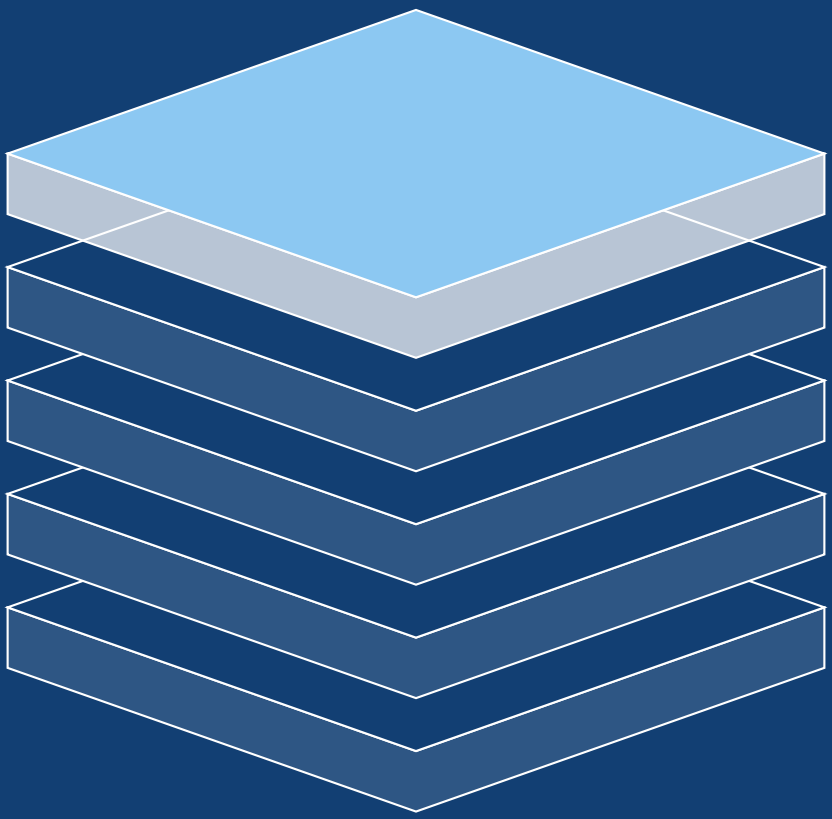
nyrvex.com

Dockerfiles

Hands-On

Kleine, schnelle und sichere
Container-Images bauen



Mit wenig Text fit fürs Interview

Dockerfiles

Hands-On

Kleine, schnelle und sichere Container-Images bauen

Mit wenig Text fit fürs Interview

Thomas Zachmann

Dockerfiles Hands-On. Kleine, schnelle und sichere Container-Images bauen.
Stand: November 2026. © 2026 Thomas Zachmann. Lizenz: CC BY-ND 4.0.

Geschrieben für Docker Engine 29 und Buildx 0.38 mit BuildKit. Die Dockerfiles, Befehle und Ausgaben in diesem Buch wurden nicht gebaut oder ausgeführt. Image-Tags, Größen und Ausgaben sind Beispiele und können je nach Version abweichen. Prüfe Tags und Versionen, bevor du sie übernimmst.

Schriften: Charter, Fira Sans, JetBrains Mono.

Inhalt

So liest du dieses Buch 5

Der Start **7**
1 Was ist ein Dockerfile? 8
2 Images und Layer 10
3 Das erste Image 13
4 Kontext und .dockerignore 15

Die Anweisungen **17**
5 FROM 18
6 RUN 20
7 COPY und ADD 23
8 CMD und ENTRYPOINT 25
9 ENV und ARG 28
10 WORKDIR und USER 30
11 EXPOSE, VOLUME, LABEL 32
12 HEALTHCHECK 34
13 SHELL, ONBUILD, Direktiven 36

Besser bauen **39**
14 Der Build-Cache 40
15 BuildKit 43
16 Multi-Stage-Builds 46
17 Secrets und SSH 49
18 Multi-Arch mit buildx 52
19 Kleine Images 55

