

O'REILLY®



Free Sampler

# Using Docker

---

DEVELOPING AND DEPLOYING SOFTWARE WITH CONTAINERS

Adrian Mouat

# Using Docker

Docker containers offer simple, fast and robust methods for developing, distributing and running software, especially in dynamic and distributed environments. With this hands-on guide, you'll learn *why* containers are so important, *what* you'll gain by adopting Docker, and *how* to make it part of your development process.

Ideal for developers, operations engineers, and system administrators—especially those keen to embrace a DevOps approach—*Using Docker* will take you from basics to running dozens of containers on a multi-host system with networking and scheduling. The core of the book walks you through the steps needed to develop, test, and deploy a web application with Docker.

- Get started with Docker by building and deploying a simple web application
- Use Continuous Deployment techniques to push your application to production multiple times a day
- Learn various options and techniques for logging and monitoring multiple containers
- Examine networking and service discovery: how do containers find each other and how do you connect them?
- Orchestrate and cluster containers to address load-balancing, scaling, failover, and scheduling
- Secure your system by following the principles of defense-in-depth and least privilege
- Exploit containers to build microservice architectures

**Adrian Mouat** is Chief Scientist at Container Solutions. He's worked on a wide range of software projects, from small web apps to large-scale data analysis software.

“*Using Docker* is a detailed practical guide for the Docker ecosystem as containerized microservice applications move from dev/test to production.”

— **Adrian Cockcroft**  
Technology Fellow, Battery Ventures

“*Using Docker* is a deep and comprehensive overview of Docker and the container ecosystem. A practical focus with lots of examples makes it easy to apply the concepts and techniques to real projects.”

— **Pini Reznik**  
CTO, Container Solutions

SYSTEM ADMINISTRATION

US \$59.99

CAN \$68.99

ISBN: 978-1-491-91576-9



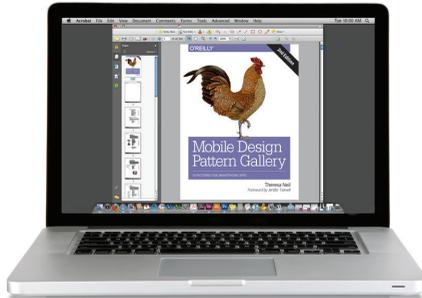
Twitter: @oreillymedia  
facebook.com/oreilly

# O'Reilly ebooks.

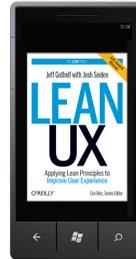
Your bookshelf on your devices.



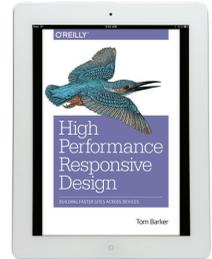
PDF



Mobi



ePub



DAISY

When you buy an ebook through [oreilly.com](http://oreilly.com) you get lifetime access to the book, and whenever possible we provide it to you in four DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

**Learn more at [ebooks.oreilly.com](http://ebooks.oreilly.com)**

You can also purchase O'Reilly ebooks through the iBookstore, the [Android Marketplace](http://AndroidMarketplace), and [Amazon.com](http://Amazon.com).

**O'REILLY®**

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 15055

## Using Docker

by Adrian Mouat

Copyright © 2016 Adrian Mouat. All rights reserved.

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](mailto:corporate@oreilly.com).

**Editor:** Brian Anderson

**Production Editor:** Melanie Yarbrough

**Copyeditor:** Christina Edwards

**Proofreader:** Amanda Kersey

**Indexer:** WordCo Indexing Services

**Interior Designer:** David Futato

**Cover Designer:** Randy Comer

**Illustrator:** Rebecca Demarest

December 2015: First Edition

### Revision History for the First Edition

2015-12-07: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491915769> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Using Docker*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

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

978-1-491-91576-9

[LSI]

---

# Table of Contents

**Preface**..... xi

---

## **Part I. Background and Basics**

**1. The What and Why of Containers**..... **3**

- Containers Versus VMs 4
- Docker and Containers 6
- Docker: A History 8
- Plugins and Plumbing 10
- 64-Bit Linux 10

