

Docker

IN ACTION

Jeff Nickoloff



MANNING



Docker in Action

Docker in Action

JEFF NICKOLOFF



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

©2016 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 elemental chlorine.

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

Development editor: Cynthia Kane
Technical development editor: Robert Wenner
Technical proofreader: Niek Palm
Copyeditor: Linda Recktenwald
Proofreader: Corbin Collins
Typesetter: Marija Tudor
Cover designer: Marija Tudor

ISBN: 9781633430235

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – EBM – 21 20 19 18 17 16

contents

foreword xi
preface xiii
acknowledgments xv
about this book xvii
about the cover illustration xix

PART 1 KEEPING A TIDY COMPUTER 1

1 *Welcome to Docker* 3

1.1 What is Docker? 4

Containers 4 ▪ *Containers are not virtualization* 5
Running software in containers for isolation 5 ▪ *Shipping containers* 7

1.2 What problems does Docker solve? 7

Getting organized 8 ▪ *Improving portability* 9 ▪ *Protecting your computer* 10

1.3 Why is Docker important? 11

1.4 Where and when to use Docker 11

1.5 Example: “Hello, World” 12

1.6 Summary 14

2 *Running software in containers* 15

2.1 Getting help with the Docker command line 15

- 2.2 Controlling containers: building a website monitor 16
 - Creating and starting a new container* 17 ▪ *Running interactive containers* 18 ▪ *Listing, stopping, restarting, and viewing output of containers* 20
- 2.3 Solved problems and the PID namespace 21
- 2.4 Eliminating metaconflicts: building a website farm 24
 - Flexible container identification* 25 ▪ *Container state and dependencies* 28
- 2.5 Building environment-agnostic systems 30
 - Read-only file systems* 30 ▪ *Environment variable injection* 32
- 2.6 Building durable containers 35
 - Automatically restarting containers* 36 ▪ *Keeping containers running with supervisor and startup processes* 37
- 2.7 Cleaning up 39
- 2.8 Summary 40

3 *Software installation simplified* 41

- 3.1 Identifying software 42
 - What is a repository?* 42 ▪ *Using tags* 43
- 3.2 Finding and installing software 44
 - Docker Hub from the command line* 44 ▪ *Docker Hub from the website* 46 ▪ *Using alternative registries* 48 ▪ *Images as files* 48 ▪ *Installing from a Dockerfile* 50
- 3.3 Installation files and isolation 51
 - Image layers in action* 51 ▪ *Layer relationships* 53
 - Container file system abstraction and isolation* 53 ▪ *Benefits of this toolset and file system structure* 54 ▪ *Weaknesses of union file systems* 54
- 3.4 Summary 55

4 *Persistent storage and shared state with volumes* 56

- 4.1 Introducing volumes 57
 - Volumes provide container-independent data management* 58
 - Using volumes with a NoSQL database* 58
- 4.2 Volume types 61
 - Bind mount volumes* 62 ▪ *Docker-managed volumes* 64

- 4.3 Sharing volumes 66
 - Host-dependent sharing* 66 ▪ *Generalized sharing and the volumes-from flag* 67
- 4.4 The managed volume life cycle 69
 - Volume ownership* 69 ▪ *Cleaning up volumes* 70
- 4.5 Advanced container patterns with volumes 71
 - Volume container pattern* 72 ▪ *Data-packed volume containers* 73 ▪ *Polymorphic container pattern* 74
- 4.6 Summary 75

5 *Network exposure* 77

- 5.1 Networking background 78
 - Basics: protocols, interfaces, and ports* 78 ▪ *Bigger picture: networks, NAT, and port forwarding* 79
- 5.2 Docker container networking 81
 - The local Docker network topology* 81 ▪ *Four network container archetypes* 82
- 5.3 Closed containers 83
- 5.4 Bridged containers 85
 - Reaching out* 85 ▪ *Custom name resolution* 86 ▪ *Opening inbound communication* 89 ▪ *Inter-container communication* 91 ▪ *Modifying the bridge interface* 92
- 5.5 Joined containers 94
- 5.6 Open containers 96
- 5.7 Inter-container dependencies 97
 - Introducing links for local service discovery* 97 ▪ *Link aliases* 99 ▪ *Environment modifications* 100 ▪ *Link nature and shortcomings* 102
- 5.8 Summary 103

6 *Limiting risk with isolation* 104

- 6.1 Resource allowances 105
 - Memory limits* 105 ▪ *CPU* 107 ▪ *Access to devices* 109
- 6.2 Shared memory 109
 - Sharing IPC primitives between containers* 110 ▪ *Using an open memory container* 111

- 6.3 Understanding users 112
 - Introduction to the Linux user namespace* 112 ▪ *Working with the run-as user* 113 ▪ *Users and volumes* 115
- 6.4 Adjusting OS feature access with capabilities 117
- 6.5 Running a container with full privileges 118
- 6.6 Stronger containers with enhanced tools 119
 - Specifying additional security options* 120 ▪ *Fine-tuning with LXC* 121
- 6.7 Build use-case-appropriate containers 122
 - Applications* 122 ▪ *High-level system services* 123
 - Low-level system services* 123
- 6.8 Summary 123

PART 2 PACKAGING SOFTWARE FOR DISTRIBUTION ... 125

7 *Packaging software in images* 127

- 7.1 Building Docker images from a container 127
 - Packaging Hello World* 128 ▪ *Preparing packaging for Git* 129 ▪ *Reviewing file system changes* 129 ▪ *Committing a new image* 130 ▪ *Configurable image attributes* 131
- 7.2 Going deep on Docker images and layers 132
 - An exploration of union file systems* 132 ▪ *Reintroducing images, layers, repositories, and tags* 135 ▪ *Managing image size and layer limits* 138
- 7.3 Exporting and importing flat file systems 140
- 7.4 Versioning best practices 141
- 7.5 Summary 143