Basis und Sprachen **57**
20 Basis-Images wählen 58
21 Node.js 61
22 Python 63
23 Go 65
24 Java 67
25 Statische Seiten mit nginx 69

Sicherheit und Laufzeit **71**
26 Non-Root 72
27 PID 1 und Signale 74
28 Pinning 77
29 Scannen und SBOM 79
30 Signieren und Provenance 82

Alltag **85**
31 Lint: hadolint und Build Checks 86
32 CI 89
33 Wenn der Build scheitert 92
34 Wenn der Container scheitert 94
35 Drei Dockerfiles verbessern 97

Das Interview **101**
36 Die Fragen 102
37 Live-Aufgaben 106
38 Spickzettel 108
Stichwörter 111

So liest du dieses Buch

Du hast bald ein Interview und musst Dockerfiles können. Nicht irgendwann, sondern bald. Dieses Buch ist genau dafür gebaut: wenig Text, viele kleine Beispiele, und zu jedem Thema die Frage, die dir wirklich gestellt wird.

Die Regeln

- **Tippe alles selbst.** Nichts kopieren. Deine Finger lernen mit.
- **Jeder Befehl einzeln.** Es gibt keine Skripte. Du tippst, du schaust, was passiert.
- **Jedes Beispiel steht für sich.** Lege einen leeren Ordner an, schreibe das Dockerfile, baue es.
- **Frage immer nach dem Warum.** Im Interview zählt nicht die Anweisung, sondern ihr Effekt auf Layer, Cache und Sicherheit.

> **MERKE**
>
> Die Beispiele in diesem Buch wurden nicht gebaut oder ausgeführt. Tags und Größen sind Beispiele. Prüfe, welche Version gerade aktuell ist.

Die Kästen

> **DAS BILD IM KOPF**
>
> *Ein Vergleich aus dem Alltag. Wenn du den behältst, kannst du das Thema herleiten.*

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

Eselsbrücke: Ein Befehl zum Tippen, mit Merkhilfe darunter.

> **STOLPERFALLE**
>
> Der Fehler, den fast alle machen. Und den Interviewer gern abfragen.

IM INTERVIEW

Eine typische Frage?

Eine Antwort in zwei, drei Sätzen. So kurz darf sie sein.

Dein Lernplan

Du hast ...	Dann lies ...
1 Abend	Kapitel 2, 8, 14, 16 und 26, dann Kapitel 36 (die Fragen) und 38 (Spickzettel).
2 Tage	Tag 1: Teil I und II. Tag 2: Teil III und die Kapitel 26 bis 29, dann Kapitel 36.
1 Woche	Tag 1: Teil I und II. Dann ein Teil pro Tag. Tag 6: die Live-Aufgaben aus Kapitel 37. Tag 7: Fragen wiederholen.

TEIL I

Der Start

Was ein Dockerfile ist, was ein Image ist und wie du das erste baust.

1

Was ist ein Dockerfile?

Ein Rezept, aus dem immer wieder dasselbe Image entsteht.

Ein Dockerfile ist eine Textdatei mit Anweisungen. Docker führt sie von oben nach unten aus. Am Ende steht ein Image. Die Kernidee passt in eine Zeile:

Dockerfile + Kontext = Image

Aus dem Image startest du beliebig viele Container.

> **DAS BILD IM KOPF**
>
> *Das Dockerfile ist das Rezept. Das Image ist der tiefgekühlte Kuchen, immer gleich und leicht zu verschicken. Der Container ist das Stück, das gerade auf dem Teller liegt.*

Die drei Begriffe

Begriff	Was es ist	Befehl
Dockerfile	Die Bauanleitung als Text.	`docker build`
Image	Unveränderliches Paket aus Dateien und Metadaten.	`docker push`, `pull`
Container	Ein laufender Prozess aus einem Image.	`docker run`

Warum nicht einfach installieren?

- **Reproduzierbar.** Das Rezept liegt im Git, jeder baut dasselbe.
- **Portabel.** Das Image läuft auf dem Laptop und im Cluster gleich.
- **Prüfbar.** Jede Änderung ist ein Diff im Code-Review.

