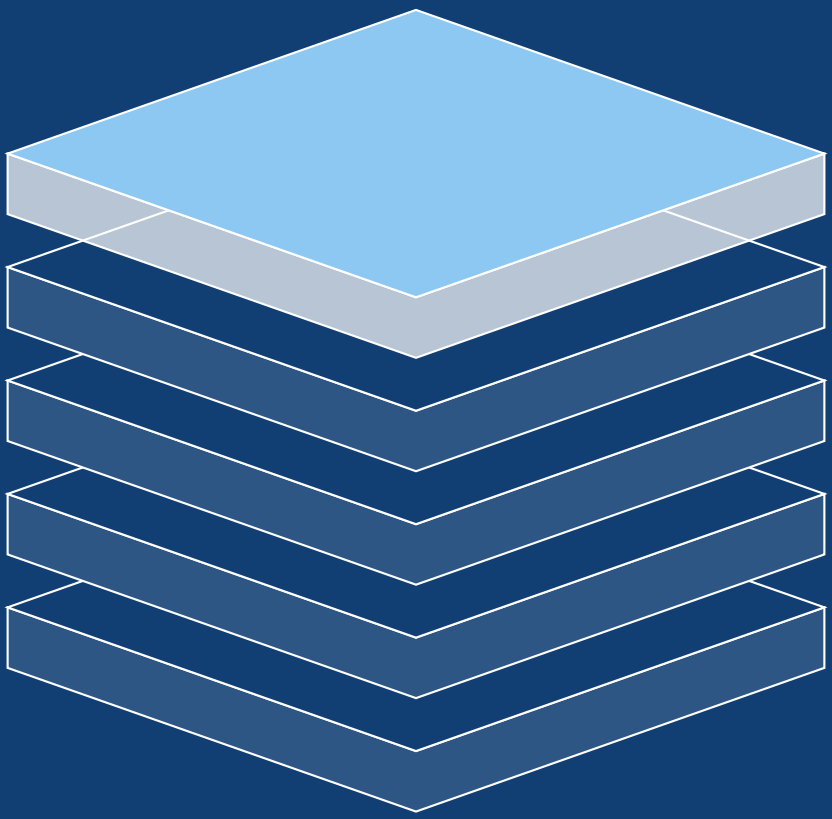
nyrvex.com

Dockerfiles

Hands-On

Build small, fast, and secure
container images



Interview-ready with minimal text

Dockerfiles

Hands-On

Build small, fast, and secure container images

Interview-ready with minimal text

Thomas Zachmann

Dockerfiles Hands-On. Build small, fast, and secure container images.
As of November 2026. © 2026 Thomas Zachmann. License: CC BY-ND 4.0.

Written for Docker Engine 29 and Buildx 0.38 with BuildKit. The Dockerfiles, commands, and output in this book have not been built or run. Image tags, sizes, and output are examples and may vary by version. Check tags and versions before you adopt them.

Typefaces: Charter, Fira Sans, JetBrains Mono.

Contents

How to Read This Book 5

Getting Started **7**
1 What Is a Dockerfile? 8
2 Images and Layers 10
3 Your First Image 13
4 Context and .dockerignore 15

The Instructions **17**
5 FROM 18
6 RUN 19
7 COPY and ADD 22
8 CMD and ENTRYPOINT 24
9 ENV and ARG 27
10 WORKDIR and USER 29
11 EXPOSE, VOLUME, LABEL 31
12 HEALTHCHECK 33
13 SHELL, ONBUILD, Directives 35

Building Better **37**
14 The Build Cache 38
15 BuildKit 41
16 Multi-Stage Builds 43
17 Secrets and SSH 46
18 Multi-Arch with buildx 49
19 Small Images 52

Bases and Languages **55**
20 Choosing Base Images 56
21 Node.js 59
22 Python 61
23 Go 63
24 Java 65
25 Static Sites with nginx 67

Security and Runtime **69**
26 Non-Root 70
27 PID 1 and Signals 72
28 Pinning 75
29 Scanning and SBOM 77
30 Signing and Provenance 80

Day to Day **83**
31 Lint: hadolint and Build Checks 84
32 CI 87
33 When the Build Fails 90
34 When the Container Fails 92
35 Improving Three Dockerfiles 95

The Interview **99**
36 The Questions 100
37 Live Exercises 104
38 Cheat Sheet 106
Index 109

How to Read This Book

You have an interview coming up and you need to know Dockerfiles. Not someday. Soon. This book is built for exactly that: little text, lots of small examples, and for every topic the question you will actually be asked.

The Rules

- **Type everything yourself.** Copy nothing. Your fingers learn too.
- **One command at a time.** There are no scripts. You type, you watch what happens.
- **Every example stands alone.** Create an empty folder, write the Dockerfile, build it.
- **Always ask why.** In the interview, what counts isn't the instruction but its effect on layers, cache, and security.

> **REMEMBER**
>
> The examples in this book have not been built or run. Tags and sizes are examples. Check which version is current.

The Boxes

> **PICTURE THIS**
>
> *An everyday comparison. Hold on to it and you can work out the topic from there.*

```
docker build -t app:1.0 .
```

Memory aid: A command to type, with a memory aid below it.

> **PITFALL**
>
> The mistake almost everyone makes. And the one interviewers love to ask about.

IN THE INTERVIEW

A typical question?

An answer in two or three sentences. It's allowed to be that short.

Your Study Plan

You have ...	Then read ...
1 evening	Chapters 2, 8, 14, 16, and 26, then chapters 36 (The Questions) and 38 (cheat sheet).
2 days	Day 1: Parts I and II. Day 2: Part III and chapters 26 to 29, then chapter 36.
1 week	Day 1: Parts I and II. Then one part per day. Day 6: the live exercises from chapter 37. Day 7: review the questions.

PART I

Getting Started

What a Dockerfile is, what an image is, and how to build your first one.

1

What Is a Dockerfile?

A recipe that produces the same image every time.

A Dockerfile is a text file with instructions. Docker runs them from top to bottom. At the end you have an image. The core idea fits on one line:

Dockerfile + context = image

From the image you start as many containers as you like.

> **PICTURE THIS**
>
> *The Dockerfile is the recipe. The image is the frozen cake, always the same and easy to ship. The container is the slice sitting on your plate right now.*

The Three Terms

Term	What it is	Command
Dockerfile	The build instructions as text.	`docker build`
Image	Immutable package of files and metadata.	`docker push`, `pull`
Container	A running process from an image.	`docker run`

Why Not Just Install It?

- **Reproducible.** The recipe lives in Git, and everyone builds the same thing.
- **Portable.** The image runs the same on your laptop and in the cluster.
- **Reviewable.** Every change is a diff in code review.

The Smallest Useful Example

Dockerfile

```
FROM alpine:3.23
CMD ["echo", "Hello Container"]
```

Two lines: What do I build on, and what should happen at startup?

IN THE INTERVIEW

What is the difference between an image and a container?

An image is an immutable package of layers and metadata. A container is a running instance of it, with a thin writable layer on top. One image can produce any number of containers.

KEY TAKEAWAYS

- Dockerfile + context = image.
- The image is the template, the container is the process.
- The Dockerfile belongs in Git.

2

Images and Layers

An image is a stack. Understand that and you understand almost everything.

An image consists of layers. Each layer stores only the changes from the layer below it. At runtime, Docker stacks the layers and presents them as a single filesystem.

```
Container layer  (writable, runtime only)
-----------------------------------------
COPY . /app      (Layer 3, read-only)
RUN npm ci       (Layer 2, read-only)
FROM node:slim   (Layer 1, read-only)
```

> **PICTURE THIS**
>
> *Transparencies on an overhead projector. Each sheet shows only what's new. What gets crossed out on one sheet is still there on the sheet below.*

What Creates a Layer

Instruction	Effect
`RUN`, `COPY`, `ADD`	Create a layer with files.
`ENV`, `CMD`, `LABEL`, `USER`, `EXPOSE`, `WORKDIR`	Mainly change metadata. No file content.

The Most Important Consequence

Deleting in a later layer does not make the image smaller.

Dockerfile (bad)

```
FROM alpine:3.23
RUN wget -O /tmp/big.tar https://example.com/big.tar
RUN rm /tmp/big.tar
```

The second layer only records: "This file is gone." The whole file is still in the first layer. The delete has to happen in the same `RUN`.

> **PITFALL**
>
> This applies to secrets too. A password that lands in layer 3 and is deleted in layer 4 stays readable in the image. Anyone with pull access can extract it.

Copy-on-Write

When a container changes a file from a lower layer, Docker first copies it up into the container layer. The lower layer stays unchanged. That's why many containers can share one image.

Layers Are Shared

Layers are identified by their hash. Ten images on the same 100 MB base take up those 100 MB only once, and a pull downloads only the missing layers.

What's Inside an Image?

An image consists of three things:

- **Manifest:** the list of layers.
- **Config:** metadata such as `CMD`, `ENV`, and `USER`.
- **Layers:** the files as archives.

For multiple architectures there is also an **index**, see chapter 18. All of it is standardized in the OCI format.

Taking a Look

```
docker history app:1.0
```

Memory aid: The image's history: every instruction with its size.

```
docker image inspect app:1.0
```

Memory aid: The ID card: `Cmd`, `Env`, `User`, layer hashes.

```
dive app:1.0
```

Memory aid: Dive into the layers: this extra tool shows files per layer and wasted space.

IN THE INTERVIEW

Why doesn't an image get smaller when I delete a file in a later `RUN`?

Layers are immutable. The later layer only contains a deletion marker, and the file stays in the earlier layer. The delete has to happen in the same `RUN`, or you use a multi-stage build.

IN THE INTERVIEW

How do layers save storage?

They are content-addressed. Identical layers exist only once on disk and in the registry. On pull and push, only what's missing gets transferred.

KEY TAKEAWAYS

- An image is a stack of immutable layers.
- `RUN`, `COPY`, `ADD` create file layers.
- Deleting in a later layer saves nothing.
- `docker history` and `dive` show where the size is.

3

Your First Image

Build, run, tag, push.

A Small Project

server.py

```python
from http.server import HTTPServer, SimpleHTTPRequestHandler
print("Server running", flush=True)
HTTPServer(("", 8000), SimpleHTTPRequestHandler).serve_forever()
```

Dockerfile

```dockerfile
FROM python:3.14-slim
WORKDIR /app
COPY server.py .
EXPOSE 8000
CMD ["python", "server.py"]
```

Build and Run

```
docker build -t app:1.0 .
```

Memory aid: `-t` as in **t**ag: Give the image a name. The dot at the end is the context, meaning this folder.

```
docker run --rm -p 8000:8000 app:1.0
```

Memory aid: `-p` is host port:container port. `--rm` clears the container away at the end.

```
docker image ls app
```

Memory aid: Shows the image and its size.

An Image's Name

```
registry.example.com/team/app:1.0
|__ Registry ______| |_ Repo _| |Tag
```

Without a registry, Docker Hub applies. Without a tag, `latest` applies.

```
docker tag app:1.0 registry.example.com/team/app:1.0
```

Memory aid: A second name for the same image, like a sticker. Nothing gets copied.

```
docker push registry.example.com/team/app:1.0
```

Memory aid: Push it to the registry. Only missing layers are transferred.

> **PITFALL**
>
> `latest` does not mean "the newest." It's just the default name when you don't give a tag. It can be ancient and can change at any time. Use fixed tags or digests, see chapter 28.

IN THE INTERVIEW

What does the dot in `docker build .` mean?

