accesto

Docker Deep Dive

Optimization techniques
for better Docker containers

0

Why you should read this book?

There are many reasons why you should. Just to name a few: cutting build times by half to save lots of time every day, saving hundreds of megabytes of disk space, improving project security, mastering Docker, or just paying attention to the quality of Docker images and runtimes you create.

Still not interested? By using the described rules, I was able to optimize an image from over 3GB to 800 MB in the development version, and cut the build time by 4x.
I hope you are interested now :)

Foreword

In the last few years, I have noticed a lot of developers waste their time with Docker. They run suboptimal configurations that slow them down, cause trouble, and cause irritation. What's even more important, I noticed that in many cases there are some simple changes that when introduced, would improve the situation by a lot. I decided to write this e-book in order to share some simple tips and make people work with Docker in a more efficient way.
I will cover the theory behind Docker, some simple optimization techniques, and also some nice tools.

In this book I have decided to skip the very basics of Docker, so if you are new to it, this is not a good point to start. You should know how to start, stop and work with containers and images. My idea was to describe the in-depth details of how Docker works, but not dig too deep into the internals of a Docker implementation. I would call the level of detail in this e-book the sweet spot on the axis of Docker knowledge. This is information that every developer should have. I focused mostly on performance, as it was causing the majority of the problems in the teams I had the chance to work with.

I first cover the theory, then provide some optimization tricks and finally talk about some interesting tools. You can alter the order and adjust it to your needs, i.e. read some optimizations tips first, and then go back to the theory, so you can understand why the tips work.

This is the first version released in May 2021, and my plan is to update it every couple of months, so if you have any questions, feedback, or suggestions - contact me:

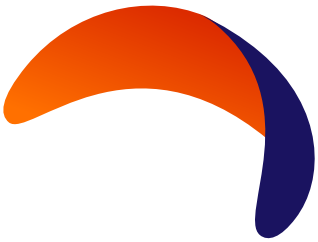
Michal Kurzeja in

CTO @ Accesto

CONTACT ME +

SCHEDULE A CALL +

Table of contents



1

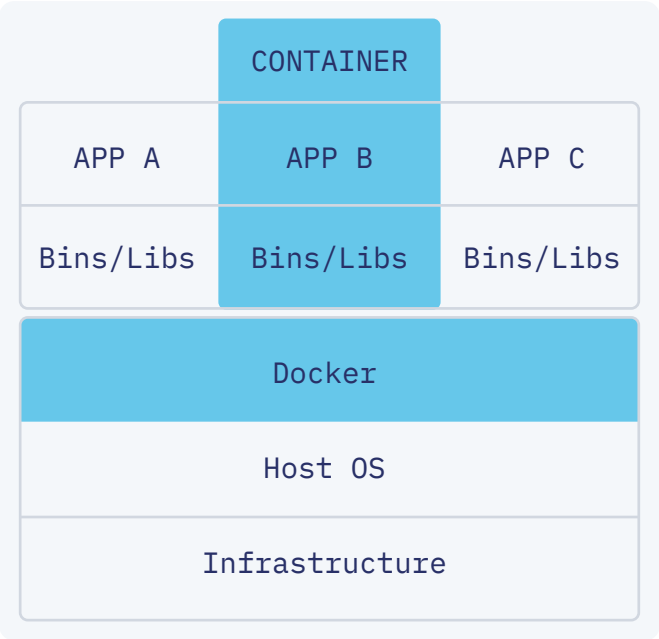
Theory

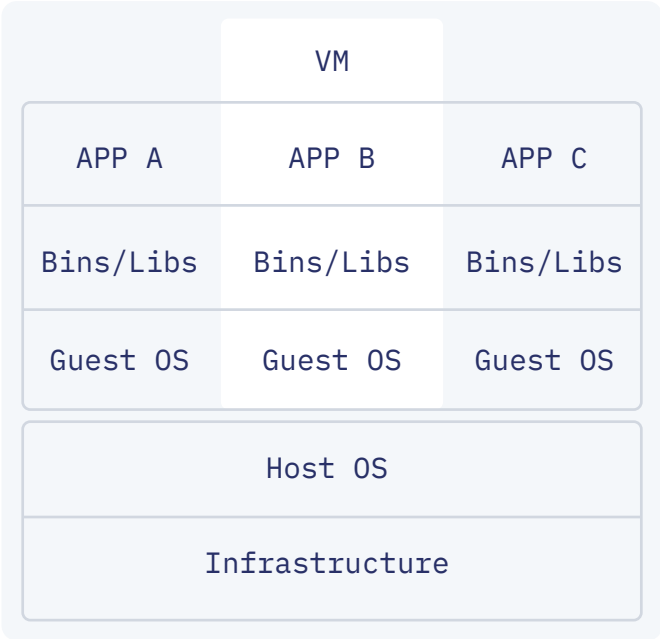
In this e-book I won't describe every detail of Docker architecture and implementation. My goal is to introduce some fundamental concepts behind Docker that help to understand the optimizations showcased. I don’t want you to blindly follow the described optimization steps, rather to understand the ins and outs of why they work.

To be honest, Docker is based on some very simple ideas that if combined, offer a robust tool to run your application. All the building blocks are rather easy to understand. I believe that focusing on this first, and later on commands, etc., will make you more proficient with Docker and containers in general.

