Containers

March 14, 2022

Brae Webb

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Containers

Software Architecture

March 14, 2022

Brae Webb

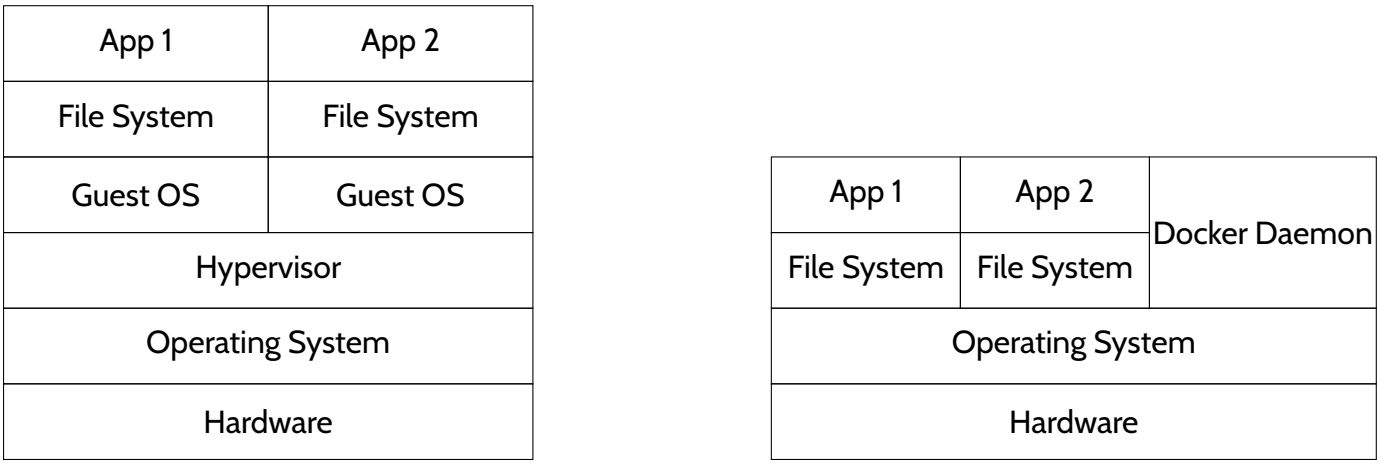
> I don't care if it works on your machine! We are not shipping your machine!
>
> — Vidiu Platon

1 Introduction

As developers, we often find ourselves relying on some magical tools and technologies. Version control, operating systems, databases, and containers, to name a few. Containers, and specifically, Docker, are magical tools which have gained wide-spread industry adoption. Over the past decade Docker has enabled some fanciful architectures and developer workflows. Docker is the proposed solution to the age-old programmer proverb, "it works on my machine!". However, before we subscribe to that belief, let's see how Docker actually works to learn what it is, and what it is not.

2 History and Fundamentals

Relative to other tools in the magical suite, Docker is new on the scene. Docker was first made available publicly in 2013 at PyCon.[1] Pitched to deliver the isolation benefits of Virtual Machines (VMs) while maintaining efficient execution. Virtual machines themselves are a much older invention, dating back to the 1960s, around the time that UQ was having it's first computer installed.[2] The concept of a virtual machine, unlike its implementation, is straight-forward — use software to simulate hardware. From there, one can install an operating system on the simulated hardware and start using a completely isolated, completely new computer without any additional hardware. Of course, this process puts great strain on the real hardware and as such, VMs are deployed sparingly.



(a) Two virtual machines running on a host

(b) Two containers running on a host

Figure 1: Comparison of virtual machines and containers

[1] https://www.youtube.com/watch?v=wW9CAH9nSLs
[2] https://www.youtube.com/watch?v=DB1Y4GrfrZk

Unlike virtual machines, containers do not run on virtual hardware, instead, containers run on the operating system of the host machine. The obvious advantage of this is containers run much more efficiently than virtual machines. Containers however, manage to remain isolated and it is at this point that we should explain how Docker actually works. Docker is built upon two individually fascinating technologies; namespaces, and layered filesystems.

