A Glimpse into Docker

Dhiman Saha [1]   Sourya Kakarla [2]

[1]IIT Bhilai

[2]IIT Kharagpur

October 11, 2020

Outline

Operating System, Processes and Docker

Definition

An Operating System is processing of different processes in a collective unity. Our mind/being is an operating system if you want to have a live example to relate with. Isn't it?

Definition

A Process is processing of symbols/information which might or might knot interact with other processes in the whole OS.

Definition

A Virtual Machine is like running of a whole new Operating System in the original host OS. It is like the original OS simulates another OS in its processing.

Definition

A Docker Container is like a new way of doing things related to the Virtual Machine framework. It's main selling paints are that it's user-friendly and developer friendly.

- One of the reasons it is developer friendly because it takes care of packaging dependencies into a Docker Image which is like the basis for running the Container in the OS.
- One of the reasons it is user friendly because it takes care of seamlessway of installing dependencies and running the Container which is like an isolated process/OS.
- The isolation is key because it apparently really takes care of the problem of conflicting interests of processes regarding their dependencies.

- Limit on the resources used by the containers like memory usage, device use like webcam, graphic card etc. This is key from a security and governing point of view.
- To give an application the view that is running in an abstracted out OS belonging to its own needs.
- Sometimes it is said that Docker apps run even faster when running as Docker containers rather than when they are run as normal applications in the OS.

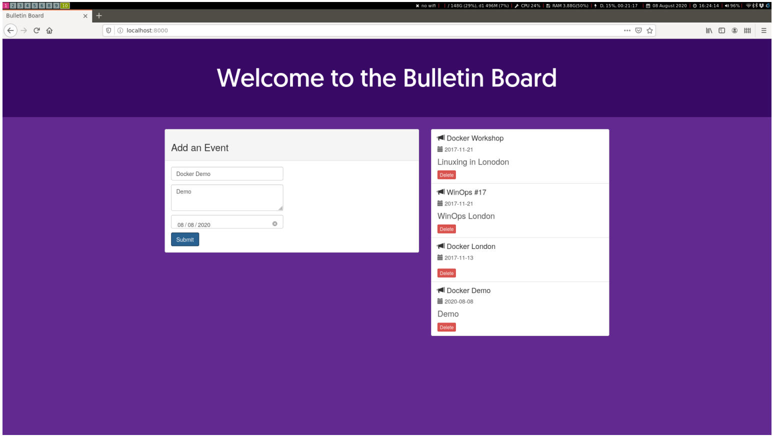
Installing Docker on Ubuntu

Please follow the steps given here on the official installation guide to install the Docker Engine.

Simple example of running a Docker App

We will discuss other features and motives behind Docker Container tech later, for now let's just dive into it!! We will start with a simple Node.js app which is given in the official Docker website.

Bulletinboard accessible on localhost:8000


1 2 3 4 5 6 8 9 10
no wifi | / 148G (29%), d1 496M (7%) | CPU 24% | RAM 3.88G(50%) | D, 15%, 00:21:17 | 08 August 2020 | 16:24:14 | 96%
Bulletin Board
localhost:8000

Welcome to the Bulletin Board

Add an Event

Docker Demo

Demo

08 / 08 / 2020

Submit

Docker Workshop
2017-11-21
Linuxing in Lonodon
Delete

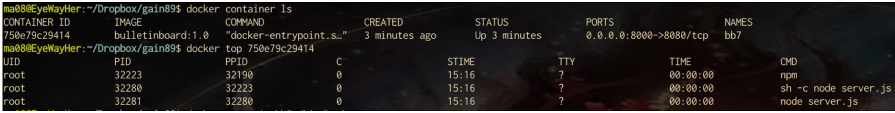
WinOps #17
2017-11-21
WinOps London
Delete

