Deep dive into containers with Docker

CS695 – Topics in Virtualization and Cloud Computing

Debojeet Das

Why containers?

- Share the host OS kernel, eliminating the need for full OS replication per application : efficient
- Package applications with all dependencies : “Build Once, Run Anywhere”
- Start in seconds (vs minutes for VMs) : “Quick Deployment”
- Ideal for microservices, serverless, and cloud-native applications

[Image Credit - https://dev.to/ben/meme-monday-1o5g#comment-22ekf]

Docker Terminologies

- Dockerfile: (Like source code) List of instructions to build an image
- Docker image: (Like compiled binary)
- Docker container: (Like running process) Runtime instance of an image
- Docker registry: (Like GitHub) Repository or store of images
- Docker engine: The docker daemon process running on the host which manages images and containers

```
$ docker info
```

Docker is a server-client application. The docker engine (server) implements the container management and exposes HTTP API for communication which is used by docker CLI (client).

Docker Terminologies

```
Client: Docker Engine - Community
 Version:    26.0.0
 Context:    default
 Debug Mode: false
 Plugins:
  buildx: Docker Buildx (Docker Inc.)
    Version:  v0.13.1
    Path:     /usr/libexec/docker/cli-plugins/docker-buildx
  compose: Docker Compose (Docker Inc.)
    Version:  v2.25.0
    Path:     /usr/libexec/docker/cli-plugins/docker-compose
```

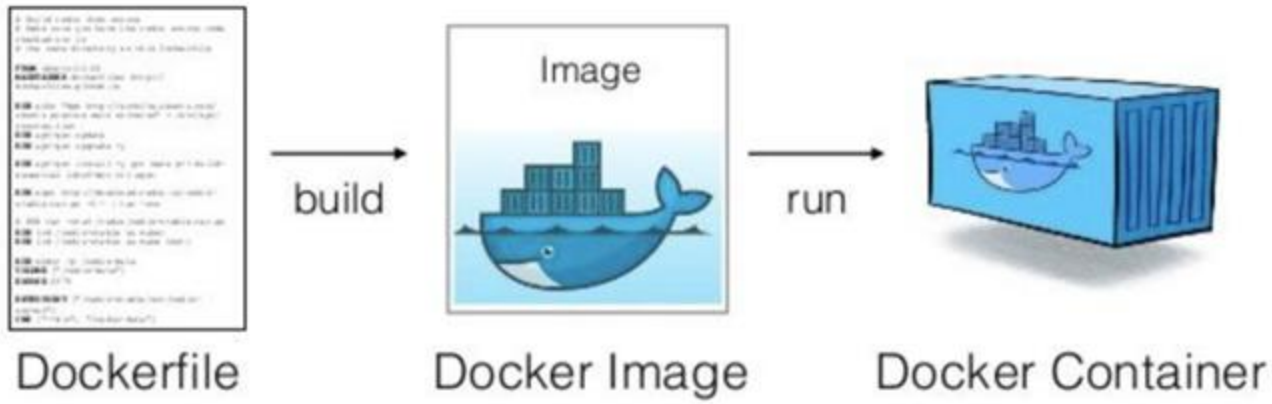
Client Details

```
Server:
 Containers: 2
  Running: 1
  Paused: 0
  Stopped: 1
 Images: 13
 Server Version: 26.0.0
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Using metacopy: false
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: systemd
 Cgroup Version: 2
```

Server Details

Cgroup version 2 is being used here

Docker Images



Container images can either be built locally or “pulled” from a registry (which was built by someone).

Let’s try to run a container by pulling a docker image first.

We will use a docker image based on Alpine Linux with a complete package index and only 5 MB in size!

```
$ docker pull alpine:3.18
```

```
$ docker image inspect alpine:3.18
```

[image taken from ACM India Winter School on "Full-stack Networking (FSN)"]
[image registry - https://hub.docker.com/]

