Summary of the book Docker Deep Dive (Nigel Poulton)

Edition February 2018

April 28, 2020

The structure of this document is not the same as the one in the book. The installation of Docker, as well as the chapters "Tools for the enterprise" and "Enterprise-grade features" are not covered. Only Linux examples are given (no Windows), and they can all be reproduced with Docker installed on a host, or on Play with Docker (https://labs.play-with-docker.com/). The summary is organized as follows:

Contents

1. Important concepts and definitions

The past

Applications run businesses, and most of them run on servers. In the past, we could only run **1 application per server**. So, new application = buy a new server. However, nobody knew the performance requirements of the new app -> they bought big servers, which ended operating at 5-10% of their potential capacity. It was a waste of resources.

Virtual machines

Then came the VMs, which allowed running **multiple applications on the same server**. It was a huge improvement, but there was still a **problem**: every VM requires its own dedicated OS and consume **resources** (CPU, RAM and storage). Also, every OS needs a license, patching and monitoring. Finally, VMs are slow to boot and portability isn't great (migrating and moving VM workloads between hypervisors and cloudplatforms isn't easy. A hypervisor is a virtualization platform that allows multiple OS to work on the same physical machine). Nowadays, VMs are still very common.

Containers

Containers are a trendy virtualization technology that **address the shortcomings of the VM model**. A container is similar to a VM, but every container does not require its own OS. All containers in a single host share a single OS, freeing up huge amounts of system resources and reducing licensing and monitoring costs. Unlike VMs, containers are fast to start and ultra-portable: it's easy to move container workloads from a laptop to the cloud, and then to VMs or bare metal in a data center. There are Linux and Windows containers. It is vital to understand that a running container shares the kernel of the host machine it is running on -> a containerized app designed to run on a host with a Windows kernel will not run on a Linux host. But in practice, it is actually possible to run Linux containers on Windows machines.

Images

An image is an **object that contains an OS filesystem and an application** (it's like a VM template). Every container is started from an image. So, an image is in fact a stopped container. We can either create our own image, or use an existing one to start a container (see for example https://hub.docker.com/). Getting images onto the docker host is called "pulling".

Docker

Containers have existed for a long time, but were complex and out of reach of most organizations. Docker is the company that **made containers simple**, both for Linux and Windows. It basically creates and manages containers. Note that it is possible to run Linux containers on Mac using Docker for Mac. With Docker, running an application is as simple as pulling an image and running a container. No need to worry about setup, dependencies, or config... It just works. Nowadays, Docker is a very popular container runtime but there are many others. Docker consists of **two major components: the Docker client** and **the Docker engine** (or "daemon"). Once installed, the client-server connection can be tested with the command "docker version".

Kubernetes

It is an open-source project from Google that has become the leading **orchestrator of containerized apps**. An orchestrator is a software that helps deploying and managing containers. Kubernetes (k8s) is a higher-level platform than docker, and uses this latter as its default container runtime (it's Docker that starts and stops containers, pulls images, etc.). It is possible to swap out Docker for another container runtime. In the future, k8s might use containerd instead of Docker by default.

DevOps

The term *DevOps* refers to a movement that aims to unite software development (Dev) and IT operations (Ops). Concretely, it consists of a set of practices that allow to shorten the development life cycle and provide continuous delivery. In a DevOps approach, we try to automate and monitor every step of the creation of a software from the development, integration, tests, delivery and deployment, exploitation and maintenance of infrastructures. DevOps is an **agile approach** (unlike the waterfall approach), in which the product is developped with the customer.

---

2. Introduction

Let's introduce Docker in the DevOps perspective.

The Ops perspective

We will download an image, start a new container, log in to this container, run a command inside of it, and finally destroy it. The easiest way to reproduce the following commands is on Play with Docker (https://labs.play-with-docker.com/).

Let's start by pulling an image:

```
# docker image pull ubuntu:latest
```

Note that "docker pull ubuntu:latest" also works. We can then see the image with the following command:

```
# docker image ls
```

We can refer to the image with its ID (typing the first few characters of the ID is enough for Docker to figure out what image we're referring to). This image contains enough of an OS, code and dependencies to run the application it is designed for. In this case, the image has a stripped-down version of the Ubuntu Linux filesystem.

Now that we have an image, we can launch a container from it. The syntax is *docker container run <image:version> <application/process>* (we must specifiy an application or process):

```
# docker container run -it ubuntu:latest /bin/bash
```

This command changes the shell prompt: this is because we are now in a Bash Shell on Ubuntu. Here, *docker container run* tells the Docker daemon (server) to start a new container. The flags *-it* tell Docker to make the container interactive and to attach our current shell to the container's terminal. The *<image:version>* is the latest version of Ubuntu, and the *<application>* we're running is /bin/bash.

In the container, we can list the running processes:

```
# ps -elf
```

There are 2 processes: */bin/bash* and *ps -elf*. However, this latter stopped existing by the time the output was printed. In reality, there is only */bin/bash* running (it is optimal, nothing else is using resources).

We can exit the container with *Ctr-PQ* (it detaches our shell from the terminal of the container) without killing it. This brings us back to our Docker host's terminal, and the container is still running in the background. Now, if we check again the running processes with *ps -elf*, we see there are more than in the container.

We didn't kill the container, and we can see it with the following command:

```
# docker container ls
```

The container is still running, and we can re-attach to it. The syntax is *docker container exec <options> <container name/ID> <command/app>*:

```
# docker container exec -it determined_nash bash
```

Note that *determined_nash* was the name of my container. let's get out of the container with *Ctrl-PQ* and kill it:

```
# docker container stop determined_nash
# docker container rm determined_nash
# docker container ls -a
```

The flag *-a* in the last command lists all the containers, even the ones that were stopped. It is possible to kill a container in one command, but it is considered a good practice to proceed in two steps. In short, this gives a chance to the application/process to stop properly.

