Docker should be fast, not slow: a practical guide to building fast, small docker images

A software article by Efron Amber Licht

July 2023

Part 2 of a series on starting software.

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

Your average build process is glacially slow. I firmly believe most projects build between 50-200x slower than they should, and produce images 10-100x larger than they should. This costs your average software house hundreds of thousands of dollars a year in wasted developer time and hardware/cloud costs. Docker was a tool designed to help solve this problem, but it's widely misused in a way that makes the problem worse, not better. In this article, we'll talk about the costs of slow builds, quickly review the basics of Docker, and then annotate & optimize a real-world dockerfile: the one that generates the very website you're reading! We'll get a `50x` speedup in build time, and a nearly `100x` reduction in image size.

This article is thematically connected to my previous article,), but you don't need to read that to understand this one.

Motivation & back-of-the-envelope math

Why are fast builds important? As a developer, this is obvious. It's infuriating to play "hurry-up-and-wait", making a tiny change and then waiting minutes or hours to see whether or not it worked. But 'annoyance' does not always equal 'priority'. Most software is built at for-profit companies, and the quickest way to get a company to care about something is to show them the money.

Let's pretend we're an exec at a California-based software company. Our developers work 1920 hours a year (48 weeks, 40 hrs/week). We pay them \(140k/year, or \)73/hr, plus 20% for overhead and benefits, so it costs us roughly $85/hr/dev.

They spend about an hour a day waiting for builds to finish. This is a low estimate, in my experience, but let's be conservative. That's 240 hr/year, or **$20,400/dev/year**.

Let's look at those costs in a table, for different numbers of developers and different amounts of time spent waiting for builds:

devs	build time (minutes/day)	cost/year, in thousands of dollars
10	60	$204K
10	30	$102K
10	6	$20K
20	60	$408K
20	30	$204K
20	6	$40K
50	60	$1,020K
50	30	$510K
50	6	$102K
200	60	$4,080K
200	30	$2,040K
200	6	$408K

This is *just* the cost of developer time. It doesn't include the cost of machine-hours on your CI provider, or the spiraling costs of having to use bigger and bigger machines to *run* code on those bloated containers, which can approach or even exceed the cost of developer time. It also doesn't include the cost of developer morale, which is hard to quantify but very real. How much does it cost to replace a developer who quit because he couldn't stand waiting for builds anymore?

From my point of view, you *can't afford not to optimize your build process* - that's leaving hundreds of thousands of dollars on the table! And yet almost no one does. We build slow, bad software slowly, and we pay through the nose for it, and then we pat ourselves on the back for being so clever, for using the 'cool' new tools, for being 'cloud native'. We prioritize the *appearance* of a good build process over the *reality* of one.

Docker is obviously not the only reason builds are slow, but nowadays it's so omnipresent throughout the build process that it's a good place to start. We develop in docker containers, we test in docker containers, we build in docker containers, we deploy in docker containers. If we can understand how to make docker builds fast & cheap, we should realize those benefits throughout the entire build process.

Docker review

Let's quickly review the basics of Docker. This article isn't meant to be a docker tutorial, but a little refresher never hurt anyone.

Docker crash course

You can think of Docker Container as a virtual machine made by applying layers of commands to a base disk image. A dockerfile is a 'recipe' for building a container, either starting from scratch, or from a known-good recipe (called a 'base image')

Each layer is a command, like `RUN` or `COPY`, that executes in order. When you call `docker build`, Docker executes each command in order. Assuming they all succeed, the resulting container (disk image) is cached and ready to be run.

This can sound a little abstract. Let's build a simple Go program the 'normal' way, then build it in Docker.

Our program is a simple 'hello world':

```
1  package main
2  import "fmt"
3  func main() {fmt.Println("Hello, docker!")}
```

If we were to try and run this program on a fresh install of Ubuntu, we'd need to:

- Install the go compiler.
- Write our program to a file.
- Compile our program.
- Run the program.