Das kleinste sinnvolle Beispiel

Dockerfile

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

Zwei Zeilen: Worauf baue ich auf, und was soll beim Start passieren?

IM INTERVIEW

Was ist der Unterschied zwischen Image und Container?

Ein Image ist ein unveränderliches Paket aus Layern und Metadaten. Ein Container ist eine laufende Instanz davon, mit einer dünnen beschreibbaren Schicht oben drauf. Aus einem Image entstehen beliebig viele Container.

DAS NIMMST DU MIT

- Dockerfile + Kontext = Image.
- Image ist die Vorlage, Container der Prozess.
- Das Dockerfile gehört ins Git.

2

Images und Layer

Ein Image ist ein Stapel. Wer das versteht, versteht fast alles.

Ein Image besteht aus Layern. Jeder Layer speichert nur die Änderungen gegenüber dem Layer darunter. Zur Laufzeit legt Docker die Layer übereinander und zeigt sie als ein Dateisystem.

```
Container-Layer  (beschreibbar, nur zur Laufzeit)
-------------------------------------------
COPY . /app      (Layer 3, nur lesen)
RUN npm ci       (Layer 2, nur lesen)
FROM node:slim   (Layer 1, nur lesen)
```

> **DAS BILD IM KOPF**
>
> *Transparente Folien auf dem Overhead-Projektor. Jede Folie zeigt nur, was neu ist. Was auf einer Folie durchgestrichen wird, steht auf der Folie darunter trotzdem weiter.*

Was einen Layer erzeugt

Anweisung	Wirkung
`RUN`, `COPY`, `ADD`	Erzeugen einen Layer mit Dateien.
`ENV`, `CMD`, `LABEL`, `USER`, `EXPOSE`, `WORKDIR`	Ändern vor allem Metadaten. Kein Dateiinhalt.

Die wichtigste Konsequenz

Löschen in einem späteren Layer macht das Image nicht kleiner.

Dockerfile (schlecht)

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

Der zweite Layer vermerkt nur: „Diese Datei ist weg.“ Die ganze Datei liegt weiter im ersten Layer. Das Löschen muss im selben `RUN` passieren.

> **STOLPERFALLE**
>
> Das gilt auch für Geheimnisse. Ein Passwort, das in Layer 3 landet und in Layer 4 gelöscht wird, bleibt im Image lesbar. Jeder mit Pull-Recht kann es herausziehen.

Copy-on-Write

Ändert ein Container eine Datei aus einem unteren Layer, kopiert Docker sie erst nach oben in den Container-Layer. Der untere Layer bleibt unverändert. Darum teilen sich viele Container ein Image.

Layer werden geteilt

Layer werden über ihren Hash identifiziert. Zehn Images auf derselben 100-MB-Basis belegen diese 100 MB nur einmal, und ein Pull lädt nur fehlende Layer.

Was steckt im Image?

Ein Image besteht aus drei Dingen:

- **Manifest:** die Liste der Layer.
- **Config:** Metadaten wie `CMD`, `ENV` und `USER`.
- **Layer:** die Dateien als Archive.

Für mehrere Architekturen gibt es zusätzlich einen **Index**, siehe Kapitel 18. Alles ist im OCI-Format standardisiert.

Nachschauen

```
docker history app:1.0
```

Eselsbrücke: Die Geschichte des Images: jede Anweisung mit ihrer Größe.

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

Eselsbrücke: Der Ausweis: `Cmd`, `Env`, `User`, Layer-Hashes.

```
dive app:1.0
```

Eselsbrücke: Tauche in die Schichten: Das Zusatzwerkzeug zeigt Dateien pro Layer und verschwendeten Platz.

IM INTERVIEW

Warum wird ein Image nicht kleiner, wenn ich eine Datei in einem späteren `RUN` lösche?

Layer sind unveränderlich. Der spätere Layer enthält nur einen Löschvermerk, die Datei bleibt im früheren Layer. Löschen muss im selben `RUN` geschehen, oder du nimmst einen Multi-Stage-Build.

