Docker
IN ACTION

Jeff Nickoloff

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Docker in Action
Version 10

welcome

Thank you for purchasing the MEAP to Docker in Action. We're excited to see the book reach this stage and we're looking forward to its continued development and eventual release. Docker is software that is available for Windows, OSX, and Linux based operating systems and works with those operating systems to package, ship and run applications in lightweight containers. It simplifies software logistics for users, developers, and administrators. On a computer, Docker tidily keeps software organized, conflict free, and simple to add, remove, and update.

This book is both for software consumers who are curious about how to use Docker to find, install, and manage software, and for developers and system administrators who want to explore Docker as an option for deployment and delivery of software to customers and managed users.

Our first MEAP release includes three chapters from part 1. Part 1 is about Docker from a software consumer's point of view. In chapter 1, I'll tell you how Docker works at a high level and begin to explore some of its powerful benefits. In chapter 2, you'll learn how it helps you manage resources on your computer. In chapter 3, cover how to use Docker to find, install, and manage software through simple, realistic walkthroughs of example use cases you will likely encounter. The rest of Part 1 will cover more advanced ways to interact with software running inside of containers and container security.

Part 2 is for software developers, distributors or anyone curious about how software is packaged for Docker. It covers using Docker to package software and explores different ways of distributing your Docker-ready software privately and publically.

Part 3 covers advanced use-cases and bigger ideas like continuous integration, integration testing and orchestrated deployments. It is written for system administrators or those wondering how to use Docker within their own network for simplifying software administration and deployment tasks.

Please be sure to post any questions, comments, or suggestions you have about the book in the Author Online forum. Your feedback is essential in developing the best book possible.

—Jeff Nickoloff

brief contents

Part 1: Keeping a tidy and trustworthy computer

1 Welcome to Docker

2 Running software in containers

3 Software installation simplified

4 Persistent storage and shared state with volumes

5 Network exposure

6 Limiting risk with isolation

Part 2: Packaging software for distribution

7 Packaging software in images

8 Build automation and advanced image considerations

9 Public and private software distribution

10 Running Customized Registries

Part 3: Rapid and flexible deployment

11 Declarative environments with Docker Compose

12 Clusters with Machine and Swarm

1 Welcome to Docker

This chapter covers

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

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

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

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

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

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

Launched in March 2013, Docker works with your operating system to package, ship, and run software. You can think of Docker as a software logistics provider that will save you time and let you focus on high-value activities. You can use Docker with network applications like web servers, databases, and mail servers and with

terminal applications like text editors, compilers, network analysis tools, and scripts; in some cases it's even used to run GUI applications like web browsers and productivity software.

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

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

1.1 What is Docker?

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

1.1.1 Containers

Historically, Unix-style operating systems have used the term jail to describe a modified runtime environment for a program that prevents that program from accessing protected resources. Since 2005, after the release of Sun's Solaris 10 and Solaris Containers, container has become the preferred term for such a runtime environment. The goal has expanded from preventing access to protected resources to isolating a process from all resources except where explicitly allowed.

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

With Docker, users get containers at a much lower cost. As Docker and its container engines improve, you get the latest and greatest jail features. Instead of keeping up with the rapidly evolving and highly technical world of building strong application jails, you can let Docker handle the bulk of that for you. This will save you a lot of time and money and bring peace of mind.

1.1.2 Containers are not virtualization

Without Docker, businesses typically use hardware virtualization (also known as virtual machines) to provide isolation. Virtual machines provide virtual hardware on which an operating system and other programs can be installed. They take a long time (often minutes) to create and require significant resource overhead because they run a whole copy of an operating system in addition to the software you want to use.

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

1.1.3 Running software in containers for isolation

As noted earlier, containers have existed for decades. Docker uses Linux namespaces and cgroups, which have been part of Linux since 2007. Docker doesn't provide the container technology. It specifically makes it simpler to use. To understand what containers look like on a system, let's first establish a baseline. Figure 1.1 shows a basic example on simplified computer system architecture.