**2. Installation**..... **13**

- Installing Docker on Linux 13
  - Run SELinux in Permissive Mode 14
  - Running Without sudo 15
- Installing Docker on Mac OS or Windows 15
- A Quick Check 17

**3. First Steps**..... **19**

- Running Your First Image 19
- The Basic Commands 20
- Building Images from Dockerfiles 24
- Working with Registries 27
  - Private Repositories 29
- Using the Redis Official Image 30
- Conclusion 33

<b>4. Docker Fundamentals.....</b>	<b>35</b>
The Docker Architecture	35
Underlying Technologies	36
Surrounding Technologies	37
Docker Hosting	39
How Images Get Built	39
The Build Context	39
Image Layers	41
Caching	43
Base Images	44
Dockerfile Instructions	46
Connecting Containers to the World	49
Linking Containers	49
Managing Data with Volumes and Data Containers	51
Sharing Data	53
Data Containers	54
Common Docker Commands	55
The run Command	56
Managing Containers	59
Docker Info	62
Container Info	62
Dealing with Images	63
Using the Registry	66
Conclusion	67

---

## Part II. The Software Lifecycle with Docker

<b>5. Using Docker in Development.....</b>	<b>71</b>
Say “Hello World!”	71
Automating with Compose	81
The Compose Workflow	83
Conclusion	84
<b>6. Creating a Simple Web App.....</b>	<b>85</b>
Creating a Basic Web Page	86
Taking Advantage of Existing Images	88
Add Some Caching	93
Microservices	96
Conclusion	97

<b>7. Image Distribution.....</b>	<b>99</b>
Image and Repository Naming	99
The Docker Hub	100
Automated Builds	102
Private Distribution	104
Running Your Own Registry	104
Commerical Registries	111
Reducing Image Size	111
Image Provenance	113
Conclusion	114
<b>8. Continuous Integration and Testing with Docker.....</b>	<b>115</b>
Adding Unit Tests to Identidock	116
Creating a Jenkins Container	121
Triggering Builds	128
Pushing the Image	129
Responsible Tagging	129
Staging and Production	131
Image Sprawl	131
Using Docker to Provision Jenkins Slaves	132
Backing Up Jenkins	132
Hosted CI Solutions	133
Testing and Microservices	133
Testing in Production	135
Conclusion	135
<b>9. Deploying Containers.....</b>	<b>137</b>
Provisioning Resources with Docker Machine	138
Using a Proxy	141
Execution Options	147
Shell Scripts	148
Using a Process Manager (or systemd to Rule Them All)	150
Using a Configuration Management Tool	153
Host Configuration	157
Choosing an OS	157
Choosing a Storage Driver	157
Specialist Hosting Options	160
Triton	160
Google Container Engine	162
Amazon EC2 Container Service	162
Giant Swarm	165
Persistent Data and Production Containers	167

Sharing Secrets	167
Saving Secrets in the Image	167
Passing Secrets in Environment Variables	168
Passing Secrets in Volumes	168
Using a Key-Value Store	169
Networking	170
Production Registry	170
Continuous Deployment/Delivery	171
Conclusion	171
<b>10. Logging and Monitoring.....</b>	<b>173</b>
Logging	174
The Default Docker Logging	174
Aggregating Logs	176
Logging with ELK	176
Docker Logging with syslog	187
Grabbing Logs from File	193
Monitoring and Alerting	194
Monitoring with Docker Tools	194
cAdvisor	196
Cluster Solutions	197
Commercial Monitoring and Logging Solutions	201
Conclusion	201

---

## Part III. Tools and Techniques

<b>11. Networking and Service Discovery.....</b>	<b>205</b>
Ambassadors	206
Service Discovery	210
etcd	210
SkyDNS	215
Consul	219
Registration	223
Other Solutions	225
Networking Options	226
Bridge	226
Host	227
Container	228
None	228
New Docker Networking	228
Network Types and Plugins	230