IM INTERVIEW

Wie sparen Layer Speicher?

Sie sind inhaltsadressiert. Gleiche Layer liegen nur einmal auf der Platte und in der Registry. Beim Pull und Push wird nur übertragen, was fehlt.

DAS NIMMST DU MIT

- Ein Image ist ein Stapel unveränderlicher Layer.
- `RUN`, `COPY`, `ADD` erzeugen Dateien-Layer.
- Löschen im späteren Layer spart nichts.
- `docker history` und `dive` zeigen, wo die Größe steckt.

3

Das erste Image

Bauen, starten, taggen, hochladen.

Ein kleines Projekt

server.py

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

Dockerfile

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

Bauen und starten

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

Eselsbrücke: `-t` wie **t**ag: Gib dem Image einen Namen. Der Punkt am Ende ist der Kontext, also dieser Ordner.

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

Eselsbrücke: `-p` ist Host-Port:Container-Port. `--rm` räumt den Container am Ende weg.

```
docker image ls app
```

Eselsbrücke: Zeigt das Image und seine Größe.

Der Name eines Images

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

Ohne Registry gilt Docker Hub. Ohne Tag gilt `latest`.

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

Eselsbrücke: Ein zweiter Name fürs selbe Image, wie ein Etikett. Nichts wird kopiert.

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

Eselsbrücke: Schiebe es zur Registry. Nur fehlende Layer werden übertragen.

STOLPERFALLE

`latest` heißt nicht „das Neueste“. Es ist nur der Standardname, wenn du keinen Tag nennst. Er kann uralt sein und sich jederzeit ändern. Nutze feste Tags oder Digests, siehe Kapitel 28.

IM INTERVIEW

Was bedeutet der Punkt in `docker build .`?

Das ist der Build-Kontext, also der Ordner, dessen Dateien der Builder sehen darf. Das Dockerfile wird standardmäßig dort gesucht. Mit `-f` nennst du eine andere Datei.

DAS NIMMST DU MIT

- `docker build -t name:tag .`
- `docker run --rm -p host:container name:tag`
- Tags sind Etiketten, `latest` ist keine Garantie.

4

Kontext und .dockerignore

Was der Builder sehen darf, entscheidest du.

Der **Build-Kontext** ist die Menge der Dateien, auf die `COPY` und `ADD` zugreifen können. Dateien außerhalb des Kontexts sind unerreichbar.

> **STOLPERFALLE**
>
> `COPY ../geheim.txt .` funktioniert nicht. Der Build kann den Kontext nicht verlassen. Das ist Absicht: Ein Dockerfile aus dem Internet soll nicht deine Platte lesen.

Warum `.dockerignore` wichtig ist

Ohne sie landet mit `COPY . .` alles im Image: `.git`, `node_modules`, `.env`, Testdaten.

- **Sicherheit:** Keine Geheimnisse im Image.
- **Geschwindigkeit:** Weniger Dateien müssen zum Builder.
- **Cache:** Eine geänderte `README.md` macht `COPY . .` nicht ungültig.

`.dockerignore`

```
.git
.env
.env.*
node_modules
dist
*.log
Dockerfile
.dockerignore
# Ausnahme: diese Datei doch mitnehmen
!.env.example
```

Muster	Bedeutung
`*.log`	Alle `.log`-Dateien in der obersten Ebene.
`**/*.log`	Alle `.log`-Dateien in jeder Tiefe.
`!datei`	Ausnahme. Die letzte passende Zeile gewinnt.

DAS BILD IM KOPF

Der Kontext ist der Koffer, den du zum Handwerker trägst. `.dockerignore` *ist die Liste, was zu Hause bleibt. Wer das ganze Haus mitbringt, wartet lange und zeigt zu viel.*

MERKE

Die Datei muss im Wurzelverzeichnis des Kontexts liegen. Für ein bestimmtes Dockerfile gibt es auch `Dockerfile.dockerignore` neben dem Dockerfile.