Docker London
2017-11-13
Delete

Docker Demo
2020-08-08
Demo
Delete

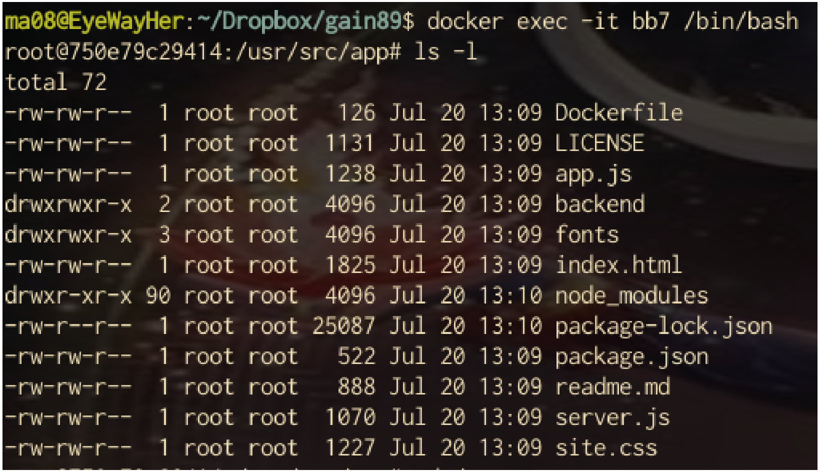
Figure: Node.Js App

Playing around with the Docker CLI



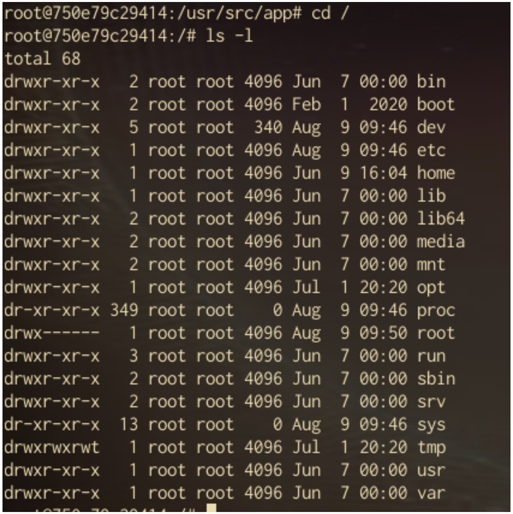
```
ma08@EyeWayHer:~/Dropbox/gain89$ docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
750e79c29414        bulletinboard:1.0   "docker-entrypoint.s…"   3 minutes ago       Up 3 minutes        0.0.0.0:8000->8080/tcp   bb7
ma08@EyeWayHer:~/Dropbox/gain89$ docker top 750e79c29414
UID                 PID                 PPID                C                   STIME               TTY                 TIME                CMD
root                32223               32190               0                   15:16               ?                   00:00:00            npm
root                32280               32223               0                   15:16               ?                   00:00:00            sh -c node server.js
root                32281               32280               0                   15:16               ?                   00:00:00            node server.js
```

Playing around with the Docker CLI



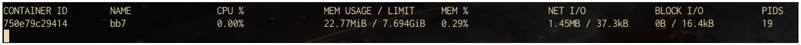
```
ma08@EyeWayHer:~/Dropbox/gain89$ docker exec -it bb7 /bin/bash
root@750e79c29414:/usr/src/app# ls -l
total 72
-rw-rw-r--  1 root root   126 Jul 20 13:09 Dockerfile
-rw-rw-r--  1 root root  1131 Jul 20 13:09 LICENSE
-rw-rw-r--  1 root root  1238 Jul 20 13:09 app.js
drwxrwxr-x  2 root root  4096 Jul 20 13:09 backend
drwxrwxr-x  3 root root  4096 Jul 20 13:09 fonts
-rw-rw-r--  1 root root  1825 Jul 20 13:09 index.html
drwxr-xr-x 90 root root  4096 Jul 20 13:10 node_modules
-rw-r--r--  1 root root 25087 Jul 20 13:10 package-lock.json
-rw-rw-r--  1 root root   522 Jul 20 13:09 package.json
-rw-rw-r--  1 root root   888 Jul 20 13:09 readme.md
-rw-rw-r--  1 root root  1070 Jul 20 13:09 server.js
-rw-rw-r--  1 root root  1227 Jul 20 13:09 site.css
```