Networking Solutions	230
Overlay	231
Weave	233
Flannel	237
Project Calico	242
Conclusion	246
<b>12. Orchestration, Clustering, and Management.....</b>	<b>249</b>
Clustering and Orchestration Tools	250
Swarm	251
Fleet	257
Kubernetes	263
Mesos and Marathon	271
Container Management Platforms	282
Rancher	282
Clocker	283
Tutum	285
Conclusion	286
<b>13. Security and Limiting Containers.....</b>	<b>289</b>
Things to Worry About	290
Defense-in-Depth	292
Least Privilege	292
Securing Identidock	293
Segregate Containers by Host	295
Applying Updates	296
Avoid Unsupported Drivers	299
Image Provenance	300
Docker Digests	300
Docker Content Trust	301
Reproducible and Trustworthy Dockerfiles	305
Security Tips	307
Set a User	307
Limit Container Networking	309
Remove Setuid/Setgid Binaries	311
Limit Memory	312
Limit CPU	313
Limit Restarts	314
Limit Filesystems	314
Limit Capabilities	315
Apply Resource Limits (ulimits)	316
Run a Hardened Kernel	318

Linux Security Modules	318
SELinux	319
AppArmor	322
Auditing	322
Incident Response	323
Future Features	324
Conclusion	324
<b>Index.....</b>	<b>327</b>

---

# The What and Why of Containers

Containers are fundamentally changing the way we develop, distribute, and run software. Developers can build software locally, knowing that it will run identically regardless of host environment—be it a rack in the IT department, a user’s laptop, or a cluster in the cloud. Operations engineers can concentrate on networking, resources, and uptime and spend less time configuring environments and battling system dependencies. The use and uptake of containers is increasing at a phenomenal rate across the industry, from the smallest start ups to large-scale enterprises. Developers and operations engineers should expect to regularly use containers in some fashion within the next few years.

Containers are an encapsulation of an application with its dependencies. At first glance, they appear to be just a lightweight form of virtual machines (VMs)—like a VM, a container holds an isolated instance of an operating system (OS), which we can use to run applications.

However, containers have several advantages that enable use cases that are difficult or impossible with traditional VMs:

- Containers share resources with the host OS, which makes them an order of magnitude more efficient. Containers can be started and stopped in a fraction of a second. Applications running in containers incur little to no overhead compared to applications running natively on the host OS.
- The portability of containers has the potential to eliminate a whole class of bugs caused by subtle changes in the running environment—it could even put an end to the age-old developer refrain of “but it works on my machine!”
- The lightweight nature of containers means developers can run dozens of containers at the same time, making it possible to emulate a production-ready dis-

tributed system. Operations engineers can run many more containers on a single host machine than using VMs alone.

- Containers also have advantages for end users and developers outside of deploying to the cloud. Users can download and run complex applications without needing to spend hours on configuration and installation issues or worrying about the changes required to their system. In turn, the developers of such applications can avoid worrying about differences in user environments and the availability of dependencies.

More importantly, the fundamental goals of VMs and containers are different—the purpose of a VM is to fully emulate a foreign environment, while the purpose of a container is to make applications portable and self-contained.

## Containers Versus VMs

Though containers and VMs seem similar at first, there are some important differences, which are easiest to explain using diagrams.

**Figure 1-1** shows three applications running in separate VMs on a host. The hypervisor<sup>1</sup> is required to create and run VMs, controlling access to the underlying OS and hardware as well as interpreting system calls when necessary. Each VM requires a full copy of the OS, the application being run, and any supporting libraries.

In contrast, **Figure 1-2** shows how the same three applications could be run in a containerized system. Unlike VMs, the host's kernel<sup>2</sup> is shared with the running containers. This means that containers are always constrained to running the same kernel as the host. Applications Y and Z use the same libraries and can share this data rather than having redundant copies. The container engine is responsible for starting and stopping containers in a similar way to the hypervisor on a VM. However, processes running inside containers are equivalent to native processes on the host and do not incur the overheads associated with hypervisor execution.