8 *Build automation and advanced image considerations* 145

- 8.1 Packaging Git with a Dockerfile 146
- 8.2 A Dockerfile primer 149
 - Metadata instructions* 150 ▪ *File system instructions* 153
- 8.3 Injecting downstream build-time behavior 156
- 8.4 Using startup scripts and multiprocess containers 159
 - Environmental preconditions validation* 159 ▪ *Initialization processes* 160

- 8.5 Building hardened application images 161
 - Content addressable image identifiers* 162 ▪ *User permissions* 163 ▪ *SUID and SGID permissions* 165
- 8.6 Summary 166

9 *Public and private software distribution* 168

- 9.1 Choosing a distribution method 169
 - A distribution spectrum* 169 ▪ *Selection criteria* 170
- 9.2 Publishing with hosted registries 172
 - Publishing with public repositories: Hello World via Docker Hub* 172 ▪ *Publishing public projects with automated builds* 175 ▪ *Private hosted repositories* 177
- 9.3 Introducing private registries 179
 - Using the registry image* 181 ▪ *Consuming images from your registry* 182
- 9.4 Manual image publishing and distribution 183
 - A sample distribution infrastructure using the File Transfer Protocol* 185
- 9.5 Image source distribution workflows 188
 - Distributing a project with Dockerfile on GitHub* 189
- 9.6 Summary 190

10 *Running customized registries* 192

- 10.1 Running a personal registry 194
 - Reintroducing the Image* 194 ▪ *Introducing the V2 API* 195
 - Customizing the Image* 197
- 10.2 Enhancements for centralized registries 198
 - Creating a reverse proxy* 199 ▪ *Configuring HTTPS (TLS) on the reverse proxy* 201 ▪ *Adding an authentication layer* 205
 - Client compatibility* 208 ▪ *Before going to production* 210
- 10.3 Durable blob storage 212
 - Hosted remote storage with Microsoft's Azure* 213 ▪ *Hosted remote storage with Amazon's Simple Storage Service* 214
 - Internal remote storage with RADOS (Ceph)* 216
- 10.4 Scaling access and latency improvements 217
 - Integrating a metadata cache* 217 ▪ *Streamline blob transfer with storage middleware* 219

- 10.5 Integrating through notifications 221
- 10.6 Summary 227

PART 3 MULTI-CONTAINER AND MULTI-HOST ENVIRONMENTS 229

11 *Declarative environments with Docker Compose* 231

- 11.1 Docker Compose: up and running on day one 232
 - Onboarding with a simple development environment* 232
 - A complicated architecture: distribution and Elasticsearch integration* 234
- 11.2 Iterating within an environment 236
 - Build, start, and rebuild services* 237 ▀ *Scale and remove services* 240 ▀ *Iteration and persistent state* 242
 - Linking problems and the network* 243
- 11.3 Starting a new project: Compose YAML in three samples 243
 - Prelaunch builds, the environment, metadata, and networking* 244 ▀ *Known artifacts and bind-mount volumes* 245 ▀ *Volume containers and extended services* 246
- 11.4 Summary 247

12 *Clusters with Machine and Swarm* 248

- 12.1 Introducing Docker Machine 249
 - Building and managing Docker Machines* 250 ▀ *Configuring Docker clients to work with remote daemons* 252
- 12.2 Introducing Docker Swarm 255
 - Building a Swarm cluster with Docker Machine* 255 ▀ *Swarm extends the Docker Remote API* 258
- 12.3 Swarm scheduling 261
 - The Spread algorithm* 261 ▀ *Fine-tune scheduling with filters* 263 ▀ *Scheduling with BinPack and Random* 267
- 12.4 Swarm service discovery 269
 - Swarm and single-host networking* 269 ▀ *Ecosystem service discovery and stop-gap measures* 271 ▀ *Looking forward to multi-host networking* 272
- 12.5 Summary 274
 - index* 275

foreword

I heard about Docker for the first time in a YouTube video that was posted to Hacker News from PyCon 2013. In his five-minute lightning talk entitled “The Future of Linux Containers,” the creator of Docker, Solomon Hykes, was unveiling the future of how we ship and run software to the public—not just in Linux, but on nearly all platforms and architectures. Although he was abruptly silenced at the five-minute mark, it was clear to me that this technique of running Linux applications in sandboxed environments, with its user-friendly command-line tool and unique concepts such as image layering, was going to change a lot of things.

Docker vastly changed many software development and operations paradigms all at once. The ways we architect, develop, ship, and run software before and after Docker are vastly different. Although Docker does not prescribe a certain recipe, it forces people to think in terms of microservices and immutable infrastructure.

Once Docker was more widely adopted, and as people started to investigate the low-level technologies utilized by Docker, it became clearer that the secret to Docker’s success was not the technology itself, but the human-friendly interface, APIs, and ecosystem around the project.

Many big companies such as Google, Microsoft, and IBM have gathered around the Docker project and worked together to make it even better rather than creating a competitor to it. In fact, companies like Microsoft, Joyent, Intel, and VMware have swapped out Docker’s Linux containers implementation but kept the novel Docker command-line interface for their own container offerings. In only two years, many new companies have sprouted up to enhance the developer experience and fill in the blanks of the Docker ecosystem—the sign of a healthy and enthusiastic community around Docker.

For my own part, I began helping Microsoft adopt and contribute to Docker by publishing Microsoft’s first official Docker image for cross-platform ASP.NET. My next

contribution was porting the Docker command-line interface to Windows. This project helped many Windows developers become familiar with Docker and laid the foundation for Microsoft's long journey of contributing to the Docker project. The Windows porting project also skyrocketed me to the top Docker contributor spot for more than two months. Later on, we contributed many other bits and pieces to make sure Docker became a first-class citizen on Microsoft's Azure cloud offering. Our next big step is Windows Containers, a new feature in Windows Server 2016, which is fully integrated with Docker.

It is exciting to know that we're still at the start of the containers revolution. The scene moves incredibly fast, as new technologies and open source tools emerge daily. Everything we take for granted today can and will change in the next few months. This is an area where innovators and the greatest minds of our industry are collaborating to build tools of mass innovation and make the problem of shipping and running software at scale one less thing to worry about for the rest of the software industry.

