**Trimming Down Docker Image size**

*Shipping lightweight Docker images to production environments is the standard practice in the industry.*

Before diving into reducing image size, you need to know a few terms before that. For clarity of course!

### **Container Image and Container**

The image is like a recipe that specifies what ingredients and steps are needed to create a dish, while the container is like the dish itself that is ready to be served and consumed.

Image provides the blueprint for the container, while the container provides the actual execution environment for the application or service.

You don't say "I'll create a container", rather you say "I'll create a container image" and then you say "I'll start a container using the image".

*Containers are like bagpack.....*

*"pack your stuff in a bagpack and take it anywhere."*

*"pack your application in containers and deploy anywhere"*

### **Container Registry**

When you create an image, (its just a bunch of files), you can store it in a centralised file server so that it is available for others to use. Or you can pull an images from the registry and start containers on your local system. This central file server is called the Registry.

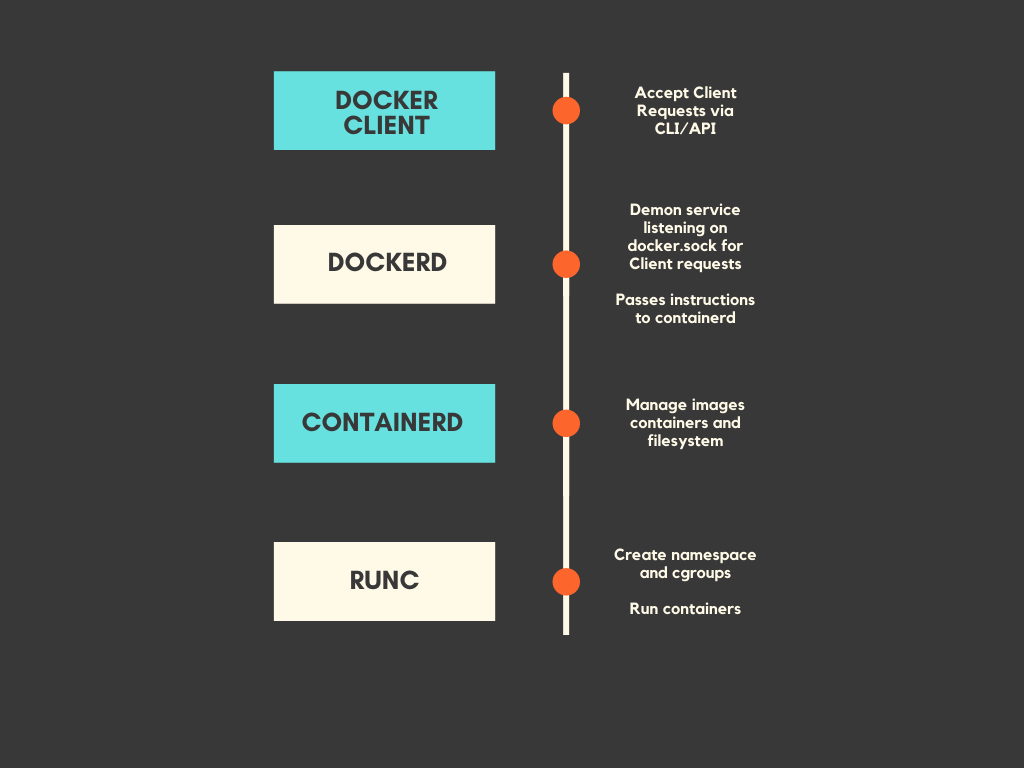
### **Container Runtime and Container Engine**

If containers are processes, they are bound to have a process lifecycle. And Container Runtimes step in to handle the container lifetime i.e. creating containers, running them and monitoring and managing containers and destroying them.

For many out there, Docker is synonymous with everything containers.

