cognizant®

Optimization techniques for Docker containers

16 Jun 2022

Agenda

Docker container vs VM

Entrypoint vs CMD

Optimization techniques

- **Using proper base images**
- **Optimizing dependencies**
- **Cleaning cache**
- **.docker ignore**
- **Logs**
- **Entrypoint optimization**
- **MultiStage Images**

Tools and Commands

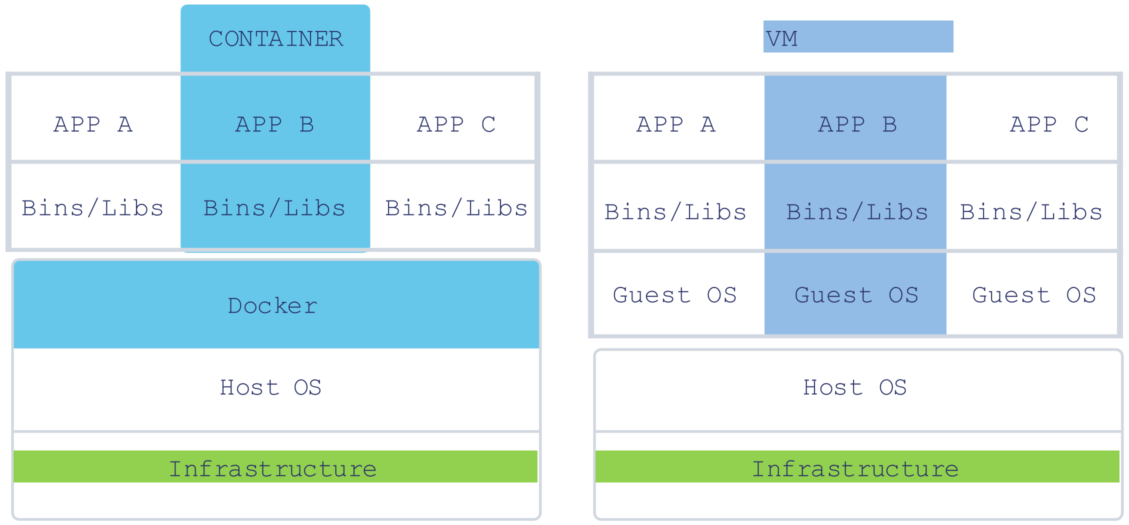
- **Dive**
- **Hadolint**
- **Pumba**

Docker container vs VM

Docker container vs VM

Docker containers differ from Virtual Machines

It is better to understand the difference between VM and Container



As you can see at the bottom - infrastructure is the same. Both Docker and VM run on physical machines, and that's the only thing they share.

VMs run on Hypervisor, and Docker runs on the Host OS. This means that a virtual machine has direct access to the CPU (in most cases). In the case of modern VMs, isolation is accomplished at the hardware level. In the case of Docker, all processes run on the Host OS and are isolated on the Host OS. Sharing the Host OS allows you to skip the Guest OS in each container, and that makes containers significantly smaller and less resource consuming.

There is a thin layer between the container and the Host OS. Docker is actually mostly a facade that combines some Host OS capabilities. When you run a container, it configures the Host OS in a way where the container is isolated, but runs inside the same system.

A container does not need to have a Guest OS. All processes are run directly in the Host OS.

Advantage of Docker

Startup time

containers do not boot a system, so they are significantly faster than VMs

Size

instead of GB, in the case of VMs, it's just a couple of MB with the required libraries, application files, etc.

The availability of ready to use images

it's easy to build, update and distribute Docker images. You can find a ready to use image for almost every software you can imagine. VM images are heavy and harder to update so they are not that common. In most cases, you will mainly find images for operating systems and you will need to install the apps inside on your own.

cognizant

Ephemeral

Docker containers are rather ephemeral. If you have a Redis container, the way to update the Redis version is to kill the existing container and run a new one. You don't backup containers [you backup volumes], and in general, you treat them as something rather short-lived. This is different to VMs as VMs are treated in most cases as normal machines. You ssh into, update the components etc. You need to keep this difference in mind, as it has a huge influence on the way we use containers.

Resource usage

Docker does not need to run a Guest OS, so the resource usage will be lower. This is not true for macOS and Windows, but it's definitely true for the core idea of running Linux Containers.

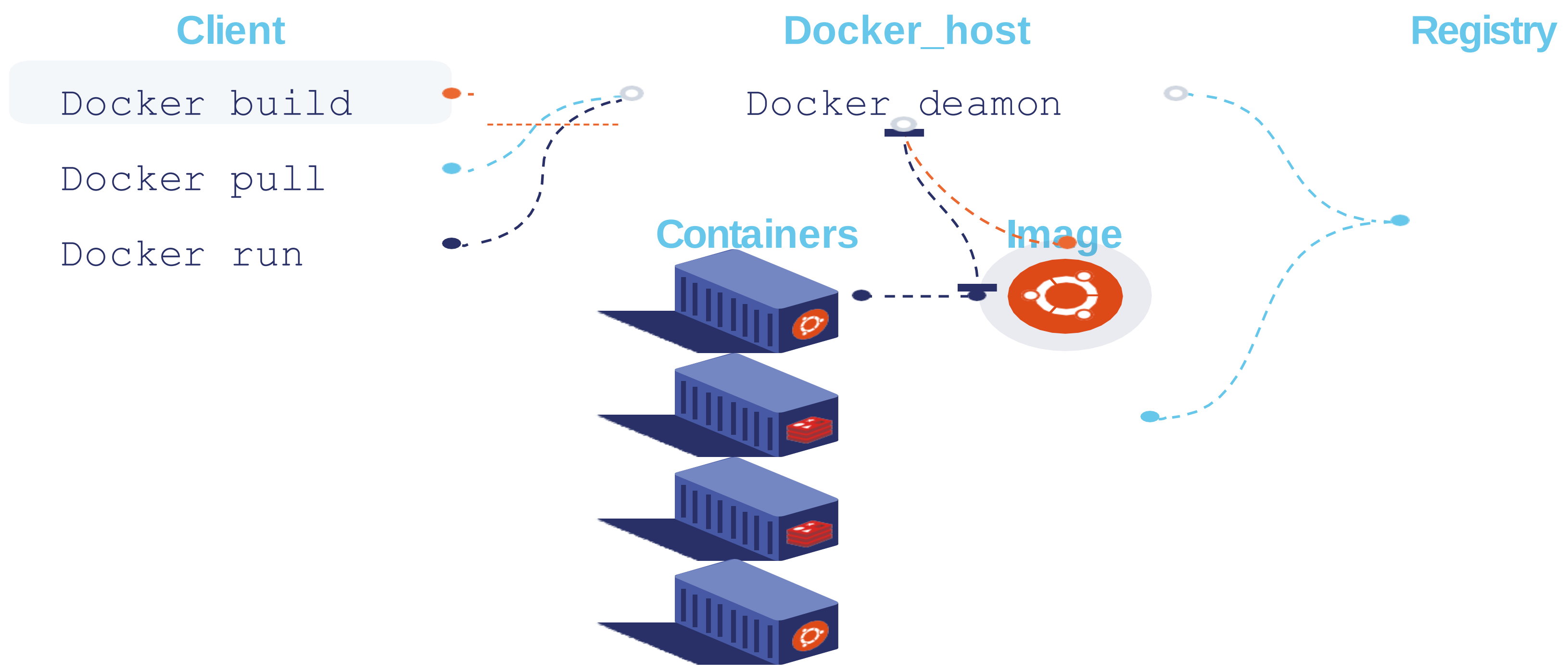
Docker images

Docker images are in most cases, built from Dockerfiles and Dockerfiles are easy to version using f.e. git. That brings all the benefits of code, ex. code review, merge requests etc.

