**The complexity of writing an efficient NodeJS Docker image**

[Samuel Bodin](https://www.linkedin.com/in/bodinsamuel/)

Aug 08, 2023

Docker is an invaluable tool for managing containers, but it can be **challenging to grasp** its inner workings, especially for newcomers. While modern hosting platforms like Vercel and Netlify often eliminate the need for writing Dockerfiles for NodeJS processes, understanding Docker optimization becomes crucial when handling your infrastructure. By achieving optimal efficiency, you can **save both time and money** in the long run.

**tl;dr** it's possible to save up to 80% of time to build and image size, but the complexity is not always worth it.

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-a-brief-overview-of-dockers-internals)**A Brief Overview of Docker's Internals**

Before diving into the optimization techniques, let's briefly explore how Docker operates internally. If you're primarily interested in the end result, feel free to skip this section. Docker employs a **layered approach**, where each modification to the filesystem creates a new layer on top of the previous one. Imagine a delicious mille-feuille pastry, where each layer adds to the overall flavor. Except **you actually want less** flavor in this scenario.

# Creates a Layer

FROM node

# Creates a Layer

WORKDIR /app/tmp

# Creates a Layer

RUN apk update && apk add curl

# Creates a Layer

COPY . /app/tmp

RUN echo "hello world"

This layering strategy allows Docker to **cache** the results of each command. However, there is a caveat: these layers are included in the final Docker image during the build process. Consequently, if unnecessary layers are present in the build, and the image undergoes multiple intermediate steps, more layers will be **transmitted over the wire**.

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-context)**Context**

In a typical NodeJS build process within Docker, several steps are involved:

1. Use the correct Node base image
2. Copy the source code
3. Install the dependencies
4. Build/Compile the source code
5. Run the process

While these steps are generally necessary, it's worth examining if all of them are truly required. Upon closer inspection, we can identify two layers in the process that can be eliminated from the final Docker image. First, the step of "copying the source code" becomes irrelevant once we have compiled it, as the source is no longer utilized. Second, the installation of "devDependencies" is only necessary during development and is not required to run the process in a production environment.

To illustrate these optimization techniques, let's consider an example using Specfy's own backend, which utilizes common tools such as Eslint, Typescript, Fastify, Vite, and Prisma. I will be using a Macbook Pro 2023 M2 14".

Now, let's delve into the optimization steps.

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-1-initial-dockerfile)**1. Initial Dockerfile**

Now that we understand the context and the optimization goals, we can begin by creating a simple Dockerfile that launches our NodeJS process.

Here's our base:

FROM node:18.16.1

WORKDIR /app/tmp

# Copy source code

COPY . /app/tmp

# Install the dependencies

RUN npm install

# Build the source code

RUN true \

&& cd pkgs/api \

&& npm run prod:build

EXPOSE 8080

To build from scratch every time you need to docker system prune and docker build --pull --no-cache ./

The output will look like this:

[+] Building 106.8s (10/10) FINISHED

=> [internal] load build definition from Dockerfile 0.0s

=> => transferring dockerfile: 200B 0.0s

=> [internal] load .dockerignore 0.0s

=> => transferring context: 35B 0.0s

=> [internal] load metadata for docker.io/library/node:18.16.1 37.8s

=> [1/5] FROM docker.io/library/node:18.16.1@sha256:f4698d49371c8a9fa7dd78b97fb2a532213903066e47966542b3b1d403449da4 8.8s

=> => resolve docker.io/library/node:18.16.1@sha256:f4698d49371c8a9fa7dd78b97fb2a532213903066e47966542b3b1d403449da4 0.0s

=> => sha256:52cd8b9f214029f30f0e95d330c83ab4499ebafaac853fdce81a8511a1bdafd7 45.60MB / 45.60MB 7.8s

=> => sha256:f0eeafef34cdd8f38ffee08ec08b9864563684021057b8e61120c237dd108442 2.28MB / 2.28MB 1.4s

