task on another node. This is true even for services deployed in `global` mode (distinct from `endpoint` mode), where an instance runs on each cluster node.

13.4 Placing service tasks on the cluster

In this section, we will investigate how Swarm places tasks around the cluster and tries to run the desired number of service replicas within declared constraints. First, we will introduce the coarse-grained controls Swarm has for managing task placement. Then we'll show you how to control the placement of tasks by using affinity and anti-affinity to built-in and operator-specified node labels.

We will use a five-node swarm cluster created from a Play with Docker template, depicted in figure 13.9.

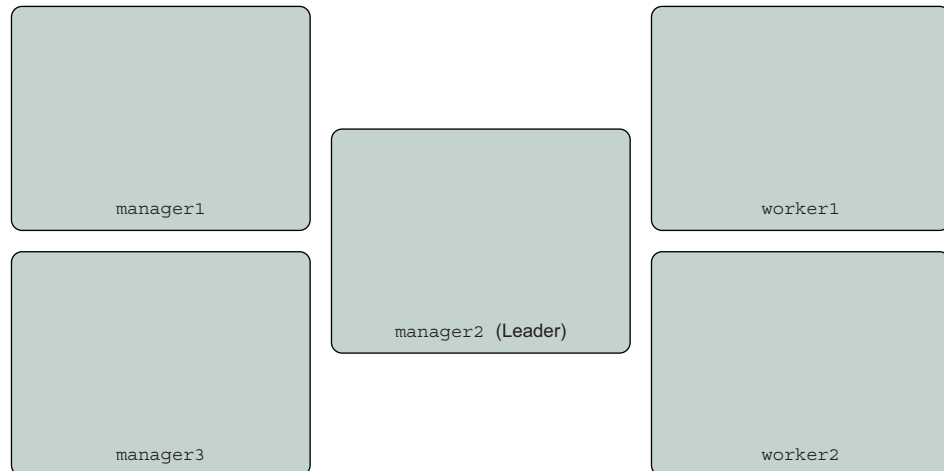


Figure 13.9 The test Swarm cluster

This cluster has three manager nodes and two worker nodes, which are named:

- manager1
- manager2
- manager3
- worker1
- worker2

13.4.1 Replicating services

The default, and most commonly used, deployment mode for a Docker service is *replicated*. Swarm will try to keep the number of replicas specified in a service’s definition running at all times. Swarm continually reconciles the desired state of the service specified by the Docker Compose definition or `docker service` command, and the state of the service’s tasks on the cluster. This reconciliation loop, illustrated in figure 13.10, will continuously start or stop tasks to match so that the service has the desired number of healthy replicas.

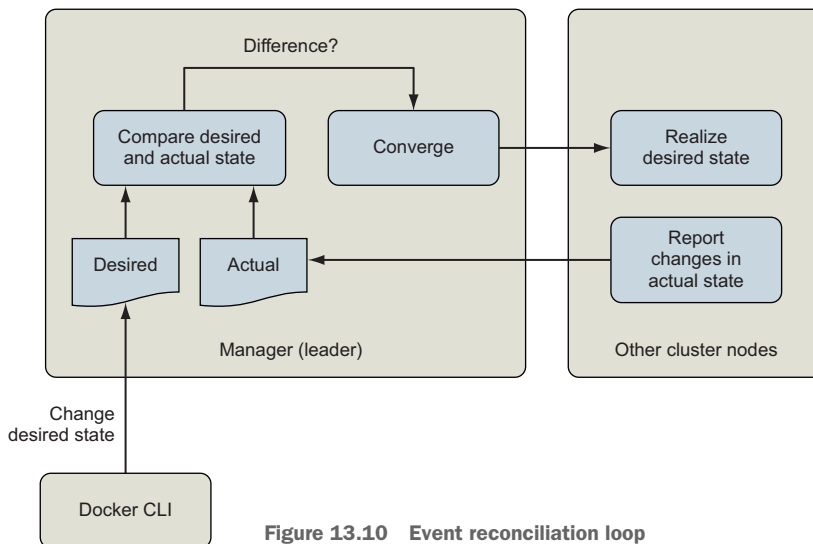


Figure 13.10 Event reconciliation loop

Replicating a service is useful because you can scale the service to as many replicas as needed to handle the load and that your cluster has resources to support.

In this mode, Swarm will schedule a service task to start on a cluster node that has sufficient compute resources (memory, CPU) and that satisfies the service’s labelled constraints. Swarm tries to spread the service’s tasks across the cluster’s nodes. This strategy is helpful for improving service availability and smoothing load across nodes. We will control where tasks run in the next section. For now, let’s see what happens when we start scaling our example application’s `api` service.

The api service is configured to have two replicas by default. The deployment definition also reserves and limits the CPU and memory resources that each container can use:

```

deploy:
  replicas: 2
  restart_policy:
    condition: on-failure
    max_attempts: 10
    delay: 5s
  update_config:
    parallelism: 1
    delay: 5s
  resources:
    limits:
      cpus: '0.50'
      memory: 15M
    reservations:
      cpus: '0.25'
      memory: 15M

```

When Swarm schedules each api task, it will look for a node with at least 15 MB of memory and 0.25 CPUs that have not been reserved for other tasks. Once a node with sufficient resources has been identified, Swarm will create a container for the task that is limited to (again) 15 MB of memory and may use up to 0.5 CPUs.

In aggregate, the api service begins with two replicas that reserve a total of 0.5 CPUs and 30 MB of memory. Now let's scale up our service a bit with five replicas:

```
docker service scale multi-tier-app_api=5
```

The service now reserves 75 MB of memory and 1.25 CPUs in aggregate. Swarm was able to find resources for the api service's tasks and spread them across this cluster:

```

$ docker service ps multi-tier-app_api \
  --filter 'desired-state=running' \
  --format 'table {{.ID}} {{.Name}} {{.Node}} {{.CurrentState}}'

```

ID	NAME	NODE	CURRENT STATE
dekzyqgcc7fs	multi-tier-app_api.1	worker1	Running 4 minutes ago
3el58dg6yewv	multi-tier-app_api.2	manager1	Running 5 minutes ago
qqc72ylzi34m	multi-tier-app_api.3	manager3	Running about a minute ago
miyugogsv2s7	multi-tier-app_api.4	manager2	Starting 4 seconds ago
zrpl00aua29y	multi-tier-app_api.7	worker1	Running 17 minutes ago

Now let's demonstrate what it looks like when a service reserves all of a cluster's resources. You should follow along only if you are using a cluster where it is OK for you to exhaust cluster resources and prevent other tasks from being scheduled. We'll undo all of this when we're done, but at some point, no new tasks that reserve CPU can be scheduled. We recommend that you do not run this resource exhaustion exercise on Play with Docker (PWD) because the underlying machines are shared by everyone using PWD.

First, let's increase the CPU reservation of our api tasks from a quarter of a CPU to an entire CPU:

```
docker service update multi-tier-app_api --reserve-cpu 1.0 --limit-cpu 1.0
```

You will see Docker shuffling tasks around as it re-creates the containers for each task with the new limits on a node with capacity.

Now let's try scaling the service to a larger number of replicas that will exhaust the cluster's available resources. For example, if you're running a five-node cluster and each node has 2 CPUs, then there should be 10 CPUs reservable in total.