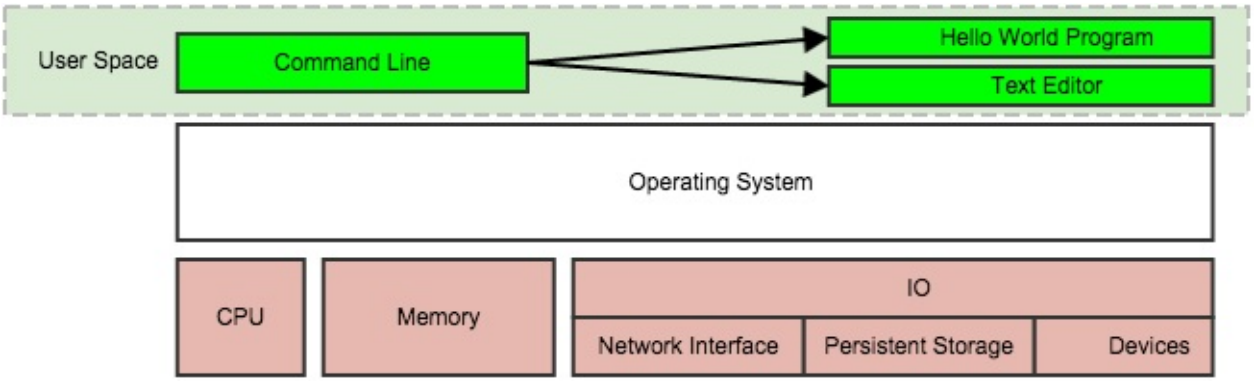

Figure 1.1 A basic computer stack running two programs that were started from the command line

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

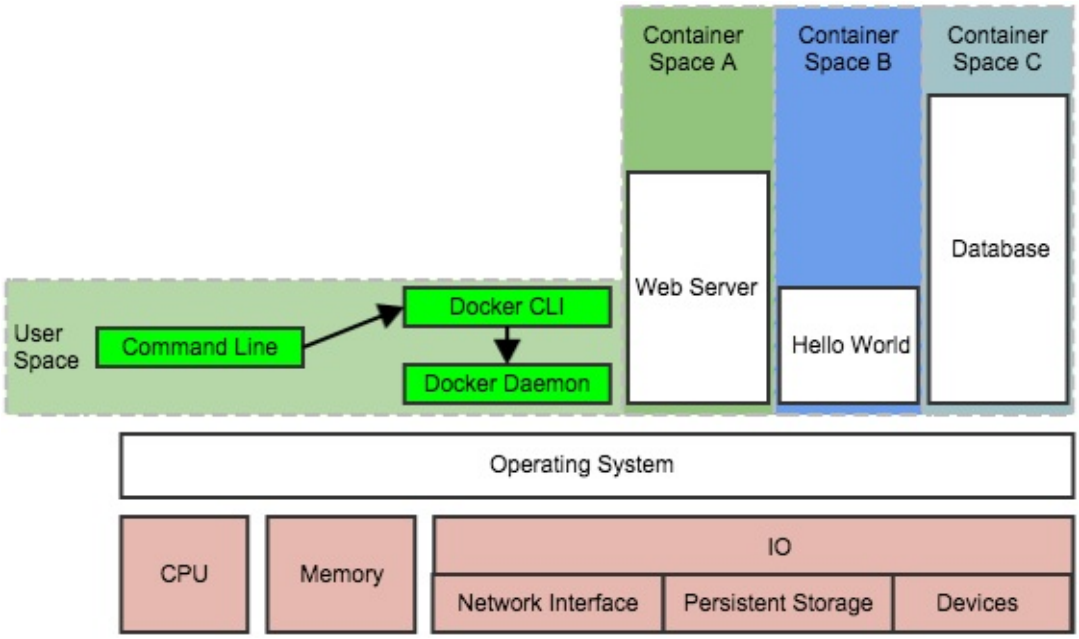

Figure 1.2 Docker running three containers on a basic Linux computer system

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

Figure 1.2 also shows three running containers. Each is running as a child process of the Docker daemon, wrapped with a container, and the delegate process is running in its own memory subspace of the user space. Programs running inside a container can access only their own memory and resources as scoped by the container.

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

container features. The specific aspects are as follows:

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

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

1.1.4 Shipping containers

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

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

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

1.2 What problems does Docker solve?

Using software is complex. Before installation you have to consider what operating

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

Most computers have more than one application installed and running. And most applications have dependencies on other software. What happens when two or more applications you want to use don't play well together? Disaster. Things are only made more complicated when two or more applications share dependencies:

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

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

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

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

1.2.1 Getting organized

Without Docker, a computer can end up looking like a junk drawer. Applications have all sorts of dependencies. Some applications depend on specific system

libraries for common things like sound, networking, graphics, and so on. Others depend on standard libraries for the language they're written in. Some depend on other applications, such as how a Java program depends on the Java Virtual Machine or a web application might depend on a database. It's common that a running program requires exclusive access to some scarce resource like a network connection or a file.