That's the build context, the folder whose files the builder is allowed to see. The Dockerfile is looked for there by default. With `-f` you name a different file.

KEY TAKEAWAYS

- `docker build -t name:tag .`
- `docker run --rm -p host:container name:tag`
- Tags are stickers, and `latest` is no guarantee.

4

Context and .dockerignore

You decide what the builder gets to see.

The **build context** is the set of files that `COPY` and `ADD` can access. Files outside the context are out of reach.

> **PITFALL**
>
> `COPY ../secret.txt .` does not work. The build cannot leave the context. That's on purpose: A Dockerfile from the internet shouldn't be able to read your disk.

Why `.dockerignore` Matters

Without it, `COPY . .` puts everything in the image: `.git`, `node_modules`, `.env`, test data.

- **Security:** No secrets in the image.
- **Speed:** Fewer files have to reach the builder.
- **Cache:** A changed `README.md` doesn't invalidate `COPY . .`.

`.dockerignore`

```
.git
.env
.env.*
node_modules
dist
*.log
Dockerfile
.dockerignore
# Exception: include this file after all
!.env.example
```

Pattern	Meaning
`*.log`	All `.log` files at the top level.
`**/*.log`	All `.log` files at any depth.
`!file`	Exception. The last matching line wins.

PICTURE THIS

The context is the suitcase you carry to the repairman. `.dockerignore` is the list of what stays home. Bring the whole house and you'll wait a long time and show too much.

REMEMBER

The file must be in the root directory of the context. For a specific Dockerfile there is also `Dockerfile.dockerignore` next to the Dockerfile.

Other Dockerfiles

```
docker build -f docker/Dockerfile.prod -t app:1.0 .
```

Memory aid: `-f` as in **f**ile: The recipe is elsewhere, the context is still the dot.

IN THE INTERVIEW

Why do you need `.dockerignore`?

It limits the context. That keeps secrets like `.env` out of the image, makes the build faster, and stops irrelevant files from invalidating the `COPY` cache.

KEY TAKEAWAYS

- `COPY` sees only the context.
- `.git`, `.env`, and `node_modules` belong in `.dockerignore`.
- `.dockerignore` protects security, speed, and cache.

PART II

The Instructions

Every Dockerfile instruction, its quirks, and its interview questions.

5

FROM

Every image stands on the shoulders of another.

`FROM` sets the base. Everything else builds on it.

```
FROM python:3.14-slim
FROM python:3.14-slim AS build
FROM --platform=linux/amd64 python:3.14-slim
FROM scratch
```

- `AS build` names the stage for multi-stage builds, see chapter 16.
- `--platform` selects the architecture, see chapter 18.
- `scratch` is not an image but pure emptiness. Good for static binaries.

Variables in `FROM`

Only `ARG` may come before `FROM`:

```
ARG PYTHON_VERSION=3.14
FROM python:${PYTHON_VERSION}-slim
```

> **PITFALL**
>
> An `ARG` before `FROM` applies only to `FROM` lines. If you want to use the value later in the stage, write `ARG PYTHON_VERSION` again there, without a value.

IN THE INTERVIEW

What is `FROM scratch`?

An empty start with no shell, libraries, or package manager. You copy in only your static binary. That gives you very small images with a minimal attack surface, but also nothing to debug with.

KEY TAKEAWAYS

- `FROM` is the base, `AS` names the stage.
- `scratch` is empty.
- An `ARG` before `FROM` must be redeclared in the stage.

6

RUN

Commands at build time. Every `RUN` is a layer.

`RUN` executes a command in the build container and saves the result as a layer.

Two Forms

```
RUN apt-get update              # shell form
RUN ["python", "--version"]     # exec form
```

The shell form runs through `/bin/sh -c`. It understands `&&`, pipes, and variables.

The exec form starts the program directly, without a shell.

The Pattern for Packages

Dockerfile

```
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
       curl \
       ca-certificates \
    && rm -rf /var/lib/apt/lists/*
```

Why like this?

- **Everything in one `RUN`:** Otherwise the package lists stay behind in a layer of their own.
- **`update` and `install` together:** Otherwise the cache uses stale package lists.
- **`--no-install-recommends`:** No recommended extras.
- **Clean up in the same `RUN`:** Otherwise it saves nothing.
- **One line per package:** Diffs stay readable.

On Alpine it's shorter:

```
RUN apk add --no-cache curl ca-certificates
```

> **PITFALL**
>
> `RUN apt-get update` on a line of its own is a classic. The layer comes from the cache, and weeks later `install` tries to fetch packages from outdated lists. That fails or installs old versions.

Heredocs

With BuildKit, longer scripts stay readable:

```
RUN <<EOF
set -eux
apt-get update
apt-get install -y --no-install-recommends curl
rm -rf /var/lib/apt/lists/*
EOF
```

The whole thing remains a single layer.

> **PITFALL**
>
> `cd` lasts for only one `RUN`. Every instruction starts fresh. `RUN cd /app` followed by `RUN make` does not run in `/app`. Use `WORKDIR`.

> **PITFALL**
>
> With `RUN curl ... | sh`, only the exit code of the last command counts. If `curl` fails, the build keeps going anyway. The fix is `pipefail`, see chapter 13.

> **IN THE INTERVIEW**
>
> **Why put `apt-get update` and `install` in the same `RUN`?**
>
> Because of the cache. If `update` stands alone, its layer comes from the cache while the `install` line changes. Then `install` works with old package lists.

> **IN THE INTERVIEW**
>
> **Why not lots of small `RUN` lines?**
>
> Each one creates a layer, and cleaning up in the next layer saves nothing. Related steps go in one `RUN`. What changes at different rates stays separate.

KEY TAKEAWAYS

- Shell form with `/bin/sh -c`, exec form without a shell.
- Install and clean up in the same `RUN`.
- Heredocs for longer scripts.
- `cd` only works within one `RUN`.

7

COPY and ADD

Getting files into the image. Nearly always with `COPY`.

```
COPY package.json package-lock.json ./
COPY src/ /app/src/
COPY --chown=app:app config.yaml /etc/app/
COPY --chmod=755 start.sh /usr/local/bin/
COPY --from=build /app/dist /usr/share/nginx/html
```

The Rules

- A destination ending in `/` is a folder.
- With a folder as the source, its **contents** are copied, not the folder itself.
- Multiple sources need a folder as the destination.
- Relative destinations refer to `WORKDIR`.

> **PITFALL**
>
> `COPY src/ /app/` does not create `/app/src`. The contents of `src` land directly in `/app`. To keep the structure, write `/app/src/`.

`ADD` Can Do More

Capability	`COPY`	`ADD`
Local files	Yes	Yes
From another stage (`--from`)	Yes	No
Fetch an HTTP URL	No	Yes
Fetch a Git repository	No	Yes
Unpack a local tar	No	Yes, automatically

> **PITFALL**
>
> That `ADD` unpacks local tar archives surprises many people. An archive downloaded from a URL, on the other hand, it does not unpack. Use `COPY` if you need none of this: It does exactly what it says on the tin.

If you use `ADD` to fetch from the network, verify the checksum:

```
ADD --checksum=sha256:24454f83... https://example.com/tool.tar.
gz /tmp/
```

Modern Flags

Flag	Effect
`--link`	The layer is built independently of the layers below it. Better cache.
`--parents`	Preserves the source's folder structure.
`--exclude=pattern`	Skips files while copying.

The `--link` flag helps especially when the base changes: The layer doesn't have to be recreated.

IN THE INTERVIEW

`COPY` or `ADD`?

`COPY`, as long as it's only about files from the context, because it's predictable. `ADD` only if you deliberately want to unpack a local tar or fetch a URL with a checksum.

KEY TAKEAWAYS

- `COPY` is the default.
- `ADD` unpacks local tars and fetches URLs.
- `COPY --from` pulls files from other stages.
- `--link` makes layers more independent.

8

CMD and ENTRYPOINT

The favorite question of every interview.

Both determine what runs at container start.

- `ENTRYPOINT`: **What** runs. The program.
- `CMD`: **With what** it runs. The default arguments. Without `ENTRYPOINT`, it's the whole command.

```
ENTRYPOINT ["curl"]
CMD ["--help"]
```

Invocation	Runs in the container
`docker run image`	`curl --help`
`docker run image https://example.com`	`curl https://example.com`
`docker run --entrypoint sh image`	`sh`

> **PICTURE THIS**
>
> *`ENTRYPOINT` is the coffee machine, `CMD` the preset "Espresso" button. The guest may press a different button. For a different machine, they need a screwdriver: `--entrypoint`.*

Exec Form and Shell Form

```
CMD ["node", "server.js"]   # exec form: good
CMD node server.js          # shell form: careful
```

The shell form starts `/bin/sh -c "node server.js"`. The shell is then PID 1, and your application receives no `SIGTERM`. More on that in chapter 27.

How the Two Work Together

ENTRYPOINT	CMD	Result
none	`["a", "b"]`	`a b`
`["x"]`	none	`x`
`["x"]`	`["a"]`	`x a`
`x` (shell form)	any	`/bin/sh -c x`. `CMD` is ignored.

> **PITFALL**
>
> `ENTRYPOINT` in shell form swallows everything. Neither `CMD` nor arguments from `docker run` reach your program. Always use the exec form with square brackets and double quotes. Single quotes are not valid JSON, and Docker then falls back to the shell form.

The Entrypoint Script

You often need to prepare something before startup:

entrypoint.sh

```
#!/bin/sh
set -e
echo "Preparing configuration"
exec "$@"
```

```
COPY entrypoint.sh /usr/local/bin/
ENTRYPOINT ["entrypoint.sh"]
CMD ["nginx", "-g", "daemon off;"]
```

The `exec "$@"` is crucial. It replaces the shell with your program, which receives `CMD` as arguments. That makes it PID 1, so it receives signals.

> **IN THE INTERVIEW**
>
> **What is the difference between `CMD` and `ENTRYPOINT`?**
>
> `ENTRYPOINT` sets the program, `CMD` the default arguments. Arguments to `docker run` replace `CMD`. You replace `ENTRYPOINT` only with `--entrypoint`. I write both in exec form.

IN THE INTERVIEW

Why `exec "$@"` in the entrypoint script?

Without `exec`, the shell stays PID 1 and starts the program as a child. Signals like `SIGTERM` then don't get through. `exec` replaces the shell with the program.

KEY TAKEAWAYS

- `ENTRYPOINT`: program. `CMD`: default arguments.
- Always exec form with `["..."]`.
- Entrypoint scripts end with `exec "$@"`.

9

ENV and ARG

One variable for runtime, one for build time.

	`ARG`	`ENV`
Applies during build	Yes, from its declaration in the stage	Yes, from the instruction on
Applies in the container	No	Yes
How to set	`--build-arg` on `docker build`	`-e` on `docker run`

```
ARG VERSION=1.0.0
ENV APP_ENV=production APP_DIR=/app
RUN echo "Building $VERSION for $APP_ENV"
```

```
docker build --build-arg VERSION=2.0.0 -t app:2 .
```

Memory aid: The site owner names the version at build time.

```
docker run -e APP_ENV=staging app:2
```

Memory aid: `-e` as in **e**nvironment: overrides `ENV` at startup.

> **PICTURE THIS**
>
> *`ARG` is the builder's scrap of paper: Once construction is done, it's gone. `ENV` is the house rules: They hang in the hallway forever.*

A Common Pattern

Set a value at build time and keep it at runtime:

```
ARG APP_VERSION=dev
ENV APP_VERSION=$APP_VERSION
```

PITFALL

Secrets belong in neither. `ENV` values sit in the metadata, `ARG` values in the build history. `docker history` and `docker inspect` show them. For secrets, there's chapter 17.