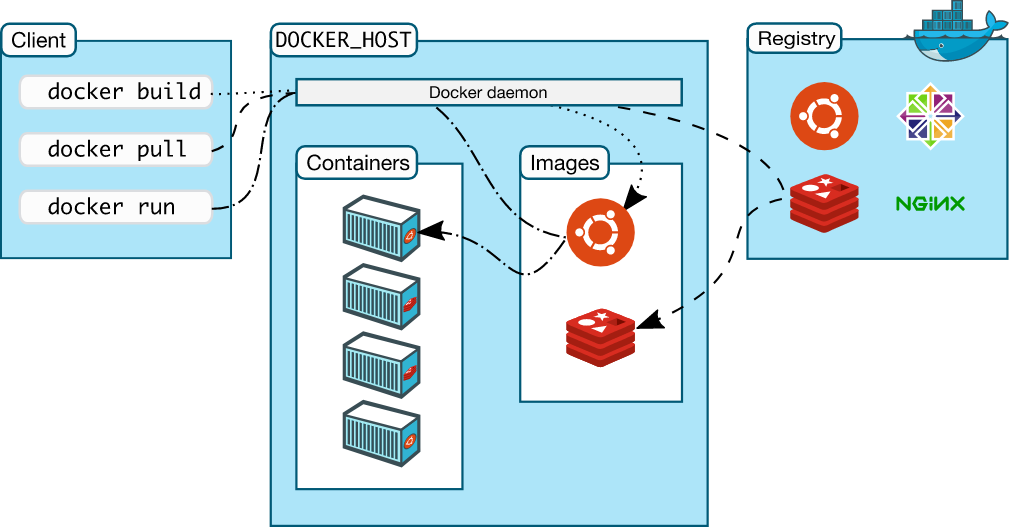
## **Meet Docker!**

Docker overall follows a simple client-server paradigm. There are four main components:



Docker today has now become the de facto technology for deploying containerised applications. In fact, it was one the primary forces behind the cloud boom.

*The entire source code for Docker (including containerd and runc) is written in Go.*



Docker Images are a great way to package software applications along with their libraries, tools, files, and other dependencies and run them as lightweight containers.

A Docker image is made up of layers. and with every layer you add on, more space will be taken up by the image. Therefore, the more layers in the image, the more space the image will require.

**Enter Dive!**

Dive is a tool by Alex Goodman(<https://github.com/wagoodman/dive>). It is used for exploring a docker image, layer contents, and discovering ways to shrink the size of your Docker/OCI image.

* Breaks down the image contents in the Docker image layer by layer
* Indicate what’s changed in each layer
* Shows the total size of the image
* You can build a Docker image and do an immediate analysis with one command: dive build -t some-tag .
* Estimate “image efficiency”
* CI Integration

**How to install**

*Ubuntu/Debian*

wget https://github.com/wagoodman/dive/releases/download/v0.9.2/dive\_0.9.2\_linux\_amd64.deb

sudo apt install ./dive\_0.9.2\_linux\_amd64.deb

*RHEL/Centos*

curl -OL https://github.com/wagoodman/dive/releases/download/v0.9.2/dive\_0.9.2\_linux\_amd64.rpm

rpm -i dive\_0.9.2\_linux\_amd64.rpm

*Mac*

*Homebrew*

brew install dive

*MacPorts*

sudo port install dive

*Docker*

docker pull wagoodman/dive

or

docker pull quay.io/wagoodman/dive

When running you'll need to include the docker socket file:

docker run --rm -it \

-v /var/run/docker.sock:/var/run/docker.sock \

wagoodman/dive:latest <dive arguments...>

**Usage example**

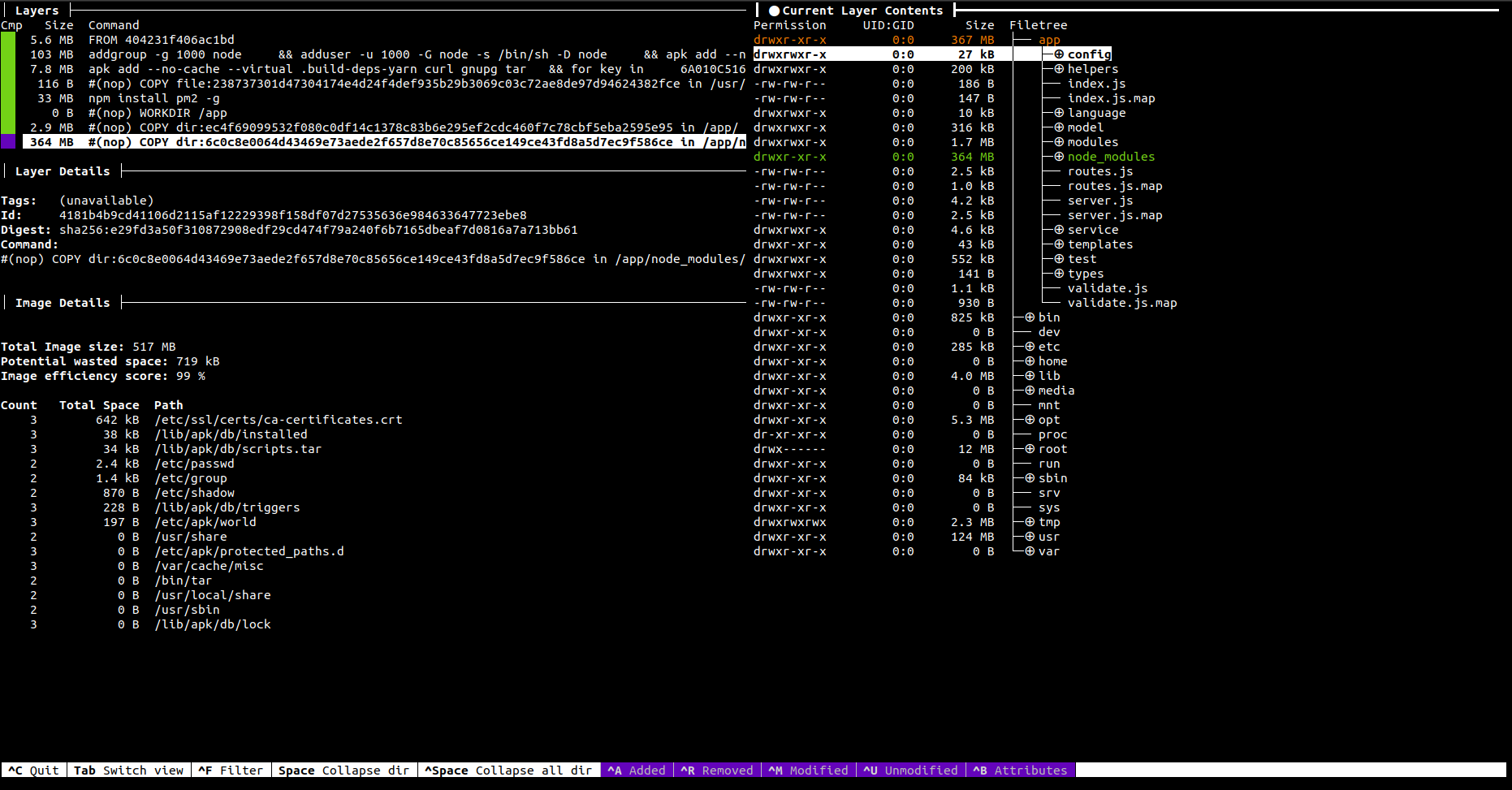
To analyze a Docker image run dive with:

dive <your-image-tag>

if you want to build your image then analyzing it with:

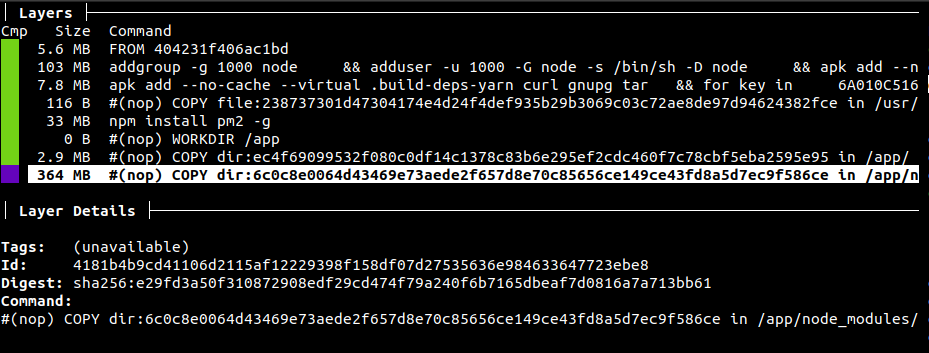
dive build -t <some-tag> .

