Containerize your Apps with Docker and Kubernetes

Deploy, scale, orchestrate, and manage containers with Docker and Kubernetes

PREVIEW

Packt

www.packt.com

Dr. Gabriel N. Schenker

Containerize your Apps with Docker and Kubernetes

Deploy, scale, orchestrate, and manage containers with Docker and Kubernetes

Dr. Gabriel N. Schenker

Packt

BIRMINGHAM - MUMBAI

Containerize your Apps with Docker and Kubernetes

Commissioning Editor: Vijin Boricha
Acquisition Editor: Shrilekha Inani
Content Development Editors: Ronn Kurien
Technical Editor: Swathy Mohan
Copy Editor: Safis Editing
Project Coordinator: Jagdish Prabhu
Proofreader: Safis Editing
Indexers: Mariammal Chettiyar
Graphics: Tom Scaria
Production Coordinator: Nilesh Mohite

First published: September 2018

Production reference: 1260918

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78961-036-9

`www.packtpub.com`

Contributors

About the author

Dr. Gabriel N. Schenker has more than 25 years of experience as an independent consultant, architect, leader, trainer, mentor, and developer. Currently, Gabriel works as Senior Curriculum Developer at Confluent after coming from a similar position at Docker. Gabriel has a Ph.D. in Physics, and he is a Docker Captain, a Certified Docker Associate, and an ASP Insider. When not working, Gabriel enjoys time with his wonderful wife Veronicah and his children.

About the reviewer

Xijing Zhang is currently a technical curriculum developer at Docker after graduating from the University of Southern California as an electrical engineer. Previously, she has interned on the Failure Analysis Team at SanDisk and has held multiple research positions at USC and Tsinghua University. She has worked on projects dealing with making air conditioners more efficient, nuclear power safety, and single photon emission.

Peter McKee is a Software Architect and Senior Software Engineer at Docker, Inc. He leads the technical team that delivers the Docker Success Center. He's been leading and mentoring teams for more than 20 years. When not building things with software, he spends his time with his wife and seven kids in beautiful Austin, TX.

Table of Contents

Preface

Containerization is said to be the best way to implement DevOps and the main goal of this book is to provide end-to-end deployment solutions for your Azure environment.

This book will initiate with the implementation of deploying and managing containers along with getting you up and running with Docker and Kubernetes. Then, this book will explain operations for container management and orchestration in Docker using Azure's cloud solutions. You will also learn to deploy and manage highly scalable applications along with setting-up production ready Kubernetes cluster on Azure in an intact environment. Lastly, The book will also help you leverage Microsoft's Docker and Kubernetes tools to build apps that can be quickly deployed on Azure.

By the end of the book, you will get hands-on with some more advanced topics to further extend your knowledge about Docker and Kubernetes.

Who this book is for

If you are a developer, system administrator, or DevOps engineer who wants to use Docker and Kubernetes to run your mission-critical applications scalable, securely, and highly available on-prem or in the cloud, then this book is for you. In order to learn from this book, you should have some basic Linux/Unix skills such as installing packages, editing files, managing services, and so on. If you have some basic virtualization experience that would be an added advantage.

What this book covers

Chapter 1, *What Are Containers and Why Should I Use Them?*, this chapter focuses on the software supply chain and the friction within it. It then presents containers as a means to reduce this friction and add enterprise-grade security on top of it. In this chapter, we also look into how containers and the ecosystem around them are assembled. We specifically point out the distinction between the upstream OSS components (Moby) that form the building blocks of the downstream products of Docker and other vendors.

Chapter 2, *Setting up a Working Environment*, in this chapter discussed in detail how to set up an ideal environment for developers, DevOps and operators that can be used when working with Docker containers.

Chapter 3, *Working with Containers*, this chapter teaches how start, stop and remove containers. The chapter also teaches how to inspect containers to retrieve additional metadata of it. Furthermore, it introduces how to run additional processes or how to attach to the main process in an already running container. It is also showing how to retrieve logging information from a container that is produced by the processes running inside it. Finally, the chapter introduces the inner workings of a container including such things as Linux namespaces and cgroups.

Chapter 4, *Creating and Managing Container Images*, this chapter introduces the different ways how to create container images that serve as templates for containers. It introduces the inner structure of an image and how it is built.

Chapter 5, *Data Volumes and System Management*, this chapter introduces data volumes that can be used by stateful components running in containers. The chapter also introduces system level commands that are used to gather information about Docker and the underlying OS as well as commands to clean the system from orphaned resources. Finally, it introduces the system events generated by the Docker engine.

Chapter 6, *Distributed Application Architecture*, this chapter introduces the concept of a distributed application architecture and discusses the various patterns and best practices that are required to run a distributed application successfully. Finally, it discusses the additional requirements that need to be fulfilled to run such an application in production.

Chapter 7, *Single-Host Networking*, this chapter introduces the Docker container networking model and its single host implementation in the form of the bridge network. The chapter introduces the concept of software-defined networks and how they are used to secure containerized applications. Finally, it introduces how container ports can be opened to the public and thus make containerized components accessible from the outside world.

Chapter 8, *Docker Compose*, this chapter introduces the concept of an application consisting of multiple services each running in a container and how Docker Compose allows us to easily build, run and scale such an application using a declarative approach.

Chapter 9, *Orchestrators*, this chapter introduces the concept of orchestrators. It teaches why orchestrators are needed and how they conceptually work. The chapter will also provide an overview of the most popular orchestrators and name a few of their pros and cons.

Chapter 10, *Orchestrating Containerized Applications with Kubernetes*, this chapter introduces Kubernetes. Kubernetes is currently the clear leader in the container orchestration space. It starts with a high-level overview of the architecture of a Kubernetes cluster and then discusses the main objects used in Kubernetes to define and run containerized applications.

Chapter 11, *Deploying, Updating, and Securing an Application with Kubernetes*, this chapter teaches how to deploy, update and scale applications into a Kubernetes cluster. It also explains how zero downtime deployments are achieved to enable disruption free updates and rollbacks of mission-critical applications. This chapter also introduces Kubernetes secrets as a means to configure services with and protect sensitive data.

Chapter 12, *Running a Containerized App in the Cloud*, this chapter shows how to deploy a complex containerized application into a hosted Kubernetes cluster on Microsoft Azure using the Azure Kubernetes Service (AKS) offering. First it explains how to provision a Kubernetes cluster, second it shows how to host the Docker images in the Azure Container Registry and finally it demonstrates how to deploy, run, monitor, scale and upgrade the application. The chapter also demonstrates how to upgrade the version of Kubernetes in the cluster without causing any downtime.

To get the most out of this book

Ideally you have access to a laptop or personal computer with Windows 10 Professional or a recent version of Mac OS X installed. A computer with any popular Linux OS installed works too. If you're on a Mac you should install Docker for Mac and if you're on Windows then install Docker for Windows. You can download them from here: `https://www.docker.com/community-edition`.

If you are on an older version of Windows or are using Windows 10 Home edition, then you should install Docker Toolbox. You can find the Docker Toolbox here: `https://docs.docker.com/toolbox/toolbox_install_windows/`.

On the Mac, use the Terminal application, and on Windows, use a PowerShell console to try out the commands you will be learning. You also need a recent version of a browser such as Google Chrome, Safari or Internet Explorer. Of course you will need internet access to download tools and container images that we are going to use and explore in this book.

To follow *Chapter 12*, Running a Containerized App in the Cloud, you need access to Microsoft Azure. If you do not have an existing account on Azure it is possible to request a trial account here at `https://azure.microsoft.com/en-us/free/`.

Download the EPUB/mobi and example code files

An EPUB and mobi version of this book is available free of charge on Github. You can download them and the code bundle at `https://github.com/PacktPublishing/Containerize-your-Apps-with-Docker-and-Kubernetes`.

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `http://www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is hosted on GitHub at `https://github.com/appswithdockerandkubernetes/labs`.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it from `https://www.packtpub.com/sites/default/files/downloads/9781789610369_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example; "The content of each layer is mapped to a special folder on the host system, which is usually a subfolder of `/var/lib/docker/`."

A block of code is set as follows:

```
COPY . /app
COPY ./web /app/web
COPY sample.txt /data/my-sample.txt
ADD sample.tar /app/bin/
ADD http://example.com/sample.txt /data/
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
FROM python:2.7
RUN mkdir -p /app
WORKDIR /app
COPY ./requirements.txt /app/
RUN pip install -r requirements.txt
CMD ["python", "main.py"]
```