2.1 Namespaces

The first technology, namespaces, is built into the Linux kernel. Linux namespaces were first introduced into the kernel in 2002, inspired by the concept introduced by the Plan 9[3] operating system from Bell Labs in the 1980s. Namespaces enable the partitioning and thus, isolation, of various concepts managed and maintained by an operating system. Initially namespaces were implemented to allow isolated filesystems (the so-called 'mount' namespace). Eventually, as namespaces were expanded to include process isolation, network isolation, and user isolation, the ability to mimic an entirely new isolated machine was realised; containers were born.[4]

Namespaces provide a mechanism to create an isolated environment within your current operating system. The creation of such an isolated environment with namespaces has been made quite easy — you can create an isolated namespace with just a series of bash commands. Niklas Dzösch's talk 'Docker without Docker', uses just 84 lines of Go code (which Docker itself is written in), to create, well, Docker without Docker.[5] But namespaces are just the first technology which enables Docker. How do you prepopulate these isolated environments with everything you need to run a program? To see what is in the isolated environment, we would run `ls` which, oh, requires the `ls` binary. Furthermore, to even consider running a command we need a shell to type it in, so, oh, we should also have a `bash` binary. And so on until, oh, finally, we have a Linux kernel at least![6]

2.2 Layered Filesystem

A core principle of Unix operating systems is that everything is a file. Naturally then, if we want to start using an isolated environment, all we need to do is copy in all the files which together form an operating system, right? Well, yes, kind of. In principle this is all you need do but this would hardly enable the popularity Docker enjoys today.

Say that you want to send your coworker a Docker container which has nginx (a tool for routing web traffic) setup in just the way you need to pass incoming traffic to various components of your application. Let's assume that you have setup nginx in Ubuntu. All you would need to do is zip up all the files which compose the Ubuntu operating system (an impressively small 188MB), then all the files installed by nginx (about 55MB) and finally all the configuration files which you have modified, somewhere in the order of 1000 bytes or 1 KB. In total you are sending your coworker about 243MBs worth of data, less than a gigabyte, so they are not too upset.

Now once we have finished developing our application and we are ready to package it up and send it to the world. Rather than trying to support every known operating system, we bundle all our services in Docker containers, one for nginx, one for mysql, one for our web application, etc, etc. If your applications containers are as popular as nginx, this means one *billion* downloads of your container. At a size of 243MBs, you have contributed 243 *petabytes* to the world's collective bandwidth usage, and that is just your nginx container.

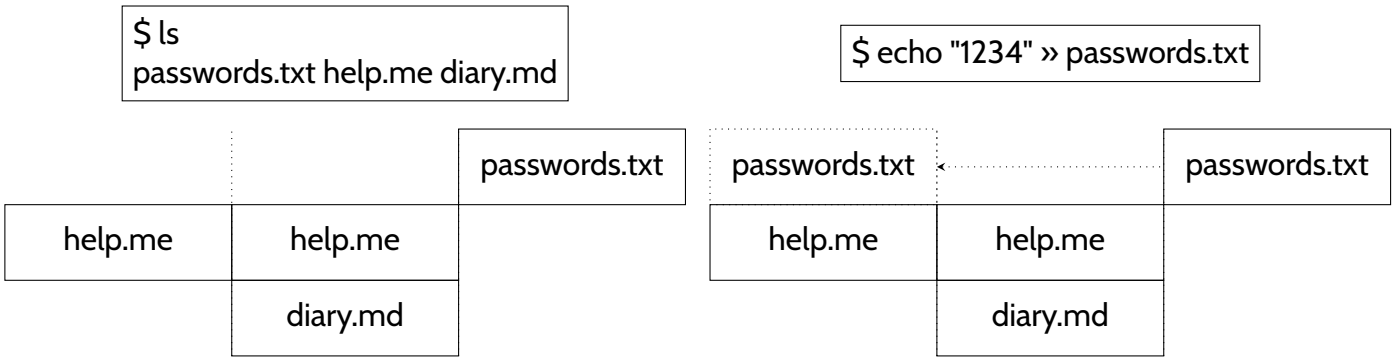
[3] https://p9f.org/

[4] Of course, containers in a rudamentry form existed before introduction to the linux kernel but we have to start somewhere. https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016

[5] https://www.youtube.com/watch?v=7H_eF6hvWg

[6] You might be asking yourself, wait but I have a Windows operating system and I can still run Docker, what gives? The answer, ironically enough, is a virtual machine!

Docker's success over other containerisation applications comes from the way it avoids this looming data disaster. A concept known as the layered, or overlayed, or the stacked filesystem solves the problem. First proposed in the early 1990s, layered filesystems enable layers of immutable files to be stacked below a top-level writable system. This has the effect of producing a seemingly mutable (or writable) filesystem while never actually modifying the collection of immutable files. Whenever a immutable file is 'written to', a copy of the file is made and all modifications are written to this file. When reading or listing files, the filesystem overlays the layers of immutable files together with the layer of mutable files. This gives the appearance of one homogeneous filesystem.