Andere Dockerfiles

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

Eselsbrücke: `-f` wie **f**ile: Das Rezept liegt woanders, der Kontext bleibt der Punkt.

IM INTERVIEW

Warum braucht man `.dockerignore`?

Sie begrenzt den Kontext. So bleiben Geheimnisse wie `.env` aus dem Image, der Build wird schneller, und irrelevante Dateien machen den Cache von `COPY` nicht ungültig.

DAS NIMMST DU MIT

- `COPY` sieht nur den Kontext.
- `.git`, `.env` und `node_modules` gehören in `.dockerignore`.
- `.dockerignore` schützt Sicherheit, Tempo und Cache.

TEIL II

Die Anweisungen

Jede Dockerfile-Anweisung, ihre Tücken und ihre Interviewfragen.

5

FROM

Jedes Image steht auf den Schultern eines anderen.

`FROM` legt die Basis fest. Alles Weitere baut darauf auf.

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

- `AS build` gibt der Stufe einen Namen für Multi-Stage, siehe Kapitel 16.
- `--platform` wählt die Architektur, siehe Kapitel 18.
- `scratch` ist kein Image, sondern das leere Nichts. Gut für statische Binärdateien.

Variablen in `FROM`

Nur `ARG` darf vor `FROM` stehen:

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

> **STOLPERFALLE**
>
> Ein `ARG` vor `FROM` gilt nur für `FROM`-Zeilen. Willst du den Wert später im Stage nutzen, schreibe dort noch einmal `ARG PYTHON_VERSION` ohne Wert.

> **IM INTERVIEW**
>
> **Was ist `FROM scratch`?**
>
> Ein leerer Start ohne Shell, Bibliotheken und Paketmanager. Du kopierst nur deine statische Binärdatei hinein. Das ergibt sehr kleine Images mit minimaler Angriffsfläche, aber auch nichts zum Debuggen.

DAS NIMMST DU MIT

- `FROM` ist die Basis, `AS` benennt die Stufe.
- `scratch` ist leer.
- `ARG` vor `FROM` muss im Stage neu deklariert werden.

6

RUN

Befehle zur Bauzeit. Jeder `RUN` ist ein Layer.

`RUN` führt einen Befehl im Bau-Container aus und speichert das Ergebnis als Layer.

Zwei Schreibweisen

```
RUN apt-get update              # Shell-Form
RUN ["python", "--version"]     # Exec-Form
```

Die Shell-Form läuft über `/bin/sh -c`. Sie kennt `&&`, Pipes und Variablen. Die Exec-Form startet das Programm direkt, ohne Shell.

Das Muster für Pakete

Dockerfile

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

Warum so?

- **Alles in einem `RUN`:** Sonst bleiben die Paketlisten in einem eigenen Layer.
- **`update` und `install` zusammen:** Sonst nutzt der Cache alte Paketlisten.
- **`--no-install-recommends`:** Keine empfohlenen Extras.
- **Aufräumen im selben `RUN`:** Sonst spart es nichts.
- **Eine Zeile pro Paket:** Diffs bleiben lesbar.

Für Alpine ist es kürzer:

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

STOLPERFALLE

`RUN apt-get update` in einer eigenen Zeile ist ein Klassiker. Der Layer kommt aus dem Cache, und Wochen später will `install` Pakete aus veralteten Listen laden. Das schlägt fehl oder installiert alte Versionen.

Heredocs

Mit BuildKit gehen längere Skripte lesbar:

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

Das Ganze bleibt ein einziger Layer.

STOLPERFALLE

`cd` hält nur für ein `RUN`. Jede Anweisung startet neu. `RUN cd /app` und danach `RUN make` läuft nicht in `/app`. Nutze `WORKDIR`.

STOLPERFALLE

Bei `RUN curl ... | sh` zählt nur der Exit-Code des letzten Befehls. Schlägt `curl` fehl, läuft der Build trotzdem weiter. Abhilfe bringt `pipefail`, siehe Kapitel 13.