=> => sha256:55a1a1affff207f32bb4b7c3efcff310d55215d1302bb3308099a079177154e7 450B / 450B 1.4s

=> => sha256:f4698d49371c8a9fa7dd78b97fb2a532213903066e47966542b3b1d403449da4 1.21kB / 1.21kB 0.0s

=> => sha256:9ffaff1b88cf5635b1b79123c9c0bbf79813da17ec6b28a7b215cab4d797a384 2.00kB / 2.00kB 0.0s

=> => sha256:8e80a3bf661ba38d6d8b92729057dddd71531831f774f710cb4bf66c818a370c 7.26kB / 7.26kB 0.0s

=> => extracting sha256:52cd8b9f214029f30f0e95d330c83ab4499ebafaac853fdce81a8511a1bdafd7 0.8s

=> => extracting sha256:f0eeafef34cdd8f38ffee08ec08b9864563684021057b8e61120c237dd108442 0.1s

=> => extracting sha256:55a1a1affff207f32bb4b7c3efcff310d55215d1302bb3308099a079177154e7 0.0s

=> [internal] load build context 0.1s

=> => transferring context: 68.01kB 0.1s

=> [2/5] WORKDIR /app/tmp 0.2s

=> [3/5] COPY . /app/tmp 0.1s

=> [4/5] RUN npm install 47.4s

=> [5/5] RUN true && cd pkgs/api && npm run prod:build 8.6s

=> exporting to image 10.3s

=> => exporting layers 10.3s

=> => writing image sha256:b025ac6558b78d16781531ba1590abcdeb8c1fe98243cef383938b3f2b40c007 0.0s

=> => naming to docker.io/library/test 0.0s

**Results**:

From Scratch

106.8s

Partial Cache

69s

Full Cache

0.9s

Image Size

2.48GB

It's worth noting that Docker cache hit is highly efficient and can significantly reduce the build time, often eliminating around 99% of the initial build time.

However, one drawback is the large size of the final image, which **currently stands at 2.48GB**. This size is far from optimal and should be addressed to improve efficiency.

In the next steps, we will focus on optimizing the Docker build process to reduce the image size while maintaining the necessary functionality.

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-2-dockerignore)**2. .dockerignore**

This is the initial step. We won't go too much into details but it's very important to ignore some folders and can drastically change the build time or size. Especially if you are building locally. An other important aspect is to not push your [secrets in the image](https://www.bleepingcomputer.com/news/security/thousands-of-images-on-docker-hub-leak-auth-secrets-private-keys/). Even if you delete them inside, they will live in the intermediate layers.

# Private keys can be in there

\*\*/.env

\*\*/personal-service-account.json

# Dependencies

\*\*/node\_modules

# Useless and heavy folders

\*\*/build

\*\*/dist

[...]

\*\*/.next

\*\*/.vercel

\*\*/.react-email

# Other useless files in the image

.git/

.github/

[...]

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-3-using-slim-or-alpine-images)**3. Using Slim or Alpine images**

After this basic step, we can start optimizing the build by using Slim or Alpine image. While [Alpine](https://www.docker.com/blog/how-to-use-the-alpine-docker-official-image) images are often recommended for their lightweight nature, it's important to exercise caution, especially with NodeJS applications.

Alpine images utilize musl libc, which operates slightly differently from the regular glibc. This difference can introduce issues such as segfaults, memory problems, DNS issues, and more, particularly when working with NodeJS. For example, at my previous job Algolia, we faced DNS errors while using Alpine in the Crawler (as mentioned in my [blog post](https://www.algolia.com/blog/engineering/30-days-to-improve-our-crawler-performance-by-50-percent/)) and Segfault when using Isolated V8.

Considering these concerns, it's advisable to choose **Slim** images, which offer less benefits in terms of reduced image size but offers more stability and compatibility.