**DIve screen**

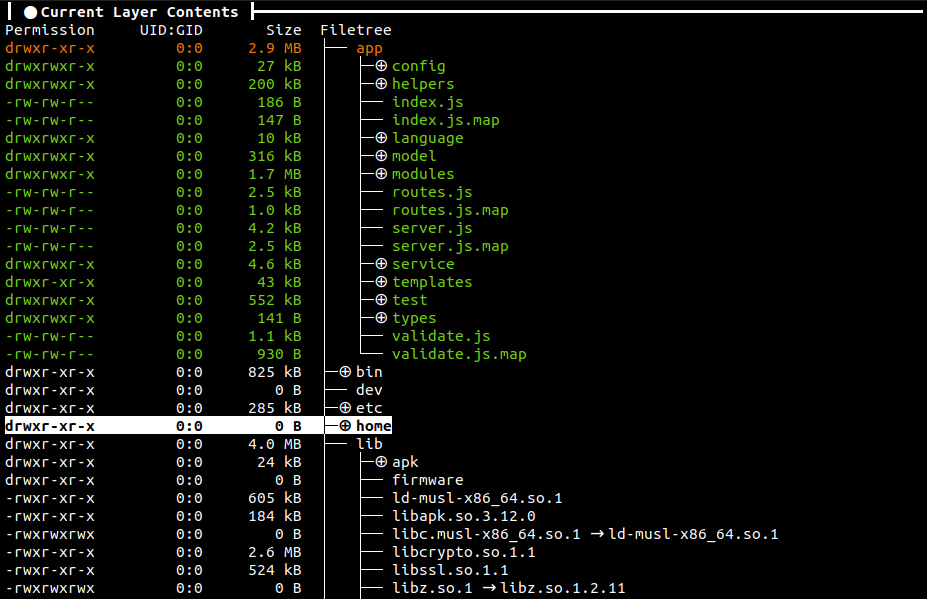
****

dive node-multistage:latest

When you run the dive command on a docker file what you see is something like this on the terminal, i.e. analysis result. The screen is split into 3 parts.



At the top left, we can go through each layer created in our Docker Image.



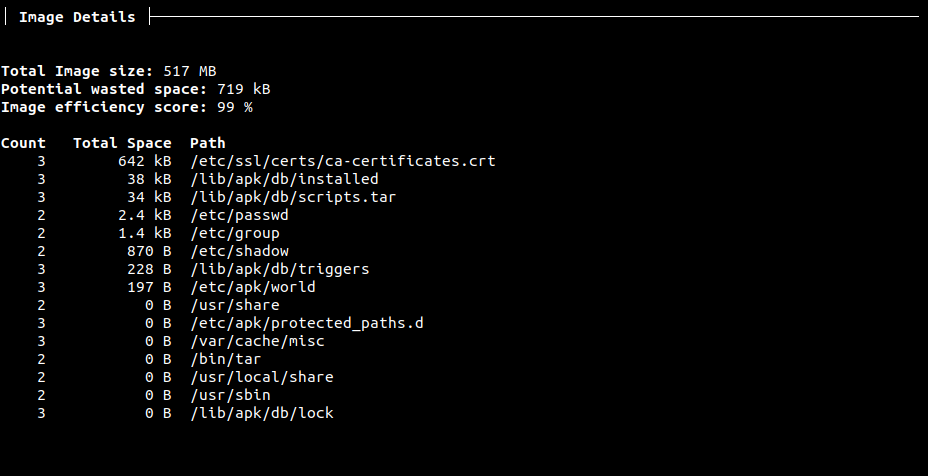
On the right pane, we can see all the files that are present on the selected image layer.

The files are further shown in 3 colors.

green - New files

yellow - Edited files

red - Deleted files

****

Finally, the bottom left pane is showing the Image details like image size, potential wasted space and image efficiency.

Key shortcuts- <https://github.com/wagoodman/dive#keybindings>

<https://gochronicles.com/dive/>

<https://medium.com/nexton/how-to-optimize-docker-images-using-dive-dc590f45dbf5>

**Optimizing the docker image**

FROM node:16.13.0

WORKDIR /usr/src/app

COPY . /usr/src/app/

RUN apt get update -y

RUN npm install

RUN npm run build

CMD [ "npm",”run”,"start" ]

**Change the Base Image**

Do you really need an image with a full-fledged OS to run your application?

E.g. Alpine image is only 5 MB in size and has access to a package repository that is much more complete than other BusyBox based images.