IM INTERVIEW

Warum `apt-get update` und `install` im selben `RUN`?

Wegen des Caches. Steht `update` allein, kommt sein Layer aus dem Cache, während sich die `install`-Zeile ändert. Dann arbeitet `install` mit alten Paketlisten.

IM INTERVIEW

Warum nicht viele kleine `RUN`-Zeilen?

Jede erzeugt einen Layer, und Aufräumen im nächsten Layer spart nichts. Zusammengehörige Schritte kommen in ein `RUN`. Getrennt bleibt, was sich unterschiedlich oft ändert.

DAS NIMMST DU MIT

- Shell-Form mit `/bin/sh -c`, Exec-Form ohne Shell.
- Installieren und Aufräumen im selben `RUN`.
- Heredocs für längere Skripte.
- `cd` wirkt nur in einem `RUN`.

7

COPY und ADD

Dateien ins Image bringen. Fast immer mit `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
```

Die Regeln

- Ein Ziel mit `/` am Ende ist ein Ordner.
- Bei einem Ordner als Quelle wird der **Inhalt** kopiert, nicht der Ordner selbst.
- Mehrere Quellen brauchen einen Ordner als Ziel.
- Relative Ziele beziehen sich auf `WORKDIR`.

> **STOLPERFALLE**
>
> `COPY src/ /app/` legt nicht `/app/src` an. Der Inhalt von `src` landet direkt in `/app`. Willst du die Struktur behalten, schreibe `/app/src/`.

`ADD` kann mehr

Fähigkeit	`COPY`	`ADD`
Lokale Dateien	Ja	Ja
Aus einer anderen Stufe (`--from`)	Ja	Nein
HTTP-URL laden	Nein	Ja
Git-Repository laden	Nein	Ja
Lokales Tar auspacken	Nein	Ja, automatisch

> **STOLPERFALLE**
>
> Dass `ADD` lokale Tar-Archive auspackt, überrascht viele. Ein geladenes Archiv von einer URL packt es dagegen nicht aus. Nimm `COPY`, wenn du nichts davon brauchst: Es tut genau das, was draufsteht.

Lädst du mit `ADD` aus dem Netz, prüfe die Prüfsumme:

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

Moderne Schalter

Schalter	Wirkung
`--link`	Der Layer wird unabhängig von den Layern darunter gebaut. Besserer Cache.
`--parents`	Behält die Ordnerstruktur der Quelle bei.
`--exclude=muster`	Lässt Dateien beim Kopieren aus.

Der Schalter `--link` hilft besonders, wenn sich die Basis ändert: Der Layer muss nicht neu erzeugt werden.

IM INTERVIEW

`COPY` oder `ADD`?

`COPY`, solange es nur um Dateien aus dem Kontext geht, denn es ist vorhersehbar. `ADD` nur, wenn du gezielt ein lokales Tar auspacken oder eine URL mit Prüfsumme laden willst.

DAS NIMMST DU MIT

- Standard ist `COPY`.
- `ADD` packt lokale Tars aus und lädt URLs.
- `COPY --from` holt Dateien aus anderen Stufen.
- `--link` macht Layer unabhängiger.

8

CMD und ENTRYPOINT

Die Lieblingsfrage jedes Interviews.

Beide bestimmen, was beim Containerstart läuft.

- `ENTRYPOINT`: **Was** läuft. Das Programm.
- `CMD`: **Womit** es läuft. Die Standard-Argumente. Ohne `ENTRYPOINT` ist es der ganze Befehl.

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

Aufruf	Läuft im Container
`docker run bild`	`curl --help`
`docker run bild https://example.com`	`curl https://example.com`
`docker run --entrypoint sh bild`	`sh`

> **DAS BILD IM KOPF**
>
> *`ENTRYPOINT` ist die Kaffeemaschine, `CMD` die voreingestellte Taste „Espresso". Der Gast darf eine andere Taste drücken. Für eine andere Maschine braucht er einen Schraubenzieher: `--entrypoint`.*

Exec-Form und Shell-Form