Any command-line input or output is written as follows:

```
az group create --name pets-group --location westeurope
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `customercare@packtpub.com`, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
What Are Containers and Why Should I Use Them?

This first chapter of this book will introduce you to the world of containers and their orchestration. This book assumes you have no prior knowledge in the area of containers, and will give you a very practical introduction into the topic.

In this chapter, we are focusing on the software supply chain and the friction within it. We then present containers as a means to reduce this friction and add enterprise-grade security on top of it. In this chapter, we also look into how containers and the ecosystem around them are assembled. We specifically point out the distinction between the upstream **Operations Support System** (**OSS**) components, united under the code name Moby, that form the building blocks of the downstream products of Docker and other vendors.

The chapter covers the following topics:

- What are containers?
- Why are containers important?
- What's the benefit for me or for my company?
- The Moby project
- Docker products
- The container ecosystem
- Container architecture

After completing this module, you will be able to:

- Explain in a few simple sentences to an interested layman what containers are, using an analogy such as physical containers
- Justify to an interested layman why containers are so important, using an analogy such as physical containers versus traditional shipping, or apartment homes versus single family homes, and so on
- Name at least four upstream open source components that are used by the Docker products, such as Docker for Mac/Windows
- Identify at least three Docker products

Technical requirements

This chapter is a theoretical introduction into the topic. Therefore, there are no special technical requirements for this chapter.

What are containers?

A software container is a pretty abstract thing and thus it might help if we start with an analogy that should be pretty familiar to most of the readers. The analogy is a shipping container in the transportation industry.

We transport huge amounts of goods on trains, ships, and trucks. We unload them at target locations, which may be another means of transportation. Goods are often diverse and complex to handle. Before the invention of shipping containers, this unloading from one means of transportation and loading onto another was a really complex and tedious process. Take, for example, a farmer bringing a cart full of apples to a central train station where the apples are then loaded onto a train, together with all the apples from many other farmers. Or think of a winemaker bringing his barrels of wine with a truck to the port where they are unloaded, and then transferred to a ship that will transport the barrels overseas. Every type of good was packaged in its own way and thus had to be handled in its own way. Any loose goods risked being stolen or damaged in the process. Then, there came the container, and it totally revolutionized the transportation industry.

The container is just a metallic box with standardized dimensions. The length, width, and height of each container is the same. This is a very important point. Without the world agreeing on a standard size, shipping containers would not have become so successful. Nowadays, companies who want to have their goods transported from A to B package those goods into these standardized containers. Then, they call a shipper which comes with a standardized means for transportation. This can be a truck designed to load a container or a train whose wagons can each transport one or several containers. Finally, we have ships that are specialized in transporting huge amounts of containers. The shippers never need to unpack and repackage goods. For a shipper, a container is a just a black box; they are not interested in what is in it nor should they care in most cases. It is just a big iron box with standard dimensions. The packaging of goods into containers is now fully delegated to the parties that want to have their goods shipped, and they should know best how to handle and package those goods. Since all containers have the same standardized shape and dimensions, the shippers can use standardized tools to handle containers, that is, cranes that unload containers, say from a train or a truck, and load them onto a ship or vice versa. One type of crane is enough to handle all the containers that come along over time. Also, the means of transportation can be standardized, such as container ships, trucks, and trains. Because of all this standardization, all the processes in and around shipping goods could be standardized, and thus made much more efficient than they were before the age of containers.

I think by now you should have a good understanding of why shipping containers are so important and why they revolutionized the whole transportation industry. I chose this analogy because the software containers that we are going to look at here fulfill the exact same role in the software supply chain as shipping containers do in the supply chain of physical goods.

Let's discuss what developers used to do when they developed a new application. Once an application was completed in the eyes of the developers, they would hand this application over to the operations engineers that were then supposed to install it on the production servers and get it running. If the operations engineers were lucky, they even got an accurate document with installation instructions from the developers. So far so good, and life was easy. But things got a bit out of hand when there were many teams of developers in an enterprise that created quite different types of applications, yet all needed to be installed on the same production servers and kept running there. Usually, each application has some external dependencies such as which framework it was built on or what libraries it uses and so on.

Sometimes, two applications would use the same framework but in different versions that might or might not be compatible between each other. Our operations engineer's life became much harder over time. They had to be really creative with how they could load their servers, or their 'ship', with different applications without breaking something. Installing a new version of a certain application was a complex project on its own and often needed months of planning and testing. In other words, there was a lot of friction in the software supply chain. But these days, companies rely more and more on software and the release cycles become shorter and shorter. We cannot afford anymore to just have a new release maybe twice a year. Applications need to be updated in a matter of weeks or days, or sometimes even multiple times per day. Companies that do not comply risk going out of business due to the lack of agility. So, *what's the solution?*

A first approach was to use **virtual machines** (**VMs**). Instead of running multiple applications all on the same server, companies would package and run a single application per VM. With it, the compatibility problems were gone and life seemed good again. Unfortunately, the happiness didn't last for long. VMs are pretty heavy beasts on their own since they all contain a full-blown OS such as Linux or Windows Server and all that for just a single application. This is as if in the transportation industry you would use a gigantic ship just to transport a truck load of bananas. What a waste. That can never be profitable. The ultimate solution to the problem was to provide something much more lightweight than VMs but also able to perfectly encapsulate the goods it needed to transport. Here, the goods are the actual application written by our developers plus (and this is important) all the external dependencies of the application, such as framework, libraries, configurations, and more. This holy grail of a software packaging mechanism was the Docker container.

Developers use Docker containers to package their applications, frameworks, and libraries, and then they ship those containers to the testers or to the operations engineers. For the testers and operations engineers, the container is just a black box. Crucially, it is a standardized black box. All containers, no matter what application runs inside them, can be treated equally. The engineers know that if any container runs on their servers, then any other containers should run too. And this is actually true, apart from some edge cases which always exist. Thus, Docker containers are a means to package applications and their dependencies in a standardized way. Docker then coined the phrase—Build, ship and run anywhere.

Why are containers important?

These days, the time between new releases of an application becomes shorter and shorter, yet the software itself doesn't become any simpler. On the contrary, software projects increase in complexity. Thus, we need a way to tame the beast and simplify the software supply chain.

Improving security

We also hear every day how much more cyber crimes are on the rise. Many well-known companies are affected by security breaches. Highly sensitive customer data gets stolen, such as social security numbers, credit card information, and more. But not only customer data is compromised, sensitive company secrets are also stolen.

Containers can help in many ways. First of all, Gartner has found in a recent report that applications running in a container are more secure than their counterparts not running in a container. Containers use Linux security primitives such as Linux kernel namespaces to sandbox different applications running on the same computers and **control groups** (**cgroups**), to avoid the noisy neighbor problem where one bad application is using all available resources of a server and starving all other applications.

Due to the fact that container images are immutable, it is easy to have them scanned for known vulnerabilities and exposures, and in doing so, increase the overall security of our applications.

Another way we can make our software supply chain more secure when using containers is to use **content trust**. Content trust basically ensures that the author of a container image is who they pretend to be and that the consumer of the container image has a guarantee that the image has not been tampered with in transit. The latter is known as a **man-in-the-middle** (**MITM**) attack.

All that I have just said is of course technically also possible without using containers, but since containers introduce a globally accepted standard, it makes it so much easier to implement those best practices and enforce them.

OK, but security is not the only reason why containers are important. There are other reasons as explained in the next two sections.

Simulating production-like environments

One of them is the fact that containers make it easy to simulate a production-like environment, even on a developer's laptop. If we can containerize any application, then we can also containerize, say, a database such as Oracle or MS SQL Server. Now, everyone who has ever had to install an Oracle database on a computer knows that this is not the easiest thing to do and it takes a lot of space away on your computer. You wouldn't want to do that to your development laptop just to test whether the application you developed really works end to end. With containers at hand, I can run a full-blown relational database in a container as easily as saying 1, 2, 3. And when I'm done with testing, I can just stop and delete the container and the database is gone without leaving a trace on my computer.

Since containers are very lean compared to VMs, it is not uncommon to have many containers running at the same time on a developer's laptop without overwhelming the laptop.

Standardizing infrastructure

A third reason why containers are important is that operators can finally concentrate on what they are really good at, provisioning infrastructure, and running and monitoring applications in production. When the applications they have to run on a production system are all containerized, then operators can start to standardize their infrastructure. Every server becomes just another Docker host. No special libraries of frameworks need to be installed on those servers, just an OS and a container runtime such as Docker.

Also, the operators do not have to have any intimate knowledge about the internals of the applications anymore since those applications run self-contained in containers that ought to look like black boxes to the operations engineers, similar to how the shipping containers look to the personnel in the transportation industry.

What's the benefit for me or for my company?

Somebody once said that today, every company of a certain size has to acknowledge that they need to be a software company. Software runs all businesses, period. As every company becomes a software company, there is a need to establish a software supply chain. For the company to remain competitive, their software supply chain has to be secure and efficient. Efficiency can be achieved through thorough automation and standardization. But in all three areas, security, automation, and standardization, containers have proven their superiority.

Large and well-known enterprises have reported that when containerizing existing legacy applications (many call them traditional applications) and establishing a fully automated software supply chain based on containers, they can reduce the cost used for maintenance of those mission-critical applications by a factor of 50 to 60% and they can reduce the time between new releases of these traditional applications by up to 90%.

That said, the adoption of container technology saves these companies a lot of money, and at the same time it speeds up the development process and reduces the time to market.

The Moby project

Originally, when the company Docker introduced Docker containers, everything was open source. Docker didn't have any commercial products at this time. The Docker engine which the company developed was a monolithic piece of software. It contained many logical parts, such as the container runtime, a network library, a RESTful API, a command-line interface, and much more.

Other vendors or projects such as Red Hat or Kubernetes were using the Docker engine in their own products, but most of the time they were only using part of its functionality. For example, Kubernetes did not use the Docker network library of the Docker engine but provided its own way of networking. Red Hat in turn did not update the Docker engine frequently and preferred to apply unofficial patches to older versions of the Docker engine, yet they still called it the **Docker engine**.

Out of all these reasons and many more, the idea emerged that Docker had to do something to clearly separate the Docker open source part from the Docker commercial part. Furthermore, the company wanted to prevent competitors from using and abusing the name Docker for their own gains. This was the main reason why the Moby project was born. It serves as the umbrella for most of the open source components Docker developed and continues to develop. These open source projects do not carry the name Docker in them anymore.

The Moby project encompasses components for image management, secret management, configuration management, and networking and provisioning, to name just a few. Also, part of the Moby project are special Moby tools that are, for example, used to assemble components into runnable artifacts.

Some of the components that technically would belong to the Moby project have been donated by Docker to the **Cloud Native Computing Foundation** (**CNCF**) and thus do not appear in the list of components anymore. The most prominent ones are `containerd` and `runc` which together form the container runtime.

Docker products

Docker currently separates its product lines into two segments. There is the **Community Edition** (**CE**) which is closed source yet completely free, and then there is the **Enterprise Edition** (**EE**) which is also a closed source and needs to be licensed on a yearly basis. The enterprise products are backed by 24 x 7 support and are supported with bug fixes much longer than their CE counterparts.

Docker CE

The Docker community edition includes products such as the Docker Toolbox, Docker for Mac, and Docker for Windows. All these three products are mainly targeting developers.

Docker for Mac and Docker for Windows are easy-to-install desktop applications that can be used to build, debug, and test Dockerized applications or services on a Mac or on Windows. Docker for Mac and Docker for Windows are complete development environments which deeply integrated with their respective hypervisor framework, networking, and filesystem. These tools are the fastest and most reliable way to run Docker on a Mac or on Windows.

Under the umbrella of the CE, there are also two products that are more geared towards operations engineers. Those products are Docker for Azure and Docker for AWS.

For example, with Docker for Azure, which is a native Azure application, you can set up Docker in a few clicks, optimized for and integrated to the underlying Azure **Infrastructure as a Service** (**IaaS**) services. It helps operations engineers to accelerate the time it takes to build and run Docker applications in Azure.

Docker for AWS works in a very similar way, but for Amazon's cloud.

Docker EE

The Docker EE consists of the two products **Universal Control Plane** (**UCP**) and **Docker Trusted Registry** (**DTR**) that both run on top of Docker Swarm. Both are Swarm applications. Docker EE builds on top of the upstream components of the Moby project and adds enterprise-grade features such as **role-based access control** (**RBAC**), multi tenancy, mixed clusters of Docker Swarm and Kubernetes, web-based UI, and content trust, as well as image scanning on top of it.

The container ecosystem

There has never been a new technology introduced in IT that has penetrated the landscape as quickly and thoroughly as containers. Any company that doesn't want to be left behind cannot ignore containers. This huge interest in containers from all sectors of the industry has triggered a lot of innovation in this sector. Numerous companies have specialized in containers and either provide products that build on top of this technology or build tools that support it.

Initially, Docker didn't have a solution for container orchestration thus other companies or projects, open source or not, tried to close this gap. The most prominent one is Kubernetes which was initiated by Google and then later donated to the CNCF. Other container orchestration products are Apache Mesos, Rancher, Red Hat's Open Shift, Docker's own Swarm, and more.

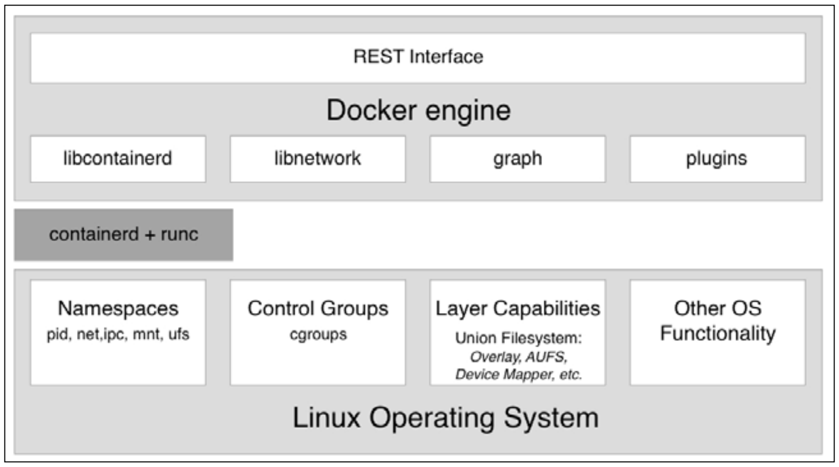
More recently, the trend goes towards a service mesh. This is the new buzz word. As we containerize more and more applications, and as we refactor those applications into more microservice-oriented applications, we run into problems that simple orchestration software cannot solve anymore in a reliable and scalable way. Topics in this area are service discovery, monitoring, tracing, and log aggregation. Many new projects have emerged in this area, the most popular one at this time being Istio, which is also part of the CNCF.

Many say that the next step in the evolution of software is functions, or more precisely, **Functions as a Service** (**FaaS**). Some projects exist that provide exactly this kind of service and are built on top of containers. One prominent example is OpenFaaS.

We have only scratched the surface of the container ecosystem. All big IT companies such as Google, Microsoft, Intel, Red Hat, IBM, and more are working feverishly on containers and related technologies. The CNCF which is mainly about containers and related technologies, has so many registered projects, that they do not all fit on a poster anymore. It's an exciting time to work in this area. And in my humble opinion, this is only the beginning.

Container architecture

Now, let's discuss on a high level how a system that can run Docker containers is designed. The following diagram illustrates what a computer on which Docker has been installed looks like. By the way, a computer which has Docker installed is often called a Docker host, because it can run or host Docker containers:



High-level architecture diagram of the Docker engine

In the preceding diagram, we see three essential parts:

- On the bottom, we have the Linux operating system
- In the middle dark gray, we have the container runtime
- On the top, we have the Docker engine

Containers are only possible due to the fact that the Linux OS provides some primitives, such as namespaces, control groups, layer capabilities, and more which are leveraged in a very specific way by the container runtime and the Docker engine. Linux kernel namespaces such as **process ID** (**pid**) namespaces or **network** (**net**) namespaces allow Docker to encapsulate or sandbox processes that run inside the container. Control groups make sure that containers cannot suffer from the noisy neighbor syndrome, where a single application running in a container can consume most or all of the available resources of the whole Docker host. Control groups allow Docker to limit the resources, such as CPU time or the amount of RAM that each container gets maximally allocated.

The container runtime on a Docker host consists of `containerd` and `runc`. `runc` is the low-level functionality of the container runtime and `containerd`, which is based on `runc`, provides the higher-level functionality. Both are open source and have been donated by Docker to the CNCF.

The container runtime is responsible for the whole life cycle of a container. It pulls a container image (which is the template for a container) from a registry if necessary, creates a container from that image, initializes and runs the container, and eventually stops and removes the container from the system when asked.

The Docker engine provides additional functionality on top of the container runtime, such as network libraries or support for plugins. It also provides a REST interface over which all container operations can be automated. The Docker command-line interface that we will use frequently in this book is one of the consumers of this REST interface.

Summary

In this chapter, we looked at how containers can massively reduce the friction in the software supply chain and on top of that, make the supply chain much more secure.

In the upcoming chapter, we will familiarize ourselves with containers. We will learn how to run, stop, and remove containers and otherwise manipulate them. We will also have a pretty good overview over the anatomy of containers. For the first time, we're really going to get our hands dirty and play with these containers, so stay tuned.

Questions

Please solve the following questions to assess your learning progress:

1. Which statements are correct (multiple answers are possible)?
    1. A container is kind of a lightweight VM
    2. A container only runs on a Linux host
    3. A container can only run one process
    4. The main process in a container always has PID 1
    5. A container is one or more processes encapsulated by Linux namespaces and restricted by cgroups
2. Explain to an interested layman in your own words, maybe using analogies, what a container is.
3. Why are containers considered to be a game changer in IT? Name three to four reasons.
4. What does it mean when we claim: *If a container runs on a given platform then it runs anywhere...?* Name two to three reasons why this is true.
5. True or False: *Docker containers are only really useful for modern greenfield applications based on microservices*. Please justify your answer.
6. How much does a typical enterprise save when containerizing their legacy applications?
    1. 20%
    2. 33%
    3. 50%
    4. 75%
7. Which two core concepts of Linux are containers based on?

Further reading

Here is a list of links that lead to more detailed information regarding topics we have discussed in this chapter:

- *Docker overview* at `https://docs.docker.com/engine/docker-overview/`
- *The Moby project* at `https://mobyproject.org/`
- *Docker products* at `https://www.docker.com/get-docker`
- *Cloud Native Computing Foundation* at `https://www.cncf.io/`
- *containerd – industry standard container runtime* at `https://containerd.io/`

2
Setting up a Working Environment

In the last chapter, we learned what Docker containers are and why they're important. We learned what kinds of problem containers solve in a modern software supply chain.

In this chapter, we are going to prepare our personal or working environment to work efficiently and effectively with Docker. We will discuss in detail how to set up an ideal environment for developers, DevOps, and operators that can be used when working with Docker containers.

This chapter covers the following topics:

- The Linux command shell
- PowerShell for Windows
- Using a package manager
- Choosing a code editor
- Docker Toolbox
- Docker for macOS and Docker for Windows
- Minikube
- Cloning the Source Code Repository

After completing this chapter, you will be able to do the following:

- Use an editor on your laptop that is able to edit simple files such as a Dockerfile or a `docker-compose.yml` file
- Use a shell such as Bash on macOS and PowerShell on Windows to execute Docker commands and other simple operations, such as navigating the folder structure or creating a new folder
- Install Docker for macOS or Docker for Windows on your computer
- Execute simple Docker commands such as `docker version` or `docker container run` on your Docker for macOS or Docker for Windows
- Successfully install Docker Toolbox on your computer
- Use `docker-machine` to create a Docker host on VirtualBox
- Configure your local Docker CLI to remote access a Docker host running in VirtualBox

Technical requirements

For this chapter, you will need either macOS or Windows, preferably Windows 10 Professional, installed. You should also have free internet access to download applications and the permission to install those applications on your laptop.

The Linux command shell

Docker containers were first developed on Linux for Linux. It is thus natural that the primary command-line tool used to work with Docker, also called a shell, is a Unix shell; remember, Linux derives from Unix. Most developers use the Bash shell. On some lightweight Linux distributions, such as **Alpine**, Bash is not installed and consequently one has to use the simpler Bourne shell, just called **sh**. Whenever we are working in a Linux environment, such as inside a container or on a Linux VM, we will use either `/bin/bash` or `/bin/sh`, depending on their availability.

Although macOS X is not a Linux OS, Linux and OS X are both flavors of Unix and thus support the same types of tools. Among those tools are the shells. So, when working on a macOS, you will probably be using the Bash shell.

In this book, we expect from the readers a familiarity with the most basic scripting commands in Bash, and PowerShell if you are working on Windows. If you are an absolute beginner, then we strongly recommend that you familiarize yourself with the following cheat sheets:

- *Linux Command Line Cheat Sheet* by Dave Child at `http://bit.ly/2mTQr8l`
- *PowerShell Basic Cheat Sheet* at `http://bit.ly/2EPHxze`

PowerShell for Windows

On a Windows computer, laptop, or server, we have multiple command-line tools available. The most familiar is the command shell. It has been available on any Windows computer for decades. It is a very simple shell. For more advanced scripting, Microsoft has developed PowerShell. PowerShell is very powerful and very popular among engineers working on Windows. On Windows 10, finally, we have the so-called **Windows Subsystem for Linux**, which allows us to use any Linux tool, such as the Bash or Bourne shells. Apart from this, there also exist other tools that install a Bash shell on Windows, for example, the Git Bash shell. In this book, all commands will use Bash syntax. Most of the commands also run in PowerShell.

Our recommendation for you is thus to either use PowerShell or any other Bash tool to work with Docker on Windows.

Using a package manager

The easiest way to install software on a macOS or Windows laptop is to use a good package manager. On a macOS, most people use **Homebrew** and on Windows, **Chocolatey** is a good choice.

Installing Homebrew on a macOS

Installing Homebrew on a macOS is easy; just follow the instructions at `https://brew.sh/`.

The following is the command to install Homebrew:

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

Once the installation is finished, test whether Homebrew is working by entering `brew --version` in the Terminal. You should see something like this:

```
$ brew --version
Homebrew 1.4.3
Homebrew/homebrew-core (git revision f4e35; last commit 2018-01-11)
```

Now, we are ready to use Homebrew to install tools and utilities. If we, for example, want to install the Vi text editor, we can do so like this:

```
$ brew install vim
```

This will then download and install the editor for you.

Installing Chocolatey on Windows

To install the Chocolatey package manager on Windows, please follow the instructions at `https://chocolatey.org/` or just execute the following command in a PowerShell Terminal that you have run as administrator:

```
PS> Set-ExecutionPolicy Bypass -Scope Process -Force; iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/install.
ps1'))
```

Once Chocolatey is installed, test it with the command `choco` without additional parameters. You should see output similar to the following:

```
PS> choco
Chocolatey v0.10.3
```

To install an application such as the Vi editor, use the following command:

```
PS> choco install -y vim
```

The `-y` parameter makes sure that the installation happens without asking for reconfirmation. Please note that once Chocolatey has installed an application, you need to open a new PowerShell window to use it.

Choosing a code editor

Using a good code editor is essential to working productively with Docker. Of course, which editor is the best is highly controversial and depends on your personal preference. A lot of people use Vim, or others such as Emacs, Atom, Sublime, or **Visual Studio** (**VS**) Code, to just name a few. If you have not yet decided which editor is best suited for you, then I highly recommend that you try VS Code. This is a free and lightweight editor, yet it is very powerful and is available for macOS, Windows, and Linux. Give it a try. You can download VS Code from `https://code.visualstudio.com/download`.

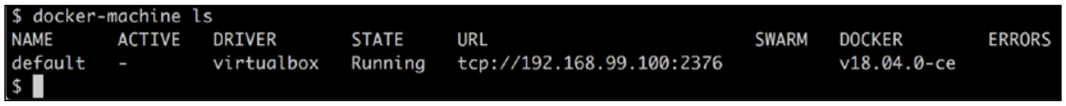
But if you already have a favorite code editor, then please continue using it. As long as you can edit text files, you're good to go. If your editor supports syntax highlighting for Dockerfiles and JSON and YAML files, then even better.

Docker Toolbox

Docker Toolbox has been available for developers for a few years. It precedes the newer tools such as Docker for macOS and Docker for Windows. The toolbox allows a user to work very elegantly with containers on any macOS or Windows computer. Containers must run on a Linux host. Neither Windows or macOS can run containers natively. Thus, we need to run a Linux VM on our laptop, where we can then run our containers. Docker Toolbox installs VirtualBox on our laptop, which is used to run the Linux VMs we need.

As a Windows user, you might already be aware that there exists so-called Windows containers that run natively on Windows. And you are right. Recently, Microsoft has ported the Docker engine to Windows and it is now possible to run Windows containers directly on a Windows Server 2016 without the need for a VM. So, now we have two flavors of containers, Linux containers and Windows containers. The former only run on Linux host and the latter only run on a Windows Server. In this book, we are exclusively discussing Linux containers, but most of the things we learn also apply to Windows containers.

Let's use `docker-machine` to set up our environment. Firstly, we list all Docker-ready VMs we have currently defined on our system. If you have just installed Docker Toolbox, you should see the following output:



```
$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        ERRORS
default   -        virtualbox   Running   tcp://192.168.99.100:2376           v18.04.0-ce
$ 
```

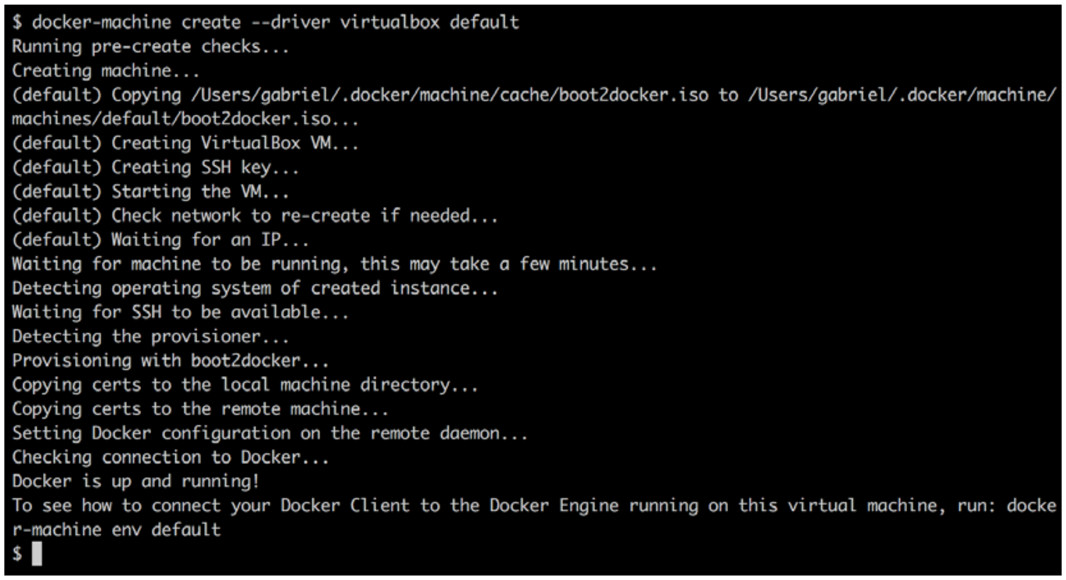
List of all Docker-ready VMs

The IP address used might be different in your case, but it will be definitely in the `192.168.0.0/24` range. We can also see that the VM has Docker version `18.04.0-ce` installed.

If, for some reason, you don't have a default VM or you have accidentally deleted it, you can create it using the following command:

```
$ docker-machine create --driver virtualbox default
```

The output you should see looks as follows:



```
$ docker-machine create --driver virtualbox default
Running pre-create checks...
Creating machine...
(default) Copying /Users/gabriel/.docker/machine/cache/boot2docker.iso to /Users/gabriel/.docker/machine/
machines/default/boot2docker.iso...
(default) Creating VirtualBox VM...
(default) Creating SSH key...
(default) Starting the VM...
(default) Check network to re-create if needed...
(default) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: docke
r-machine env default
$ 
```

Creating the VM called default in VirtualBox

To see how to connect your Docker client to the Docker Engine running on this virtual machine, run the following command:

```
$ docker-machine env default
```

Once we have our VM called `default` ready, we can try to SSH into it:

```
$ docker-machine ssh default
```

When executing the preceding command, we are greeted by a boot2docker welcome message.

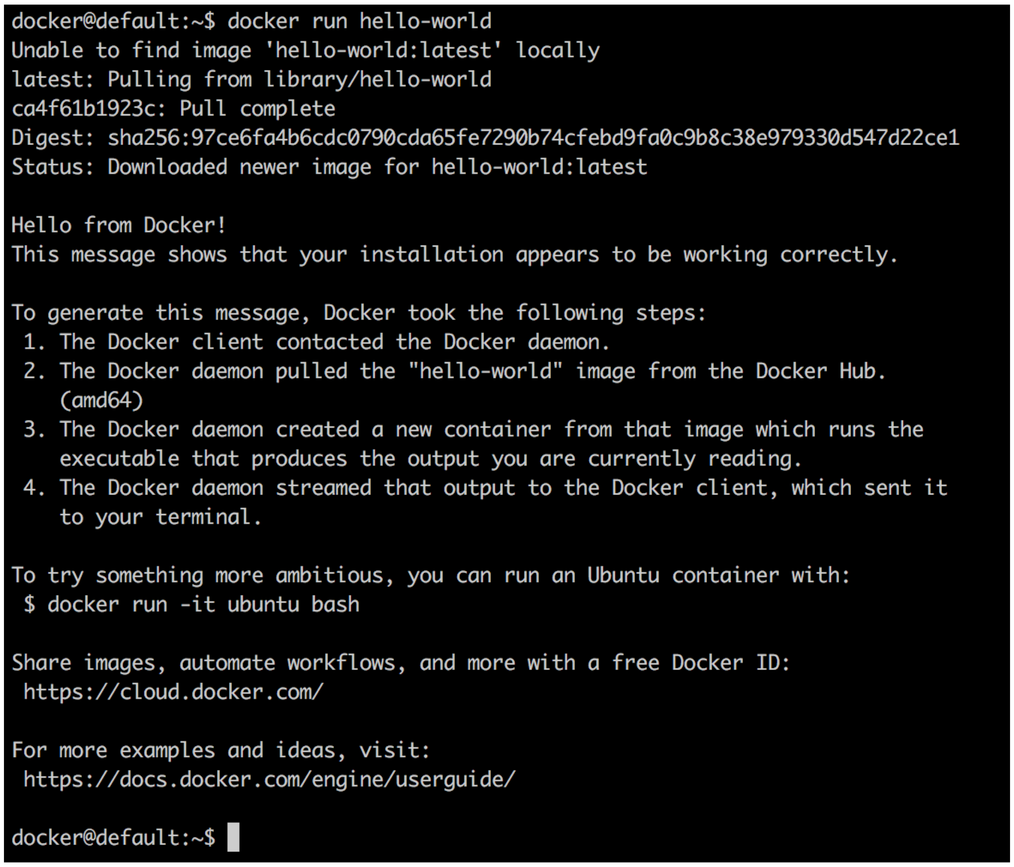
Type `docker --version` in the Command Prompt as follows:

```
docker@default:~$ docker --version
Docker version 18.06.1-ce, build e68fc7a
```

Now, let's try to run a container:

```
docker@default:~$ docker run hello-world
```

This will produce the following output:



```
docker@default:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:97ce6fa4b6cdc0790cda65fe7290b74cfebd9fa0c9b8c38e979330d547d22ce1
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/

docker@default:~$
```

Running the Docker Hello World container

Docker for macOS and Docker for Windows

If you are using a macOS or have Windows 10 Professional installed on your laptop, then we strongly recommend that you install Docker for macOS or Docker for Windows. These tools give you the best experience when working with containers. Note, older versions of Windows or Windows 10 Home edition cannot run Docker for Windows. Docker for Windows uses Hyper-V to run containers transparently in a VM but Hyper-V is not available on older versions of Windows nor is it available in the Home edition.

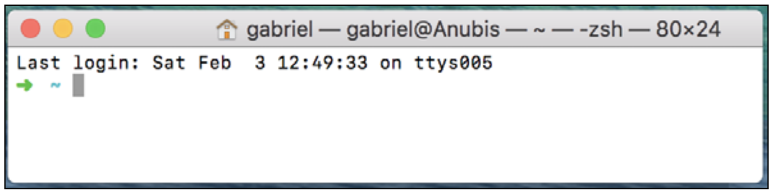
Installing Docker for macOS

Navigate to the following link to download Docker for macOS at `https://docs.docker.com/docker-for-mac/install/`.

> There is a stable version and a so-called edge version of the tool available. In this book, we are going to use some newer features and Kubernetes, which at the time of writing are only available in the edge version. Thus, please select this version.

To start the installation:

1. Click on the **Get Docker for Mac (Edge)** button and follow the instructions.
2. Once you have successfully installed Docker for macOS, open a Terminal. Press command + spacebar to open Spotlight and type `terminal`, then hit *enter*. The Apple Terminal will open as follows:



```
Last login: Sat Feb  3 12:49:33 on ttys005
➜ ~
```

Apple Terminal window

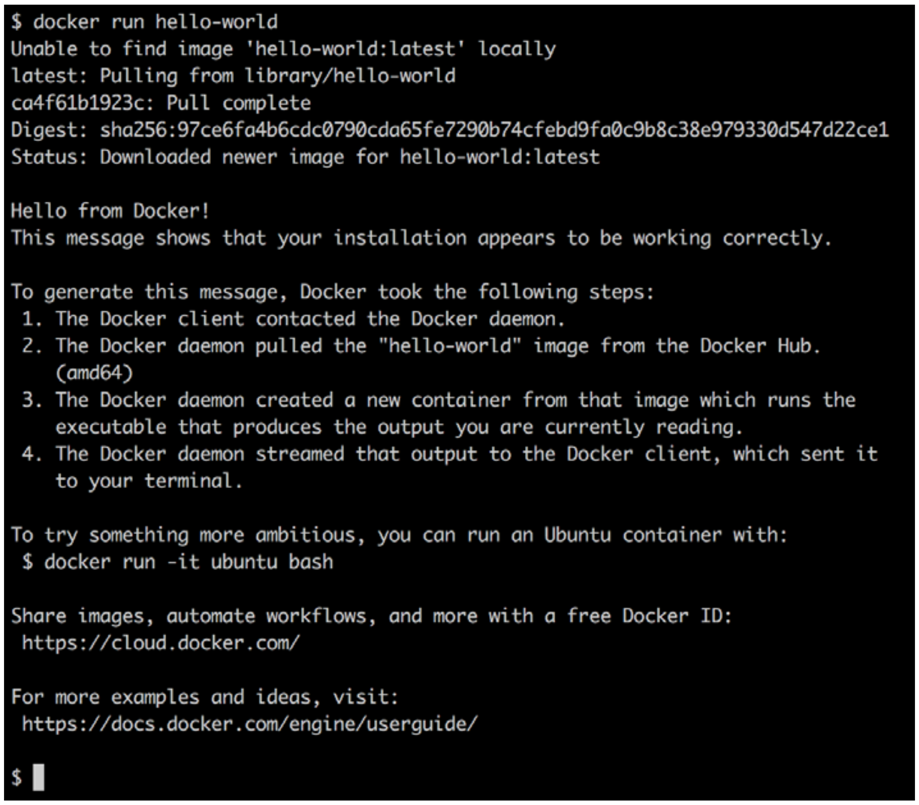
3. Type `docker --version` in the Command Prompt and hit *enter*. If Docker for macOS is correctly installed, you should get an output similar to the following:

```
$ docker -version
Docker version 18.02.0-ce-rc2, build f968a2c
```

4. To see whether you can run containers, enter the following command into the Terminal and hit *enter*:

```
$ docker run hello-world
```

If all goes well, your output should look something like the following:



```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:97ce6fa4b6cdc0790cda65fe7290b74cfebd9fa0c9b8c38e979330d547d22ce1
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/

$ ▌
```

Running the Hello World container on Docker for macOS

Congratulations, you are now ready to work with Docker containers.

Installing Docker for Windows

Note, you can only install Docker for Windows on Windows 10 Professional or Windows Server 2016 since it requires Hyper-V, which is not available on older Windows versions or on the Home edition of Windows 10. If you are using Windows 10 Home or an older version of Windows, you will need to stick with Docker Toolbox.

1. Navigate to the following link to download Docker for Windows at `https://docs.docker.com/docker-for-windows/install/`.

   There is a stable version and a so-called edge version of the tool available. In this book, we are going to use some newer features and Kubernetes, which at the time of writing are only available in the edge version. Thus, please select this version.

2. To start the installation, click on the **Get Docker for Windows (Edge)** button and follow the instructions. With Docker for Windows, you can develop, run, and test Linux containers and Windows containers. In this book, though, we are only discussing Linux containers.
3. Once you have successfully installed Docker for Windows. Open a PowerShell window and type `docker --version` in the Command Prompt. You should see something like the following:

```
PS> docker --version
Docker version 18.04.0-ce, build 3d479c0
```

Using docker-machine on Windows with Hyper-V

If you have Docker for Windows installed on your Windows laptop, then you also have Hyper-V enabled. In this case, you can't use Docker Toolbox since it uses VirtualBox, and Hyper-V and VirtualBox cannot coexist and run at the same time. In this case, you can use `docker-machine` with the Hyper-V driver.

1. Open a PowerShell console as an administrator. Install `docker-machine` using Chocolatey as follows:

```
PS> choco install -y docker-machine
```

2. Using Window's Hyper-V manager create a new internal switch called `DM Internal Switch`, where `DM` stands for **docker-machine**.
3. Create a VM called `default` in Hyper-V with the following command:

   ```
   PS> docker-machine create --driver hyperv --hyperv-virtual-
   switch "DM Internal Switch" default
   ```

   [You must run the preceding command in administrator mode or it will fail.]

   You should see the following output generated by the preceding command:

   ```
   Running pre-create checks...
   (boot2docker) Image cache directory does not exist, creating
   it at C:\Users\Docker\.docker\machine\cache...
   (boot2docker) No default Boot2Docker ISO found locally,
   downloading the latest release...
   (boot2docker) Latest release for github.com/boot2docker/
   boot2docker is v18.06.1-ce
   ....
   ....
   Checking connection to Docker...
   Docker is up and running!
   To see how to connect your Docker Client to the Docker Engine
   running on this virtual machine, run: C:\Program Files\Doc
   ker\Docker\Resources\bin\docker-machine.exe env default
   ```

4. To see how to connect your Docker client to the Docker Engine running on this virtual machine, run the following:

   ```
   C:\Program Files\Docker\Docker\Resources\bin\docker-machine.
   exe env default
   ```

5. Listing all VMs generated by `docker-machine` gives us the following output:

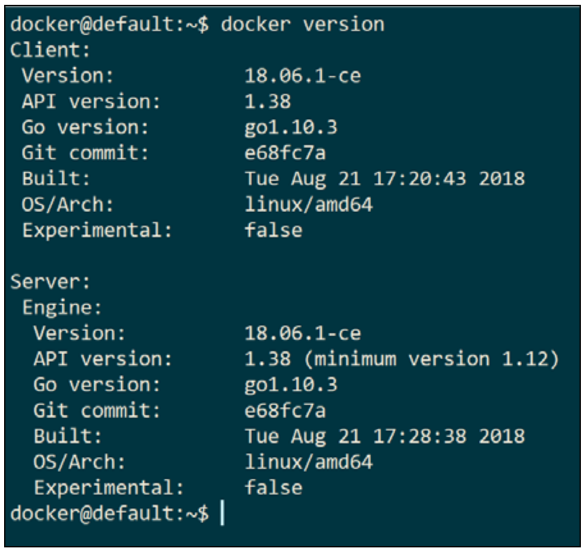
   ```
   PS C:\WINDOWS\system32> docker-machine ls
   NAME          ACTIVE   DRIVER   STATE     URL
   SWARM   DOCKER        ERRORS
   default .     -        hyperv   Running   tcp://[...]:2376
   v18.06.1-ce
   ```

6. Now, let's SSH into our `boot2docker` VM:

   ```
   PS> docker-machine ssh default
   ```

You should be greeted by the welcome screen.

We can test the VM by executing our `docker version` command, which is shown as follows:



```
docker@default:~$ docker version
Client:
 Version:           18.06.1-ce
 API version:       1.38
 Go version:        go1.10.3
 Git commit:        e68fc7a
 Built:             Tue Aug 21 17:20:43 2018
 OS/Arch:           linux/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.06.1-ce
  API version:      1.38 (minimum version 1.12)
  Go version:       go1.10.3
  Git commit:       e68fc7a
  Built:            Tue Aug 21 17:28:38 2018
  OS/Arch:          linux/amd64
  Experimental:     false
docker@default:~$
```

Version of the Docker client (CLI) and server

This is definitely a Linux VM, as we can see on the `OS/Arch` entry, and has Docker `18.06.1-ce` installed.

Minikube

If you cannot use Docker for macOS or Windows or, for some reason, you only have access to an older version of the tool that does not yet support Kubernetes, then it is a good idea to install Minikube. Minikube provisions a single-node Kubernetes cluster on your workstation and is accessible through **kubectl**, which is the command-line tool used to work with Kubernetes.

Installing Minikube on macOS and Windows

To install Minikube for macOS or Windows, navigate to the following link at `https://kubernetes.io/docs/tasks/tools/install-minikube/`.

Follow the instructions carefully. If you have the Docker Toolbox installed, then you already have a hypervisor on your system since the Docker Toolbox installer also installed VirtualBox. Otherwise, I recommend that you install VirtualBox first.

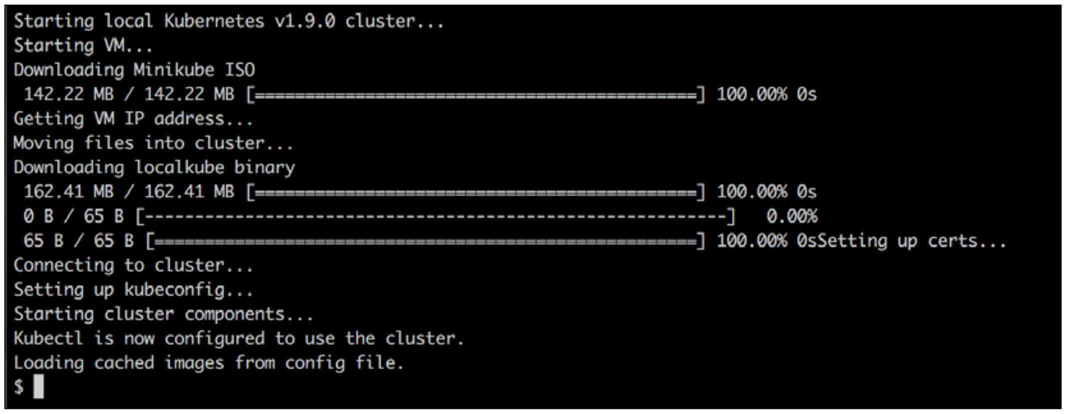
If you have Docker for macOS or Windows installed, then you already have `kubectl` installed with it, thus you can skip that step too. Otherwise, follow the instructions on the site.

Finally, select the latest binary for Minikube for macOS or Windows and install it. For macOS, the latest binary is called `minikube-darwin-amd64` and for Windows it is `minikube-windows-amd64`.

Testing Minikube and kubectl

Once Minikube is successfully installed on your workstation, open a Terminal and test the installation.

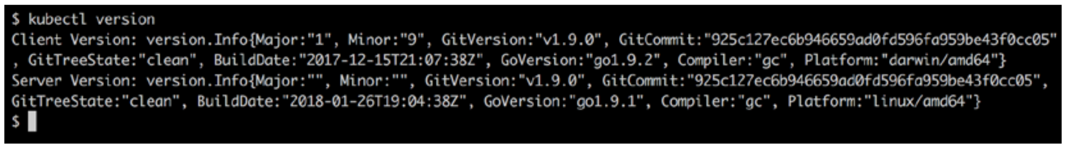
1. First, we need to start Minikube. Enter `minikube start` at the command line. The output should look like the following:



```
Starting local Kubernetes v1.9.0 cluster...
Starting VM...
Downloading Minikube ISO
 142.22 MB / 142.22 MB [============================================] 100.00% 0s
Getting VM IP address...
Moving files into cluster...
Downloading localkube binary
 162.41 MB / 162.41 MB [============================================] 100.00% 0s
 0 B / 65 B [----------------------------------------------------------]   0.00%
 65 B / 65 B [======================================================] 100.00% 0sSetting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
$ 
```

Starting Minikube

2. Now, enter `kubectl version` and hit *enter* to see something like the following screenshot:



```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"9", GitVersion:"v1.9.0", GitCommit:"925c127ec6b946659ad0fd596fa959be43f0cc05"
, GitTreeState:"clean", BuildDate:"2017-12-15T21:07:38Z", GoVersion:"go1.9.2", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"", Minor:"", GitVersion:"v1.9.0", GitCommit:"925c127ec6b946659ad0fd596fa959be43f0cc05",
GitTreeState:"clean", BuildDate:"2018-01-26T19:04:38Z", GoVersion:"go1.9.1", Compiler:"gc", Platform:"linux/amd64"}
$ 
```

Determining the version of the Kubernetes client and server

   If the preceding command fails, for example, by timing out, then it could be that your `kubectl` is not configured for the right context. `kubectl` can be used to work with many different Kubernetes clusters. Each cluster is called a context.

3. To find out which context `kubectl` is currently configured for, use the following command:

   ```
   $ kubectl config current-context
   minikube
   ```

   The answer should be `minikube`, as shown in the preceding output.

4. If this is not the case, use `kubectl config get-contexts` to list all contexts that are defined on your system and then set the current context to `minikube` as follows:

   ```
   $ kubectl config use-context minikube
   ```

   The configuration for `kubectl`, where it stores the contexts, is normally found in `~/.kube/config`, but this can be overridden by defining an environment variable called `KUBECONFIG`. You might need to unset this variable if it is set on your computer.

   For more in-depth information about how to configure and use Kubernetes contexts, consult the link at `https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/`.

   Assuming Minikube and `kubectl` work as expected, we can now use `kubectl` to get information about the Kubernetes cluster.

5. Enter the following command:

   ```
   $ kubectl get nodes
   NAME       STATUS    ROLES     AGE       VERSION
   minikube   Ready     <none>    47d       v1.9.0
   ```

Evidently, we have a cluster of one node, which in my case has Kubernetes `v1.9.0` installed on it.

Cloning the source code repository

This book is accompanied by source code publicly available in a GitHub repository at `https://github.com/appswithdockerandkubernetes/labs`. Clone that repository to your local machine.

First create a new folder for example, in your home folder such as `apps-with-docker-and-kubernetes` and navigate to it:

```
$ mkdir -p ~/apps-with-docker-and-kubernetes \
  cd apps-with-docker-and-kubernetes
```

And then clone the repository with the following command:

```
$ git clone https://github.com/appswithdockerandkubernetes/labs.
git
```

Summary

In this chapter, we set up and configured our personal or working environment so that we can productively work with Docker containers. This equally applies for developers, DevOps, and operations engineers. In that context, we made sure that we use a good editor, have Docker for macOS or Windows installed, and can also use `docker-machine` to create VMs in VirtualBox or Hyper-V which we can use to run and test containers.

In the next chapter, we're going to learn all the important facts about containers. For example, we will explore how we can run, stop, list, and delete containers, but more than that, we will also dive deep into the anatomy of containers.

Questions

On the basis of your reading of this chapter, please answer the following questions:

1. What is `docker-machine` used for? Name three to four scenarios.
2. True or false? With Docker for Windows, one can develop and run Linux containers.
3. Why are good scripting skills (such as Bash or PowerShell) essential for a productive use of containers?
4. Name three to four Linux distributions on which Docker is certified to run.
5. Name all the Windows versions on which you can run Windows containers.

Further reading

Consider the following link for further reading:

- *Run Docker on Hyper-V with Docker Machine* at `http://bit.ly/2HGMPiI`

3
Working with Containers

In the previous chapter, you learned how to optimally prepare your working environment for the productive and frictionless use of Docker. In this chapter, we are going to get our hands dirty and learn everything that is important to work with containers. Here are the topics we're going to cover in this chapter:

- Running the first container
- Starting, stopping, and removing containers
- Inspecting containers
- Exec into a running container
- Attaching to a running container
- Retrieving container logs
- Anatomy of containers

After finishing this chapter you will be able to do the following things:

- Run, stop, and delete a container based on an existing image, such as NGINX, busybox, or alpine
- List all containers on the system
- Inspect the metadata of a running or stopped container
- Retrieve the logs produced by an application running inside a container
- Run a process such as `/bin/sh` in an already-running container.
- Attach a Terminal to an already-running container
- Explain in your own words to an interested layman the underpinnings of a container

Technical requirements

For this chapter, you should have installed Docker for Mac or Docker for Windows. If you are on an older version of Windows or are using Windows 10 Home Edition, then you should have Docker Toolbox installed and ready to use. On macOS, use the Terminal application, and on Windows, a PowerShell console to try out the commands you will be learning.

Running the first container

Before we start, we want to make sure that Docker is installed correctly on your system and ready to accept your commands. Open a new Terminal window and type in the following command:

```
$ docker -v
```

If everything works correctly, you should see the version of Docker installed on your laptop output in the Terminal. At the time of writing, it looks like this:

```
Docker version 17.12.0-ce-rc2, build f9cde63
```

If this doesn't work, then something with your installation is not right. Please make sure that you have followed the instructions in the previous chapter on how to install Docker for Mac or Docker for Windows on your system.

So, you're ready to see some action. Please type the following command into your Terminal window and hit return:

```
$ docker container run alpine echo "Hello World"
```

When you run the preceding command the first time, you should see an output in your Terminal window similar to this:

```
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
2fdfe1cd78c2: Pull complete
Digest: sha256:ccba511b...
Status: Downloaded newer image for alpine:latest
Hello World
```

Now that was easy! Let's try to run the very same command again:

```
$ docker container run alpine echo "Hello World"
```

The second, third, or nth time you run the preceding command, you should see only this output in your Terminal:

```
Hello World
```

Try to work out why the first time you run a command you see a different output than all the subsequent times. Don't worry if you can't figure it out, we will explain the reasons in detail in the following sections of the chapter.

Starting, stopping, and removing containers

You have successfully run a container in the previous section. Now we want to investigate in detail what exactly happened and why. Let's look again at the command we used:

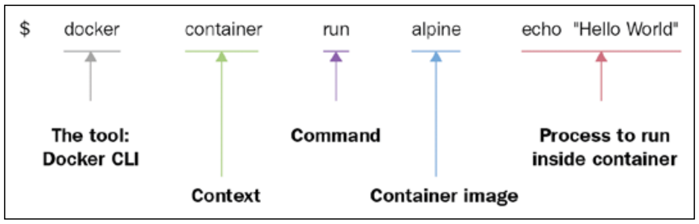
```
$ docker container run alpine echo "Hello World"
```

This command contains multiple parts. First and foremost, we have the word `docker`. This is the name of the Docker **command-line interface** (**CLI**), which we are using to interact with the Docker engine that is responsible to run containers. Next, we have the word `container`, which indicates the context we are working with. As we want to run a container, our context is the word `container`. Next is the actual command we want to execute in the given context, which is `run`.

Let me recap—so far, we have `docker container run`, which means, *Hey Docker, we want to run a container....*

Now we also need to tell Docker which container to run. In this case, this is the so-called `alpine` container. Finally, we need to define what kind of process or task shall be executed inside the container when it is running. In our case, this is the last part of the command, `echo "Hello World"`.

The following figure can help you to get a better approach to the whole thing:



Anatomy of the docker container run expression

Now that we have understood the various parts of a command to run a container, let's try to run another container with a different process running inside it. Type the following command into your Terminal:

```
$ docker container run centos ping -c 5 127.0.0.1
```

You should see output in your Terminal window similar to the following:

```
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
85432449fd0f: Pull complete
Digest: sha256:3b1a65e9a05...
Status: Downloaded newer image for centos:latest
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.022 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.019 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.029 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.030 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.029 ms

--- 127.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4103ms
rtt min/avg/max/mdev = 0.021/0.027/0.029/0.003 ms
```

What changed is that, this time, the container image we're using is `centos` and the process we're executing inside the `centos` container is `ping -c 5 127.0.0.1`, which pings the loopback address five times until it stops.

Let's analyze the output in detail:

- The first line is as follows:

  ```
  Unable to find image 'centos:latest' locally
  ```

  This tells us that Docker didn't find an image named `centos:latest` in the local cache of the system. So, Docker knows that it has to pull the image from some registry where container images are stored. By default, your Docker environment is configured such as that images are pulled from the Docker Hub at `docker.io`. This is expressed by the second line, as follows:

  ```
  latest: Pulling from library/centos
  ```

- The next three lines of output are as follows:

  ```
  85432449fd0f: Pull complete
  Digest: sha256:3b1a65e9a05...
  Status: Downloaded newer image for centos:latest
  ```

  This tells us that Docker has successfully pulled the image centos:latest from the Docker Hub.

All the subsequent lines of the output are generated by the process we ran inside the container, which is the ping tool in this case. If you have been attentive so far, then you might have noticed the keyword `latest` occurring a few times. Each image has a version (also called a tag), and if we don't specify a version explicitly, then Docker automatically assumes it as latest.

If we run the preceding container again on our system, the first five lines of the output will be missing since, this time, Docker will find the container image cached locally and thus won't have to download it first. To verify this, try it out.

Running a random quotes container

For the subsequent sections of this chapter, we need a container that runs continuously in the background and produces some interesting output. That's why we have chosen an algorithm that produces random quotes. The API that produces those free random quotes can be found at `https://talaikis.com/random_quotes_api/`.

Now the goal is to have a process running inside a container that produces a new random quote every five seconds and outputs the quote to STDOUT. The following script will do exactly that:

```
while :
do
    wget -qO- https://talaikis.com/api/quotes/random
    printf 'n'
    sleep 5
done
```

Try it in a Terminal window. Stop the script by pressing *Ctrl*+ *C*. The output should look similar to this:

```
{"quote":"Martha Stewart is extremely talented. Her designs are
picture perfect. Our philosophy is life is messy, and rather than
being afraid of those messes we design products that work the way we
live.","author":"Kathy Ireland","cat":"design"}
```

```
{"quote":"We can reach our potential, but to do so, we must
reach within ourselves. We must summon the strength, the will,
and the faith to move forward - to be bold - to invest in our
future.","author":"John Hoeven","cat":"faith"}
```

Each response is a JSON-formatted string with the quote, its author, and its category.

Now, let's run this in an `alpine` container as a daemon in the background. For this, we need to compact the preceding script into a one-liner and execute it using the `/bin/sh -c "..."` syntax. Our Docker expression will look as follows :

```
$ docker container run -d --name quotes alpine \
   /bin/sh -c "while :; do wget -qO- https://talaikis.com/api/quotes/
random; printf '\n'; sleep 5; done"
```

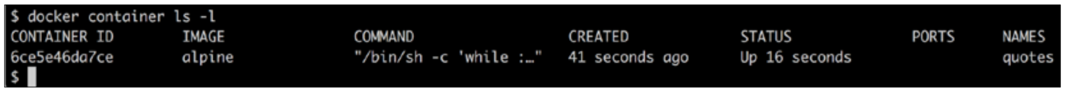
In the preceding expression, we have used two new command-line parameters, `-d` and `--name`. The `-d` tells Docker to run the process running in the container as a Linux daemon. The `--name` parameter in turn can be used to give the container an explicit name. In the preceding sample, the name we chose is `quotes`.

If we don't specify an explicit container name when we run a container, then Docker will automatically assign the container a random but unique name. This name will be composed of the name of a famous scientist and and adjective. Such names could be `boring_borg` or `angry_goldberg`. Quite humorous our Docker engineers, *isn't it?*

One important takeaway is that the container name has to be unique on the system. Let's make sure that the quotes container is up and running:

```
$ docker container ls -l
```

This should give us something like this:



```
$ docker container ls -l
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
6ce5e46da7ce        alpine              "/bin/sh -c 'while :…"   41 seconds ago      Up 16 seconds                           quotes
$
```

Listing the last run container

The important part of the preceding output is the `STATUS` column, which in this case is `Up 16 seconds`. That is, the container has been up and running for 16 seconds now.

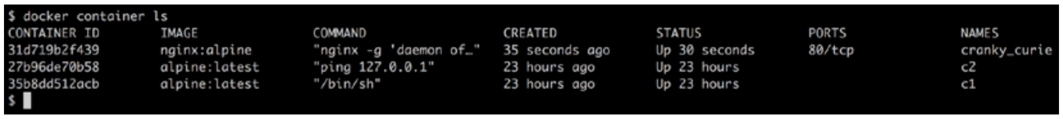
Don't worry if the last Docker command is not yet familiar to you, we will come back to it in the next section.

Listing containers

As we continue to run containers over time, we get a lot of them in our system. To find out what is currently-running on our host, we can use the container `list` command as follows:

```
$ docker container ls
```

This will list all currently-running containers. Such a list might look similar to this:



```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
31d719b2f439        nginx:alpine        "nginx -g 'daemon of…"   35 seconds ago      Up 30 seconds       80/tcp              cranky_curie
27b96de70b58        alpine:latest       "ping 127.0.0.1"         23 hours ago        Up 23 hours                             c2
35b8dd512acb        alpine:latest       "/bin/sh"                23 hours ago        Up 23 hours                             c1
$ 
```

List of all containers running on the system

By default, Docker outputs seven columns with the following meanings:

Column	**Description**
`Container ID`	The unique ID of the container. It is a SHA-256.
`Image`	The name of the container image from which this container is instantiated.
`Command`	The command that is used to run the main process in the container.
`Created`	The date and time when the container was created.
`Status`	The status of the container (created, restarting, running, removing, paused, exited, or dead).
`Ports`	The list of container ports that have been mapped to the host.
`Names`	The name assigned to this container (multiple names are possible).

If we want to list not only the currently running containers but all containers that are defined on our system, then we can use the command-line parameter `-a` or `--all` as follows:

```
$ docker container ls -a
```

This will list containers in any state, such as `created`, `running`, or `exited`.

Sometimes, we want to just list the IDs of all containers. For this, we have the parameter `-q`:

```
$ docker container ls -q
```

You might wonder where this is useful. The following command demonstrates where it can be very helpful:

```
$ docker container rm -f $(docker container ls -a -q)
```

Lean back and take a deep breath. Then, try to find out what the preceding command does. Don't read any further until you find the answer or give up.

Right: the preceding command deletes all containers that are currently defined on the system, including the stopped ones. The `rm` command stands for remove, and it will be explained further down.

In the previous section, we used the parameter `-l` in the list command. Try to use Docker help to find out what the `-l` parameter stands for. You can invoke help for the list command as follows:

```
$ docker container ls -h
```

Stopping and starting containers

Sometimes, we want to (temporarily) stop a running container. Let's try this out with the quotes container we used previously. Run the container again with this command:

```
$ docker container run -d --name quotes alpine \
   /bin/sh -c "while :; do wget -qO- https://talaikis.com/api/quotes/
random; printf '\n'; sleep 5; done"
```

Now, if we want to stop this container then we can do so by issuing this command:

```
$ docker container stop quotes
```

When you try to stop the quotes container, you will probably note that it takes a while for this command to be executed. To be precise, it takes about 10 seconds. *Why is this the case?*

Docker sends a Linux `SIGTERM` signal to the main process running inside the container. If the process doesn't react to this signal and terminate itself, Docker waits for 10 seconds and then sends `SIGKILL`, which will kill the process forcefully and terminate the container.

In the preceding command, we have used the name of the container to specify which container we want to stop. But we could also have used the container ID instead.

How do we get the ID of a container? There are several ways of doing so. The manual approach is to list all running containers and find the one that we're looking for in the list. From there, we copy its ID. A more automated way is to use some shell scripting and environment variables. If, for example, we want to get the ID of the quotes container, we can use this expression:

```
$ export CONTAINER_ID=$(docker container ls | grep quotes | awk
'{print $1}')
```

Now, instead of using the container name, we can use the variable `$CONTAINER_ID` in our expression:

```
$ docker container stop $CONTAINER_ID
```

Once we have stopped the container, its status changes to `Exited`.

If a container is stopped, it can be started again using the `docker container start` command. Let's do this with our quotes container. It is good to have it running again, as we'll need it in the subsequent sections of this chapter:

```
$ docker container start quotes
```

Removing containers

When we run the `docker container ls -a` command, we can see quite a few containers that are in `Exited` status. If we don't need these containers anymore, then it is a good thing to remove them from memory, otherwise they unnecessarily occupy precious resources. The command to remove a container is:

```
$ docker container rm <container ID>
```

Another command to remove a container is:

```
$ docker container rm <container name>
```

Try to remove one of your exited containers using its ID.

Sometimes, removing a container will not work as it is still running. If we want to force a removal, no matter what the condition of the container currently is, we can use the command-line parameter `-f` or `--force`.

Inspecting containers

Containers are runtime instances of an image and have a lot of associated data that characterizes their behavior. To get more information about a specific container, we can use the `inspect` command. As usual, we have to provide either the container ID or name to identify the container of which we want to obtain the data. So, let's inspect our sample container:

```
$ docker container inspect quotes
```

The response is a big JSON object full of details. It looks similar to this:

```
    [
        {
            "Id": "c5c1c68c87...",
            "Created": "2017-12-30T11:55:51.223271182Z",
            "Path": "/bin/sh",
            "Args": [
                "-c",
                "while :; do wget -qO- https://talaikis.com/api/
quotes/random; printf '\n'; sleep 5; done"
            ],
            "State": {
                "Status": "running",
                "Running": true,
                ...
            },
            "Image": "sha256:e21c333399e0...",
            ...
            "Mounts": [],
            "Config": {
                "Hostname": "c5c1c68c87dd",
                "Domainname": "",
                ...
            },
            "NetworkSettings": {
                "Bridge": "",
                "SandboxID": "2fd6c43b6fe5...",
                ...
            }
        }
    ]
```

The output has been shortened for readability.

Please take a moment to analyze what you got. You should see information such as:

- The ID of the container
- The creation date and time of the container
- From which image the container is built and so on

Many sections of the output, such as `Mounts` or `NetworkSettings` don't make much sense right now, but we will certainly discuss those in the upcoming chapters of the book. The data you're seeing here is also named the **metadata** of a container. We will be using the `inspect` command quite often in the remainder of the book as a source of information.

Sometimes, we need just a tiny bit of the overall information, and to achieve this, we can either use the **grep tool** or a **filter**. The former method does not always result in the expected answer, so let's look into the latter approach:

```
$ docker container inspect -f "{{json .State}}" quotes | jq
```

The `-f` or `--filter` parameter is used to define the filter. The filter expression itself uses the **Go template** syntax. In this example, we only want to see the state part of the whole output in the JSON format.

To nicely format the output, we pipe the result into the jq tool:

```
{
  "Status": "running",
  "Running": true,
  "Paused": false,
  "Restarting": false,
  "OOMKilled": false,
  "Dead": false,
  "Pid": 6759,
  "ExitCode": 0,
  "Error": "",
  "StartedAt": "2017-12-31T10:31:51.893299997Z",
  "FinishedAt": "0001-01-01T00:00:00Z"
}
```

Exec into a running container

Sometimes, we want to run another process inside an already-running container. A typical reason could be to try to debug a misbehaving container. *How can we do this?* First, we need to know either the ID or the name of the container, and then we can define which process we want to run and how we want it to run. Once again, we use our currently-running quotes container and we run a shell interactively inside it with the following command:

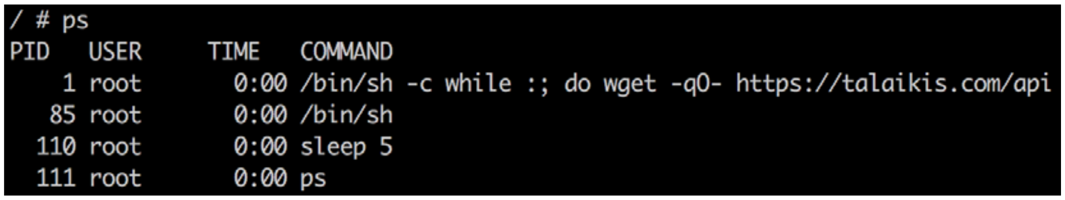
```
$ docker container exec -i -t quotes /bin/sh
```

The flag `-i` signifies that we want to run the additional process interactively, and `-t` tells Docker that we want it to provide us with a TTY (a terminal emulator) for the command. Finally, the process we run is `/bin/sh`.

If we execute the preceding command in our Terminal, then we will be presented with a new prompt. We're now in a shell inside the quotes container. We can easily prove that by, for example, executing the `ps` command, which will list all running processes in the context:

```
# / ps
```

The result should look somewhat similar to this:



```
/ # ps
PID   USER     TIME   COMMAND
    1 root      0:00 /bin/sh -c while :; do wget -qO- https://talaikis.com/api
   85 root      0:00 /bin/sh
  110 root      0:00 sleep 5
  111 root      0:00 ps
```

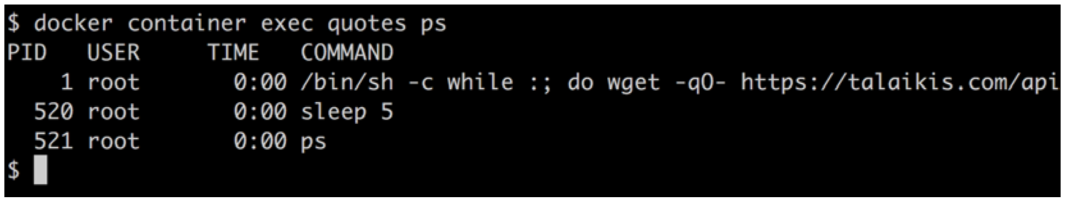
List of Processes running inside the quotes Container

We can clearly see that the process with `PID 1` is the command that we have defined to run inside the quotes container. The process with `PID 1` is also named the main process.

Leave the container by entering `exit` at the prompt. We cannot only execute additional processes interactive in a container. Please consider the following command:

```
$ docker container exec quotes ps
```

The output evidently looks very similar to the preceding output:



```
$ docker container exec quotes ps
PID   USER     TIME   COMMAND
    1 root       0:00 /bin/sh -c while :; do wget -qO- https://talaikis.com/api
  520 root       0:00 sleep 5
  521 root       0:00 ps
$ 
```

List of Processes running inside the quotes Container

We can even run processes as daemon using the flag `-d` and define environment variables using the `-e` flag variables as follows:

```
$ docker container exec -it \
    -e MY_VAR="Hello World" \
    quotes /bin/sh
# / echo $MY_VAR
Hello World
# / exit
```

Attaching to a running container

We can use the `attach` command to attach our Terminal's standard input, output, and error (or any combination of the three) to a running container using the ID or name of the container. Let's do this for our quotes container:

```
$ docker container attach quotes
```

In this case, we will see every five seconds or so a new quote appearing in the output.

To quit the container without stopping or killing it, we can press the key combination *Ctrl* + *P Ctrl*+ *Q*. This detaches us from the container while leaving it running in the background. On the other hand, if we want to detach and stop the container at the same time, we can just press *Ctrl* + *C*.

Let's run another container, this time an Nginx web server:

```
$ docker run -d --name nginx -p 8080:80 nginx:alpine
```

Here, we run the Alpine version of Nginx as a daemon in a container named `nginx`. The `-p 8080:80` command-line parameter opens port `8080` on the host for access to the Nginx web server running inside the container. Don't worry about the syntax here as we will explain this feature in more detail in the *Chapter 7, Single-Host Networking*.

Let's see whether we can access Nginx, using the `curl` tool and running this command:

```
$ curl -4 localhost:8080
```

If all works correctly, you should be greeted by the welcome page of Nginx:

```
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully
installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Now, let's attach our Terminal to the `nginx` container to observe what's happening:

```
$ docker container attach nginx
```

Once you are attached to the container, you first will not see anything. But now open another Terminal, and in this new Terminal window, repeat the `curl` command a few times using the following script:

```
$ for n in {1..10}; do curl -4 localhost:8080; done
```

You should see the logging output of Nginx, which looks similar to this:

```
172.17.0.1 - - [06/Jan/2018:12:20:00 +0000] "GET / HTTP/1.1" 200 612
"-" "curl/7.54.0" "-"
172.17.0.1 - - [06/Jan/2018:12:20:03 +0000] "GET / HTTP/1.1" 200 612
"-" "curl/7.54.0" "-"
172.17.0.1 - - [06/Jan/2018:12:20:05 +0000] "GET / HTTP/1.1" 200 612
"-" "curl/7.54.0" "-"
```

Quit the container by pressing *Ctrl* + *C*. This will detach your Terminal and, at the same time, stop the `nginx` container.

To clean up, remove the `nginx` container with the following command:

```
$ docker container rm nginx
```

Retrieving container logs

It is a best practice for any good application to generate some logging information that developers and operators alike can use to find out what the application is doing at a given time, and whether there are any problems to help pinpoint the root cause of the issue.

When running inside a container, the application should preferably output the log items to STDOUT and STDERR and not into a file. If the logging output is directed to STDOUT and STDERR, then Docker can collect this information and keep it ready for consumption by a user or any other external system.

To access the logs of a given container, we can use the `docker container logs` command. If, for example, we want to retrieve the logs of our `quotes` container, we can use the following expression:

```
$ docker container logs quotes
```

This will retrieve the whole log produced by the application from the very beginning of its existence.

Stop, wait a second—this is not quite true, what I just said, that the full log, from the beginning of the containers existence is available. By default, Docker uses the so-called `json-file` logging driver. This driver stores the logging information in a file. And if there is a file rolling policy defined, then `docker container logs` only retrieves what is in the current active log file and not what is in previous, rolled files that still might or might not be available on the host though.

If we want to only get a few of the latest entries, we can use the `-t` or `--tail` parameter, as follows:

```
$ docker container logs --tail 5 quotes
```

This will retrieve only the last five items the process running inside the container produced.

Sometimes, we want to follow the log that is produced by a container. This is possible when using the parameter `-f` or `--follow`. The following expression will output the last five log items and then follow the log as it is produced by the containerized process:

```
$ docker container logs --tail 5 --follow quotes
```

Logging drivers

Docker includes multiple logging mechanisms to help us get information from running containers. These mechanisms are named **logging drivers**. Which logging driver is used can be configured at the Docker daemon level. The default logging driver is `json-file`. Some of the drivers that are currently supported natively are:

Driver	**Description**
`none`	No log output for the specific container is produced.
`json-file`	This is the default driver. The logging information is stored in files, formatted as JSON.
`journald`	If the journals daemon is running on the host machine, we can use this driver. It forwards logging to the `journald` daemon.
`syslog`	If the `syslog` daemon is running on the host machine, we can configure this driver, which will forward the log messages to the `syslog` daemon.
`gelf`	When using this driver, log messages are written to a **Graylog Extended Log Format** (**GELF**) endpoint. Popular examples of such endpoints are Graylog and Logstash.
`fluentd`	Assuming that the `fluentd` daemon is installed on the host system, this driver writes log messages to it.

[If you change the logging driver, please be aware that the `docker container logs` command is only available for the `json-file` and `journald` drivers.]

