Learn Enough Docker to be Useful

Part 1: The Conceptual Landscape

[[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----b7ba70caeb4b----------------------)](https://towardsdatascience.com/@jeffhale?source=post_page-----b7ba70caeb4b----------------------)

[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----b7ba70caeb4b----------------------)

Follow

[Jan 9](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b7ba70caeb4b?source=post_page-----b7ba70caeb4b----------------------) · 7 min read

Containers are hugely helpful for improving security, reproducibility, and scalability in software development and data science. Their rise is one of the most important trends in technology today.

Docker is a platform to develop, deploy, and run applications inside containers. Docker is essentially synonymous with containerization. If you’re a current or aspiring software developer or data scientist, Docker is in your future.

https://miro.medium.com/max/30/1*EJx9QN4ENSPKZuz51rC39w.png?q=20



Don’t fret if you aren’t yet up to speed — this article will help you understand the conceptual landscape — and you’ll get to make some pizza along the way.

In the next five articles in this series we’ll jump into Docker terms, Dockerfiles, Docker images, Docker commands, and data storage. Part 2 is now live:

**[Learn Enough Docker to be Useful](https://towardsdatascience.com/learn-enough-docker-to-be-useful-1c40ea269fa8?source=post_page-----b7ba70caeb4b----------------------)**

[Part 2: A Delicious Dozen Docker Terms You Need to Know](https://towardsdatascience.com/learn-enough-docker-to-be-useful-1c40ea269fa8?source=post_page-----b7ba70caeb4b----------------------)

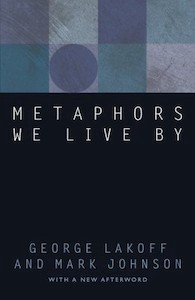
[towardsdatascience.com](https://towardsdatascience.com/learn-enough-docker-to-be-useful-1c40ea269fa8?source=post_page-----b7ba70caeb4b----------------------)

By the end of the series (and with a little practice) you should know enough Docker to be useful 😃!

**Docker Metaphors**

First, I’m going to shed some light on Docker metaphors.

[[https://miro.medium.com/max/20/1*poqn_j2R9xTk940n9wE9Lw.jpeg?q=20](https://www.goodreads.com/book/show/34459.Metaphors_We_Live_By)](https://www.goodreads.com/book/show/34459.Metaphors_We_Live_By)

[[](https://www.goodreads.com/book/show/34459.Metaphors_We_Live_By)](https://www.goodreads.com/book/show/34459.Metaphors_We_Live_By)

[They’re everywhere! Just check out this book.](https://www.goodreads.com/book/show/34459.Metaphors_We_Live_By)

[Google’s second definition for Metaphor](https://www.google.com/search?q=metaphor+definition&oq=metaphor+defini&aqs=chrome.0.0j69i57j0l4.2999j1j4&sourceid=chrome&ie=UTF-8) is what we want:

*a thing regarded as representative or symbolic of something else, especially something abstract.*

Metaphors help us make sense of new things. For example, the metaphor of a physical container helps us quickly grasp the essence of a virtual container.

https://miro.medium.com/max/30/1*ndncU4a3uNsQ_oy2YrNLBA.jpeg?q=20



A physical container

**Container**

Like a physical plastic container, a Docker container:

1. **Holds things**— Something is either inside the container or outside the container.
2. **Is portable**— It can be used on your local machine, your coworker’s machine, or a cloud provider’s servers (e.g. AWS). Sort of like that box of childhood knickknacks you keep moving with you from home to home.
3. **Has clear interfaces for access** — Our physical container has a lid for opening and putting things in and taking things out. Similarly, a Docker container has several mechanisms for interfacing with the outside world. It has ports that can be opened for interacting through the browser. You can configure it to interact with data through the command line.
4. **Can be obtained from a remote location**— You can get another empty plastic container from Amazon.com when you need it. Amazon gets its plastic containers from manufacturers who stamp them out by the thousands from a single mold. In the case of a Docker container, an offsite registry keeps an image, which is like a mold, for your container. Then when you need a container you can make one from the image.

Unlike a virtual Docker container, a new plastic container from Amazon will cost you money and won’t come with a copy of your goods inside. Sorry 💸.

**Living Instance**

A second way you can think of a Docker container is as**an instance of a living thing**. An instance is something that exists in some form. It’s not just code. It’s code that has brought something to life. Like other living things, the instance will eventually die — meaning the container will shut down.

https://miro.medium.com/max/24/1*t-uVUfbywQsDnwQoYAEbgA.jpeg?q=20



An instance of a monster

A Docker container is a Docker image brought to life.

**Software**

In addition to the container metaphor and the living instance metaphor, you can think of a Docker container as **a software program**. After all, it is software. At its most basic level a container is a set of instructions that manipulate other bits.

https://miro.medium.com/max/30/1*0D45gdLlWgvMBu9Xwr0RrA.jpeg?q=20



Containers are code

While a Docker container is running, it generally has programs running inside it. The programs in a container perform actions so your application will do something.

For example, the code in a Docker container might have sent you the content you are reading on this webpage right now. Or it might take your voice command to Amazon Alexa and decode it into instructions another program in a different container will use.

With Docker you can run multiple containers simultaneously on a host machine. And like other software programs, Docker containers can be run, inspected, stopped, and deleted.

**Concepts**

**Virtual Machines**

Virtual machines are the precursors to Docker containers. Virtual machines also isolate an application and its dependencies. However, Docker containers are superior to virtual machines because they take fewer resources, are very portable, and are faster to spin up. Check out [this article](https://medium.freecodecamp.org/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b)for a great discussion of the similarities and differences.

**Docker Image**

I mentioned images above. What’s an image? I’m glad you asked! The meaning of the term *image* in the context of Docker doesn’t map all that well to a physical image.

https://miro.medium.com/max/30/1*Wv9nvbm0XRLSGQ9nqTzpdA.jpeg?q=20



Images

Docker images are more like blueprints, cookie cutters, or molds. Images are the immutable master template that is used to pump out containers that are all exactly alike.

https://miro.medium.com/max/30/1*n53WlDyD9mxVcOu17Rj86Q.jpeg?q=20



Cookie cutters

An image contains the Dockerfile, libraries, and code your application needs to run, all bundled together.

**Dockerfile**

A [Dockerfile](https://docs.docker.com/engine/reference/builder/) is a file with instructions for how Docker should build your image.

The Dockerfile refers to a base image that is used to build the initial image layer. Popular official base images include [python](https://hub.docker.com/_/python/), [ubuntu](https://hub.docker.com/_/ubuntu), and [alpine](https://hub.docker.com/_/alpine).

Additional layers can then be stacked on top of the base image layers, according to the instructions in the Dockerfile. For example, a Dockerfile for a machine learning application could tell Docker to add NumPy, Pandas, and Scikit-learn in an intermediate layer.

Finally, a thin, writable layer is stacked on top of the other layers according to the Dockerfile code. (You understand that a thin layer is small in size because you intuitively understand the *thin* metaphor, right 😃?)

I’ll explore Dockerfiles in more depth in future articles in this series.

**Docker Container**

A Docker image plus the command docker run image\_name creates and starts a container from an image.

**Container Registry**

If you want other people to be able to make containers from your image, you send the image to a container registry. [Docker Hub](https://hub.docker.com/) is the largest registry and the default.

Phew! That’s a lot of pieces. Let’s put this all together in terms of making a pizza.

**Cooking with Docker**

https://miro.medium.com/max/30/1*v6WWacmOsrPYtkGXUu-cbA.jpeg?q=20



Landscape Metaphor

* The recipe is like the *Dockerfile*. It tells you what to do to get to your end goal.
* The ingredients are the *layers*. You’ve got crust, sauce, and cheese for this pizza.

Think of the recipe and the ingredients combined as an all-in-one pizza-making-kit. It’s the *Docker image*.

The recipe (Dockerfile) tells us what we’re going to do. Here’s the plan:

* The crust is preformed and immutable, it’s like a basic ubuntu parent image. It’s the *bottom layer* and gets built first.
* Then you’ll add some cheese. Adding this second layer to the pizza is like *installing an external library*— for example NumPy.
* Then you’ll sprinkle on some basil. The basil is like the *code in a file* that you wrote to run your app.

Alright, let’s get cooking.

https://miro.medium.com/max/30/1*rihuhM7hCvWaJhuw7Hjvzg.jpeg?q=20



Oven

* The oven that bakes the pizza is like the Docker platform. You installed the oven into your house when you moved in so you could make things with it. Similarly, you installed Docker onto your computer so you could cook up containers.
* You start your oven by turning a knob. The docker run image\_namecommand is like your knob — it creates and starts your container.
* The cooked pizza is like a Docker container.
* Eating the pizza is like using your app.

Like making a pizza, making an app in a Docker container takes some work, but at the end you have something great. Enjoy 🍕!

**Wrap**

That’s the conceptual framework. In [Part 2 of this series](https://towardsdatascience.com/learn-enough-docker-to-be-useful-1c40ea269fa8) I clarify some of the terms you’ll see in the Docker ecosystem. Follow me to make sure you don’t miss it!

Hopefully this overview has helped you better understand the Docker landscape. I also hope it has also opened your eyes to the value of metaphors in understanding new technologies.