Docker containers and its hidden details

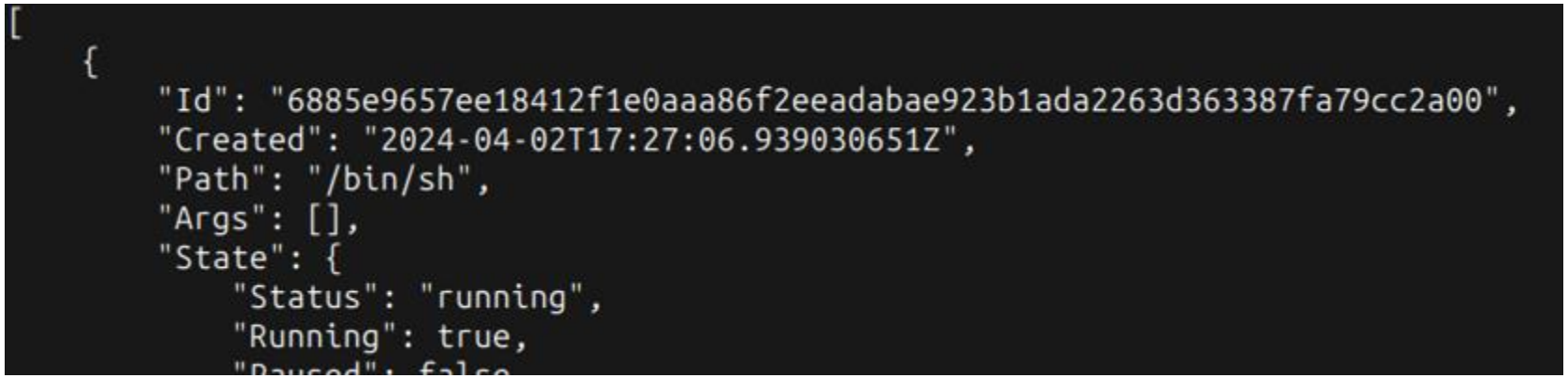
Let’s run a container and understand its internals!

```
$ docker run -it [--name <container-name>] alpine:3.18
```

```
$ docker ps
```

```
CONTAINER ID      IMAGE          COMMAND      CREATED             STATUS             PORTS     NAMES
6885e9657ee1      alpine:3.18.   "/bin/sh"    23 minutes ago      Up 23 minutes                test
```

```
$ docker inspect <CONTAINER ID>or<NAME>
```



Docker containers and its hidden details

Interesting details

```
$ docker inspect <CONTAINER ID>or<NAME>
```

```
"ResolvConfPath": "/var/lib/docker/containers/6885e9657ee18412f1e0aaa86f2eeadabae923b1ada2263d363387fa79cc2a00/resolv.conf",
"HostnamePath": "/var/lib/docker/containers/6885e9657ee18412f1e0aaa86f2eeadabae923b1ada2263d363387fa79cc2a00/hostname",
"HostsPath": "/var/lib/docker/containers/6885e9657ee18412f1e0aaa86f2eeadabae923b1ada2263d363387fa79cc2a00/hosts",
```

System configuration files

```
"LowerDir": "/var/lib/docker/overlay2/b0cfc7f88dcd15158b5c80b95fc68da62ff436f28d683f2483882e7fa82ee04e-init/diff
"MergedDir": "/var/lib/docker/overlay2/b0cfc7f88dcd15158b5c80b95fc68da62ff436f28d683f2483882e7fa82ee04e/merged",
"UpperDir": "/var/lib/docker/overlay2/b0cfc7f88dcd15158b5c80b95fc68da62ff436f28d683f2483882e7fa82ee04e/diff",
"WorkDir": "/var/lib/docker/overlay2/b0cfc7f88dcd15158b5c80b95fc68da62ff436f28d683f2483882e7fa82ee04e/work"
```

OverlayFS

```
"NetworkSettings": {
    "Bridge": "",
    "SandboxID": "99e89b1463600362f0d686af8a4984f4c4c5c0194d8d35c78e4bbee8a7a6fa09",
    "SandboxKey": "/var/run/docker/netns/99e89b146360",
```