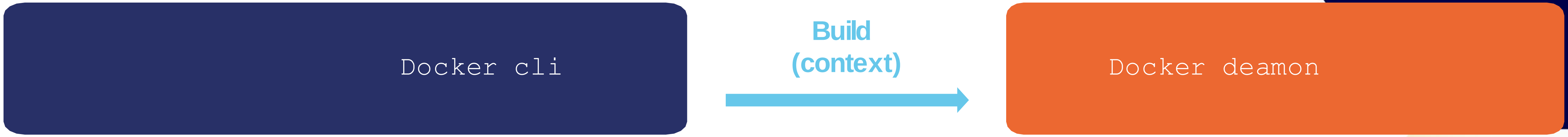
Docker Architecture

Docker architecture

when you run Docker CLI, it actually does not perform the actions. Each time you run commands like docker pull, docker run, docker build - they are passed by the CLI tool to the docker daemon.



Docker build context



When you run docker build, the CLI tool will take the Dockerfile you would like to build, compress all the files in the build directory, and send this information to the daemon. The bigger the build directory is, the more time, network, CPU etc it takes.

So, follow optimization tricks to reduce the size of the context that needs to be sent.

Docker image layers

You have likely already heard that Docker images are built using layers, and each instruction in a Dockerfile is a new layer right? But what does this mean? Let's take a look at a simple Dockerfile:

```
FROM ubuntu:latest

LABEL Owner="Viswanathan Mani"

RUN apt-get update && apt-get upgrade -y

RUN apt-get install nginx -y

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

When a step is finished, it is saved as a layer. A layer is a set of changes - files written, deleted etc. Docker gets a hash of this layer and stores it together with the layer.

The hash is later used to check if the cached layers are still up to date.

Step1 - FROM is always using cache unless you tell Docker to not do this. By default, the image is not pulled if it is present, unless you force it. So in many cases you might think you have the newest Ubuntu, but in fact it might be X months old. This is one of the reasons why you should not use the latest tag.

In our case, the image exists already, the checksum is the same, so Docker does not invalidate the next steps.

Step2 - As long as the label is the same (name and value), the checksum will remain the same, so all the next layers are not invalidated.

Step3- This might be a surprise, but the commands won't be run. The checksums on Step 2 did not change, and I did not alter the commands in this step, so Docker won't even run them. The reasoning behind this is that if the files we work on are the same, and the command is the same - then Docker assumes the result will also be the same. Quite often, if we are forced to run `apt-get install` the resulting layer would change because some libraries might be updated. In such a case, the checksum would be different and the next steps would have to be run. But this is not the case now, so the current checksum is the same as the one in the cache.

Step4- Is a similar situation as above, no changes, checksum remains the same.

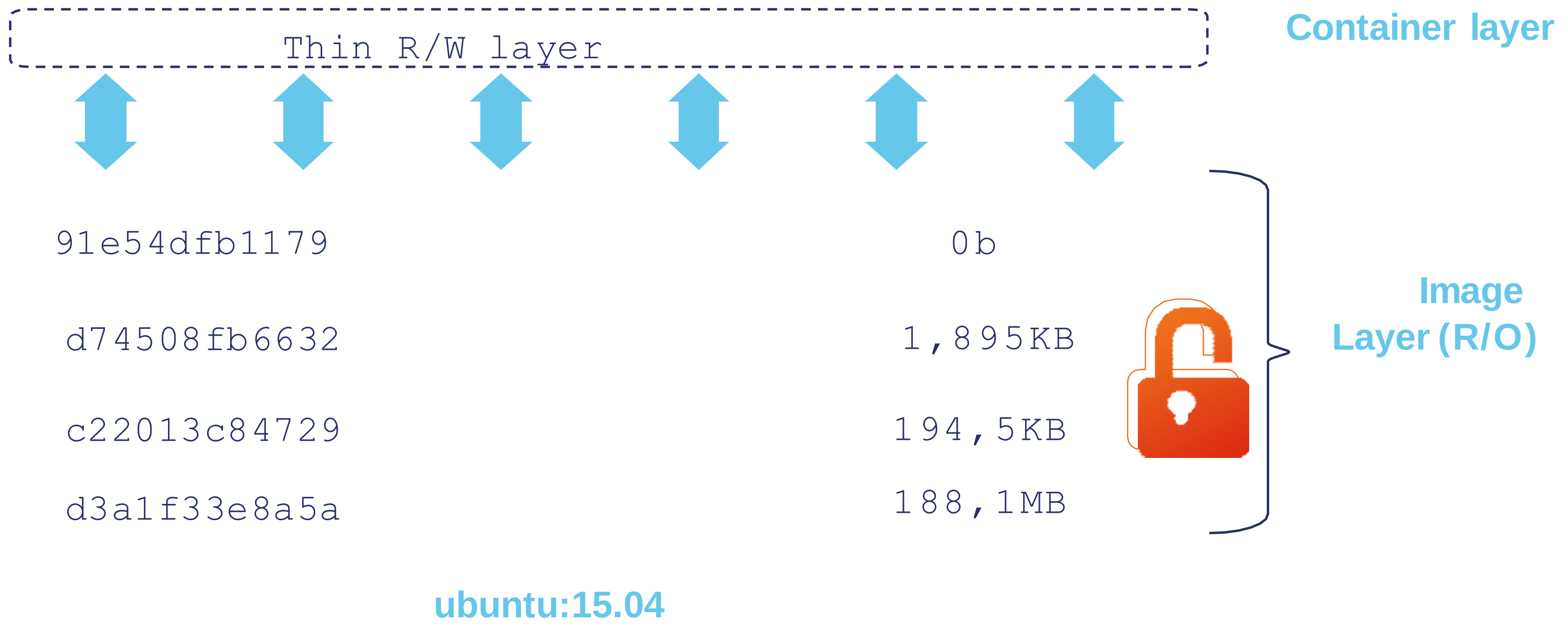
If we had a COPY/ADD step in between, the cache for the layer and all successors would be invalidated only when the copied files change.

Two layers are the same when their checksum is the same. So if there is a case where you have two Dockerfiles, but some layers will have the same checksum (because the Dockerfiles are very similar, and the first steps are identical) - the layers will be shared between the two images, and stored only once.

Layers also influence how containers are being launched.

Containers vs layers

We already know that images are built of layers, and each layer is cached. Now let's take a look at what happens when you launch a container. Obviously, a container needs to have all the files that an image provides. Docker implements quite a simple, yet powerful and performant approach that makes use of the stored image layers, and only adds a thin read/write layer on top of the image:



- If our container reads a file, the read request goes through the RW layer and then down the image layer stack as long as it finds the file. The first layer that has the file, is the last layer that was modifying this file during the build phase. So when a layer in the build phase has changed a file, it has a copy of the final content of the file in itself.
- When your container writes a file, it's written in the RW layer so next time your application tries to read the file,
- it will be returned from the R/W layer, as it is the first layer on the stack that has that file.
- If you launch the same image multiple times - nothing is copied - you have one instance of the image layers, and each new container gets a private R/W layer.
- This approach is both efficiently handling disk space and also IO performance.

Entrypoint vs CMD

Entrypoint vs CMD

- **RUN is done during the build phase, while both CMD and ENTRYPOINT are done inside a starting container. So, if you specify a RUN command, then build the image and launch 10 containers using it, the instruction will be run only once, and the resulting**
- **file changes are saved as a layer in the image. If you use CMD/ENTRYPOINT instead, then the command would be run 10 times.**
- **Both CMD and ENTRYPOINT allow you to specify the actual process that is running in the container, but they work in different ways.**

```
ENTRYPOINT [“npm”]
CMD [“run”, “serve”]
```

And we have built it as the my_app image.

Now let's take a look at the docker run command. It has the following format:

```
docker run IMAGE [CMD]
```

So we can make use of our my_app image and run:

```
docker run my_app
```

- This will run npm run serve - because npm is the entrypoint, and the default CMD is run serve.