If you found this helpful please share it on your favorite social media so other people can find it, too. 👏

I write about Python, Docker, data science, and more. If any of that’s of interest to you, read more [here](https://medium.com/@jeffhale) and follow me on Medium. 😄

# Learn Enough Docker to be Useful

## Part 2: A Delicious Dozen Docker Terms You Need to Know

[[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----1c40ea269fa8----------------------)](https://towardsdatascience.com/@jeffhale?source=post_page-----1c40ea269fa8----------------------)

[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----1c40ea269fa8----------------------)

Follow

[Jan 16](https://towardsdatascience.com/learn-enough-docker-to-be-useful-1c40ea269fa8?source=post_page-----1c40ea269fa8----------------------) · 6 min read

In [Part 1 of this series](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b7ba70caeb4b) we explored the conceptual landscape of Docker containers. We discussed the reasons Docker containers are important and several ways to think about them. And we made one into a pizza 🍕. In this article I’ll share a dozen additional terms from the Docker ecosystem that you need to know.

**Docker Ecosystem Terms**

I’ve broken Docker terms into two categories for easier mental model creation: *Essentials* and *Scaling*. Let’s hit the eight essentials first.

**Docker Essentials**

[**Docker Platform**](https://docs.docker.com/engine/docker-overview/#the-docker-platform)isDocker’s software that provides the ability to package and run an application in a container on any Linux server. Docker Platform bundles code files and dependencies. It promotes easy scaling by enabling portability and reproducibility.

https://miro.medium.com/max/30/1*EJx9QN4ENSPKZuz51rC39w.png?q=20



[**Docker Engine**](https://www.docker.com/products/docker-engine)is the client-server application. The Docker company divides the Docker Engine into two products. [*Docker Community Edition (CE)*](https://docs.docker.com/install/) is free and largely based on [open source tools](https://opensource.stackexchange.com/questions/5436/is-docker-still-free-and-open-source). It’s probably what you’ll be using. [*Docker Enterprise*](https://www.docker.com/products/docker-enterprise) comes with additional support, management, and security features. Enterprise is how the Docker firm keeps the lights on.

https://miro.medium.com/max/30/1*FNB0kxJPvZplgsCv5xfpHg.jpeg?q=20



Engine makes things run

[**Docker Client**](https://docs.docker.com/engine/docker-overview/) is the primary way you’ll interact with Docker. When you use the [Docker Command Line Interface (CLI)](https://docs.docker.com/engine/reference/commandline/cli/) you type a command into your terminal that starts with docker. Docker Client then uses the Docker API to send the command to the Docker Daemon.

[[https://miro.medium.com/max/30/0*rgvX6TzfW4HqNpiQ?q=20](https://docs.docker.com/engine/docker-overview/)](https://docs.docker.com/engine/docker-overview/)

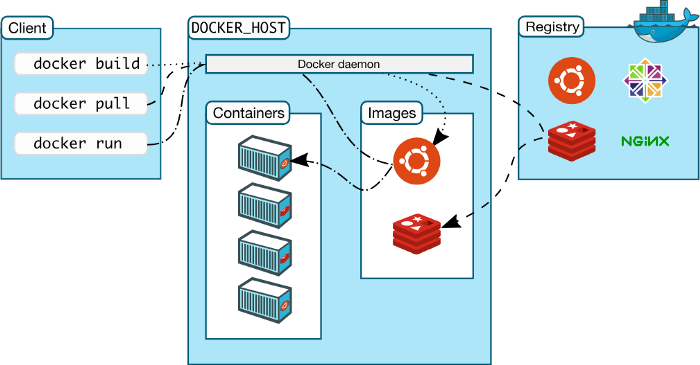
[[](https://docs.docker.com/engine/docker-overview/)](https://docs.docker.com/engine/docker-overview/)

Diagram from [the Docker docs](https://docs.docker.com/engine/docker-overview/)

[**Docker Daemon**](https://docs.docker.com/engine/docker-overview/)is the Docker server that listens for Docker API requests. The Docker Daemon manages images, containers, networks, and volumes.

[**Docker Volumes**](https://docs.docker.com/storage/volumes/) are the best way to store the persistent data that your apps consume and create. We’ll have more to say about Docker Volumes in Part 5 of this series. Follow me to make sure you don’t miss it.

https://miro.medium.com/max/30/1*ADBUW-1PG0_v7guEcwqizg.jpeg?q=20



Volumes

A[**Docker Registry**](https://hub.docker.com/)is the remote location where Docker Images are stored.You push images to a registry and pull images from a registry. You can host your own registry or use a provider’s registry. For example, [AWS](https://aws.amazon.com/ecr/) and [Google Cloud](https://cloud.google.com/container-registry/) have registries.

[**Docker Hub**](https://hub.docker.com/) is the largest registry of Docker images. It’s also the default registry. You can find images and store your own images on Docker Hub for free.

https://miro.medium.com/max/30/1*vNAAZJ34aVfqzX6mhmFDzA.png?q=20



Hubs and spokes

A[**Docker Repository**](https://docs.docker.com/docker-hub/repos/) is a collection of Docker images with the same name and different tags. The *tag* is the identifier for the image.

Usually a repository has different versions of the same image. For example, [*Python*](https://hub.docker.com/_/python)is the name of the most popular official Docker image repository on Docker Hub. *Python:3.7-slim*refers to the version of the image with the *3.7-slim* tag in the Python repository. You can push a repository or a single image to a registry.

Now let’s look at Docker terms related to scaling multiple Docker containers.

**Scaling Docker**

The following four concepts relate to using multiple containers at once.

[**Docker Networking**](https://docs.docker.com/engine/tutorials/networkingcontainers/)allows you to connect Docker containers together. Connected Docker containers could be on the same host or multiple hosts. For more information on Docker networking, see [this post](https://www.oreilly.com/learning/what-is-docker-networking).

https://miro.medium.com/max/25/1*xNFQn4kScZu9IotaUblHWg.png?q=20



[Docker Bridge Network](https://docs.docker.com/engine/tutorials/networkingcontainers/)

[**Docker Compose**](https://docs.docker.com/compose/)is a tool that makes it easier to run apps that require multiple Docker containers. Docker Compose allows you to move commands into a docker-compose.yml file for reuse. The Docker Compose command line interface (cli) makes it easier to interact with your multi-container app. Docker Compose comes free with your installation of Docker.

[**Docker Swarm**](https://docs.docker.com/engine/swarm/) is a product to orchestrate container deployment. The [official Docker tutorial](https://docs.docker.com/get-started/#recap-and-cheat-sheet) has you using Docker Swarm in its fourth section. I would suggest you not spend time on Docker Swarm unless you have a compelling reason to do so.

https://miro.medium.com/max/30/1*T1u345aWrH17H7llzh8QnA.jpeg?q=20



Bee swarm

[**Docker Services**](https://docs.docker.com/get-started/part3/#introduction) are the different pieces of a distributed app. From the [docs](https://docs.docker.com/get-started/part3/#introduction):

*Services are really just “containers in production.” A service only runs one image, but it codifies the way that image runs — what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.*

Docker services allow you to scale containers across multiple Docker Daemons and make Docker Swarms possible.

There you have it: a dozen delicious Docker terms you should know.

**Recap**

Here’s the one line explanation to help you keep these dozen terms straight.

**Basics**

*Platform* — the software that makes Docker containers possible  
*Engine* — client-server app (CE or Enterprise)  
*Client* — handles Docker CLI so you can communicate with the Daemon  
*Daemon*— Docker server that manages key things  
*Volumes* — persistent data storage  
*Registry* — remote image storage  
*Docker Hub*— default and largest Docker Registry  
*Repository* — collection of Docker images, e.g. Alpine

**Scaling**

*Networking* — connect containers together  
*Compose* — time saver for multi-container apps  
*Swarm* — orchestrates container deployment  
*Services* — containers in production

Because we’re keeping with food metaphors, and everyone loves a baker’s dozen, we have one more related term for you: *Kubernetes*.

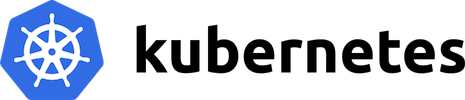
https://miro.medium.com/max/30/1*tMnpIVxRu--tx82F6l5_HA.jpeg?q=20



One more donut with extra icing and sprinkles

[**Kubernetes**](https://kubernetes.io/)automates deployment, scaling, and management of containerized applications. It’s the clear winner in the container orchestration market. Instead of Docker Swarm, use Kubernetes to scale up projects with multiple Docker containers. Kubernetes isn’t an official part of Docker; it’s more like Docker’s BFF.

https://miro.medium.com/max/30/1*3wCo9j0gEklP6kHEOFpthA.png?q=20



I have a whole series on Kubernetes in the works. Kubernetes is pretty awesome.

Now that you know the conceptual landscape and common terms I suggest you try out Docker.

**Baking with Docker**

If you haven’t worked with Docker before, it’s time to get in the kitchen and make something!

Docker runs locally on Linux, Mac, and Windows. If you’re on a Mac or Windows machine, install the latest stable version of Docker Desktop [here](https://www.docker.com/products/docker-desktop). As a bonus, it comes with Kubernetes. If you’re installing Docker elsewhere, go [here](https://docs.docker.com/install/) to find the version you need.

After you have Docker installed, do the first two parts of the [Docker tutorial](https://docs.docker.com/get-started/). Then meet back here for more Docker fun. In the next four parts of this series we’ll dive into Dockerfiles, Docker images, the Docker CLI, and dealing with data. Follow me to make sure you don’t miss the adventure.