The Dev perspective

This time, we **focus** more on the **application**: we will clone a nodejs web app from Github, inspect its Dockerfile, containerize it, and run it as a container. The linux app can be cloned from https://github.com/nigelpoulton/psweb.git:

```
# git clone https://github.com/nigelpoulton/psweb.git
```

We can then change directory (with *cd*) into the cloned repo's directory and list its contents:

```
# cd psweb
# ls -al
```

There is a *Dockerfile*, which is a document describing how to build an app into a Docker image. We can see the content of the file with the *cat* command:

```
# cat Dockerfile
```

Each line of this file represents an instruction that is used to build an image. In this case, the instructions are:
FROM alpine
LABEL maintainer="nigelpoulton@hotmail.com"
RUN apk add –update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT [“node”, “./app.js”]

So, we use the **Dockerfile** to **build** a Docker **image**:

```
# docker image build -t test:latest .
```

This commands creates a new image called *test:latest*, and we can make sure it was created with

```
# docker image ls
```

We now have an image with the app inside. We can start a container from this image and test the app:

```
# docker container run -d \
# --name web1 \
# --publish 8080:8080 \
# test:latest
```

To see the result (on Play With Docker), click on the button *OPEN PORT* above the terminal and type 8080 (otherwise open a browser and navigate to localhost:8080). The process of building an image from code and then building a container from this image is called *containerizing* an app.

---

3. Technical

The docker engine (or server)

Past architecture

At the beginning, the docker engine had two major components:

- **The Docker daemon**: it used to be a monolithic binary that contained all the code for the Docker client, the API, the container runtime, image builds, and much more. This is not what Docker wanted (slow and hard to innovate), and the work of breaking apart the daemon into smaller tools is still an ongoing process. To this day, it has already seen all of the container execution and container runtime code entirely removed. Is is becoming more and more modular, and many components can be swapped, thanks to the OCI (Open Container Initiative) standards.
- **LXC**: provided the daemon with access to fundamental building-blocks of containers that existed in the Linux kernel (*namespaces* and *control groups*). **Problems**: LXC is Linux specific and Docker didn't want to rely on an external tool for the core of the project. They replaced LXC with their own platform-agnostic tool called *libcontainer*, which provides Docker with accesss to the fundamental container building-blocks that exist inside the kernel.

Current architecture

The **current Docker engine architecture** is the following.

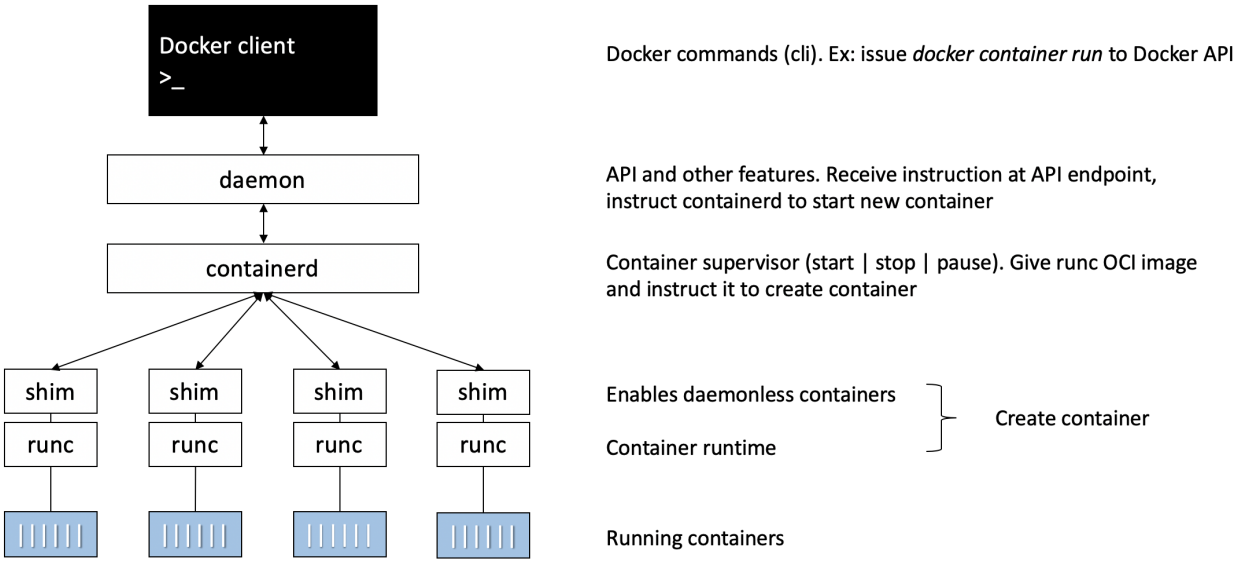


Figure 1: Current Docker engine architecture

The old monolithic daemon was broken down, giving birth to new layers:

- **runc**: tool that manages container runtime code. It's the reference implementation of the OCI container-runtime-spec. In fact, runc is a CLI wrapper for libcontainer. It has a **single purpose: create containers**.

- **containerd**: tool that used to only manage container **lifecycle operations** (*start*, *stop*, *pause*, *rm*, ...). It now also takes care of **image management** (push and pull, ...).

The daemon still performs image management, image builds, REST API, authentication, security, core networking and orchestration.

Creation of a container: under the hoods

So, what happens when we create a new container via the docker CLI (the Docker client) ?

```
# docker container run --name ctr1 -it alpine:latest sh
```

Based on Fig. 1, the Docker client converts the command into the appropriate API payload and POSTs it to the correct API endpoit.
We saw that the REST API is implemented in the Daemon, which is the layer under the Docker client.
Once the daemon receives the command to create a new container, it makes a call to containerd via a CRUD-style API over gRPC.
Containerd converts the required Docker image into an OCI bundle and tells runc to use it to create a new container.
Runc interfaces with the OS kernel to pull together all of the constructs necessary to create a container (namespaces, cgroups, ...). The container process is started as a child-process of runc, and as soon as it is started runc will exit. This means we can run hundreds of containers without having to run hundreds of runc instances.
Once a container's parent runc process exits, the associated containerd-shim process becomes the container's parent. Among other things, shims have the responsibility to keep any stdin/stdout streams open (so that when the daemon is restarted, the container doesn't terminate due to pipes being closed) and to report the container's exit status back to the daemon.

The **advantage** of this model (code to start and manage containers out of the daemon) is that it makes it possible to **perform maintenance and upgrades on the daemon without impacting running containers**. In the old model, every daemon upgrade would kill all containers on the host.

In Linux, all these components are implemented in separate binaries: *dockerd* (docker daemon), *docker-containerd*, *docker-containerd-shim* and *docker-runc*.