```
docker run my_app help
```

This will run npm help - because passing a command to docker run overrides the default CMD, but does not override the ENTRYPOINT.

`ENTRYPOINT`

is always launched, no matter the option you pass to docker to run

`CMD`

holds some additional arguments that are passed to the entrypoint and that you can easily override

And that's true quite often, but there are other ways you can combine both. i.e. quite often entrypoint has only some initialization scripts and cmd holds the whole default command. In such cases the entrypoint script needs to handle that and run the CMD when it finishes.

```
ENTRYPOINT /path/to/my/entrypoint.sh
```

and the same applies to CMD. So instead of the [“command”, “argument1”] format, you can just pass a string. But be careful with that, as in such cases Docker behaves differently!

	No ENTRYPOINT	**ENTRYPOINT exec_entry p1_entry**	**ENTRYPOINT [“exec_entry”, “p1_entry”]**
No CMD	Error	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD [“exec_cmd”, “p1_cmd”]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_ cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

if you use the string format for entrypoint, the main side effect will be that the cmd is skipped!

If you stick with the array format for entrypoint, but pass cmd as a string the result will be rather strange:

```
exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd
```

Process handling

- **Docker suggests separating services between containers,**
- **it makes updating and debugging things a lot easier.**
- **Docker assumes that it manages only the root process of your container.**
- **If your entrypoint or cmd spawn a new process, Docker will assume that when it stops the main process, all the child processes will be stopped too. This is true for most applications (Apache, Nginx, etc.), but may not be true for some poorly written applications, or when you launch multiple processes in your entrypoint and do it the wrong way.**

A good process tree inside a container looks like this: ✓

- parent process
  - child process 1
    - child process 1a
    - child process 1b
  - child process 2
    - child process 2a
    - child process 2b

And a wrong process tree would look like this: ✗

- parent process 1
  - child process 1a
  - child process 1b
- parent process 2
  - child process 2a
  - child process 2b

Optimization Techniques

Using proper base images

- This is one of the simplest, widely used optimizations.
- Eeach Docker image is built on top of a different one, called a based image.
- Docker comes with multiple base images for different operating systems like Ubuntu. Such images are extended by many vendors to provide additional tools like programming languages (NodeJS, PHP, Java, .NET), databases (MySQL, PostgreSQL, Elasticsearch, Redis), tools (Busybox) and more. Most often, your project image will be based on one of these images.
- If your image is based on a PHP image, and the PHP image is based on Debian,
- it may contain a bit more stuff than you actually need to run your project e.g, tools, libraries etc. And although the images are not that big at all - around 140MB for PHP, you may want to save some space and use the alpine version of them. For a PHP image, it goes from 140MB to roughly 30MB, so a saving of 110MB ~78%.

Using proper base images

The base "distro" image for Alpine is about 5MB, while Debian image takes ~125MB. The base system is made as small as possible by deleting everything that is not required - libraries, tools etc. Also, what is needed is optimized, instead of bash there is ash etc.

Let's compare two nginx images to see how much you could potentially save by switching to an Alpine based image:

Repository	TAG	Image ID	Created	Size
Nginx	alpine	eb9291454164	2 weeks ago	22,6MB
Nginx	latest	35c43ace9216	2 weeks ago	133MB

Point to consider

- you need to create a simple image that serves a small static webpage, alpine looks like the right choice. Using the alpine version will save you 83% of the total image size.
- Alpine images might be missing some packages that your app needs.
- Alpine package repository is also less stable compared to Debian/Ubuntu, so you might have trouble installing the right NodeJS version from the package etc.
- Alpine is not using glibc - a library used to compile code. Alpine has its own tool called musl. There are some inconsistencies and in rare cases, code compiled on your machine (using glibc) won't work on Alpine - you will need to use musl to compile the codebase.

Distroless

- Because of the issues with the size of normal containers and the downsides of Alpine Linux, Google came up with the idea of Distroless images.
- "Distroless" images contain only your application and its runtime dependencies. They do not contain package managers, shells or any other programs you would expect to find in a standard Linux distribution.
- If you pull the base image, it will be only 16MB. Because it does not have anything besides the bare minimum,
- you won't be able to run docker exec -it <container> sh - as no shell is present in the container. Working with distroless images is a bit tricky because:
  - If you need to debug such a container, you need to change it to a :debug tag, as it adds shell and makes debugging possible.
  - Building distroless images is more tricky, and requires multi-stage builds

Optimizing dependencies

Optimizing dependencies is another popular quick-win. Instead of switching to an Alpine version, you can keep an eye on what you actually add to the base image. Quite often, we quickly introduce some changes and do not pay attention to what is really added to the image. This may quickly become the elephant in the corner. I've once optimized a project where the initial image had over 3GB!

One of the things that are often overlooked are dependencies.

Let's compare the two following Dockerfiles - each do the same job, create a nginx image that serves a static HTML file.

```
– FROM ubuntu:latest
– RUN     apt-get update && \
     – apt-get install nginx -y
– EXPOSE 80
– WORKDIR /var/www/html
– COPY index.html /var/www/html/
– CMD [“nginx”, “-g”, “daemon off;”]
```

Looking good right?

Optimizing dependencies

```
FROM ubuntu:latest
RUN  apt-get update && \

     apt-get install nginx -y  -no-install-recommends ;\

      rm -rf /var/lib/apt/lists/*
EXPOSE 80
WORKDIR /var/www/html
COPY index.html /var/www/html/
CMD ["nginx", "-g", "daemon off;"]
```

It's a very simple change, but it can save a fair bit of space. We just skip recommended packages during the installation phase, and after it's done, we do a very simple cleanup of some cache files.

```
Repository           TAG          Image ID      Created        Size
image_size           pre          3ecafd7a41ac  2 months ago   158MB
image_size           post         50dac569b8c   3 months ago   132MB
```

Cleaning cache

In the previous tip, WE cleared the cache for apt-get, but I would like to emphasize it a bit more. In case of a modern application, you are probably using a package manager

- like npm, Yarn, Composer, Maven etc. Most of such tools have their own cache.

dockerignore

- .dockerignore is similar to .gitignore. In fact, the format is exactly the same. .gitignore prevents files from being committed and tracked by GIT, so what does the docker equivalent do? It prevents the files from being sent in the
- Local vendors, logs, cache should not be involved in the build process. By adding them to .dockerignore, you save both time and space as those files might not be needed at all in the final image.
- .dockerignore - skipping files that should not be shared for security reasons. As mentioned in the previous slides when running a Docker build, all local files are compressed and sent to the Docker daemon.
- In your local copy of the project, you might have some files that should not be shared in a container image e.g., Private keys, credentials etc, or even a Readme that holds some data that would be better not to expose. Just add those files and/or directories to .dockerignore. They won't be sent to the daemon, so there is no way to add them to the final image.

Order of Instructions

Each layer has a checksum that depends on the files in that layer, ordering the instructions properly will introduce huge benefits:

```
FROM node:15
RUN   npm install -g http-server
WORKDIR /app
COPY . /app/
RUN   npm install  RUN    npm run build  EXPOSE 8080
CMD [ “http-server”, “dist” ]
```