Both VMs and containers can be used to isolate applications from other applications running on the same host. VMs have an added degree of isolation from the hypervisor and are a trusted and battle-hardened technology. Containers are comparatively new, and many organizations are hesitant to completely trust the isolation features of containers before they have a proven track record. For this reason, it is common to

---

1 The diagram depicts a *type 2* hypervisor, such as Virtualbox or VMWare Workstation, which runs on top of a host OS. *Type 1* hypervisors, such as Xen, are also available where the hypervisor runs directly on top of the bare metal.

2 The kernel is the core component in an OS and is responsible for providing applications with essential system functions related to memory, CPU, and device access. A full OS consists of the kernel plus various system programs, such as init systems, compilers, and window managers.

find hybrid systems with containers running inside VMs in order to take advantage of both technologies.

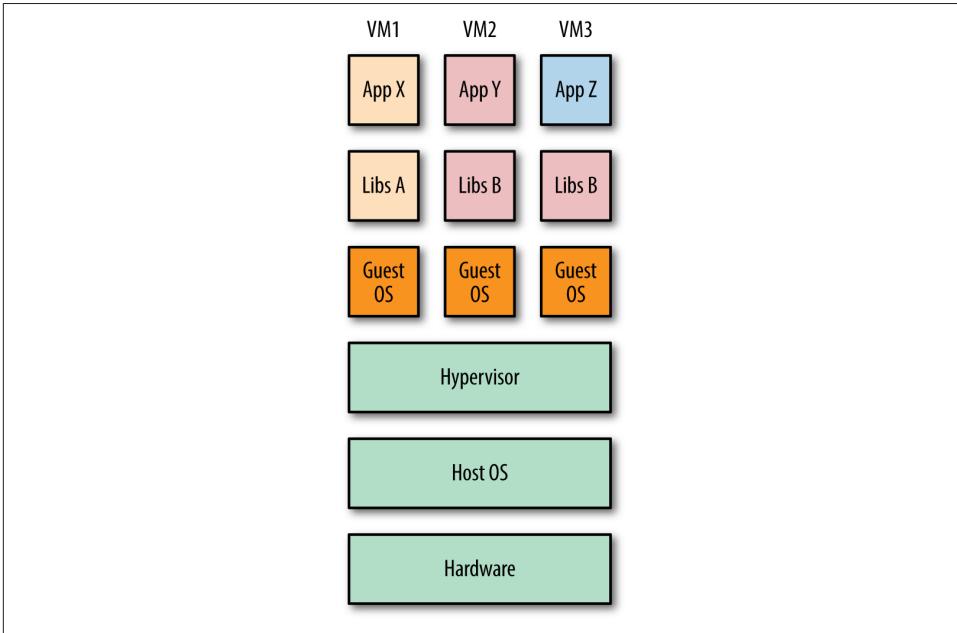


Figure 1-1. Three VMs running on a single host

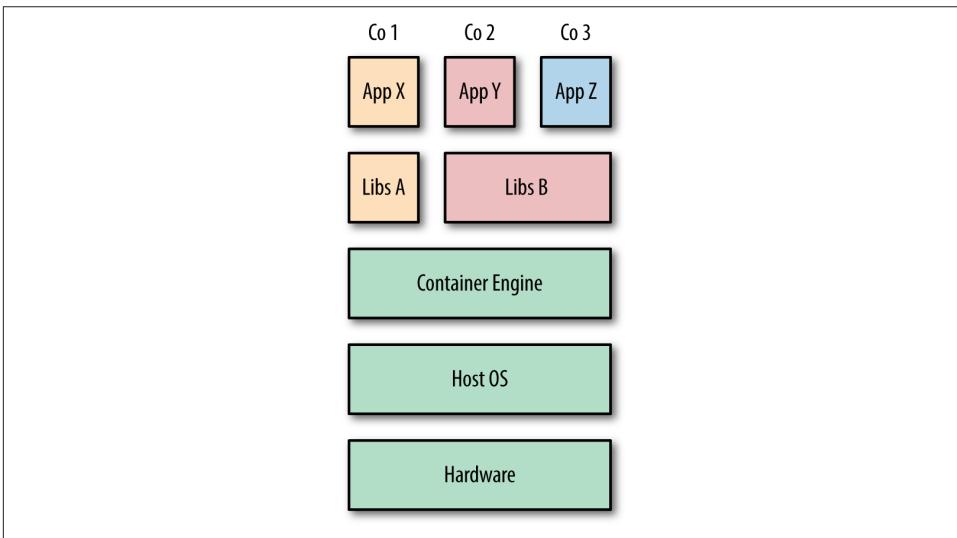


Figure 1-2. Three containers running on a single host

# Docker and Containers

Containers are an old concept. For decades, UNIX systems have had the `chroot` command that provides a simple form of filesystem isolation. Since 1998, FreeBSD has had the `jail` utility, which extended `chroot` sandboxing to processes. Solaris Zones offered a comparatively complete containerization technology around 2001 but was limited to the Solaris OS. Also in 2001, Parallels Inc, (then SWsoft) released the commercial Virtuozzo container technology for Linux and later open sourced the core technology as OpenVZ in 2005.<sup>3</sup> Then Google started the development of CGroups for the Linux kernel and began moving its infrastructure to containers. The Linux Containers (LXC) project started in 2008 and brought together CGroups, kernel namespaces, and `chroot` technology (among others) to provide a complete containerization solution. Finally, in 2013, Docker brought the final pieces to the containerization puzzle, and the technology began to enter the mainstream.

Docker took the existing Linux container technology and wrapped and extended it in various ways—primarily through portable images and a user-friendly interface—to create a complete solution for the creation and distribution of containers. The Docker platform has two distinct components: the Docker Engine, which is responsible for creating and running containers; and the Docker Hub, a cloud service for distributing containers.