The following output comes from a cluster with 10 reservable CPUs. The postgres service has reserved 1 CPU. The api service can be scaled successfully to 9 replicas:

```
$ docker service scale multi-tier-app_api=9
multi-tier-app_api scaled to 9
overall progress: 9 out of 9 tasks
1/9: running [=====>]
... snip ...
9/9: running [=====>]
verify: Service converged
```

All 10 CPUs are now reserved by the api and postgres services. When scaling the service to 10 replicas, the docker program appears to hang:

```
docker service scale multi-tier-app_api=10
multi-tier-app_api scaled to 10
overall progress: 9 out of 10 tasks
1/10: running [=====>]
... snip ...
10/10: no suitable node (insufficient resources on 5 nodes) <←
                                                    Insufficient resources to
                                                    create the 10th api task
```

The output reports there are insufficient resources on the cluster's five nodes to launch a 10th task. The trouble occurs when Swarm tries to schedule a task for the 10th api service task slot. When you run the cluster out of reservable resources, you will need to interrupt the docker stack deploy command with a `^C` keystroke to get your terminal back or wait for the command to time out. The Docker command will suggest that you run `docker service ps multi-tier-app_api` to get more info and check whether the service converges.

Go ahead and do that now and verify that api tasks are distributed across all of the cluster nodes and Swarm is unable to schedule the last task. In this case, we know the cluster will never converge unless we increase cluster capacity or reduce the desired number of replicas. Let's revert our changes.

Autoscaling services

Docker Swarm does not support autoscaling services by using built-in functionality. Third-party solutions can use resource usage metrics such as CPU or memory utilization or application-level metrics such as HTTP requests per task. The Docker Flow project is a good place to start, <https://monitor.dockerflow.com/auto-scaling/>.

We have a few options for reverting the scaling changes. We can redeploy our stack from its source definition, roll back the service configuration changes with the `docker service rollback` subcommand, or “roll forward” and set the service scale directly to something that will work. Try rolling back:

```
$ docker service rollback multi-tier-app_api
multi-tier-app_api
rollback: manually requested rollback
overall progress: rolling back update: 9 out of 9 tasks
... snip ...
verify: Service converged
```

The `service rollback` subcommand shifts a service’s desired configuration back by one version. The previous configuration of `multi-tier-app_api` had nine replicas. You can confirm that this configuration has taken effect by running `docker service ls`. The output should show that the `multi-tier-app_api` service has the pre-exhaustion number of replicas running; for example, 9/9. You might wonder what will happen if you run `rollback` again. If you execute another `rollback`, Docker will restore the config with 10 service replicas, exhausting resources again. That is, Docker will roll back the `rollback`, leaving us where we started. Since we’d like to undo multiple changes, we’ll need another method.

The cleanest approach in our case is to redeploy the service from its source definition:

```
docker stack deploy --compose-file docker-compose.yml multi-tier-app
```

Take a look at the service’s tasks with `docker service ps` to ensure that the service has returned to the state declared in the Docker Compose application definition:

```
docker service ps multi-tier-app_api \
  --filter 'desired-state=running' \
  --format 'table {{.ID}} {{.Name}} {{.Node}} {{.CurrentState}}'
ID          NAME                NODE          CURRENT STATE
h0to0a2l8m87 multi-tier-app_api.1 worker1       Running about a minute ago
v6sq9m14q3tw multi-tier-app_api.2 manager2      Running about a minute ago
```

The manual scaling changes are gone. There are two `api` tasks, as expected.

Notice that one task is running on the `worker1` node and the other is running on `manager2` nodes. This isn’t really the task placement we’d like for most deployments. Usually, we’d like to implement architectural goals like these:

- Reserve manager nodes for running the Swarm control plane so they have dedicated compute resources
- Isolate services that publish ports because they are easier to attack than private services

We’ll accomplish these goals and more using Swarm’s built-in features for constraining where tasks run in the next section.

13.4.2 Constraining where tasks run

We often want to control which nodes in a cluster that an application runs on. We might want to do this in order to isolate workloads into different environments or security zones, take advantage of special machine capabilities such as GPUs, or reserve a set of nodes for a critical function.

Docker services provide a feature called *placement constraints* that allow you to control the nodes a service's tasks can be assigned to. With placement constraints, you can say where service tasks should or should not run. The constraints can use both built-in and user-defined properties of your cluster nodes. We'll work through examples of each.

In the previous section, we saw that the `api` service was distributed to all nodes when scaled up. The `api` service ran on manager nodes as well on the same node as the `postgres` database, as shown in figure 13.11.

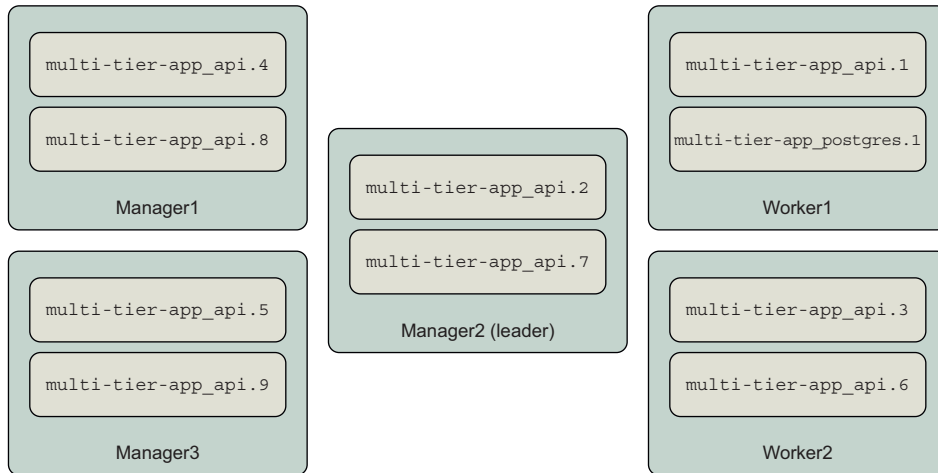


Figure 13.11 API service tasks are everywhere

Many system architects would adjust this deployment architecture so that manager nodes are dedicated to the running Swarm cluster. This is a good idea for important clusters because if a service consumes resources such as CPU, Swarm might fall behind in supervising tasks and responding to operational commands that would affect the operation of all services on the cluster. Also, because Swarm managers control the cluster, access to those nodes (and the Docker Engine API) should be controlled tightly. We can use Swarm's node availability and service placement constraints to do this.

Let's start by ensuring that our services do *not* run on manager nodes. All nodes in a Swarm cluster are available to run service tasks by default. However, we can reconfigure the availability of nodes by using the `docker node update` command's `--availability` option. There are three availability options: `active`, `pause`, and `drain`. The `active`

option means the schedule can assign new tasks to the node. The pause option means existing tasks will continue to run, but no new tasks will be scheduled to the node. The drain option means existing tasks will be shut down and restarted on another node, and no new tasks will be scheduled to that node.

So we can set the availability of the manager nodes to drain to keep service tasks from running on them:

```
docker node update --availability drain manager1
docker node update --availability drain manager2
docker node update --availability drain manager3
```

Once you run those commands, the output of `docker node ls` should reflect the changes in availability:

```
docker node ls --format 'table {{ .ID }} {{ .Hostname }} {{ .Availability }}'
ID                               HOSTNAME AVAILABILITY
ucetqsmbh23vuk6mwy9itv3xo       manager1 Drain
b0jajao5mkzdd3ie91qltewvj       manager2 Drain
kxfab99xvqv71tm39zbeveglj       manager3 Drain
rbw0c466qqi0d7k4niw0lo3nc       worker1 Active
u2382qjg6v9vr8z5lfwqrg5hf       worker2 Active
```

We can verify that Swarm has migrated the multi-tier-app service tasks to the worker nodes:

```
docker service ps multi-tier-app_api multi-tier-app_postgres \
  --filter 'desired-state=running' \
  --format 'table {{ .Name }} {{ .Node }}'
NAME                               NODE
multi-tier-app_postgres.1         worker2
multi-tier-app_api.1              worker1
multi-tier-app_api.2              worker2
```

If you run `docker container ps` on the manager nodes, you should not see any containers related to service tasks, either.

