Docker with Kubernetes + Swarm Container Images: Where To Find Them and How To Build Them

Table of Contents

1. What's in an Image? (And What Isn't)

- An image contains the binaries and the dependencies for the application and the metadata on how to run it.
- Official definition: "An image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime."
- Inside the image, there's **not** actually a complete OS. There's **no** kernel, kernel modules (e.g. drivers).
- An image can be as small as one file (the app binary) like a golang static binary
- Or it could be as big as a Ubuntu distro with apt, and Apache, PHP, and more installed.

2. Images and Their Layers

It uses something called the union file system to present a series of file system changes as an actual system.

Let's get a list of images on our system cache:

- ***`docker image ls`***

For the same image ID, we can have different tags. Images are however recognized by their IDs.

Let's do a quick docker image history on nginx:

- ***docker image history [OPTIONS] IMAGE***

This does not list the things that have happened in the container, but it actually is a history of the image layers.

Every image starts from a blank layer known as scratch. Then every set of changes that happens after that on the file system, in the image, is another layer.

Some changes may involve a simple metadata change, whereas some may involve big data changes.

```
(base) hetanshmehta@Hetanshs-MacBook-Pro ~ % docker image history nginx
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
4392e5dad77d        4 weeks ago         /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon…   0B
<missing>           4 weeks ago         /bin/sh -c #(nop)  STOPSIGNAL SIGTERM           0B
<missing>           4 weeks ago         /bin/sh -c #(nop)  EXPOSE 80                    0B
<missing>           4 weeks ago         /bin/sh -c #(nop)  ENTRYPOINT ["/docker-entr…   0B
<missing>           4 weeks ago         /bin/sh -c #(nop) COPY file:cc7d4f1d03426ebd…   1.04kB
<missing>           4 weeks ago         /bin/sh -c #(nop) COPY file:b96f664d94ca7bbe…   1.96kB
<missing>           4 weeks ago         /bin/sh -c #(nop) COPY file:d68fadb480cbc781…   1.09kB
<missing>           4 weeks ago         /bin/sh -c set -x     && addgroup --system -…   62.9MB
<missing>           4 weeks ago         /bin/sh -c #(nop)  ENV PKG_RELEASE=1~buster     0B
<missing>           4 weeks ago         /bin/sh -c #(nop)  ENV NJS_VERSION=0.4.1        0B
<missing>           4 weeks ago         /bin/sh -c #(nop)  ENV NGINX_VERSION=1.19.0     0B
<missing>           6 weeks ago         /bin/sh -c #(nop)  LABEL maintainer=NGINX Do…   0B
<missing>           6 weeks ago         /bin/sh -c #(nop)  CMD ["bash"]                 0B
<missing>           6 weeks ago         /bin/sh -c #(nop) ADD file:7780c81c33e6cc5b6…   69.2MB
(base) hetanshmehta@Hetanshs-MacBook-Pro ~ %
```

- ***`docker image inspect [OPTIONS] IMAGE [IMAGE…]`***

Display detailed information on one or more images. Returns JSON metadata about the image. Besides the basic info (image ID, tags), we get all sorts of details around how this image expects to be run. Gives option to expose certain ports inside the image and handle environment variables.

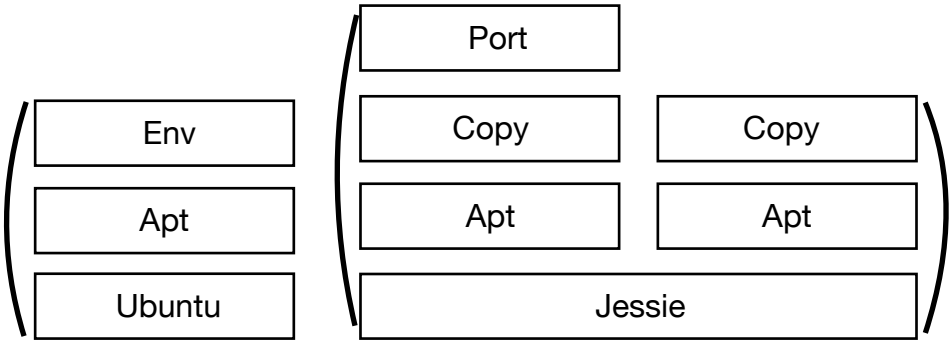
```
(base) hetanshmehta@Hetanshs-MacBook-Pro ~ % docker image inspect nginx
[
    {
        "Id": "sha256:4392e5dad77dbaf6a573650b0fe1e282b57c5fba6e6cea00a27c7d4b68539b81",
        "RepoTags": [
            "nginx:latest"
        ],
        "RepoDigests": [
            "nginx@sha256:c870bf53de0357813af37b9500cb1c2ff9fb4c00120d5fe1d75c21591293c34d"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2020-06-02T16:23:40.499754581Z",
        "Container": "d3a0a2b94e92ed2d1733f8f9a8864a412f3ff39ac9f3f9f27eaca81e673f3a76",
        "ContainerConfig": {
            "Hostname": "d3a0a2b94e92",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "80/tcp": {}
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "NGINX_VERSION=1.19.0",
                "NJS_VERSION=0.4.1",
                "PKG_RELEASE=1~buster"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"nginx\" \"-g\" \"daemon off;\"]"
            ],
            "ArgsEscaped": true,
            "Image": "sha256:5b52d99b7985627947b3a79d9858507d546b0c4f09ba72e21a75a1bcd14d5d1e",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": [
                "/docker-entrypoint.sh"
            ],
            "OnBuild": null,
            "Labels": {
                "maintainer": "NGINX Docker Maintainers <docker-maint@nginx.com>"
            },
            "StopSignal": "SIGTERM"
        },
```

2.1. Visualizing Layers

When we start an image, i.e., when we create a new image, we're starting with one layer. Every layer gets its own unique SHA that helps the system identify if that layer is indeed the same as another layer.

All other images can access the layers from cache and build something on top of it - saving a lot of space and time. As the layers have a unique SHA, it's guaranteed to be the exact layer it needs to add/remove.

If we decide that we want to have the same image to be the base image for more layers, then it's only every storing one copy of each layer.



For instance, let's say we have Ubuntu at the very bottom as the first layer.Then we create a Dockerfile, which adds some more files and that's another layer on top of that image (We may us `apt` for that).Then we also add an environment variable change (`env`) which completes our image.

We might have a different image that starts from `debian:jessie` and then on that image we may use `apt` to install some stuff - like MySQL, we may copy some file over, open a port, etc.

If we have another image that's also using the same version of jessie, it can have its own changes on top of the layer that we have in our cache. This is where the fundamental concept of cache of images help us save a lot of time and space.

2.2. Container Layers

When we run a container off of an image, all Docker does is, it creates a **new read/write layer** for that container on top of the image.

If we ran two containers at the same time off of the same image, container 1 and container 2 would only be showing, *the **difference** in file space between what's happened on that live container running and what's happening in the base image (which is read-only).*

When we use a container to change a file which is in the image, the file system will take that file out of the image and copy it into the differencing, and store a copy of that file in the container layer. This is known as **copy-on-write (COW)**.