Part 3 on Dockerfiles is now available here:

**[Learn Enough Docker to be Useful](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5?source=post_page-----1c40ea269fa8----------------------)**

[Part 3: A Dozen Dandy Dockerfile Instructions](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5?source=post_page-----1c40ea269fa8----------------------)

[towardsdatascience.com](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5?source=post_page-----1c40ea269fa8----------------------)

If you found this article helpful, please share it on Twitter, Facebook, LinkedIn and your favorite forums. Thanks for reading!

Learn Enough Docker to be Useful

Part 3: A Dozen Dandy Dockerfile Instructions

[[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----b0b44222eef5----------------------)](https://towardsdatascience.com/@jeffhale?source=post_page-----b0b44222eef5----------------------)

[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----b0b44222eef5----------------------)

Follow

[Jan 28](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5?source=post_page-----b0b44222eef5----------------------) · 12 min read

This article is all about Dockerfiles. It’s the third installment in a six-part series on Docker. If you haven’t read [Part 1](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b7ba70caeb4b), read it first and see Docker container concepts in a whole new light. 💡 [Part 2](https://towardsdatascience.com/learn-enough-docker-to-be-useful-1c40ea269fa8) is a quick run-through of the Docker ecosystem. In [future articles](https://towardsdatascience.com/slimming-down-your-docker-images-275f0ca9337e), I’ll look at slimming down Docker images, Docker CLI commands, and using data with Docker.

Let’s jump into the dozen Dockerfile instructions to know!

https://miro.medium.com/max/30/1*Sh6i23HzQcvVy_DUHnmrIw.jpeg?q=20



Jump in. True picture ;)

**Docker Images**

Recall that a Docker container is a Docker image brought to life. It’s a self-contained, minimal operating system with application code.

The Docker image is created at build time and the Docker container is created at run time.

The Dockerfile is at the heart of Docker. The Dockerfile tells Docker how to build the image that will be used to make containers.

Each Docker image contains a file named *Dockerfile*with no extension. The Dockerfile is assumed to be in the current working directory when docker build is called to create an image. A different location can be specified with the file flag (-f).

Recall that a container is built from a series of layers. Each layer is read only, except the final container layer that sits on top of the others. The Dockerfile tells Docker which layers to add and in which order to add them.

Each layer is really just a file with the changes since the previous layer. In Unix, pretty much everything is a [file](https://en.wikipedia.org/wiki/Everything_is_a_file).

The base image provides the initial layer(s). A base image is also called a parent image.

When an image is pulled from a remote repository to a local machine only layers that are not already on the local machine are downloaded. Docker is all about saving space and time by reusing existing layers.

https://miro.medium.com/max/30/1*AqNC_3Nefyo-enzgRjTD4w.jpeg?q=20



A base (jumping) image

A Dockerfile instruction is a capitalized word at the start of a line followed by its arguments. Each line in a Dockerfile can contain an instruction. Instructions are processed from top to bottom when an image is built. Instructions look like this:

FROM ubuntu:18.04  
COPY . /app

Only the instructions FROM, RUN, COPY, and ADD create layers in the final image. Other instructions configure things, add metadata, or tell Docker to do something at run time, such as expose a port or run a command.

In this article, I’m assuming you are using a Unix-based Docker image. You can also used Windows-based images, but that’s a slower, less-pleasant, less-common process. So use Unix if you can.

Let’s do a quick once-over of the dozen Dockerfile instructions we’ll explore.

**A Dozen Dockerfile Instructions**

FROM — specifies the base (parent) image.  
LABEL —provides metadata. Good place to include maintainer info.  
ENV — sets a persistent environment variable.  
RUN —runs a command and creates an image layer. Used to install packages into containers.  
COPY — copies files and directories to the container.  
ADD — copies files and directories to the container. Can upack local .tar files.  
CMD — provides a command and arguments for an executing container. Parameters can be overridden. There can be only one CMD.  
WORKDIR — sets the working directory for the instructions that follow.  
ARG — defines a variable to pass to Docker at build-time.  
ENTRYPOINT — provides command and arguments for an executing container. Arguments persist.   
EXPOSE — exposes a port.  
VOLUME — creates a directory mount point to access and store persistent data.

Let’s get to it!

**Instructions and Examples**

A Dockerfile can be as simple as this single line:

FROM ubuntu:18.04

[**FROM**](https://docs.docker.com/engine/reference/builder/#from)

A Dockerfile must start with a FROM instruction or an ARG instruction followed by a FROM instruction.

The FROMkeyword tells Docker to use a base image that matches the provided repository and tag. A base image is also called a [parent image](https://docs.docker.com/develop/develop-images/baseimages/).

In this example, *ubuntu*is the image repository. Ubuntu is the name of an [official Docker repository](https://hub.docker.com/_/ubuntu) that provides a basic version of the popular Ubuntu version of the Linux operating system.

https://miro.medium.com/max/26/1*OiaeQ0JXfdeluoJq-liKYg.png?q=20



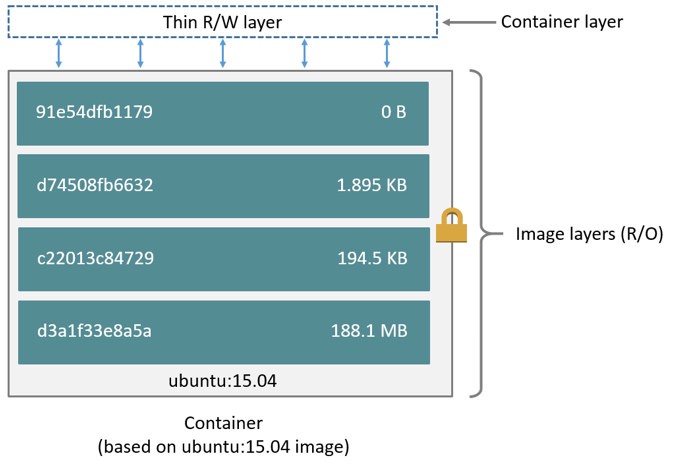
Linux mascot Tux

Notice that this Dockerfile includes a tag for the base image: *18.04* . This tag tells Docker which version of the image in the *ubuntu* repository to pull. If no tag is included, then Docker assumes the *latest*tag*,*by default. To make your intent clear, it’s good practice to specify a base image tag.

When the Dockerfile above is used to build an image locally for the first time, Docker downloads the layers specified in the *ubuntu* image. The layers can be thought of as stacked upon each other. Each layer is a file with the set of differences from the layer before it.

When you create a container, you add a writable layer on top of the read-only layers.

https://miro.medium.com/max/30/1*l8YiwkfvUQsG_uGv9OyJDw.jpeg?q=20



From the [Docker Docs](https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/#images-and-layers)

Docker uses a copy-on-write strategy for efficiency. If a layer exists at a previous level within an image, and another layer needs read access to it, Docker uses the existing file. Nothing needs to be downloaded.

When an image is running, if a layer needs modified by a container, then that file is copied into the top, writeable layer. Check out the Docker docs [here](https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/) to learn more about copy-on-write.

**A More Substantive Dockerfile**

Although our one-line image is concise, it’s also slow, provides little information, and does nothing at container run time. Let’s look at a longer Dockerfile that builds a much smaller size image and executes a script at container run time.

FROM python:3.7.2-alpine3.8  
LABEL maintainer="[jeffmshale@gmail.com](mailto:jeffmshale@gmail.com)"  
ENV ADMIN="jeff"RUN apk update && apk upgrade && apk add bashCOPY . ./appADD <https://raw.githubusercontent.com/discdiver/pachy-vid/master/sample_vids/vid1.mp4> \  
/my\_app\_directoryRUN ["mkdir", "/a\_directory"]CMD ["python", "./my\_script.py"]

Whoa, what’s going on here? Let’s step through it and demystify.

The base image is an official Python image with the tag *3.7.2-alpine3.8*. As you can see from its [source code](https://github.com/docker-library/python/blob/ab8b829cfefdb460ebc17e570332f0479039e918/3.7/alpine3.8/Dockerfile), the image includes Linux, Python and not much else. Alpine images are popular because they are small, fast, and secure. However, Alpine images don’t come with many operating system niceties. You must install such packages yourself, should you need them.

[**LABEL**](https://docs.docker.com/engine/reference/builder/#label)

The next instruction is LABEL. LABEL adds metadata to the image. In this case, it provides the image maintainer’s contact info. Labels don’t slow down builds or take up space and they do provide useful information about the Docker image, so definitely use them. More about LABEL metadata can be found [here](https://docs.docker.com/config/labels-custom-metadata/).

https://miro.medium.com/max/30/1*BxNuK7OE4yohf4ASHPemdQ.jpeg?q=20



[**ENV**](https://docs.docker.com/engine/reference/builder/#env)

ENV sets a persistent environment variable that is available at container run time. In the example above, you could use the ADMIN variable when when your Docker container is created.

ENV is nice for setting constants. If you use a constant several places in your Dockerfile and want to change its value at a later time, you can do so in one location.

https://miro.medium.com/max/30/1*RqSnnTIeR5dcUeUMfVP4jw.jpeg?q=20



ENVironment

With Dockerfiles there are often multiple ways to accomplish the same thing. The best method for your case is a matter of balancing Docker conventions, transparency, and speed. For example, RUN, CMD, and ENTRYPOINT serve different purposes, and can all be used to execute commands.