Placement constraints work by expressing that a service either should or should not run on a node based on certain metadata. The general form of a constraint is as follows:

```
<node attribute> equals or does not equal <value>
```

When a service is constrained to running in a node, we say it has *affinity* for that node. When it must not run on a node, we say it has *anti-affinity* for that node. You will see these terms used throughout this and other discussions of service placement. Swarm's constraint language denotes equality (a match) with `==`, and inequality with `!=`. When a service defines multiple constraints, a node must satisfy all constraints for the task to be scheduled there. That is, multiple constraints are AND'd together. For example, suppose you want to run a service on swarm worker nodes that are not in the public security zone. Once you have configured the cluster's zone metadata, you could achieve

this by running the service with these constraints: `node.role == worker` and `node.labels.zone != public`.

Docker supports several node attributes that can be used as the basis of constraints:

- `node.id`—The unique identifier for the node in the Swarm cluster (for example, `ucetqsbh23vuk6mwy9itv3xo`)
- `node.hostname`—The node’s hostname, (for example, `worker2`)
- `node.role`—The node’s role in the cluster, either manager or worker
- `node.labels.<label name>`—A label applied to the node by an operator (for example, a node with a `zone=public` label would have a node attribute of `node.labels.zone=public`)
- `engine.labels`—A set of labels describing key properties of the node and Docker Engine such as Docker version and operating system (for example, `engine.labels.operatingsystem==ubuntu 16.04`)

Let’s continue organizing our system by separating the worker nodes of our cluster into a public and a private zone. Once we have these zones, we will update the `api` and `postgres` services so their tasks run only in the desired zone.

You can label Swarm cluster nodes with your own metadata by using the `docker node update` command’s `--label-add` option. This option accepts a list of key/value pairs that will be added to the node’s metadata. There is also a `--label-rm` option to remove metadata from a node. This metadata will be available for use in constraining tasks to particular nodes.

Let’s identify `worker1` as part of the private zone, and `worker2` as part of the public zone:

```
$ docker node update --label-add zone=private worker1
worker1
$ docker node update --label-add zone=public worker2
worker2
```

Now constrain the `api` service to the public zone. The `docker service create` and `update` commands have options to add and remove task-scheduling constraints, `--constraint-add` and `--constraint-rm`, respectively. The constraint we added to the service tells Swarm to schedule only `api` service tasks on nodes with a zone label that equals `public`:

```
docker service update \
  --constraint-add 'node.labels.zone == public' \
  multi-tier-app_api
```

If all goes well, Docker will report that the `api` service’s tasks have converged to the new state:

```
multi-tier-app_api
overall progress: 2 out of 2 tasks
1/2: running [=====>]
```



```
2/2: running  [======>]
verify: Service converged
```

You can verify that the api tasks have been rescheduled to the worker2 node:

```
docker service ps multi-tier-app_api \
  --filter 'desired-state=running' \
  --format 'table {{ .Name }} {{ .Node }}'
NAME NODE
multi-tier-app_api.1 worker2
multi-tier-app_api.2 worker2
```

Unfortunately, we can't display node label information in `docker service ps` output nor see the labels we've added with `docker node ls`. Currently, the only way to see a node's labels is to inspect the node. Here's a quick bash shell script to show the host-name, role, and label information for all nodes in a cluster:

```
for node_id in `docker node ls -q | head`; do
  docker node inspect \
    --format '{{ .Description.Hostname}} {{ .Spec.Role}} {{ .Spec.Labels}}' \
    "${node_id}";
done;
```

This script should output the following:

```
manager1 manager map[]
manager2 manager map[]
manager3 manager map[]
worker1 worker map[zone:private]
worker2 worker map[zone:public]
```

This isn't great, but it's better than trying to recall which nodes have which labels.

The final adjustment we need to make to this system is to relocate the postgres database to the private zone. Before doing that, issue some queries to the api service's `/counter` endpoint with `curl`:

```
curl http://127.0.0.1:8080/counter
```

The `/counter` endpoint inserts a record into a table with an auto-incrementing `id` column. When the api service responds, it prints out all of the IDs in the column. If you issue three requests to the endpoint, the output of the third response should be similar to the following:

```
# curl http://127.0.0.1:8080/counter
SERVER: c098f30dd3c4
DB_ADDR: postgres
DB_PORT: 5432
ID: 1
ID: 2
ID: 3
```

This may seem like a bit of a diversion, but inserting these records will help demonstrate a key point in a moment.

Let's constrain the postgres task to the private zone:

```
$ docker service update --constraint-add 'node.labels.zone == private' \
  multi-tier-app_postgres
multi-tier-app_postgres
overall progress: 1 out of 1 tasks
1/1: running [=====>]
verify: Service converged
```

The postgres task is now running on the worker1 node:

```
$ docker service ps multi-tier-app_postgres \
  --filter 'desired-state=running' \
  --format 'table {{ .Name }} {{ .Node }}'
NAME                NODE
multi-tier-app_postgres.1 worker1
```

Now, if you issue a request to the /counter endpoint, you will see this:

```
$ curl http://127.0.0.1:8080/counter
SERVER: c098f30dd3c4
DB_ADDR: postgres
DB_PORT: 5432
ID: 1
```

The counter has been reset. Where did our data go? It was lost because the postgres database used a db-data volume that is local to a cluster node. Strictly speaking, the data wasn't lost. If the postgres task migrates back to the worker2 node, it will mount the original volume and resume counting from 3. If you were following along on your own cluster and didn't notice data loss, it's probably because postgres happened to deploy to worker1 to start with. This lack of determinism and potential for data loss is not a good situation. What can we do about it?

The default Docker volume storage driver uses the node's storage. This storage driver does not share or back up data across the Swarm cluster. There are Docker storage drivers that add features like this, including Docker CloudStor and Rex-Ray. Those drivers will enable you to create and share a volume across the cluster. You should investigate and test these drivers carefully before committing important data to them.

Another approach to ensure that a task runs consistently on a given node is to constrain it to a specific node. Relevant constraint options are node hostname, node ID, or a user-defined label. For now, let's constrain the postgres task to the worker1 node to ensure that it doesn't move from that node, even if the private zone expands:

```
docker service update --constraint-add 'node.hostname == worker1' \
  multi-tier-app_postgres
```

Now, the postgres service will not move from that node. Inspection of the service's placement constraints shows that two are active:

```
$ docker service inspect \
  --format '{{json .Spec.TaskTemplate.Placement.Constraints }}' \
  multi-tier-app_postgres
["node.hostname == worker1", "node.labels.zone == private"]
```

If we wanted to continue with these placement constraints, we would specify these in the example application's docker-compose.yml. Here is how to express the postgres service's constraints:

```
services:
  postgres:
    # ... snip ...
    deploy:
      # ... snip ...
      placement:
        constraints:
          - node.labels.zone == private
          - node.hostname == worker1
```

Now the placement constraints won't be lost the next time we deploy the application by using docker stack deploy.

NOTE You will need to perform a lot of careful engineering in order to safely run databases with important data on container clusters such as Swarm. The exact strategy will likely be specific to the database implementation so that you can take advantage of the database's specific replication, backup, and data recovery strengths.

Now that we have explored how to constrain services to particular nodes, let's go in the complete opposite direction and deploy a service everywhere with global services.

13.4.3 Using global services for one task per node

You can deploy one service task to each node in the Swarm cluster by declaring the service's mode to be global. This is useful when you want to scale a service along with the size of the cluster. Common use cases include logging and monitoring services.

Let's deploy a second instance of our echo service as a global service:

```
docker service create --name echo-global \
  --mode global \
  --publish '9000:8' \
  busybox:1.29 nc -v -lk -p 8 -e /bin/cat
```

Uses "global" instead of "replicated"

Publishes port 9000 to avoid collision with echo service

If you run `docker service ls`, you will see that the `echo-global` service is operating in global mode. The mode of the other services we have deployed so far will be replicated, the default.