Images

Images are like a stopped container. We pull images from an image registry (like Docker Hub), and use it to start one (*docker container run*) or more (*docker service create*) containers.
Images are made up of **multiple layers** stacked on each other and represented as a single object. They contain a **cut-down OS** and all of the files and dependencies required to run an app. They are usually small and lightweight. For example, the official Alpine Linux Docker image is about 4MB in size! Windows-based images are a little bit different and tend to be bigger.

Figure 2: Docker image

It is possible to stop a container and create a new image from it: images are considered *build-time* constructs, whereas containers are *run-time* constructs. Once a container is started, the two constructs become dependant on each other. We cannot delete the image until the last container using it has been stopped and destroyed.

On Linux, the image repository is located at */var/lib/docker/<storage-driver>*. We can see if our host has any images in its local repository with the following command:

```
# docker image ls
```

Repositories and pulling images

The syntax to pull an image is *docker image pull <image>*. For example:

```
# docker image pull ubuntu:latest
```

We can then check the size and other information with *docker image ls*.

An image registry (like Docker Hub) contains multiple repositories, which can contain multiple images. There are official and unofficial repositories. On Docker Hub, official repositories contain images that have been vetted by Docker -> they contain up-to-date, secure and high-quality code. Unofficial repositories can contain malicious images. Most of the popular OS and apps have their own official repositories on Docker Hub.

When working with an image from an official repository, the syntax for *docker image pull* is *docker image pull <repository>:<tag>*. In the previous command, we pulled the latest image from the ubuntu repository. Note that *:latest* it the default value and it is not necessary to specify it. Also, *latest* isn't necessary the most recent image; for example, the most recent image in the alpine repository is tagged as *edge* (*latest* is the most recent stable image).
The syntax to pull images from an unofficial repository is *docker image pull <Docker Hub user-name/organization>/<repository>:<tag>*. Example:

```
# docker image pull nigelpoulton/tu-demo:v2
```

To pull an image from a third party registry (such as Google Container Registry GCR), we need to prepend the repository name with the DNS name of the registy. The last command would become:

```
# docker image pull gcr.io/nigelpoulton/tu-demo:v2
```

Finally, we can pull all of the images in a repository by adding the flag *-a* to the *docker image pull* command. A same image can have different tags (they refer to the same image).

Filters

The output of *docker image ls* can be filtered with the flag *–filter*. Example:

```
# docker image ls --filter dangling=true
```

A **dangling** image is an image that is no longer tagged, and appears in listings as *<none>:<none>*. This happens when building a new image and tagging it with an existing tag. Dangling images can be deleted with the *docker image prune* command. Adding the flag *-a* also removes unused images (not in use by any container).

Filters supported by docker are:

- dangling: true or false
- before: requires an image name/ID as argument, and returns all images created before it
- since: same as *before*, but returns images created after the specified image
- label: filters images based on the presence of a label or label and value.

The *–format* flag formats the output of the command. Example:

```
# docker image ls --format "{{.Repository}}: {{.Tag}}: {{.Size}}"
```

This command returns all images, but only display repo, tag and size. Note that we can also use tools from the OS such as grep and awk.

We can also search Docker Hub from the CLI with the *docker search* command. For example, we can search all repos with "nigelpoulton" in the "NAME" field:

```
# docker search nigelpoulton
```

Another example with a filter:

```
# docker search alpine --filter "is-official=true"
```

By default, *docker search* displays 25 lines of results. This can be changed with the *–limit* flag.

Image layers

We saw that images are made up of stacked read-only layers, and represented as a single object. In fact, we see those layers in the output of a *docker image pull* command:

```
# docker image pull ubuntu:latest
```

The output of this command is the following:

latest: Pulling from library/ubuntu
d51af753c3d3: Pull complete
fc878cd0a91c: Pull complete
6154df8ff988: Pull complete
fee5db0ff82f: Pull complete

Digest: sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e333bbceed5c54d7
Status: Downloaded newer image for ubuntu:latest

Each line in this output that ends with "Pull complete" represents a layer in the pulled image. So, this image has 4 layers.
Another way to see the layers of an image is to inspect it with the *docker image inspect* command:

```
# docker image inspect ubuntu:latest
```

In this output, we see the SHA256 hashes of the 4 layers.

All docker images start with a base layer. Layers are added on top when changes are made or when new content is added. Over-simplified example: we create an image based off Ubuntu Linux 16.04 (first layer) and add the Python package (second layer). Finally, we add a security patch. This would result in the following three layers image:

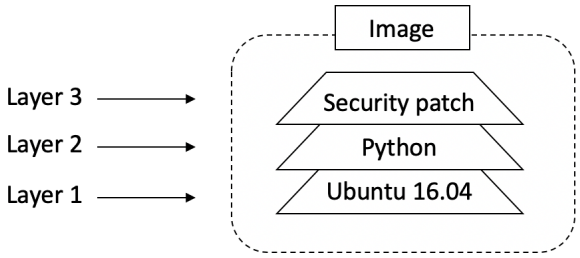


Figure 3: Image layers

It's important to understand that the **image** is always the **combination of all layer**s. For example, the last image has 3 layers and each of these layer has some files that are combined.

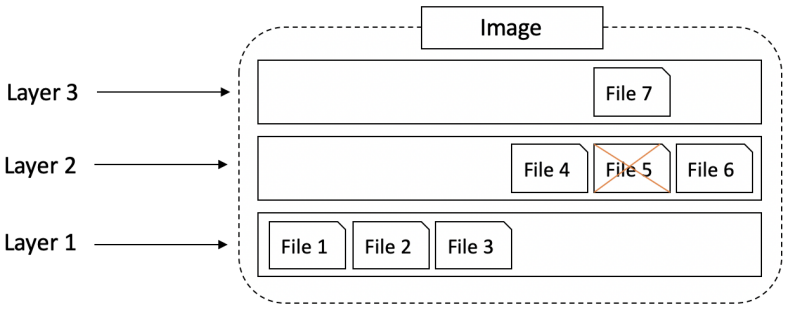


Figure 4: Files in image layers

Here, the overall image only represents 6 files. This is because file 7 in the top layer is an updated version of the file 5 directly below (inline). In this situation, the file in the higher layer obscures the file directly below it. This allows updates versions of files to be added as new layers to the image.

Docker uses a storage driver (snapshotter in newer versions) that is responsible for stacking layers and presenting them as a single unified filesystem. Examples of storage drivers on Linux include *AUFS*, *overlay2*, *devicemapper*, *btrfs* and *zfs*.