FROM node:16.13.0-slim

**Collapse Commands**

Another handy trick was to collate multiple commands into one. For example, if there are two commands that look like this:

RUN apt-get update -y

RUN npm install

Replace them with a single like this.

RUN apt-get update -y && npm install

*Shipping lightweight Docker images to production environments is the standard practice in the industry.*

**Multi-Stage Builds**

With multi-stage builds, you have just one Docker, but it's partitioned into "stages" or sections by using the FROM directive.

Every new FROM statement kicks off a new build stage with the base image mentioned in it. And if you ever need to use an artifact from another stage using the COPY --from=<stage-name> command.

FROM node:16.13.0 AS build

WORKDIR /app

COPY package\*.json /app/

RUN npm install

COPY . /app/

RUN apt-get update -y && npm run build

FROM node:14.15.5-alpine3.13 AS server

WORKDIR /usr/src/app

COPY --from=build /app/dist /usr/src/app/

COPY --from=build /app/node\_modules /usr/src/app/node\_modules/

CMD [ "npm",”run”,"start" ]

**Use ".dockerignore"**

Just the way you have .gitignore, Docker has a .dockerignore file. This excludes files that are not necessary for your image thus reducing the size of the image.

**Caching**

It doesn’t affect the size of the image but definitely helps faster builds. Place instructions that are likely to change as low in the Dockerfile as possible.

**Dive CI integration**

We can add Dive to our CI workflow to verify that the image being pushed at the end of it is of good quality and size.

When we add the ci flag into the dive command, the tool avoids the results in its interactive UI. This allows the workflow to continue with its execution.

There are two ways for implementation-

1. Integrate dive using docker container

docker run --rm -it \

-v /var/run/docker.sock:/var/run/docker.sock \

wagoodman/dive:latest --ci node-multistage:dive --lowestEfficiency=0.8 --highestUserWastedPercent=0.45

1. Install dive on CI runner, create .dive-ci file that you can put at the root of your repo. You can override the CI config file path with the --ci-config option.

```rules:

# If the efficiency is measured below X%, mark as failed.

# Expressed as a ratio between 0-1.

lowestEfficiency: 0.8

# If the amount of wasted space is at least X or larger than X, mark as failed.

# Expressed in B, KB, MB, and GB.

highestWastedBytes: 20MB

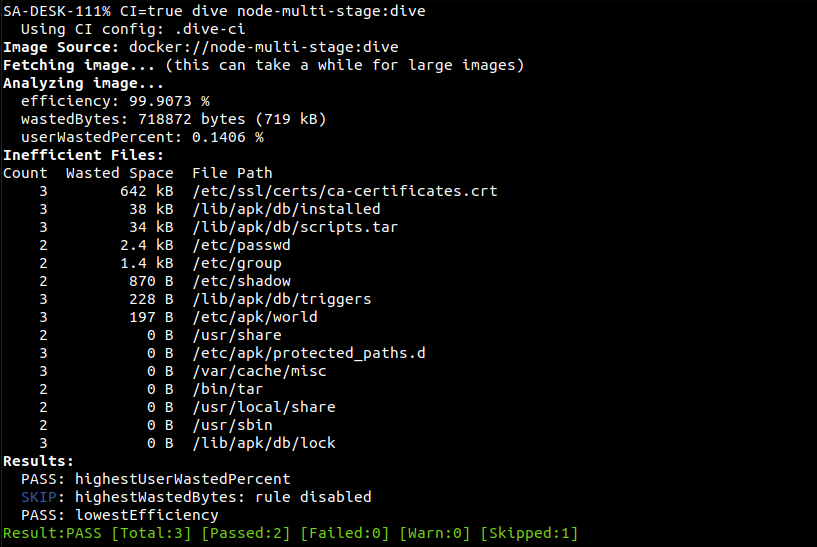
# If the amount of wasted space makes up for X% or more of the image, mark as failed.

# Note: the base image layer is NOT included in the total image size.

# Expressed as a ratio between 0-1; fails if the threshold is met or crossed.

highestUserWastedPercent: 0.45```

Add “CI=true” flag for skipping UI.

****

This will make the CI workflow to fail if the image efficiency goes down of the 80%, and the wasted space is more than 45%.

Using dive in out CI workflow can help us in creating optimized docker images. We can specify specific requirements for the image size efficiency, wasted space management which will fail or pass out CI pipeline.