You can verify that Swarm has deployed one task on each node that is available for task scheduling. This example uses the previous section's Swarm cluster, in which only the worker nodes are available for tasks. The `docker service ps` command confirms there is one task on each of those nodes:

```
docker service ps echo-global \
  --filter 'desired-state=running' \
  --format 'table {{ .Name }} {{ .Node }}'
NAME                               NODE
echo-global.u2382qjg6v9vr8z51fwqrg5hf worker2
echo-global.rbw0c466qqi0d7k4niw01o3nc worker1
```

You can interact with `echo-global` service just as you did the `echo` service. Send a few messages by using the following command:

```
[worker1] $ echo 'hello' | nc 127.0.0.1 -w 3 9000 ← Sends to echo-global service
                                                    using published port
```

Remember that client connections will be routed through the service's virtual IP (see section 13.3.5). Because of Docker networking's routing behavior, a client of a global service may connect to a task on another node instead of its own. The probability of connecting to a global service task on another node goes up with size of the cluster because connections are balanced uniformly. You can see the connection-based load balancing happening if you inspect the logs with `docker service logs --follow --timestamps echo-global` and send messages to the service.

The following output was produced by connecting to `worker1` and sending messages 1 second apart:

```
2019-02-23T23:51:01.042747381Z echo-global.0.rx3o7rgl6gm9@worker2 |
connect to [::ffff:10.255.0.95]:8 from [::ffff:10.255.0.3]:40170
([::ffff:10.255.0.3]:40170)
2019-02-23T23:51:02.134314055Z echo-global.0.hp01yak2txv2@worker1 |
connect to [::ffff:10.255.0.94]:8 from [::ffff:10.255.0.3]:40172
([::ffff:10.255.0.3]:40172)
2019-02-23T23:51:03.264498966Z echo-global.0.rx3o7rgl6gm9@worker2 |
connect to [::ffff:10.255.0.95]:8 from [::ffff:10.255.0.3]:40174
([::ffff:10.255.0.3]:40174)
2019-02-23T23:51:04.398477263Z echo-global.0.hp01yak2txv2@worker1 |
connect to [::ffff:10.255.0.94]:8 from [::ffff:10.255.0.3]:40176
([::ffff:10.255.0.3]:40176)
2019-02-23T23:51:05.412948512Z echo-global.0.rx3o7rgl6gm9@worker2 |
connect to [::ffff:10.255.0.95]:8 from [::ffff:10.255.0.3]:40178
([::ffff:10.255.0.3]:40178)
```

The `nc` client program sending messages was running on `worker1`. This log output shows that the client's connections that were routed bounce between the task on `worker2`, with IP ending in `.95`, and the task on `worker1`, with IP ending in `.94`.

13.4.4 Deploying real applications onto real clusters

The preceding exercises have shown how Swarm tries to converge an application's actual deployed resources to the desired state indicated in the application's deployment descriptor.

The desired state of the cluster changes as applications are updated, cluster resources such as nodes and shared networks are added or removed, or new configurations and secrets are provided by operators. Swarm processes these events and updates state in the internal log replicated among the cluster's manager nodes. When Swarm sees an event that changes the desired state, the leader of the managers issues commands to the rest of the cluster to converge to the desired state. Swarm will converge to the desired state by starting or updating service tasks, overlay networks, and other resources within the constraints specified by operators.

You may be wondering which features of Swarm to start with. First, make an inventory of the kinds of applications you want to run and the types of Swarm resources they will need. Second, think about how you want to organize those applications running on the cluster. Third, decide whether the cluster will support stateful services such as databases and determine a strategy for managing data safely.

Remember, you can start by deploying stateless services that use only the networking, configuration, and secret management features of Swarm. This approach provides the opportunity to learn more about Swarm and the way services operate in a multihost environment without putting data at risk.

With thoughtful design and proactive monitoring of the cluster's resources, you can ensure that applications have the resources they need, when they need it. You can also set expectations for which activities and data the cluster should host.

Summary

This chapter explored many aspects of running Docker services on a cluster of hosts managed by Swarm. The exercises demonstrated how to deploy several kinds of applications using Swarm's most important and commonly used features. We saw what happens when a cluster runs out of resources and showed how to recover from that condition. We also reconfigured a cluster and service deployment to implement user-defined architectural goals. The key points to understand from this chapter are:

- Administrators define cluster-scoped resources such as networks, configurations, and secrets that are shared by services.
- Services define their own service-scoped resources in addition to using cluster-scoped resources.
- Swarm managers store and manage updates to the desired state of the cluster.

- Swarm managers converge actual application and resource deployments to the desired state when sufficient resources are available.
- Service tasks are ephemeral, and service updates cause tasks to be replaced with new containers.
- Service tasks can be scaled to the desired state as long as the cluster has enough resources available on nodes that fulfill the service's placement constraints.

Symbols

/counter endpoint 269, 295
(hash sign) 231–232

A

-a flag 57, 129
abstractions 12
access control 177
access to devices 105
active option 293
ADD instruction 153, 155
adminer interface 239
adminer key 234, 236
adminer service 237, 239, 241
affinity 293
agents 22, 24
all-in-one images 200–201
All-in-One pattern 199–201
--all-tags option 136
Amazon Web Services (AWS)
 51, 267
anti-affinity 293
Apache Cassandra project 71
apache2 process 43
api service 248, 272, 276–277,
 282, 284–285, 289, 292
apk package manager 208
AppArmor profile 117, 119
app-image 202–204, 207–209
app-image-debug stage 203
application artifacts 199
applications 119–120
 configurations and, separating
 config resource 249–250

 deploying applications 250
 managing config resources
 directly 251–255
 deploying to Swarm
 clusters 267–278
 cluster resource types
 267–268
 defining application and its
 dependencies by using
 Docker services 268–273
 using docker stack
 command 273–278
apt package manager 8, 208
apt-get tool 127, 145
apt-get update, Debian 56
ARG instruction 159
ARG VERSION instruction 159
artifact confidentiality 177
artifact integrity 177
attachable network 85
AUDIT_CONTROL
 capability 115
auto tag 145
automated resurrection and
 replication 222–224
automated rollout 224–226
autoscaling services 290
availability control 177
--availability option 292
AWS (Amazon Web
 Services) 51, 267
aws program 156
AWS_ACCESS_KEY_ID 156
AWS_DEFAULT_REGION 156
AWS_SECRET_ACCESS_KEY
 156

B

backoff strategy 41
backoff-detector container
 42
bandwidth overhead 176
base.version label 151, 160
best practices 1
bind mounts 73
blind mount points 64–67
block sequences (lists) 232
block style 232
bridge network 83–84, 91
build command 146
Build Plus Multiple Runtimes
 pattern 199, 202
Build Plus Runtime
 pattern 199–200
builder stage 160, 162
BUILD_ID tag 207–209,
 213–215
BusyBox init 165

C

-c flag 140
CA (certificate authority) 161
CAIID (content-addressable
 image identifier) 168
CAP drop 7
--cap-add flag 115
--cap-drop flag 115
CAP_NET_ADMIN 169
Cassandra client tool
 (CQLSH) 71, 73
cass-shared volume 71–72