REMEMBER

When a `--build-arg` changes, the cache is invalidated from the first place that uses it. That's why the example `ARG BUILD_DATE` goes as late as possible in the Dockerfile, or you'd rebuild everything every time.

A Real Mistake

You see this pattern far too often:

```
ENV POSTGRES_PASSWORD=secret
```

The password is in the image forever. The right way is to pass it in at startup from a secret.

IN THE INTERVIEW

What is the difference between `ENV` and `ARG`?

`ARG` exists only during the build, `ENV` also in the running container. Neither is suitable for secrets, because they end up in the image's history or metadata.

KEY TAKEAWAYS

- `ARG`: build time. `ENV`: build time and runtime.
- Neither is a place for secrets.
- Changed `ARG`s invalidate the cache from the point of use.

10

WORKDIR and USER

Where does it run, and as whom?

WORKDIR

```
WORKDIR /app
COPY . .
RUN make
```

`WORKDIR` sets the current directory for everything that follows: `RUN`, `COPY`, `CMD`. If the folder is missing, Docker creates it. Relative paths build on the previous `WORKDIR`.

> **REMEMBER**
>
> Use absolute paths. A relative `WORKDIR` depends on the base image and can change with its next version.

USER

Without `USER`, everything runs as `root`.

```
RUN useradd --system --uid 10001 --no-create-home app
COPY --chown=app:app . /app
USER 10001
CMD ["./server"]
```

`USER` affects the `RUN`, `CMD`, and `ENTRYPOINT` that come after it.

> **PITFALL**
>
> `COPY` doesn't care about `USER`. Copied files belong to `root` by default. If the application needs to write there, use `--chown` or change the permissions beforehand.

> **PITFALL**
>
> `USER app` with a name instead of a number fails on `runAsNonRoot` in Kubernetes, because the kubelet cannot verify the name. The number `USER 10001` works.

IN THE INTERVIEW

Why shouldn't a container run as `root`?

If someone breaks out of the application, they are `root` in the container. That makes escaping to the host easier. With a dedicated user without privileges, the damage shrinks. Details in chapter 26.

KEY TAKEAWAYS

- `WORKDIR` instead of `cd`, always absolute.
- Set `USER` with a number.
- `COPY` ignores `USER`, hence `--chown`.

11

EXPOSE, VOLUME, LABEL

Three instructions that do less than many people think.

EXPOSE

```
EXPOSE 8080
EXPOSE 8125/udp
```

`EXPOSE` is documentation. It doesn't open a port. You have to publish it at startup with `-p 8080:8080`. `-P` publishes all `EXPOSE` ports on random host ports.

VOLUME

```
VOLUME /var/lib/data
```

`VOLUME` marks a path as a place for persistent data. At startup, Docker creates an anonymous volume there unless you mount something else.

> **PITFALL**
>
> Anything a later `RUN` writes to a `VOLUME` path is lost. Prepare the contents beforehand. Many people skip `VOLUME` entirely and mount explicitly at startup, so no forgotten anonymous volumes pile up.

LABEL

Labels are metadata as key and value. There are standard names:

```
LABEL org.opencontainers.image.title="app" \
      org.opencontainers.image.version="1.0.0" \
      org.opencontainers.image.source="https://github.com/acme/
app"
```

> **REMEMBER**
>
> `MAINTAINER` is deprecated. Use `LABEL org.opencontainers.image.authors`.

STOPSIGNAL

```
STOPSIGNAL SIGQUIT
```

Sets the signal Docker sends on stop. The default is `SIGTERM`. The example suits nginx, which shuts down gracefully on `SIGQUIT`.

IN THE INTERVIEW

Does `EXPOSE` open a port?

No. It only documents which port the application listens on. You make the port reachable with `-p` at startup.

KEY TAKEAWAYS

- `EXPOSE` documents, `-p` publishes.
- Data written after `VOLUME` is lost.
- `LABEL` with `org.opencontainers.image.*`.

12

HEALTHCHECK

Is the process running, or does it actually work?

```
HEALTHCHECK --interval=30s --timeout=3s --start-period=20s --
retries=3 \
  CMD curl -f http://localhost:8080/health || exit 1
```

Docker runs the command regularly inside the container. Exit code 0 means `healthy`, 1 means `unhealthy`.

Option	Default	Meaning
`--interval`	30 s	Time between checks.
`--timeout`	30 s	Anything longer counts as failed.
`--start-period`	0 s	Grace period at startup. Failures don't count.
`--start-interval`	5 s	Time between checks during the grace period.
`--retries`	3	This many failures in a row until `unhealthy`.

```
docker inspect --format '{{json .State.Health}}' web
```

Memory aid: The container's medical report: status and the latest checks.

`HEALTHCHECK NONE` turns off an inherited check.

> **PICTURE THIS**
>
> *A running process is like a light on in the office. It's on, but that says nothing about whether anyone is working. The healthcheck phones in and expects an answer.*

> **PITFALL**
>
> `curl` isn't in every image, and certainly not in distroless. There you need a small check program or a subcommand of your application, such as `CMD ["/app", "healthcheck"]`.

PITFALL

Kubernetes ignores `HEALTHCHECK`. There, liveness, readiness, and startup probes apply. The instruction helps with Docker and Compose.

IN THE INTERVIEW

What's the point of `HEALTHCHECK` if the process is running anyway?

A process can be running and still be hung. The healthcheck tests actual function. Docker reports the state, and Compose can wait for it with `depends_on`.

KEY TAKEAWAYS

- 0 is `healthy`, 1 is `unhealthy`.
- `--start-period` gives it time to start.
- Kubernetes uses probes, not `HEALTHCHECK`.

13

SHELL, ONBUILD, Directives

The rarely used tools that interviewers love to ask about.

SHELL

`SHELL` changes the shell for the shell form of `RUN`, `CMD`, and `ENTRYPOINT`. The most important use case is `pipefail`:

```
SHELL ["/bin/bash", "-o", "pipefail", "-c"]
RUN curl -fsSL https://example.com/setup.sh | bash
```

Now the build aborts if `curl` fails. Without `pipefail`, only the last link of the pipe counts.

> **REMEMBER**
>
> Whether the default `sh` supports `pipefail` depends on the image. That's why `bash` is named explicitly here, and it must be installed in the image.

ONBUILD

`ONBUILD` stores an instruction for later. It runs only when someone builds on this image with `FROM`.

```
FROM node:24-slim
WORKDIR /app
ONBUILD COPY package*.json ./
ONBUILD RUN npm ci
ONBUILD COPY . .
```

> **PICTURE THIS**
>
> *A note in the assembly kit: "Whoever puts this together must do step A first." The manufacturer doesn't do it.*

This is meant for in-house base images. It does hide steps, though, so many people avoid it.

Parser Directives

Directives sit at the very top of the Dockerfile, as special comments.

```
# syntax=docker/dockerfile:1
# check=error=true
FROM alpine:3.23
```

Directive	Effect
`syntax`	Selects the Dockerfile version. New features without a Docker update.
`escape`	Changes the escape character, useful on Windows.
`check`	Controls build checks, see chapter 31.

IN THE INTERVIEW

What does `# syntax=docker/dockerfile:1` do?

It sets the frontend BuildKit loads to understand the Dockerfile. That gets you new features such as current mount options without updating Docker.

KEY TAKEAWAYS

- `SHELL` for `pipefail`.
- `ONBUILD` runs only in the child image.
- Directives go at the very top.

PART III

Building Better

Cache, BuildKit, multi-stage, secrets, multi-arch, and small images.

14

The Build Cache

The difference between ten seconds and ten minutes.

Docker checks every instruction from top to bottom: Is there already a layer for this? If so, it uses it. As soon as one instruction is rebuilt, all the following ones usually are too.

A cache miss at the top rebuilds everything below

> **PICTURE THIS**
>
> *Dominoes. Tip one and everything behind it falls. That's why whatever changes often belongs at the very end.*

When Is the Cache Valid?

Instruction	What gets checked
`RUN`	Only the instruction's text. Not the result!
`COPY`, `ADD`	A checksum of the file contents. The timestamp doesn't count.
`ENV`, `ARG`	The value.

> **PITFALL**
>
> `RUN apt-get install curl` and `RUN git clone ...` look the same to the cache no matter what has changed out on the network. Docker doesn't refetch anything just because something new exists.

The Most Important Rule: Order

Dockerfile (bad)

```
FROM node:24-slim
WORKDIR /app
COPY . .
RUN npm ci
```

Every code change invalidates `COPY . .`, so `npm ci` runs again too.

Dockerfile (good)

```
FROM node:24-slim
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
```

Now `npm ci` runs only when the dependencies change.

> **REMEMBER**
>
> The rule is: from rarely changed to often changed. First the system, then the dependencies, then your code.

Bypassing the Cache

```
docker build --no-cache -t app:1.0 .
```

Memory aid: Everything fresh, no layer from the cache.

```
docker build --pull -t app:1.0 .
```

Memory aid: Check whether there's a newer version of the base.

Cache Mounts

A cache mount is a folder that persists between builds without ending up in the image. Ideal for package managers:

```
RUN --mount=type=cache,target=/root/.npm npm ci
```

If `package-lock.json` changes, `npm ci` runs again, but the downloads are still there.

> **PITFALL**
>
> Cache mounts live locally on the builder. `--cache-to` doesn't export them. On a fresh CI runner they're empty. They are an acceleration, not a dependency: The build must also work without them.

Cache in CI

A fresh runner knows no old layers. You load the cache from a backend:

```
docker buildx build --push -t registry/app:1.0 \
  --cache-from type=registry,ref=registry/app:cache \
  --cache-to type=registry,ref=registry/app:cache,mode=max .
```

`mode=max` also stores the layers of intermediate stages. Other backends are `gha`, `local`, and `s3`.

IN THE INTERVIEW

How do you optimize a Dockerfile for the cache?

Rarely changed things at the top, often changed things at the bottom. First copy the lockfiles and install dependencies, then the code. Add a good `.dockerignore`, and in CI an exported cache.

IN THE INTERVIEW

How does Docker detect whether a `COPY` has changed?

Through a checksum of the file contents and permissions. The modification time plays no part. For `RUN`, by contrast, it compares only the instruction's text.

KEY TAKEAWAYS

- A cache miss invalidates everything below it.
- `RUN` is recognized by its text, `COPY` by its content.
- Lockfile first, code last.
- Cache mounts for package managers, `--cache-to` for CI.

15

BuildKit

The engine behind `docker build`.

BuildKit is Docker's build engine. It has been the default since Docker Engine 23.0. **Buildx** is its CLI part. So today, `docker build` already runs on BuildKit.

What BuildKit Does Differently

- **Graph instead of list:** It plans the whole build as a dependency graph.
- **Parallel:** Independent stages run at the same time.
- **Frugal:** Stages nobody needs are skipped.
- **Targeted contexts:** It transfers only files that are actually needed.
- **New features:** Secrets, SSH, cache mounts, heredocs, multi-arch.

> **PICTURE THIS**
>
> *The old builder works like someone doggedly cooking a recipe line by line. BuildKit reads the whole recipe first and cooks on three burners at once.*

Reading the Output

```
docker build --progress=plain -t app:1.0 .
```

Memory aid: Full output of every line instead of the collapsed progress view. Good for debugging.

```
docker buildx ls
```

Memory aid: Which builders exist, and what can they do?

```
docker buildx du
```

Memory aid: **du** as on Linux: How much space does the build cache take up?

```
docker builder prune
```

Memory aid: Clean up the cache. With `-a`, everything unused as well.