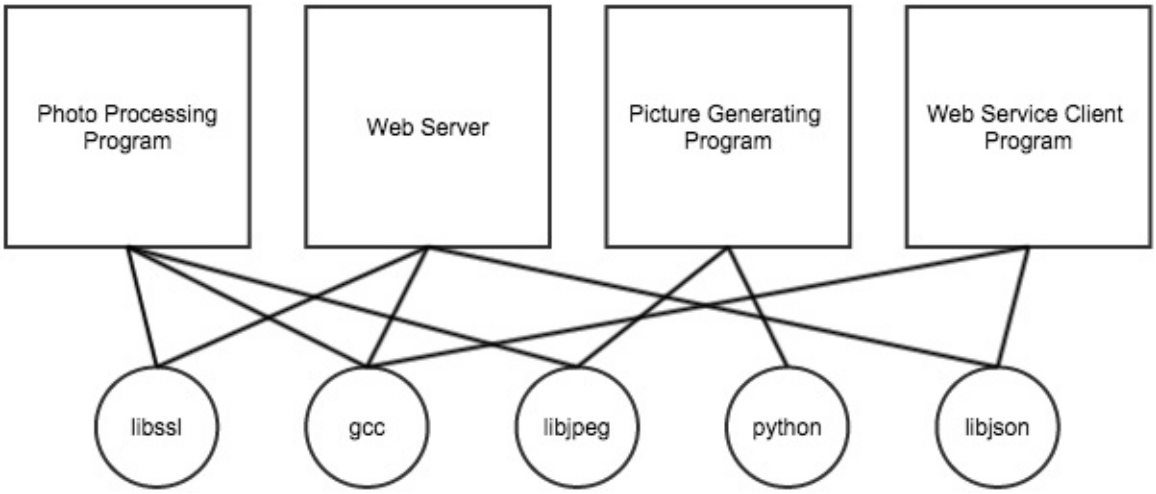

Figure 1.3 Dependency relationships of example programs

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

Docker keeps things organized by isolating everything with containers and image Figure 1.4 illustrates these same applications and their dependencies running inside containers. With the links broken and each application neatly contained, understanding the system is an approachable task.

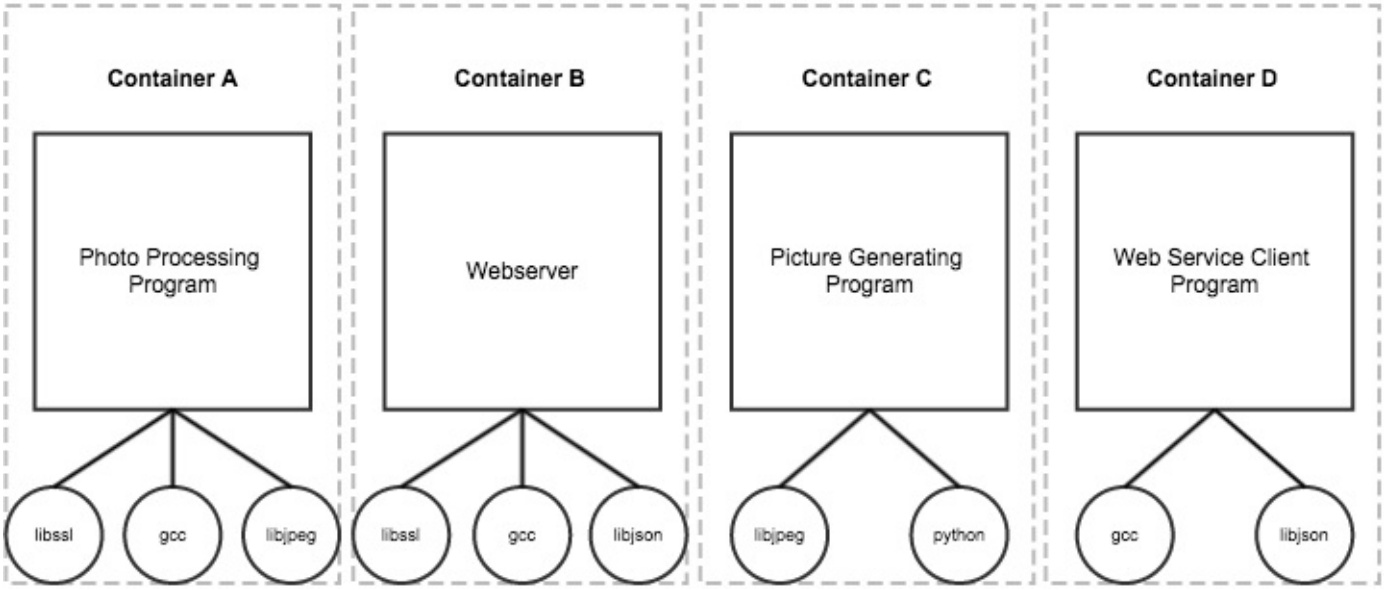

Figure 1.4 Example programs running inside containers with copies of their dependencies