CD (continuous delivery) 207, 213
 CentOS 165
 certificate authority (CA) 161
 CERT_PRIVATE_KEY_FILE variable 261
 CFS (Completely Fair Scheduler) 103
 cgroups 6–7
 Chef Inspec tool 210
 chroot system call 7, 59–60
 CI (continuous integration) 2, 198
 CID (container ID) file 30
 -cidfile 30
 cleanup, containers and 44–45
 CLI (command-line interface) 6
 CLIENT_ID environment variable 39
 Cloudflare 90
 CMD instruction 153–155, 179
 command line, working with registries from 50–51
 command-line interface (CLI) 6
 command property 232
 commit subcommand 135
 Completely Fair Scheduler (CFS) 103
 Compose
 declarative service environments with 229–237
 collections of 233–237
 YAML primer 231–233
 load balancing, service discovery, and networks with 239–242
 --compose-file option 259
 composite key 49
 confidentiality, artifact 177
 config command 251
 Config property 108
 config.common.yml file 247
 ConfigID 253–254
 configs key 249
 Configuration Image per Deployment Stage pattern 251
 configurations 268
 applications and, separating 247–255
 config resource 249–250
 deploying applications 250–251
 managing config resources directly 251–255

distribution and management 245–247
 secrets and 255–263
 container ID (CID) file 30
 container networking
 custom DNS configuration 93–97
 externalizing network management 97
 firewalls, lack of 93
 network policies, lack of 93
 NodePort publishing 91–92
 Container Structure Test (CST) tool 210
 ContainerConfig.OnBuild 156
 containers 3, 5–8, 11, 220
 building environment-agnostic system 34–40
 environment variable injection 37–40
 read-only filesystems 34–37
 building images from 126–131
 committing new images 129–130
 configuring image attributes 130–131
 packaging “Hello, World” 126–127
 preparing packaging for Git 127–128
 reviewing filesystem changes 128
 cleanup and 44–45
 conflicts between, eliminating 28–34
 container state and dependencies 31–34
 flexible container identification 28–31
 creating and starting 21–22
 durable 40–44
 automatically restarting containers 41–42
 using PID 1 and init systems 42–44
 filesystem abstraction and isolation 59–60
 for isolation 6–7
 interactive, running 22–23
 listing, stopping, restarting, and viewing output of 23–25
 PID namespace and 25–27

shipping containers 7–8
 virtualization different than 5–6
 content-addressable image identifier (CAID) 168
 context 119
 context switch 104
 continuous delivery (CD) 207, 213
 continuous deployment 2
 continuous integration (CI) 2, 198
 COPY instruction 153, 155, 247
 copy-on-write pattern 60, 134
 corporate storage networks 188
 cost, of distribution 176
 CPU resources 102–104
 cpus option 103
 --cpuset-cpus flag 104
 --cpu-shares flag 102
 --cpu-shares value 103
 CQLSH (Cassandra client tool) 71, 73
 create command 74
 CST (Container Structure Test) tool 210
 curl command 277, 295
 current state 223
 custom DNS configuration 93–97
 custom image distribution infrastructure 189

D

DAB (Distributed Application Bundle) 274
 DAEMON Tools 165
 daemons 21
 databases 36
 db-data volume 296
 Debian 165
 Debian Linux Buster 58
 Debian Stretch base image 49
 debian:buster-slim 58
 debian:stretch platform 143
 declarative service environments with Compose 229–237
 collections of services 233–237
 YAML primer 231–233
 default stage 204
 depends_on key 272
 deploy command 219

- deploy key 272
- deploy property 235
- DEPLOY_ENV variable 249, 251
- desired state 223
- detach (-d) flag 21–22
- detached containers 20
- dev tag 213
- device flag 105
- devices, access to 105
- diaweb container 67
- diff command 35
- diff subcommand 128, 133
- DigitalOcean 267
- Distributed Application Bundle (DAB) 274
- distribution
 - image source-distribution workflows 194–196
 - manual image publishing and 188–193
 - method of, choosing 175–178
 - image distribution spectrum 175
 - selection criteria 176–178
 - private registries 183–187
 - consuming images from registry 187
 - performance of 184–185
 - registry image, using 186–187
 - publishing with hosted registries 178–183
 - private hosted repositories 181–183
 - public repositories 179–181
- Distribution registry 183
- dns flag 96
- DNS service, Cloudflare 90
- dns-search flag 96
- Docker
 - containers and 8
 - “Hello World” example 3–5
 - importance of 12–13
 - in larger ecosystem 14
 - overview of 3–8
 - problems solved by 8–11
 - getting organized 9
 - improving portability 10–11
 - protecting computer 11
 - use of, when and where 13–14
 - See also* containers
- Docker API 112
- docker build command 53, 136, 197, 203
- docker CLI 260–261, 273
- Docker CloudStor 296
- docker command 231, 275
- docker command-line tool 14–15, 179
- docker config command 252, 254
- docker config create command 261
- docker container commit command 129–131, 134, 136
- docker container create command 100, 102, 104, 116, 118, 169
- docker container diff command 133, 138
- docker container export command 139, 188
- docker container ps command 221–222, 224, 228
- docker container rm -f command 222
- docker container run command 101–102, 104, 116, 118, 152, 169, 181
- docker cp command 15, 139
- docker create command 26, 30, 32–33, 64, 74, 91
- docker diff command 35
- Docker Engine API 265, 273
- Docker Enterprise Edition 281
- docker exec command 26, 228
- docker help command 14
- Docker Hub 4, 21, 50, 54–56, 71, 178
- docker image build command 145–148, 150, 159, 188, 194, 203, 208
- docker image inspect command 160
- docker image load command 189
- docker image ls registry 193
- docker image pull command 168, 181
- docker image push command 182
- docker image save command 138, 188
- docker image tag command 141
- docker images command 53, 56–57
- docker import command 140
- docker inspect command 32, 35, 76, 108, 152
- docker kill command 45
- docker load command 48, 52–53, 192
- docker login command 51, 182
- docker logout command 51
- docker logs agent command 24
- docker logs command 24–25
- docker logs diaweb command 66
- docker logs mailer command 24
- docker network command 84
- docker network list command 89
- docker network ls command 84
- docker node ls command 295
- docker node update command 292
- docker plugin command 78
- docker port command 92
- docker ps -a command 45
- docker ps command 23, 31–32, 44, 92, 166
- docker pull command 50, 56–57, 186
- docker push command 179
- docker rename command 29
- docker restart agent command 24
- docker restart mailer command 24
- docker restart web command 24
- docker rm command 44–45
- docker rm -f command 45
- docker rm -v flag 77
- docker rmi command 55, 58
- docker rmi syntax 58
- docker run --rm flag 77
- docker run command 3–4, 21, 26, 30, 33, 50, 54–55, 64, 74, 89, 91, 93, 96
- docker save command 48, 52–53, 194
- docker secret command 275
- docker secret create command 260
- docker service command 288
- docker service create command 230, 294
- docker service inspect command 253, 280
- docker service inspect hello-world command 223