FROM node:18.16.1-bullseye-slim

WORKDIR /app/tmp

# Copy source code

COPY . /app/tmp

# Install the dependencies

RUN npm install

# Build the backend

RUN true \

&& cd pkgs/api \

&& npm run prod:build

EXPOSE 8080

**Results**:

From Scratch

57.5s

from 106.8s

47%

Partial Cache

45s

from 69s

34%

Full Cache

1.0s

Image Size

1.61GB

from 2.48GB

36%

That was an easy step and we managed to reduce our image size by 36%. Classic distribs contains a lot of unecessary tools and binaries that are removed in Slim distribs. Additionally, the build time improved by 47% (~40seconds), amazing especially considering that it was achieved with just one line of modification. Please note that this base image is still ~230MB (uncompressed), so we will never reach lower than that.

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-4-multi-stage-build)**4. Multi Stage build**

We now turn our attention to the multi-stage builds trick. For those unfamiliar with this concept, multi-stage builds are like **portals within a Dockerfile**. You can simply change the context entirely but you are allowed to keep some things with you, and only the final stage will be sent. This is perfect to copy the compiled source and completely removes node\_modules or remove some part of the source code.

Running RUN rm -rf node\_modules will actually create a new layer of 0B and not reduce the final image size

# ------------------

# New tmp image

# ------------------

FROM node:18.16.1-bullseye-slim as tmp

WORKDIR /app/tmp

# Copy source code

COPY . /app/tmp

# Install the dependencies

RUN npm install

# Build the backend

RUN true \

&& cd pkgs/api \

&& npm run prod:build

# ------------------

# Final image

# ------------------

FROM node:18.16.1-bullseye-slim as web

# BONUS: Do not use root to run the app to avoid privilege escalation, container escape, etc.

USER node

WORKDIR /app

COPY --from=tmp --chown=node:node /app/tmp /app

EXPOSE 8080

On this step we have changed a bit more things, we added a second FROM that tells Docker this is a multi-staged build and a COPY. This is portal we were talking about, we created a second image, where the only filesystem operation is a copy. So we removed all others intermediate layers and if what we copy is small we saved a lot.

**Results**:

From Scratch

65.2s

from 106.8s

38%

Partial Cache

50s

from 69s

27%

Full Cache

1.2s

Image Size

1.32GB

from 2.48GB

46%

In this example I copied all the files from the previous steps, but we still managed to **save ~300MB**, at the relative cost of few seconds of build time.

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-5-caching-dependencies)**5. Caching Dependencies**

You will have notice at this point, installing the dependencies is the main **bottleneck**. This step often takes around 30 seconds, even when the dependencies themselves haven't changed. That's because if your repo change it will invalidate all the subsequent cached layers as soon as you do COPY . /app/tmp To address this issue, we can introduce another multi-stage image to leverage caching effectively.

Let's examine the updated Dockerfile:

# ------------------

# package.json cache

# ------------------

FROM node:18.16.1-bullseye-slim as deps

# We need JQ to clean the package.json

RUN apt update && apt-get install -y bash jq

WORKDIR /tmp

# Copy all the package.json

COPY package.json ./

COPY pkgs/api/package.json ./pkgs/api/package.json

# ------------------

# New tmp image

# ------------------

FROM node:18.16.1-bullseye-slim as tmp

WORKDIR /app/tmp

# Copy and install dependencies separately from the app's code

# To leverage Docker's cache when no dependency has changed

COPY --from=deps /tmp ./

COPY package-lock.json ./

# Install every dependencies

RUN npm install

# Copy source code

COPY . /app/tmp

# Build the backend

RUN true \

&& cd pkgs/api \

&& npm run prod:build

# ------------------

# Final image

# ------------------

FROM node:18.16.1-bullseye-slim as web

# BONUS: Do not use root to run the app

USER node

WORKDIR /app

COPY --from=tmp --chown=node:node /app/tmp /app

EXPOSE 8080

In this enhanced Dockerfile, we introduce a new stage named deps. This stage solely copies the package.json files, enabling us to leverage Docker's caching mechanism effectively. We'll see in a next step why we need to use a third image to achieve that.