```
docker build -t instruction_order:pre -f Dockerfile.pre
```

npm install takes a while.

- **The result of npm install depends on two files: package. json and package-lock.json. Only changes in these two files will require running a npm install.**
- **If you fix a typo in an html file, why would you run the npm install again?**
- **You wouldn't, but in the case of the above Dockerfile, you do. That's because the small html file change also changes the layer checksum at** `line 6` **, thus making the build process longer than it should be.**

Order of Instructions

```
FROM node:15
RUN   npm install -g http-server
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
EXPOSE 8080
CMD [ “http-server”, “dist” ]
```

Logs

Dockerized applications that still follow the “old” ways of managing logs - writing logs to files on the disk. This can have several drawbacks in containerized applications. You will probably scale your application, and if you do, then in order to check such logs, you will need to start a session in each of the running containers and search for the related logs. This is time-consuming and frustrating! The second issue is that containers are ephemeral meaning they have a short lifespan. In some situations, you may need logs from a container that does not exist anymore!

When running in production, you should have all the best possible ways to debug at hand. Or at least you should be able to understand why it failed for user Viswa on Monday at 10.30 am. Without logs, you are doomed.

Using stdout, stderr and the logging driver

start sending the logs to stdout and stderr instead of files. Docker will gather such logs, and you will be able to browse them in a slightly easier manner than with files.

Using `docker logs`, you will be able to quickly browse the logs, but as you might have noticed - this is not the best UI to search, combine logs from different sources etc.

Docker allows you to configure a so-called logging driver. By default, it's writing all the logs from stdout and stderr to files on the disk, but you can easily forward them to Graylog or Logstash, CloudWatch etc.

Logs

data volumes

I nstead of redirecting logs to stdout and stderr, you can keep writing them to the disk, but must create a data volume for them. Using this approach, you can combine logs from different instances and sends them to a selected log aggregation tool. It may be Logstash, Datadog, whatever you like.
This approach takes a bit more to configure, but it has some advantages - you can easily filter the logs by source file, you can configure rules, and you have all the logs in one place!

Send logs directly to a log aggregation server

You can also alter your application and make it send the logs to a log aggregation server directly, skipping the intermediate steps.

Entrypoint optimizations

Huge readme instructions of what you need to run to set-up the project

means there is a lack of automation, scripts, makefiles etc. In a perfect world, this would be only one command `docker-compose up`

Slow entrypoints that take minutes!!! to start

Developers tend to add dozens of scripts to the entrypoint in order to automate the set-up. Things like building front-end applications, installing vendors etc. This is a bad idea, for several reasons. It does not really make sense to run all the compilations, migrations, vendors etc., each time you start the containers. It is slow for development and even worse for production.

For development purposes, Suggestion is to keep the entrypoint thin, running only critical tasks. You can make it even more sophisticated and check if the vendor directory exists - if not, run installation, if yes, skip this step.

init.d

If you ever used an image like MySQL or PostgreSQL (or similar) you might have noticed that they offer an initialization mechanism.

You simply copy/link files into the containers init folder and they will be run/imported during the first start. This is very useful for local MySQL instances that you would like to propagate with initial data, but it also can be useful while creating your own containers

1. Check if file /docker-entrypoint-initialized exists
2. If yes, go to step 6
3. List files in /docker-entrypoint-init.d/
4. Execute each of the listed files
5. Create a file /docker-entrypoint-initialized
6. Run CMD

As a result, you can mount initialization scripts in your docker-compose.yaml file, and they will be run only when the container is fully recreated. Instead of using a file like /docker-entrypoint-initialized, you can simply check for the existence of some files like vendors or other files that are linked to the local disk or mounted to a volume - by doing so, the init scripts won't be rerun after the image is rebuilt.

Multi-stage images

Multistage helps to organize dockerfiles for development, test and production

Multistage builds allow for better organization ((you don’t need a debugger in production, but need one in development) of the images, Also it allows for the building of smaller images in an easy

clean up the artifacts that are not needed any more.

Multi-stage basics

- ✓ **you can add many FROM statements in your Dockerfile.**
- ✓ **Each FROM statement will start a new build. In other words, you can specify multiple images inside one Dockerfile.**
- ✓ **you can copy files between builds.**

Multi-stage images

```
FROM node:lts-alpine AS build
ENV VUE_APP_ENV=prod
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install

COPY . .

RUN npm run build
FROM nginx:alpine AS production
COPY --from=build /usr/src/app/dist /usr/share/nginx/html
```

In line 1,we start to build a container based on the Node image. We run npm install, npm run build etc. As a result, such an image contains: dist files for the vue app, but also some files we don't need to serve the app: npm cache, but also source files, node itself etc.

we can start building another image on line 9 - based on a thin Nginx image. We copy only the dist files (using the --from argument that references the name set on line 1),and we can be 100% sure the final image holds only what is really needed to run the app.

In this case, we could also add a third build stage to our Dockerfile, describing our development image:

```
FROM node:lts-slim AS development

WORKDIR /usr/src/app

CMD ["sh", "-c", "npm install && npm run serve"]

docker build -t multistage:development --target development
```

Labels

❖ Container labels won't make your container run any faster, or use less disk space.

❖ When you work in larger organizations or teams, this might save debugging time. if someone has an issue with your image, but does not know where to start, they might run docker image inspect, and if you added labels linked to docs, or some contact details will make their life easier.

❖ Labels can also be attached to containers, volumes, networks, etc.

```
PROJECT: WEBAPP

ENV: dev
```

Question -1

1. **Given the Dockerfile below, what is the best way to reduce the layers in the output Docker image?**