- docker service inspect my-data-bases_postgres
 - command 241
 - docker service logs
 - command 277
 - docker service ls command 222, 298
 - docker service ps command 223, 277, 291, 298
 - docker service ps greetings_prod_api
 - command 262
 - docker service ps hello-world
 - command 223–224, 226–227
 - docker service ps multi-tier-app_api 290
 - docker service remove
 - command 237
 - docker service rollback
 - subcommand 291
 - docker service scale
 - command 224
 - docker stack command 269, 273
 - docker stack deploy
 - command 234–235, 250, 259, 290, 297
 - docker stack ps command 235
 - docker stack subcommands 234
 - docker start command 32
 - docker stats command 101
 - docker stop command 25, 45
 - Docker Swarm
 - clustering with 264–267
 - communicating with services
 - running on Swarm cluster 278–287
 - load balancing 286–287
 - overlay networks 281–286
 - routing client requests
 - to services by using Swarm routing mesh 278–281
 - deploying applications to Swarm clusters 267–278
 - cluster resource types 267–268
 - defining application and its dependencies 268–273
 - using docker stack
 - command 273–278
 - placing service tasks on clusters 287, 299
 - constraining where tasks run 292–297
 - deploying real applications onto real clusters 299
 - replicating services 288–291
 - using global services for one task per node 297–299
 - docker swarm init 267
 - docker swarm join 267
 - docker swarm join-token 267
 - docker tag command 58, 136–137
 - docker top command 42
 - docker volume command 68
 - docker volume create
 - command 68, 74
 - docker volume inspect
 - command 68
 - docker volume list command 77
 - docker volume ls command 238
 - docker volume prune
 - command 78
 - docker volume remove
 - command 77
 - Dockerfiles 50, 53–54, 125, 150
 - distributing projects with, on GitHub 194–196
 - downstream build-time behavior, injecting 156–159
 - filesystem instructions 153–156
 - maintainable, creating 159–162
 - metadata instructions
 - naming Dockerfiles 150
 - organizing metadata with labels 151
 - overview 149–150
 - overview 148–149
 - packaging Git with 145–148
 - docker_hello_world
 - keyspace 71
 - dockerinaction client 40
 - dockerinaction username 180
 - docker-machine 267
 - docker-machine ip
 - command 103
 - docker-machine ssh
 - command 111
 - docker.sock 112
 - dockremap 113
 - double-quote style 232
 - downstream build-time behavior, injecting 156–159
 - downstream Dockerfile 156–157
 - drain option 293
 - driver_opts 269
 - Drone 198
 - dst parameter 66
 - durable containers 40–44
 - automatically restarting containers 41–42
 - using PID 1 and init systems 42–44
-
- ## E
- echo command 4
 - echo hello world value 232
 - echo service 286, 298
 - echo-global service 298
 - email flag 179
 - endpoint-mode property 286
 - end-to-end encryption 281
 - engine.labels attribute 294
 - entrypoint flag 43, 129
 - ENTRYPOINT instruction 147, 152–154, 162, 200
 - entrypoints 43, 129
 - env (-e) flag 38, 150
 - ENV instruction 150–152
 - environment property 234
 - environment variable
 - injection 37–40
 - environment-agnostic system, building 34–40
 - environment variable injection 37–40
 - read-only filesystems 34–37
 - environmental preconditions
 - validation 163–164
 - env_specific_config resource 249–250, 252, 255
 - env_specific_config_v1 254
 - env_specific_config_v2 255
 - env_specific_config_vNNN 255
 - escape sequences 232
 - Ethernet interface 81, 85
 - exec command 29
 - exec subcommand 43
 - exit command 23
 - expertise, required for distribution method 178
 - export command 140, 194
 - export subcommand 139
 - EXPOSE instruction 151
 - external: true property 242

F

f option 107
 --file (-f) flag 45, 146, 150
 file extensions 52
 filesystems
 instructions, Dockerfiles
 153–156
 structure 60
 union filesystems, weaknesses
 of 60–61
 firewalls, lack of 93
 First Secret Problem 256–257
 flag 94
 flat filesystems, exporting and
 importing 139–141
 flexible container
 identification 28–31
 flow collections 232
 foo network 242
 --force (-f) flag 78
 --format (-f) option 108
 FROM instruction 146, 152,
 157–158, 160, 168, 179, 202,
 204, 208
 FROM ubuntu:latest 146
 FTP (File Transport
 Protocol) 188
 FTP-based distribution
 infrastructure 193
 ftp-server container 191

G

Gathering Metadata
 message 208
 GID (group ID) 112
 Git 53
 packaging Dockerfiles
 with 145–148
 preparing packaging for
 127–128
 GitHub, distributing projects
 with Dockerfiles on
 194–196
 global mode 224, 280, 287,
 297
 global value 84
 Go programming language 139
 Golang 108, 161
 golang repository 143
 Google Container Registry 178
 gosu program 171
 greetings service 247, 249–251,
 259, 261–263

greetings_dev service 253
 greetings_dev_env_specific_
 config resource 253
 group ID (GID) 112

H

hadolint linting tool 208
 hardening images 167–172
 content addressable image
 identifiers 168
 SUID and SGID permissions
 171–172
 user permissions 169–171
 hash sign (#) 231–232
 health checks 166–167
 HEALTHCHECK instruction
 166–167, 228, 277
 --health-cmd flag 167
 “Hello, World” 3–5, 220–229
 automated resurrection
 and replication
 222–224
 automated rollout 224–226
 packaging 126–127
 service health and rollback
 226–229
 hello-registry image 51
 hello-world service 222–226,
 229, 231
 high-level system services 120
 Homebrew 8
 host driver 84
 host network 84
 host value 26
 .HostConfig.CapAdd
 member 116
 .HostConfig.CapDrop
 member 116
 hosted registries, publishing
 with 178–183
 private hosted repositories
 181–183
 public repositories
 179–181
 hosted registry 178
 --hostname flag 88
 hostname flag 93, 96
 HTTP (Hypertext Transfer
 Protocol) 81
 HTTP POST message 149
 http-client image 162
 http-client program 161
 httping program 228
 https 53

I

--i command 53
 id command 108
 identifier (unique ID) 58
 IMAGE ID column 58
 image layers 57–58
 image pipelines 197–216
 goals of 198–199
 orchestrating build with
 make 205–209
 patterns for building
 images 199–204
 all-in-one images
 200–201
 image build pattern
 maturity 199
 separate build and
 runtime images
 201–202
 variations of runtime
 image via multistage
 builds 202–204
 patterns for tagging
 images 212–216
 background 212–213
 configuration image per
 deployment stage
 214–215
 continuous delivery with
 unique tags 213
 semantic versioning
 215–216
 record metadata at image
 build time 204–209
 testing images in build
 pipelines 209–211
 image property 232, 234
 image source distribution,
 GitHub 195
 image source-distribution
 workflows 194–196
 image tag mutation 212
 image-dev container 129
 ImageMagick 258
 images
 as files 52–53
 building from containers
 126–131
 committing new images
 129–130
 configuring image
 attributes 130–131
 packaging “Hello, World”
 126–127

- images (*continued*)
 - preparing packaging for
 - Git 127–128
 - reviewing filesystem changes 128
 - hardening 167–172
 - content addressable image identifiers 168
 - SUID and SGID
 - permissions 171–172
 - user permissions 169–171
 - startup scripts and multiprocess containers, using 162–167
 - environmental preconditions validation 163–164
 - health checks 166–167
 - initialization processes 164–166
 - See* Dockerfiles
 - imperative pattern 229
 - import command 140, 189, 194
 - indexes 8, 50
 - info subcommand 61
 - ingress network 240, 279–281, 284–285
 - init command 221
 - init option 165
 - init systems 42–44
 - initialization processes 164–166
 - in-memory storage 67–68
 - insecure.key 260
 - inspect command 108, 252
 - installing software 56–61
 - Docker registries, working with from command line 50–51
 - from Dockerfiles 53–54
 - installation files and isolation
 - container filesystem abstraction and isolation 59–60
 - filesystem structure 60
 - image layers 57–58
 - layer relationships 58–59
 - union filesystems, weaknesses of 60–61
 - using alternative registries 51–52
 - using Docker Hub from website 54–56
 - working with images as files 52–53
 - integrity, artifact 177
 - interactive (-i) option 22
 - interactive containers, running 22–23
 - interfaces, network 81
 - intermediate images 56
 - interprocess communication (IPC) 105
 - IP (Internet Protocol) 81
 - IP address 81
 - IPC (interprocess communication) 105
 - ipc flag 106
 - IPC namespace 7
 - IPC primitives 106–107
 - ipc=host option 107
 - iptables rule 279
 - ipvlan driver 89
 - ipvs rule 279
 - isolation, containers and 6–7
- J**