The Docker Engine provides a fast and convenient interface for running containers. Before this, running a container using a technology such as LXC required significant specialist knowledge and manual work. The Docker Hub provides an enormous number of public container images for download, allowing users to quickly get started and avoid duplicating work already done by others. Further tooling developed by Docker includes *Swarm*, a clustering manager; *Kitematic*, a GUI for working with containers; and *Machine*, a command-line utility for provisioning Docker hosts.

By open sourcing the Docker Engine, Docker was able to grow a large community around Docker and take advantage of public help with bug fixes and enhancements. The rapid rise of Docker meant that it effectively became a *de facto* standard, which led to industry pressure to move to develop independent formal standards for the container runtime and format. In 2015, this culminated in the establishment of the Open Container Initiative, a “governance structure” sponsored by Docker, Microsoft, CoreOS, and many other important organizations, whose mission is to develop such a standard. Docker’s container format and runtime forms the basis of the effort.

The uptake of containers has largely been driven by developers, who for the first time were given the tools to use containers effectively. The fast start-up time of Docker

---

<sup>3</sup> OpenVZ never achieved mass adoption, possibly because of the requirement to run a patched kernel.

containers is essential to developers who crave quick and iterative development cycles where they can promptly see the results of code changes. The portability and isolation guarantees of containers ease collaboration with other developers and operations; developers can be sure their code will work across environments, and operations can focus on hosting and orchestrating containers rather than worrying about the code running inside them.

The changes brought about by Docker are significantly changing the way we develop software. Without Docker, containers would have remained in the shadows of IT for a long time to come.

## The Shipping Metaphor

The Docker philosophy is often explained in terms of a shipping-container metaphor, which presumably explains the Docker name. The story normally goes something like this:

When goods are transported, they have to pass through a variety of different *means*, possibly including trucks, forklifts, cranes, trains, and ships. These means have to be able to handle a wide variety of goods of different sizes and with different requirements (e.g., sacks of coffee, drums of hazardous chemicals, boxes of electronic goods, fleets of luxury cars, and racks of refrigerated lamb). Historically, this was a cumbersome and costly process, requiring manual labor, such as dock workers, to load and unload items by hand at each transit point (Figure 1-3).