[**RUN**](https://docs.docker.com/engine/reference/builder/#run)

RUN creates a layer at build-time. Docker commits the state of the image after each RUN.

RUN is often used to install packages into an image*.*In the example above, RUN apk update && apk upgrade tells Docker to update the packages from the base image*.*&& apk add bash tells Docker to install *bash* into the image.

*apk*stands for [Alpine Linux package manager](https://www.cyberciti.biz/faq/10-alpine-linux-apk-command-examples/). If you’re using a Linux base image in a flavor other than Alpine, then you’d install packages with RUN *apt-get* instead of *apk*. *apt* stand for *advanced package tool*. I’ll discuss other ways to install packages in a later example.

https://miro.medium.com/max/30/1*ovB1vggQ-wDamyzj4YFaqQ.jpeg?q=20



RUN

RUN — and its cousins, CMD and ENTRYPOINT — can be used in exec form or shell form. Exec form uses JSON array syntax like so: RUN ["my\_executable", "my\_first\_param1", "my\_second\_param2"].

In the example above, we used shell form in the format RUN apk update && apk upgrade && apk add bash.

Later in our Dockerfile we used the preferred exec form with RUN ["mkdir", "/a\_directory"] to create a directory. Don’t forget to use double quotes for strings with JSON syntax for exec form!

[**COPY**](https://docs.docker.com/engine/reference/builder/#copy)

The COPY . ./appinstruction tells Docker to take the files and folders in your local build context and add them to the Docker image’s current working directory. Copy will create the target directory if it doesn’t exist.

https://miro.medium.com/max/30/1*xHw_3tj8JooiRp-yWZCIyQ.jpeg?q=20



COPY

[**ADD**](https://docs.docker.com/engine/reference/builder/#add)

ADD does the same thing as COPY, but has two more use cases. ADD can be used to move files from a remote URL to a container and ADD can extract local TAR files.

I used ADD in the example above to copy a file from a remote url into the container’s *my\_app\_directory*. The [Docker docs](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/) don’t recommend using remote urls in this manner because you can’t delete the files. Extra files increase the final image size.

The [Docker docs](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#add-or-copy) also suggest using COPY instead of ADD whenever possible for improved clarity. It’s too bad that Docker doesn’t combine ADD and COPY into a single command to reduce the number of Dockerfile instructions to keep straight 😃.

Note that the ADD instruction contains the \ line continuation character. Use it to improve readability by breaking up a long instruction over several lines.

[**CMD**](https://docs.docker.com/engine/reference/builder/#cmd)

CMD provides Docker a command to run when a container is started. It does not commit the result of the command to the image at build time. In the example above, CMD will have the Docker container run the my\_*script.py* file at run time.

https://miro.medium.com/max/30/1*534Bheu2kcpBWl8J4vX4rA.jpeg?q=20



That’s a CMD!

A few other things to know about CMD:

* Only one CMD instruction per Dockerfile. Otherwise all but the final one are ignored.
* CMD can include an executable. If CMD is present without an executable, then an ENTRYPOINT instruction must exist. In that case, both CMD and ENTRYPOINT instructions should be in JSON format.
* Command line arguments to docker run override arguments provided to CMD in the Dockerfile.

**Ready for more?**

Let’s introduce a few more instructions in another example Dockerfile.

FROM python:3.7.2-alpine3.8  
LABEL maintainer="[jeffmshale@gmail.com](mailto:jeffmshale@gmail.com)"# Install dependencies  
RUN apk add --update git# Set current working directory  
WORKDIR /usr/src/my\_app\_directory# Copy code from your local context to the image working directory  
COPY . .# Set default value for a variable  
ARG my\_var=my\_default\_value# Set code to run at container run time  
ENTRYPOINT ["python", "./app/my\_script.py", "my\_var"]# Expose our port to the world  
EXPOSE 8000# Create a volume for data storage  
VOLUME /my\_volume

Note that you can use comments in Dockerfiles. Comments start with #.

Package installation is a primary job of Dockerfiles. As touched on earlier, there are several ways to install packages with RUN.

You can install a package in an Alpine Docker image with *apk. apk*is like *apt-get*in regular Linux builds. For example, packages in a Dockerfile with a base Ubuntu image can be updated and installed like this: RUN apt-get update && apt-get install my\_package.

In addition to *apk* and *apt-get*, Python packages can be installed through [*pip*](https://pypi.org/project/pip/), [*wheel*](https://pythonwheels.com/), and *[conda](https://medium.com/@chadlagore/conda-environments-with-docker-82cdc9d25754)*. Other languages can use various installers.

The underlying layers need to provide the install layer with the the relevant package manger. If you’re having an issue with package installation, make sure the package managers are installed before you try to use them. 😃

You can use RUN with pip and list the packages you want installed directly in your Dockerfile. If you do this concatenate your package installs into a single instruction and break it up with line continuation characters (\). This method provides clarity and fewer layers than multiple RUN instructions.

Alternatively, you can list your package requirements in a file and RUN a package manager on that file. Folks usually name the file *requirements.txt*. I’ll share a recommended pattern to take advantage of build time caching with *requirements.txt*in the next article.

[**WORKDIR**](https://docs.docker.com/v17.09/engine/reference/builder/#workdir)

WORKDIR changes the working directory in the container for the COPY, ADD, RUN, CMD, and ENTRYPOINT instructions that follow it. A few notes:

* It’s preferable to set an absolute path with WORKDIR rather than navigate through the file system with cd commands in the Dockerfile.
* WORKDIR creates the directory automatically if it doesn’t exist.
* You can use multiple WORKDIR instructions. If relative paths are provided, then each WORKDIR instruction changes the current working directory.

https://miro.medium.com/max/30/1*IriKIqsuPjl-A9zqU4NUyw.jpeg?q=20



WORKDIRs of some sort

[**ARG**](https://docs.docker.com/engine/reference/builder/#arg)

ARG defines a variable to pass from the command line to the image at build-time. A default value can be supplied for ARG in the Dockerfile, as it is in the example: ARG my\_var=my\_default\_value.

Unlike ENV variables, ARG variables are not available to running containers. However, you can use ARG values to set a default value for an ENV variable from the command line when you build the image. Then, the ENV variable persists through container run time. Learn more about this technique [here](https://vsupalov.com/docker-build-time-env-values/).

[**ENTRYPOINT**](https://docs.docker.com/engine/reference/builder/#entrypoint)

The ENTRYPOINT instruction also allows you provide a default command and arguments when a container starts. It looks similar to CMD, but ENTRYPOINT parameters are not overwritten if a container is run with command line parameters.

Instead, command line arguments passed to docker run my\_image\_name are appended to the ENTRYPOINT instruction’s arguments. For example, docker run my\_image bash adds the argument *bash* to the end of the ENTRYPOINT instruction’s existing arguments.

https://miro.medium.com/max/30/1*8kfhJgqmW34UYMhDnHUcFA.jpeg?q=20



ENTRYPOINT to somewhere

A Dockerfile should have at least one CMD or ENTRYPOINT instruction.

The [Docker docs](https://docs.docker.com/v17.09/engine/reference/builder/#understand-how-cmd-and-entrypoint-interact) have a few suggestions for choosing between CMD and ENTRYPOINT for your initial container command:

* Favor ENTRYPOINT when you need to run the same command every time.
* Favor ENTRYPOINT when a container will be used as an executable program.
* Favor CMD when you need to provide extra default arguments that could be overwritten from the command line.

In the example above, ENTRYPOINT ["python", "my\_script.py", "my\_var"]has the container run the the python script *my\_script.py*with the argument *my\_var*when the container starts running*. my\_var*could then be used by *my\_script* via [argparse](https://docs.python.org/3/library/argparse.html). Note that *my\_var* has a default value supplied by ARG earlier in the Dockerfile. So if an argument isn’t passed from the command line, then the default argument will be used.

Docker recommends you generally use the exec form of ENTRYPOINT: ENTRYPOINT ["executable", "param1", "param2"]. This form is the one with JSON array syntax.

[EXPOSE](https://docs.docker.com/engine/reference/builder/#expose)

The EXPOSE instruction shows which port is intended to be published to provide access to the running container. EXPOSE does not actually publish the port. Rather, it acts as a documentation between the person who builds the image and the person who runs the container.

https://miro.medium.com/max/23/1*S2QyTx1gOWtsIoXPzyhCPQ.jpeg?q=20



EXPOSEd

Use docker run with the -p flag to publish and map one or more ports at run time. The uppercase -P flag will publish all exposed ports.

[**VOLUME**](https://docs.docker.com/engine/reference/builder/#volume)

VOLUME specifies where your container will store and/or access persistent data. Volumes are the topic of a forthcoming article in this series, so we’ll investigate them then.

https://miro.medium.com/max/26/1*xZT5mao1DZlKu0Zsl9Zc7w.jpeg?q=20



VOLUME

Let’s review the dozen Dockerfile instructions we’ve explored.

**Important Dockerfile Instructions**

FROM — specifies the base (parent) image.  
LABEL —provides metadata. Good place to include maintainer info.  
ENV — sets a persistent environment variable.  
RUN —runs a command and creates an image layer. Used to install packages into containers.  
COPY — copies files and directories to the container.  
ADD — copies files and directories to the container. Can upack local .tar files.  
CMD — provides a command and arguments for an executing container. Parameters can be overridden. There can be only one CMD.  
WORKDIR — sets the working directory for the instructions that follow.  
ARG — defines a variable to pass to Docker at build-time.  
ENTRYPOINT — provides command and arguments for an executing container. Arguments persist.   
EXPOSE — exposes a port.  
VOLUME — creates a directory mount point to access and store persistent data.