- jail 5
- Jenkins 198
- JVM (Java Virtual Machine) 9, 11
- K**

- key/value pairs 232
- kill program, Linux 43
- Kubernetes 14, 97
- L**

- label confinement, SELinux 118
- label flag 151
- LABEL instruction 150–152, 204
- LABEL maintainer 146
- Label Schema project 151, 204
- label-add option 294
- label-rm option 294
- LAMP (Linux, Apache, MySQL PHP) stack 42
- lamp-test container 43
- latest tag 49, 136, 142, 213
- layer IDs 135
- layer relationships 58–59
- level label, SELinux 118
- LFTP client 191
- lighthouse container 88
- linting tool 208
- Linux Security Modules (LSM) 118
- Linux, Apache, MySQL PHP (LAMP) stack 42
- Linux's user (USR) namespace 113–114
- lists (block sequences) 232
- load balancing 239–242, 286–287
- load command 194
- local value 84
- local volume driver 270
- local volume plugin 68
- localhost node 278
- logging service 285
- login command 179
- logs command 25
- longevity control 177
- loopback interface 81, 85–86
- low-level system services 120–121
- LSM (Linux Security Modules) 118
- M**

- m flag 129
- MAC_ADMIN capability 115
- macvlan driver 89
- mailer container 22, 24
- mailer program 20
- mailer-base image 149
- mailer-logging image 154
- Major.Minor.Patch scheme 213, 215
- make command 205
- manager1 node 274, 277, 288
- manager2 node 288, 291
- manager3 node 288
- manual image publishing 188–193
- mariadb container 102
- MariaDB database 233
- mariadb key 234, 236
- mariadb service 237
- maturity 199
- Maven tool 201
- Memcached 82
- memory flag 101
- memory limits 101–102
- Mercurial 53
- metaconflicts, eliminating 28–34
 - container state and dependencies 31–34
 - flexible container identification 28–31

metadata
 naming Dockerfiles 150
 organizing metadata with labels 151
 overview 149–150
 record metadata at image build time 204, 209
 MNT namespaces 7, 60, 63
 mod_ubuntu container 131, 135–136
 MongoDB 70
 --mount flag 64, 67
 --mount option 66
 mount points 63–67
 blind 64–67
 shared 73–76
 multistage Dockerfile 160
 multi-tier-app_api 276, 280
 multi-tier-app_postgres 276
 multi-tier-app_private network 282–283, 286–287
 multi-tier-app_public 282
 mutable tags 212
 my-databases stack 240
 my-databases_default network 242
 my-databases_mariadb 237
 MySQL 34, 70, 82

N

--name flag 29
 namespaces, Linux 6
 NAT (network address translation) 91
 nc client program 299
 --net host flag 117
 NET namespace 7
 NET_ADMIN capability 115
 NET_RAW capability 115
 network address translation (NAT) 91
 --network flag 106
 --network host option 89
 network interface 81
 network-explorer container 86–88
 networks 268
 overview 81–83
 policies for, lack of 93
 with Compose 239–242
 networks property 241
 NGINX web server 20–21, 26, 31, 65, 166
 nginx:latest 21, 49, 66

--no-cache flag 148
 node.hostname attribute 294
 node.id attribute 294
 NodePort publishing 91–92
 node.role attribute 294
 --no-healthcheck flag 229
 none network 84, 90–91, 97
 --no-trunc option 31
 npm package manager 8, 208
 nslookup lighthouse 88
 null driver 84

O

-o flag 52
 ONBUILD instruction 156–158
 onbuild suffixes 159
 --output (-o) option 140
 overlay networks 89, 281–286
 discovering services on 282–284
 isolating service-to-service communication with 284–286
 OverlayFS 60

P

package managers 8
 passwd file 110
 --password flag 179
 pause option 293
 peer-to-peer networks 188
 Permission denied message 111
 pg-data volume 239
 pgdata volume 238
 PID (process identifier) namespace 7, 25–27, 113
 PID 1 systems 42–44
 --pid flag 26
 pip package manager 208
 placement constraints 292
 plain style 232
 plath container 74
 Play with Docker (PWD) 267, 289
 polymorphic tools 70
 port conflict 27
 portability, improving 10–11
 ports 81
 ports property 234
 postgres database 239
 postgres key 234
 postgres service 237–238, 241, 270–271, 290, 297

postgres user 170
 POSTGRES_* environment variable 271–272
 POSTGRES_DB variable 271
 POSTGRES_HOST variable 282
 POSTGRES_PASSWORD variable 271
 PostgreSQL database 233, 267–268
 postgresql driver 239
 POSTGRES_USER variable 271
 private hosted repositories 182
 private network 269, 271–272, 283
 private registries 183–187
 consuming images from registry 187
 performance of 184–185
 registry image, using 186–187
 private zone 295
 --privileged flag 116
 process identifier (PID) namespace 7, 25–27, 113
 protecting computer 11
 protocols, network 81
 --prune flag 237
 ps command 26
 public hosted repositories 180
 public network 269, 272, 284
 --publish (-p) option 91
 PublishMode 280
 pull commands 178
 push command 180, 186
 PWD (Play with Docker) 267, 289

Q

Quay.io 178, 182
 --quiet (-q) flag 147

R

reader container 75
 read-only filesystems 34–37
 --read-only flag 34
 readonly=true argument 66
 registries 8
 alternative 51–52
 working with from command line 50–51
 registry repository 186
 remove command 77
 replicas property 235

- replicated mode 224
- replicated services 288
- repositories 136
- reserved status 166
- resource controls 99–121
 - adjusting OS feature access with capabilities 114–116
 - allowances, setting 100–105
 - access to devices 105
 - CPU 102–104
 - memory limits 101–102
 - building use-case-appropriate containers 119–121
 - applications 119–120
 - high-level system services 120
 - low-level system services 120–121
 - running containers with full privileges 116–117
 - sharing memory 105–107
 - strengthening containers with enhanced tools 117–119
 - users and 107–114
 - Linux’s user (USR) namespace 113–114
 - run-as users 108–110
 - UID remapping 113–114
 - volumes and 111–113
- restart flag 41, 102
- restart_policy 272
- REX-Ray 78, 296
- rm command 107
- rm flag 72
- rm operation 252
- rm option 55
- role label, SELinux 118
- roll forward command 291
- rollback 226–229
- rollback flag 226
- rollout, automated 224–226
- rsync tool 193–194
- rules 205
- run command 22, 66, 74, 186
- RUN instruction 147, 153, 158, 170, 208
- run-as users 108–110
- runit 25, 165
- Running state 223
- Running x minutes ago state 223
- runtime stage 160, 162
- Rust programming language 139

