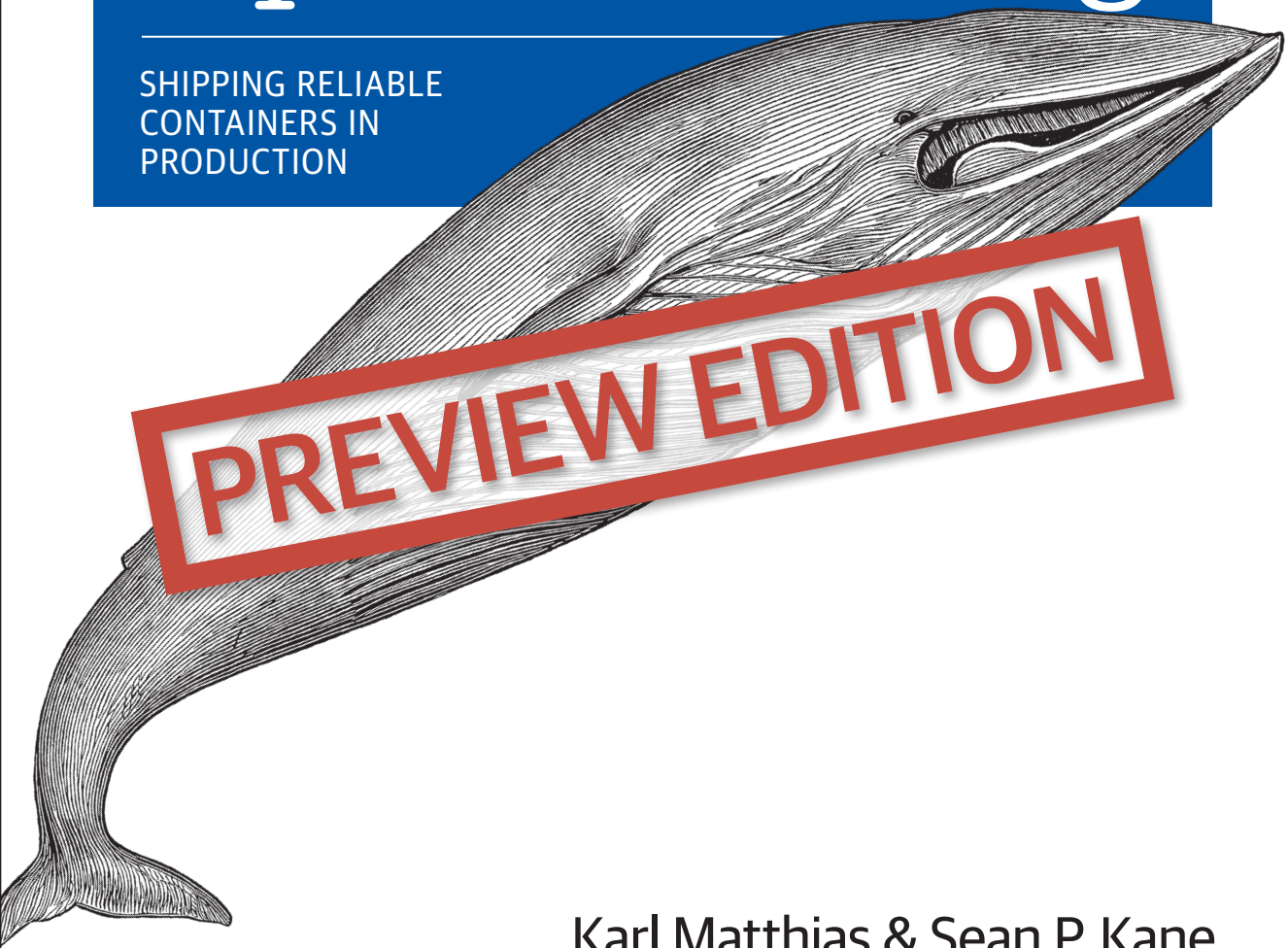
O'REILLY®

Docker
Up & Running

SHIPPING RELIABLE CONTAINERS IN PRODUCTION

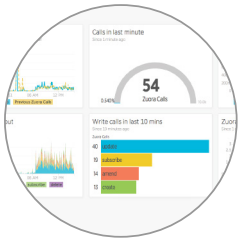


PREVIEW EDITION

Karl Matthias & Sean P. Kane

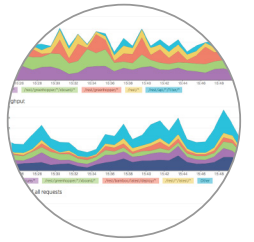
Relying on Docker?

Use New Relic to manage the performance of all your apps.

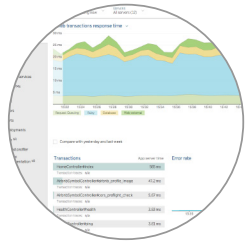


Mobile Developers
End-to-end visibility, 24/7 alerting, and crash analysis.

Front-end Developers
Deep insights into your browser-side app's engine.



IT Operations
Faster delivery.
Fewer bottlenecks.
More stability.



App Owners
Track engagement.
Pinpoint issues.
Optimize usability.

Move from finger-pointing blame to data-driven accountability.
Find the truth with a single source of data from multiple views.
newrelic.com/docker

New Relic.

This Preview Edition of *Docker: Up and Running*, *Chapters 1–3*, is a work in progress. The final book is currently scheduled for release in June 2015 and will be available at *oreilly.com* and other retailers once it is published.

FIRST EDITION

Docker: Up and Running

Karl Matthias and Sean P. Kane

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Docker: Up and Running
by Karl Matthias and Sean P. Kane

Printed in the United States of America.

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

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

Editor: Brian Anderson
Production Editor: Melanie Yarbrough
Copyeditor: Gillian McGarvey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2015: First Edition

Revision History for the First Edition
2015-03-18: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491917572* for release details.

978-1-491-91757-2

[LSI]

To my wife and children who make everything worth it.

—Sean P. Kane

For my Mom who got me to read and my Dad who read to me. And for my wife and daughters who are my bedrock.

—Karl Matthias

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/docker-up-and-running*.