Docker container vs VM

Docker containers differ significantly from Virtual Machines and thus the applications of Docker are totally different. It is good to understand that difference. We will dive deeper in the following sections, but let's cover the basics first.





Take a look at the above picture. 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.

One step higher and you can see 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. So, on one hand, Docker is less secure as the isolation is weaker and easier to exploit, but on the other hand, 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 in the case of Docker, marked as Docker on the picture. 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.

So, the biggest difference from our perspective, is the fact that a container does not need to have a Guest OS. All processes are run directly in the Host OS. This will cause most of the following differences:

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.

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

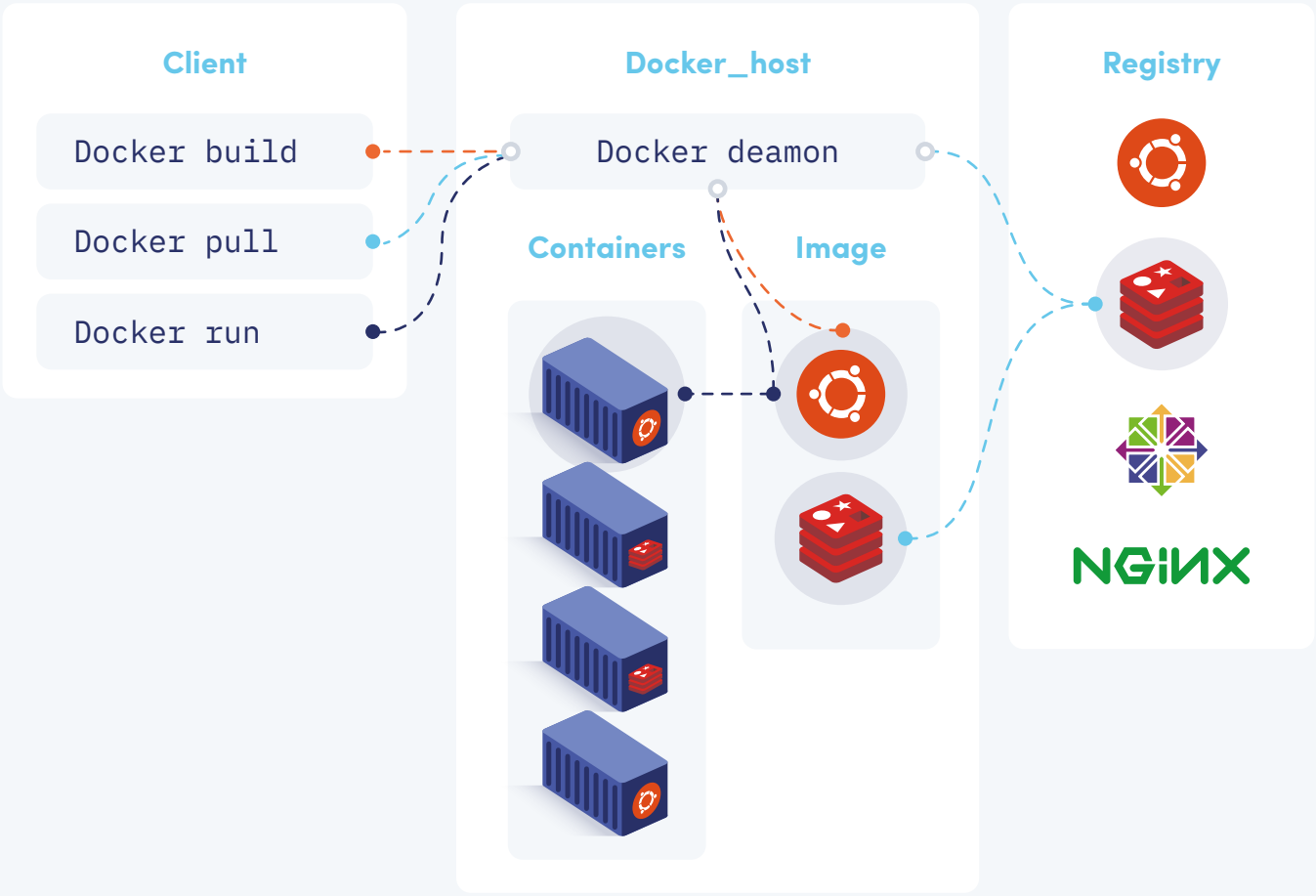
Docker does not need to run a Guest OS, so the resource usage will be lower. This is not true for macOS and Windows, but it's definitely true for the core idea of running Linux Containers.

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

One of the most important pieces of information about Docker architecture that you need to know in order to understand the optimizations described in this book is that 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.



This is not noticeable in most cases, but it plays a huge role in the case of a docker build.