Now you know a dozen Dockerfile instructions to make yourself useful! Here’s a bonus bagel: a [cheat sheet](https://kapeli.com/cheat_sheets/Dockerfile.docset/Contents/Resources/Documents/index) with all the Dockerfile instructions. The five commands we didn’t cover are USER, ONBUILD, STOPSIGNAL, SHELL, and HEALTHCHECK. Now you’ve seen their names if you come across them. 😃

https://miro.medium.com/max/30/1*wO-_QeIh-YFXRZeEMrczRw.jpeg?q=20



Bonus bagel

**Wrap**

Dockerfiles are perhaps the key component of Docker to master. I hope this article helped you gain confidence with them. We’ll revisit them in the [next article in this series on slimming down images](https://towardsdatascience.com/slimming-down-your-docker-images-275f0ca9337e). Follow [me](https://medium.com/@jeffhale) to make sure you don’t miss it!

Slimming Down Your Docker Images

Part 4 of Learn Enough Docker to be Useful

[[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----275f0ca9337e----------------------)](https://towardsdatascience.com/@jeffhale?source=post_page-----275f0ca9337e----------------------)

[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----275f0ca9337e----------------------)

Follow

[Jan 31](https://towardsdatascience.com/slimming-down-your-docker-images-275f0ca9337e?source=post_page-----275f0ca9337e----------------------) · 9 min read

In this article you’ll learn how to speed up your Docker build cycles and create lightweight images. Keeping with our food metaphors, we’re going to be eating salad 🥗 as we slim down our Docker images — no more pizza, donuts, and bagels.

In Part 3 of this series we covered a dozen Dockerfile instructions to know. If you missed it, check out the article here:

**[Learn Enough Docker to be Useful](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5?source=post_page-----275f0ca9337e----------------------)**

[Part 3: A Dozen Dandy Dockerfile Instructions](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5?source=post_page-----275f0ca9337e----------------------)

[towardsdatascience.com](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5?source=post_page-----275f0ca9337e----------------------)

Here’s the cheatsheet.

FROM — specifies the base (parent) image.  
LABEL —provides metadata. Good place to include maintainer info.  
ENV — sets a persistent environment variable.  
RUN —runs a command and creates an image layer. Used to install packages into containers.  
COPY — copies files and directories to the container.  
ADD — copies files and directories to the container. Can upack local .tar files.  
CMD — provides a command and arguments for an executing container. Parameters can be overridden. There can be only one CMD.  
WORKDIR — sets the working directory for the instructions that follow.  
ARG — defines a variable to pass to Docker at build-time.  
ENTRYPOINT — provides command and arguments for an executing container. Arguments persist.   
EXPOSE — exposes a port.  
VOLUME — creates a directory mount point to access and store persistent data.

Let’s now look at how we can fashion our Dockerfiles to save time when developing images and pulling containers.

**Caching**

One of Docker’s strengths is that it provides caching to help you more quickly iterate your image builds.

When building an image, Docker steps through the instructions in your Dockerfile, executing each in order. As each instruction is examined, Docker looks for an existing intermediate image in its cache that it can reuse instead of creating a new (duplicate) intermediate image.

If cache is invalidated, the instruction that invalidated it and all subsequent Dockerfile instructions generate new intermediate images. As soon as the cache is invalidated, that’s it for the rest of the instructions in the Dockerfile.

So starting at the top of the Dockerfile, if the base image is already in the cache it is reused. That’s a hit. Otherwise, the cache is invalidated.

https://miro.medium.com/max/20/1*FOCF2hBIRuQ0nB8o-VJCxA.jpeg?q=20



Also a hit

Then the next instruction is compared against all child images in the cache derived from that base image. Each cached intermediate image is compared to see if the instruction finds a cache hit. If it’s a cache miss, the cache is invalidated. The same process is repeated until the end of the Dockerfile is reached.

Most new instructions are simply compared with those in the intermediate images. If there’s a match, then the cached copy is used.

For example, when a RUN pip install -r requirements.txt instruction is found in a Dockerfile, Docker searches for the same instruction in its locally cached intermediate images. The content of the old and new *requirements.txt* files are not compared.

This behavior can be problematic if you update your *requirements.txt* file with new packages and use RUN pip install and want to rerun the package installation with the new package names. I’ll show a few solutions in a moment.

Unlike other Docker instructions, ADD and COPY instructions do require Docker to look at the contents of the file(s) to determine if there is a cache hit. The checksum of the referenced file is compared against the checksum in the existing intermediate images. If the file contents or metadata have changed, then the cache is invalidated.

Here are a few tips for using caching effectively.

* Caching can be turned off by passing --no-cache=True with docker build.
* If you are going to be making changes to instructions, then every layer that follows will be rebuilt frequently. To take advantage of caching, put instructions that are likely to change as low as you can in your Dockerfile.
* Chain RUN apt-get update and apt-get install commands to avoid cache miss issues.
* If you’re using a package installer such as pip with a *requirements.txt* file, then follow a model like the one below to make sure you don’t receive a stale intermediate image with the old packages listed in *requirements.txt*.

COPY requirements.txt /tmp/  
RUN pip install -r /tmp/requirements.txt  
COPY . /tmp/

Those are the suggestions for using Docker build caching effectively. If you have others please share them in the comments or on Twitter [@discdiver](http://twitter.com/discdiver).

**Size Reduction**

Docker images can get large. You want to keep them small so they can pulled quickly and use few resources. Let’s skinny down your images!

https://miro.medium.com/max/30/1*RFi86DbJQy39RjNQrQiD6A.jpeg?q=20



Go for a salad instead of a bagel

An Alpine base image is a full Linux distribution without much else. It is usually under 5 MB to download, but it requires you to spend more time writing the code for the dependencies you need to build a working app.

https://miro.medium.com/max/30/1*NnmWS0yPVhS_5nKW8jzSLg.jpeg?q=20



Alpine comes from Alps

If you need Python in your container, the Python Alpine build is a nice compromise. It contains Linux and Python and you supply most everything else.

An image I built with the latest Python Alpine build with a *print(“hello world”)* script weighs in at 78.5 MB. Here’s the Dockerfile:

FROM python:3.7.2-alpine3.8  
COPY . /app  
ENTRYPOINT [“python”, “./app/my\_script.py”, “my\_var”]

On the Docker Hub website the base image is listed as 29 MB. When the child image is built it downloads and installs Python, making it grow larger.

Besides using Alpine base images, another method for reducing the size of your images is using multistage builds. This technique also adds complexity to your Dockerfile.

**Multistage Builds**

https://miro.medium.com/max/30/1*DiW3ESMm1z2jAln_I62UAA.jpeg?q=20



https://miro.medium.com/max/30/1*8Ww_aVrssjTltt6f3EFQrQ.jpeg?q=20



One stage + another stage = multistage

Multistage builds use multiple FROM instructions. You can selectively copy files, called build artifacts, from one stage to another. You can leave behind anything you don’t want in the final image. This method can reduce your overall image size.

Each FROM instruction

* begins a new stage of the build.
* leaves behind any state created in prior stages.
* can use a different base.

Here’s a modified example of a multistage build from the [Docker docs](https://docs.docker.com/develop/develop-images/multistage-build/):

FROM golang:1.7.3 AS build  
WORKDIR /go/src/github.com/alexellis/href-counter/  
RUN go get -d -v golang.org/x/net/html   
COPY app.go .  
RUN CGO\_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .FROM alpine:latest   
RUN apk --no-cache add ca-certificates  
WORKDIR /root/  
COPY --from=build /go/src/github.com/alexellis/href-counter/app .  
CMD ["./app"]

Note that we name the first stage by appending a name to the FROM instruction to name. The named stage is then be referred to in theCOPY --from= instruction later in the Dockerfile.

Multistage builds make sense in some cases where you’ll be making lots of containers in production. Multistage builds can help you squeeze every last ounce (gram if you think in metric) out of your image size. However, sometimes multistage builds add more complexity that can make images harder to maintain, so you probably won’t use them in most builds. See further discussion of the tradeoffs [here](https://blog.realkinetic.com/building-minimal-docker-containers-for-python-applications-37d0272c52f3) and advanced patterns [here](https://medium.com/@tonistiigi/advanced-multi-stage-build-patterns-6f741b852fae).

In contrast, everyone should use a .dockerignore file to help keep their Docker images skinny.

**.dockerignore**

*.dockerignore* files are something you should know about as a person who knows enough Docker to be d̶a̶n̶g̶e̶r̶o̶u̶s̶ useful.

.dockerignore is similar to .gitignore. It’s a file with a list of patterns for Docker to match with file names and exclude when making an image.

https://miro.medium.com/max/30/1*yzTSAJgy7dM9qolu8qx_TA.jpeg?q=20



Just .dockerignore it

Put your .dockerignore file in the same folder as your Dockerfile and the rest of your build context.

When you run docker build to create an image, Docker checks for a .dockerignore file. If one is found, it then goes through the file line by line and uses Go’s [filepath.Match rules](https://golang.org/pkg/path/filepath/" \l "Match) — and a few of [Docker’s own rules](https://docs.docker.com/v17.09/engine/reference/builder/#dockerignore-file) — to match file names for exclusion. Think Unix-style glob patterns, not regular expressions.