The result is a **significant improvement** in build time, particularly when only the source code has been modified. With this approach, we achieved an incredible 30 seconds reduction in build time, 50% improvement from the previous iteration and 71% improvement overall. It's important to note that these steps are now cached, except for the npm run build step, which can never be cached except if there is absolutely no change.

**Results**:

From Scratch

62.0s

from 106.8s

41%

Partial Cache

20s

from 69s

71%

Full Cache

1.5s

Image Size

1.32GB

from 2.48GB

46%

Cached output example

[...]

=> CACHED [tmp 3/13] RUN mkdir -p ./pkgs/api/ 0.0s

=> CACHED [deps 2/6] RUN apk add --no-cache bash jq 0.0s

=> CACHED [deps 3/6] COPY package.json /tmp 0.0s

=> CACHED [deps 4/6] COPY pkgs/api/package.json /tmp/api\_package.json 0.0s

=> CACHED [tmp 6/13] COPY --from=deps /tmp/package.json ./package.json 0.0s

=> CACHED [tmp 7/13] COPY --from=deps /tmp/api\_package.json ./pkgs/api/package.json 0.0s

=> CACHED [tmp 10/13] COPY package-lock.json ./ 0.0s

=> CACHED [tmp 11/13] RUN npm install 0.0s

=> [tmp 12/13] COPY . /app/tmp 0.1s

=> [tmp 13/13] RUN true && cd pkgs/api && npm run prod:build 8.5s

=> CACHED [web 2/3] WORKDIR /app 0.0s

=> [web 3/3] COPY --from=tmp --chown=node:node /app/tmp /app 9.4s

=> exporting to image 5.8s

[...]

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-6-cleaning-dependencies)**6. Cleaning dependencies**

While we have made significant progress in optimizing the build time, achieving a sub-20-second build time may prove challenging due to the compilation itself that often takes 10-15 seconds. At this stage, we must address the issue of the image size, which currently stands at 1.32GB. This size is far from ideal and can impact deployment efficiency, resource consumption, and scalability.

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-61-removing-devdependencies)**6.1 Removing devDependencies**

We can remove the development dependencies that are no longer required once the source code has been built. This step ensures that we only retain the relevant dependencies needed for running the application, while discarding tools like Typescript, Webpack, and other development-specific dependencies.

# Clean dev dependencies

RUN true \

&& npm prune --omit=dev

With a simple one-liner, we have achieved remarkable results in terms of image size reduction. Approximately 53% or around 600MB of dependencies have been eliminated from the final image. One might say that the NodeJS ecosystem is bloated, but I still love it very much.

**Results**:

From Scratch

60.6s

from 106.8s

43%

Partial Cache

18s

from 69s

73%

Full Cache

0.8s

Image Size

0.61GB

from 2.48GB

75%

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-62-pre-filtering)**6.2 Pre-Filtering**

An even better approach is to avoid downloading unnecessary dependencies altogether. This is the most challenging aspect of our optimization process as it goes beyond the scope of Docker itself.

To accomplish this, I have chosen to implement a bash script that removes unnecessary dependencies before the `` command is executed. Alternatively, you can also consider using a different package.json file specifically tailored for production use. Unfortunately, unlike in Ruby, there is currently no built-in way to tag dependencies and selectively install them. Therefore, we must rely on manual intervention to achieve our goal.

Dependencies that can be safely removed are: formatting libraries, testing libraries, release libraries, and any other libraries that are only required during development and not at compile time.

To simplify this process, I have created a small bash script that utilizes JQ to transform the package.json file. In addition to removing devDependencies, I have also removed other fields that are typically irrelevant and can be modified without affecting the installation.