Network settings

Docker containers and its hidden details

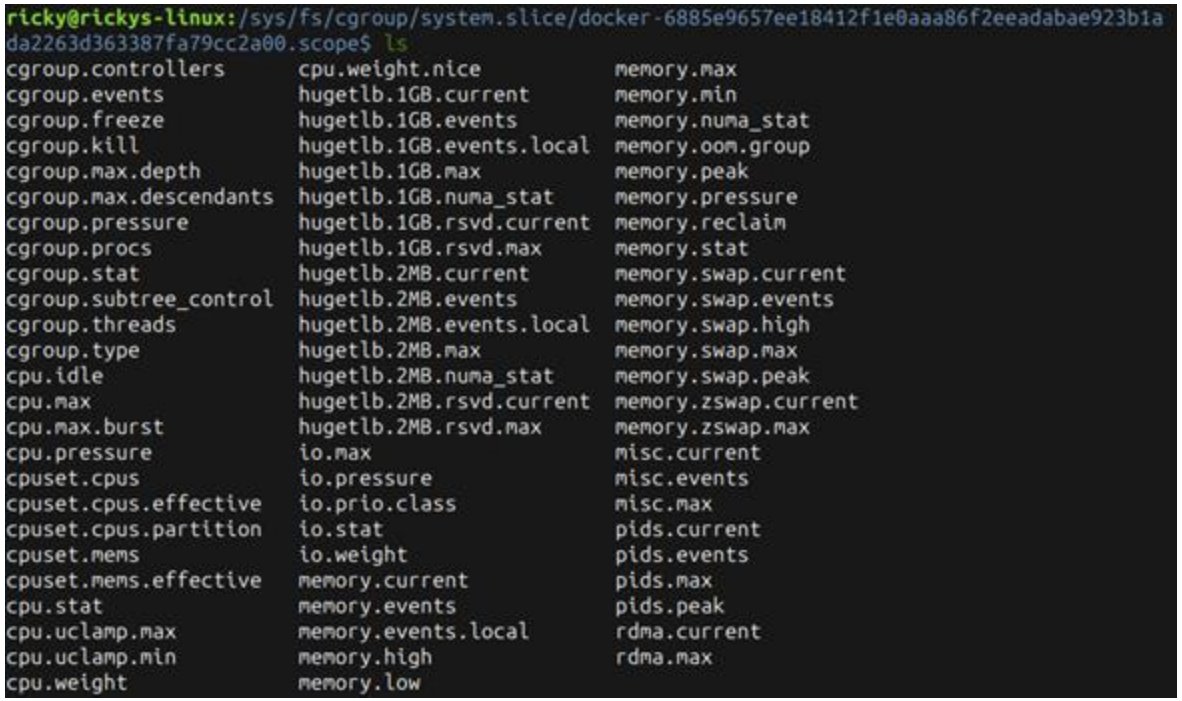
cgroup

```
$ cd /sys/fs/cgroup/cpu/docker/<container-id>
```

For cgroup v1

```
$ cd /sys/fs/cgroup/system.slice/docker-<container-id>.scope
```

For cgroup v2



Docker containers and its hidden details

Let's kill the container

```
$ docker rm <name>
$ docker stop <name>
$ docker inspect <name>
$ docker rm <name>
$ docker inspect <name>
```

```
Error response from daemon: You cannot
remove a running container e036efae...
Stop the container before attempting
removal or force remove
```

```
"State": {
    "Status": "exited",
    "Running": false,
    "Paused": false,
    "Restarting": false,
}
```

```
[]
Error: No such object: <name>
```

Docker containers and its hidden details

We saw the container in running status and exited status. What is this status?
Container is an instance of an image with a process running. Status is the state of that process.

1. Created - Not started, no CPU or memory is used.
   a. Using `docker create`
2. Running - Process is running
3. Exited - Process terminates, no CPU or memory used
   a. Naturally - ML training job
   b. Manually - `docker stop`
   c. Error - code panic
4. Restarting `docker run --restart=always centos:7 sleep 5`
   a. By default if command finishes, container exits. But, if restart policy is always, container restarts
5. Paused - Process is suspended, CPU is released, memory is consumed
   a. `docker pause <name>`
   b. `docker unpause <name>` - resumes the container from where it stopped
6. Removing - In the process of being removed
   a. `docker rm`

DIY:
Try running docker stats command in each of these status to examine the CPU and memory usage


Me After Setting A
Docker Container Up:
I'm something of a DevOps Engineer Myself

Docker Build

Build is a key part of container software development life cycle allowing us to package and bundle our code and ship it anywhere

Pulling vs building an image

When to pull?