Don't forget to update the link above.
To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

We'd like to send a heartfelt thanks to the many people who helped make this book possible:

- Nic Benders and Bjorn Freeman-Benson, at New Relic, who went far above and beyond in supporting this effort, and who ensured that we had time to pursue it.
- Dana Lawson at New Relic who enthusiastically supported Karl's contribution.
- Laurel Ruma at O'Reilly who initially reached out to us about writing a Docker book, and Mike Loukides who helped get everything on track.
- Gillian McGarvey and Melanie Yarbrough, for their efforts copy-writing the manuscript, and helping it appear like we were actually paying attention in our high school English classes.
- A special thanks to our editor, Brian Anderson, who ensured that we knew what we were getting into, and guided us along every step of the way.
- All of our peers at New Relic, who have been along for the whole Docker ride and provided us with much of the experience that's reflected here.
- World Cup Coffee, McMenamins Ringlers Pub, and Old Town Pizza in Portland, OR, who kindly let us use their tables and power, long after our dishes were empty.
- Our draft reviewers, who helped ensure that we were on the right track at various points throughout the writing process: Ksenia Burlachenko, who gave us our very first reviewand Andrew T. Bakers.
- A special call out is due to Alice Goldfuss and Tom Offermann who gave us detailed and consistently useful feedback.
- Our families, for being supportive and giving us the required quiet time when we needed it.
- *Sean*: My dad, who brought the first computer we ever had into the house, and my mother for always encouraging I pursue my passions.

CHAPTER 1

Introduction

The Birth of Docker

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

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

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

Why Read This Book?

Today, there are many conversations, projects, and articles on the Internet involving Docker. So why should you devote precious hours to reading this book?

Even though there is a lot of information out there, Docker is a new technology and it is evolving incredibly quickly. Even during the time that that it took us to write this book, Docker, Inc., released four versions of Docker plus a few new tools into their ecosystem. Getting your arms around the scope of what Docker provides, under-

standing how it fits into your workflow, and getting integration right are not trivial tasks. Few companies or engineering teams have been running it in production for more than a year.

We[1] have worked for over a year and a half helping to deploy a production Docker platform, within the Site Engineering team at New Relic. We have had the opportunity to implement Docker early in its life cycle and can share what we have learned with you, so that you can enjoy the wins while avoiding many of the bumps in the road that we experienced. While the online documentation for the Docker project is useful, we'll attempt to give you a bigger picture and expose you to many of the best practices that we have learned along the way.

You should finish this book with enough information to understand what Docker is, how it's important, how to get it running, how to deploy your applications with it, and be armed with pointers towards the next steps you can take. It will hopefully be a quick trip through an interesting technology with some very practical applications.

The Promise of Docker

While ostensibly viewed as a virtualization platform, Docker is far more than that. Docker's domain spans a few crowded segments of the industry that include technologies like KVM, Xen, OpenStack, Mesos, Capistrano, Fabric, Ansible, Chef, Puppet, SaltStack, and so on. There is something very telling about the list of products that Docker competes with, and maybe you've spotted it already. For example, most engineers would not say that virtualization products compete with configuration management tools, yet both technologies are being disrupted by Docker. The technologies in that list are also generally acclaimed for their ability to improve productivity and that's what is causing a great deal of the buzz. Docker sits right in the middle of some of the most enabling technologies of the last decade.

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

Lots of new technologies come and go and a dose of skepticism about the newest rage is always healthy. Without digging deeper, it would be easy to dismiss Docker as just another technology that solves a few very specific problems for developers or opera-

1 The authors, Karl and Sean

tions teams. If you look at Docker as a virtualization or deployment technology alone, it might not seem very compelling. But Docker is much more than it seems on the surface.

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

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

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

Beyond helping with communication issues, Docker is opinionated about software architecture in a way that encourages more robustly crafted applications. Its architectural philosophy centers around atomic or throw-away containers. During deployment, the whole running environment of the old application is thrown away with it. Nothing in the environment of the application will live longer than the application itself and that's a simple idea with big repercussions. It means applications are not likely to accidentally rely on artifacts left by a previous release. It means that ephem-

eral debugging changes are less likely to live on in future releases that picked them up from the local filesystem. And it means that applications are highly portable between servers because all state has to be included directly into the deployment artifact and be immutable, or sent to an external dependency like a database, cache, or file server.

This leads to applications that are not only more scalable, but more reliable. Instances of the application container can come and go with little repercussion on the uptime of the frontend site. These are proven architectural choices that have been successful for non-Docker applications, but the design choices included in Docker's own design mean that Dockerized applications will follow these best practices by requirement and that's a good thing.

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

Packaging software in a way that leverages the skills developers already have.
: Many companies have had to create positions for release and build engineers in order to manage all the knowledge and tooling required to create software packages for their supported platforms. Tools like rpm, mock, dpkg, and pbuilder can be complicated to use, and each one must be learned independently. Docker wraps up all your requirements together into one package that is defined in a single file.

Bundling application software and required OS filesystems together in a single standardized image format.
: In the past, you typically needed to package not only your application, but many of the dependencies that it relied on, including libraries and daemons. However, you couldn't ever ensure that 100% of the execution environment was identical. All of this made packaging difficult to master, and hard for many companies to accomplish reliably. Often someone running Scientific Linux would resort to trying to deploy a community package tested on Red Hat Linux hoping that the package was close enough to what they needed. With Docker you deploy your application along with every single file required to run it. Docker's layered images make this an efficient process that ensures your application is running in the expected environment.

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

When developers commit changes to a version control system, a new Docker image can be built, which can go through the whole testing process and be deployed to production without any need to recompile or repackage at any step in the process.

Abstracting software applications from the hardware without sacrificing resources.

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

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

In the first year, newcomers to the project were surprised to find out that Docker wasn't already production-ready, but a steady stream of commits from the open source Docker community has moved the project forward at a very brisk pace. That pace seems to only pick up steam as time goes on. As Docker has now moved well into the 1.x release cycle, stability is good, production adoption is here, and many companies are looking to Docker as a solution to some of the serious complexity issues that they face in their application delivery processes.

What Docker Isn't

Docker can be used to solve a wide breadth of challenges that other categories of tools have traditionally been enlisted to fix; however, Docker's breadth of features often means that it lacks depth in specific functionality. For example, some organizations will find that they can completely remove their configuration management tool when they migrate to Docker, but the real power of Docker is that although it can replace some aspects of more traditional tools, it is usually completely compatible with them as well. In the following list, we explore some of the tool categories that Docker

doesn't directly replace but that can often be used in conjunction to achieve great results:

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

Cloud Platform (Openstack, CloudStack, etc.)
: Like Enterprise virtualization, the container workflow shares a lot of similarities on the surface with cloud platforms. Both are traditionally leveraged to allow applications to be horizontally scaled in response to changing demand. Docker, however, is not a cloud platform. It only handles deploying, running, and managing containers on pre-existing docker hosts. It doesn't allow you to create new host systems (instances), object stores, block storage, and the many other resources that are typically associated with a cloud platform.