Which we'd do like this:

IN

```
1  # use the apt package manager to install go
2  sudo apt-get update && sudo apt-get install -y golang
3  # write our program to main.go
4  echo 'package main; import "fmt"; func main()
   {fmt.Println("Hello, docker!")}' > main.go
5  go build -o helloworld main.go
   # compile our program into an executable called helloworld
```

```
./helloworld # run our program
```

OUT

```
Hello, docker!
```

Docker works pretty much the same way.

```
FROM ubuntu:20.04
```

FROM chooses a base image: this is a 'fresh' install of ubuntu

```
RUN apt-get update && apt-get install -y golang
RUN echo 'package main; import "fmt"; func main() {fmt.Println("Hello, docker!")}' > main.go
RUN go build -o helloworld main.go # compile our program into an executable called helloworld
```

`RUN` executes a command in the container. Here, we install go, write our program to a file, and compile it.

```
ENTRYPOINT ./helloworld # run our program
```

Entry point is the command that runs when we run the container. Here, we run our program.

We build a docker image from this Dockerfile like this:
IN

```
docker build -t helloworld . # -t names our image 'helloworld', and '.' tells docker to look for a Dockerfile in the current directory
docker run helloworld # run the image tagged 'helloworld'.
```

70+ seconds later, we get our output:

OUT

```
Hello, docker!
```

`echo` is hardly a reasonable way to write a Go program. We can `COPY` files from our host machine into the container, so let's write our program on our host machine and copy it in.

```
// in main.go
package main
import "fmt"
func main() {
```

```
5       fmt.Println("Hello, docker!")
6   }
```

```
1   FROM ubuntu:latest
2   RUN apt-get update && apt-get install -y golang
3   COPY ./main.go ./main.go # copy main.go from the host
    machine to the container
4   RUN go build -o helloworld main.go # compile our program
5   ENTRYPOINT ./helloworld # run our program
```

IN

```
1   docker build -t helloworld .
2   docker run helloworld
```

OUT

```
1   Hello, docker!
```

It works... and it ran in only **0.9s!** Why? Because docker doesn't just **build** layers, it **caches** them. Our first `docker build` had to do these steps:

1. Download the ubuntu image's layers
2. Assemble those layers into a disk image
3. Run `apt-get update` and `apt-get install -y golang`, installing Go and it's many dependencies
4. Write our program to a file
5. Compile and run our program

But our second `docker build` was able to re-use the stored cache from the first build and start in the middle. This is why Docker *can* be fast: if done properly, you can only rebuild the parts of your container that have changed. A small, well-ordered Dockerfile can be *very* fast.

Note 'well-ordered', though. If we'd written our dockerfile like this:

```
1   FROM ubuntu:latest
2   COPY ./main.go ./main.go # copy main.go from the host
    machine to the container
3   RUN apt-get update && apt-get install -y golang
4   RUN go build -o helloworld main.go # compile our program
5   ENTRYPOINT ./helloworld # run our program
```

With this dockerfile, since `COPY` is before `RUN apt-get update && apt-get install -y golang`, we'd have to re-download and re-install Go every time we changed our program - adding 30s or more to each and every build. And this design is the *norm*, not the exception. **Most Dockerfiles are written in a way that makes them extremely slow to build.**

And more than that, they're extremely large:

IN:

```
docker images
```

OUT

```
REPOSITORY   TAG       IMAGE ID       CREATED
SIZE
helloworld   latest    933765bdb4bd   About an hour ago
860MB
```

But this too is a fixable problem. We can separate the image that contains all the tools we need to build our program from the image that contains our program. This is called a 'multi-stage build'.

Rather than starting from `ubuntu:latest`, we can have a *builder* stage that starts from `golang:latest` and compiles our program, then a *run* stage that starts from `ubuntu:latest` and copies our compiled program from the builder stage.