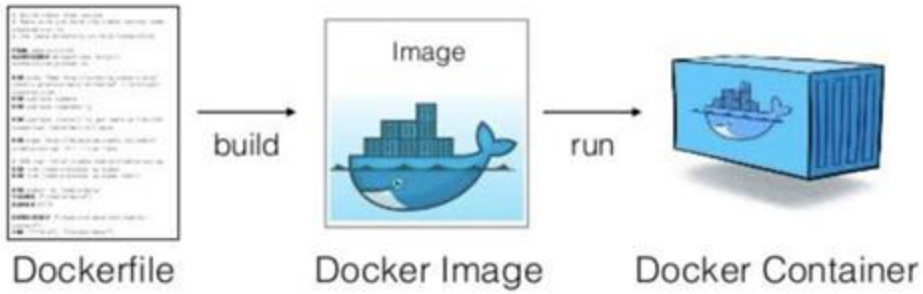
- When using someone's created image:
  - If you want to play with python, get a python image
- To access your own created image
  - Create image, push it to a registry and the pull it from elsewhere

When to build?

- To create an environment/recipe for sharing/running deterministically
- For creating any application for running on the cloud

Docker Build

Build is a key part of container software development life cycle allowing us to package and bundle our code and ship it anywhere




```
FROM memcached:1.6
RUN apt update && apt install -y stress-ng
```

Networking experiments

```
FROM ubuntu:jammy
# Install golang
RUN cd downloads \
    && wget https://go.dev/dl/go1.19.1.linux-amd64.tar.gz \
    && tar -C /usr/local -xvf go1.19.1.linux-amd64.tar.gz
ENV PATH $PATH:/usr/local/go/bin
```

k8s network simulator

Docker Build

Basic build commands

Command	Description
FROM *image* | scratch	Use a pre-existing docker image as a base image for the build
COPY *path dst*	Add files to the image. Copy from *path* in host into container at *dst*
RUN *args...*	Run arbitrary commands inside the container
WORKDIR *path*	Set the default working directory
ENV *name value*	Set an environment variable
ENTRYPOINT/CMD ["executable", "param1", "param2"]	Set the command to execute (when the container starts)

Task - Create Docker image

Goal: Create a “Hello World” application for container using Flask running on ubuntu.

1. Create `hello.py` with the following lines:
2. To run this in baremetal you will need to install python3 and flask and then run the application.

Similarly the Dockerfile should create a container image, which has all the dependencies installed and that automatically starts the application.

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"
```

https://www.cse.iitb.ac.in/~debojeetdas/ta/hello.py

Task - Create Docker image

```
FROM ubuntu:22.04

# install app dependencies


# copy the flask app


# final configuration and running the application
```

Task - Create Docker image

```
FROM ubuntu:22.04

# install app dependencies
RUN apt-get update && apt-get install -y python3 python3-pip
RUN pip install flask==3.0.*

# copy the flask app
COPY hello.py /

# final configuration and running the application
ENV FLASK_APP=hello
CMD ["flask", "run", "--host", "0.0.0.0", "--port", "8000"]
```

Task - Create Docker image