Playing around with the Docker CLI



```
root@750e79c29414:/usr/src/app# cd /
root@750e79c29414:/# ls -l
total 68
drwxr-xr-x   2 root root 4096 Jun  7 00:00 bin
drwxr-xr-x   2 root root 4096 Feb  1  2020 boot
drwxr-xr-x   5 root root  340 Aug  9 09:46 dev
drwxr-xr-x   1 root root 4096 Aug  9 09:46 etc
drwxr-xr-x   1 root root 4096 Jun  9 16:04 home
drwxr-xr-x   1 root root 4096 Jun  7 00:00 lib
drwxr-xr-x   2 root root 4096 Jun  7 00:00 lib64
drwxr-xr-x   2 root root 4096 Jun  7 00:00 media
drwxr-xr-x   2 root root 4096 Jun  7 00:00 mnt
drwxr-xr-x   1 root root 4096 Jul  1 20:20 opt
dr-xr-xr-x 349 root root    0 Aug  9 09:46 proc
drwx------   1 root root 4096 Aug  9 09:50 root
drwxr-xr-x   3 root root 4096 Jun  7 00:00 run
drwxr-xr-x   2 root root 4096 Jun  7 00:00 sbin
drwxr-xr-x   2 root root 4096 Jun  7 00:00 srv
dr-xr-xr-x  13 root root    0 Aug  9 09:46 sys
drwxrwxrwt   1 root root 4096 Jul  1 20:20 tmp
drwxr-xr-x   1 root root 4096 Jun  7 00:00 usr
drwxr-xr-x   1 root root 4096 Jun  7 00:00 var
```

Runtime Metrics



```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
750e79c29414        bb7                 0.00%               22.77MiB / 7.694GiB   0.29%               1.45MB / 37.3kB     0B / 16.4kB         19
```

Dockerfile

Now let's have a look at the Dockerfile of the bulletinboard. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

Dockerfile Commands Explanation

The dockerfile defined in the bulletinboard example takes the following steps:

- Start FROM the pre-existing node:current-slim image. This is an official image, built by the node.js vendors and validated by Docker to be a high-quality image containing the Node.js Long Term Support (LTS) interpreter and basic dependencies.
- Use WORKDIR to specify that all subsequent actions should be taken from the directory /usr/src/app in your image filesystem (never the hosts filesystem).
- COPY the file package.json from your host to the present location (.) in your image (so in this case, to /usr/src/app/package.json)
- RUN the command npm install inside your image filesystem (which will read package.json to determine your apps node dependencies, and install them)
- COPY in the rest of your apps source code from your host to your image filesystem.

Docker Images

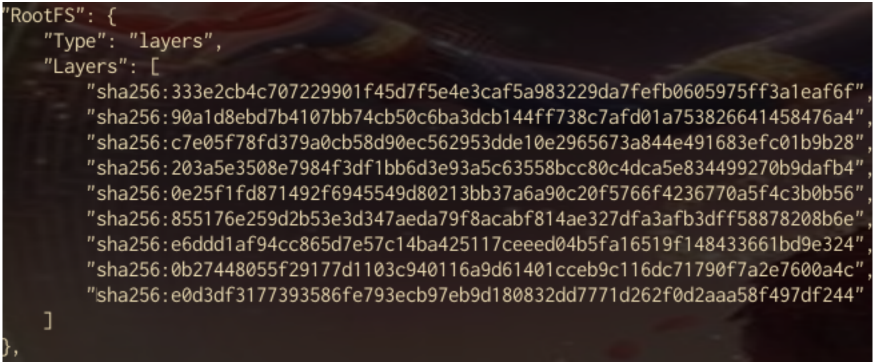
An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Layers in the Docker Image

In the Bulletinboard example we shall examine the layers of the image by using

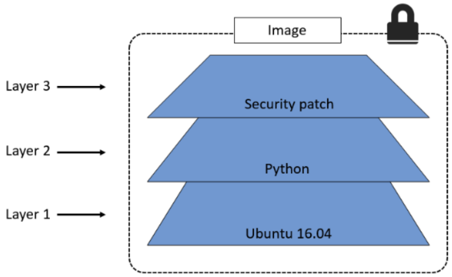
The relevant part of the output which shows the SHA256 hashes of the layers of the image is



```
"RootFS": {
    "Type": "layers",
    "Layers": [
        "sha256:333e2cb4c707229901f45d7f5e4e3caf5a983229da7fefb0605975ff3a1eaf6f",
        "sha256:90a1d8ebd7b4107bb74cb50c6ba3dcb144ff738c7afd01a753826641458476a4",
        "sha256:c7e05f78fd379a0cb58d90ec562953dde10e2965673a844e491683efc01b9b28",
        "sha256:203a5e3508e7984f3df1bb6d3e93a5c63558bcc80c4dca5e834499270b9dafb4",
        "sha256:0e25f1fd871492f6945549d80213bb37a6a90c20f5766f4236770a5f4c3b0b56",
        "sha256:855176e259d2b53e3d347aeda79f8acabf814ae327dfa3afb3dff58878208b6e",
        "sha256:e6ddd1af94cc865d7e57c14ba425117ceeed04b5fa16519f148433661bd9e324",
        "sha256:0b27448055f29177d1103c940116a9d61401cceb9c116dc71790f7a2e7600a4c",
        "sha256:e0d3df3177393586fe793ecb97eb9d180832dd7771d262f0d2aaa58f497df244"
    ]
},
```