Configuration Management (Puppet, Chef, etc.)
: Although Docker can significantly improve an organization's ability to manage applications and their dependencies, it does not directly replace more traditional configuration management. Dockerfiles are used to define how a container should look at build time, but they do not manage the container's ongoing state, and cannot be used to manage the Docker host system.

Deployment Framework (Capistrano, Fabric, etc.)
: Docker eases many aspects of deployment by creating self-contained container images that encapsulate all the dependencies of an application and can be deployed, in all environments, without changes. However, Docker can't be used to automate a complex deployment process by itself. Other tools are usually still needed to stitch together the larger workflow automation.

Workload Management Tool (Mesos, Fleet, etc.)
: The Docker server does not have any internal concept of a cluster. Additional orchestration tools (including Docker's own Swarm tool) must be used to coordinate work intelligently across a pool of Docker hosts and track the current state of all the hosts and their resources, and keep an inventory of running containers.

Development Environment (Vagrant, etc.)
: Vagrant is a virtual machine management tool for developers that is often used to simulate server stacks that closely resemble the production environment an application is destined to be deployed in. Among other things, Vagrant makes it

easy to run Linux software on Mac OS X and Windows-based workstations. Since the Docker server only runs on Linux, Docker provides boot2docker and Docker Machine to allow developers to quickly launch Linux-based Docker machines on various platforms. Boot2docker and Docker Machine are sufficient for many standard Docker workflows, but they don't provide the breadth of features found in Vagrant.

CHAPTER 2

Docker at a Glance

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

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

Process Simplification

Docker can simplify both workflows and communication, and that usually starts with the deployment story. Traditionally, the cycle of getting an application to production often looks something like the following (illustrated in Figure 2-1):

1. Application developers request resources from operations engineers.
2. Resources are provisioned and handed over to developers.
3. Developers script and tool their deployment.
4. Operations engineers and developers tweak the deployment repeatedly.
5. Additional application dependencies are discovered by developers.
6. Operations engineers work to install the additional requirements.
7. Loop over steps 5 and 6 *N* more times.
8. The application is deployed.

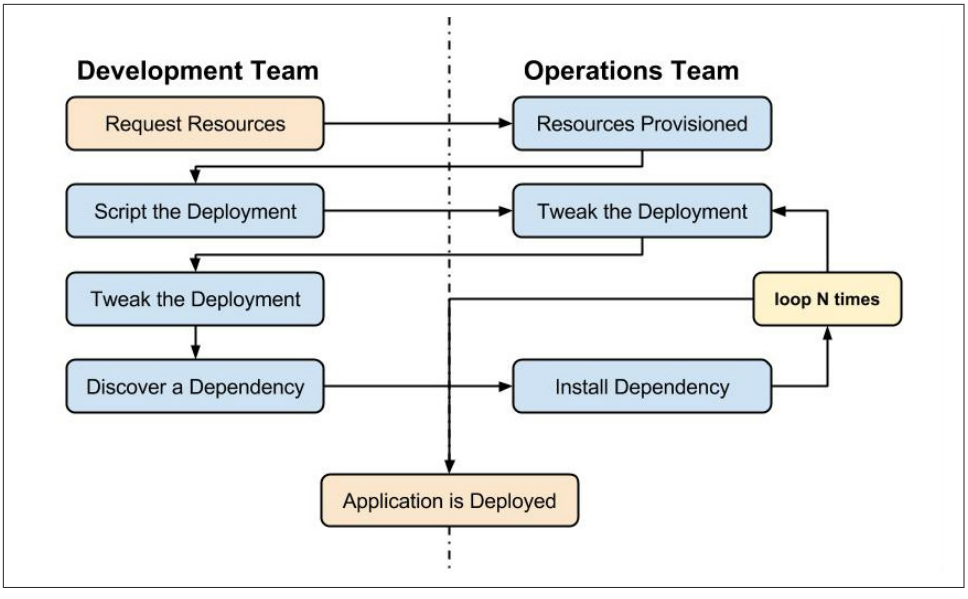


Figure 2-1. A non-Docker deployment workflow

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

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

Docker doesn't try to be Heroku, but it provides a clean separation of responsibilities and encapsulation of dependencies, which results in a similar boost in productivity. It also allows even more fine-grained control than Heroku by putting developers in control of everything down to the OS distribution on which they ship their application.

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

By using an image repository as the hand-off point, Docker allows the responsibility of building the application image to be separated from the deployment and operation of the container.

What that means in practice is that development teams can build their application with all of its dependencies, run it in development and test environments, and then just ship the exact same bundle of application and dependencies to production. Because those bundles all look the same from the outside, operations engineers can then build or install standard tooling to deploy and run the applications. The cycle described in Figure 2-1 then looks somewhat like this (illustrated in Figure 2-2):

1. Developers build the Docker image and ship it to the registry
2. Operations engineers provide configuration details to the container and provision resources
3. Developers trigger deployment

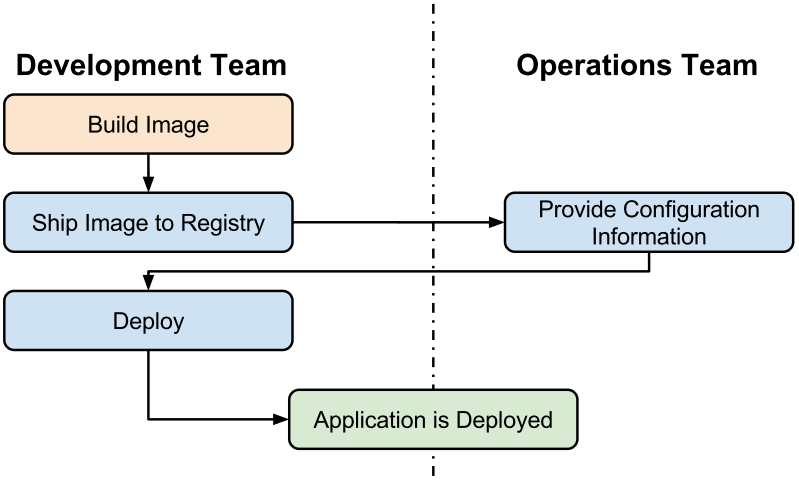


Figure 2-2. A Docker deployment workflow

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

Broad Support and Adoption

Docker is increasingly well supported, with the majority of the large public clouds announcing at least some direct support for it. For example, Docker runs on AWS Elastic Beanstalk, Google AppEngine, IBM Cloud, Microsoft Azure, Rackspace Cloud, and many more. At DockerCon 2014, Google's Eric Brewer announced that Google would be supporting Docker as its primary internal container format. Rather than just being good PR for these companies, what this means for the Docker community is that there is starting to be a lot of money backing the stability and success of the Docker platform.