The next figure shows the same 3-layer image as it will appear to the system (all 3 layers stacked and merged):

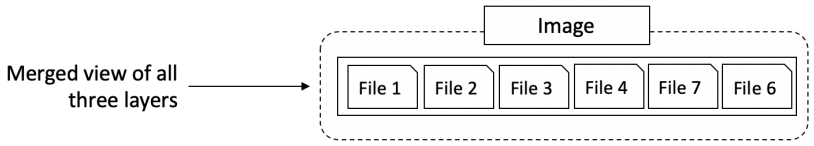


Figure 5: Merging of files

Multiple images share layers, for efficiencies in space and performance. Let's look at the following command:

```
# docker image pull -a nigelpoulton/tu-demo
# docker image ls
```

This command pulls all tagged images in the nigelpoulton/tu-demo repository. In the output, it's written "Already exists" for some layers. This is because Docker recognized that it had copies of some image layers. Here, docker pulled the *latest* image firt. Then, when it pulled the *v1* and *v2* images, it noticed that it already had some of the layers that make up those images.

So far, we've seen how to pull an image by tag. However, it can be a problem because tags are mutable (it's possible to tag an image with the wrong tag).
This is where image digests come to the rescue. All images get a cryptographic content hash (immutable), and we refer to it as the *digest*. We can see the digest of an image with:

```
# docker image ls --digests alpine
```

Using the digest ensures that we get exactly the image we expect. At the time of the writing of the book, the only way to determine an image's digest is to pull it by tag and then make a note of its digest. However, this should change in the future.
An image is just a configuration object that lists the layers and some metadata. The layers are where the data lives. Each layer is independant and has no concept of being part of a collective image.
Each image is identified by an ID that is a hash of the config object, and each layer is identified by an ID that is a hash of its content -> changing an image or its layers will change those IDs -> image and layers are immutable. We call these hashes content hashes.
Problem: when we push and pull images, their layers are compressed. This changes their content, and hence their content hashes... So, how can we check that the image we downloaded is safe ? Each layer also gets a **distribution hash**, which is the hash of the compressed version of the layer.

Multi-architecture images

Is the image we're pulling built for the architecture we're running on ? Docker now supports multi-architecture images. This means a single image can have an image for Linux on x64, Linux on PowerPC, Windows x64, ARM, etc... So, a single image tag can support different architectures. To make this happen, the registry API supports 2 important constructs:

- **Manifest lists**: list of architectures supported by a particular image tag. Each supported architecture has its own manifest.
- **Manifests**: list detailing the layers each architecture is composed from.

Below is an example of the official golang image.

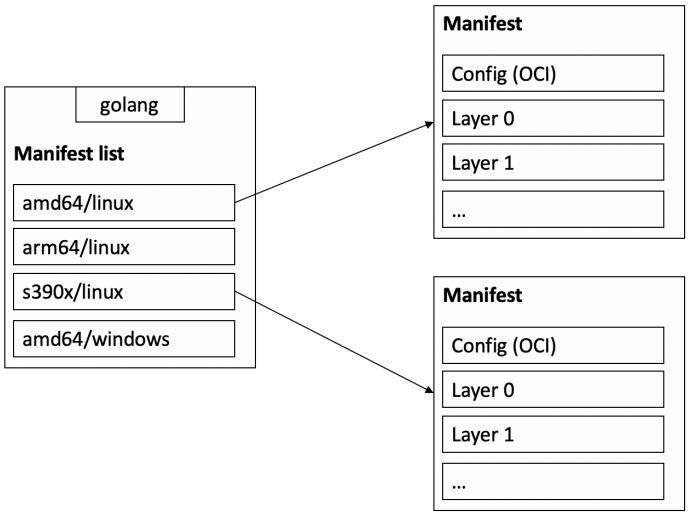


Figure 6: Multi-architecture images

Let's imagine we're running Docker on a Raspberry Pi (Linux running on ARM architecture). When we pull an image, our docker client makes the relevant calls to the Docker Registry API running on Docker Hub. If a manifest list exists for the image, it will be parsed to see if an entry exists for Linux on ARM. If it is the case, the manifest for that image is retrieved and parsed for the crypto ID's of the layers that make up the image. Each layer is then pulled from Docker Hub.
The following example shows how this work by pulling the official golang image:

```
# docker container run --rm golang go version
```

We see the version of Go as well as the CPU architecture of the host. Note that we don't need to tell Docker we need the Linux x64 or Windows x64 versions of the image. Docker takes care of getting the right image for the platform and architecture we're running. At the time of the writing of the book, all official images have manifest lists.

Deleting images

Finally, when we're done with an image, we can delete it from our Docker host with the *docker image rm* command. We cannot delete an image that is being used by a running or a stopped container. If an image layer is shared by more than one image, that layer will not be deleted until all images that reference it have been deleted.

For example, we can delete the golang image with (see image with *docker image ls*):

```
# docker image rm golang
```

Note that we could also use the image ID (just the few first characters). We can delete all image on a docker host with the following command:

```
# docker image rm $(docker image ls -q) -f
```

because *docker image ls -q* returns a list containing just the images IDs.

Containers

A container is the **execution environment of an image**. We can start one or more containers from an image. We can create our own images or start from an existing one (see https://hub.docker.com/ for example).

Before looking at the commands for containers, let's see the differences between containers and VMs in detail. To illustrate this difference, we will imagine the following situation: we want to run 4 applications on a physical server (both need a host such as a laptop, a bare metal server, etc). How does the scenario differ between VMs and containers?

The VM model

In the VM model, the physical server and the hypervisor (a virtulization platform that allows multiple OS to work on the same physical machine) are started. The hypervisor uses the physical resources of the system (CPU, RAM, storage, ...). Then, it carves these resources into virtualized ones in order to create a software called a *virtual machine*. In our scenario, this process is replicated 4 times. We have to install an OS and our application on each of the 4 VMs.

The container model

When the server is started, our chosen OS also starts. The OS also uses physical resources. Then, we install a *container engine* (for example Docker) over the OS. A container engine is in fact an OS whose kernel allows the existence of several isolated instance, the containers. Docker uses resources of the OS such as processes, the file system and the network stack to create one or more containers.
Each container looks drop for drop like a real OS, and we can run an application in each of them. In our scenario, we create 4 containers.