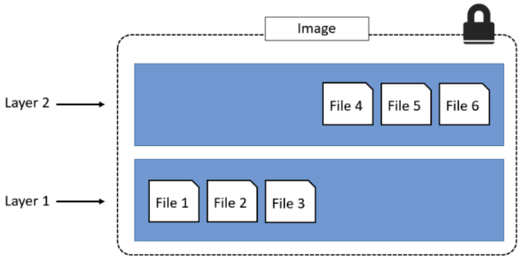
Layers in the Docker Image

All Docker images start with a base layer, and as changes are made and new content is added, new layers are added on top. As an over-simplified example, you might create a new image based off Ubuntu Linux 16.04. This would be your images first layer. If you later add the Python package, this would be added as a second layer on top of the base layer. If you then added a security patch, this would be added as a third layer at the top.



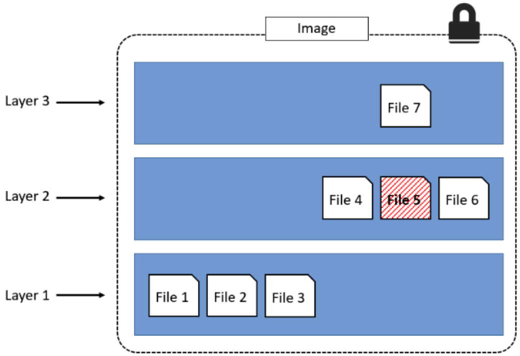
Layers in the Docker Image

Its important to understand that as additional layers are added, the image is always the combination of all layers. Take a simple example of two layers as shown. Each layer has 3 files, but the overall image has 6 files as it is the combination of both layers.



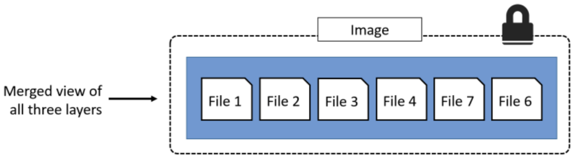
Layers in the Docker Image

In the slightly more complex example of the three-layered image in Figure 6.7, the overall image only presents 6 files in the unified view. This is because file 7 in the top layer is an updated version of file 5 directly below (inline). In this situation, the file in the higher layer obscures the file directly below it. This allows updated versions of files to be added as new layers to the image.



Unified File System

Docker employs 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 . As their names suggest, each one is based on a Linux filesystem or block-device technology, and each has its own unique performance characteristics.



Sharing Image Layers

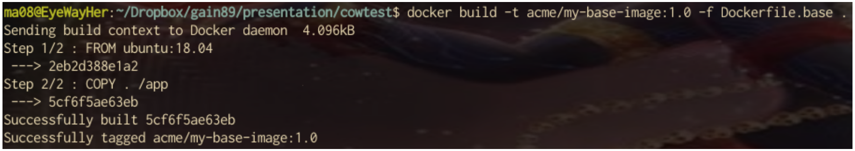
Multiple images can, and do, share layers. This leads to efficiencies in space and performance.