1.2.2 Improving portability

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

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

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

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

1.2.3 Protecting your computer

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

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

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

Like physical jail cells, anything inside a container can only access things that are inside it as well. There are exceptions to this rule but only when explicitly created by the user. Containers limit the scope of impact that a program can have on othe running programs, the data it can access, and system resources. Figure 1.5 illustrates the difference between running software outside and inside a container

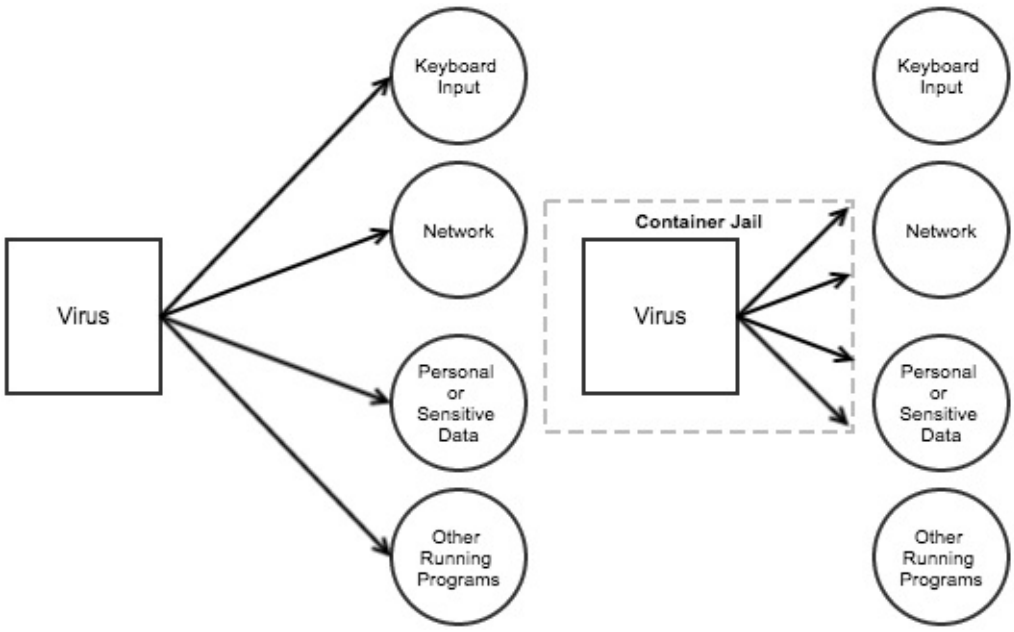

Figure 1.5 Left: a malicious program with direct access to sensitive resources. Right: a malicious program inside a container.

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

1.3 Why is Docker important?

Docker provides what is called an abstraction. Abstractions allow you to work with complicated things in simplified terms. So, in the case of Docker, instead of focusing on all of the complexities and specifics associated with installing an application, all we need consider is what software we'd like to install. Like a crane

loading a shipping container onto a ship, the process of installing any software with Docker is identical to any other. The shape or size of the thing inside the shipping container may vary, but the way that the crane picks up the container w always be the same. All of the tooling is reusable for any shipping container.

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

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

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

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

Fourth, we’re finally starting to see better adoption of some of the more advanced isolation features of operating systems. This might seem minor, but quite a few people are trying to make computers more secure through isolation at the operating system level. It’s been a shame that their hard work has taken so long t see mass adoption. Containers have existed for decades in one form or another. It great that Docker helps us take advantage of those features without all the complexity.