Through his many online articles about Docker and microservices, Jeff Nickoloff has shown himself to be the champion of the nascent Docker community. His well-written, thorough explanations of some very technical topics have allowed developers to quickly learn and use the Docker ecosystem for all its benefits, and, equally important, he notes its drawbacks. In this book, he goes from zero to Docker, shows practices of deploying Docker in production, and demonstrates many features of Docker with comprehensive descriptions and comparisons of various ways of achieving the same task.

While reading this book, not only will you learn how to use Docker effectively, you'll also grasp how it works, how each detailed feature of Docker is meant to be used, and the best practices concocted for using Docker in production. I personally had many "Oh, that's what this feature is for" moments while reading this book. Although writing a book about a technology that moves at an incredible pace is very much like trying to paint a picture of a car moving at 60 mph, Jeff has done a fantastic job at both covering cutting-edge features in Docker and laying a solid foundation throughout the book. This foundation builds an appreciation and understanding for the philosophy of containers and microservices that is unlikely to change, no matter what Docker looks like in the coming months and years.

I hope you find this book as enjoyable and educational as I did.

AHMET ALP BALKAN
OPEN SOURCE SOFTWARE ENGINEER AT MICROSOFT,
DOCKER CONTRIBUTOR

preface

In 2011, I started working at Amazon.com. In that first week my life was changed as I learned how to use their internal build, dependency modeling, and deployment tooling. This was the kind of automated management I had always known was possible but had never seen. I was coming from a team that would deploy quarterly and take 10 hours to do so. At Amazon I was watching rolling deployments push changes I had made earlier that day to hundreds of machines spread all over the globe. If big tech firms had an engineering advantage over the rest of the corporate landscape, this was it.

Early in 2013, I wanted to work with Graphite (a metrics collection and graphing suite). One day I sat down to install the software and start integrating a personal project. At this point I had several years of experience working with open source applications, but few were as dependent on such large swaths of the Python ecosystem. The installation instructions were long and murky. Over the next several hours, I discovered many undocumented installation steps. These were things that might have been more obvious to a person with deeper Python ecosystem knowledge. After pouring over several installation guides, reading through configuration files, and fighting an epic battle through the deepest parts of dependency hell, I threw in the towel.

Those had been some of the least inspiring hours of my life. I wanted nothing to do with the project. To make matters worse, I had altered my environment in a way that was incompatible with other software that I use regularly. Reverting those changes took an embarrassingly long time.

I distinctly remember sitting at my desk one day in May that year. I was between tasks when I decided to check Hacker News for new ways to grow my skillset. Articles about a technology called Docker had made the front page a few times that week. That evening I decided to check it out. I hit the site and had the software installed within a few minutes. I was running Ubuntu on my desktop at home, and Docker only had two dependencies: LXC and the Linux kernel itself.

Like everyone else, I kicked the tires with a “Hello, World” example, but learned little. Next I fired up Memcached. It was downloaded and running in under a minute. Then I started WordPress, which came bundled with its own MySQL server. I pulled a couple different Java images, and then Python images. Then my mind flashed back to that terrible day with Graphite. I popped over to the Docker Index (this was before Docker Hub) and did a quick search.

The results came back, and there it was. Some random user had created a Graphite image. I pulled it down and created a new container. It was running. A simple but fully configured Graphite server was running on my machine. I had accomplished in less than a minute of download time what I had failed to do with several hours a few months earlier. Docker was able to demonstrate value with the simplest of examples and minimum effort. I was sold.

Over the next week, I tried the patience of a close friend by struggling to direct our conversations toward Docker and containers. I explained how package management was nice, but enforcing file system isolation as a default solved several management problems. I rattled on about resource efficiency and provisioning latency. I repeated this conversation with several other colleagues and fumbled through the container story. Everyone had the same set of tired questions, “Oh, it’s like virtualization?” and “Why do I need this if I have virtual machines?” The more questions people asked, the more I wanted to know. Based on the popularity of the project, this is a story shared by many.

I began including sessions about Docker when I spoke publicly. In 2013 and 2014, only a few people had heard of Docker, and even fewer had actually tried the software. For the most part, the crowds consisted of a few skeptical system administrator types and a substantial number of excited developers. People reacted in a multitude of ways. Some were pure rejectionists who clearly preferred the status quo. Others could see problems that they experienced daily solved in a matter of moments. Those people reacted with an excitement similar to mine.

In the summer of 2014, an associate publisher with Manning called me to talk about Docker. After a bit more than an hour on the phone he asked me if there was enough content there for a book. I suggested that there was enough for a few books. He asked me if I was interested in writing it, and I became more excited than I had been for some time. That fall I left Amazon.com and started work on *Docker in Action*.

Today, I’m sitting in front of the finished manuscript. My goal in writing this book was to create something that would help people of mixed backgrounds get up to speed on Docker as quickly as possible, but in such a way that they understand the underlying mechanisms. The hope is that with that knowledge, readers can understand how Docker has been applied to certain problems, and how they might apply it in their own use-cases.

acknowledgments

I believe that I've spent enough of my life doing easy things. Before I began this book, I knew that writing it would require a high degree of discipline and an unending stream of motivation. I was not disappointed.

First I'd like to acknowledge Manning Publications for the opportunity to publish this work. I'd like to thank Ahmet Alp Baken for writing a foreword to the book, as well as Niek Palm for giving the whole manuscript a technical proofread. Many others reviewed the manuscript and offered comments at various stages of development, including Robert Wenner, Jean-Pol Landrain, John Guthrie, Benoît Benedetti, Thomas Peklak, Jeremy Gailor, Fernando Fraga Rodrigues, Gregor Zurowski, Peter Sellars, Mike Shepard, Peter Krey, Fernando Kobayashi, and Edward Kuns.