The image below shows the differencess between the 2 models.

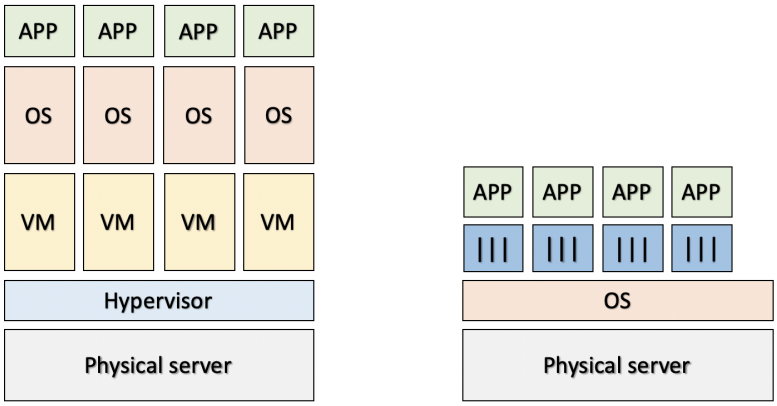


Figure 7: Difference between VMs (left) and containers (right)

So, a **hypervisor virtualizes hardware, while containers virtualize an OS**. It is very important to understand that **containers share the host OS**. In theory, an application developped for Linux containers will not work on Windows (although it's possible in practice).

So far, it seems that the two models are quite identical. However, we see on the image above that each VM is a software that contains a virtual CPU, a virtual RAM, virtual storage disks, and so on.
Those virtual resources are handled by an OS, and the problem is that every OS consumes CPU, RAM, storage, etc. Those OS also require a licence, patching, monitoring, and are vulnerable attack vectors...
This is what we call the **VM tax**: each OS that we install consumes resources.

The advantage of the container model is that it has a single kernel that runs on the host's OS, and this single OS can run hundreds of containers. This means there is only one OS consuming resources, one license, one possible vector of attack, etc. Also, containers contain lightweight OS with no kernel (which is shared with the host and already started); they start way faster than a VM. The time to start a container only depends on the time required to start the application it runs.
This difference becomes more and more as the number of applications increases.

All those differences are in favor of the container model.

Running containers

Let's see the commands to run containers. The following examples can be replicated on any Docker host or on Play with Docker (https://labs.play-with-docker.com/).

The syntax to create a container is *docker container run <options> <image>:<tag> <app>*. Example:

```
# docker container run -it ubuntu /bin/bash
```

We're in the container and the 12 digits after *@* are the first 12 digits of the container's ID. We see that the daemon didn't find the image locally (in cache), so it pulled it from Docker Hub and cached it. We can see it by exiting the container with* Ctrl+d* and by rerunning the previous command.

The command above starts a Linux container with Bash shell.
The flag *-it* connects our terminal to the container's shell and makes it interactive.
We didn't specify the *<tag>*, and it takes the *latest* image of ubuntu by default. We can check we're in Linux with the command *uname*, and in Ubuntu with *uname -v*.
Finally, we asked to run Bash shell, that's why we can use commands such as *ls*, *cd*, and so on. However, some commands don't work because the image is optimized for containers and doesn't contain all the commands and packages... For example, the command *nano file.txt* won't work.

So, running containers is very easy, fast and platform-agnostic.

Container processes

In the previous example, we started a container that runs Bash shell. It was the single and unique process running in the container. We can see it with the command *ps -elf*: there is ps -elf itself, but it is dead by the time it is displayed. Windows containers are a little bit different and run more processes.

If we use the command *exit* to get out of Bash shell, we also exit the container -> a container cannot live without running a process (same for Windows if we kill the main process).

So, how can we exit a container without killing it ? Use the command *Ctrl -PQ*, which brings us back to our docker host's shell. The container still runs in the background, as shown by the command:

```
# docker container ls
```

We can attach back to the container with the command:

```
# docker container exec -it <ID> bash
```

"exec" let us run a new process inside of a running container. If we execute *ps -elf* again, we now see 2 bash: this is because the previous command creates a new bash process to which we attach. If we exit this bash, it

doesn't kill the container. The container runs until the app it runs stops, or until we stop or kill it. We can see it by running the following command:

```
# docker container run alpine:latest sleep 10
```

Finally, we can stop and kill a container with the 2 following commands:

```
# docker container stop <ID>
# docker container rm <ID>
```

We saw it is a good practice to proceed in two steps, although it is possible to do it with one command (*docker container rm -f*). In fact, *docker container stop* sends a **SIGTERM** signal to the PID 1 process inside of the container. In then have 10 seconds to clean things up and shut itself down. If it doesn't exit within 10 seconds, it will receive a **SIGKILL**.

Containers lifecycle

It is a myth that containers can't persist data. let's look at the lifecycle of an Ubuntu container.

```
# docker container run --name persi -it ubuntu:latest /bin/bash
```

We create a container called "persi" in which we'll write data. Inside the container:

cd /tmp
ls -al
echo "foobar" > newfile.txt
ls -al
cat newfile.txt

Now, we get out of the container with *Ctrl -PQ*, and check it is still here with *docker container ls*. We even stop the container:

```
# docker container stop <ID/name>
```

And check once again its presencee with *docker container ls*. It's not here... To see stopped containers, we need to add the flag *-a*:

```
# docker container ls -a
```

We see in its status that it's "exited". Although it's not running, all its configuration and content still exist in the file system of the Docker host. It can then be restarted any time:

```
# docker container start persi
```

We see it again with *docker container ls*. Now, we can check the data were persisted. We reconnect to the container:

```
# docker container exec -it persi bash
```

and we can look for our file, which is still here. The best way to persist data in a container is in a volume, but we will see this later.

Let's kill and delete the container. First, we get out of the container with *Ctrl -PQ*. We then stop it:

```
# docker container stop persi
```

and delete it:

```
# docker container rm persi
```

We will see that if we persist data on a volume, they are safe even if we kill and remove the container.

Self-healing containers with restart policies

It's often a good idea to run containers with a restart policy. It enables Docker to automatically restart them after certain events of failures have occured.

Restart policies are applied per container, and can be configured imperatively on the command line as part of *docker container run* commands, or declaratively in Compose files for use with Docker Compose and Dockerr Stacks.