We could write our Dockerfile like this:

```
FROM golang:latest as builder
COPY ./main.go ./main.go
RUN go build -o /helloworld main.go
FROM ubuntu:latest
COPY --from=builder /helloworld ./helloworld
ENTRYPOINT ./helloworld
```

This is pretty much the same as our previous Dockerfile, but with a few changes:

- We start from `golang:latest` instead of `ubuntu:latest`, so we don't have to install Go.
- We give our first stage a name: `as builder`. This is so we can refer to it later.
- We have a second stage, which starts from `ubuntu:latest`, and *never installs the Go compiler*.
- We use `COPY --from=builder` to copy our compiled program from the builder stage to the run stage.

Let's check how long this takes to build:

IN

```
time docker build -t multiplestage .
```

3.548s

And how big is the image?

```
docker images
```

```
REPOSITORY   TAG       IMAGE ID       CREATED
SIZE
multiplestage   latest    c64446a469ee   26 seconds
ago       79.7MB
```

base image	builder image	cache?	Time	Size
`ubuntu:latest`	<—	no	70.0s	860MB
`ubuntu:latest`	<—	yes	0.9s	860MB
`golang:latest`	`ubuntu:latest`	no	3.5s	79.7MB
`golang:latest`	`ubuntu:latest`	yes	0.81s	79.7MB

This is a *huge* improvement. We've gone from 70+ seconds to 3.5 seconds, and from 860MB to 79.7MB.

Quick Dockerfile review

Now that we've taken a peek at a couple of toy examples, let's review dockerfile syntax. Then we'll get down to *real* optimization on this blog's dockerfile.

`FROM` chooses a base image

This is the starting point for your container, usually either a linux distribution or a distribution with some tools added in. You can think of it as a save state of a virtual machine. If you're using a base image that's already on your machine, it's loaded from the cache. Otherwise, it's downloaded from the internet. All dockerfiles either are `FROM scratch` or `FROM` another dockerfile.

`COPY` copies files from a host machine into the container

The host is usually your local machine, but it can be another container earlier in the Dockerfile. If the file is the same as the last time you ran `COPY`, the cache is preserved. Otherwise, the cache is invalidated. **blanket copies (`COPY . .`) and copies too early in the Dockerfile are the most common cause of slow builds.**

`RUN` runs a command in the container

This is usually but not always a shell script, invoked as though using `sh -c`. That is, it works the way
you'd expect shell commands to: you can join commands with &&, etc.

```
RUN mkdir -p foo/bar && touch foo/bar/baz
```

Is equivalent to

```
sh -c 'mkdir -p foo/bar && touch foo/bar/baz'
```

Which is the same as running this in `sh` or `zsh` or `bash`:

```
mkdir -p foo/bar && touch foo/bar/baz
```

`RUN` will not usually invalidate the cache, but slow or unnecessary commands will slow down your build.

- `### ENTRYPOINT` specifies the command that actually executes with `docker run`

Dockerfile performance: a few rules of thumb

Dockerfiles should be ordered from least-frequently-changed to most-frequently-changed

Usually, dependencies change less frequently than source code, and source code changes less often than assets.

Dockerfiles should be as specific and granular as possible

The more specific a layer is, the less likely it is to be invalidated. `COPY` only the bits you need, immediately before you need them. Don't copy your entire source tree if you only need one file. Don't copy your entire assets directory if you only need one image. Don't copy your entire dependencies directory if you only need one binary. Avoid `node` and other enormous package managers that install hundreds of dependencies.

Dockerfiles should be as small as possible

The image needs to be downloaded and cached on every machine that uses it. This takes time and space. Small is fast.

The Image that builds your application should be different from the image that runs your application

You normally need far more tools to BUILD your application than you do to RUN your application. Build your application in one image, and copy the minimal subset needed to run it into a second image. In other words, ship the pizza, not the oven.

That should be enough to start.

Annotated Dockerfile for `eblog`

In the rest of this post, we'll walk through the Dockerfile for this blog, and explain how it works and why it's structured the way it is. Then we'll optimize it even further.