The Drivers

Driver	What for
`docker`	Default, in the Docker daemon.
`docker-container`	Dedicated builder container. For multi-arch and more cache backends.
`kubernetes`	Builders as pods in the cluster.
`remote`	Connection to an existing BuildKit daemon.

More Control in `RUN`

```
RUN --network=none make test
```

This step has no network. That way you notice if a test is quietly downloading something.

Multiple Contexts

```
docker build --build-context libs=../shared-libs -t app:1.0 .
```

```
COPY --from=libs . /opt/libs
```

This gets you files outside the main context, but explicitly and visibly.

IN THE INTERVIEW

What does BuildKit offer over the old builder?

Parallel builds via a dependency graph, skipping unneeded stages, better caching, and new features such as secrets, SSH mounts, and cache mounts.

KEY TAKEAWAYS

- BuildKit is the default, Buildx the command.
- `--progress=plain` shows everything.
- `docker-container` driver for multi-arch.
- `docker buildx du` and `builder prune` against full disks.

16

Multi-Stage Builds

The most important technique for small images.

Build tools don't belong in the finished image. A multi-stage build uses multiple `FROM`s. Only the last stage becomes the image. You copy over the rest selectively.

Dockerfile

```
# Stage 1: build
FROM golang:1.26 AS build
WORKDIR /src
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 go build -o /out/app .

# Stage 2: ship
FROM gcr.io/distroless/static-debian12:nonroot
COPY --from=build /out/app /app
ENTRYPOINT ["/app"]
```

The Go image is big. The result contains only the base and your binary.

> **PICTURE THIS**
>
> *The workshop and the store. The workshop is full of saws and shavings. Into the store you carry only the finished chair.*

Useful Tricks

```
docker build --target build -t app:dev .
```

Memory aid: `--target`: Build only up to this stage. The target is handy for tests or development.

```
FROM node:24-slim AS deps
WORKDIR /app
COPY package*.json ./
```

```
RUN npm ci

FROM deps AS test
COPY . .
RUN npm test

FROM deps AS build
COPY . .
RUN npm run build
```

One stage can build on another. Because BuildKit skips unneeded stages, `test` runs only when you choose it as the target.

> **PITFALL**
>
> BuildKit doesn't build a stage the target stage doesn't depend on. If you think the tests ran because they're in the Dockerfile, you're wrong. Build them explicitly in CI with `--target test`.

> **REMEMBER**
>
> `COPY --from` also works from another image: `COPY --from=nginx:alpine /etc/nginx/nginx.conf .`

What You Get

- **Smaller:** No compiler, no source code, no dev dependencies.
- **More secure:** Fewer packages, fewer vulnerabilities.
- **Clean:** Intermediate data from the build stages doesn't end up in the final result.

> **PITFALL**
>
> Even the build stage isn't automatically safe for secrets. A token in a layer of the build stage stays in the build cache. Use secret mounts, see chapter 17.

> **IN THE INTERVIEW**
>
> **What is a multi-stage build, and what is it for?**
>
> Multiple `FROM`s in one Dockerfile. Early stages build, and the last one contains only the result, which you fetch via `COPY --from`. That keeps compilers and source code out, and the image small and secure.

IN THE INTERVIEW

What does `--target` do?

It ends the build at a specific stage. That lets you build a test, development, or production variant from one Dockerfile.

KEY TAKEAWAYS

- Only the last stage becomes the image.
- `COPY --from=stage` fetches the result.
- `--target` picks a stage.
- Unused stages don't run.

17

Secrets and SSH

Secrets for the build that don't end up in the image.

A build sometimes needs a token, for a private package, say. That token must not show up in a layer or in the history.

What Doesn't Work

Attempt	Problem
`ENV TOKEN=...`	It's in the metadata.
`ARG TOKEN`	It's in the history.
`COPY .npmrc` followed by `rm`	It's still in the layer.

Secret Mounts

Dockerfile

```
# syntax=docker/dockerfile:1
FROM node:24-slim
WORKDIR /app
COPY package*.json ./
RUN --mount=type=secret,id=npmrc,target=/
root/.npmrc,required=true npm ci
```

```
docker build --secret id=npmrc,src=$HOME/.npmrc -t app:1.0 .
```

Memory aid: You give the builder the secret for this one `RUN` only.

During this `RUN`, the secret exists as a file, by default under `/run/secrets/` if you don't name a target. Afterward it's gone, and it doesn't end up in the layer.

> **PICTURE THIS**
>
> *The repairman gets the key for this one room only and hands it back afterward. It's nowhere to be found in the finished house.*

It works as an environment variable too:

```
RUN --mount=type=secret,id=API_KEY,env=API_KEY \
    ./script-with-token.sh
```

```
docker build --secret id=API_KEY -t app:1.0 .
```

Memory aid: Without `src`, Docker reads the value from the environment variable of the same name in your shell.

> **PITFALL**
>
> A step's result can still contain the secret. If a tool writes the token to a file or a log, it's saved. Check what the command leaves behind.

SSH Mounts

For private Git repositories you forward your SSH agent instead of copying a key.

Dockerfile

```
# syntax=docker/dockerfile:1
FROM alpine:3.23
RUN apk add --no-cache git openssh-client
RUN mkdir -p ~/.ssh && ssh-keyscan github.com >> ~/.ssh/
known_hosts
RUN --mount=type=ssh git clone git@github.com:acme/
internal.git /src
```

```
docker build --ssh default -t app:1.0 .
```

Memory aid: `default` is your running SSH agent. The key itself never leaves it.

> **REMEMBER**
>
> Build secrets are the wrong place for operations. Runtime secrets arrive at startup from Vault, Kubernetes Secrets, or Docker Secrets.

> **IN THE INTERVIEW**
>
> **How do you use a token in the build without it ending up in the image?**
>
> With a BuildKit secret mount: `--secret` at build time, `RUN --mount=type=secret` in the Dockerfile. The secret is available only during that step and ends up in neither layer nor history.

IN THE INTERVIEW

Is a multi-stage build enough to protect secrets?

Only to a degree. The final image is clean, but the build layers may sit in the cache or in the cache's registry. You're safe only if the secret never gets into a layer.

KEY TAKEAWAYS

- Never via `ENV`, `ARG`, or `COPY`.
- `--mount=type=secret` for tokens.
- `--mount=type=ssh` for private repositories.
- Runtime secrets arrive at startup.

18

Multi-Arch with buildx

One name, many architectures.

Macs and many cloud servers run on `arm64`, classic servers on `amd64`. A **multi-arch image** is an **image index**: a list that points to a separate image per platform.

```
app:1.0  (Index)
  |-- linux/amd64 --> sha256:aaa...
  |-- linux/arm64 --> sha256:bbb...
```

The client automatically picks the matching variant on pull.

Creating a Builder

```
docker buildx create --name multi --driver docker-container
--use
```

Memory aid: A dedicated builder container that can handle multiple platforms.

```
docker buildx build --platform linux/amd64,linux/arm64 -t
registry/app:1.0 --push .
```

Memory aid: Build both variants and push them to the registry together with the index.

```
docker buildx imagetools inspect registry/app:1.0
```

Memory aid: Look inside the index: Which platforms are in it?

Three Routes to a Foreign Architecture

Route	How	Speed
QEMU	Emulates the foreign CPU.	Slow
Native	One builder per architecture.	Fast, but expensive
Cross-compile	The compiler produces foreign code.	Fast

You set up QEMU with this command:

```
docker run --privileged --rm tonistiigi/binfmt --install all
```

Memory aid: Teaches the kernel to start foreign binaries through an emulator. Needs `--privileged`, so only on a trusted machine.

Cross-Compile in the Dockerfile

The compiler should run natively, but the result should fit the target:

Dockerfile

```
FROM --platform=$BUILDPLATFORM golang:1.26 AS build
ARG TARGETOS TARGETARCH
WORKDIR /src
COPY . .
RUN CGO_ENABLED=0 GOOS=$TARGETOS GOARCH=$TARGETARCH go build -
o /out/app .

FROM scratch
COPY --from=build /out/app /app
ENTRYPOINT ["/app"]
```

Variable	Meaning
`BUILDPLATFORM`	Platform the build runs on.
`TARGETPLATFORM`	Platform the build is for.
`TARGETOS`, `TARGETARCH`	The individual parts of the target.

PICTURE THIS

An interpreter (QEMU) translates every sentence live: It works, but it's slow going. A cross-compile is a book you write directly in the target language.

PITFALL

An `arm64` image on `amd64` without an emulator ends in `exec format error`. That is almost always a wrong architecture or a script without a shebang.

REMEMBER

You describe multiple targets and variants with `docker buildx bake` in a file called `docker-bake.hcl`. That replaces long command lines in CI.

IN THE INTERVIEW

How do you build an image for `amd64` and `arm64`?

With `docker buildx build --platform linux/amd64,linux/arm64 --push`. That creates an image index. For speed I cross-compile with `--platform=$BUILDPLATFORM` instead of using QEMU.

IN THE INTERVIEW

What is an image index?

A list of per-platform images under one tag. The client picks the matching manifest on pull.

KEY TAKEAWAYS

- The index points to one image per platform.
- `buildx build --platform ... --push`.
- QEMU is slow, cross-compiling is fast.
- `exec format error`: check the architecture.

19

Small Images

Less size means less pull time and less attack surface.

Why Size Matters

- **Faster:** Pull, push, and scaling in the cluster.
- **More secure:** Every package removed is one less potential vulnerability.
- **Cheaper:** Less storage and transfer.

The Levers, Sorted by Impact

Lever	How
Multi-stage	Compilers and source code stay out.
Small base	`slim`, distroless, or `scratch`.
Runtime dependencies only	`npm ci --omit=dev`, no test packages.
Clean `.dockerignore`	No accidental files.
Clean up in the same `RUN`	Don't leave package caches in layers.
No recommendations	`--no-install-recommends`, `apk --no-cache`.

The Process

```
docker history --no-trunc app:1.0
```

Memory aid: Which step is to blame?

```
dive app:1.0
```

Memory aid: Which files are they, and what's duplicated?

```
docker system df
```

Memory aid: **df** as on Linux: How much space do images, containers, and cache take up?

PITFALL

The goal is not the smallest image at any cost. An Alpine image that behaves strangely in production because of musl is worse than a slightly larger `slim` image. Correctness and security first, then size.

REMEMBER

The registry stores layers compressed. `docker image ls` shows the unpacked value. Numbers from the registry are therefore smaller.

IN THE INTERVIEW

How do you make an image smaller?

I measure with `docker history` and `dive`. Then multi-stage, a small base, runtime dependencies only, and cleaning up in the same `RUN`.

KEY TAKEAWAYS

- Multi-stage is the biggest lever.
- Measure first, then cut.
- Smaller also means more secure, but correctness comes first.

PART IV

Bases and Languages

The right foundation and ready-made patterns for Node, Python, Go, and Java.

20

Choosing Base Images

A decision that shapes everything else.

Base	C library	Shell	Good for
`debian`, `ubuntu`	glibc	Yes	Building, tools, debugging.
`*-slim`	glibc	Yes	A good default for runtime.
`alpine`	musl	Yes	Very small, if everything runs on musl.
distroless	glibc or none	No	Production with a small attack surface.
`scratch`	none	No	Static binaries.