There are 3 restart policies at the time of writing:

- **always**: will always restart a stopped container, unless it has been explicitly stopped (*docker container stop*). Example: start an interactive container with *–restart always*, tell it to run a shell process, and type exit. Then, we look at the status with *docker container ls*.

```
# docker container run --restart always -it ubuntu /bin/bash
```

- **unless-stopped**: the main difference between *always* and *unless-stopped* is that containers with the *–restart unless-stopped* policy will not be restarted when the daemon restarts if they were in the stopped (exited) state. Example: we create 2 containers with 1 policy each, stop them, and restart Docker (on a host's terminal with Docker installed). The flag *-d* in the following commands stands for *daemon* mode: it makes the container start in the background, without attaching it to our terminal.

```
# docker container run --restart always -d --name always alpine sleep 1d
# docker container run --restart unless-stopped -d --name unless-stopped alpine sleep 1d
# docker container ls
# docker container stop always unless-stopped
# docker container ls -a
# systemctl restart docker # <- for Linux, cmd+R on mac
# docker container ls -a
```

- **on-failure**: restarts a container if it exits with a non-zero exit code. It will also restart containers when the Docker daemon restarts, even containers that were in the stopped state.

Web server example

Let's look at a Linux web server example. The image we will use uses a very simple web server on port 8080. We start by cleaning up existing containers on our system:

```
# docker rm $(docker ps -a -q)
```

We then run the following command:

```
# docker container run -d --name webserver -p 80:8080 \
# nigelpoulton/pluralsight-docker-ci
```

We saw that the flag *-d* starts the container in the background, and this is why our prompt didn't change. We used the flag *-p* and specified 80:8080. This flag maps ports on the Docker host to ports inside the container. Here, we're mapping port 80 on the host to port 8080 in the container. This means that traffic hitting the host on port 80 will be redirected to port 8080 inside the container.
The image we're using defines a web service that listens on port 8080, so our container will come up running a web server listening on port 8080. *docker container ls* shows the ports that are mapped with the syntax *host-port:container-port*.
Now that the container is running and ports are mapped, we can connect to the container by pointing a web browser at the IP address or DNS name of the Docker host on port 80 (on Play with Docker, click on "OPEN PORT" and type 80).

Inspecting containers

We just ran a container without specifying any app. Yet the container ran a web service... Why ?! When building a Docker image, it is possible to embed an instruction that lists the default app to run. In the previous case, we can see it by inspecting the image:

```
# docker image inspect nigelpoulton/pluralsight-docker-ci
```

The entries after *Cmd* show the command/app that the container will run by default. Here, we see the following commands: */bin/sh -c "cd /src && node ./app.js"*.
Building images with default commands is very common, as it makes starting containers easier.

Containerizing an app

Docker is about taking applications and running them in containers. **Containerizing** is the process of taking an app and configure it to run as a container. In this section we will see how to containerize a simple Linux web app.

At this point, we already know how it should look: - We write code for an app - Create a Dockerfile that describes the app, its dependencies and how to run it - Feed the Dockerfile into *docker image build* - Docker builds the app into a Docker image -> it containerizes it - We can ship the app and run it as a container or service

Containerize a single-container app

We will containerize a simple single-container Node.js web app. Once the app is containerized and tested, we move to **production** with **multi-stage builds**.

Write code for an app

Here, we will use an app written by Nigel. The app code for this example comes from https://github.com/nigelpoulton/psweb. Clone the sample app:

```
# git clone https://github.com/nigelpoulton/psweb
```

It creates a new directory called psweb. *cd psweb* to it and list its content with *ls -al*. This directory contains all of the application source code. The code can be viewed with *cat app.js*.

Create a Dockerfile

Let's inspect the Dockerfile with *cat Dockerfile*. This file tells Docker how to build the app into an image. The directory containing the app is referred to as the **build context**. It's a good practice to keep the Dockerfile in the root directory of the build context.

The Dockerfile has 2 purposes: - To describe the app - To tell Docker how to create an image with the app inside (containerize it)

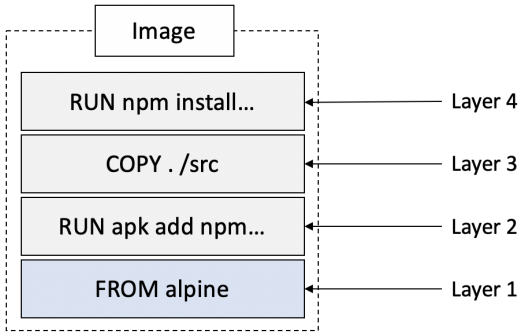
The present Dockerfile contains the following:

FROM alpine
LABEL maintainer="nigelpoulton@hotmail.com"
RUN apk add –update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT [“node”, “./app.js”]

We can comment lines in a Dockerfile with “#”. Non-comment line are instructions of the form *Instruction* argument. If an instruction is adding content (files and programs), it will create a new layer. If it is adding instructions on how to build the image and run the app, it will create metadata.
The *docker image build* command will read the Dockerfile one line at a time. For each instruction, Docker looks to see if it already has an image layer for that instruction in its cache. If it does, it uses it, otherwise it builds a new layer. However, as soon as no layer was found for an instruction, the cache is no longer used for the rest of the entire build -> impacts how we write Dockerfiles (instructions likely to change towards the end!). Docker checks that cached files haven't changed. If it did, it doesn't use that layer but builds a new one.

All Dockerfiless start with the *FROM* instruction, which will be the base layer of the image. The rest of the app will be added on top as additional layers. Because it's a Linux app, it's important that the FROM instruction refers to a Linux-based image.
Then, the Dockerfile creates a LABEL that specifies "nigelpoulton@hotmail.com" as the maintainer of the image. Labels are key-value pairs and an excellent way of adding custom metadata to an image. Listing a maintainer is considered a best practice. The *RUN apk add –update nodejs nodejs-npm* instruction uses the Alpine *apk* package manager too install nodejs and nodejs-npm into the image. The RUN instruction installs these packages as a new image layer.
Then, the *COPY . /src* instruction copies in the app files from the build context. It copies these files into the image as a new layer.
Next, the Dockerfile uses the *WORKDIR* instruction to set the working directory for the rest of the instructions in the file. This directory is relative to the image -> the info is added as metadata to the image config and not as a new layer.
Then, the *RUN npm install* instruction uses *npm* to install application dependencies listed in the *package.json* file in the build context (Among other things, the *package.json* contains "dependencies": { "express": "4.17.1", "pug": "2.0.4", "mocha": "7.1.1", "supertest": "4.0.2" }). In installs them as a new layer in the image. The app exposes a web service on TCP port 8080, so the Dockerfile documents this with the *EXPOSE 8080* instruction. This is added as metadata and not an image layer.
Finally, the *ENTRYPOINT* instruction is used to set the main application that the image (container) should run. This is also added as metadata.