#!/usr/bin/env bash

# clean\_package.json.sh

list=($1)

regex='^jest|stylelint|turbowatch|prettier|eslint|semantic|dotenv|nodemon|renovate'

for file in "${list[@]}"; do

echo "cleaning $file"

jq "walk(if type == \"object\" then with\_entries(select(.key | test(\"$regex\") | not)) else . end) | { name, type, dependencies, devDependencies, packageManager, workspaces }" <"$file" >"${file}\_tmp"

# jq empties the file if we write directly

mv "${file}\_tmp" "${file}"

done

# ------------------

# package.json cache

# ------------------

FROM node:18.16.1-bullseye-slim as deps

RUN apt update && apt-get install -y bash jq

WORKDIR /tmp

# Each package.json needs to be copied manually unfortunately

COPY prod/clean\_package\_json.sh package.json ./

COPY pkgs/api/package.json ./pkgs/api/package.json

RUN ./clean\_package\_json.sh './package.json'

RUN ./clean\_package\_json.sh './\*\*/\*/package.json'

[...]

I have now a very small stage to manipulate the package.json that can be executed in parallel. While these layers will always be invalidated due to the modification of the file system, their impact is minimal as they are short-lived.

We don't save on the final image size since all devDependencies were already ignored, but we saved a few seconds (about 5 seconds or 10% decrease) on the initial install time and we save on intermediate layers size. This step is definitely far less impacting but always good to take.

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-why-we-dont-install-on-the-first-image)**Why we don't install on the first image?**

When we copy the package.json in the deps image the cache will be invalidated very often, wether a dependencies version has changed or we updated a script or incremented the version. Subsequently that means the npm install will often be invalidated, and that's a waste of resource if we actually just modified the script field.

To leverage cache even on unrelated change, we clean the package.json and copy them in the tmp image. Those files will have a constant hash because we removed all fields that are not dependencies related. Docker understand that nothing has changed and can skip npm install

**Good to know:** if you don't use this technique and you increment the version field when you release, it will invalidate the cache every time.

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-7-own-your-stack)**7. Own your stack**

The last step is much more personal because there is no low hanging fruit anymore, yet my image still weights 613MB (uncompressed). To further refine the image, we need to delve into the layers themselves and identify any significant dependencies that may have inadvertently made their way into the image.

Breakdown:

1. My code: 11MB
2. Docker Image + Layer: 240MB
3. Dependencies: 362MB

Yes, more than 300Mb of "production" dependencies. The biggest offenders being:

# du -h -d1 node\_modules/ | sort -h

[...]

1.0M node\_modules/markdown-it

1.1M node\_modules/js-beautify

1.4M node\_modules/@babel

1.5M node\_modules/acorn-node

1.6M node\_modules/ts-node

1.9M node\_modules/@manypkg

2.7M node\_modules/ajv-formats

2.7M node\_modules/env-schema

3.2M node\_modules/@swc

3.2M node\_modules/fast-json-stringify

4.2M node\_modules/fastify

4.4M node\_modules/@types

4.4M node\_modules/react-dom

4.5M node\_modules/sodium-native

5.0M node\_modules/lodash

6.6M node\_modules/tw-to-css

6.7M node\_modules/@fastify

8.7M node\_modules/react-email

12M node\_modules/@specfy

14M node\_modules/prisma

17M pkgs/api/node\_modules/.prisma

19M node\_modules/@octokit

39M node\_modules/@prisma

65M node\_modules/typescript

In there, unfortunately a lot of uncompressable bloated dependencies. One notable example is the @octokit package, which occupies a significant amount of space at 19MB, despite being used for only four (4) REST API calls.

Prisma is surprinsingly big and duplicated for some reason which amounts to a **stagering 50Mb**, to do some Database call.

Furthermore some packages are there because others packages are wrongly installing their dependencies as prod dependencies unfortunately. For example, Next.js always installs Typescript as prod dependency. After realising this and also wanting to have a CDN, I moved all my Frontends to Vercel. Removing app and website from my repo saved another 50MB.