Eblog (this site) works like this: I write articles (like this one) in markdown. Three go programs I wrote in the `cmd` directory convert the markdown to html (`rendermd`), generate the index page (`buildindex`), and zip up those rendered assets into a zip fil (`prezip`). That zip file is embedded into the binary for `eblog` itself. Then that binary is copied to a minimal docker image and uploaded to `fly.io`, which actually serves the site and handles DNS, etc.

We can thus break the build process into three stages:

- **Tooling** builds the commands in `cmd`.
- **Build** renders the assets using those tools and builds the `eblog` binary.
- **Run** actually runs the `eblog` binary.

I'll put the whole dockerfile here, then we'll walk through it line by line.

```
1   # start from a minimal golang image.
2   FROM golang:1.20.5-alpine3.18 as tooling
3   # set GOPATH to an empty string to force go to use module
    mode,
4   # and set CGO_ENABLED to 0 to disable cgo, which we don't
    need.
5   ENV GOPATH="" CGO_ENABLED=0
6   # install binutils, which contains the strip tool we'll use
    to remove debug symbols from our binaries
7   RUN apk add --no-cache binutils
8
9   # copy in the list of dependencies and source for tools and
    libraries
10  COPY ./go.mod ./go.mod
11  COPY ./go.sum ./go.sum
12  COPY ./observability ./observability
13  COPY ./cmd ./cmd
14  RUN go mod download # download dependencies, using the
    cache
15  # build tools
16  RUN go build -o rendermd -trimpath ./cmd/rendermd &&\go
    build -o buildindex -trimpath ./cmd/buildindex\
17  && go build -o prezip -trimpath ./cmd/prezip
18
19  # strip debug symbols from our tool binaries to reduce
    their size & remove binutils, since we're done with it
20  RUN strip ./rendermd ./buildindex ./prezip\
21  && apk del binutils
22
23
24  FROM tooling as builder
25  # copy in the source code for the app and the articles
26  COPY ./server ./server
27  COPY ./articles ./articles
28  # get the current commit hash, for logs. doing this by hand
    means we don't have to install git.
29  COPY .git/logs/refs/heads/master server/commit.txt
```

```
30  # run our tools to build the static assets for our app so
    we can embed them in the final binary
31  RUN ./rendermd ./articles ./server/static\
32  &&  ./buildindex ./server/static\
33  &&  ./prezip ./server/static > ./server/static/assets.zip
34  # build the server
35  RUN go mod download\
36  && go test ./server/...\
37  && go build -o /app -trimpath ./server
38
39  # start from an even more minimal base image
40  FROM alpine:3.18 as run
41  # expose port 8080 so the webserver can listen to it
42  EXPOSE 8080
43  # copy in our app from the previous stage
44  COPY --from=builder /app /app
45  # add ca-certificates to allow https requests. note that we
    do this AFTER the copy: we always want the latest certs,
    even if the app hasn't changed!
46  RUN apk add --no-cache ca-certificates
47  ENTRYPOINT ./app
```

Stage 1: Tooling

```
1   FROM golang:1.20.5-alpine3.18 as tooling
```

Nothing new here. We use the `golang:1.20.5-alpine3.18` image as our base image, and we give it a name so we can refer to it later.

Our golang:1.20.5-alpine3.18 base image contains the golang compiler & tools at a svelte **99.77 MiB**.

The standard `golang` images are based off debian and take 314.84 MiB.

Always start with the smallest base image you can, and work your way up. You can add more layers, but you can't take them away. I prefer fixed tags to `latest`, since I don't want my builds to change out from under me.

```
1   ENV GOPATH="" CGO_ENABLED=0
```

Environment variables persist from the line they're defined to the end of the Dockerfile, even across FROM statements, which can be a bit counterintuitive.