In this and most other difficult ventures, success is dependent on the collective contributions of a support network. I wouldn't be here today without contributions from the following:

- Portia Dean, for her partnership and support over the last year. Portia, you are my partner, my righteous and stubborn center. Without you I would have lost my mind somewhere in this maze of a year. I've loved the adventure and can't wait for what comes next.
- My parents, Kathy and Jeff Nickoloff, Sr., for supporting my technical curiosity from a young age and cultivating my strong will.
- Neil Fritz, for hacking out projects with me over the last 15 years and always being open to getting Slices Pizza.
- Andy Will and the strong engineers of PHX2, for welcoming me to Amazon and always raising our technical bar. Working with them was an education in itself.
- Nick Ciubotariu, for fighting the good fight and raising the bar for technical leadership.

- Cartel Coffee Lab, I spent more time in your HQ than I did my own house this year. You have one of the best roasts in the world. People in San Francisco are missing out.

Finally, I want to acknowledge my like-minded friends around the world who've shared in some part of this journey through learning, sharing, challenging, or just listening. #nogui

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 multi-container projects and multi-host environments. This includes coverage of the Docker Compose, Machine, 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 multi-purpose 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 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.

About the author

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. If you’d like to chat or work together, you can find him at <http://allingeek.com>, or on Twitter as @allingeek.

Author Online

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 and subscribe to it, point your web browser to www.manning.com/books/docker-in-action. This page provides information on how to get on the forum once you’re registered, what kind of help is available, and the rules of conduct on the forum.

Manning’s commitment to our readers is to provide a venue where a meaningful dialog 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 Author Online remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The Author Online 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 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.

Part 1

Keeping a Tidy Computer

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.

A thorough understanding of the material in this part is a solid foundation for every reader to take on the rapidly growing Docker and container ecosystem. Like the Docker tool set itself, the pieces covered here provide building blocks to solving larger problems. For that reason, I suggest that you try to resist the urge to skip ahead. It may take some time to get to the specific question that is on your mind, but I'm confident that you'll have more than a few revelations along the way.

Welcome to Docker

This chapter covers

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

If you’re anything like me, you prefer to do only what is necessary to accomplish an unpleasant or mundane task. It’s likely that you’d prefer tools that are simple to use to great effect over those that are complex or time-consuming. If I’m right, then I think you’ll be interested in learning about Docker.

Suppose you like to try out new Linux software but are worried about running something malicious. Running that software with Docker is a great first step in protecting your computer because Docker helps even the most basic software users take advantage of powerful security tools.

If you’re a system administrator, making Docker the cornerstone of your software management toolset will save time and let you focus on high-value activities because Docker minimizes the time that you’ll spend doing mundane tasks.

If you write software, distributing your software with Docker will make it easier for your users to install and run it. Writing your software in a Docker-wrapped development environment will save you time configuring or sharing that environment, because from the perspective of your software, every environment is the same.

Suppose you own or manage large-scale systems or data centers. Creating build, test, and deployment pipelines is simplified using Docker because moving any software through such a pipeline is identical to moving any other software through.

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 high-value activities. You can use Docker with network applications like web servers, databases, and mail servers and with terminal applications like text editors, compilers, network analysis tools, and scripts; in some cases it's even used to run GUI applications like web browsers and productivity software.

NOT JUST LINUX Docker is Linux software but works well on most operating systems.

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

1.1 **What is Docker?**

Docker is a command-line program, a background daemon, and a set of remote services that take a logistical approach to solving common software problems and simplifying your experience installing, running, publishing, and removing software. It accomplishes this using a UNIX technology called containers.

1.1.1 **Containers**

Historically, UNIX-style operating systems have used the term *jail* to describe a modified runtime environment for a program that prevents that program from accessing protected resources. Since 2005, after 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 preventing access to protected resources 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, and misconfigured containers have lulled others into a false sense of security. We need a solution to this problem, 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. As Docker and its container engines improve, you get the latest and greatest jail features. Instead of keeping up with the rapidly evolving and highly technical world of building strong application jails, you can let Docker handle the bulk of that for you. This will save you a lot of time and money and bring peace of mind.

1.1.2 Containers are not virtualization

Without Docker, businesses typically use hardware virtualization (also known as virtual machines) to provide isolation. Virtual machines provide virtual 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 copy of an operating system in addition to the software you want to use.

Unlike virtual machines, Docker containers don't use hardware virtualization. Programs running inside Docker containers interface directly with the host's Linux kernel. Because there's no additional layer between the program running inside the container and the computer's operating system, no resources are wasted by running redundant software or simulating virtual hardware. This is an important distinction. Docker is not a virtualization technology. Instead, it helps you use the container technology already built into your operating system.

1.1.3 Running software in containers for isolation

As noted earlier, containers 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 1.1 shows a basic example running on a simplified computer system architecture.

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,

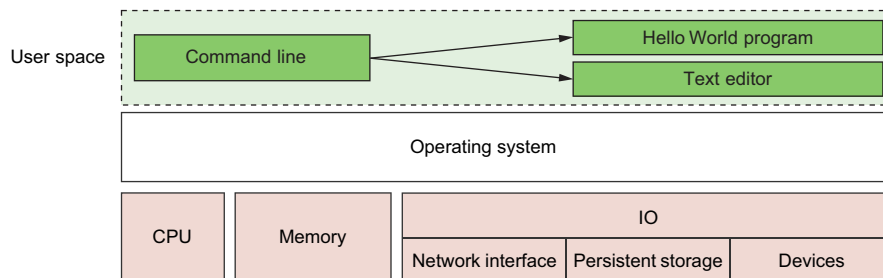


Figure 1.1 A basic computer stack running two programs that were started from the command line