```
FROM ubuntu:18.04
RUN apt install mysql-server apache2
RUN mkdir -p /opt/app
COPY /app/config.xml /opt/app/conf
COPY /app/script.sh /opt/app/bin
```

- ❑ Combine the RUN commands separated by &&
- ❑ Combine the COPY commands spearated by &&
- ❑ Run apt-get clean --dry-run after all commands

Ans: 2

Question -2

What happens if you push two images built with the same Dockerfile, but they're tagged differently?

- ❑ The registry will only store the first image pushed but will create two separate tags that point to the same layer
- ❑ The registry will create two separate images and one tag each
- ❑ The second push will result in a no-op situation

Ans: A

Tools and commands

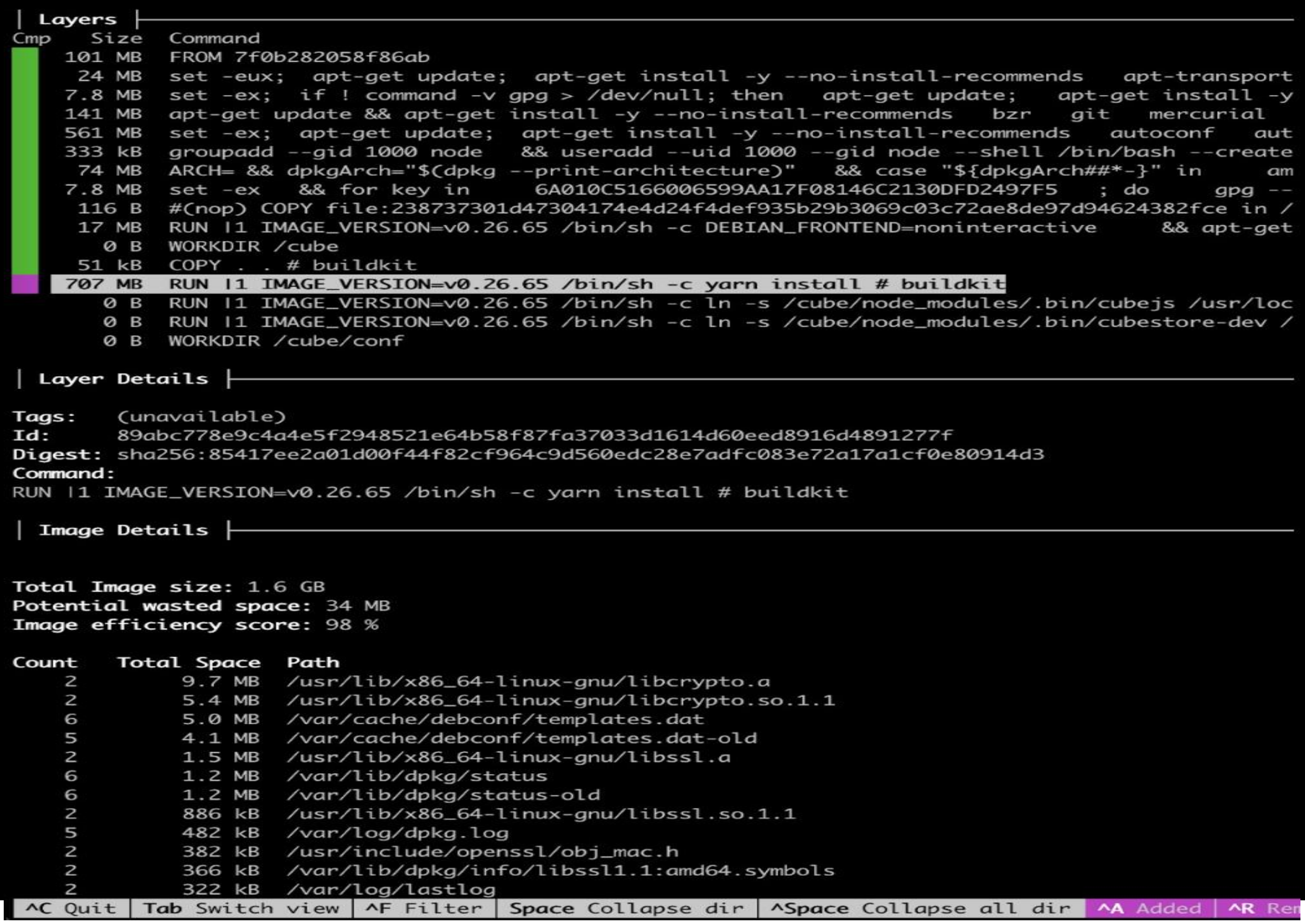
Dive

Dive is a great tool when you are working to optimize the image size and need more insight into why the image has taken some much space. It allows you to explore the docker image layers, and discover ways to shrink it.

The installation is quite straightforward.

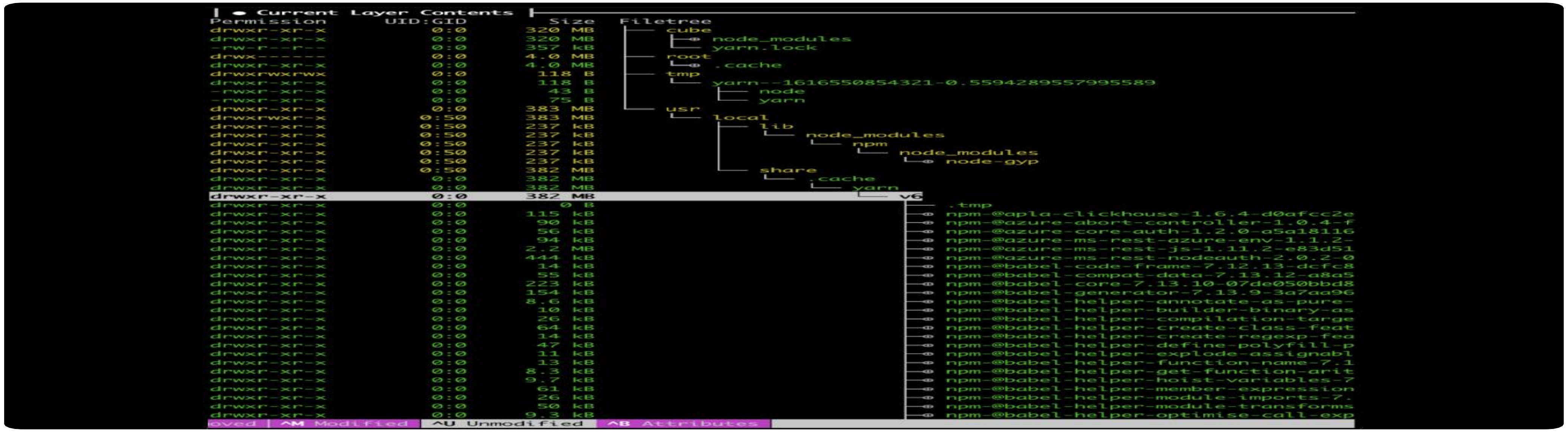
GitHub - wagoodman/dive: A tool for exploring each layer in a docker image

Dive



```
│ Layers ├──────────────────────────────────────────────────────────────────────────
Cmp   Size  Command
    101 MB  FROM 7f0b282058f86ab
     24 MB  set -eux;  apt-get update;  apt-get install -y --no-install-recommends   apt-transport
    7.8 MB  set -ex;  if ! command -v gpg > /dev/null; then   apt-get update;   apt-get install -y
    141 MB  apt-get update && apt-get install -y --no-install-recommends   bzr   git   mercurial
    561 MB  set -ex;  apt-get update;  apt-get install -y --no-install-recommends   autoconf   aut
    333 kB  groupadd --gid 1000 node   && useradd --uid 1000 --gid node --shell /bin/bash --create
     74 MB  ARCH= && dpkgArch="$(dpkg --print-architecture)"   && case "${dpkgArch##*-}" in     am
    7.8 MB  set -ex   && for key in      6A010C5166006599AA17F08146C2130DFD2497F5   ; do     gpg --
     116 B  #(nop) COPY file:238737301d47304174e4d24f4def935b29b3069c03c72ae8de97d94624382fce in /
     17 MB  RUN |1 IMAGE_VERSION=v0.26.65 /bin/sh -c DEBIAN_FRONTEND=noninteractive     && apt-get
       0 B  WORKDIR /cube
     51 kB  COPY . . # buildkit
    707 MB  RUN |1 IMAGE_VERSION=v0.26.65 /bin/sh -c yarn install # buildkit
       0 B  RUN |1 IMAGE_VERSION=v0.26.65 /bin/sh -c ln -s /cube/node_modules/.bin/cubejs /usr/loc
       0 B  RUN |1 IMAGE_VERSION=v0.26.65 /bin/sh -c ln -s /cube/node_modules/.bin/cubestore-dev /
       0 B  WORKDIR /cube/conf

│ Layer Details ├────────────────────────────────────────────────────────────────────

Tags:    (unavailable)
Id:      89abc778e9c4a4e5f2948521e64b58f87fa37033d1614d60eed8916d4891277f
Digest:  sha256:85417ee2a01d00f44f82cf964c9d560edc28e7adfc083e72a17a1cf0e80914d3
Command:
RUN |1 IMAGE_VERSION=v0.26.65 /bin/sh -c yarn install # buildkit

│ Image Details ├────────────────────────────────────────────────────────────────────


Total Image size: 1.6 GB
Potential wasted space: 34 MB
Image efficiency score: 98 %