The resulting image contains 4 layers as follows:



We can also view the instructions that were used to build the image:

```
# docker image history web:latest
```

Each line corresponds to an instruction in the Dockerfile. The lines with non-zero values in *SIZE* are lines which created new layers. We can see these 4 layers with *docker image inspect* command.

Containerize the app (build the image)

We can now build the image by feeding the Dockerfile into *docker image build*. The following command will build a new image called web:latest. The period (.) at the end of the line tells Docker to use the shell's current working directory as the build context.

```
# docker image build -t web:latest .
```

We can then see the image with *docker image ls*. The app is containerized, and we can inspect it:

```
# docker image inspect web:latest
```

We see the settings that were configured from the Dockerfile. In fact, what happens when we *build* an image is: spin up a temporary container > run the Dockerfile instruction inside of that container > save the results as a new image layer > remove the temporary container.

Pushing the image

We have an image locally, but generally we want to make it available to others. This is why we can store it in an image registry. Docker Hub is the most common public image registry and is the default push location for *docker image push* commands. Log in to Docker Hub:

```
# docker login
```

And log in with your Docker credentials. Before pushing an image, it has to be tagged in a special way. This is because Docker needs the following information: registry, repository and tag. The default vales are: Registry=docker.io and Tag=latest. Docker doesn't have a default value for the repository. So, we tag the image:

```
# docker image tag web:latest <Docker ID>/web:latest
```

docker images ls now shows 2 images (the previous command didn't overwrite the existing image, but created a second image with the new tag). The Repository value taken will be the one we see under REPOSITORY in the *docker image ls* output. We can now push the image to Docker Hub:

```
# docker image push <Docker ID>/web:latest
```

We can navigate to https://hub.docker.com/, and the image should be there.

Run the app and test it

We run a container called c1, based on the web:latest image we just created:

```
# docker container run -d --name c1 \
# -p 80:8080 \
# web:latest
```

Note that \ at the end of the line allows to split the instruction on several lines. We check the port mapping and that the container is running with *docker container ls*.

If the previous output is good and the container is running, we can test it by opening a web browser and browsing to localhost:80 or, on Play with Docker, by clicking on *OPEN PORT* and typing 80.

Moving to production with Multi-stage builds

We want to ship images containing only the stuff **needed** to production, that is a lightweight, fast and safe image. In the past, it was common to create a *Dockerfile.dev* and a *Dockerfile.prod*. Nowadays, it is easier to use **multi-stage builds**. In multi-stage builds, we have a single Dockerfile containing multiple *FROM* instructions, and each of them is a new build stage that can easily *COPY* arrtefacts from previous stages.

Example: let's clone an existing app from Github:

```
# git clone https://github.com/nigelpoulton/atsea-sample-shop-app
```

Then, let's change directory with *cd atsea-sample-shop-app/app/* and inspect it with *ls -l*. We see the Dockerfile, and we can inspect it with *cat Dockerfile*.
We see 3 blocks of instructions (= 3 build stages), each starting with a different *FROM*: - Stage 0 is called *storefront*: it pulls thee node:latest image (~600MB). It sets the working directory, copies in some app code, and uses 2 *RUN* instructions. This addss 3 layers and considerable size. - Stage 1 is called *appserver*: it pulls the maven:latest image (~700MB). It adds 4 layers of content (2 *COPY* and 2 *RUN*), resulting in an even bigger image. - Stage 2 isn't named here, but in the book it's called *production*: it pulls the java:8-jdk-alpine image (~150MB). It adds a user, set the WD, and copies some app code from the image produced by the *storefront* stage. After that, it sets a different WD and copies in the app code from the image produced by the *appserver* stage. Finally, it sets the main app for the image to run when it's started as a container.

It's important to note that *COPY –from* instructions only copy production-related app code, and not build artefacts. We can now build it:

```
# docker image build -t multi:stage .
```

The *multi:stage* tag is arbitrary, we could have tagged it differently. We can see the list of images pulled and created by the build operation with *docker image ls*. The multi:stage image is significantly smaller than the other ones.

Squashing an image

Squashing is the process of squasing all the layers of an image into a single layer. It might be good if an image starts to have a lot of layers (especially if we want to build other images from it). The negative side is that squashed images do not share layers with other images on the host (storage inefficiencies). We can create a squashed image by adding the *–squash* flag to the *docker image build* command. Squashing is not a best practice, but can be useful sometimes.

Good practices

When building Linux images with the apt package manager, use the *no-install-recommend* flag with the *apt-get install* command. This makes sure that apt only installs main dependencies. When building Windows images, don't use the MSI package manager (not space efficient).

Deploying apps with Docker Compose

Most modern apps are made of multiple smaller services (**microservices**) that interact to form a useful app. Example : an app with the 4 following services: web front-end, ordering, catalog, back-end database.

Docker Compose is an external tool (python library) that helps deploying and managing all those services. It lets us describe an entire app in a single declarative configuration **YAML file**. We then pass the YAML file to the *docker compose* binary, and Compose deploys it via the Docker Engine API. Once deployed, we can manage the app's lifecycle with a simple set of commands.

Docker Compose has to be installed on top of the Docker engine.

Installing Compose

Docker Compose is installed as part of Docker for Mac (DfM), and it's the same for Windows (DfW). The book details the installation for Linux and Windows Server, but this is not covered here.
Note that Docker Compose is also available on Play With Docker.

In a terminal (or on Play With Docker), we can check that we have Docker Compose with the following command:

```
# docker-compose --version
```

Compose files

Compose uses YAML files to define multi-service applications. YAML is a subset of JSON, so we can also use JSON.
The default name for the Compose YAML file is *docker-compose.yml*.