```
FROM ubuntu:22.04

# install app dependencies
RUN apt-get update && apt-get install -y python3 python3-pip
RUN pip install flask==3.0.*

# copy the flask app
COPY hello.py /

# final configuration and running the application with exposed port
ENV FLASK_APP=hello
EXPOSE 8000
CMD ["flask", "run", "--host", "0.0.0.0", "--port", "8000"]
```

https://www.cse.iitb.ac.in/~debojeetdas/task/Dockerfile

Let’s build and run your container

1. Build the docker image
   $ docker build -t test:latest .
2. See if the image is present and run it.
   $ docker images
   $ docker run -p 127.0.0.1:8000:8000 test:latest

Go to terminal and do curl to 127.0.0.1:8000 to see the application in action.

If you have docker hub account you can push the image just like git.

1. Docker login to registry
   $ docker login --username username
2. Rename/Tag your image
   $ docker tag my-image username/my-repo
3. Push the image
   $ docker push username/my-repo

Docker Compose

Docker Compose is a tool for defining and running multi-container applications. (Just like task 4)

You specify multiple docker containers and it brings them all up.

It sets up a single network for your entire application, all containers join them and can reach each other on this network.

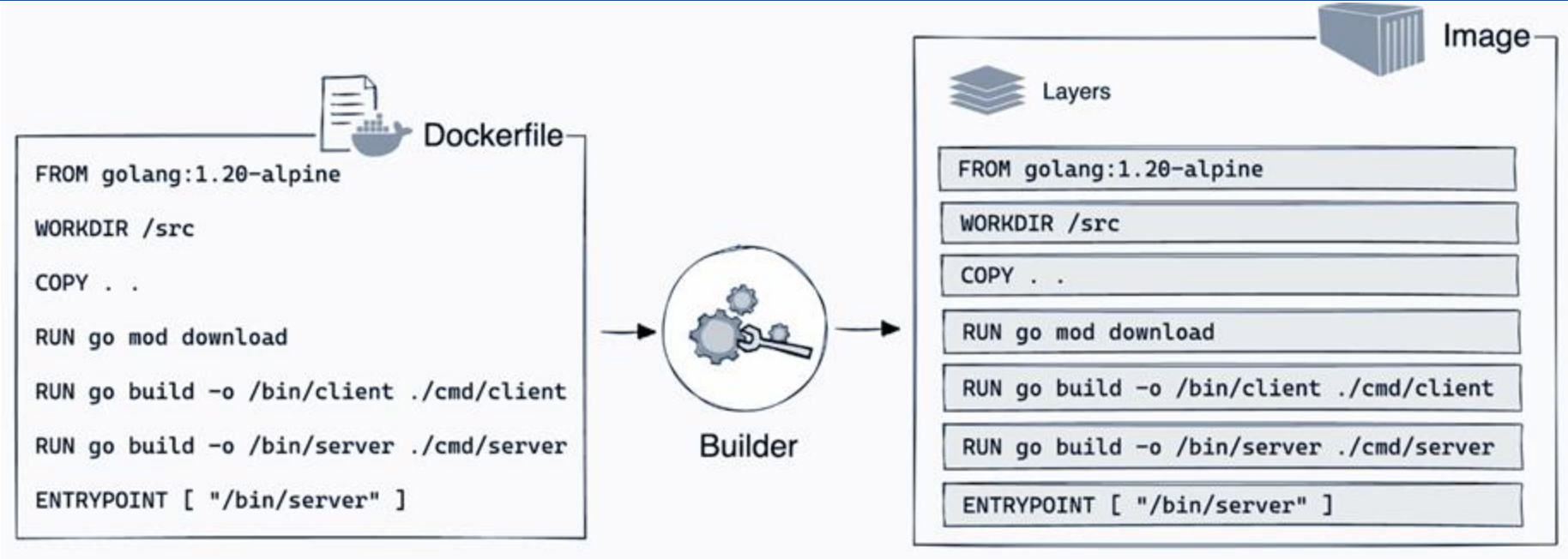
Checkout a simple example – https://www.cse.iitb.ac.in/~debojeetdas/ta/compose.tar.gz

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

```
$docker compose up
```

(to setup container deployments specified in the docker compose yaml file)

Docker Internals - Layers



Docker image is built as a series of layers, each layer represents a line in the Dockerfile.

- Every command that modifies the filesystem is a new layer.
- The layer only captures the diff from the previous layer.
- Layers are shared across different images.

Docker Internals - Layers

How many layers will be here?

1. `FROM ubuntu:22.04`

```
# install app dependencies
```

2. `RUN apt-get update && apt-get install -y python3 python3-pip`
3. `RUN pip install flask==3.0.*`