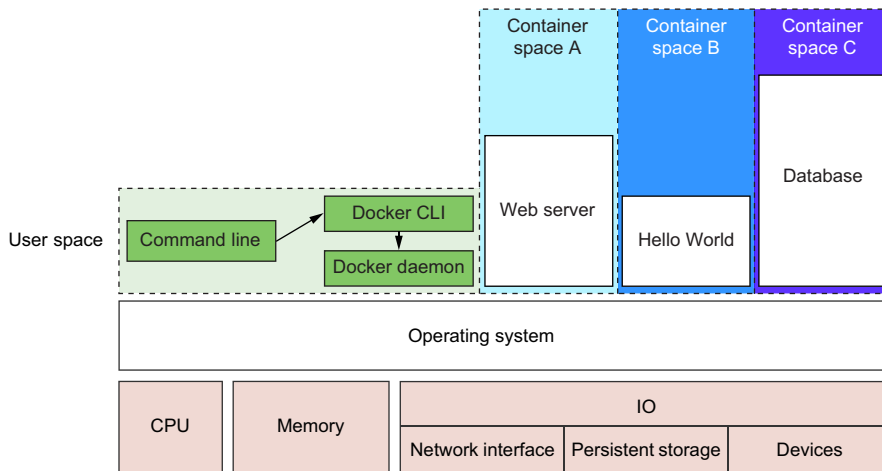


Figure 1.2 Docker running three containers on a basic Linux computer system

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

Figure 1.2 also shows three running containers. Each is running as a child process of the Docker daemon, 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.

The containers that Docker builds are isolated with respect to eight aspects. Part 1 of this book covers each of these aspects through an exploration of Docker container features. The specific aspects are as follows:

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

Linux namespaces and cgroups take care of containers at runtime. Docker uses another set of technologies to provide containers for files that act like shipping containers.

1.1.4 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. 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*.

A Docker image is a bundled snapshot of all the files that should be available to a program running inside a container. You can create as many containers from an image as you want. But when you do, containers that were started from the same image don't share changes to their file system. When you distribute software with Docker, you distribute these images, and the receiving computers create containers from them. Images are the shippable units in the Docker ecosystem.

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

1.2 What problems does Docker solve?

Using software is complex. Before installation you have to consider what operating system you're using, the resources the software requires, what other software is already installed, and what other software it depends on. You need to decide where it should be installed. Then you need to know how to install it. It's surprising how drastically installation processes vary today. The list of considerations is long and unforgiving. Installing software is at best inconsistent and overcomplicated.

Most computers have more than one application installed and running. And most applications have dependencies on other software. What happens when two or more applications you want to use don't play well together? Disaster. Things are only made more complicated when two or more 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 simple truth is that the more software you use, the more difficult it is to manage. Even if you can spend the time and energy required to figure out installing and running applications, how confident can you be about your security? Open and closed source programs release security updates continually, and being aware of all of the issues is often impossible. The more software you run, the greater the risk that it's vulnerable to attack.

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

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

1.2.1 Getting organized

Without Docker, a computer can end up looking like a junk drawer. Applications have all sorts of dependencies. Some applications depend on specific system libraries for common things like sound, networking, graphics, and so on. Others depend on standard libraries for the language they're written in.

Some depend on other applications, such as how a Java program depends on the Java Virtual Machine or a web application might depend on a database. It's common for a running program to require exclusive access to some scarce resource such as a network connection or a file.

Today, without Docker, applications are spread all over the file system and end up creating a messy web of interactions. Figure 1.3 illustrates how example applications depend on example libraries without Docker.

Docker keeps things organized by isolating everything with containers and images. Figure 1.4 illustrates these same applications and their dependencies running inside

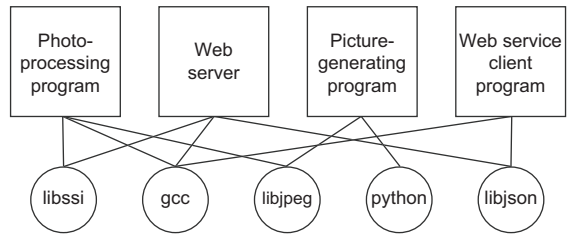


Figure 1.3 Dependency relationships of example program

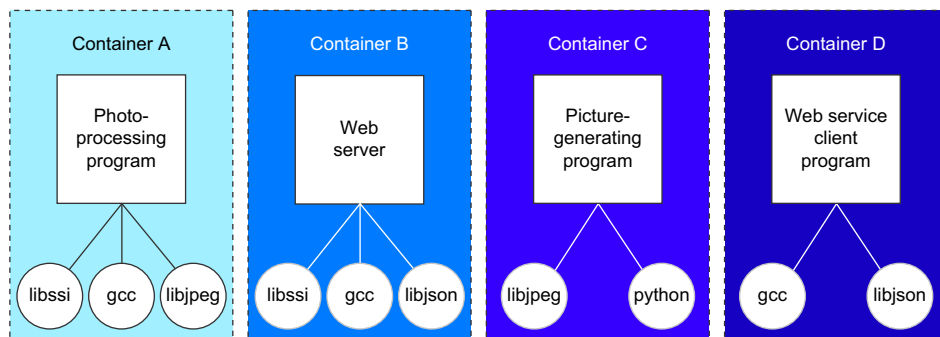


Figure 1.4 Example programs running inside containers with copies of their dependencies

containers. With the links broken and each application neatly contained, understanding the system is an approachable task.

1.2.2 Improving portability

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

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

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

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

regardless of the language it was written in, the operating system it was designed for, or the state of the environment where it's running.

1.2.3 *Protecting your computer*

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

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

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

Like physical jail cells, anything inside a container can only access things that are inside it as well. There are exceptions to this rule but only when explicitly created by the user. Containers limit the scope of impact that a program can have on other running programs, the data it can access, and system resources. Figure 1.5 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 defense in-depth strategy. It is far too commonly skipped or implemented in a half-hearted manner.