S

- scanners, using to identify vulnerabilities 212
- scope property 84, 86
- scp tooling 193
- scratch image 139, 162
- scratch repository 146
- \$SECRET variable 271
- secrets 255–263, 268
- secrets key 260, 270
- security modules 7
- security-opt flag 118
- sed command 67
- SELinux 117–119
- semantic versioning 215–216
- Serverspec 210
- Service converged 225
- service create subcommand 221
- service rollback subcommand 291
- service-level agreement (SLA) 177
- services 21, 220, 268
 - autoscaling 290
 - declarative service environments with Compose 229–237
 - collections of services 233–237
 - YAML primer 231–233
 - “Hello World” 220–229
 - automated resurrection and replication 222–224
 - automated rollout 224–226
 - service health and rollback 226–229
- load balancing, service discovery, and networks with Compose 239–242
- placing service tasks on clusters 287–299
 - constraining where tasks run 292–297
- deploying real applications onto real clusters 299
- replicating services 288–291
 - using global services for one task per node 297–299
- running on Swarm cluster, communicating with 278–287
 - load balancing 286–287
 - overlay networks 281–286
 - routing client requests to services by using Swarm routing mesh 278–281
 - stateful services and preserving data 237–239
- services key 270–271
- services property 234
- ServiceV2 flag 225
- setgid attribute 168
- setuid attribute 168
- sh program 22
- shared mount points 73–76
- shell form 152
- shell variables 30
- shipping containers 7–8
- short-form flags 23
- SIG_HUP signal 45
- SIG_KILL signal 45
- Simple Email Service 155–156
- SLA (service-level agreement) 177
- software
 - finding 50
 - identifying 48–50
 - installation files and isolation 56–61
 - container filesystem abstraction and isolation 59–60
 - filesystem structure 60
 - image layers 57–58
 - layer relationships 58–59
 - union filesystems, weaknesses of 60–61
- installing
 - from Dockerfiles 53–54
 - using alternative registries 51–52
 - using Docker Hub from website 54–56
 - working with Docker registries from command line 50–51
 - working with images as files 52–53
- Spring Boot framework 200
- src parameter 66
- SSH protocol 193
- stack deploy command 236
- stacks 231
- start period 167

startup scripts and multiprocess
containers 44, 162–167
environmental preconditions
validation 163–164
health checks 166–167
initialization processes
164–166
stateful services 237–239
STATUS column 166
stop command 29
storage
in-memory 67–68
mount points 63–67
blind 64–67
shared 73–76
volume plugins 78–79
volumes 68–73
cleaning up 77–78
container-independent
data management
using 70
using with NoSQL
databases 71–73
--storage-driver (-s) option 61
subgid map 113
subuid map 113
success status 166
SUID and SGID
permissions 171–172
suid-enabled program 110
supervisord process 25, 43,
165
swap space 102
Swarm 221, 223, 226–227, 230,
257, 265
Swarm Admin Guide 266
swarm value 84
Swarm3K project 266
SYS_ADMIN capability 115
SYSLOG capability 115
SYS_MODULE capability 115
SYS_NICE capability 115
SYS_RAWIO capability 115
SYS_RESOURCE capability
115
System up message 23
systemd 165
SYS_TIME capability 115
SysV 165

T

-t option 53
--tag (-t) flag 146
tag command 186

tagging images, patterns
for 212–216
background 212–213
configuration image per
deployment stage
214–215
continuous delivery with
unique tags 213
semantic versioning
215–216
TAR archive files 52
--target option 203
tasks 268
TCP (Transmission Control
Protocol) 81
testing images in build
pipelines 209–211
time-out 167
tini 165
TLS (Transport Layer
Security) 161
tmpfs device 67–68
tmpfs filesystem 257, 272
tmpfs-mode option 68
tmpfs-size option 68
Tomcat 70
top subcommand 42
Transmission Control Protocol
(TCP) 81
Transport Layer Security
(TLS) 161
transportation speed, of distribu-
tion method 176
Travis CI 198
tree root point 63
trusted repository 21
--tty (-t) option 22–23
type label, SELinux 118
type option 67
type=bind option 66

U

Ubuntu image 127, 165
ubuntu:latest 131
ubuntu:latest image 128, 138
ubuntu-git images 137, 139, 145
ubuntu-git:removed image 138
UDP (User Datagram
Protocol) 81
UFS (union filesystem) 59–61,
126, 134, 166
UID (user ID) 110
UID remapping 113–114
unconfined value 118

union filesystem (UFS) 59–61,
126, 134, 166
unique ID (identifier) 58
unique tags 213
Unique Tags scheme 213
update command 226
update_config 273
--update-failure-action flag 227
--update-max-failure-ratio
flag 227
Upstart 165
--user (-u) flag 109–110
user (USR) namespace 7,
113–114, 169
User Datagram Protocol
(UDP) 81
user ID (UID) 110
USER instruction 152, 169
user label, SELinux 118
user namespaces 114
user permissions 169–171
user space memory 6
--username flag 179
username/repository pattern 55
user-network 85
users-remap option 113, 170
USR (Linux's user)
namespace 113–114
USR (user) namespace 7,
113–114, 169
UTS namespace 7

V

v option 107
v2 tag 49, 225
VCS (version-control
system) 204
version property 234
VERSION variable 150, 159
version-control system
(VCS) 204
versioning practices 141–143
VIP (virtual IP) addresses 239,
241, 279
vip endpoint mode 286
virtual machine (VM) 2, 5–6,
10, 111
VirtualIPs 283
virtualization, containers differ-
ent than 5–6
visibility, of distribution
method 176
VM (virtual machine) 2, 5–6,
10, 111

_vNNN character sequence 255
 --volume flag 153
 VOLUME instruction 153
 volume plugins 78–79
 volumes 30, 68–73, 111–113, 268
 cleaning up 77–78
 container-independent data management using 70
 using with NoSQL databases 71–73
 volumes key 270
 volumes property 237–238
 --volumes-from flag 75–76
 vulnerabilities, using scanners to identify 212

W

watcher program 20
 web browsers 119
 WEB_HOST environment variable 164

wget program 22
 whoami command 108
 Windows Subsystem for Linux (WSL) 205
 WordPress 34–37, 40, 43, 103, 163
 WORDPRESS_AUTH_KEY environment variable 38
 WORDPRESS_AUTH_SALT environment variable 38
 WORDPRESS_DB_HOST environment variable 38
 WORDPRESS_DB_NAME environment variable 38–39
 WORDPRESS_DB_PASSWORD environment variable 38
 WORDPRESS_DB_USER environment variable 38
 WORDPRESS_LOGGED_IN_KEY environment variable 38
 WORDPRESS_LOGGED_IN_SALT environment variable 38

WORDPRESS_NONCE_KEY environment variable 38
 WORDPRESS_NONCE_SALT environment variable 38
 WORDPRESS_SECURE_AUTH_KEY environment variable 38
 WORDPRESS_SECURE_AUTH_SALT environment variable 38
 WORKDIR instruction 151
 wp container 35, 44
 WSL (Windows Subsystem for Linux) 205

Y

YAML (Yet Another Markup Language) 231–233, 245
 YUM 8

Z

zone label 266, 294

Docker IN ACTION Second Edition

Nickoloff • Kuenzli



The idea behind Docker is simple—package just your application and its dependencies into a lightweight, isolated virtual environment called a container. Applications running inside containers are easy to install, manage, and remove. This simple idea is used in everything from creating safe, portable development environments to streamlining deployment and scaling for microservices. In short, Docker is everywhere.

Docker in Action, Second Edition teaches you to create, deploy, and manage applications hosted in Docker containers running on Linux. Fully updated, with four new chapters and revised best practices and examples, this second edition begins with a clear explanation of the Docker model. Then, you go hands-on with packaging applications, testing, installing, running programs securely, and deploying them across a cluster of hosts. With examples showing how Docker benefits the whole dev lifecycle, you'll discover techniques for everything from dev-and-test machines to full-scale cloud deployments.

What's Inside

- Running software in containers
- Packaging software for deployment
- Securing and distributing containerized applications

Written for developers with experience working with Linux.

Jeff Nickoloff and **Stephen Kuenzli** have designed, built, deployed, and operated highly available, scalable software systems for nearly 20 years.

“Jeff and Stephen took their battle-hardened experience and updated this already great book with new details and examples.”

—From the Foreword by
Bret Fisher, Docker Captain
and Container Consultant

“Strikes the perfect balance between instructional manual and reference book. Ideal for everyone from beginner to seasoned pro.”

—Paul G. Brown
Diversified Services Network

“A must-have for those looking to level-up their organization's software and infrastructure virtualization.”

—Chris Phillips, Dell

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/docker-in-action-second-edition

ISBN-13: 978-1-61729-476-1
ISBN-10: 1-61729-476-4



9 781617 294761