So \*.jpg will exclude files with a *.jpg* extension. And videos will exclude the videos folder and its contents.

You can explain what you’re doing in your .dockerignore with comments that start with a#.

Using .dockerignore to exclude files you don’t need from your Docker image is a good idea. .dockerignore can:

* help you keep your secrets from being revealed. No one wants passwords in their images.
* reduce image size. Fewer files means smaller, faster images.
* reduce build cache invalidation. If logs or other files are changing and your image is having its cache invalidated because of it, that’s slowing down your build cycle.

Those are the reasons to use a .dockerignore file. Check out the [docs](https://docs.docker.com/v17.09/engine/reference/builder/#dockerignore-file) for more details.

**Size Inspection**

Let’s look at how to find the size of Docker images and containers from the command line.

* To view the approximate size of a running container, you can use the command docker container ls -s.
* Running docker image ls shows the sizes of your images.
* To see the size of the intermediate images that make up your image use docker image history my\_image:my\_tag.
* Running docker image inspect my\_image:tag will show you many things about your image, including the sizes of each layer. Layers are subtly different than the images that make up an entire image. But you can think of them as the same for most purposes. Check out [this great article](https://windsock.io/explaining-docker-image-ids/)by Nigel Brown if you want to dig into layer and intermediate image intricacies.
* Installing and using the [dive](https://github.com/wagoodman/dive) package makes it easy to see into your layer contents.

I updated the above section Feb. 8, 2019 to use management command names. In the next part of this series we’ll dive further into common Docker commands. Follow me to make sure you don’t miss it.

Now let’s look at a few best practices to slim things down.

**Eight Best Practices to Reduce Image Sizes & Build Times**

1. Use an official base image whenever possible. Official images are updated regularly and are more secure than un-official images.  
2. Use variations of Alpine images when possible to keep your images lightweight.  
3. If using apt, combine RUN apt-get update with apt-get install in the same instruction. Then chain multiple packages in that instruction. List the packages in alphabetical order over multiple lines with the \ character. For example:

RUN apt-get update && apt-get install -y \  
 package-one \  
 package-two   
 && rm -rf /var/lib/apt/lists/\*

This method reduces the number of layers to be built and keeps things nice and tidy.   
4. Include && rm -rf /var/lib/apt/lists/\* at the end of the RUN instruction to clean up the apt cache so it isn’t stored in the layer. See more in the [Docker Docks](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/). Thanks to [Vijay Raghavan Aravamudhan](https://towardsdatascience.com/u/6c50ea34d16f?source=post_page-----275f0ca9337e----------------------) for this suggestion. Updated Feb. 4, 2019.  
5. Use caching wisely by putting instructions likely to change lower in your Dockerfile.  
6. Use a .dockerignore file to keep unwanted and unnecessary files out of your image.  
7. Check out [dive](https://github.com/wagoodman/dive) — a very cool tool for inspecting your Docker image layers and helping you trim the fat.   
8. Don’t install packages you don’t need. Duh! But common.

**Wrap**

Now you know how to make Docker images that build quickly, download quickly, and don’t take up much space. As with eating healthy, knowing is half the battle. Enjoy your veggies!🥗

https://miro.medium.com/max/30/1*Stn2tBUIPslHrfaFsRAZDQ.jpeg?q=20



Healthy and yummy

In the [next article in this series](https://towardsdatascience.com/15-docker-commands-you-should-know-970ea5203421), I dig into essential Docker commands. Follow [me](https://medium.com/@jeffhale) to make sure you don’t miss it.

15 Docker Commands You Should Know

Part 5 of *Learn Enough Docker to be Useful*

[[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----970ea5203421----------------------)](https://towardsdatascience.com/@jeffhale?source=post_page-----970ea5203421----------------------)

[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----970ea5203421----------------------)

Follow

[Feb 5](https://towardsdatascience.com/15-docker-commands-you-should-know-970ea5203421?source=post_page-----970ea5203421----------------------) · 10 min read

In this article we’ll look at 15 Docker CLI commands you should know. If you haven’t yet, check out the rest of this series on [Docker concepts](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b7ba70caeb4b), [the ecosystem](https://towardsdatascience.com/learn-enough-docker-to-be-useful-1c40ea269fa8), [Dockerfiles](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5), and [keeping your images slim](https://towardsdatascience.com/slimming-down-your-docker-images-275f0ca9337e). In [Part 6](https://towardsdatascience.com/pump-up-the-volumes-data-in-docker-a21950a8cd8) we’ll explore data with Docker. I’ve got a series on Kubernetes in the works too, so follow me to make sure you don’t miss the fun!

There are about a billion Docker commands (give or take a billion). The [Docker docs](https://docs.docker.com/engine/reference/commandline/cli/) are extensive, but overwhelming when you’re just getting started. In this article I’ll highlight the key commands for running vanilla Docker.

https://miro.medium.com/max/30/1*e91PqDA0DJI9kgwaXfxngg.jpeg?q=20



Fruit theme

At risk of taking the food metaphor thread running through these articles too far, let’s use a fruit theme. Veggies provided sustenance in the [article on slimming down our images](https://towardsdatascience.com/slimming-down-your-docker-images-275f0ca9337e). Now tasty fruits will give us nutrients as we learn our key Docker commands.

**Overview**

Recall that a Docker image is made of a Dockerfile + any necessary dependencies. Also recall that a Docker container is a Docker image brought to life. To work with Docker commands, you first need to know whether you’re dealing with an image or a container.

* **A Docker image either exists or it doesn’t.**
* **A Docker container either exists or it doesn’t.**
* **A Docker container that exists is either running or it isn’t.**

Once you know what you’re working with you can find the right command for the job.

**Commmand Commonalities**

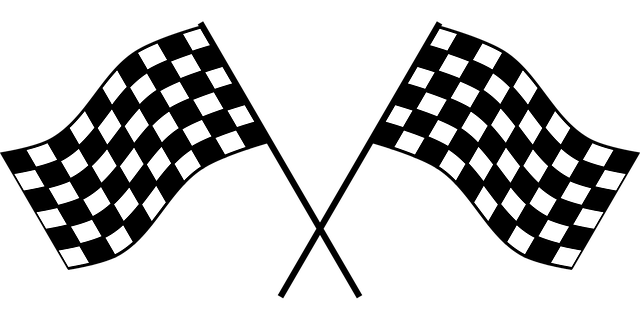
Here are a few things to know about Docker commands:

* Docker CLI management commands start with docker, then a space, then the management category, then a space, and then the command. For example, docker container stop stops a container.
* A command referring to a specific container or image requires the name or id of that container or image.

For example, docker container run my\_app is the command to build and run the container named *my\_app*. I’ll use the name my\_container to refer to a generic container throughout the examples. Same goes for my\_image, my\_tag, etc.

I’ll provide the command alone and then with common flags, if applicable. A flag with two dashes in front is the full name of the flag. A flag with one dash is a shortcut for the full flag name. For example, -p is short for the --port flag.

https://miro.medium.com/max/30/1*GXCbJUG6G2EgcUAA4oxX2Q.png?q=20



Flags provide options to commands

The goal is to help these commands and flags stick in your memory and for this guide to serve as a reference. This guide is current for Linux and Docker Engine Version 18.09.1 and API [version 1.39](https://docs.docker.com/engine/api/version-history/).

First, we’ll look at commands for containers and then we’ll look at commands for images. Volumes will be covered in the next article. Here’s the list of 15 commands to know — plus 3 bonus commands!

**Containers**

Use docker container my\_command

create — Create a container from an image.   
start — Start an existing container.   
run — Create a new container and start it.   
ls — List runningcontainers.   
inspect — See lots of info about a container.  
logs — Print logs.   
stop — Gracefully stop running container.   
kill —Stop main process in container abruptly.   
rm— Delete a stopped container.

**Images**

Use docker image my\_command

build — Build an image.  
push — Push an image to a remote registry.  
ls — List images.   
history — See intermediate image info.  
inspect — See lots of info about an image, including the layers.   
rm — Delete an image.

**Misc**

docker version — List info about your Docker Client and Server versions.  
docker login — Log in to a Docker registry.  
docker system prune — Delete all unused containers, unused networks, and dangling images.

**Containers**

**Container Beginnings**

The terms create, start, and run all have similar semantics in everyday life. But each is a separate Docker command that creates and/or starts a container. Let’s look at creating a container first.

**docker container create my\_repo/my\_image:my\_tag** — Create a container from an image.

I’ll shorten my\_repo/my\_image:my\_tagto my\_image for the rest of the article.

There are [a lot of possible flags](https://docs.docker.com/engine/reference/commandline/create/#parent-command) you could pass to create.

**docker container create -a STDIN my\_image**

-a is short for --attach. Attach the container to STDIN, STDOUT or STDERR.

Now that we’ve created a container let’s start it.

**docker container start my\_container** — Start an existing container.

Note that the container can be referred to by either the container’s ID or the container’s name.

**docker container start my\_container**

https://miro.medium.com/max/30/1*hdLjLfbTWcXQIjgy2D2t0A.jpeg?q=20



Start

Now that you know how to create and start a container, let’s turn to what’s probably the most common Docker command. It combines both create and start into one command: run.

**docker container run my\_image**—Create a new container and start it. It also has [a lot of options](https://docs.docker.com/engine/reference/commandline/run/#parent-command). Let’s look at a few.