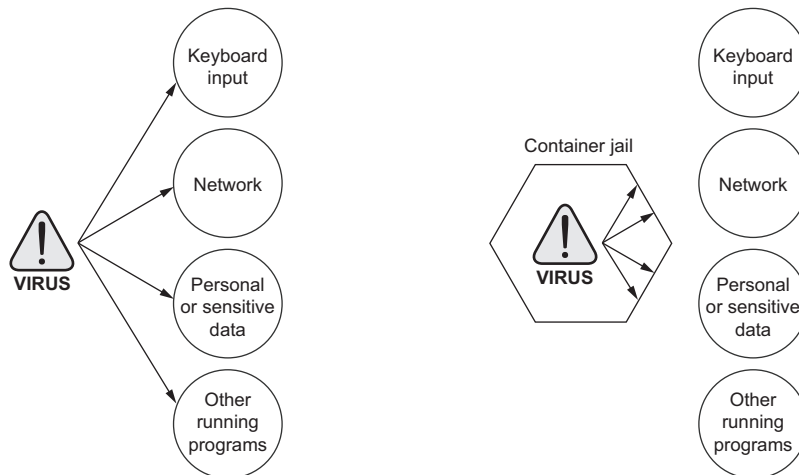


Figure 1.5 Left: a malicious program with direct access to sensitive resources. Right: a malicious program inside a container.

1.3 Why is Docker important?

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

This is also the case for application removal. When you want to remove software, you simply tell Docker which software to remove. No lingering artifacts will remain because they were all carefully contained and accounted for. 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 will change the system administration and software development landscape. Docker is important because it makes containers available to everyone. Using it saves time, money, and energy.

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

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

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

1.4 Where and when to use Docker

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

Docker *can* run almost anywhere, but that doesn't mean you'll want to do so. For example, currently Docker can only run applications that can run on a Linux operating

system. This means that if you want to run an OS X or Windows native application, you can't yet do so through Docker.

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

Beyond the added in-depth benefit of defense, using Docker for day-to-day tasks helps keep your computer clean. Keeping a clean computer will prevent you from running into shared resource issues and ease software installation and removal. That same ease of installation, removal, and distribution simplifies management of computer fleets and could radically change the way companies think about maintenance.

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

1.5 **Example: “Hello, World”**

I like to get people started with an 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/installation/>. OS X and Windows users will install the full Docker suite of applications using the Docker Toolbox. Once you have Docker installed and an active internet connection, head to your command prompt and type the following:

```
docker run dockerinaction/hello_world
```

TIP Docker runs as the root user on your system. On some systems you'll need to execute the `docker` command line using `sudo`. Failing to do so will result in a permissions error message. You can eliminate this requirement by creating a “docker” group, setting that group as the owner of the docker socket, and adding your user to that group. Consult the Docker online documentation for your distribution for detailed instructions, or try it both ways and stick with the option that works for you. For consistency, this book will omit the `sudo` prefix.

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

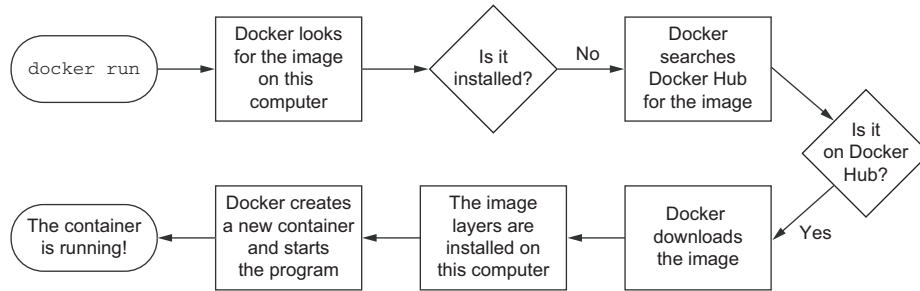


Figure 1.6 What happens after running `docker run`

“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 to start a new container. This single command triggers a sequence (shown in figure 1.6) that installs, runs, and stops a program inside a container.

Second, the program that you tell it to run in a container is `dockerinaction/hello_world`. This is called the repository (or image) name. For now, you can think of the repository name as the name of the program you want to install or run.

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. Any suggestions you have on how they might be improved are always welcome.

The first time you give the command, Docker has to figure out if `dockerinaction/hello_world` is already installed. If it’s unable to locate it on your computer (because it’s the first thing you do with Docker), Docker makes a call to Docker Hub. Docker Hub is a public registry provided by Docker Inc. Docker Hub replies to Docker running on your computer where `dockerinaction/hello_world` can be found, and Docker starts the download.

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

```
echo "hello world"
```

After the command prints “hello world” to the terminal, it exits, and the container is automatically 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 runs the program again.

When you give the command a second time, Docker will check again to see if `dockerinaction/hello_world` is installed. This time it finds it and can build a new

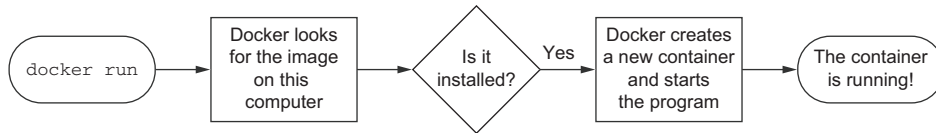


Figure 1.7 Running `docker run` a second time. Because the image is already installed, Docker can start the new container right away.

container and execute it right away. I want to emphasize an important detail. When you use `docker run` the second time, it creates a second container from the same repository (figure 1.7 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.

Congratulations! You're now an official Docker user. Take a moment to reflect on how straightforward that was.

1.6 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, a background daemon, and a set of remote services. It's integrated with community tools provided by Docker Inc.
- The container abstraction is at the core of its logistical approach.
- Working with containers instead of software creates a consistent interface and enables the development of more sophisticated tools.
- Containers help keep your computers tidy because software inside containers can't interact with anything outside those containers, and no shared dependencies can be formed.
- Because Docker is available and supported on Linux, OS X, and Windows, most software packaged in Docker images can be used on any computer.
- Docker doesn't provide container technology; it hides the complexity of working directly with the container software.

Running software in containers

This chapter covers

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

Before the end of this chapter you'll understand all the basics for working with containers and how Docker helps solve clutter and conflict problems. You're going to work through examples that introduce Docker features as you might encounter them in daily use.

