What Docker really is

W. Adam Koszek

Koszek ORG

wojciech@koszek.com

2022-10-05T17:47:10Z

It felt strange when Docker first came out, since I couldn't really understand what it is. It is an interesting feeling when people market and brand a new technology in a way which makes it obscure and hard to understand. So below you have my explanation of Docker, the way I would like to see it couple of months ago.

Docker in one paragraph

Docker is a manager for isolated buckets of software called containers. I gives you a way to say: "My software is based on Ubuntu; it must have these 4 packages installed, and these 5 commands must be executed to make it run continually". These steps will be done at "build" time, during which your ~1GB image will be created. From this image containers with your software can be started on many OSes.

More details.

Build is basically a nice term for downloading all the junk like binaries, software libraries and data-files, stitching them all together through the command line tools and making sure that whatever is in this "bundle" is exactly what you requested and what you need. This image is specified in `Dockerfile`.

The image is a filesystem snapshot and has a name. For example, `wkoszek/base` is a name of the image where I keep basic software for simple projects. Or `ruby:2.2.4` is an image with Ruby `2.2.4`. You can then take this build image and export it to the server. Later, other people can download it too, and start their containers with your software. Container is the instance of the image. Basically you can make many active running containers out of the image if you want. To compare it: think of the program on a disk, and a process running in the memory. And some programs you can start a many copies of . Docker image is similar.

Virtualized or not?

Now Docker is branded as a virtualization platform. This is partially true. It's more of a separation manager. When you run on Linux and you want to start a container, Docker tools will talk to the kernel and tell it to create a separate "jail" for new programs about to be started. You can make this jail have limited CPU time, limited memory or just simply have no limits. Afterwards you can start a program in a jail and this will make this program be isolated. If someone were to take over the program from the jail, they'd be able to destroy only things within a jail, but not the main system.

So Docker runs on Linux and is basically based on Linux features.

What happens when you run Windows or Mac? This is where it gets tricky. Docker was branded as multi-platform solution, able to run the same software, anywhere, with no changes.

How to make your Linux-based image run on Mac?

That's simple. You run Linux VM on Mac, and within this VM you run Docker. Docker tools which run on OSX are smart enough to talk to this VM first. And in the VM everything is as explained above. It's pretty ... primitive when you think about it, but it works.

Once again: Docker is only multi-platform when running Linux on your native platform is viable. In fact the old Docker on OSX required running VirtualBox in the background. New Docker Beta runs Linux in a Apple-branded virtualization framework, and Docker inside of it. I haven't looked at how Microsoft Windows is handled, but I know they have their virtualization features too, so I suspect it's equivalent.

Let me add to that: whatever you're running inside of Docker are real Linux programs compiled for your computer architecture. If you have 32-bit ARM CPU like Raspberry PI, and you want to grab a ready-to-use image, you must select the right image type. Otherwise you'll attempt to fetch x64 image and it won't work.

How to start many containers?

The idea of containers shows its real benefits if you deploy a system with many elements. For example you can think of a modern website as a software with a powerful database (SQL), fast memory storage (Redis/Memcached), engine (Rails/NodeJS), HTTP server etc. And it's beneficial to have it all loosely coupled and separated, so that you can test/mock individual elements without impacting others. Containers help with that, because they keep all these packages in isolated jails, and let them communicate only when you let them.

So in the above example one could put SQL database in one container and all memory storage, HTTP storage and Web engine in their own separate containers.

And you could say: I'm OK with SQL database talking to a Web engine, but not to the memory storage. And to do this, you may use Docker Compose.

Docker Compose explains the whole system in a file called `docker-compose.yml`. This is where you can say that Web engine needs a memory storage and a database, and Compose will start things for you in a correct order.

Summary

That's pretty much it. Intentionally I kept you away from technical details, since Docker documentation does a good job at it.

Let me know if you've found this article useful and whether screencast demonstrating this topic step-by-step could be beneficial