```
# copy the flask app
```

4. `COPY hello.py /`

5. `RUN rm -r $HOME/.cache`

```
# final configuration and running the application with exposed port
```

x. `ENV FLASK_APP=hello`

x. `EXPOSE 8000`

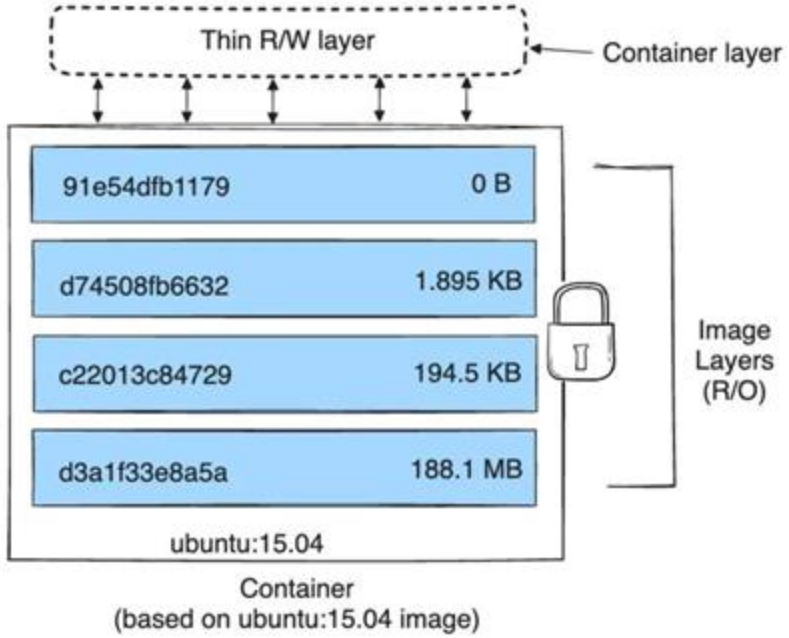
x. `CMD ["flask", "run", "--host", "0.0.0.0", "--port", "8000"]`

Docker Internals - Layers

Docker image is built as a series of layers, each layer represents a line in the Dockerfile.

- When we run a container, a new writable layer (called container layer) is created. Other layers are read-only.
- The difference between a container and an image is in this writable layer
- When a container is deleted, the container layer is also deleted
- Multiple containers can share the same base image and have their own state in the container layer

<u>Copy-on-Write (COW) Strategy</u>
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.



 [image taken from docs.docker.com]

Understanding OverlayFS

OverlayFS is a union filesystem. Docker uses overlay2 storage driver as its storage driver. (default)
It may run on top of XFS or ext4

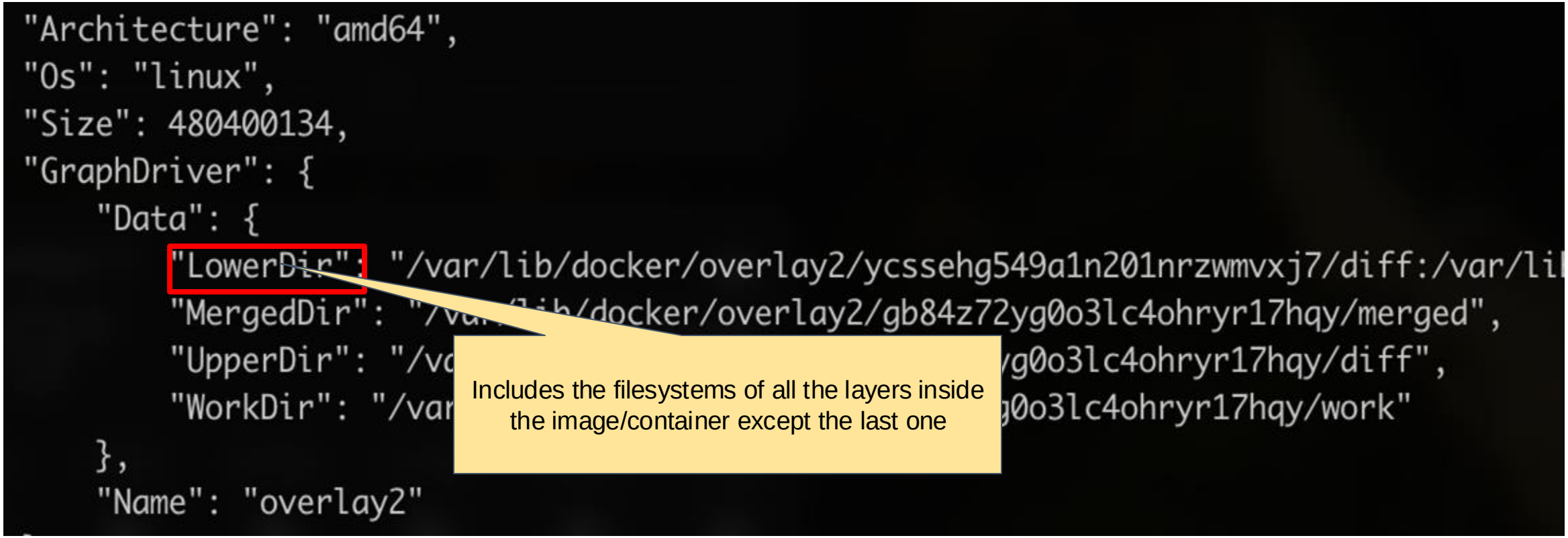
```
$ docker build -t hellocs695:latest .
$ docker images
$ docker image inspect hellocs695:latest
```

A n-layer image, will create n+1 directories for the container
- The layer is mounted when container is started using mount command.

The directories are stored in `/var/lib/docker/overlay2`
They contain the following subdirectories:
- **diff**: layer's contents.
- **link**: shortened identifier for mounting.
- **lower**: its parent. (not available in lowest layer)
- **work**: used internally by OverlayFS (not available in lowest layer)
- **merged**: unified contents (not available in image layers)

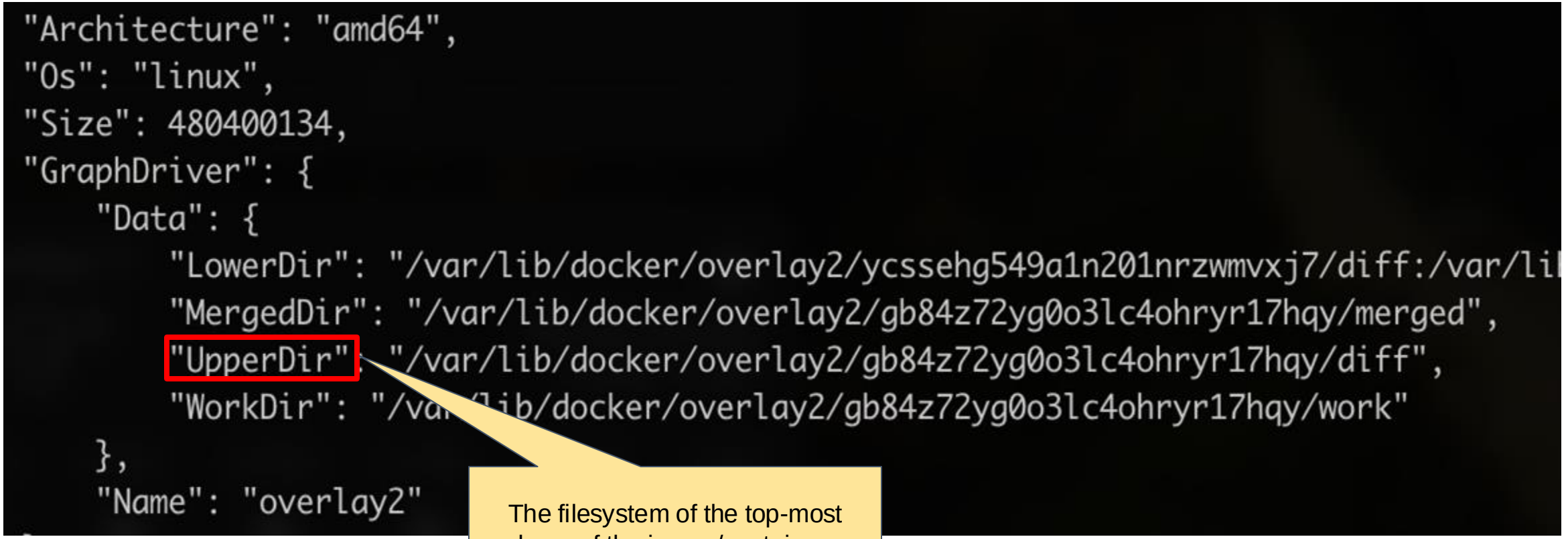
Understanding OverlayFS