(a) Read operation in a layered filesystem

(b) Write operation in a layered filesystem

Figure 2: Read and write operations in a layered filesystem. The leftmost column in each diagram represents the most recent 'writable' layer.

Docker uses this technique to reduce the amount of duplicated files. If you have Docker containers which run nginx, MySQL, and Python but all containers run on Ubuntu, then your computer will only need to store one copy of the Ybuntu filesystem. Each container will store the changes to the base operating system required to install each application and project that layer over the immutable Ubuntu filesystem.

2.3 Summary

While Docker itself only came out in 2013, the two primary technologies which it is composed of, namespaces and the layered filesystem, were around since the early 1990s. Docker combines these technologies to enable applications to run in an isolated environment which can be efficiently replicated across different host computers. The reproducablity of Docker containers and the fact that they are so light weight makes them an ideal target for two important aspects of modern development; developers simulating production environments locally, and duplicating production machines to scale for large loads of traffic.

3 Docker FROM scratch

Now that we understand the fundamentals of how Docker works, let's start building our very first Docker container. To follow along, you will need to have Docker installed on your computer[7] and have access to basic Unix command line applications.[8]

To start with, we will write a `Dockerfile` which builds a Docker image without an operating system and just prints 'Hello World' [1]. The Docker 'code' is written in a `Dockerfile` which is then 'compiled' into a Docker image and finally run as a Docker container. The first command in your `Dockerfile` should always be `FROM`. This command tells Docker what immutable filesystem we want to start from, often, this will be your chosen operating environment. In this exercise, since we do not want an operating system, we start with `FROM scratch`, a no-op instruction that tells Docker to start with an empty filesystem.

[7] https://docs.docker.com/get-docker/

[8] For windows users, we recommend Windows Subsystem for Linux.

Let's get something in this container. For this, we will use the `COPY` command which copies files from the host machine (your computer) into the container. For now, we will write `COPY hello-world /`, which says to copy a file from the current directory named `hello-world` into the root directory (`/`) of the container. We do not yet have a `hello-world` file but we can worry about that later. Finally, we use the `CMD` command to tell the container what to do when it is run. This should be the command which starts your application.

```
FROM scratch
COPY hello-world /
CMD ["/hello-world"]
```

Next, we will need a minimal hello world program. Unfortunately, we will have to use C here as better programming languages have a bit too much overhead. For everyone who has done CSSE2310, this should be painfully familiar, create a `main` function, print hello world, with a new line, and return 0.

```
#include <stdio.h>

int main() {
    printf ("Hello World\n");
    return 0;
}
```

Let's try running this container. Try to guess if this is going to work. Why? Why not?
First, the hello world program needs to be compiled into a binary file.

```
    >> gcc -o hello-world hello-world.c
    >> ls
    Dockerfile hello-world hello-world.c
```

Next we will use the `Dockerfile` to build a Docker image and run that image. Images are stored centrally for a user account so to identify the image, we need to tag it when it is built, we will use 'hello'.

```
    >> docker build --tag hello .
    >> docker run hello
    standard_init_linux.go:228: exec user process caused: no such file or
       directory
```

Unless this is Docker's unique way of saying hello world, something has gone terribly wrong. Here we are illustrating the power of Docker isolation as well as the difficulty of not having an operating system. This very simple hello world program still relies on other libraries in the operating system, libraries which are not available in the empty Docker container. `ldd` tells us exactly what `hello-world` depends on. The hello world program can be statically compiled so that it does not depend on any libraries [2].

```
>> ldd hello-world
    linux-vdso.so.1 (0x00007ffc51db1000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fcff8553000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fcff8759000)
>> gcc -o hello-world hello-world.c -static
>> ldd hello-world
    not a dynamic executable
>> docker build --tag hello .
>> docker run hello
Hello World
```

Now we have a Docker image built from scratch, without an operating system, which can say 'Hello World'!

> If you are interested in exploring in more depth, try using the '`docker image inspect hello`' and '`docker history hello`' commands.

References

[1] C. Xu, "Docker: From scratch." https://codeburst.io/docker-from-scratch-2a84552470c8, July 2020.

[2] henszey, "Smallest x86 ELF hello world." http://timelessname.com/elfbin/.