2.1 *Getting help with the Docker command line*

You'll use the `docker` command-line program throughout the rest of this book. To get you started with that, I want to show you how to get information about

commands from the `docker` program itself. This way you'll understand how to use the exact version of Docker on your computer. Open a terminal, or command prompt, and run the following command:

```
docker help
```

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

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

```
docker help cp
```

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

2.2 **Controlling containers: building a website monitor**

Most examples in this book will use real software. Practical examples will help introduce Docker features and illustrate how you will use them in daily activities. In this first example, you're going to install a web server called NGINX. Web servers are programs that make website files and programs accessible to web browsers over a network. You're not going to build a website, but you are going to install and start a web server with Docker. If you follow the instructions in this example, the web server will be available only to other programs on your computer.

Suppose a new client walks into your office and makes you an outrageous offer to build them a new website. They want a website that's closely monitored. This particular client wants to run their own operations, so they'll want the solution you provide to email their team when the server is down. They've also heard about this popular web server software called NGINX and have specifically requested that you use it. Having read about the merits of working with Docker, you've decided to use it for this project. Figure 2.1 shows your planned architecture for the project.

This example uses three containers. The first will run NGINX; the second will run a program called a mailer. Both of these will run as detached containers. *Detached* means that the container will run in the background, without being attached to any input or output stream. A third program, called an agent, will run in an interactive container. Both the mailer and 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

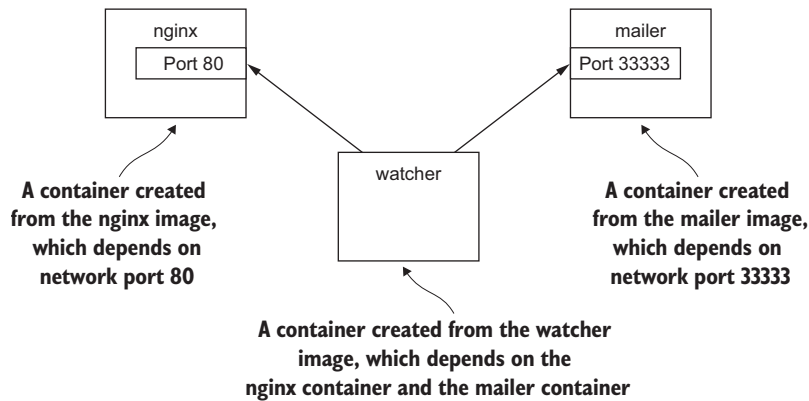


Figure 2.1 The three containers that you'll build in this example

- 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.

2.2.1 Creating and starting a new container

When installing software with Docker, we say that we're installing an *image*. There are different ways to install an image and several sources for images. Images are covered in depth in chapter 3. In this example we're going to download and install an image for NGINX from Docker Hub. Remember, Docker Hub is the public registry provided by Docker Inc. The NGINX image is from what Docker Inc. calls a trusted repository. Generally, the person or foundation that publishes the software controls the trusted repositories for that software. Running the following command will download, install, and start a container running NGINX:

```
docker run --detach \
  --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 container will get a similar unique identifier. It's common for users to capture this output

with a variable for use with other commands. You don't need to do so for the purposes of this example. After the identifier is displayed, it might not seem like anything has happened. That's because you used the `--detach` option and started the program in the background. This means that the program started but isn't attached to your terminal. It makes sense to start NGINX this way because we're going to run a few different programs.

Running detached containers is a perfect fit for programs that sit quietly in the background. That type of program is called a *daemon*. A daemon generally interacts with other programs or humans over a network or some other communication tool. 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 is a mailer. A mailer waits for connections from a caller and then sends an email. The following command will install and run a mailer that will work for this example:

```
docker run -d \  
  --name mailer \  
  nginx
```

 **Start detached**


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

2.2.2 *Running interactive containers*

Programs that interact with users tend to feel more interactive. A terminal-based text editor is a great example. The `docker` command-line tool is a perfect example of an interactive terminal program. These types of programs might take input from the user or display output on the terminal. 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:latest /bin/sh
```

 **Create a virtual terminal
and bind stdin**

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

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

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

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

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

It's possible to create an interactive container, manually start a process inside that container, and then detach your terminal. You can do so by holding down the 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 last example and send a message with the mailer if the web server stops. This command will start the agent in an interactive container using the short-form flags:

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

← Create a virtual terminal
and bind stdin

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

```
System up.
```

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

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

2.2.3 ***Listing, stopping, restarting, and viewing output of containers***

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

```
docker ps
```

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

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

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

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

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

```
docker logs web
```

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

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

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

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

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

The logs for mailer should look something like this:

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

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

```
System up.
```

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

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

```
docker stop web
docker logs mailer
```

Wait a couple seconds and check the mailer logs

Stop the web server by stopping the container

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

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

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

Learning the basic Docker features is one thing, but understanding why they're useful and how to use them in building more comprehensive systems is another task entirely. The best place to start learning that is with the process identifier namespace provided by Linux.

2.3 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 the set of possible numbers that identify processes. Linux provides facilities 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. From the perspective of a process in one namespace, PID 1 might refer to an init system process like `runit` or `supervisord`. In a different namespace, PID 1 might refer to a command shell like `bash`. Creating a PID namespace for each container is a critical feature of Docker. Run the following to see it in action:

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

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

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

PID	USER	COMMAND
1	root	/bin/sh -c sleep 30000
5	root	sleep 30000
6	root	ps

Command 2 above should generate a slightly different process list:

PID	USER	COMMAND
1	root	/bin/sh -c nc -l -p 0.0.0.0:80
7	root	nc -l -p 0.0.0.0:80
8	root	ps

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

Without a PID namespace, the processes running inside a container would share the same ID space as those in other containers or on the host. A container would be able to determine what other processes were running on the host machine. Worse, namespaces transform many authorization decisions into domain decisions. That means processes in one container might be able to control processes in other containers. Docker would be much less useful without the PID namespace. The Linux features that Docker uses, such as namespaces, help you solve whole classes of software problems.