Count   Total Space  Path
    2        9.7 MB  /usr/lib/x86_64-linux-gnu/libcrypto.a
    2        5.4 MB  /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1
    6        5.0 MB  /var/cache/debconf/templates.dat
    5        4.1 MB  /var/cache/debconf/templates.dat-old
    2        1.5 MB  /usr/lib/x86_64-linux-gnu/libssl.a
    6        1.2 MB  /var/lib/dpkg/status
    6        1.2 MB  /var/lib/dpkg/status-old
    2        886 kB  /usr/lib/x86_64-linux-gnu/libssl.so.1.1
    5        482 kB  /var/log/dpkg.log
    2        382 kB  /usr/include/openssl/obj_mac.h
    2        366 kB  /var/lib/dpkg/info/libssl1.1:amd64.symbols
    2        322 kB  /var/log/lastlog
 ^C Quit │ Tab Switch view │ ^F Filter │ Space Collapse dir │ ^Space Collapse all dir │ ^A Added │ ^R Rem
```

Dive

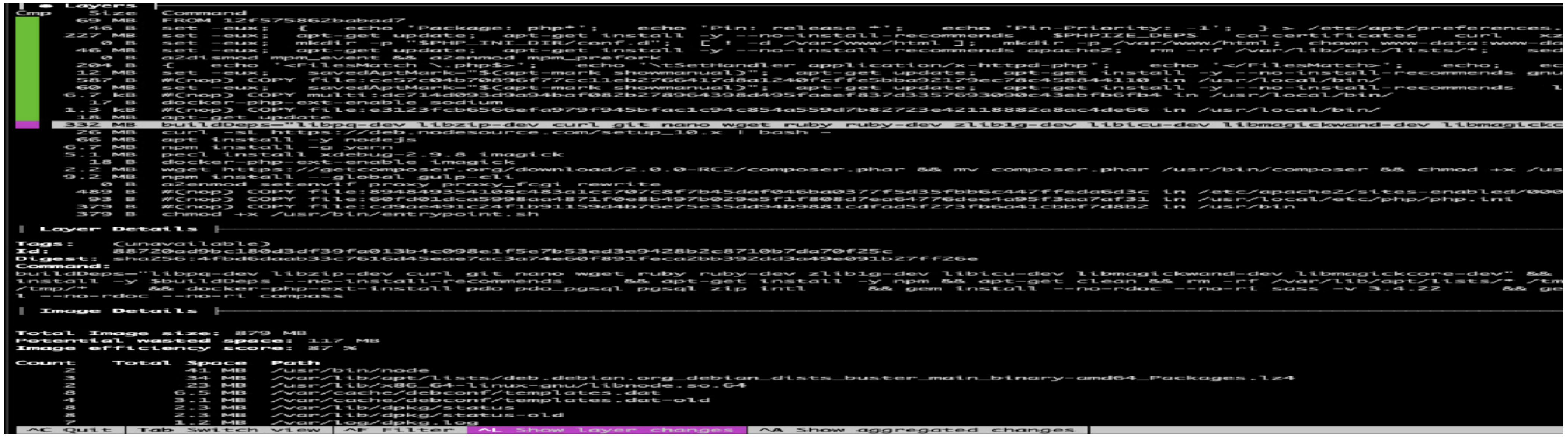


```
│ ● Current Layer Contents ├──────────────────────────────────────────────
Permission     UID:GID       Size  Filetree
drwxr-xr-x         0:0     320 MB  ├── cube
drwxr-xr-x         0:0     320 MB  │   ├─⊕ node_modules
-rw-r--r--         0:0     357 kB  │   └── yarn.lock
drwx------         0:0     4.0 MB  ├── root
drwxr-xr-x         0:0     4.0 MB  │   └─⊕ .cache
drwxrwxrwx         0:0      118 B  ├── tmp
drwxr-xr-x         0:0      118 B  │   └── yarn--1616550854321-0.5594289557995589
-rwxr-xr-x         0:0       43 B  │       ├── node
-rwxr-xr-x         0:0       75 B  │       └── yarn
drwxr-xr-x         0:0     383 MB  └── usr
drwxrwxr-x        0:50     383 MB      └── local
drwxr-xr-x        0:50     237 kB          ├── lib
drwxr-xr-x        0:50     237 kB          │   └── node_modules
drwxr-xr-x        0:50     237 kB          │       └── npm
drwxr-xr-x        0:50     237 kB          │           └── node_modules
drwxr-xr-x        0:50     237 kB          │               └─⊕ node-gyp
drwxr-xr-x        0:50     382 MB          └── share
drwxr-xr-x         0:0     382 MB              └── .cache
drwxr-xr-x         0:0     382 MB                  └── yarn
drwxr-xr-x         0:0     382 MB                      └── v6
drwxr-xr-x         0:0        0 B                          ├── .tmp
drwxr-xr-x         0:0     115 kB                          ├─⊕ npm-@apla-clickhouse-1.6.4-d0afcc2e
drwxr-xr-x         0:0      90 kB                          ├─⊕ npm-@azure-abort-controller-1.0.4-f
drwxr-xr-x         0:0      56 kB                          ├─⊕ npm-@azure-core-auth-1.2.0-a5a18116
drwxr-xr-x         0:0      94 kB                          ├─⊕ npm-@azure-ms-rest-azure-env-1.1.2-
drwxr-xr-x         0:0     2.2 MB                          ├─⊕ npm-@azure-ms-rest-js-1.11.2-e83d51
drwxr-xr-x         0:0     444 kB                          ├─⊕ npm-@azure-ms-rest-nodeauth-2.0.2-0
drwxr-xr-x         0:0      14 kB                          ├─⊕ npm-@babel-code-frame-7.12.13-dcfc8
drwxr-xr-x         0:0      55 kB                          ├─⊕ npm-@babel-compat-data-7.13.12-a8a5
drwxr-xr-x         0:0     223 kB                          ├─⊕ npm-@babel-core-7.13.10-07de050bbd8
drwxr-xr-x         0:0     154 kB                          ├─⊕ npm-@babel-generator-7.13.9-3a7aa96
drwxr-xr-x         0:0     8.6 kB                          ├─⊕ npm-@babel-helper-annotate-as-pure-
drwxr-xr-x         0:0      10 kB                          ├─⊕ npm-@babel-helper-builder-binary-as
drwxr-xr-x         0:0      26 kB                          ├─⊕ npm-@babel-helper-compilation-targe
drwxr-xr-x         0:0      64 kB                          ├─⊕ npm-@babel-helper-create-class-feat
drwxr-xr-x         0:0      14 kB                          ├─⊕ npm-@babel-helper-create-regexp-fea
drwxr-xr-x         0:0      47 kB                          ├─⊕ npm-@babel-helper-define-polyfill-p
drwxr-xr-x         0:0      11 kB                          ├─⊕ npm-@babel-helper-explode-assignabl
drwxr-xr-x         0:0      13 kB                          ├─⊕ npm-@babel-helper-function-name-7.1
drwxr-xr-x         0:0     8.3 kB                          ├─⊕ npm-@babel-helper-get-function-arit
drwxr-xr-x         0:0     9.7 kB                          ├─⊕ npm-@babel-helper-hoist-variables-7
drwxr-xr-x         0:0      61 kB                          ├─⊕ npm-@babel-helper-member-expression
drwxr-xr-x         0:0      26 kB                          ├─⊕ npm-@babel-helper-module-imports-7.
drwxr-xr-x         0:0      50 kB                          ├─⊕ npm-@babel-helper-module-transforms
drwxr-xr-x         0:0     9.3 kB                          ├─⊕ npm-@babel-helper-optimise-call-exp
oved | ^M Modified   ^U Unmodified   ^B Attributes
```

On the right side, you can browse dirs and files added to see which files actually take so much space. You can filter the list to hide unmodified changes and collapsed all dirs to then expand only the directories you are interested in. So the first thing you notice is that the cube directory takes up 320MB, and that's almost all consumed by node_modules. There's probably not much to save here, so lets go further.

Another 383MB are taken up by /usr. It looks like the yarn cache takes 382MB! That's definitely something we could remove from the image and save a lot of space. In this case, we should just modify this RUN command to delete the cache after it finishes the yarn install. The clean-up needs to be done in the same step that is adding the unwanted files because of the layers.

Dive