Save the following file as hello.sh

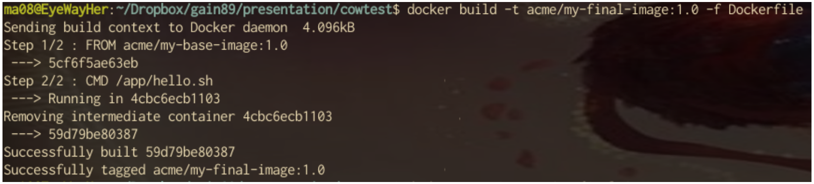
Save the following file as Dockerfile.base

Save the following file as Dockerfile

cowtest

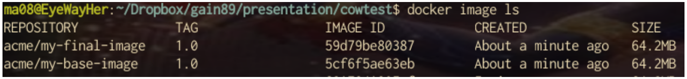


```
ma08@EyeWayHer:~/Dropbox/gain89/presentation/cowtest$ docker build -t acme/my-base-image:1.0 -f Dockerfile.base .
Sending build context to Docker daemon  4.096kB
Step 1/2 : FROM ubuntu:18.04
 ---> 2eb2d388e1a2
Step 2/2 : COPY . /app
 ---> 5cf6f5ae63eb
Successfully built 5cf6f5ae63eb
Successfully tagged acme/my-base-image:1.0
```

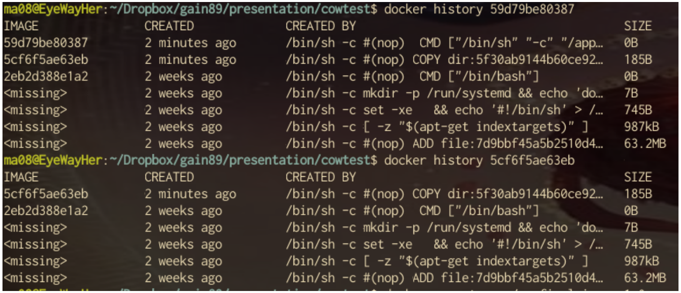


```
ma08@EyeWayHer:~/Dropbox/gain89/presentation/cowtest$ docker build -t acme/my-final-image:1.0 -f Dockerfile
Sending build context to Docker daemon  4.096kB
Step 1/2 : FROM acme/my-base-image:1.0
 ---> 5cf6f5ae63eb
Step 2/2 : CMD /app/hello.sh
 ---> Running in 4cbc6ecb1103
Removing intermediate container 4cbc6ecb1103
 ---> 59d79be80387
Successfully built 59d79be80387
Successfully tagged acme/my-final-image:1.0
```

cowtest



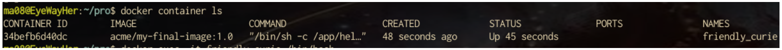
```
ma08@EyeWayHer:~/Dropbox/gain89/presentation/cowtest$ docker image ls
REPOSITORY             TAG              IMAGE ID         CREATED              SIZE
acme/my-final-image    1.0              59d79be80387     About a minute ago   64.2MB
acme/my-base-image     1.0              5cf6f5ae63eb     About a minute ago   64.2MB
```



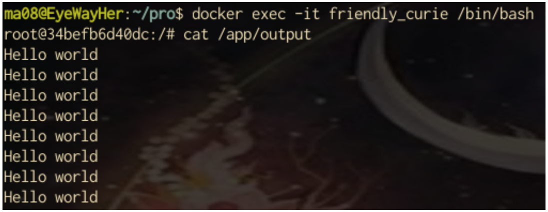
```
ma08@EyeWayHer:~/Dropbox/gain89/presentation/cowtest$ docker history 59d79be80387
IMAGE           CREATED          CREATED BY                                      SIZE
59d79be80387    2 minutes ago    /bin/sh -c #(nop)  CMD ["/bin/sh" "-c" "/app…   0B
5cf6f5ae63eb    2 minutes ago    /bin/sh -c #(nop) COPY dir:5f30ab9144b60ce92…   185B
2eb2d388e1a2    2 weeks ago      /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>       2 weeks ago      /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
<missing>       2 weeks ago      /bin/sh -c set -xe    && echo '#!/bin/sh' > /…  745B
<missing>       2 weeks ago      /bin/sh -c [ -z "$(apt-get indextargets)" ]     987kB
<missing>       2 weeks ago      /bin/sh -c #(nop) ADD file:7d9bbf45a5b2510d4…   63.2MB
ma08@EyeWayHer:~/Dropbox/gain89/presentation/cowtest$ docker history 5cf6f5ae63eb
IMAGE           CREATED          CREATED BY                                      SIZE
5cf6f5ae63eb    2 minutes ago    /bin/sh -c #(nop) COPY dir:5f30ab9144b60ce92…   185B
2eb2d388e1a2    2 weeks ago      /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>       2 weeks ago      /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
<missing>       2 weeks ago      /bin/sh -c set -xe    && echo '#!/bin/sh' > /…  745B
<missing>       2 weeks ago      /bin/sh -c [ -z "$(apt-get indextargets)" ]     987kB
<missing>       2 weeks ago      /bin/sh -c #(nop) ADD file:7d9bbf45a5b2510d4…   63.2MB
```