> **PICTURE THIS**
>
> *`debian` is the furnished apartment, `slim` the apartment with a bed and a table, `alpine` the tiny studio with its own odd measurements. Distroless is the hotel room with no toolbox. `scratch` is the empty room.*

Alpine and musl

Alpine uses **musl** instead of **glibc** as its C library and BusyBox instead of GNU tools. That keeps it small but brings differences:

- Binaries built for glibc won't run. The error message is often a misleading `not found`.
- Python packages don't always have prebuilt `musllinux` wheels. Then a long compile from source follows.
- Memory management and DNS behave differently than with glibc.

> **PITFALL**
>
> "Alpine is always better because it's smaller" is wrong. For Python and Java, `slim` is often the better choice. For Go with `CGO_ENABLED=0` it makes no difference, because the binary is static.

Distroless

Distroless images contain only the runtime and its dependencies: no shell, no package manager. That shrinks the attack surface considerably.

```
FROM gcr.io/distroless/static-debian12:nonroot
COPY app /app
ENTRYPOINT ["/app"]
```

The tags give things away: `:nonroot` runs without root privileges, and `:debug` includes a BusyBox shell for troubleshooting.

> **PITFALL**
>
> Without a shell, the shell form, `$VAR` in `CMD`, and entrypoint scripts no longer work. There's no `docker exec ... sh` either. For troubleshooting you use the `:debug` tag, `docker debug`, or in Kubernetes `kubectl debug`.

A Decision Guide

1. Static binary? `scratch` or distroless `static`.
2. Node, Python, Java? `slim` or distroless, if it suits the application.
3. Tiny tool whose musl behavior you know? `alpine`.
4. Just for the build stage? The regular big image. Size doesn't matter there.

> **REMEMBER**
>
> Use official images or a trusted source. Many companies mirror base images into their own registry and maintain them centrally. Vendors offer hardened versions as well.

IN THE INTERVIEW

Alpine or slim?

For most runtimes `slim`, because glibc brings fewer surprises. Alpine if the application runs reliably on musl and size really matters. For static binaries, go straight to distroless or `scratch`.

IN THE INTERVIEW

What is distroless?

Images without a shell or package manager, containing only the runtime. Fewer packages mean fewer vulnerabilities, and an attacker finds no tools. The price is harder troubleshooting.

KEY TAKEAWAYS

- `slim` is the good default.
- musl breaks some glibc assumptions.
- distroless has no shell, so no shell form.
- The build stage is allowed to be big.

21

Node.js

Separate your dependencies, and please don't use `npm start`.

Dockerfile

```
# syntax=docker/dockerfile:1
FROM node:24-slim AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci --omit=dev

FROM node:24-slim AS build
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM node:24-slim
ENV NODE_ENV=production
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY --from=build /app/dist ./dist
USER node
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

Here's what happens:

- `deps` produces only the production dependencies.
- `build` installs everything and builds the code.
- The last stage takes both results, without dev packages.
- `USER node` uses the user the official image ships with.

The Rules

- `npm ci` instead of `npm install`: It sticks exactly to the lockfile and aborts on discrepancies.
- `--omit=dev` leaves out dev packages.
- `node_modules` goes in `.dockerignore`. Native modules from a Mac won't run on Linux.

- Set `NODE_ENV=production`. Many libraries behave differently because of it.

> **PITFALL**
>
> `CMD ["npm", "start"]` is a trap. `npm` then sits between Docker and your application and doesn't forward signals cleanly. Start directly with `node`.

Shutting Down Cleanly

As PID 1, Node has to handle `SIGTERM` itself:

```
process.on('SIGTERM', () => {
  server.close(() => process.exit(0))
})
```

More on this in chapter 27.

> **IN THE INTERVIEW**
>
> **Why `npm ci` instead of `npm install` in the Dockerfile?**
>
> `npm ci` installs exactly what's in the lockfile, deletes `node_modules` first, and fails on discrepancies. That makes builds reproducible.

KEY TAKEAWAYS

- Copy the lockfile first, use `npm ci`.
- Three stages: `deps`, `build`, runtime.
- Start `node` directly, not `npm`.

22

Python

A virtual environment that can move house.

Dockerfile

```
# syntax=docker/dockerfile:1
FROM python:3.14-slim AS build
RUN python -m venv /opt/venv
COPY requirements.txt .
RUN /opt/venv/bin/pip install --no-cache-dir -r requirements.txt

FROM python:3.14-slim
ENV PYTHONUNBUFFERED=1 PATH=/opt/venv/bin:$PATH
RUN useradd --system --uid 10001 --no-create-home app
WORKDIR /app
COPY --from=build /opt/venv /opt/venv
COPY src/ ./src/
USER 10001
EXPOSE 8000
CMD ["gunicorn", "--bind", "0.0.0.0:8000", "src.app:app"]
```

Here's what happens:

- The `venv` sits at a fixed path so it can move to the last stage.
- Both stages use the same Python base. Otherwise compiled packages won't fit.
- `PYTHONUNBUFFERED=1` writes logs immediately instead of buffering them.
- `--no-cache-dir` leaves out the pip cache.

> **PITFALL**
>
> A `requirements.txt` with `requests` and no version is not reproducible. Use a pinned lockfile, for instance with `pip-compile`, Poetry, or `uv.lock`.

With uv

`uv` is a fast package manager with a lockfile:

```
RUN pip install uv
COPY pyproject.toml uv.lock ./
RUN uv sync --frozen --no-dev --no-install-project
```

```
COPY . .
RUN uv sync --locked --no-dev
```

`--frozen` takes the lockfile unchanged, and `--no-dev` leaves out development packages. `--no-install-project` installs only the dependencies, since the code isn't in the image yet. That keeps this layer in the cache. `--locked` aborts if the lockfile no longer matches.

> **PITFALL**
>
> Native packages need `gcc` and headers to build. Install them only in the build stage. Runtime libraries like `libpq`, however, belong in the last stage. If they're missing, you won't get an `ImportError` until startup.

IN THE INTERVIEW

Why do you copy a `venv` between stages?

Compilers and headers stay in the build stage, while the installed packages travel along at one path. Both stages need the same base so the binary extensions fit.

KEY TAKEAWAYS

- `venv` at a fixed path, copied to the last stage.
- `PYTHONUNBUFFERED=1` for logs.
- Lockfile, no loose versions.
- Build tools in the build stage, runtime libraries in the final image.

23

Go

One binary, almost no image.

Go produces a single file. If it's statically linked, it needs almost nothing.

Dockerfile

```
# syntax=docker/dockerfile:1
FROM golang:1.26 AS build
WORKDIR /src
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 go build -trimpath -ldflags="-s -w" -o /out/
app ./cmd/app

FROM gcr.io/distroless/static-debian12:nonroot
COPY --from=build /out/app /app
EXPOSE 8080
ENTRYPOINT ["/app"]
```

Flag	Effect
`CGO_ENABLED=0`	Static binary with no C library.
`-trimpath`	No local paths in the file.
`-ldflags="-s -w"`	Leave out the symbol table and debug info. Smaller.

With `scratch`

`scratch` contains nothing, so no CA certificates either. Without them, HTTPS calls fail.

```
FROM scratch
COPY --from=build /etc/ssl/certs/ca-certificates.crt /etc/ssl/
certs/
COPY --from=build /out/app /app
USER 10001
ENTRYPOINT ["/app"]
```

Distroless `static` already ships with certificates, time zones, and a `nonroot` user. That's the more convenient route.

PITFALL

Without `CGO_ENABLED=0`, the binary may be dynamically linked against glibc, for example with packages like `net`. In `scratch`, startup then reports `no such file or directory`, even though the file is there.

IN THE INTERVIEW

Why is Go so good for small images?

It builds a static binary. With `CGO_ENABLED=0` and multi-stage you get an image of a few megabytes, without compilers and libraries.

KEY TAKEAWAYS

- `CGO_ENABLED=0` for static linking.
- Copy `go.mod` and `go.sum` first.
- `scratch` needs CA certificates.
- Distroless `static` is more convenient.

24

Java

JRE instead of JDK, and layers for the cache.

Dockerfile

```
# syntax=docker/dockerfile:1
FROM eclipse-temurin:25-jdk AS build
WORKDIR /src
COPY mvnw pom.xml ./
COPY .mvn .mvn
RUN ./mvnw -B dependency:go-offline
COPY src src
RUN ./mvnw -B package -DskipTests

FROM eclipse-temurin:25-jre
RUN useradd --system --uid 10001 --no-create-home app
WORKDIR /app
COPY --from=build /src/target/*.jar app.jar
USER 10001
EXPOSE 8080
ENTRYPOINT ["java", "-XX:MaxRAMPercentage=75", "-jar",
"app.jar"]
```

- `dependency:go-offline` comes before `COPY src` so the dependency download stays in the cache.
- The final image contains only the JRE.
- `MaxRAMPercentage` sizes the heap to the container limit.

> **REMEMBER**
>
> Modern JVMs detect container limits. Without a setting, the heap takes only a quarter of the memory. That's why you set `MaxRAMPercentage`, say to 75.

Layered JARs with Spring Boot

A single big JAR changes completely with every build. Spring Boot can split it into layers. The stages below replace the last stage of the Dockerfile above. So the JAR comes from the `build` stage. It needs the layer index that the Spring Boot plugins generate by default.

```
FROM eclipse-temurin:25-jre AS extract
WORKDIR /x
COPY --from=build /src/target/*.jar app.jar
RUN java -Djarmode=tools -jar app.jar extract --layers --
destination out

FROM eclipse-temurin:25-jre
WORKDIR /app
COPY --from=extract /x/out/dependencies/ ./
COPY --from=extract /x/out/spring-boot-loader/ ./
COPY --from=extract /x/out/snapshot-dependencies/ ./
COPY --from=extract /x/out/application/ ./
ENTRYPOINT ["java", "-jar", "app.jar"]
```

The `app.jar` at the end is no longer the big JAR. It contains only the application code and points to the unpacked libraries. With new code, only the small last layer changes, and pushes get much shorter.

Your Own JRE with jlink

`jlink` builds a runtime from only the modules your application needs. `jdeps` finds them.