```
│ ● Layers ├
Cmp   Size  Command
     69 MB  FROM 12f575862babad7
     46 B   set -eux;   {     echo 'Package: php*';     echo 'Pin: release *';     echo 'Pin-Priority: -1';   } > /etc/apt/preferences.
    227 MB  set -eux;   apt-get update;   apt-get install -y --no-install-recommends   $PHPIZE_DEPS   ca-certificates   curl   xz
      0 B   set -eux;   mkdir -p "$PHP_INI_DIR/conf.d";   [ ! -d /var/www/html ];   mkdir -p /var/www/html;   chown www-data:www-da
     46 MB  set -eux;   apt-get update;   apt-get install -y --no-install-recommends apache2;   rm -rf /var/lib/apt/lists/*;   sed
      0 B   a2dismod mpm_event && a2enmod mpm_prefork
    204 B   {     echo '<FilesMatch \.php$>';     echo '\tSetHandler application/x-httpd-php';     echo '</FilesMatch>';     echo;     ec
     12 MB  set -eux;   savedAptMark="$(apt-mark showmanual)";   apt-get update;   apt-get install -y --no-install-recommends gnu
    587 B   #(nop) COPY file:ce57c04b70896f77cc11eb2766417d8a1240fcffe5bba92179ec78c4588844110 in /usr/local/bin/
     60 MB  set -eux;   savedAptMark="$(apt-mark showmanual)";   apt-get update;   apt-get install -y --no-install-recommends   l
    6.7 kB  #(nop) COPY multi:dc714d093d9a94baf082b278964398d495faeef837d3357693090c43ebfb6fb4 in /usr/local/bin/
     17 B   docker-php-ext-enable sodium
    1.3 kB  #(nop) COPY file:e3123fcb6566efa979f945bfac1c94c854a559d7b82723e42118882a8ac4de66 in /usr/local/bin/
     18 MB  apt-get update
    332 MB  buildDeps="libpq-dev libzip-dev curl git nano wget ruby ruby-dev zlib1g-dev libicu-dev libmagickwand-dev libmagickc
     26 MB  curl -sL https://deb.nodesource.com/setup_10.x | bash -
     66 MB  apt install -y nodejs
    6.7 MB  npm install -g yarn
    5.1 MB  pecl install xdebug-2.9.8 imagick
     18 B   docker-php-ext-enable imagick
    2.2 MB  wget https://getcomposer.org/download/2.0.0-RC2/composer.phar && mv composer.phar /usr/bin/composer && chmod +x /us
    9.2 MB  npm install --global gulp-cli
      0 B   a2enmod setenvif proxy proxy_fcgi rewrite
    489 B   #(nop) COPY file:894849354108c483a1cc707c8f7b45daf046ba0377f5d35fbbb6c447ffeda6d3c in /etc/apache2/sites-enabled/000
     93 B   #(nop) COPY file:60fd01dca5998aa4871f0e8b497b029e5f1f808d7ea64776dee4a95f3aa7af31 in /usr/local/etc/php/php.ini
    379 B   #(nop) COPY file:cd9ae491c24f1b91159d4b76e75e35dd94b9881cdfad5f273fb6a41cbbf7d8b2 in /usr/bin
    379 B   chmod +x /usr/bin/entrypoint.sh

│ Layer Details ├──────────────────────────────────────────────────────────────

Tags:     (unavailable)
Id:       88720ad9bc180d3df39fa013b4c098e1f5e7b53ed3e9428b2c8710b7da70f25c
Digest:   sha256:4fbd6daab33c7616d45eae7ac3a74e60f891feca2bb392dd3a49e091b27ff26e
Command:
buildDeps="libpq-dev libzip-dev curl git nano wget ruby ruby-dev zlib1g-dev libicu-dev libmagickwand-dev libmagickcore-dev" &&
install -y $buildDeps --no-install-recommends     && apt-get install -y npm && apt-get clean && rm -rf /var/lib/apt/lists/* /tm
/tmp/*     && docker-php-ext-install pdo pdo_pgsql pgsql zip intl      && gem install --no-rdoc --no-ri sass -v 3.4.22     && ge
1 --no-rdoc --no-ri compass

│ Image Details ├──────────────────────────────────────────────────────────────

Total Image size: 879 MB
Potential wasted space: 117 MB
Image efficiency score: 87 %

Count   Total Space  Path
    2        41 MB   /usr/bin/node
    3        34 MB   /var/lib/apt/lists/deb.debian.org_debian_dists_buster_main_binary-amd64_Packages.lz4
    2        23 MB   /usr/lib/x86_64-linux-gnu/libnode.so.64
    8       6.5 MB   /var/cache/debconf/templates.dat
    4       3.1 MB   /var/cache/debconf/templates.dat-old
    8       2.3 MB   /var/lib/dpkg/status
    8       2.3 MB   /var/lib/dpkg/status-old
    7       1.2 MB   /var/log/dpkg.log
```

The selected layer is adding a lot of different stuff. It runs apt-get, docker-php-ext-install and also gem install. It would be probably better to split that, as it is now impossible to detect which of these is adding the most overhead to the space used.

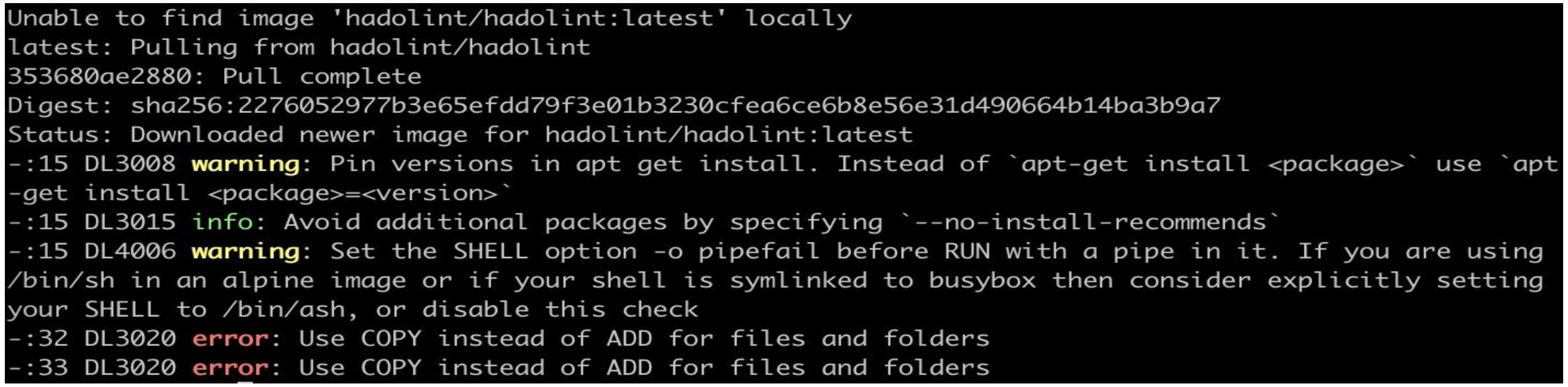
Hadolint

Hadolint is a great every-day-use tool. It is a linter, for Dockerfiles. It can be used for a one-time check, or it can be set as a step in your CI process to continuously check the quality of your Dockerfiles.

Hadolint will detect some of the problems and will give you clear information about what is wrong.

The easiest way to run Hadolint is to use a Docker image, and you do not need to have an image build.