Using a container-specific logging driver

We have seen that the logging driver can be set globally in the Docker daemon configuration file. But we can also define the logging driver on a container by container basis. In the following example, we are running a `busybox` container and use the `--log-driver` parameter to configure the `none` logging driver:

```
$ docker container run --name test -it \
    --log-driver none \
    busybox sh -c 'for N in 1 2 3; do echo "Hello $N"; done'
```

We should see the following:

```
Hello 1
Hello 2
Hello 3
```

Now, let's try to get the logs of the preceding container:

```
$ docker container logs test
```

The output is as follows:

```
Error response from daemon: configured logging driver does not support
reading
```

This is to be expected, since the `none` driver does not produce any logging output. Let's clean up and remove the `test` container:

```
$ docker container rm test
```

Advanced topic – changing the default logging driver

Let's change the default logging driver of a Linux host. The easiest way to do this is on a real Linux host. For this purpose, we're going to use Vagrant with an Ubuntu image:

```
$ vagrant init bento/ubuntu-17.04
$ vagrant up
$ vagrant ssh
```

Once inside the Ubuntu VM, we want to edit the Docker daemon configuration file. Navigate to the folder `/etc/docker` and run `vi` as follows:

```
$ vi daemon.json
```

Enter the following content:

```
{
  "Log-driver": "json-log",
  "log-opts": {
    "max-size": "10m",
    "max-file": 3
  }
}
```

Save and exit Vi by first pressing *Esc* and then typing `:w:q` and finally hitting the *Enter* key.

The preceding definition tells the Docker daemon to use the `json-log` driver with a maximum log file size of 10 MB before it is rolled, and the maximum number of log files that can be present on the system is `3` before the oldest file gets purged.

Now we have to send a `SIGHUP` signal to the Docker daemon so that it picks up the changes in the configuration file:

```
$ sudo kill -SIGHUP $(pidof dockerd)
```

Note that the preceding command only reloads the config file and does not restart the daemon.

Anatomy of containers

Many individuals wrongly compare containers to VMs. However, this is a questionable comparison. Containers are not just lightweight VMs. OK then, *what is the correct description of a container?*

Containers are specially encapsulated and secured processes running on the host system.

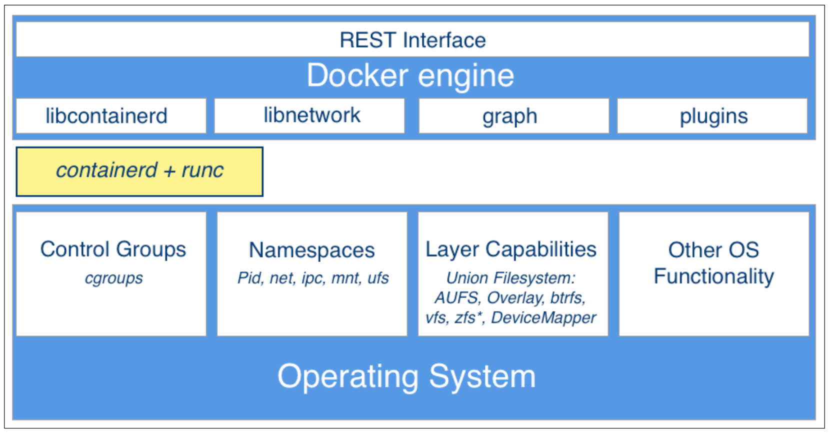
Containers leverage a lot of features and primitives available in the Linux OS. The most important ones are **namespaces** and **cgroups**. All processes running in containers share the same Linux kernel of the underlying host operating system. This is fundamentally different compared with VMs, as each VM contains its own full-blown operating system.

The startup times of a typical container can be measured in milliseconds, while a VM normally needs several seconds to minutes to startup. VMs are meant to be long-living. It is a primary goal of each operations engineer to maximize the uptime of their VMs. Contrary to that, containers are meant to be ephemeral. They come and go in a quick cadence.

Let's first get a high-level overview of the architecture that enables us to run containers.

Architecture

Here, we have an architectural diagram on how this all fits together:



High level architecture of Docker

On the lower part of the the preceding figure, we have the Linux operating system with its cgroups, namespaces, and layer capabilities as well as other functionality that we do not need to explicitly mention here. Then, there is an intermediary layer composed of **containerd** and **runc**. On top of all that now sits the Docker engine. The Docker engine offers a RESTful interface to the outside world that can be accessed by any tool, such as the Docker CLI, Docker for Mac, and Docker for Windows or Kubernetes to just name a few.

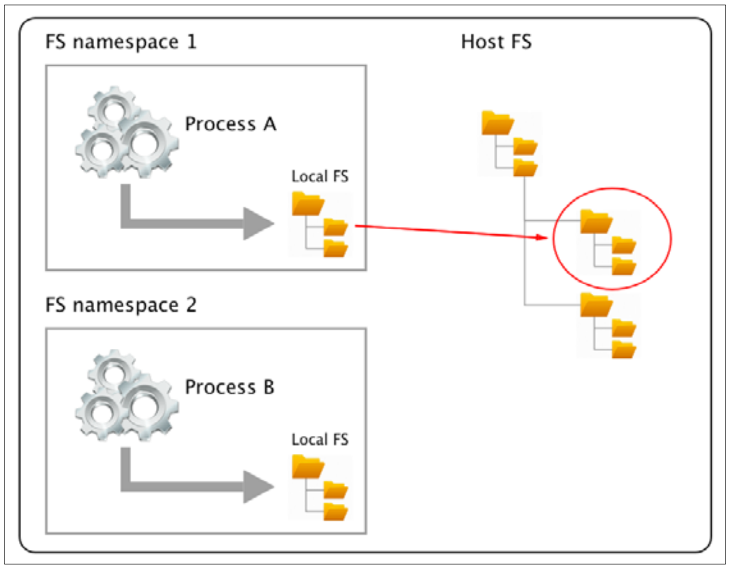
Let's now describe the main building blocks in a bit more detail.

Namespaces

Linux namespaces had been around for years before they were leveraged by Docker for their containers. A namespace is an abstraction of global resources such as filesystems, network access, process tree (also named PID namespace) or the system group IDs, and user IDs. A Linux system is initialized with a single instance of each namespace type. After initialization, additional namespaces can be created or joined.

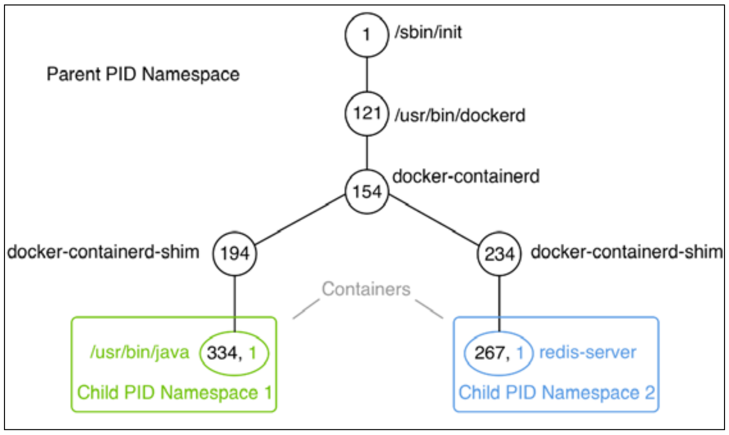
The Linux namespaces originated in 2002 in the 2.4.19 kernel. In kernel version 3.8, user namespaces were introduced and with it, namespaces were ready to be used by containers.

If we wrap a running process, say, in a filesystem namespace, then this process has the illusion that it owns its own complete filesystem. This of course is not true; it is only a virtual FS. From the perspective of the host, the contained process gets a shielded subsection of the overall FS. It is like a filesystem in a filesystem:



The same applies for all the other global resources for which namespaces exist. The user ID namespace is another example. Having a user namespace, we can now define a user `jdoe` many times on the system as long at it is living in its own namespace.

The PID namespace is what keeps processes in one container from seeing or interacting with processes in another container. A process might have the apparent PID **1** inside a container, but if we examine it from the host system, it would have an ordinary PID, say **334**:



Process tree on a Docker host

In a given namespace, we can run one to many processes. That is important when we talk about containers, and we have experienced that already when we executed another process in an already-running container.

Control groups (cgroups)

Linux cgroups are used to limit, manage, and isolate resource usage of collections of processes running on a system. Resources are CPU time, system memory, network bandwidth, or combinations of these resources, and so on.

Engineers at Google have originally implemented this feature starting in 2006. The cgroups functionality was merged into the Linux kernel mainline in kernel version 2.6.24, which was released in January 2008.

Using cgroups, administrators can limit the resources that containers can consume. With this, one can avoid, for example, the classical *noisy neighbor* problem, where a rogue process running in a container consumes all CPU time or reserves massive amounts of RAM and, as such, starves all the other processes running on the host, whether they're containerized or not.

Union filesystem (UnionFS)

The UnionFS forms the backbone of what is known as container images. We will discuss container images in detail in the next chapter. At this time, we want to just understand a bit better what a UnionFS is and how it works. UnionFS is mainly used on Linux and allows files and directories of distinct filesystems to be overlaid and with it form a single coherent file system. In this context, the individual filesystems are called branches. Contents of directories that have the same path within the merged branches will be seen together in a single merged directory, within the new, virtual filesystem. When merging branches, the priority between the branches is specified. In that way, when two branches contain the same file, the one with the higher priority is seen in the final FS.

Container plumbing

The basement on top of which the Docker engine is built; we can also call it the **container plumbing** and is formed by the two component—**runc** and **containerd**.

Originally, Docker was built in a monolithic way and contained all the functionality necessary to run containers. Over time, this became too rigid and Docker started to break out parts of the functionality into their own components. Two important components are runc and containerd.

Runc

Runc is a lightweight, portable container runtime. It provides full support for Linux namespaces as well as native support for all security features available on Linux, such as SELinux, AppArmor, seccomp, and cgroups.

Runc is a tool for spawning and running containers according to the **Open Container Initiative** (**OCI**) specification. It is a formally specified configuration format, governed by the **Open Container Project** (**OCP**) under the auspices of the Linux Foundation.

Containerd

Runc is a low-level implementation of a container runtime; containerd builds on top of it, and adds higher-level features, such as image transfer and storage, container execution, and supervision, as well as network and storage attachments. With this, it manages the complete life cycle of containers. Containerd is the reference implementation of the OCI specifications and is by far the most popular and widely-used container runtime.

Containerd has been donated to and accepted by the CNCF in 2017. There exist alternative implementations of the OCI specification. Some of them are rkt by CoreOS, CRI-O by RedHat, and LXD by Linux Containers. However, containerd at this time is by far the most popular container runtime and is the default runtime of Kubernetes 1.8 or later and the Docker platform.

Summary

In this chapter, you learned how to work with containers that are based on existing images. We showed how to run, stop, start, and remove a container. Then, we inspected the metadata of a container, extracted the logs of it, and learned how to run an arbitrary process in an already-running container. Last but not least, we dug a bit deeper and investigated how containers work and what features of the underlying Linux operating system they leverage.

In the next chapter, you're going to learn what container images are and how we can build and share our own custom images. We're also discussing the best practices commonly used when building custom images, such as minimizing their size and leveraging the image cache. Stay tuned!

Questions

To assess your learning progress please answer the following questions:

1. What are the states of a container?
2. Which command helps us to find out what is currently running on our host?
3. Which command is used to list the IDs of all containers?

Further reading

The following articles give you some more information related to the topics we discussed in this chapter:

- Docker container at `http://dockr.ly/2iLBV2I`
- Getting started with containers at `http://dockr.ly/2gmxKWB`
- Isolate containers with a user namespace at `http://dockr.ly/2gmyKdf`
- Limit container's resources at `http://dockr.ly/2wqN5Nn`