Further building its influence, Docker's containers are becoming the common format between cloud providers, offering the potential for "write once, run anywhere" cloud applications. When Docker released their `libswarm` development library at DockerCon 2014, an engineer from Orchard demonstrated deploying a Docker container to a heterogeneous mix of cloud providers at the same time. This kind of orchestration has not been easy before, and it seems likely that as these major companies continue to invest in the platform, the support and tooling will improve correspondingly.

So that covers Docker containers and tooling, but what about OS vendor support and adoption? The Docker client runs directly on most major operating systems, but because the Docker server uses Linux containers, it does not run on non-Linux systems. Docker has traditionally been developed on the Ubuntu Linux distribution, but today most Linux distributions and other major operating systems are now supported where possible.

Docker is barely two years old, but with such broad support across many platforms, there is a lot of hope for its continued growth into the future.

Architecture

Docker is a powerful technology and that often means that something comes with a high level of complexity. But the fundamental architecture of Docker is a simple client/server model, with only one executable that acts as both components, depending on how you invoke the `docker` command. Underneath those simple exteriors, Docker heavily leverages kernel mechanisms such as iptables, virtual bridging, cgroups, namespaces, and various filesystem drivers. We'll talk about some of these when we

reach the Advanced Topics chapter towards the end of the book. For now, we'll go over how the client and server work and give a brief introduction to the network layer that sits underneath a Docker container.

Client/Server Model

Docker consists of at least two parts: the client and the server. Optionally there is a third component called the registry, which stores Docker images and metadata about those images. The server does the ongoing work of running and managing your containers and you use the client to tell the server what to do. The docker daemon can run on any number of servers in the infrastructure, and a single client can address any number of servers. Clients drive all of the communication, but Docker servers can talk directly to image registries when told to do so by the client. Clients are responsible for directing servers what to do, and servers focus on hosting containerized applications.

Docker is a little different in structure from some other client/server software. Instead of having separate client and server executables, it uses the same binary for both components. When you install Docker, you get both components but the server will only launch on a supported Linux host. Launching the Docker server/daemon, is as simple as running `docker` with the `-d` command-line argument, which tells it to act like a daemon and listen for incoming connections. Each Docker host will normally have one Docker daemon running that can manage a number of containers. You can then use the `docker` command-line tool client to talk to the server.

Network Ports and Unix Sockets

The `docker` command-line tool and `docker -d` daemon talk to each other over network sockets. You can choose to have the Docker daemon listen on one or more TCP or Unix sockets. It's possible, for example, to have Docker listen on both a local Unix socket and two different TCP ports (encrypted and nonencrypted). On many Linux distributions, that is actually the default. If you want to only be able to access Docker from the local system, listening only on the Unix socket would be the most secure option. However, most people want to talk to the `docker` daemon remotely, so it usually listens on at least one TCP port.

The original TCP port that `docker` was configured to use was 4243, but that port was never registered and in fact was already used by other tools such as the Mac OS X backup client CrashPlan. As a result, Docker registered their own TCP port with IANA and it's now generally configured to use TCP port 2375 when running unencrypted, or 2376 when handling encrypted traffic. In Docker 1.3 and later, the default is to use the encrypted port on 2376, but this is easily configurable. The Unix socket is located in different paths on different operating system so you should check

where yours is located. If you have strong preferences, you can usually specify this at install time. If you don't, then the defaults will probably work for you.

Robust Tooling

Among the many things that have led to Docker's growing adoption is its simple and powerful tooling. This has been expanding ever wider since its initial release by Docker themselves, and by the Docker community at large. The tooling that Docker ships with supports both building Docker images and basic deployment to individual Docker daemons, as well as all the functionality needed to actually manage a remote Docker server. Community efforts have focused on managing whole fleets (or clusters) of Docker servers and the scheduling and orchestrating of container deployments. Docker has also launched their own orchestration toolset, including Compose (previously known as Fig), Machine, and Swarm.

Because Docker provides both a command-line tool and a remote web API, it is easy to add additional tooling in any language. The command-line tool lends itself well to scripting and a lot of power can easily be leveraged with simple shell script wrappers around the command-line tool.

Docker Command-Line Tool

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

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

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

Application Programming Interface (API)

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

Extensive documentation for the API is on the Docker site. As the ecosystem has matured, robust implementations of Docker API libraries have begun to appear for many popular languages. We've used the Go and Ruby libraries, for example, and have found them to be both robust and rapidly updated as new versions of Docker are released.

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

Container Networking

Even though Docker containers are largely made up of processes running on the host system itself, they behave quite differently from other processes at the network layer. If you think of each of your Docker containers as behaving on the network like a host on a private network, you'll be on the right path. The Docker server acts as a virtual bridge and the containers are clients behind it. A bridge is just a network device that repeats traffic from one side to another. So you can think of it like a mini virtual network with hosts attached.

The implementation is that each container has its own virtual ethernet interface connected to the Docker bridge and its own IP address allocated to the virtual interface. Docker lets you bind ports on the host to the container so that the outside world can reach your container. That traffic passes over a proxy that is also part of the Docker daemon before getting to the container.

Docker allocates the private subnet from an unused RFC 1918 block. It detects which network blocks are unused on startup and allocates one to the virtual network. That is bridged to the host's local network through an interface on the server called `docker0`. That means that all of the containers are on a network together and can talk to each other directly. But to get to the host or the outside world, they go over the `docker0` virtual bridge interface. As we mentioned, inbound traffic goes over the

proxy. This proxy is fairly high performance but can be limiting if you run high throughput applications in containers. We talk more about this in Chapter 10, and offer some solutions.