```
$ docker image inspect hellocs695:latest
```



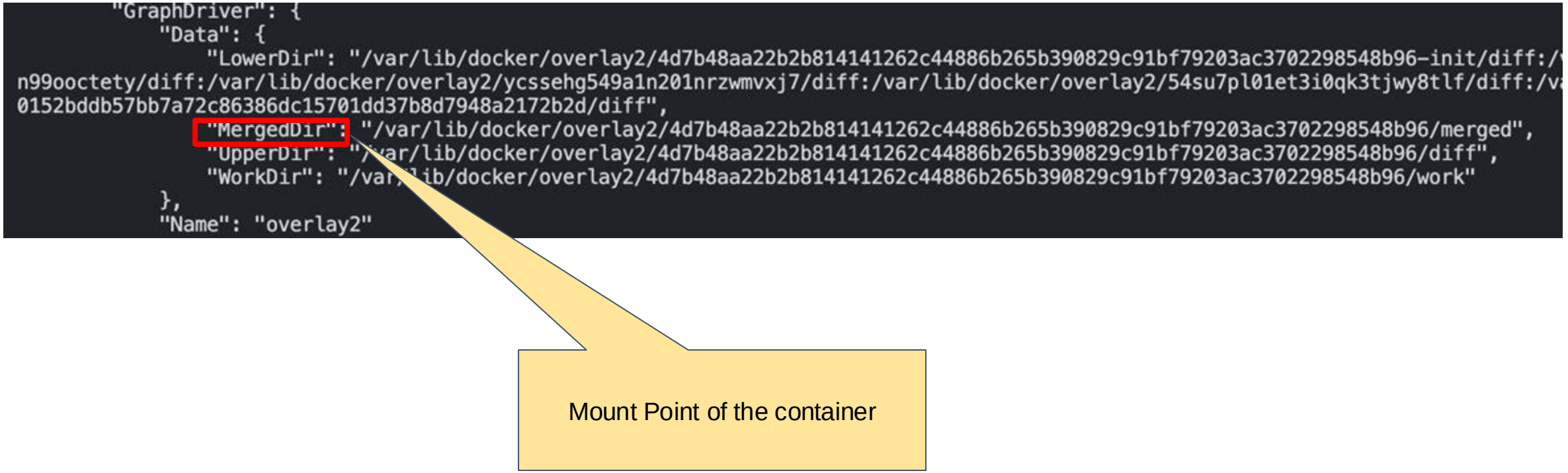
Understanding OverlayFS