The transport industry was revolutionized by the introduction of the intermodal container. These containers come in standard sizes and are designed to be moved between modes of transport with a minimum of manual labor. All transport machinery is designed to handle these containers, from the forklifts and cranes to the trucks, trains, and ships. Refrigerated and insulated containers are available for transporting temperature sensitive goods, such as food and pharmaceuticals. The benefits of standardization also extend to other supporting systems, such as the labeling and sealing of containers. This means the transport industry can let the producers of goods worry about the contents of the containers so that it can focus on the movement and storage of the containers themselves.

The goal of Docker is to bring the benefits of container standardization to IT. In recent years, software systems have exploded in terms of diversity. Gone are the days of a LAMP<sup>4</sup> stack running on a single machine. A typical modern system may include Javascript frameworks, NoSQL databases, message queues, REST APIs, and backends all written in a variety of programming languages. This stack has to run partly or completely on top of a variety of hardware—from the developer’s laptop and the in-house testing cluster to the production cloud provider. Each of these environments is

---

<sup>4</sup> This originally stood for Linux, Apache, MySQL, and PHP—common components in a web application.

different, running different operating systems with different versions of libraries on different hardware. In short, we have a similar issue to the one seen by the transport industry—we have to continually invest substantial manual effort to move code between environments. Much as the intermodal containers simplified the transportation of goods, Docker containers simplify the transportation of software applications. Developers can concentrate on building the application and shipping it through testing and production without worrying about differences in environment and dependencies. Operations can focus on the core issues of running containers, such as allocating resources, starting and stopping containers, and migrating them between servers.



*Figure 1-3. Dockers working in Bristol, England, in 1940 (by Ministry of Information Photo Division Photographer)*

## Docker: A History

In 2008, Solomon Hykes founded dotCloud to build a language-agnostic Platform-as-a-Service (PaaS) offering. The language-agnostic aspect was the unique selling point for dotCloud—existing PaaSs were tied to particular sets of languages (e.g.,

Heroku supported Ruby, and Google App Engine supported Java and Python). In 2010, dotCloud took part in Y Combinator accelerator program, where it was exposed to new partners and began to attract serious investment. The major turning point came in March 2013, when dotCloud open sourced Docker, the core building block of dotCloud. While some companies may have been scared that they were giving away their magic beans, dotCloud recognized that Docker would benefit enormously from becoming a community-driven project.

Early versions of Docker were little more than a wrapper around LXC paired with a union filesystem, but the uptake and speed of development was shockingly fast. Within six months, it had more than 6,700 stars on GitHub and 175 nonemployee contributors. This led dotCloud to change its name to Docker, Inc. and to refocus its business model. Docker 1.0 was announced in June 2014, just 15 months after the 0.1 release. Docker 1.0 represented a major jump in stability and reliability—it was now declared “production ready,” although it had already seen production use in several companies, including Spotify and Baidu. At the same time, Docker started moving toward being a complete platform rather than just a container engine, with the launch of the Docker Hub, a public repository for containers.

Other companies were quick to see the potential of Docker. Red Hat became a major partner in September 2013 and started using Docker to power its OpenShift cloud offering. Google, Amazon, and DigitalOcean were quick to offer Docker support on their clouds, and several startups began specializing in Docker hosting, such as StackDock. In October 2014, Microsoft announced that future versions of Windows Server would support Docker, representing a huge shift in positioning for a company traditionally associated with bloated enterprise software.

DockerConEU in December 2014 saw the announcement of Docker Swarm, a clustering manager for Docker and Docker Machine, a CLI tool for provisioning Docker hosts. This was a clear signal of Docker’s intention to provide a complete and integrated solution for running containers and not allowing themselves to be restricted to only providing the Docker engine.

Also that December, CoreOS announced the development of rkt, its own container runtime, and the development of the appc container specification. In June 2015, during DockerCon in San Francisco, Solomon Hykes from Docker and Alex Polvi from CoreOS announced the formation of the Open Container Initiative (then called the Open Container Project) to develop a common standard for container formats and runtimes.

Also in June 2015, the FreeBSD project announced that Docker was now supported on FreeBSD, using ZFS and the Linux compatibility layer. In August 2015, Docker and Microsoft released a “tech preview” of the Docker Engine for Windows server.