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

workflow to the docker definitely get started with the defaults, you might later find that you don't want or need this virtual network. You can disable all of this with a single switch at `docker` daemon startup time. It's not configurable per container, but you can turn it off for all containers using the `--net host` switch. When running in that mode, Docker containers just use the host's own network devices and address.

Best Use Cases

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

To begin with, Docker's architecture aims it squarely at applications that are either stateless or where the state is externalized into data stores like databases or caches. It enforces some good development principles for this class of application and we'll talk later about how that's powerful. But it means that doing things like putting a database engine inside Docker is basically like trying to swim against the current. It's not that you can't do it, or even that you shouldn't do it; it's just that it's not the most obvious use case for Docker and if it's the one you start with you may find yourself disappointed early on.

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

Containers Are Not Virtual Machines

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

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

Containers Are Lightweight

We'll get more into the details of how this works later, but creating a container takes very little space. A quick test on Docker 1.4.1 reveals that a newly created container from an existing image takes a whopping 12 kilobytes of disk space. That's pretty lightweight. One the other hand, a new virtual machine created from a golden image might require hundreds or thousands of megabytes. The reason that the new container is so small is because it is just a reference to a layered filesystem image and some metadata about the configuration.

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

Limited Isolation

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

It's often the case that many containers share one or more common filesystem layers. That's one of the more powerful design decisions in Docker but it also means that if you update a shared image, you'll need to re-create a number of containers.

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

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

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

Stateless Applications

A good example of the kind of application that containerizes well is a web application that keeps its state in a database. You might also run something like ephemeral memcache instances in containers. If you think about your web application, though, it probably has local state that you rely on, like configuration files. That might not seem like a lot of state, but it means that you've limited the reusability of your container, and made it more challenging to deploy into different environments, without maintaining configuration data in your codebase.

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

With containers, you might also find that you are always decreasing the size of your containerized application as you optimize it down to the bare essentials required to run. We have found that thinking of anything that you need to run in a distributed

way as a container can lead to some interesting design decisions. If, for example, you have a service that collects some data, processes it, and returns the result, you might configure containers on many servers to run the job and then aggregate the response on another container.

Externalizing State

If Docker works best for stateless applications, how do you best store state when you need to? Configuration is best passed by environment variables, for example. Docker supports environment variables natively and they are stored in the metadata that makes up a container configuration. That means that restarting the container will ensure that the same configuration is passed to your application each time.

Databases are often where scaled applications store state and nothing in Docker interferes with doing that for containerized applications. Applications that need to store files, however, face some challenges. Storing things to the container's filesystem will not perform well, will be extremely limited by space, and will not preserve state across a container lifecycle. Applications that need to store filesystem state should be considered carefully before putting them into Docker. If you decide that you can benefit from Docker in these cases, it's best to design a solution where the state can be stored in a centralized location that could be accessed regardless of which host a container runs on. In certain cases, this might mean a service like Amazon S3, RiakCS, OpenStack Swift, a local block store, or even mounting iSCSI disks inside the container.

Although it is possible to externalize state on an attached filesystem, it is not generally encouraged by the community, and should be considered an advanced use case. It is strongly recommended that you start with applications that don't need persistent state.

The Docker Workflow

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

Revision Control

Filesystem Layers

The first thing that Docker gives you out of the box is two forms of revision control. The first is on the building of the container itself. Docker containers are made up of stacked filesystem layers, where each set of changes made during the build process are laid on top of the previous. That's great because it means that when you do a new build, you only have to rebuild the layers that include and build upon the change your deploying. That saves time and bandwidth because containers are shipped around as layers and you don't have to ship layers that a server already has stored. If you've done deployments with many classic deployment tools, you know that you can end up shipping hundreds of megabytes of the same data to a server over and over at each deployment. That's slow, and worse, you can't really be sure exactly what changed between deployments. Because of the layering effect, and because Docker containers include all of the application dependencies, you can be quite sure where the changes happened.

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

Image Tags

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

But what you find in any scaled production environment is that each application has something different about the way in which this is handled. Or many are the same and one is different. Worse, in heterogeneous language environments, the deployment tools are often entirely different between applications and very little is shared. So the question of "What was the previous version?" can have many answers depending on whom you ask and about which application. Docker has a built-in mechanism for handling this: it provides image tagging at deployment time. You can leave multiple revisions of your application on the server and just tag them at release. This is not rocket science, and it's not functionality that is hard to find in other deployment tool-

ing, as we mention. But it can easily be made standard across all of your applications and everyone can have the same expectations about how things will be tagged for all applications.

In many examples on the Internet and in this book, you will see people use the `latest` tag. This is useful when getting started and when writing examples, as it will always grab the most recent version of a image. But since this is a floating tag, it is a bad idea to use `latest` in most workflows, as your dependencies can get updated out from under you, and it is impossible to roll back to `latest` because the old version is no longer the one tagged `latest`.

Building

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

The Docker command-line tool contains a `build` flag that will consume a Dockerfile and produce a Docker image. Each command in a Dockerfile generates a new layer in the image, so it's easy to reason about what the build is going to do by looking at the Dockerfile itself. The great part of all of this standardization is that any engineer who has worked with a Dockerfile can dive right in and modify the build of any other application. Because the Docker image is a standardized artifact, all of the tooling behind the build will be the same regardless of the language being used, the OS distribution it's based on, or the number of layers needed.

Most Docker builds are a single invocation of the `docker build` command and generate a single artifact, the container image. Because it's usually the case that most of the logic about the build is wholly contained in the Dockerfile, it's easy to create standard build jobs for any team to use in build systems like Jenkins. As a further standardization of the build process, a few companies, including eBay, actually have standardized Docker containers to do the image builds from a Dockerfile.

Testing

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

Testing a production application can take many forms, from unit testing to full integration testing in a semi-live environment. Docker facilitates better testing by guaranteeing that the artifact that passed testing will be the one that ships to production.

That can be guaranteed because we can either use the Docker SHA for the container, or a custom tag to make sure we're consistently shipping the same version of the application.

The second part of the testing story is that all testing that is run against the container will automatically include testing the application with all of the dependencies that it will ship with. If a unit test framework says tests were successful against a container image, you can be sure that you will not experience a problem with the versioning of an underlying library at deployment time, for example. That's not easy with most other technologies, and even Java WAR files, for example, don't include testing of the application server itself. That same Java application deployed in a Docker container will generally also include the application server, and the whole stack can be smoke tested before shipping to production.