1.4 Where and when to use Docker

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

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

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

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

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

1.5 Example: "Hello World"

I like to get people started with an example. In keeping with tradition we'll use "Hello World." Before you begin download and install Docker for your system. Detailed instructions are kept up to date for every available system on docker.con at https://docs.docker.com/installation/. OSX and Windows users will install the fu

Docker suite of applications using the Docker Toolbox. Once you have Docker installed and an active internet connection, head to your command prompt and type the following:

```
docker run dockerinaction/hello_world
```

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

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

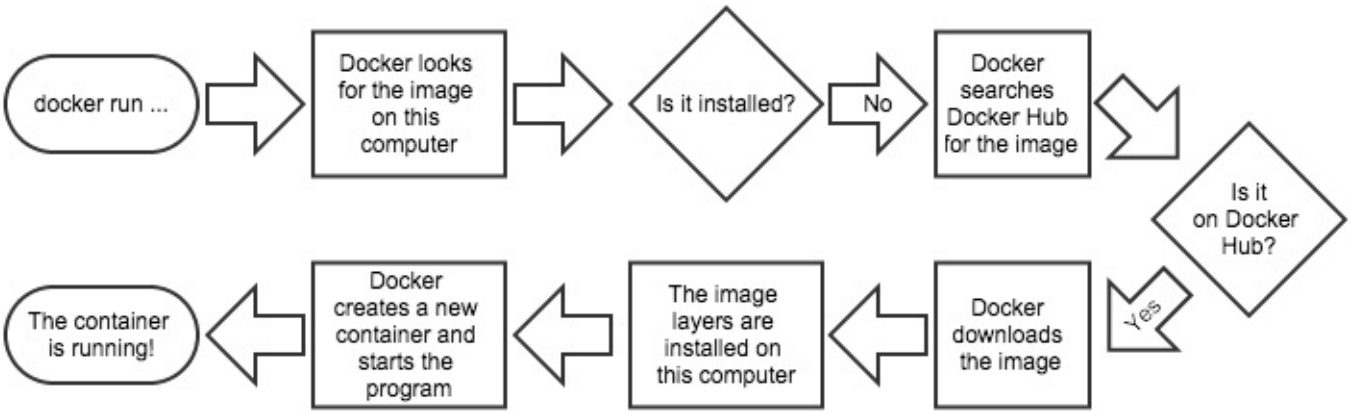

Figure 1.6 What happens after running `docker run ...`

First, you use the `docker run` command to start a new container. This single command triggers a sequence (shown in figure 1.6) that installs, runs, and stops a program inside a container.

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

> **NOTE** This repository and several others were created specifically to support

the examples in this book. By the end of part 2 you should feel comfortable examining these open source examples. Any suggestions you have on how they might be improved are always welcome.

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

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

```
echo "hello world"
```

After the command prints "hello world" to the terminal, it exits and the container automatically stopped. Understand that the running state of a container is directly tied to the state of a single running program inside the container. If a program is running, the container is running. If the program is stopped, the container is stopped. Restarting a container runs the program again.

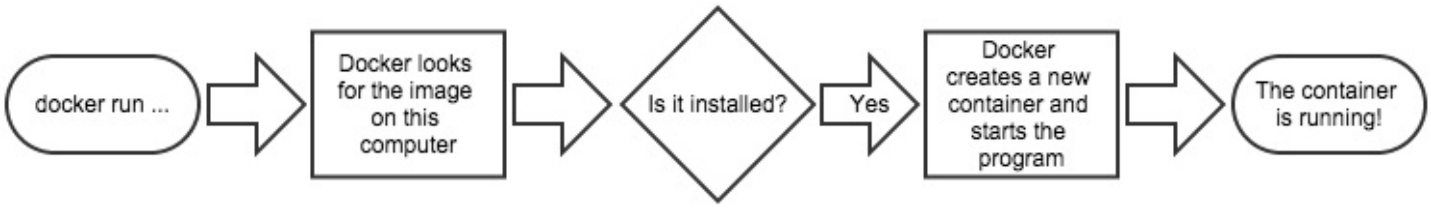

Figure 1.7 Running docker run a second time. Because the image is already installed Docker can start the new container right away.

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

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

1.6 Summary

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

- Docker takes a logistical approach to solving common software problems and simplifies your experience with installing, running, publishing, and removing software. It’s a command-line program, a background daemon, and a set of remote services. It’s integrated with community tools provided by Docker Inc
- The container abstraction is at the core of its logistical approach.
- Working with containers instead of software creates a consistent interface and enables the development of more sophisticated tools.
- Containers help keep your computers tidy because software inside containers can’t interact with anything outside its container, and no shared dependencies can be formed.
- Because Docker is available and supported on Linux, OSX, and Windows, most software packaged in Docker images can be used on any computer.
- Docker doesn’t provide container technology; it hides the complexity of working directly with the container software.