Like most Docker isolation features, you can optionally create containers without their own PID namespace. 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:latest ps
```

← Should list all processes running on the computer

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;'
```

← The output should be empty

← Start a second nginx process in the same container

The last command should display output like:

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

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

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

← Start the first nginx instance

```
docker logs webA
```

← Verify that it is working, should be empty

Start the second instance →

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

```
docker logs webB
```

← Verify that it is working, should be empty

To generalize ways that programs might conflict with each other, let's consider a parking lot metaphor. A paid parking lot has a few basic features: a payment system, a few reserved parking spaces, and numbered spaces.

Tying these features back to a computer system, a payment system represents some shared resource with a specific interface. A payment system might accept cash or credit cards or both. People who carry only cash won't be able to use a garage with a payment system that accepts only credit cards, and people without money to pay the fee won't be able to park in the garage at all.

Similarly, programs that have a dependency on some shared component such as a specific version of a programming language library won't be able to run on computers that either have a different version of that library or lack that library completely. Just like if two people who each use a different payment method want to park in the same garage that accepts only one method, conflict arises when you want to use two programs that require different versions of a library.

Reserved spaces in this metaphor represent scarce resources. Imagine that the parking garage attendant assigns the same reserved space to two cars. As long as only one driver wanted to use the garage at a time, there would be no issue. But if both wanted to use the space simultaneously, the first one in would win and the second wouldn't be able to park. As you'll see in the conflict example in section 2.7, this is the same type of conflict that happens when two programs try to bind to the same network port.

Lastly, consider what would happen if someone changed the space numbers in the parking lot while cars were parked. When owners return and try to locate their vehicles, they may be unable to do so. Although this is clearly a silly example, it's a great metaphor for what happens to programs when shared environment variables change. Programs often use environment variables or registry entries to locate other resources that they need. These resources might be libraries or other programs. When programs conflict with each other, they might modify these variables in incompatible ways.

Here are some common conflict problems:

- Two programs want to bind to the same network port.
- Two programs use the same temporary filename, and file locks are preventing that.
- Two programs want to use different versions of some globally installed library.
- Two copies of the same program 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.

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

2.4 ***Eliminating metaconflicts: building a website farm***

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

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

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

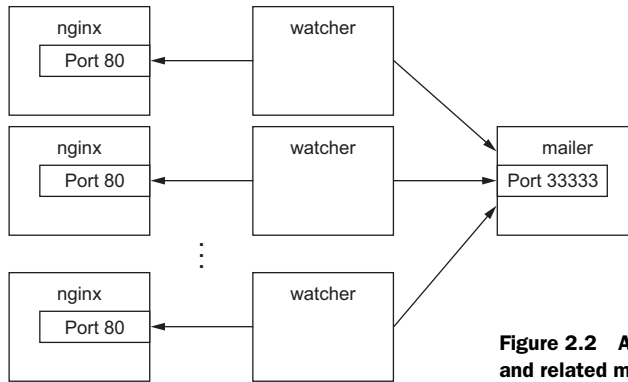


Figure 2.2 A fleet of web server containers and related monitoring agents

2.4.1 Flexible container identification

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

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

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

← Create a container named "webid"

← Create another container named "webid"

The second command here will fail with a conflict error:

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

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

```
docker rename webid webid-old
```

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

← Rename the current web container to "webid-old"

← Create another container named "webid"

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

```
7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

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

```
docker exec \
    7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5 \
ps

docker stop \
    7cb5d2b9a7eab87f07182b5bf58936c9947890995b1b94f412912fa822a9ecb5
```

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

```
docker exec 7cb5d2b9a7ea ps
docker stop 7cb5d2b9a7ea
```

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

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

```
docker create nginx
```

The result should be a line like:

```
b26a631e536d3caae348e9fd36e7661254a11511eb2274fb55f9f7c788721b0d
```

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

```
CID=$(docker create nginx:latest)
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 that the script was launched in. 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
```

← Create a new stopped container

```
cat /tmp/web.cid
```

← Inspect the file

Like the use of shell variables, this feature increases the opportunity for conflict. The name of the CID file (provided after `--cidfile`) must be known or have some known structure. Just like manual container naming, this approach uses known names in a global (Docker-wide) namespace. The good news is that Docker won't create a new container using the provided CID file if that file already exists. The command will fail just as it does when you create two containers with the same name.

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

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

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

```
CID=$(docker ps --latest --quiet)
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.

2.4.2 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)

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

Make sure mailer from first example is running

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

```
docker ps
```

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

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

```
docker ps -a
```

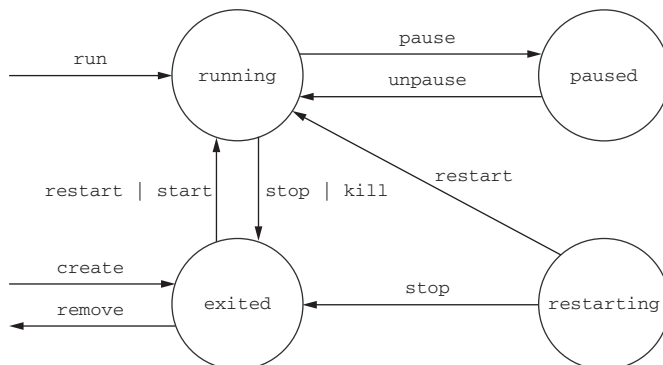


Figure 2.3 The state transition diagram for Docker containers as reported by the status column

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

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

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

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

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

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

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

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

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

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

WEB_CID=$(docker run -d nginx)

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

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

They’ve decided to focus on building their websites with WordPress (a popular open source content-management and blogging program). Luckily, WordPress is published through Docker Hub in a repository named `wordpress:4`. 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.

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

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

2.5 ***Building environment-agnostic systems***

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

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

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

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

WordPress uses a database program called MySQL to store most of its data, so it’s a good idea to start with making sure that a container running WordPress has a read-only file system.

2.5.1 ***Read-only file systems***

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

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

```
docker run -d --name wp --read-only wordpress:4
```

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