**docker container run -i -t -p 1000:8000 --rm my\_image**

-i is short for --interactive. Keep STDIN open even if unattached.

-tis short for--tty. Allocates a pseudo [terminal](http://en.wikipedia.org/wiki/Pseudo_terminal) that connects your terminal with the container’s STDIN and STDOUT.

You need to specify both -i and -t to then interact with the container through your terminal shell.

-p is short for --port. The port is the interface with the outside world.1000:8000 maps the Docker port 8000 to port 1000 on your machine. If you had an app that output something to the browser you could then navigate your browser to localhost:1000 and see it.

--rm Automatically delete the container when it stops running.

Let’s look at some more examples of run.

**docker container run -it my\_image my\_command**

sh is a command you could specify at run time.sh will start a shell session inside your container that you can interact with through your terminal. shis preferable to bash for Alpine images because Alpine images don’t come with bash installed. Type exit to end the interactive shell session.

Notice that we combined -i and -t into -it.

**docker container run -d my\_image**

-d is short for --detach. Run the container in the background. Allows you to use the terminal for other commands while your container runs.

**Checking Container Status**

If you have running Docker containers and want to find out which one to interact with, then you need to list them.

**docker container ls** — List runningcontainers. Also provides useful information about the containers.

**docker container ls -a -s**

-a is short for -all. List all containers (not just running ones).

-s is short for --size. List the size for each container.

**docker container inspect my\_container** — See lots of info about a container.

**docker container logs my\_container** — Print a container’s logs.

https://miro.medium.com/max/30/1*icdqfkCNhA8mhJ82ctt54Q.jpeg?q=20



Logs. Not sure how virtual logs are related. Maybe via reams of paper?

**Container Endings**

Sometimes you need to stop a running container.

**docker container stop my\_container** — Stop one or more running containers gracefully. Gives a default of 10 seconds before container shutdown to finish any processes.

Or if you are impatient:

**docker container kill my\_container** — Stop one or more running containers abruptly. It’s like pulling the plug on the TV. Prefer stop in most situations.

d**ocker container kill $(docker ps -q)** — Kill all running containers.

https://miro.medium.com/max/30/1*z04O3uNyI_so53j0hqIjhg.jpeg?q=20



docker kill cockroach

Then you delete the container with:

**docker container rm my\_container** — Delete one or more containers.

**docker container rm $(docker ps -a -q)** — Delete all containers that are not running.

Those are the eight essential commands for Docker containers.

To recap, you first create a container. Then, you start the container. Or combine those steps with docker run my\_container. Then, your app runs. Yippee!

Then, you stop a container with docker stop my\_container. Eventually you delete the container with docker rm my\_container.

Now, let’s turn to the magical container-producing molds called images.

**Images**

Here are seven commands for working with Docker images.

**Developing Images**

**docker image build -t my\_repo/my\_image:my\_tag .** — Build a Docker image named *my\_image* from the Dockerfile located at the specified path or URL.

-t is short for tag. Tells docker to tag the image with the provided tag. In this case *my\_tag .*

The . (period) at the end of the command tells Docker to build the image according to the Dockerfile in the current working directory.

https://miro.medium.com/max/30/1*pzypY32qru9mOB-qOb4Evw.jpeg?q=20



Build it

Once you have an image built you want to push it to a remote registry so it can be shared and pulled down as needed. Assuming you want to use [Docker Hub](https://hub.docker.com/), go there in your browser and create an account. It’s free. 😄

This next command isn’t an image command, but it’s useful to see here, so I’ll mention it.

**docker login** — Log in to a Docker registry. Enter your username and password when prompted.

https://miro.medium.com/max/30/1*ZgMR-_PYuu3ghNyPzLHrGA.jpeg?q=20



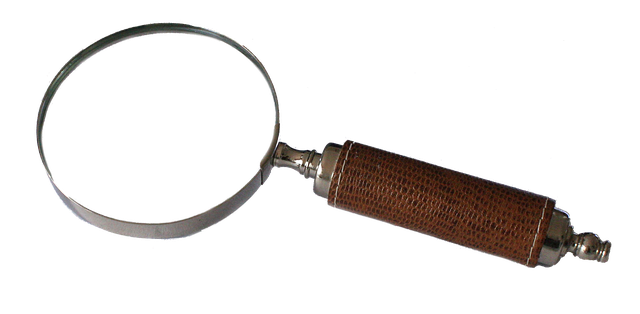
Push

**docker image push my\_repo/my\_image:my\_tag** — Push an image to a registry.

Once you have some images you might want to inspect them.

**Inspecting Images**

https://miro.medium.com/max/30/1*bldenADzHHFGZa8e3XVXUA.png?q=20



Inspection time

**docker image ls** — List your images. Shows you the size of each image, too.

**docker image history my\_image** — Display an image’s intermediate images with sizes and how they were created.

**docker image inspect my\_image** — Show lots of details about your image, including the layers that make up the image.

Sometimes you’ll need to clean up your images.

**Removing Images**

**docker image rm my\_image** — Delete the specified image. If the image is stored in a remote repository, the image will still be available there.

**docker image rm $(docker images -a -q)** — Delete all images. Careful with this one! Note that images that have been pushed to a remote registry will be preserved — that’s one of the benefits of registries. 😃

Now you know most essential Docker image-related commands. We’ll cover data-related commands in the next article.

https://miro.medium.com/max/30/1*oWepBSWQY6jQ4AndB5F5CA.jpeg?q=20



Commands are like fruit — nutritious and delicious. Err. Yeah.

**Misc**

**docker version** — List info about your Docker Client and Server versions.

**docker login**— Log in to a Docker registry. Enter your username and password when prompted.

**docker system prune** makes an appearance in the next article. Readers on [Twitter](https://twitter.com/Docker/status/1093233051906134016) and [Reddit](https://www.reddit.com/r/docker/comments/anh1se/15_docker_commands_you_should_know_part_5_of/) suggested that it would be good to add to this list. I agree, so I’m adding it.

**docker system prune** —Delete all unused containers, unused networks, and dangling images.

**docker system prune -a --volumes**

-a is short for --all. Delete unused images, [not just dangling ones](https://stackoverflow.com/a/45143234/4590385).

--volumes Remove unused volumes. We’ll talk more about volumes in the next article.

**UPDATE Feb. 7, 2019: Management Commands**

In CLI 1.13 Docker introduced management command names that are logically grouped and consistently named. The old commands still work, but the new ones make it easier to get started with Docker. The original version of this article listed the old names. I’ve updated the article to use the management command names based on reader suggestions. Note that this change only introduces two command name changes — in most cases it just means adding container or image to the command. A mapping of the commands is [here](http://blog.arungupta.me/docker-1-13-management-commands/).

**Wrap**

If you are just getting started with Docker, these are the three most important commands:

**docker container run my\_image** — Create a new container and start it. You’ll probably want some flags here.

**docker image build -t my\_repo/my\_image:my\_tag .** — Build an image.

**docker image push my\_repo/my\_image:my\_tag** — Push an image to a remote registry.

Here’s the larger list of essential Docker commands:

**Containers**

Use docker container my\_command

create — Create a container from an image.   
start — Start an existing container.   
run — Create a new container and start it.   
ls — List runningcontainers.   
inspect — See lots of info about a container.  
logs — Print logs.   
stop — Gracefully stop running container.   
kill —Stop main process in container abruptly.   
rm— Delete a stopped container.

**Images**

Use docker image my\_command

build — Build an image.  
push — Push an image to a remote registry.  
ls — List images.   
history — See intermediate image info.  
inspect — See lots of info about an image, including the layers.   
rm — Delete an image.

**Misc**

docker version — List info about your Docker Client and Server versions.  
docker login — Log in to a Docker registry.  
docker system prune — Delete all unused containers, unused networks, and dangling images.

To view the CLI reference when using Docker just enter the command docker in the command line. You can see the Docker docs [here](https://docs.docker.com/engine/reference/commandline/cli/).

Now you can really build things with Docker! As my daughter might say in emoji: 🍒 🥝 🍊 🍋 🍉 🍏 🍎 🍇. Which I think translates to “Cool!” So go forth and play with Docker!

If you missed the earlier articles in this series, check them out. Here’s the first one:

**[Learn Enough Docker to be Useful](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b7ba70caeb4b?source=post_page-----970ea5203421----------------------)**

[Part 1: The Conceptual Landscape](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b7ba70caeb4b?source=post_page-----970ea5203421----------------------)

[towardsdatascience.com](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b7ba70caeb4b?source=post_page-----970ea5203421----------------------)

In the [final article in this series](https://towardsdatascience.com/pump-up-the-volumes-data-in-docker-a21950a8cd8) we’ll spice things up with a discussion of data in Docker. Follow me to make sure you don’t miss it!

Pump up the Volumes: Data in Docker

Part 6 of Learn Enough Docker to be Useful

[[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----a21950a8cd8----------------------)](https://towardsdatascience.com/@jeffhale?source=post_page-----a21950a8cd8----------------------)

[Jeff Hale](https://towardsdatascience.com/@jeffhale?source=post_page-----a21950a8cd8----------------------)

Follow

[Feb 11](https://towardsdatascience.com/pump-up-the-volumes-data-in-docker-a21950a8cd8?source=post_page-----a21950a8cd8----------------------) · 6 min read

This article is about using data with Docker. In it, we’ll focus on Docker volumes. Check out the previous articles in the series if you haven’t yet. We covered Docker [concepts](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b7ba70caeb4b), the [ecosystem](https://towardsdatascience.com/learn-enough-docker-to-be-useful-1c40ea269fa8), [Dockerfiles](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5), [slimming down images](https://towardsdatascience.com/slimming-down-your-docker-images-275f0ca9337e), and [popular commands](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5).