- `CGO_ENABLED=0` allows our final binary to run without glibc, saving us a ton of space on our final stage.
- `GOPATH=""` makes Go always use module mode (the usual way to use Go these days), simplifying our build process.

```
1   RUN apk add --no-cache binutils
```

APK is alpine's package manager, very similar to `apt`, `yum`, `pacman`, `brew`, etc. Most dockerfiles need some kind of binary dependencies. When installing them, try to:

- `RUN` those installs *before* copying in your source code to avoid invalidating cache.
- install the **minimal subset** of packages you need to build your app.
- Set your package manager to `--no-cache` mode to avoid bloating your image with package metadata (or alternatively, delete the cache after installing your packages).
- Remove tools you don't need after you're done with them.

In this case, we're using the `strip` tool from binutils to remove debug symbols from our tool's binaries to shrink them down a bit. This isn't really necessary, but I wanted an example of a binary dependency.

```
1  COPY ./go.mod ./go.mod
2  COPY ./go.sum ./go.sum
3  COPY ./observability ./observability
4  COPY ./cmd ./cmd
```

We copy the bare minimum of files you need to execute the next layer of code. Specifically, we want to copy in `cmd` and the dependencies for `cmd`, but not `server` or `articles`, which are more likely to change. Docker has no way of knowing which files you actually need, so *any* change to *any* copied file invalidates that layer. Remember, copy **as little as possible, as late as possible**.

```
1  RUN go mod download
```

We download our dependencies here, so that we can cache them. They're specified in `go.mod` and `go.sum`, which we copied in earlier. This step is cached, so it will only run again if those files change.

```
1  RUN go build -o rendermd -trimpath ./cmd/rendermd\
2  && go build -o buildindex -trimpath ./cmd/buildindex\
3  && go build -o prezip -trimpath ./cmd/prezip
```

We build our tools here. We use the `-trimpath` flag to remove the absolute path to our source code from the binaries, which would otherwise be included by default. This step is also cached, so it will only run again if the source code changes.

```
1  RUN strip ./rendermd ./buildindex ./prezip\
2  && apk del binutilst
```

This step removes debug symbols from our binaries, saving us a bit of space, then removes the `binutils` package we installed earlier to clean up after ourselves. Removing the tools you need after you're done with them is a good way to keep your image size down.

Stage 2: Build

At this point, we have all the tools we need to build our app. Because our libraries and tools are unlikely to change, most of the time, these layers will remain cached, and our builds will be fast. Everything that follows handles application code or assets, which are more likely to change than stand-alone programs like `rendermd`.

```
FROM tooling as builder
```

A new FROM statement starts a new stage. Here, we're just starting from where the previous stage left off, but we could have started from a completely different base image if we wanted to (and we will, in step 3).

```
COPY ./server ./server
COPY ./articles ./articles
```

As usual, we copy in the bare minimum of files we need to execute the next layer of code. In this case, we're copying in our application code and our articles, which are both needed to build our app every time, since I `embed` them in the final binary. Embedding assets leads to fast serve and boot times.

```
RUN ./rendermd ./articles ./server/static \
&&  ./buildindex ./server/static \
&&  ./prezip ./server/static > ./server/static/assets.zip
```

We run the tools...

```
RUN go mod download \
&& go build -o /app -trimpath ./server
```

... and build our app.

Stage 3: Run

Almost there.

```
FROM alpine:3.18 as run
EXPOSE 8080
COPY --from=builder /app /app
RUN apk add --no-cache ca-certificates
ENTRYPOINT ./app
```

Our final image just needs to run our application, not build it, so we can start from an even more minimal base image. `alpine:3.18` is only 5MiB! We copy in our binary from the previous stage, install `ca-certificates` for the latest TLS certs, and run our app. Note that unlike before, we want to add the certificates *after* copying in our binary, since we don't want to have outdated certs.

Results

I timed the builds with `time docker build .`, using the `real` time as the metric, and I used `docker images ls` to get the size of the final image. I also did a few runs using debian-based images to see what using alpine savesus.