```
jlink --add-modules java.base,java.logging,java.net.http \
      --strip-debug --no-header-files --no-man-pages \
      --output /opt/jre
```

> **PITFALL**
>
> `ENTRYPOINT java -jar app.jar` in shell form means: The JVM receives no `SIGTERM`, and there is no orderly shutdown.

> **IN THE INTERVIEW**
>
> **How do you optimize a Java image?**
>
> Multi-stage with a JDK for building and a JRE for runtime, layered JARs for the cache, optionally `jlink`. And `MaxRAMPercentage`, so the heap fits the container limit.

KEY TAKEAWAYS

- JDK to build, JRE to run.
- Layered JARs keep pushes small.
- `MaxRAMPercentage` instead of a fixed heap size.

25

Static Sites with nginx

A single-page app with no Node at runtime.

A frontend needs Node only for building. nginx serves it.

Dockerfile

```
# syntax=docker/dockerfile:1
FROM node:24-slim AS build
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM nginxinc/nginx-unprivileged:stable-alpine
COPY nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 8080
```

nginx.conf

```
server {
  listen 8080;
  root /usr/share/nginx/html;
  location / {
    try_files $uri $uri/ /index.html;
  }
}
```

`try_files` sends unknown paths to `index.html` so the client-side router can take over.

> **REMEMBER**
>
> The `nginx-unprivileged` image runs without root privileges and therefore listens on 8080 instead of 80. The `stable-alpine` tag serves only as an example here. In production you pin it.

Why `daemon off`?

nginx normally goes to the background. In a container, that ends PID 1, and the container stops. That's why nginx runs in the foreground with `daemon off;`. The official image already handles this in its `CMD`.

> **PITFALL**
>
> `VITE_` variables or similar build values are baked into the delivered JavaScript. The image can't simply be reconfigured afterward. Load configuration at runtime or replace placeholders at startup.

> **REMEMBER**
>
> The official nginx image fills files from `/etc/nginx/templates/` with environment variables at startup using `envsubst`. That way you don't need your own entrypoint script.

IN THE INTERVIEW

Why does nginx run with `daemon off` in a container?

A container lives as long as PID 1. If nginx goes to the background, the start process ends and the container stops. So nginx stays in the foreground.

KEY TAKEAWAYS

- Node only in the build stage.
- `nginx-unprivileged` on port 8080.
- `try_files` for the client-side router.
- Containers need a foreground process.

PART V

Security and Runtime

Non-root, signals, pinning, scanning, and signing.

26

Non-Root

The simplest big security win.

By default, a container runs as `root`. Container `root` is restricted, but it's the same kernel. With a kernel hole or a misconfiguration, that quickly turns into a problem for the host.

Creating a User

Debian and Ubuntu:

```
RUN groupadd --system --gid 10001 app \
    && useradd --system --uid 10001 --gid app --no-create-home
app
```

Alpine:

```
RUN addgroup -S -g 10001 app && adduser -S -u 10001 -G app app
```

Then:

```
COPY --chown=10001:10001 . /app
USER 10001
```

> **REMEMBER**
>
> Pick a high UID like 10001. That way it won't collide with users on the host.

> **PICTURE THIS**
>
> *Even in your own hotel room, you don't leave the master key in the lock. Whoever breaks in should get this one room and nothing else.*

Typical Hurdles

- **Low ports:** Ports below 1024 are not allowed for non-root everywhere. Listen on 8080.
- **Write permissions:** The application may write only to selected folders.

- **Volumes:** A mounted volume may belong to `root`. Plan for the permissions.

Tighter Still at Startup

The image lays the foundation, and the startup does the rest:

```
docker run --read-only --tmpfs /tmp \
  --cap-drop=ALL --security-opt=no-new-privileges app:1.0
```

Flag	Effect
`--read-only`	Makes the container's filesystem read-only.
`--tmpfs /tmp`	A writable area in memory.
`--cap-drop=ALL`	All Linux capabilities gone.
`no-new-privileges`	Processes can't gain privileges, for example through `setuid`.

> **PITFALL**
>
> `USER` in the Dockerfile can be overridden at startup: `docker run --user 0` makes it `root` again. In Kubernetes you enforce it with `runAsNonRoot` and matching policies.

IN THE INTERVIEW

How do you make an image non-root?

I create a user with a fixed UID, set permissions with `--chown`, and name it with `USER 10001`. The application listens on a high port. In Kubernetes, `runAsNonRoot` checks this.

KEY TAKEAWAYS

- Dedicated UID, `USER` as a number.
- High ports, tight write permissions.
- `--read-only`, `--cap-drop=ALL`, `no-new-privileges` at startup.

27

PID 1 and Signals

Why your container takes ten seconds to die.

What Happens on `docker stop`

```
docker stop  -->  SIGTERM to PID 1
             -->  wait up to 10 seconds
             -->  SIGKILL, exit code 137
```

If the process doesn't react, it dies hard: no connections closed, no data saved.

```
docker stop -t 30 web
```

Memory aid: `-t` as in **t**imeout: Give it 30 seconds to clean up.

Why PID 1 Is Special

The first process in the container has two special roles:

1. **Signals:** The kernel ignores signals for PID 1 if the process has no handler of its own for them.
2. **Orphans:** PID 1 adopts orphaned processes and has to collect their exit. Otherwise **zombies** remain.

> **PICTURE THIS**
>
> *PID 1 is the container's janitor. He accepts the eviction notice and cleans up what others leave lying around. A shell that doesn't work for him leaves the mail on the floor.*

The Three Common Mistakes

Mistake	Fix
Shell form: `CMD node app.js`	Exec form: `CMD ["node", "app.js"]`
Script without `exec`	Last line: `exec "$@"`
`npm start` as a middleman	Start `node` directly

An Init Process

If your application starts child processes, or has no signal handler, put a small init in front of it:

```
docker run --init app:1.0
```

Memory aid: Docker puts a tiny init in front as PID 1.

In the image itself, you do it with `tini`:

```
RUN apt-get update && apt-get install -y --no-install-recommends
tini \
    && rm -rf /var/lib/apt/lists/*
ENTRYPOINT ["/usr/bin/tini", "--"]
CMD ["python", "app.py"]
```

`tini` starts your program as a child, forwards signals, and clears away zombies.

> **REMEMBER**
>
> Kubernetes has no `--init`. There the init belongs in the image, or the application handles it properly itself. And you have `terminationGracePeriodSeconds` instead of `-t`.

> **PITFALL**
>
> An application with a long shutdown needs more time than the default grace period. Exit code 137 on every deployment is a sign that it's being killed hard. Check signals and grace periods.

IN THE INTERVIEW

Why does `docker stop` sometimes take exactly ten seconds?

The `SIGTERM` isn't arriving or is being ignored. Docker waits ten seconds and sends `SIGKILL`. A typical cause is the shell form, because the shell is then PID 1 and doesn't forward the signal.

IN THE INTERVIEW

What do you need `tini` for?

As a small PID 1 process. It forwards signals and reaps orphaned processes. Otherwise zombies pile up if the application doesn't do that itself.

KEY TAKEAWAYS

- `SIGTERM`, ten seconds, `SIGKILL`.
- Exec form, `exec "$@"`, no `npm start`.
- `--init` or `tini` with child processes.

28

Pinning

Same Dockerfile, same result. Almost.

A tag is a movable sticker. `node:24-slim` may point to a different image tomorrow. A **digest** is the hash of the content and never changes.

```
FROM node:24-slim                       # moves
FROM node:24.22.2-slim                  # better
FROM node:24-slim@sha256:9f86d081...    # fixed
```

```
docker buildx imagetools inspect node:24-slim
```

Memory aid: Shows the digest behind the tag.

Level	Advantage	Disadvantage
`:latest`	Always new.	Unpredictable. Unsuitable for production.
`:24-slim`	Gets security updates.	Not reproducible.
`@sha256:...`	Absolutely fixed.	Updates don't arrive by themselves.

> **PICTURE THIS**
>
> *A tag is like "the restaurant on the corner": Tomorrow it may be a different one. A digest is the fingerprint of the dish.*

The Problem with Fixed Digests

A fixed digest gets no security updates. That's why it needs a bot: **Renovate** or **Dependabot** automatically opens a pull request when a new digest appears. That way you stay pinned and still current.

The Other Sources Too

- **Packages:** `npm ci`, `go.sum`, `uv.lock`, pinned `requirements`.
- **Downloads:** `ADD --checksum` or `sha256sum -c`.

- **System packages:** Exact `apt` versions disappear from the repositories. Many people pin the base instead and rebuild regularly.

> **PITFALL**
>
> An identical Dockerfile doesn't automatically produce an identical image. Timestamps and fresh packages change the digest. Bit-for-bit reproducible builds take more effort, for example `SOURCE_DATE_EPOCH`.

IN THE INTERVIEW

Tag or digest?

In production a digest, or at least a fixed version tag, plus an update bot. Only then is it clear what's running. The bot makes sure security updates still arrive.

KEY TAKEAWAYS

- Tags move, digests don't.
- Digest plus Renovate is pinned and current.
- Lockfiles and checksums for all sources.

29

Scanning and SBOM

What's inside, and what of it is vulnerable?

Scanning

```
trivy image --severity HIGH,CRITICAL --exit-code 1 app:1.0
```

Memory aid: Look for serious holes. `--exit-code 1` makes the pipeline fail.

```
trivy image --ignore-unfixed app:1.0
```

Memory aid: Show only holes that already have a fix.

```
docker scout cves app:1.0
```

Memory aid: Docker's own alternative. Checks dependencies against CVE databases.

```
trivy config .
```

Memory aid: Checks not the image but the Dockerfile and configuration.

Other well-known scanners are Grype, Snyk, and your registry's scanner.

> **PITFALL**
>
> A scan is a snapshot. An image that's clean today is vulnerable in three months, even though nothing has changed. Scan stored images too, and rebuild regularly.

> **REMEMBER**
>
> Not every CVE is exploitable. Check whether the affected code runs and whether there's a fix. Fixing an image full of irrelevant findings is work in the wrong place. The best remedy is often a smaller base.

SBOM

An **SBOM** (Software Bill of Materials) is the image's ingredient list. If a new hole shows up tomorrow, you see immediately which images are affected, without rescanning.

```
syft app:1.0 -o spdx-json > sbom.json
```

Memory aid: Generate the ingredient list in SPDX format.

```
docker buildx build --sbom=true --provenance=mode=max --push
-t registry/app:1.0 .
```

Memory aid: BuildKit generates the SBOM and proof of origin right during the build and attaches them to the image.

> **PICTURE THIS**
>
> *The SBOM is the ingredient list on food packaging. When a recall is announced, you don't have to open every package.*

Format	Known for
SPDX	ISO standard, strong on licenses.
CycloneDX	From the OWASP world, strong on security.

IN THE INTERVIEW

How do you deal with vulnerabilities in images?

A scan in the pipeline that aborts on `HIGH` and `CRITICAL`, provided a fix exists. Plus regular scans of stored images and automatic rebuilds. Small base images cut down the number of findings.

IN THE INTERVIEW

What is an SBOM for?

It lists all the components of an image. With a new hole like Log4Shell, you know immediately where you're affected. License checks and compliance use it too.

KEY TAKEAWAYS

- `trivy image` in the pipeline.
- Rescan stored images too.
- An SBOM is the ingredient list.
- Rebuilds are maintenance, not a special case.

30

Signing and Provenance

Prove that this image really comes from you.

A registry doesn't tell you who built an image. Two attestations close the gap:

- **Signature:** This image was approved by this identity.
- **Provenance:** How, when, and from what it was built.

Signing with cosign

You always sign the digest, never a tag.

```
cosign sign registry/app@sha256:abc123...
```

Memory aid: Keyless: You log in via OIDC, and cosign fetches a short-lived certificate. In CI it uses the pipeline's identity, and `--yes` skips the prompt.

```
cosign verify \
  --certificate-identity-regexp \
    '^https://github\.com/acme/app/\.github/workflows/.*' \
  --certificate-oidc-issuer https://token.actions.githubusercont
ent.com \
  registry/app@sha256:abc123...
```

So when verifying, you state **who** must have signed. Anchor the pattern with `^` and name the repository. A pattern like `acme/.*` would trust every workflow in the organization.

> **PICTURE THIS**
>
> *The signature is the seal on the letter. It's only useful if the recipient actually checks that it's intact and comes from the right person.*

Provenance

Provenance describes the origin: source, commit, builder, parameters. The framework for it is called **SLSA**.