Example: let's see the Compose file that define a simple Flask app (Flask is a web framework (allows to build web apps)). The app is called *counter-app* and has 2 services: *web-fe* and *redis*. It's a simple web server that counts the number of visits and stores the value in Redis (Redis is an open source, in-memory data structure store, used as a database, cache and message broker).

The Compose file is the following:

```
version: "3.5"
services:
  web-fe:
    build: .
    command: python app.py
    ports:
      - target: 5000
        published: 5000
    networks:
      - counter-net
    volumes
      - type: volume
        source: counter-vol
        target: /code
  redis:
    image: "redis:alpine"
    networks:
      counter-net:

networks:
  counter-net:
```

volumes:
  counter-vol:

We see that the file has 4 top-level keys (there could be other top-level keys such as configs and secrets):

- version: defines the version of the Compose file format. This key is mandatory and is always the first line.
- services: where we define the different application services. We have 2 services in our example: a web front-end (web-fe), and an in-memory database (redis). Compose will deploy each of these services as its own container.
- networks: tells Docker to create new networks. By default, it creates *bridge* networks (single host networks that can only connect containers on the same host). We can select different network types with the *driver* property.
- volumes: where we tell Docker to create new volumes.

In our example, the file uses the the Compose v3.5 file format, defines 2 services, defines a network called counter-net, and defines a volume called counter-
vol. The service section has 2 second-level keys: web-fe and redis. They each define a service in the app, and Compose will deploy each as a container. Let's look at those services in detail, starting with web-fe:

- *build: .* tells Docker to build a new image using the Dockerfile in the current directory. This image will be used to create the container for this service.
- *command: python app.py* tells Docker to run a Python app called app.py as the main app in the container (the app.py file must exist in the image, and the image must contain Python -> in the Dockerfile).
- *ports* tells Docker to map port 5000 inside the container to port 5000 on the host (published). Traffic sent to the Docker host on port 5000 will be redirected to port 5000 on the container. The app inside the container listents oon port 5000.
- *networks* tells Docker which network to attach the service's container to.
- *volumes* tells Docker to mount the counter-vol volume to /code inside the container.

And redis:

- *image: redis:alpine* tells Docker to start a standalone container called redis based n the redis:alpine image (will be pulled from Docker Hub).
- *netwrks*: the redis container will be attached to the counter-net network.

Because both services will be deployed on the same network (counter-net), they will be able to resolve each other by name (important since the app is configured to communicate with the redis service by name).

Deploying an app with Compose

We use an app that we clone from Github:

```
# git clone https://github.com/nigelpoulton/counter-app.git
```

If we cd in the *counter-app* directory , we see the following objects:

- Dockerfile: describes how to build the image for the web-fe service
- app.py: application code (Python Flask app)
- docker-compose.yml: Compose file that describes how Docker should deploy the app
- requirements.txt: lists the packages required for the app

In this case, *counter-app* is the *build-context*. We are ready to bring the app up by using Compose:

```
# docker-compose up &
```

docker compose up is the most common way to bring up a Compose app. It builds all required images, creates all required networks and volumes, and starts all required containers. By default, *docker-compose up* expects the name of the Compose file to be *docker-compose.yml* or *docker-compose.yaml*. If it has a different name, we must specify it with the flag *-f*.
We could have used the flag *-d* to bring up the app in the background. We didn't do it here, and the *&* we used in the command gave us the terminal window back (so we get verbose outputs).

We can now see the images with *docker image ls*: 3 images were either built or pulled as part of the deployment. *docker container ls* shows 2 containers, *docker network ls* shows the network, and *docker voolume ls* the volumes.

On Play with Docker, we can click on *OPEN PORT* and type 5000. This should redirect us to the front-end page, which counts how many times we visited the page and stores it in the Redis back-end database (refresh the page).

Managing an app with Compose

How to start, stop, delete, and get the status of apps being managed by Docker Compose?
We'll also see how the volume we're using can be used to directly inject updates to the app's web front-end.

We can bring down the app with the follwing command:

```
# docker-compose doum
```

In the output, we see that the counter-vol volume was not deleted, because it is intended to be a long-term persistent data store -> its lifecycle is decoupled from the containers it serves. This is why *docker volumes ls* still shows it.
The images used also remain on the system, so future deployments will be faster.

We can bring the app up again, this time in the background, with the command:

```
# docker-compose up -d
```

This time, it's really fast. We can see the current state of the app:

```
# docker-compose ps
```

We see the 2 containers, the commands they are running, their current state, and the network ports they are listening on.

We can list the processes running in each container with

```
# docker-compose top
```

The app can stopped without deleting its resources with

```
# docker-compose stop
```

Then, we can look at the status of the app with *docker-compose ps*, and check that the containers are still here with *docker container ls -a*.
We can restart the app with the following command:

```
# docker-compose restart
```

Finally, we can delete a stopped Compose app with with

```
# docker-compose rm
```

This doesn't delete volumes, images, and the files in the build-context. It just deletes containers and networks the app is using.
Note that we can **stop and delete** an app with the **single command** *docker-compose down*.

We saw earlier that we defined a new volume in the Compose file called *counter-vol*. This volume was mounted in the web-fe service at */code*. The first time we deployed the app, Compose checked if a volume already existed with this name. It didn't so it created it (we see it with *docker volume ls*).
We can also see from the Dockerfile that */code* is where the app is installed and executed from. In other words, the app code resides on a Docker volume.

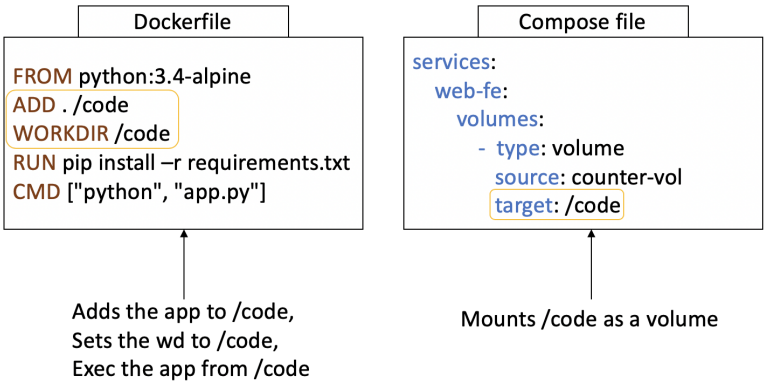


Figure 8: Image of the application

Commands