```
$ docker image inspect hellocs695:latest
```



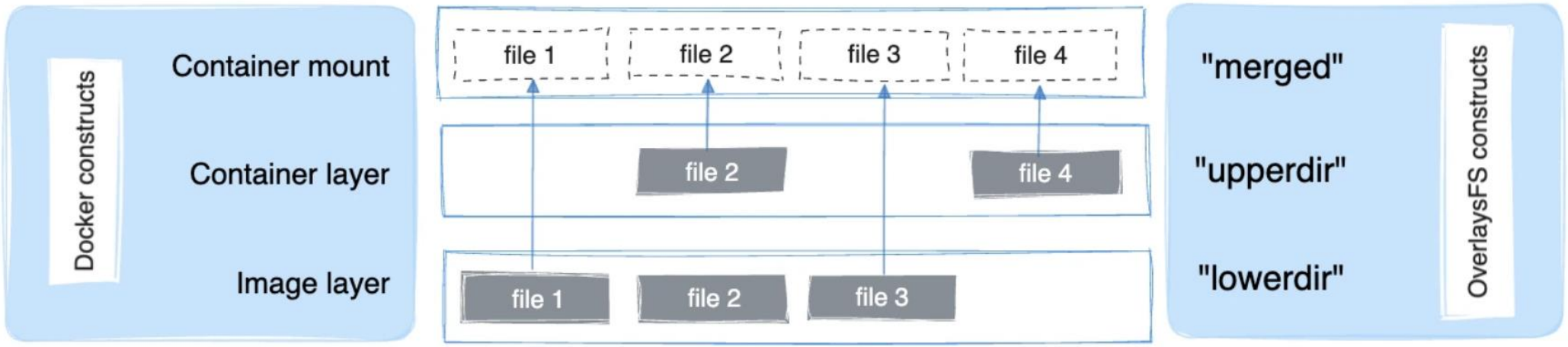
Understanding OverlayFS

```
$ docker run -p 80:8000 --name cont-test hellocs695:latest
$ docker ps
$ docker inspect cont-test
```



Understanding OverlayFS

```
$ docker run -p 80:8000 --name cont-test hellocs695:latest
$ docker ps
$ docker inspect cont-test
$ mount | grep overlay
```



 [image taken from docs.docker.com]

Understanding OverlayFS operations

File Read:

- Scenario 1: The file does not exist in the container layer
Read from the image (lowerdir). Low overhead

- Scenario 2: The file only exists in the container layer
Read directly from the container. No overhead

- Scenario 3: The file exists in both the container layer and the image layer
File's version in the container layer is read. No overhead

Understanding OverlayFS operations

File Write (First Time):

- CoW comes into play. The file is first copied into the container layer then modified.

However, OverlayFS works at the file level rather than the block level.
What is the implication?

Again, OverlayFS works with multiple layers.
What can be a side effect?

Understanding OverlayFS operations

File Delete

- When a file is deleted within a container, a whiteout file is created in the container. No change in image.

- When a directory is deleted within a container, an opaque directory is created within the container. No change in the image.

Renaming Directory

- If the directory belongs to container it is allowed. If it belongs to the image throws an error.

There are other storage drivers like **BTRFS** which operates on block level.

Thank You!!