```
docker buildx build --provenance=mode=max --push -t registry/
app:1.0 .
```

In the End, Someone Has to Check

A signature nobody checks is useless. In Kubernetes, an admission controller checks at deployment, for example Kyverno or Sigstore Policy Controller: Unsigned images are rejected.

> **PITFALL**
>
> With `mode=max`, the provenance can contain the build arguments. One more reason never to pass secrets as `--build-arg`.

IN THE INTERVIEW

How do you make sure only trusted images run?

I sign the digest with cosign in CI and generate provenance and an SBOM. At deployment, an admission controller checks the signature and the identity. Unsigned images don't get into the cluster.

KEY TAKEAWAYS

- Always sign the digest.
- Keyless cosign uses OIDC.
- Provenance explains the origin.
- Checking is part of deployment.

PART VI

Day to Day

Lint, CI, troubleshooting, and three Dockerfiles from real life.

31

Lint: hadolint and Build Checks

A robot finds half the mistakes before review.

hadolint

```
hadolint Dockerfile
```

Memory aid: **Lint** is the fluff in the dryer: The tool picks out the small mistakes.

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

Memory aid: The same without installing anything, via an image.

```
Dockerfile:3 DL3008 warning: Pin versions in apt get install
Dockerfile:5 DL3025 warning: Use arguments JSON notation for CMD
and ENTRYPOINT arguments
```

The rules you should know:

Rule	Meaning
DL3002	The last `USER` should not be `root`.
DL3003	`WORKDIR` instead of `cd`.
DL3006	Always give `FROM` a tag.
DL3007	Don't use `latest`.
DL3008	Pin versions in `apt-get install`.
DL3009	Delete the `apt` lists.
DL3015	Use `--no-install-recommends`.
DL3020	`COPY` instead of `ADD` for files.
DL3025	JSON form for `CMD` and `ENTRYPOINT`.
DL4006	Set `pipefail` with pipes.

You can turn off a rule selectively, and with a reason:

```
# Base is pinned by digest, packages come with it
# hadolint ignore=DL3008
RUN apt-get install -y --no-install-recommends curl
```

For the whole project there's the `.hadolint.yaml` file:

`.hadolint.yaml`

```
failure-threshold: warning
ignored:
  - DL3008
trustedRegistries:
  - registry.example.com
```

Build Checks

BuildKit has been doing its own checks for a while now. Warnings appear on every build, and with `--check` only the check runs, without building.

```
docker build --check .
```

Memory aid: Just check, don't build.

Examples of rules:

Rule	Finds
`JSONArgsRecommended`	`CMD` or `ENTRYPOINT` in shell form.
`SecretsUsedInArgOrEnv`	Suspicious names in `ARG` or `ENV`.
`CopyIgnoredFile`	`COPY` of a file that `.dockerignore` excludes.
`UndefinedVar`	Variables that aren't defined.
`FromAsCasing`	Inconsistent casing of `FROM` and `AS`.

As a requirement in the Dockerfile:

```
# syntax=docker/dockerfile:1
# check=error=true
```

> **PITFALL**
>
> With `error=true` and `syntax=docker/dockerfile:1`, new rules can break old builds. Pin the syntax version more precisely if you set this. The Docker docs explicitly recommend it.

IN THE INTERVIEW

How do you ensure the quality of Dockerfiles?

hadolint and `docker build --check` run in the pipeline, along with `trivy config`. That catches standard mistakes before a human does the review.

KEY TAKEAWAYS

- `hadolint` and `docker build --check`.
- Exceptions only with a reason.
- `# check=error=true` turns warnings into errors.

32

CI

From commit to signed image.

The Order

```
Lint -> Build -> Test -> Scan -> Push -> Sign
```

Only when everything is green does the image go to the registry.

GitHub Actions

.github/workflows/image.yml

```yaml
name: image
on:
  push:
    branches: [main]
jobs:
  build:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    steps:
      - uses: actions/checkout@v7
      - uses: docker/setup-buildx-action@v4
      - uses: docker/login-action@v4
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
      - uses: docker/build-push-action@v7
        with:
          context: .
          push: true
          tags: ghcr.io/acme/app:${{ github.sha }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

REMEMBER

The Action versions are examples. Check the current ones, and in security-critical pipelines pin Actions to a commit hash.

Tags

- **Commit SHA** for every image. Unique and traceable.
- **Version**, such as `1.4.2`, for releases.
- Set tags in the registry to immutable where possible.

Building Without the Docker Socket

Mounting the Docker socket into a CI job effectively gives the job root on the host. These are the alternatives:

Tool	Idea
BuildKit rootless	BuildKit as its own service without root privileges.
Buildah	Builds images without a daemon.
Kaniko	Used to build images in a pod. The Google project was archived in 2025, so check for forks.

PITFALL

Build once and promote the same image through the environments. If you rebuild for each environment, you're not testing what runs in production. Configuration arrives at startup, not at build time.

IN THE INTERVIEW

What does a good image pipeline look like?

Lint, build with an exported cache, tests, scan, push, and signature. The tag is the commit SHA. The same image then moves through all environments, and configuration comes from outside.

IN THE INTERVIEW

Why not mount the Docker socket in CI?

Whoever has the socket can start privileged containers and take over the host. Rootless BuildKit or a separate builder is better.

KEY TAKEAWAYS

- Lint, build, test, scan, push, sign.
- `cache-to` and `cache-from` on fresh runners.
- Build once, same image everywhere.
- No Docker socket in CI jobs.

33

When the Build Fails

The most common error messages and what they mean.

The Method

1. Read the complete output: `--progress=plain`.
2. Build the last working stage and look inside.
3. Rule out the cache as the cause: `--no-cache`.

```
docker build --target build -t debug .
```

Memory aid: Build up to the stage that still works.

```
docker run --rm -it debug sh
```

Memory aid: Go inside and run the broken command by hand.

Common Messages

Message	Cause
`failed to compute cache key: ... not found`	File isn't in the context or is listed in `.dockerignore`.
`pull access denied`	Wrong name, or the image is private. `docker login` needed.
`toomanyrequests`	Registry rate limit. Log in or use a mirror.
`no space left on device`	The cache is filling the disk. `docker builder prune`.
`Unable to locate package`	`apt-get update` is missing from the same `RUN`.
`404 Not Found` with `apt`	Old package lists from the cache. `--no-cache` or `update` in the same `RUN`.
`exit code: 127`	The command doesn't exist in the image.
`exec format error`	Foreign architecture without an emulator.

> **PITFALL**
>
> "Works on my machine, not in CI" often comes down to a cache that serves something stale locally. Build locally once with `--no-cache --pull`. That shows the real state.

Why Is Nothing Being Rebuilt?

- The changed file is listed in `.dockerignore`.
- The `RUN` wasn't changed, so the cache applies. The text counts, not the network.

Why Is Everything Always Rebuilt?

- `COPY . .` is too far up.
- A file changes on every build, such as `.git` in the context or a timestamp.
- A changing `--build-arg` sits far up.

IN THE INTERVIEW

A build works locally but not in CI. How do you proceed?

I build locally with `--no-cache --pull` and compare base, platform, and context. Often the local cache is serving something old, or the architecture differs.

KEY TAKEAWAYS

- `--progress=plain`, then `--target`, then look inside.
- `compute cache key`: context and `.dockerignore`.
- `--no-cache --pull` shows the true state.

34

When the Container Fails

It builds but doesn't start. Or not for long.

The First Minute

```
docker ps -a
```

Memory aid: Stopped containers too, with exit code.

```
docker logs web
```

Memory aid: What were its last words?

```
docker inspect --format '{{.State.ExitCode}} {{.State.OOMKilled}}' web
```

Memory aid: The exit code, and whether the kernel killed it over memory.

```
docker run --rm -it --entrypoint sh app:1.0
```

Memory aid: Start a shell instead of the app, and look around.

```
docker exec -it web sh
```

Memory aid: Go into a running container.

```
docker diff web
```

Memory aid: Which files has the container changed?

Exit Codes

Code	Meaning
`0`	Exited cleanly. Maybe there was nothing to do?
`1`	The application reports an error.
`126`	File found but not executable. Check the permissions.
`127`	Command not found.
`137`	`SIGKILL` (128 + 9). OOM, or `docker stop` after the grace period expired.
`139`	Segfault (128 + 11). Often the wrong library.
`143`	`SIGTERM` (128 + 15). A clean stop from outside.

Common Messages

Message	Cause
`exec format error`	Wrong architecture, or the script is missing its shebang.
`no such file or directory`, even though the file is there	Windows line endings in the script, or the loader is missing: a glibc binary on Alpine or `scratch`.
`permission denied`	File isn't executable, or the wrong user. `COPY --chmod=755`.
Container exits immediately	The process went to the background, or there was nothing to do.
Port not reachable	Application listens on `127.0.0.1` instead of `0.0.0.0`.

PICTURE THIS

The exit code is the death certificate. Above 128 means: A signal did it. Subtract 128 and you have the signal number.

PITFALL

If an application listens only on `127.0.0.1`, it can't be reached from outside the container. `-p` points to the container's network interface, not to its loopback. Bind to `0.0.0.0`.

PITFALL

Line endings are a Windows classic. A script with CRLF looks for `/bin/sh\r`, and the message says only `no such file or directory`. The fix: `.gitattributes` or `dos2unix`.

IN THE INTERVIEW

A container exits immediately with exit code 137. What do you check?

137 means `SIGKILL`. I check `OOMKilled` in `docker inspect`. If it's `true`, the memory limit doesn't fit. Otherwise it was killed from outside, for example by `docker stop` after the grace period expired.

KEY TAKEAWAYS

- `docker logs`, `inspect`, `--entrypoint sh`.
- Above 128: signal, minus 128.
- `no such file` can be the loader or CRLF.
- `0.0.0.0` instead of `127.0.0.1`.

35

Improving Three Dockerfiles

Real patterns from everyday work. Find the mistakes yourself first.

Case 1: nginx

Before:

```
FROM debian:latest
RUN apt-get update && apt-get install -y nginx; \
    echo "daemon off;" >> /etc/nginx/nginx.conf
EXPOSE 80
CMD ["/usr/sbin/nginx"]
```

What's wrong?

- `debian:latest` moves and is needlessly big.
- The `;` lets the build carry on even if `apt-get` fails.
- The `apt` lists stay in the layer.
- It runs as `root` on port 80.

After:

```
FROM nginxinc/nginx-unprivileged:stable-alpine
COPY site/ /usr/share/nginx/html/
EXPOSE 8080
```

The ready-made image already brings everything. Your own packages aren't needed at all.

Case 2: Node

Before:

```
FROM node:16.17.1
ENV NODE_ENV=production
COPY ["package.json", "package-lock.json", "./"]
RUN npm install --production
COPY . .
EXPOSE 80
CMD ["node", "app.js"]
```

What's wrong?

- Node 16 stopped getting updates long ago.
- The full version of the image is big.
- No `WORKDIR`, so the files land in `/`.
- `npm install` doesn't stick strictly to the lockfile.
- It runs as `root` on port 80.

Credit where it's due: Copying the lockfiles before the code is already right.

After:

```
FROM node:24-slim
ENV NODE_ENV=production
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci --omit=dev
COPY . .
USER node
EXPOSE 8080
CMD ["node", "app.js"]
```

The application has to listen on port 8080 for this.

Case 3: Database Test with a Password

Before:

```
FROM postgres:15-alpine
RUN apk --no-cache add bash
ENV POSTGRES_HOST=my-db.example.com
ENV POSTGRES_USER=admin
ENV POSTGRES_PASSWORD=secret
COPY ./test-connection.sh /usr/local/bin/test-connection.sh
RUN chmod +x /usr/local/bin/test-connection.sh
CMD ["/usr/local/bin/test-connection.sh"]
```

What's wrong?

- The password is in the image forever.
- `HOST` and `USER` also belong to the environment, not in the image.
- The whole Postgres server is on board, even though only `psql` is needed.
- `chmod` makes a second layer. `COPY --chmod` is enough.