Here, 'cache' means that we only had to rebuild starting from `builder`. 'no-cache' means that we had to rebuild starting from `tooling`. I'm not counting the initial download of the base image in either case.

Dockerfile	Time	% max	Space	% max
golang:latest (1 stage)	23.375s	100%	1340 MiB	100%
golang:alpine -> alpine, start at tooling	19.036s	81%	15.5 MiB	1.16%
golang:alpine -> alpine, start at build	4.243s	18%	15.5 MiB	1.16%

No, that's not a typo. We get a **5x speedup** for cached builds, and a **86x** size reduction for the final image compared to a naive 1-stage

... and we're still not done. **Not even close**.

Further optimizations: host-side caching

In some cases, we may be able to cache even more aggressively. For example, the `go mod download` tool shouldn't have to redownload *every* dependency when one changes: instead, it should only have to redownload the *changed* dependencies. Similarly, we'd like to reuse the go tool's build cache to avoid recompiling unchanged dependencies. The mount option during the `RUN` command lets us do this. We can mount a directory on the host machine into the container, and the container can read and write to it. This lets us share the cache between builds, and even between different projects.
Go stores it's build cache by default in `/root/.cache/go-build`, and it's module cache in `/go/pkg/mod`. By using the `--mount` flag during appropriate `RUN` commands, we can mount these directories from the host on to the container, reading and writing from them as if they were local directories. This lets us share the cache between builds, (and even between different projects).

We change our go mod download command to:

```
1  RUN --mount=type=cache,target=/go/pkg/mod/ go mod download
```

And our `go build` commands:

```
1  RUN --mount=type=cache,target=/root/.cache/go-build go
   build -o rendermd -trimpath ./cmd/rendermd\
2  && go build -o buildindex -trimpath ./cmd/buildindex\
3  && go build -o prezip -trimpath ./cmd/prezip
```

See the actual dockerfile on my gitlab for the full version.

Let's run it and see what happens:

build/tooling	dockerfile start	warm cache?	Time	% max	Space	% max
`golang:latest`	tooling	❌	`23.375s`	100%	`1340 MiB`	`100.00%`
`golang:alpine`	tooling	❌	`19.036s`	81%	`15.5 MiB`	`1.16%`
`golang:alpine`	tooling	✅	`7.070s`	30%	`15.5 MiB`	`1.16%`
`golang:alpine`	build	❌	`4.432s`	10%	`15.5 MiB`	`1.16%`
`golang:alpine`	build	✅	`2.420s`	2%	`15.5 MiB`	`1.16%`

If we go the whole hog and `strip` the final `app` binary, we get to 12.5MiB, **less than 1%** of the size of the naive build. (But I wouldn't recommend it: it's good to leave debug symbols in your production binaries. A 'mere' 86x improvement is just fine.)

If we go the whole hog and `strip` the final `app` binary, we get to 12.5MiB, **less than 1%** of the size of the naive build. (But I wouldn't recommend it: it's good to leave debug symbols in your production binaries. A 'mere' 86x improvement is just fine.)
… That's right. **50x** speedup, and **86x** less space. We could do even more, but I think I've made my point. **How you build your docker images matters.**

As dramatic as these numbers are, I honestly believe they're the norm, not the exception. `eblog` is a hobby project deliberately designed to have as few dependencies and features as possible. Most software projects are far less disciplined in design and have sprawling dependency graphs. They have far more to lose from a poorly designed build process and far, far more to gain from a good one. I have personally got >30x speedups and >300x size reductions on corporate projects.

Conclusion

There's no reason your docker builds can't be fast and your deployments small. It just takes a little bit of work to get there. Keep an eye on your build process and you'll reap the rewards of faster builds and smaller images… and the significant cost savings that come with them. I hope you enjoyed the article! I'm not done talking about starting up fast, though. We've covered Docker and Boot time, but there's plenty left to talk about.

MORE ARTICLES