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. This is why some of the described optimization tricks 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:

```
STEP 1   FROM ubuntu:latest

STEP 2   LABEL maintainer="myname@somecompany.com"

STEP 3   RUN apt-get update && apt-get upgrade -y

STEP 4   RUN apt-get install nginx -y

STEP 5   EXPOSE 80

STEP 6   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.

In the case of the above Dockerfile, running build for the second time would work as followed:

STEP 1 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.

STEP 2 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.

STEPS 3,4 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.

STEPS 5,6 Is a similar situation as above, no changes, checksum remains the same.

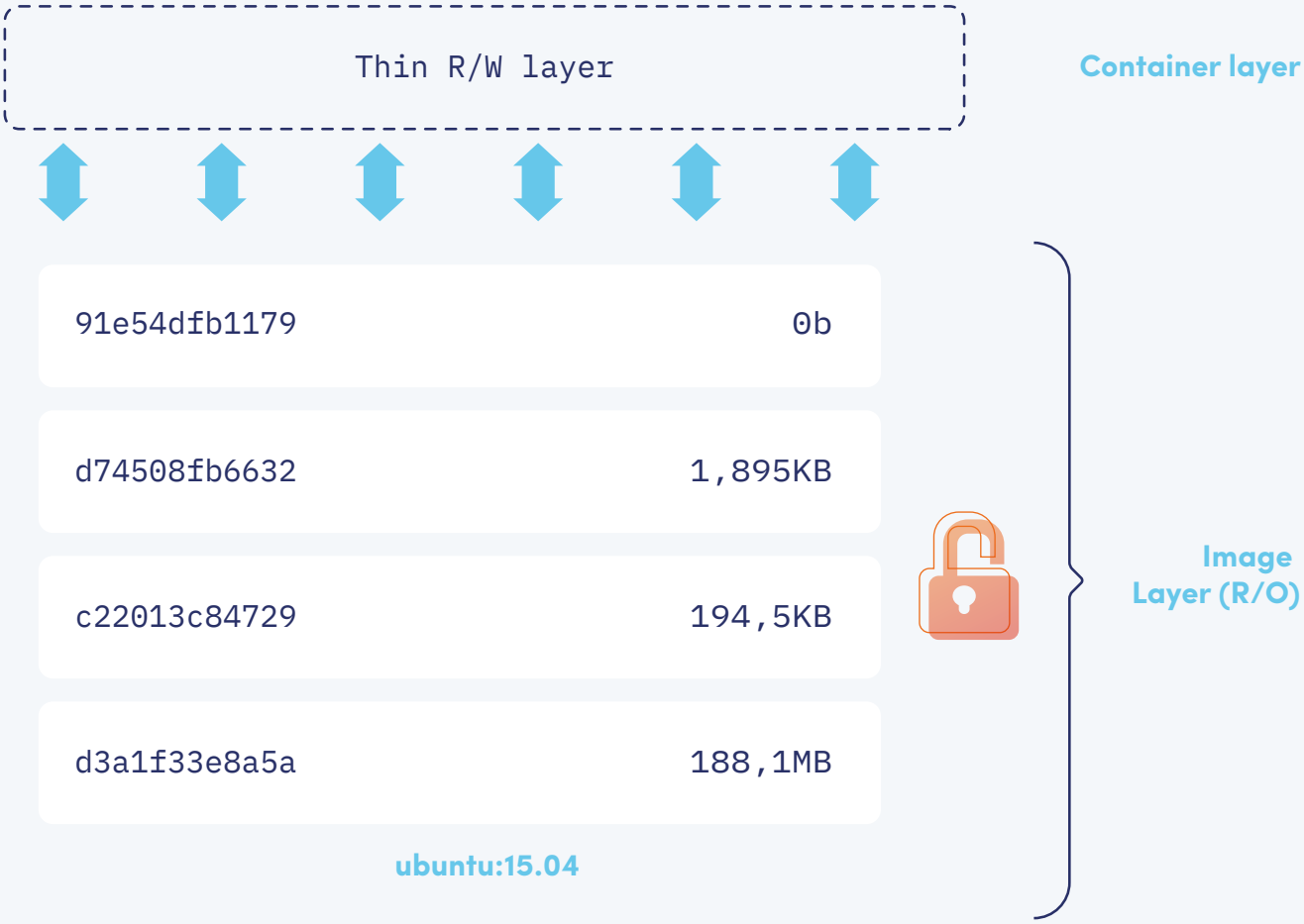
If we had a COPY/ADD step in between, the cache for the layer and all successors would be invalidated only when the copied files change.

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. Lets dive deeper into this part.

Containers vs layers

So you 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:



Now if your 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

I noticed entrypoint and CMD are confusing for a lot of people, and it's also a bit hard to explain the ins and outs of both.

Before we start, let's add one more thing to the discussion - the RUN instruction, as some people are also confused how it differs from CMD/ENTRYPOINT.

The answer is quite simple - 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.

Let's start with a simple example, assuming we have a Dockerfile with the following two lines:

```
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.

Looking at this example, you may notice that:

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.