It is seen that only the top layer differs for the two images

Running cowtest



```
ma08@EyeWayHer:~/pro$ docker container ls
CONTAINER ID   IMAGE                     COMMAND                  CREATED          STATUS          PORTS     NAMES
34befb6d40dc   acme/my-final-image:1.0   "/bin/sh -c /app/hel…"   48 seconds ago   Up 45 seconds             friendly_curie
```



```
ma08@EyeWayHer:~/pro$ docker exec -it friendly_curie /bin/bash
root@34befb6d40dc:/# cat /app/output
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
```

Docker Containers

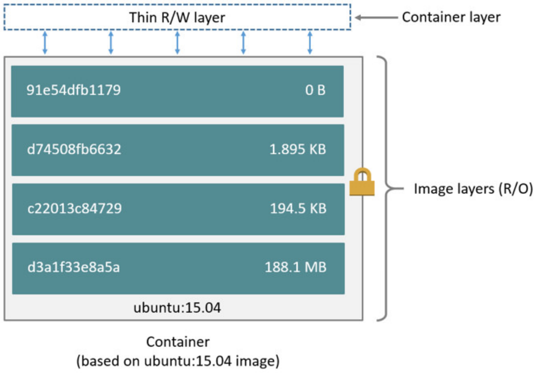
A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a containers network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Containers and layers

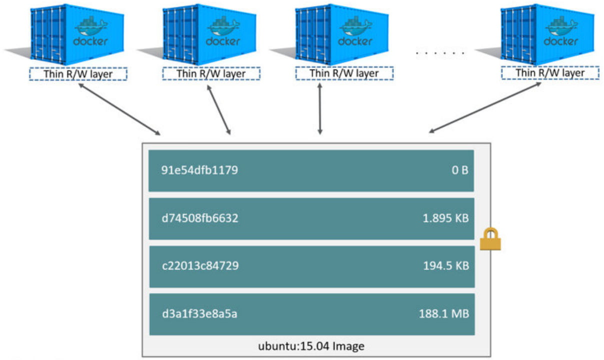
The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers(from the image). This layer is often called the container layer. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.



Containers and layers

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state.



The copy-on-write (CoW) strategy

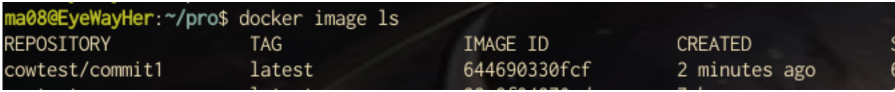
Copy-on-write is a strategy of sharing and copying files for maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file. The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers.

docker commit

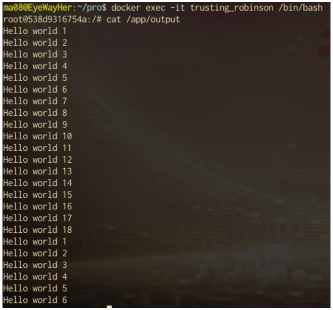
It can be useful to commit a containers file changes or settings into a new image. This allows you to debug a container by running an interactive shell, or to export a working dataset to another server.

Let's modify the cowtest code to play with docker commit. The new hello.sh

docker commit



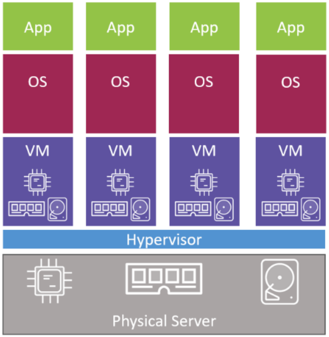
```
ma08@EyeWayHer:~/pro$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED
cowtest/commit1     latest              644690330fcf        2 minutes ago
```



```
ma08@EyeWayHer:~/pro$ docker exec -it trusting_robinson /bin/bash
root@538d9316754a:/# cat /app/output
Hello world 1
Hello world 2
Hello world 3
Hello world 4
Hello world 5
Hello world 6
Hello world 7
Hello world 8
Hello world 9
Hello world 10
Hello world 11
Hello world 12
Hello world 13
Hello world 14
Hello world 15
Hello world 16
Hello world 17
Hello world 18
Hello world 1
Hello world 2
Hello world 3
Hello world 4
Hello world 5
Hello world 6
```