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

Packaging

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

Deploying

Deployments are handled by so many kinds of tools in different shops that it would be impossible to list them here. Some of these tools include shell scripting, Capistrano, Fabric, Ansible, or in-house custom tooling. In our experience with multi-team organizations, there is usually one or two people on each team who know the magic incantation to get deployments to work. When something goes wrong, the

team is dependent on them to get it running again. As you probably expect by now, Docker makes most of that a non-issue. The built-in tooling supports a simple, one-line deployment strategy to get a build onto a host and up and running. The standard Docker client only handles deploying to a single host at a time, but there are other tools avaliable that make it easy to deploy into a cluster of Docker hosts. Because of the standardization provided by Docker, your build can be deployed into any of these systems with low complexity on the part of the development teams.

The Docker Ecosystem

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

The first important category of tools that adds functionality to the core Docker distribution contains orchestration and mass deployment tools like Docker's Swarm, New Relic's Centurion and Spotify's Helios. All of these take a generally simple approach to orchestration. For more complex environments, Google's Kubernetes and Apache Mesos are more powerful options. There are new tools shipping constantly as new adopters discover gaps and publish improvements.

Additional categories include auditing, logging, mapping, and many other tools, the majority of which leverage the Docker API directly. Recent announcements include direct support for Docker logs in Mozilla's Heka log router, for example.

The results of the broad community that is rapidly evolving around Docker is anyone's guess, but it is likely that this support will only accelerate Docker's adoption and the development of robust tools that solve many of the problems that the community struggles with.

Atomic Hosts

One additional category of software that can enhance your Docker experience is atomic hosts. Traditionally, servers and virtual machines are systems that an organization will carefully assemble, configure, and maintain to provide a wide variety of functionality that supports a broad range of usage patterns. Updates must often be applied via nonatomic operations and there are frequently many ways in which host configuration can diverge and introduce unexpected behavior into the system.

What if you could extend some of the core container usage patterns all the way down into the operating system? Instead of relying on configuration management to try and keep all of your OS components in sync, what if you could simply pull down a

new, thin OS image and reboot the server? And then if something breaks, easily roll back to the exact image you were previously using?

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

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

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

Wrap Up

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

CHAPTER 3

Installing Docker

The steps required to install Docker vary depending on the primary platform you use for development and the Linux distribution that you use to host your applications in production. Since Docker is a technology built around Linux containers, people developing on non-Linux platforms will need to use some form of virtual machine or remote server for many parts of the process.

In this chapter, we discuss the steps required to get a fully working Docker development environment set up on most modern desktop operating systems.

Although the Docker client can run on Windows and Mac OS X to control a Docker Server, Docker containers can only be built and launched on Linux. Therefore, non-Linux systems will require a virtual machine or remote server to host the Linux-based Docker server.

Important Terminology

Atomic host
: An atomic host is a small, finely tuned operating system image that supports container hosting and atomic OS upgrades.

Docker client
: The `docker` command used to control most of the Docker workflow and talk to remote Docker servers.

Docker server
: The `docker` command run in daemon mode. This basically turns a Linux system into a Docker server that can have containers deployed, launched, and torn down via a remote client.

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

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

Docker Client

The Docker client natively supports 64-bit versions of Linux and Mac OS X due to the Unix underpinnings of both operating systems. There have been reports of people getting Docker to run on 32-bit systems, but it is not currently supported.

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

On Mac OS X and Microsoft Windows, native GUI installers provide the easiest method to install and maintain prepackaged software. On Mac OS X, homebrew is also a very popular option among technical users.

To develop with Docker on non-Linux platforms, you will need to leverage virtual machines or remote Linux hosts to provide a Docker server. Boot2Docker and Vagrant, which are discussed later in this chapter, provide one approach to resolving this issue.

Linux

It is strongly recommended that you run Docker on a very recent release of your prefered Linux distribution. It is possible to run Docker on some older releases, but stability may be an significant issue. The directions below assume you are using a recent release.

Ubuntu Linux 14.04 (64-bit)

To install Linux on a current installation, run the following commands:

```
$ sudo apt-get update
$ sudo apt-get install docker.io
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
$ sudo sed -i '$acomplete -F _docker docker' /etc/bash_completion.d/docker.io
$ source /etc/bash_completion.d/docker.io
```

Fedora Linux 21 (64-bit)

To install the correct Docker packages on your system, run the following command:

```
$ sudo yum -y install docker-io
```

If you get a *Cannot start container* error, try running `sudo yum upgrade selinux-policy` and then reboot your system.

Older Fedora releases have a pre-existing package called docker, which is a KDE and GNOME2 system tray replacement docking application for WidowMaker. In newer versions of Fedora, this package has been renamed to wmdocker.

Mac OS X 10.10

GUI Installer

Download the latest Boot2Docker installer and then double-click on the downloaded program icon. Follow all of the installer prompts until the installation is finished.

The installer will also install Virtualbox, which Mac OS X requires to launch Linux virtual machines that can build Docker images and run containers.

Homebrew Installation

To install using the popular Homebrew package management system for Mac OS X, you must first install Homebrew. The Homebrew project suggests installing the software with the following command:

```
$ ruby -e \
"$(curl -fsSL \
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Running random scripts from the Internet on your system is not considered wise. It is unlikely that this script has been altered in a malicous manner, but a wise soul would read through the script first, or consider an alternative installation option.

If you already have Homebrew installed on your system, you should update it and all the installed formulas by running:

```
$ brew update
```

To install VirtualBox via Homebrew, you need to add support for an additional Homebrew repository that contains many GUI and large binary applications for Mac OS X. This can be done with one simple command:

```
$ brew install caskroom/cask/brew-cask
```

You can find more information about Homebrew Casks at caskroom.io.

Now that you have Homebrew and Cask installed and the newest software formulas have been downloaded, you can install VirtualBox with the following command:

```
$ brew cask install virtualbox
```

And then installing Docker is as simple as running:

```
$ brew install docker
$ brew install boot2docker
```

Microsoft Windows 8

Download the latest Boot2Docker installer and then double-click on the downloaded program icon. Follow all of the installer prompts, until the installation is finished.

The installer will also install Virtualbox, which Microsoft Windows requires to launch Linux virtual machines that can build Docker images and run containers.

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

Docker Server

The Docker server is integrated into the same binary that you use as the client. Running the Docker daemon manually, on a Linux system is a simple as typing something like this:

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

This command tells Docker to start in daemon mode (`-d`), create and listen to a Unix domain socket (`-H unix:///var/run/docker.sock`), and bind to all system IP addresses using the default unencrypted traffic port for docker (`-H tcp://0.0.0.0:2375`).

On non-Linux systems, you will need to set up a Linux-based virtual machine to host the Docker server.

Systemd-Based Linux

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

```
$ sudo systemctl enable docker
```

This tells systemd to enable the docker service and start it when the system boots or switches into the default runlevel.

To start the docker server, type the following:

```
$ sudo systemctl start docker
```

Upstart-Based Linux

Ubuntu uses the `upstart` init daemon although future versions are very likely going to be converting to `systemd`. Upstart replaces the traditional Unix init system with an event-driven model and supports vastly simplified init scripts, sometimes only a few lines long.

To enable the `docker` server to start when the system boots, type:

```
$ sudo update-rc.d docker.io defaults
```

To start the service immediately, you can use:

```
$ service docker.io start
```

init.d-Based Linux

Many Linux distributions used on production servers are still using a more traditional `init.d` system. If you are running something based on the Red Hat 6 ecosystem, amoung others, then you can likely use commands similar to the following to control when the `docker` server runs.

Enable the Docker service at boot with:

```
$ chkconfig docker on
```

Start the Docker service immediately:

```
$ service docker start
```

or:

```
$ /etc/init.d/docker start
```

Non-Linux VM-Based Server

If you are using Microsoft Windows or Mac OS X in your Docker workflow, the default installation provides Virtualbox and Boot2Docker, so that you can set up a Docker server for testing. These tools allow you to boot an Ubuntu-based Linux virtual machine on your local system.

Boot2Docker

To initialize Boot2Docker and download the required boot image, run the following command the first time you use Boot2Docker. You should see output similar to what is displayed below.

```
$ boot2docker init
Latest release for boot2docker/boot2docker is v1.3.1
Downloading boot2docker ISO image...
Success: downloaded https://github.com/.../boot2docker.iso
  to /Users/me/.boot2docker/boot2docker.iso
Generating public/private rsa key pair.
Your identification has been saved in /Users/me/.ssh/id_boot2docker.
Your public key has been saved in /Users/me/.ssh/id_boot2docker.pub.
The key fingerprint is:
ce:dc:61:42:fe:e1:7b:af:2d:d6:00:57:bc:ff:66:5d me@my-mbair.local
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|                E|
|          .    o.|
|         . B    .|
|        o ++oo.o+|
|         S+=+=..o|
|         .=.+..+o|
|             .o  |
|            o.   |
+-----------------+
```

Now you can start up a virtual machine with a running Docker daemon. By default, Boot2Docker will map port 2376 on your local host to the secure Docker port 2376 on the virtual machine, to make it easier to interact with the Docker server from your local system.

```
$ boot2docker up
Waiting for VM and Docker daemon to start...
...........ooo
Started.
```

```
Writing /Users/me/.boot2docker/certs/boot2docker-vm/ca.pem
Writing /Users/me/.boot2docker/certs/boot2docker-vm/cert.pem
Writing /Users/me/.boot2docker/certs/boot2docker-vm/key.pem

To connect the Docker client to the Docker daemon, please set:
    export DOCKER_TLS_VERIFY=1
    export DOCKER_HOST=tcp://172.17.42.10:2376
    export DOCKER_CERT_PATH=/Users/skane/.boot2docker/certs/boot2docker-vm
```

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

```
$ $(boot2docker shellinit)
```

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

```
$ docker info
Containers: 0
Images: 0
Storage Driver: aufs
 Root Dir: /mnt/sda1/var/lib/docker/aufs
 Dirs: 0
Execution Driver: native-0.2
Kernel Version: 3.16.4-tinycore64
Operating System: Boot2Docker 1.3.1 (TCL 5.4); ...
Debug mode (server): true
Debug mode (client): false
Fds: 10
Goroutines: 11
EventsListeners: 0
Init Path: /usr/local/bin/docker
```

To connect to a shell on your Boot2Docker-based virtual machine, you can use the following command:

```
$ boot2docker ssh
Warning: Permanently added '[localhost]:2022' (RSA) to the list of known hosts.
                        ##        .
                  ## ## ##       ==
               ## ## ## ##      ===
           /""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
           \______ o          __/
             \    \        __/
              \____\______/
 _                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 1.3.1, build master : 9a31a68 - Fri Oct 31 03:14:34 UTC 2014
```

```
Docker version 1.3.1, build 4e9bbfa
docker@boot2docker:~$
```

Docker Machine

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

The easiest way to install Docker Machine is to visit the GitHub releases page and download the correct binary for your operating system and architecture. Currently, there are versions for 32- and 64-bit versions of Linux, Windows, and Mac OS X.

For these demonstrations, you will also need to have a recent release of a hypervisor, like Virtualbox, installed on your system.

For this section, you will use a Unix-based system with Virtualbox for the examples.

First you need to download and install the `docker-machine` executable:

```
$ curl -L -o docker-machine \
https://github.com/docker/machine/releases/download/\
v0.1.0/docker-machine_darwin-amd64
$mkdir ~/bin
$ cp docker-machine ~/bin
$ export PATH=${PATH}:~/bin
$ chmod u+rx ~/bin/docker-machine
```

Once you have the `docker-machine` executable in your path, you can start to use it to set up Docker hosts. The next thing that you need to do is create a named Docker machine. You can do this using the `docker-machine create` command:

```
$ docker-machine create --driver virtualbox local
INFO[0000] Creating CA: /Users/skane/.docker/machine/certs/ca.pem
INFO[0001] Creating client certificate: /.../.docker/machine/certs/cert.pem
INFO[0000] Downloading boot2docker.iso to /.../machine/cache/boot2docker.iso...
INFO[0001] Creating SSH key...
INFO[0001] Creating VirtualBox VM...
INFO[0013] Starting VirtualBox VM...
INFO[0014] Waiting for VM to start...
INFO[0061] "local" has been created and is now the active machine.
INFO[0061] To point your Docker client at it, run this in your shell:
  $(docker-machine env local)
```

This downloads a Boot2Docker image and then creates a Virtualbox virtual machine that you can use as a Docker host. If you look at the output from the `create` command, you will see that it instructs you to run the following command:

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

This command has no output, so what does it do exactly? If you run it without the surrounding `$()`, you will see that it sets a couple of environment variables in our current shell that tell the Docker client where to find the Docker server:

```
$ docker-machine env local
export DOCKER_TLS_VERIFY=yes
export DOCKER_CERT_PATH=/Users/me/.docker/machine/machines/local
export DOCKER_HOST=tcp://172.17.42.10:2376
```

And now if you want to confirm what machines you have running, you can use the following command:

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM
local   *        virtualbox   Running   tcp://172.17.42.10:2376
```

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

Now you can pass commands to the new Docker machine by leveraging the regular `docker` command, since you have set the proper environment variables. If you did not want to set these environment variables, you could also use the `docker` and `docker-machine` commands in conjunction with one another, like so:

```
$ docker $(docker-machine config local) ps
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES
```

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

```
$ docker-machine config local
  --tls --tlscacert=/Users/me/.docker/machine/machines/local/ca.pem \
  --tlscert=/Users/me/.docker/machine/machines/local/cert.pem \
  --tlskey=/Users/me/.docker/machine/machines/local/key.pem \
  -H="tcp://172.17.42.10:2376"
```

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

```
$ docker-machine ip
172.17.42.10
```

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

```
$ docker-machine ssh local
                        ##         .
                  ## ## ##        ==
               ## ## ## ##       ===
           /""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
           \______ o          __/
             \    \        __/
              \____\______/
 _                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 1.5.0, build master : a66bce5 - Tue Feb 10 ... UTC 2015
Docker version 1.5.0, build a8a31ef
docker@local:~$ exit
```

To stop your Docker machine, you can run:

```
$ docker-machine stop local
```

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

```
$ docker-machine start local
INFO[0000] Waiting for VM to start...
```

Some of the documentation states that if you run `docker-machine stop` without specifying a machine name, the command will execute on the active machine as identified in the output of `docker-machine ls`. This does not seem to actually be the case in the current release.

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

Vagrant

If you need more flexibility with the way your Docker development environment is set up you might want to consider using Vagrant instead of Boot2Docker. Vagrant provides many advatages, including support for multiple hypervisors, infinite virtual machine images, and much more.

A common use case for leveraging Vagrant during Docker development is to support testing on images that match your production enviroment. Vagrant supports everything from broad distributions like CentOS 7 and Ubuntu 14.04 to finely-focused atomic host distributions like CoreOS and Project Atomic.

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

In the following example, you will create a CoreOS-based Docker host running the Docker daemon on the unencrypted port 2375.

In production, Docker should always be set up to only use encrypted remote connections. Althought Boot2Docker now uses encrypted communications by default, setting up Vagrant to do this in CoreOS is currently a bit too complicated for this installation example.

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

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

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

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

Inside the *coreos-vagrant* directory, create a new file called `config.rb` (using your favorite editor) that contains the following:

```
$expose_docker_tcp=2375
```

Next you'll need to leverage the built-in cloud-init tool to add some systemd unit files that will enable the Docker daemon on TCP port 2375. You can do this by creating a file called `user-data` that contains the following:

```
#cloud-config

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

        [Socket]
        ListenStream=2375
        BindIPv6Only=both
```

```
        Service=docker.service

        [Install]
        WantedBy=sockets.target
    - name: enable-docker-tcp.service
      command: start
      content: |
        [Unit]
        Description=Enable the Docker Socket for the API

        [Service]
        Type=oneshot
        ExecStart=/usr/bin/systemctl enable docker-tcp.socket
```

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

```
$ vagrant up
Bringing machine 'core-01' up with 'virtualbox' provider...
==> core-01: Box 'coreos-alpha' could not be found. Attempting to find and install...
    core-01: Box Provider: virtualbox
    core-01: Box Version: >= 308.0.1
==> core-01: Loading metadata ... 'http://.../coreos_production_vagrant.json'
    core-01: URL: http://.../coreos_production_vagrant.json
==> core-01: Adding box 'coreos-alpha' (v472.0.0) for provider: virtualbox
    core-01: Downloading: http://.../coreos_production_vagrant.box
    core-01: Calculating and comparing box checksum...
==> core-01: Successfully added box 'coreos-alpha' (v472.0.0) for 'virtualbox'!
==> core-01: Importing base box 'coreos-alpha'...
==> core-01: Matching MAC address for NAT networking...
==> core-01: Checking if box 'coreos-alpha' is up to date...
==> core-01: Setting the name of the VM: coreos-vagrant_core-01
==> core-01: Clearing any previously set network interfaces...
==> core-01: Preparing network interfaces based on configuration...
    core-01: Adapter 1: nat
    core-01: Adapter 2: hostonly
==> core-01: Forwarding ports...
    core-01: 2375 => 2375 (adapter 1)
    core-01: 22 => 2222 (adapter 1)
==> core-01: Running 'pre-boot' VM customizations...
==> core-01: Booting VM...
==> core-01: Waiting for machine to boot. This may take a few minutes...
    core-01: SSH address: 127.0.0.1:2222
    core-01: SSH username: core
    core-01: SSH auth method: private key
    core-01: Warning: Connection timeout. Retrying...
==> core-01: Machine booted and ready!
==> core-01: Setting hostname...
==> core-01: Configuring network adapters within the VM...
==> core-01: Running provisioner: file...
==> core-01: Running provisioner: shell...
    core-01: Running: inline script
```

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

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

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

```
$ docker info
Containers: 0
Images: 0
Storage Driver: btrfs
Execution Driver: native-0.2
Kernel Version: 3.16.2+
Operating System: CoreOS 472.0.0
```

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

```
$ vagrant ssh
CoreOS (alpha)
core@core-01 ~ $
```

Test the Setup

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

This step is important to ensure that all the peices are properly installed and communicating with each other as expected.

If you want to run these commands on the server, be sure that you prepend each `docker` command with `sudo`.

Ubuntu

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

Fedora

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

CentOS

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

`ubuntu:latest`, `fedora:latest`, and `centos:latest` all represent a Docker image name followed by an image tag.

Wrap Up

Now that you have a running Docker setup, you can start to look at more than the basic mechanics of getting it installed. In the next chapter, you'll explore some of the basic functionality of Docker with some hands-on work.

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

Docker recently acquired Kitematic, which has released a beta version of a very simple Docker GUI for Mac OS X. If you are on this platform and interested, you can download it from their site.