With the release of Docker 1.8, Docker introduced the content trust feature, which verifies the integrity and publisher of Docker images. Content trust is a critical component for building trusted workflows based on images retrieved from Docker registries.

## Plugins and Plumbing

As a company, Docker Inc. has always been quick to recognize it owes a lot of its success to the ecosystem. While Docker Inc. was concentrating on producing a stable, production-ready version of the container engine, other companies such as CoreOS, WeaveWorks, and ClusterHQ were working on related areas, such as orchestrating and networking containers. However, it quickly became clear that Docker Inc., was planning to provide a complete platform out of the box, including networking, storage, and orchestration capabilities. In order to encourage continued ecosystem growth and ensure users had access to solutions for a wide range of use cases, Docker Inc. announced it would create a modular, extensible framework for Docker where stock components could be swapped out for third-party equivalents or extended with third-party functionality. Docker Inc. called this philosophy “Batteries Included, But Replaceable,” meaning that a complete solution would be provided, but parts could be swapped out.<sup>5</sup>

At the time of writing, the plugin infrastructure is in its infancy, but is available. There are several plugins already available for networking containers and data management.

Docker also follows what it calls the “Infrastructure Plumbing Manifesto,” which underlines its commitment to reusing and improving existing infrastructure components where possible and contributing reusable components back to the community when new tools are required. This led to the spinning out of the low-level code for running containers into the *runC* project, which is overseen by the OCI and can be reused as the basis for other container platforms.

## 64-Bit Linux

At the time of writing, the only stable, production-ready platform for Docker is 64-bit Linux. This means your computer will need to run a 64-bit Linux distribution, and all your containers will also be 64-bit Linux. If you are a Windows or Mac OS user, you can run Docker inside a VM.

---

<sup>5</sup> Personally, I’ve never liked the phrase; all batteries provide much the same functionality and can only be swapped with batteries of the same size and voltage. I assume the phrase has its origins in Python’s “Batteries Included” philosophy, which it uses to describe the extensive standard library that ships with Python.

Support for other native containers on other platforms, including BSD, Solaris, and Windows Server, is in various stages of development. Since Docker does not natively do any virtualization, containers must always match the host kernel—a Windows Server container can only run on a Windows Server host, and a 64-bit Linux container will only run on a 64-bit Linux host.

## Microservices and Monoliths

One of the biggest use cases and strongest drivers behind the uptake of containers are *microservices*.

Microservices are a way of developing and composing software systems such that they are built out of small, independent components that interact with one another over the network. This is in contrast to the traditional *monolithic* way of developing software, where there is a single large program, typically written in C++ or Java.

When it comes to scaling a monolith, commonly the only choice is to *scale up*, where extra demand is handled by using a larger machine with more RAM and CPU power. Conversely, microservices are designed to *scale out*, where extra demand is handled by provisioning multiple machines the load can be spread over. In a microservice architecture, it's possible to only scale the resources required for a particular service, focusing on the bottlenecks in the system. In a monolith, it's scale everything or nothing, resulting in wasted resources.

In terms of complexity, microservices are a double-edged sword. Each individual microservice should be easy to understand and modify. However, in a system composed of dozens or hundreds of such services, the overall complexity increases due to the interaction between individual components.

The lightweight nature and speed of containers mean they are particularly well suited for running a microservice architecture. Compared to VMs, containers are vastly smaller and quicker to deploy, allowing microservice architectures to use the minimum of resources and react quickly to changes in demand.

For more information on microservices, see *Building Microservices* by Sam Newman (O'Reilly) and Martin Fowler's *Microservice Resource Guide*.

# Want to read more?

You can [buy this book](#) at oreilly.com in print and ebook format.

**Buy 2 books, get the 3rd FREE!**

Use discount code OPC10

All orders over \$29.95 qualify for **free shipping** within the US.

It's also available at your favorite book retailer, including the iBookstore, the [Android Marketplace](#), and [Amazon.com](#).



**O'REILLY**<sup>®</sup>

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 15055