If your familiar with Dockerfile, you probably know that you can also write

```
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

So 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

As you probably know, Docker suggests separating services between containers, so each service has its own container. And that's definitely a good practice, as it makes updating and debugging things a lot easier.

But there is more behind this rule. Docker assumes that it manages only the root process of your container. So 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

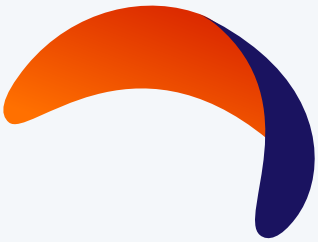
parent process 2

- child process 2a
- child process 2b

A badly written container will have issues with signal handling and zombie processes. If you would like to know more, check out this Google article.

For now, you just need to keep the above limitations in mind and consider them while writing a custom entrypoint or spawning processes in your application.

MORE POSTS ON OUR BLOG +



2

Actionable optimizations

As mentioned in the foreword, I have decided to focus mostly on optimizations because they will influence many of the daily use cases and improve many areas i.e., developer experience, observability, stability, performance. Most of the described optimizations are common good practices that I strongly advise following. These are based on my experience as the CTO at Accesto, but also from helping other teams and reviewing open source projects. Developers tend to spend hours discussing coding best practices and almost no time discussing Docker best practices - let's change that.

Using proper base images

This is one of the simplest, widely used optimizations. If you are aware of it, just skip this description and jump right to the outcome or straight to the next technique.

As you should already know, each 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.

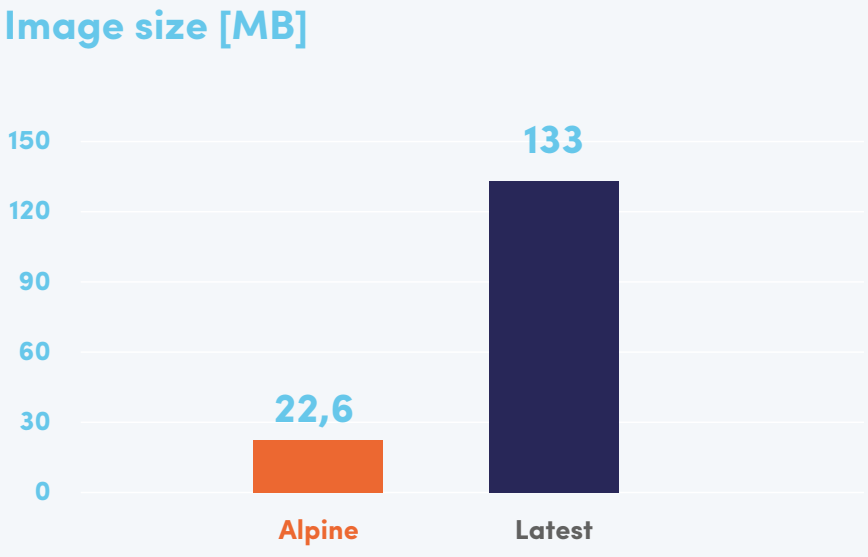
As you may have guessed, if your image is based on a PHP image, and the PHP image is based on Debian, it may contain a bit more stuff than you actually need to run your project e.g, tools, libraries etc. And although the images are not that big at all - around 140MB for PHP, you may want to save some 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%.

Why are alpine images so small?

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



So if 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.

At this point alpine images might look great, but do alpine images have a downside? Of course!

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.

In some cases, Alpine can also cause your application to run slower than on Ubuntu or Debian based images.

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. Google describes them as followed:

> "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 - we cover them later in the book.

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;"]
```

Looks normal right?