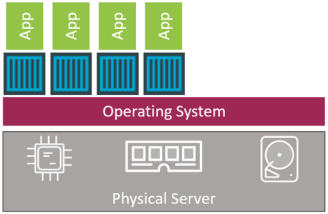
Virtual Machines vs Containers

In the VM model, the physical server is powered on and the hypervisor boots (were skipping the BIOS and bootloader code etc.). Once the hypervisor boots, it lays claim to all physical resources on the system such as CPU, RAM, storage, and NICs. The hypervisor then carves these hardware resources into virtual versions. It then packages them into a software construct called a virtual machine (VM). We then take those VMs and install an operating system and application on each one. At a high level, we can say that hypervisors perform hardware virtualization they carve up physical hardware resources into virtual versions.



Virtual Machines vs Containers

When the server is powered on, your chosen OS boots. In the Docker world this can be Linux, or a modern version of Windows that has support for the container primitives in its kernel. Similar to the VM model, the OS claims all hardware resources. On top of the OS, we install a container engine such as Docker. The container engine then takes OS resources such as the process tree, the filesystem, and the network stack, and carves them up into secure isolated constructs called containers. Each container looks smells and feels just like a real OS. Inside of each container we can run an application. Like before, were assuming a single physical server with 4 applications. Therefore, wed carve out 4 containers and run a single application inside of each. Containers perform OS virtualization, they carve up OS resources into virtual versions.



The VM Tax

The VM model then carves low-level hardware resources into VMs. Each VM is a software construct containing virtual CPU, virtual RAM, virtual disk etc. As such, every VM needs its own OS to claim, initialize, and manage all of those virtual resources. And sadly, every OS comes with its own set of baggage and overheads. For example, every OS consumes a slice of CPU, a slice of RAM, a slice of storage etc. Most need their own licenses as well as people and infrastructure to patch and upgrade them. Each OS also presents a sizable attack surface. We often refer to all of this as the OS tax, or VM tax every OS you install consumes resources!

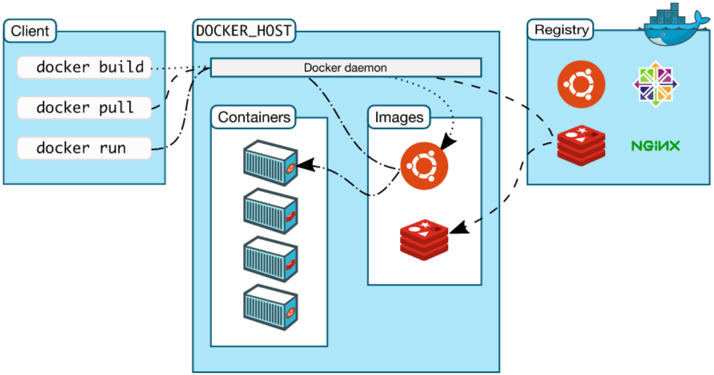
The container model has a single kernel running in the host OS. Its possible to run tens or hundreds of containers on a single host with every container sharing that single OS/kernel. That means a single OS consuming CPU, RAM, and storage. A single OS that needs licensing. A single OS that needs upgrading and patching. And a single OS kernel presenting an attack surface. All in all, a single OS tax bill!

The VM Tax

Another thing to consider is start times. Because a container isnt a full-blown OS, it starts much faster than a VM. Remember, theres no kernel inside of a container that needs locating, decompressing, and initializing not to mention all of the hardware enumerating and initializing associated with a normal kernel bootstrap. None of that is needed when starting a container! The single shared kernel, down at the OS level, is already started! Net result, containers can start in less than a second. The only thing that has an impact on container start time is the time it takes to start the application its running. This all amounts to the container model being leaner and more efficient than the VM model. We can pack more applications onto less resources, start them faster, and pay less in licensing and admin costs

Understanding the overall framework

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.



Docker Registries

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.