Pushing the food metaphor running through these articles to the breaking point, let’s compare data in Docker to spices. Just as there are many spices in the world, there are many ways to save data with Docker.

https://miro.medium.com/max/30/1*EJx9QN4ENSPKZuz51rC39w.png?q=20



Quick FYI: this guide is current for Docker Engine Version 18.09.1 and API [version 1.39](https://docs.docker.com/engine/api/version-history/).

Data in Docker can either be temporary or persistent. Let’s check out temporary data first.

**Temporary Data**

Data can be kept temporarily inside a Docker container in two ways.

By default, files created by an application inside a container are stored in the writable layer of the container. You don’t have to set anything up. This is the quick and dirty way. Just save a file and go about your business. However, when you container ceases to exist, so will your data.

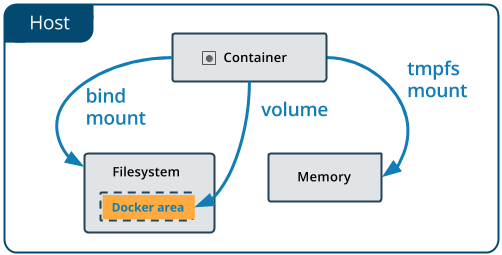
You have another option if you want better performance for saving temporary data with Docker. If you don’t need your data to persist beyond the life of the container, a *tmpfs* mount is a temporary mount that uses the host’s memory. A tmpfs mount has the benefit of faster read and write operations.

Many times you will want your data to exist even after the container is long gone. You need to persist your data.

**Persistent Data**

There are two ways to persist data beyond the life of the container. One way is to *bind mount* a file system to the container. With a bind mount, processes outside Docker also can modify the data.

https://miro.medium.com/max/30/1*9X1xxsG5K9kXAmg_5EM9bg.png?q=20



From the [Docker Docs](https://docs.docker.com/storage/volumes/)

Bind mounts are difficult to back up, migrate, or share with other Containers. Volumes are a better way to persist data.

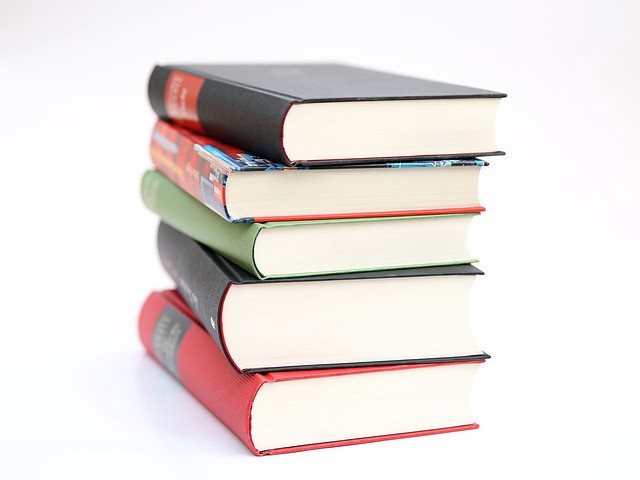
**Volumes**

A Volume is a a file system that lives on a host machine outside of any container. Volumes are created and managed by Docker. Volumes are:

* persistent
* free-floating filesystems, separate from any one container
* sharable with other containers
* efficient for input and output
* able to be hosted on remote cloud providers
* encryptable
* nameable
* able to have their content pre-populated by a container
* handy for testing

That’s a lot of useful functionality! Now let’s look at how you make a Volume.

https://miro.medium.com/max/30/1*dt0MgRXGQAueVUyGXFyUjA.jpeg?q=20



Volumes

**Creating Volumes**

Volumes can be created via a Dockerfile or an API request.

Here’s a Dockerfile instruction that creates a volume at run time:

VOLUME /my\_volume

Then, when the container is created, Docker will create the volume with any data that already exists at the specified location. Note that if you create a volume using a Dockerfile, you still need to declare the mountpoint for the volume at run time.

You can also create a volume in a Dockerfile using JSON array formatting. See [this earlier article](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5) in this series for more on Dockerfiles.

Volumes also can be instantiated at run time from the command line.

**Volume CLI Commands**

**Create**

You can create a stand-alone volume with docker volume create —-name my\_volume.

**Inspect**

List Docker volumes with docker volume ls.

Volumes can be inspected with docker volume inspect my\_volume.

**Remove**

Then you can delete the volume with docker volume rm my\_volume.

Dangling volumes are volumes not used by a container. You can remove all dangling volumes with docker volume prune. Docker will warn you and ask for confirmation before deletion.

If the volume is associated with any containers, you cannot remove it until the containers are deleted. Even then, Docker sometimes doesn’t realize that the containers are gone. If this occurs, you can use docker system prune to clean up all your Docker resources. Then you should should be able to delete the volume.

https://miro.medium.com/max/30/1*-ORM_SLKftoJjOSnhgHLVw.jpeg?q=20



Where your data might be stored

**Working with --mount vs. --volume**

You will often use flags to refer to your volumes. For example, to create a volume at the same time you create a container use the following:

docker container run --mount source=my\_volume, target=/container/path/for/volume my\_image

In the old days (i.e. pre-2017) 😏the --volume flag was popular. Originally, the -v or --volume flag was used for standalone containers and the --mount flag was used with Docker Swarms. However, beginning with Docker 17.06, you can use --mount in all cases.

The syntax for --mount is a bit more verbose, but it’s preferred over --volume for several reasons. --mount is the only way you can work with services or specify volume driver options. It’s also simpler to use.

You’ll see a lot of -v’s in existing code. Beware that the format for the options is different for --mount and --volume. You often can’t just replace a -v in your existing code with a --mount and be done with it.

The biggest difference is that the -v syntax combines all the options together in one field, while the --mount syntax separates them. Let’s see --mount in action!

https://miro.medium.com/max/30/1*B3qV4FtVSBmhp9vS-m7ReA.jpeg?q=20



Easy enough to mount

**--mount**— options are key-value pairs. Each pair is formatted like this: key=value, with a comma between one pair and the next. Common options:

* type — mount type. Options are [bind](https://docs.docker.com/storage/bind-mounts/), [volume](https://docs.docker.com/storage/volumes/), or [tmpfs](https://docs.docker.com/storage/tmpfs/). We’re all about the volume.
* source — source of the mount. For named volumes, this is the name of the volume. For unnamed volumes, this option is omitted. The key can be shortened to src.
* destination — the path where the file or directory is mounted in the container. The key can be shortened to dst or target.
* readonly —mounts the volume as [read-only](https://docs.docker.com/storage/volumes/#use-a-read-only-volume). Optional. Takes no value.

Here’s an example with lots of options:

docker run --mount type=volume,source=volume\_name,destination=/path/in/container,readonly my\_image

https://miro.medium.com/max/30/1*oegIeUqU2Lcs2I1KYlUjSQ.jpeg?q=20



Volumes are like spices — they make most things better. 🥘

**Wrap**

**Recap of Key Volume Commands**

* docker volume create
* docker volume ls
* docker volume inspect
* docker volume rm
* docker volume prune

Common options for the --mount flag in docker run --mount my\_options my\_image:

* type=volume
* source=volume\_name
* destination=/path/in/container
* readonly

Now that you’ve familiarized yourself with data storage in Docker let’s look at possible next steps for your Docker journey.

https://miro.medium.com/max/30/1*8FK19jnIkA4HwZC6ip2ICQ.jpeg?q=20



Next steps

If you haven’t read the articles in this series on Docker [concepts](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b7ba70caeb4b), the [Docker ecosystem](https://towardsdatascience.com/learn-enough-docker-to-be-useful-1c40ea269fa8), [Dockerfiles](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5), [slim images](https://towardsdatascience.com/slimming-down-your-docker-images-275f0ca9337e), and [commands,](https://towardsdatascience.com/learn-enough-docker-to-be-useful-b0b44222eef5) check those out.

If you’re looking for another article on Docker concepts to help cement your understanding, check out [Preethi Kasireddy](https://towardsdatascience.com/u/d446dafbe292?source=post_page-----a21950a8cd8----------------------)’s great article [here](https://medium.freecodecamp.org/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b).

If you want to go deeper, check out Nigel Poulton’s book *[Docker Deep Dive](https://www.amazon.com/Docker-Deep-Dive-Nigel-Poulton-ebook/dp/B01LXWQUFF)(*make sure to get the most recent version).

If you want to do a lot of building while you learn, check out James Turnbull’s [The Docker Book](https://dockerbook.com/).

I hope you found this series to be a helpful intro to Docker. If you did, please share it with others on your favorite forums or social media channels so your friends can find it, too!

I’ve started a series on orchestrating containers with Kubernetes you can read [here](https://towardsdatascience.com/key-kubernetes-concepts-62939f4bc08e?source=friends_link&sk=d46386ce56c00701dbf41aa8d308a14d). I write about articles about Python, data science, AI, and other tech topics. Check them out follow [me](https://medium.com/@jeffhale) if you’re into that stuff.