After:

```
FROM alpine:3.23
RUN apk add --no-cache postgresql-client
COPY --chmod=755 test-connection.sh /usr/local/bin/
USER 10001
CMD ["test-connection.sh"]
```

At startup, the values are passed in from outside:

```
docker run --rm --env-file db.env db-test:1.0
```

> **REMEMBER**
>
> If a secret was ever in an image, a new build isn't enough. The password counts as burned and has to be replaced.

KEY TAKEAWAYS

- Ready-made images replace many `RUN` lines.
- Configuration and secrets come from outside.
- Checklist: base, layers, cache, user, secrets.

PART VII

The Interview

Questions, live exercises, and a cheat sheet for the last evening.

36

The Questions

Short questions, short answers. For review before the interview.

Fundamentals

Image or container?

The image is the immutable template, the container the running process with a writable layer.

What is a layer?

An immutable layer holding the file changes of one step.

Which instructions create file layers?

`RUN`, `COPY`, `ADD`.

Why doesn't `rm` in a later `RUN` make the image smaller?

The file stays in the earlier layer.

What is the build context?

The files that `COPY` and `ADD` are allowed to see.

What is `.dockerignore` for?

Less data, no secrets, a more stable cache.

What is copy-on-write?

A file is copied into the container layer only when it's changed.

What does `latest` mean?

Just the default tag. No guarantee of being current.

Tag or digest?

The tag is a movable sticker, the digest a fixed content hash.

Instructions

`CMD` or `ENTRYPOINT`?

`ENTRYPOINT` is the program, `CMD` the default arguments.

Shell form or exec form?

Exec form starts directly, shell form through `/bin/sh -c`.

Why exec form?

The program becomes PID 1 and receives signals.

`COPY` or `ADD`?

`COPY`. `ADD` only for unpacking tars or URLs with a checksum.

`ENV` or `ARG`?

`ARG` only during the build, `ENV` at runtime too.

Can a secret go in `ENV`?

No. It then sits permanently in the metadata.

Does `EXPOSE` open a port?

No, it documents. Publishing is done with `-p`.

Why `WORKDIR` instead of `cd`?

`cd` works only within a single `RUN`.

Does `USER` affect `COPY`?

No. That's what `--chown` is for.

What happens to files after `VOLUME`?

Later changes to that path are lost during the build.

What does `ONBUILD` do?

It defers instructions to the image that builds on this one.

What does `SHELL` do?

It changes the shell for the shell form, often for `pipefail`.

What is `HEALTHCHECK` for?

It checks whether the application works, not just whether it runs.

Does Kubernetes use `HEALTHCHECK`?

No, probes apply there.

Cache and Build

How does the cache work?

Instruction by instruction. A miss rebuilds everything after it.

How does Docker recognize `RUN`?

By the instruction's text.

And `COPY`?

By the checksum of the files.

How do you order a Dockerfile?

Rarely changed things at the top, often changed things at the bottom.

Why copy the lockfile before the code?

That keeps the dependency layer in the cache.

Why `update` and `install` in the same `RUN`?

Otherwise `install` uses old package lists.

What is BuildKit?

The default build engine, with a graph, parallelism, and secrets.

What is a multi-stage build?

Multiple `FROM`s, and only the last stage becomes the image.

What does `--target` do?

It builds up to a specific stage.

What is a cache mount?

A folder for package caches that persists but doesn't end up in the image.

How do you cache in CI?

`--cache-to` and `--cache-from` with a registry or `gha`.

How do you use tokens during a build?

With `RUN --mount=type=secret`.

How do you clone a private repo?

With `RUN --mount=type=ssh` and `--ssh default`.

How do you build for multiple architectures?

`buildx build --platform ... --push`, ideally with cross-compiling.

What is an image index?

A list that points to one image per platform.

Bases and Languages

Alpine or slim?

`slim` as the default, Alpine when musl definitely fits.

What is musl?

Alpine's small C library, not glibc-compatible.

What is distroless?

A runtime with no shell or package manager.

What is `scratch`?

An empty image.

Why `npm ci`?

It sticks exactly to the lockfile.

Why not `npm start`?

It sits between Docker and Node and interferes with signals.

Why `PYTHONUNBUFFERED`?

Logs appear immediately.

Why `CGO_ENABLED=0`?

For a static Go binary.

Why a JRE instead of a JDK?

The compiler is superfluous at runtime.

Security and Operations

Why not `root`?

In a breakout, the damage would be greater.

Why `USER` as a number?

`runAsNonRoot` can verify only numbers.

What happens on `docker stop`?

`SIGTERM`, a ten-second wait, then `SIGKILL`.

Why is PID 1 special?

Without a handler it ignores signals, and it has to reap orphans.

What does `tini` do?

It's a small PID 1 that forwards signals and reaps zombies.

What does exit code 137 mean?

`SIGKILL`, often from OOM.

What is an SBOM?

An image's ingredient list.

What is provenance?

The proof of what an image was built from and how.

Why sign the digest?

A tag can change, a digest can't.

Why rebuild regularly?

So the base's security updates reach you.

Why is the Docker socket dangerous?

Whoever holds it effectively has root on the host.

When You Don't Know Something

Say so. Then think out loud: "I haven't used that exactly. I'd guess X, because Y. I'd look it up in the docs under Z." That shows more than guessed knowledge.

KEY TAKEAWAYS

- Short answer first. Details only if someone asks.
- Always think about layers, cache, and security.

37

Live Exercises

Five exercises that often come up in the interview.

> **REMEMBER**
>
> How it goes in the interview: Ask questions, think out loud, start small, improve. Ask first: Which language? Which target platform? How does the image run? Talk while you type.

Exercise 1: Write a Good Dockerfile for a Go Service

The skeleton from chapter 23 shows everything: two stages, dependencies first, `CGO_ENABLED=0`, distroless `nonroot`. Say why as well: size, security, cache.

Exercise 2: Why Is This Dockerfile Slow?

```
FROM python:3.14
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
CMD python app.py
```

- `COPY . /app` comes before `pip install`. Every code change reinstalls everything.
- There's no `.dockerignore`.
- The full version of the base is big.
- `CMD` is in shell form.
- It runs as `root`.

The solution:

```
FROM python:3.14-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
USER 10001
CMD ["python", "app.py"]
```

Exercise 3: The Image Is 1.2 GB. What Do You Do?

1. Measure with `docker history` and `dive`.
2. Switch the base, from full to `slim` or distroless.
3. Introduce multi-stage.
4. Remove dev dependencies and caches.
5. Check `.dockerignore`.

Exercise 4: `docker stop` Always Takes Ten Seconds

- Check the Dockerfile: Shell form? Script without `exec`? `npm start`?
- Use `docker exec web ps` to see what PID 1 is.
- Switch to exec form and handle `SIGTERM`, or add `--init`.

Exercise 5: A Token Was Found in the Image

1. Revoke the token immediately. That's the most important step.
2. Remove the image from the registry.
3. Fix the cause: Replace `ENV`, `ARG`, or `COPY` with secret mounts.
4. Rebuild and check with a secret scanner.
5. Extend the pipeline so this doesn't happen again.

IN THE INTERVIEW

How do you detect secrets in images?

I look in `docker history` and in the layers, for example with `dive`. In addition, secret scanners like Trivy's run in CI over the image and repository.

KEY TAKEAWAYS

- Ask first, then write.
- Always three sentences: size, cache, security.
- Found a secret? Revoke it first.

38

Cheat Sheet

Everything important on a few pages, for the last evening.

The Pattern for Almost Everything

```
# syntax=docker/dockerfile:1
FROM base:version AS build
WORKDIR /src
COPY lockfile ./
RUN install-dependencies
COPY . .
RUN build

FROM runtime:version
WORKDIR /app
COPY --from=build --chown=10001:10001 result .
USER 10001
EXPOSE 8080
HEALTHCHECK CMD ["/app/program", "health"]
ENTRYPOINT ["/app/program"]
```

One Sentence on Everything

Term	One-liner
Layer	Immutable, and deleting saves nothing.
Cache	Rarely at the top, often at the bottom.
Multi-Stage	Workshop and store.
`CMD`	Default arguments.
`ENTRYPOINT`	The program.
`ARG`	Only during the build.
`ENV`	At runtime too.
`EXPOSE`	Just docs.
PID 1	Exec form or `exec "$@"`.
Digest	The fixed fingerprint.

Build

```
docker build -t app:1.0 .
docker build -f Dockerfile.prod --target build -t app:dev .
docker build --no-cache --pull -t app:1.0 .
docker build --progress=plain -t app:1.0 .
docker build --check .
docker build --secret id=npmrc,src=$HOME/.npmrc -t app .
docker build --ssh default -t app .
docker buildx build --platform linux/amd64,linux/arm64 \
  -t registry/app:1.0 --push .
```

Inspecting

```
docker history app:1.0
docker image inspect app:1.0
docker buildx imagetools inspect app:1.0
dive app:1.0
docker logs web
docker run --rm -it --entrypoint sh app:1.0
docker system df
docker builder prune
```

Quality and Security

```
hadolint Dockerfile
trivy image --severity HIGH,CRITICAL --exit-code 1 app:1.0
trivy config .
syft app:1.0 -o spdx-json > sbom.json
cosign sign registry/app@sha256:...
```

Numbers You Need to Know

Value	Meaning
10 s	Grace period from `docker stop` to `SIGKILL`.
137	`SIGKILL`, that is, 128 + 9. Often OOM.
143	`SIGTERM`, that is, 128 + 15.
126, 127	Not executable, not found.
30 s, 30 s, 3	`HEALTHCHECK`: interval, timeout, retries.
10001	A typical unprivileged UID.

Checklist for Every Dockerfile

- Base pinned, small, and from a trusted source?
- Multi-stage if anything gets built?

- Lockfile copied before code?
- `.dockerignore` present?
- `CMD` and `ENTRYPOINT` in exec form?
- `USER` set as a number?
- No secrets in `ENV`, `ARG`, or layers?
- hadolint and scan green?

KEY TAKEAWAYS

- Understand layers and you understand Dockerfiles.
- Rarely at the top, often at the bottom.
- Small, non-root, pinned, scanned.

Index

ADD 22
Alpine 56
ARG 27
build cache 38
Build Checks 84
build context 15
BuildKit 41
buildx 41
Cache Mount 39
CGO_ENABLED 63
CI 87
CMD 24
COPY 22
COPY --link 23
Copy-on-Write 10
cosign 80
Digest 75
distroless 56
docker build 13
Docker Scout 77
Dockerfile 8
.dockerignore 15
ENTRYPOINT 24
ENV 27
exec format error 92
exit code 92
EXPOSE 31
FROM 18
glibc 56
hadolint 84
HEALTHCHECK 33
Heredoc 20
Image Index 49
image size 52
jlink 65
Kaniko 87
LABEL 31
Layer 10
MaxRAMPercentage 65
Multi-Arch 49
Multi-Stage 43
musl 56
nginx 67
Non-Root 70
npm ci 59
OCI 11
ONBUILD 35
PID 1 72
pipefail 35
Provenance 80
Renovate 75
RUN 19
runAsNonRoot 70
SBOM 77
scratch 18
Secret Mount 46
SHELL 35
SIGTERM 72
slim 56
SLSA 80
SSH mount 46
STOPSIGNAL 32
syntax 36
Tag 13
tini 72
Trivy 77
USER 29
uv 61
venv 61
VOLUME 31
WORKDIR 29
Zombie 72