2 Running software in containers

This chapter covers

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

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

2.1 Getting help with the Docker command line

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

```
docker help
```

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

This gives you only high-level information about what commands are available. To get detailed information about a specific command, include the command in the `<COMMAND>` argument. For example, you might use the following to find out how

to copy files from a location inside a container to a location on the host machine. Run the following as an example:

```
docker help cp
```

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

2.2 Controlling containers: building a website monitor

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

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

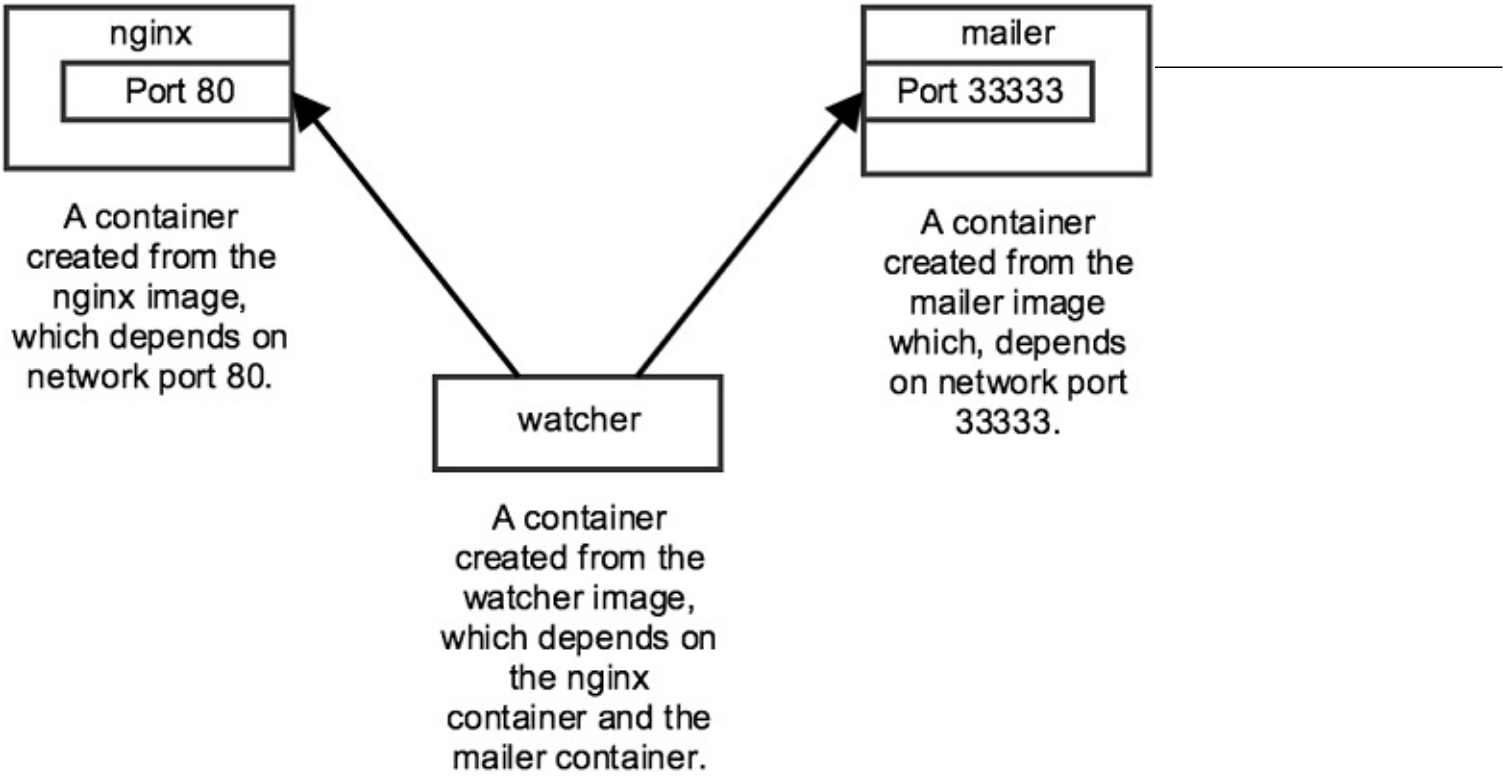

Figure 2.1 The three containers that you'll build in this example

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

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

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

2.2.1 Creating and starting a new container

When installing software with Docker, we say that we're installing an image. There are different ways to install an image and several sources for images. Images are covered in depth in chapter 3. In this example we're going to download and instal