Removing peer dependencies, while it can be risky depending on the number and importance of the packages that rely on them, can be another effective strategy for reducing the image size. In my case, removing peer dependencies resulted in a size reduction of 47MB.

# Clean dev dependencies

RUN true \

&& npm prune --omit=dev --omit=peer

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-final-results)**Final results**

After implementing various optimization techniques, we achieved notable improvements in both build time and image size. Here are the key findings:

From Scratch

60.6s

from 106.8s

43%

Partial Cache

18s

from 69s

73%

Full Cache

0.9s

Image Size

0.51GB

from 2.48GB

79%

**Savings**:

1. Fresh build: saved 40 seconds, representing a 43% decrease in build time.
2. Regular build: time reduction of 51 seconds, which corresponds to an impressive **73% decrease in build time**.
3. The compiled image size was reduced by 1.9GB, resulting in an **80% decrease in size**.

Although these results are significant, there is still a sense of bittersweetness. It becomes apparent that many of the optimization steps undertaken should ideally be automated. Additionally, despite the achieved improvements, the final image size remains relatively large for an API. All in all, it's a long and tedious process that can a long time to master, it's hard to explain to newcomers and quite hard to maintain.

**To remember**:

1. Modifications to the repository that are not ignored by .dockerignore will break the cache, necessitating a complete rebuild.
2. **Deleting files** during the build will **not save size** unless you are using Multi-Staged build.
3. Utilizing a **multi-stage build** approach simplifies the process of eliminating unnecessary layers.
4. **Cleaning up dependencies** plays a crucial role.

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-final-docker-image)**Final docker image**

# ------------------

# package.json cache

# ------------------

FROM node:18.16.1-bullseye-slim as deps

RUN apt update && apt-get install -y bash jq

WORKDIR /tmp

# Copy all the package.json

COPY prod/clean\_package\_json.sh package.json ./

COPY pkgs/api/package.json ./pkgs/api/package.json

# Remove unnecessary dependencies

RUN ./clean\_package\_json.sh './package.json'

RUN ./clean\_package\_json.sh './\*\*/\*/package.json'

# ------------------

# New tmp image

# ------------------

FROM node:18.16.1-bullseye-slim AS tmp

# Setup the app WORKDIR

WORKDIR /app/tmp

# Copy and install dependencies separately from the app's code

# To leverage Docker's cache when no dependency has change

COPY --from=deps /tmp ./

COPY package-lock.json ./

# Install every dependencies

RUN npm install

# At this stage we copy back all sources and overwrite package(s).json

# This is expected since we already installed the dependencies we have cached what was important

# At this point nothing can be cached anymore

COPY . /app/tmp

# /!\ It's counter intuitive but do not set NODE\_ENV=production before building, it will break some modules

# ENV NODE\_ENV=production

# /!\

# Build API

RUN true \

&& cd pkgs/api \

&& npm run prod:build

# Clean src

RUN true \

&& rm -rf pkgs/\*/src

# Clean dev dependencies

RUN true \

&& npm prune --omit=dev --omit=peer

# ------------------

# Final image

# ------------------

FROM node:18.16.1-bullseye-slim as web

ENV PORT=8080

ENV NODE\_ENV=production

# Do not use root to run the app

USER node

WORKDIR /app/specfy

COPY --from=tmp --chown=node:node /app/tmp /app/specfy

EXPOSE 8080

[**§**](https://www.specfy.io/blog/1-efficient-dockerfile-nodejs-in-7-steps#h-bonus)**Bonus**

Just a few points that were not covered:

1. I saved around 2MB after removing the source code, not that impactful but it's cleaner.
2. If you compile a frontend in your dockerfile be sure to clean cache file, usually it's saved for reuse in the node\_modules folder but can be deleted.
3. Compiling with [depot.dev](http://depot.dev), while being very promising it's a paid / remote solution.