Now lets change the RUN part a bit:

```
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

In this case, we saved 26 MB, so over 16% of the initial size. Not much, but as you can imagine, such small things add up.

Cleaning cache

In the previous tip, I 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.

I've checked one of our projects and composer cache was taking 54 MB. That means that a ~300MB image reduced by 18% in less than a minute of work!

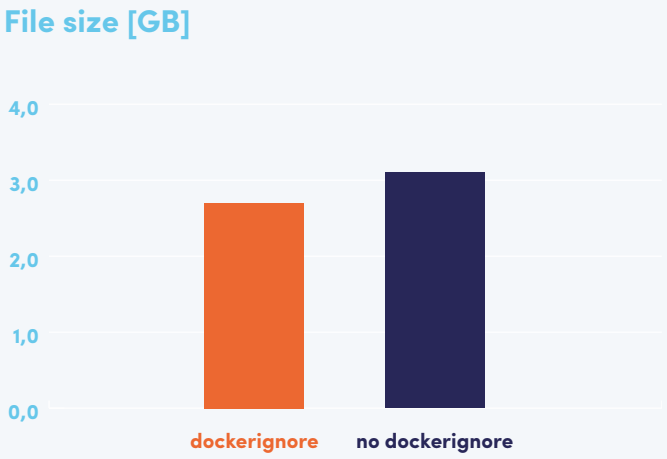
.dockerignore

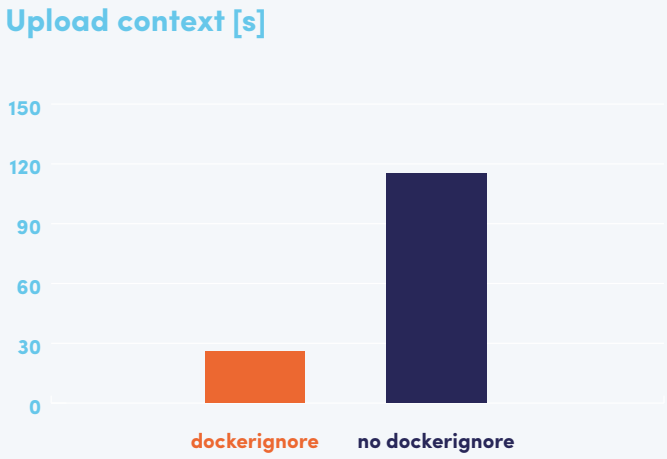
You probably know .gitignore right? You may even know .dockerignore but have you checked how much it can influence build times?

.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 build context. This is the very first step done while running a docker build. Have a look at Docker architecture and Docker build context chapters to learn more about that process.

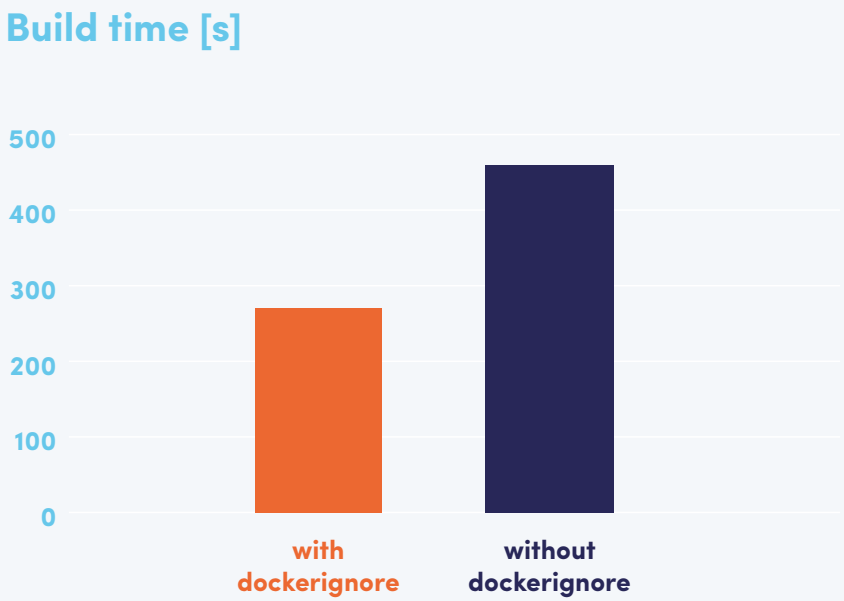
I don't think your local vendors, logs, cache should be involved in the build process. By adding them to .dockerignore, you save both time - it's not sent in the build context, it's not added to the layers - and you save space as those files might not be needed at all in the final image.

Below, you can see an example of how much time and disk space we saved in one of our projects by improving the .dockerignore file. It's astonishing how much you can achieve by such a simple change!





We saved around 90 seconds only on the build context sending phase. The difference between the initial and final version is huge, but it's mainly because I'm running on macOS and the build context phase was critical in this case. Also, on an Ubuntu laptop, the speed improvement is quite significant:



There is also a second benefit you can get by using .dockerignore - skipping files that should not be shared for **security reasons**. If you have read the Docker architecture mentioned above and build context chapters, you might have noticed that when running a Docker build, all local files are compressed and sent to the Docker daemon. In rare cases, the daemon might be on a different host and such a file transfer might be intercepted. But, let's forget about this unlikely event for now. 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.

Dockerfile instructions order

Often overlooked, yet very powerful - optimizing the order of instructions in your Dockerfile can have a tremendous impact on its build performance. The reason for this is how Docker layers work

Because each step of the build process is a new layer, nd each layer has a checksum that depends on the files in that layer, ordering the instructions properly will introduce huge benefits.

Let's take a look at the following Dockerfile:

```
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” ]
```

Quite straightforward - just a simple Dockerfile for a small Vue application. If we modify some files, we can easily re-build it to publish a new version:

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

In case of my Macbook Pro, it took 51s in total to rebuild

Such a Dockerfile might look familiar, in fact many of the projects I worked on had similar Dockerfiles. Yet, this is utterly suboptimal. Why? Because it won't make use of cache for the installation step, and as you might know - npm install takes a while.

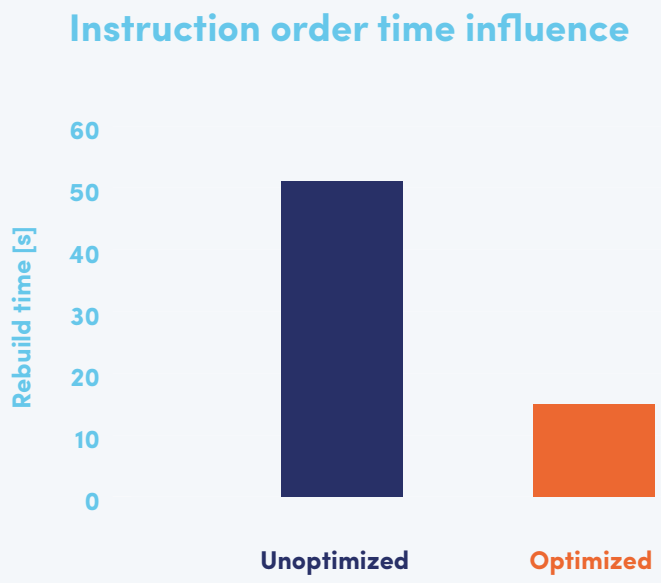
The result of npm install depends on two files: package.json and package-lock.json. Only changes in these two files will require running a npm install. If you fix a typo in an html file, why would you run the npm install again? You wouldn't, but in the case of the above Dockerfile, you do. That's because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

A fixed Dockerfile would look like this:

```
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” ]
```

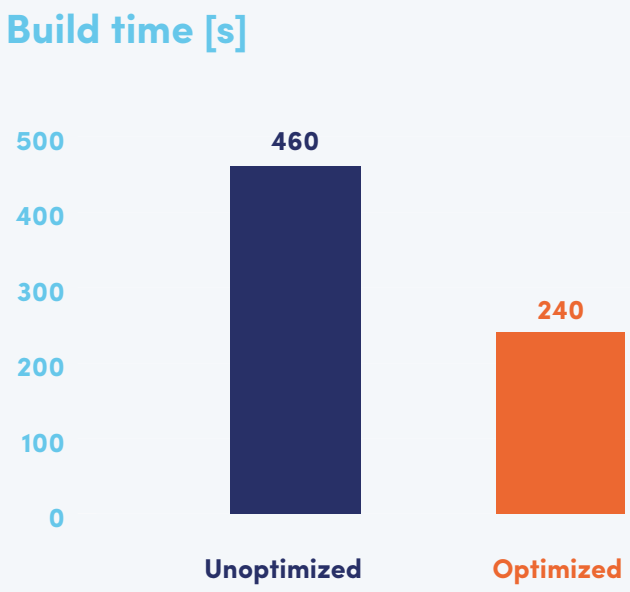
See the difference? It’s very subtle, and easy to implement.

But what's the difference in build time? Have a look:



So, in introducing this simple two-line change, we reduced **from 51s to only 15s**. That's almost **3,5 times faster**!

Now in regards to the projects I work on, there are also backend dependencies to be installed. It's accomplished by running Composer. The build time difference for both npm and composer is as follows:



So using this simple trick, I was able to optimize the build time by almost 50%, and save more than 3 minutes on each build!

Logs

I often see 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 John on Tuesday at 9:34 am. Without logs, you are doomed.

Using stdout, stderr and the logging driver

One of the simplest solutions to start with is 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.

In Kubernetes clusters, you will most probably even have a UI to browse logs of existing and old containers that use this approach.

Using data volumes and sidecar containers

Instead 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, but you can also add a sidecar container. What does a sidecar container do? With it, you can connect it to the same data volume as the main container. Inside, you have a process running that takes all logs 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

While working on different projects, I noticed two issues that can be connected with entrypoint.

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

2. Utterly slow entrypoints that take minutes!!! to start

Point 1

means there is a lack of automation, scripts, makefiles etc. My rule of thumb is that setting up a project (for development) that is dockerized should take no more than 3 steps. In a perfect world, this would be only one command `docker-compose up`, but I assume two more easy commands could also be ok.

Point 2

is very often the opposite of point 1. 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, I would suggest keeping 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.

For production purposes, keep the entrypoint as small as possible and remember - build once, run multiple times. Compilation, installation of vendors etc should be run once - during the build phase, not while starting the container.

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.

As far as your own image, the entrypoint init support might look like this:

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

If you work with Docker for a while, you probably hit some problems organizing Dockerfiles for development, tests and production. The most common issue is how to split the files, but still keep it close enough so both development and production are in sync. Ideally, you would like to keep them as similar as possible, but it's also obvious that they should not be 1:1 (you don't need a debugger in production, but need one in development). Multistage builds allow for better organization of the images, but they also **allow for the building of smaller images** in an easy way.

As you already know from the previous chapters - each layer is saved after it is built and stored in the image.
In order to keep the size small you need to remember to

clean up the artefacts that are not needed any more. But what happens if you need some artefacts in the next layer, and want to delete them afterwards? Sorry, that does not work as they are already saved in one of the layers.

There were multiple approaches to fixing this, but multi-stage is the most popular out there right now.

Multi-stage basics

When using multi-stage, 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. Moreover, you can copy files between builds. Let's take a look at an example.

We plan to build a Vue.js application container. The application does not offer SSR, so production-wise it holds only static files, and we can use Nginx to serve it.

```
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.

Thanks to multi-stage, 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"]
```

Using the COPY --from statement

It's worth mentioning that using the --from argument, you can both copy files from previous builds, as well as copy files from other images. You can easily copy the default nginx config by running:

```
COPY --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf
```

Naming stages and building images

You can, but you don't need to name each build stage.
The name is set using the AS part of the FROM statement:

```
FROM node:lts-slim AS development
```

As you probably noticed, this name is used in the COPY statements, but it is also used to build the right stage:

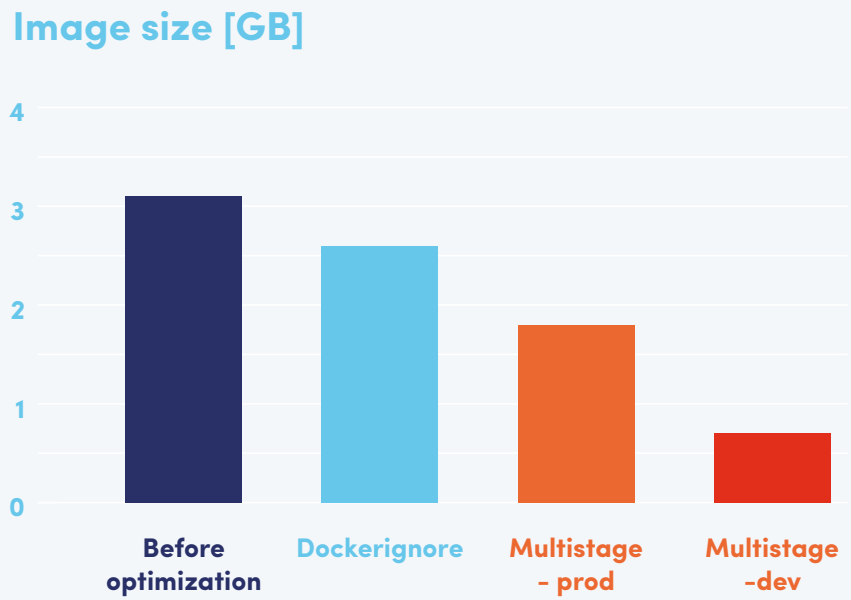
```
docker build -t multistage:development --target development
```

As a result, you can easily set what stage should be used. This can also be set in a docker-compose file:

```
VERSION: '3.8'
SERVICES:
     VUE:
          BUILD:
          CONTEXT: .
          TARGET: development
          PORTS:
               - 8080:8080
          VOLUMES:
               - ./:/usr/src/app
     VUE-PROD:
          BUILD:
          CONTEXT: .
          TARGET: production
          PORTS:
               - 80:80
          ENVIRONMENT:
               FOO: BAR
```

Example project

I tested all the described changes on one of the projects I helped to optimize Docker for. Below you can see a comparison of the image size after having taken these steps:



Here, I have focused on adding .dockerignore and introducing multi-stage to split prod and dev environments. Before my changes the image combined both prod and dev instructions, so a dev image was holding the app files and the prod image included dev tools – it doesn't make any sense!

As you can see, the initial image was over 3 GB, and after optimizations, the dev image was around 700 MB and the prod image was around 1200 MB. Still a lot, but a good starting point, as we managed to save 75% of space in the case of the dev image and 60% in the production image!

Make your SaaS scalabe!

At Accesto we help optimize and scale PHP&JavaScript applications.

READ CASE STUDIES +

SCHEDULE A CALL +

Container labels

Container labels won't make your container run any faster, or use less disk space. But, especially in larger organizations or teams, they might save debugging time. How? By adding useful information to an image. Labels may also simplify management. Take a look at this and select some annotations (labels) that might be helpful for you. Now 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 - it will make their life easier.

There is also another way how you can use labels. Labels can also be attached to containers, volumes, networks, etc. This practice is quite common in Kubernetes and helps a lot, but it can also be used in Docker. You just add labels to a container, e.g.:

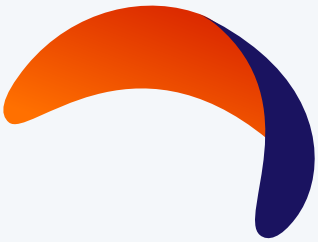
```
PROJECT: MYPROJECT
ENV: dev
```

and then you can use these labels to filter the docker ps output.

Example:

```
DOCKER RUN -L APP=NGINX NGINX
DOCKER PS --FILTER "LABEL=APP=NGINX”
```

```
➜  projects docker run -d -l app=nginx nginx
8e87f2cb4e7c0ce218e175335c5255424467dec6c3f24e6ae9abe11d98562e89
➜  projects docker ps --filter "label=app=nginx"
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
8e87f2cb4e7c        nginx               "/docker-entrypoint.…"   4 seconds ago       Up 3 seconds
➜  projects
```



3

Tools and commands

Dive

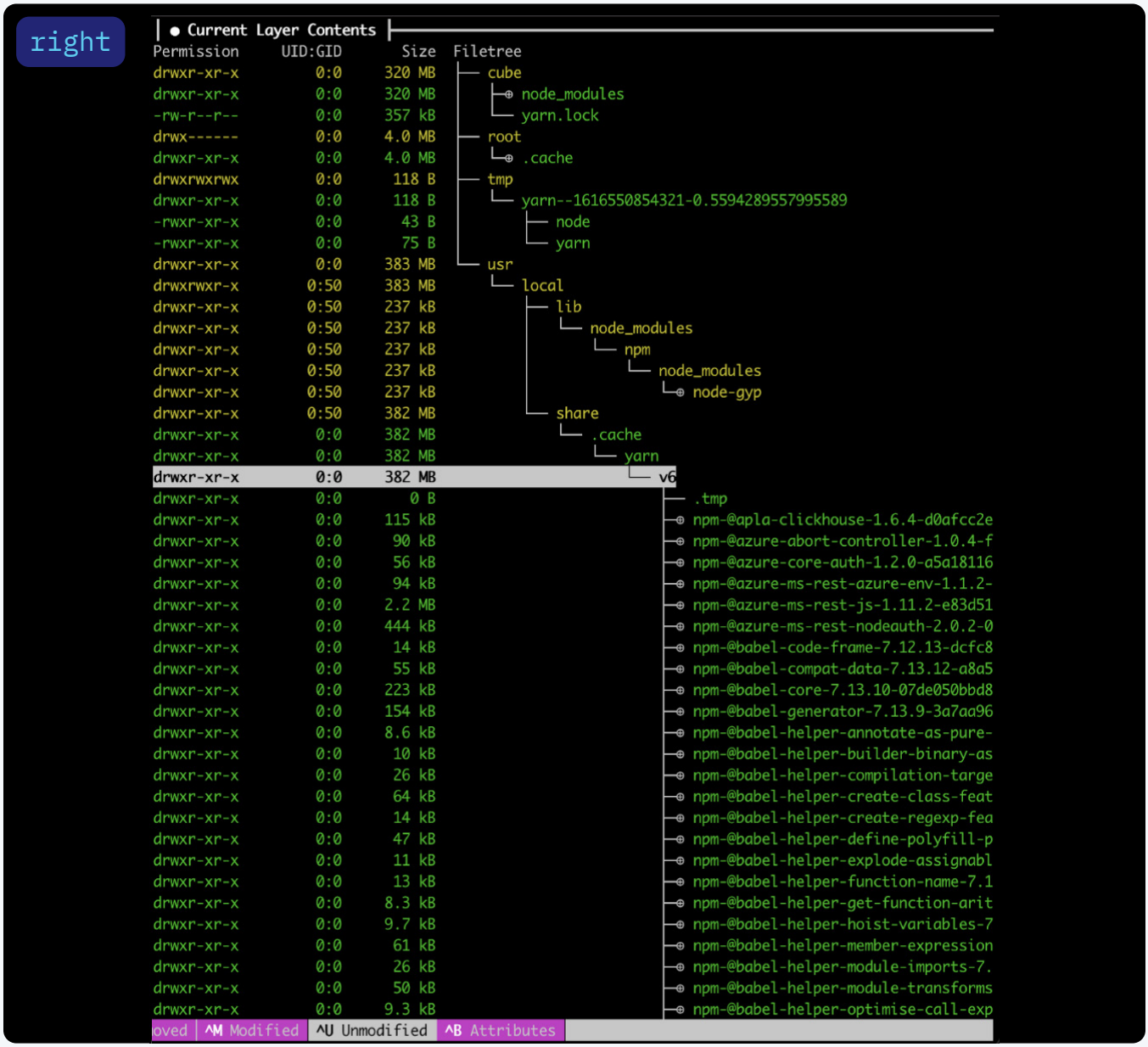
Dive is a great tool when you are working to optimize the image size and need more insight into why the image has taken some much space. It allows you to explore the docker image layers, and discover ways to shrink it.

The installation is quite straightforward and described in the repo linked above, so let's have a look at how dive can help you.

Example 1

I noticed that the official image for CubeJS is quite big, so let's delve a bit deeper into it:

On the left side, you get a list of the layers and they clearly inform you of how many MB each layer added to the overall size. On the lower left, you get a short summary of what the total image size is, and what the image efficiency score is. In this case, the efficiency score is 98%, so it's actually quite good.



On the right side, you can browse dirs and files added to see which files actually take so much space. I filtered the list to hide unmodified changes and collapsed all dirs to then expand only the directories I am 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 and I decided to follow that path. 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.

Example 2

This time let's just take a look at the left side:

One thing that I noticed is that 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.

Below that, you can see a list of files that you could probably delete. I think deleting /usr/bin/node would break the container, but I can see some .log and .dat files that look like good candidates for removal.

When you are familiar with a multi-stage build, your next step should be to extract the node and gem related stuff to a separate build phase and just copy the results. Such an upgrade will save you a lot of space and will also likely improve the way the cache is used.

Hadolint

Hadolint is a great every-day-use tool. As you might have guessed - 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 described in this book and will give you clear information about what is wrong. Let's check out how it works on an example project:

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

Hadolint works based on a provided Dockerfile:

```
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 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:

https://github.com/hadolint/hadolint/wiki/DL3020
https://github.com/hadolint/hadolint/wiki/DL3008

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 1s to finish and it's very easy to integrate with CI tools like Gitlab. I highly recommend adding it to your CI pipeline.

Kompose - convert docker-compose to Kubernetes

One of the questions I often get is how to migrate a project that has docker-compose to Kubernetes. If you google it, Kompose is probably one of the first results. Kompose just takes your docker-compose, yaml and converts it into a Kubernetes config.

While it may sound awesome, I would actually suggest against using it. It is a very nice tool to check the output but I don't think using it, especially in production, makes any sense, when your docker-compose is not a production one. Why? Here are the main reasons:

- It does not support some functions you might use locally:
  - local volume bindings
  - local Dockerfiles (instead of images)
- the names are obviously worse than ones created by people, and that makes the general output less readable
- does not generate secrets

In general, a conversion that a skilled person would perform, would be way better. I believe it's better to learn Kubernetes basics and transform the config on your own,

than to run Kompose and hoping everything will work. In most cases, it won't and you will need to adjust the files manually anyway.

Pumba

Ever heard about the chaos monkey? Pumba is a chaos monkey for Docker. If the term is new to you - it allows simulating different issues and problems like network outage, packet drops, freezing processes, etc.

While it may sound ridiculous, 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"
```

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/dock-
er.sock gaiaadm/pumba netem --duration 15s delay --time 3000
testDelay
```

If you are not familiar with the

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

part - this just allows the process inside the container to connect to your Docker deamon and manage it - in this case get access to our testDelay container.

```
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
```

As you can see, for 15 seconds, the ping got significantly higher - that's Pumba in action.

Afterword

And that's the end (at least for now)! I hope you enjoyed the book and learned some new useful stuff. Feel free to share this book with your friends and coworkers. Before you put the ebook aside and forget about it, I have two more suggestions for you.

Send me an email with your feedback, ideas for extension and some other thoughts about this ebook. Any feedback is more than welcome!

Michal Kurzeja in

CTO @ Accesto

CONTACT ME +

If you did not before, then sign-up for our newsletter - we will share our posts and future updates of this ebook.

Make your SaaS scalabe!

At Accesto we help optimize and scale PHP&JavaScript applications.

READ CASE STUDIES +

SCHEDULE A CALL +

accesto