```
docker run --rm -i hadolint/hadolint < Dockerfile
```



```
Unable to find image 'hadolint/hadolint:latest' locally
latest: Pulling from hadolint/hadolint
353680ae2880: Pull complete
Digest: sha256:2276052977b3e65efdd79f3e01b3230cfea6ce6b8e56e31d490664b14ba3b9a7
Status: Downloaded newer image for hadolint/hadolint:latest
-:15 DL3008 warning: Pin versions in apt get install. Instead of `apt-get install <package>` use `apt
-get install <package>=<version>`
-:15 DL3015 info: Avoid additional packages by specifying `--no-install-recommends`
-:15 DL4006 warning: Set the SHELL option -o pipefail before RUN with a pipe in it. If you are using
/bin/sh in an alpine image or if your shell is symlinked to busybox then consider explicitly setting
your SHELL to /bin/ash, or disable this check
-:32 DL3020 error: Use COPY instead of ADD for files and folders
-:33 DL3020 error: Use COPY instead of ADD for files and folders
```

Hadolint

Hadolint outputs all the issues it was able to spot, together with the severity, issue number and a short description. You can easily get more information about an issue by copying the ID and browsing the Hadolint wiki page on Github:

DL3008 · hadolint/hadolint Wiki · GitHub

DL3020 · hadolint/hadolint Wiki · GitHub

The detailed instructions quite often offer an example of what the corrected version would look like, so in most cases the issues will be very easy to fix. Hadolint takes about to finish and it's very easy to integrate with CI tools like GitHub. It is recommend to add it to your CI pipeline.

Pumba

it allows simulating different issues and problems like network outage, packet drops, freezing processes, etc.

it's awesome to test some common problems in distributed or microservice oriented projects. What if the network latency goes up to a couple of seconds? What if one of the services go down? Well, with Pumba you don't have to guess, you can test such a scenario within minutes!

Here is a quick example of how you can introduce a delay to one of your containers. Let's create a container that will ping google.com - the output should be easy to understand.

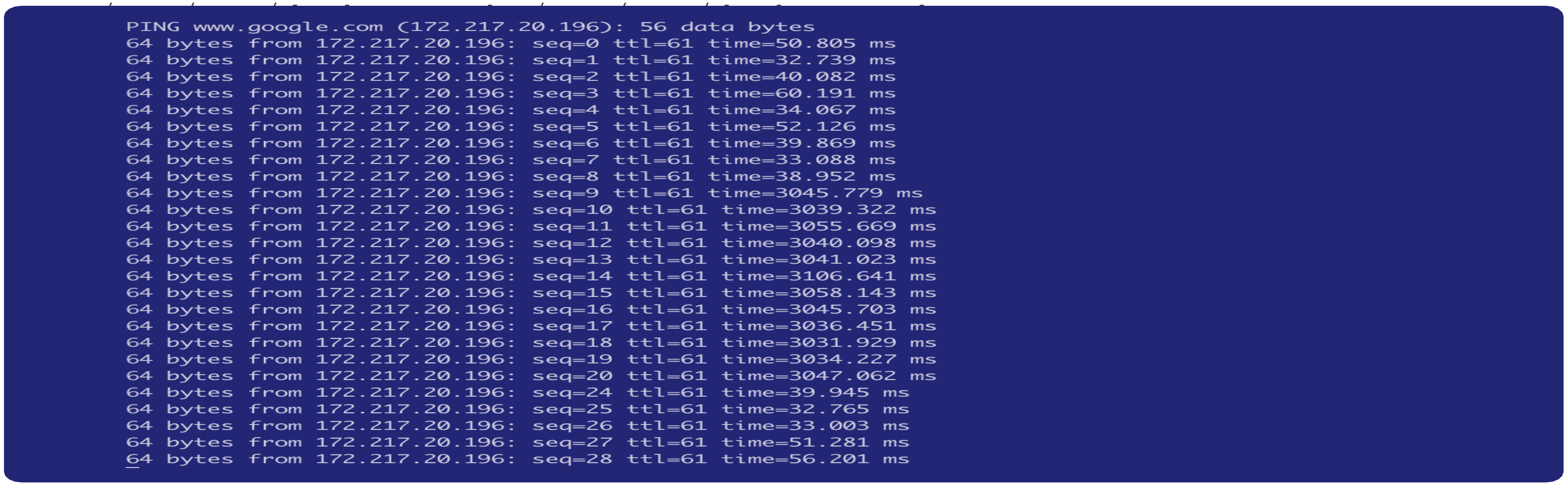
```
docker run -it --rm --name testDelay alpine sh -c "apk add
--update iproute2 && ping www.google.com"
```

Pumba

We can run Pumba in a separate container in order to introduce the delay.

```
docker run -ti --rm -v /var/run/docker.sock:/var/run/docker.sock

gaiaadm/pumba netem --duration 15s delay --time 3000  testDelay
```

If you are not familiar with the



```
PING www.google.com (172.217.20.196): 56 data bytes
64 bytes from 172.217.20.196: seq=0 ttl=61 time=50.805 ms
64 bytes from 172.217.20.196: seq=1 ttl=61 time=32.739 ms
64 bytes from 172.217.20.196: seq=2 ttl=61 time=40.082 ms
64 bytes from 172.217.20.196: seq=3 ttl=61 time=60.191 ms
64 bytes from 172.217.20.196: seq=4 ttl=61 time=34.067 ms
64 bytes from 172.217.20.196: seq=5 ttl=61 time=52.126 ms
64 bytes from 172.217.20.196: seq=6 ttl=61 time=39.869 ms
64 bytes from 172.217.20.196: seq=7 ttl=61 time=33.088 ms
64 bytes from 172.217.20.196: seq=8 ttl=61 time=38.952 ms
64 bytes from 172.217.20.196: seq=9 ttl=61 time=3045.779 ms
64 bytes from 172.217.20.196: seq=10 ttl=61 time=3039.322 ms
64 bytes from 172.217.20.196: seq=11 ttl=61 time=3055.669 ms
64 bytes from 172.217.20.196: seq=12 ttl=61 time=3040.098 ms
64 bytes from 172.217.20.196: seq=13 ttl=61 time=3041.023 ms
64 bytes from 172.217.20.196: seq=14 ttl=61 time=3106.641 ms
64 bytes from 172.217.20.196: seq=15 ttl=61 time=3058.143 ms
64 bytes from 172.217.20.196: seq=16 ttl=61 time=3045.703 ms
64 bytes from 172.217.20.196: seq=17 ttl=61 time=3036.451 ms
64 bytes from 172.217.20.196: seq=18 ttl=61 time=3031.929 ms
64 bytes from 172.217.20.196: seq=19 ttl=61 time=3034.227 ms
64 bytes from 172.217.20.196: seq=20 ttl=61 time=3047.062 ms
64 bytes from 172.217.20.196: seq=24 ttl=61 time=39.945 ms
64 bytes from 172.217.20.196: seq=25 ttl=61 time=32.765 ms
64 bytes from 172.217.20.196: seq=26 ttl=61 time=33.003 ms
64 bytes from 172.217.20.196: seq=27 ttl=61 time=51.281 ms
64 bytes from 172.217.20.196: seq=28 ttl=61 time=56.201 ms
```

Thank you

Email : Viswanathan.mani@cognizant.com