```
CMD ["node", "server.js"]   # Exec-Form: gut
CMD node server.js          # Shell-Form: Vorsicht
```

Die Shell-Form startet `/bin/sh -c "node server.js"`. Dann ist die Shell PID 1, und deine Anwendung bekommt kein `SIGTERM`. Mehr dazu in Kapitel 27.

Wie beide zusammenspielen

ENTRYPOINT	CMD	Ergebnis
keins	`["a", "b"]`	`a b`
`["x"]`	keins	`x`
`["x"]`	`["a"]`	`x a`
`x` (Shell-Form)	beliebig	`/bin/sh -c x`. `CMD` wird ignoriert.

> **STOLPERFALLE**
>
> `ENTRYPOINT` in Shell-Form verschluckt alles. Weder `CMD` noch Argumente von `docker run` erreichen dein Programm. Nimm immer die Exec-Form mit eckigen Klammern und doppelten Anführungszeichen. Einfache Anführungszeichen sind kein gültiges JSON, und Docker fällt dann auf die Shell-Form zurück.

Das Entrypoint-Skript

Oft musst du vor dem Start etwas vorbereiten:

entrypoint.sh

```
#!/bin/sh
set -e
echo "Bereite Konfiguration vor"
exec "$@"
```

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

Das `exec "$@"` ist entscheidend. Es ersetzt die Shell durch dein Programm, das `CMD` als Argumente bekommt. Damit wird es PID 1 und empfängt Signale.

> **IM INTERVIEW**
>
> **Was ist der Unterschied zwischen `CMD` und `ENTRYPOINT`?**
>
> `ENTRYPOINT` legt das Programm fest, `CMD` die Standard-Argumente. Argumente bei `docker run` ersetzen `CMD`. `ENTRYPOINT` ersetzt man nur mit `--entrypoint`. Beides schreibe ich in Exec-Form.

IM INTERVIEW

Warum `exec "$@"` im Entrypoint-Skript?

Ohne `exec` bleibt die Shell PID 1 und startet das Programm als Kind. Signale wie `SIGTERM` kommen dann nicht an. `exec` ersetzt die Shell durch das Programm.

DAS NIMMST DU MIT

- `ENTRYPOINT`: Programm. `CMD`: Standard-Argumente.
- Immer Exec-Form mit `["..."]`.
- Entrypoint-Skripte enden mit `exec "$@"`.

9

ENV und ARG

Eine Variable für die Laufzeit, eine für die Bauzeit.

	`ARG`	`ENV`
Gilt beim Bauen	Ja, ab der Deklaration im Stage	Ja, ab der Anweisung
Gilt im Container	Nein	Ja
Setzen	`--build-arg` bei `docker build`	`-e` bei `docker run`

```
ARG VERSION=1.0.0
ENV APP_ENV=production APP_DIR=/app
RUN echo "Baue $VERSION für $APP_ENV"
```

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

Eselsbrücke: Der Bauherr nennt die Version beim Bauen.

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

Eselsbrücke: `-e` wie **e**nvironment: überschreibt `ENV` beim Start.

> **DAS BILD IM KOPF**
>
> *`ARG` ist der Notizzettel des Handwerkers: Nach dem Bau ist er weg. `ENV` ist die Hausordnung: Sie hängt für immer im Flur.*

Ein häufiges Muster

Wert zur Bauzeit festlegen und zur Laufzeit behalten:

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

> **STOLPERFALLE**
>
> Geheimnisse gehören in keine von beiden. `ENV`-Werte stehen in den Metadaten, `ARG`-Werte in der Build-Historie. `docker history` und `docker inspect` zeigen sie. Für Geheimnisse gibt es Kapitel 17.

MERKE

Ändert sich ein `--build-arg`, wird der Cache ab der ersten Stelle ungültig, die ihn nutzt. Darum steht das Beispiel `ARG BUILD_DATE` möglichst spät im Dockerfile, sonst baust du jedes Mal alles neu.

Ein echter Fehler

Dieses Muster findet man leider oft:

```
ENV POSTGRES_PASSWORD=geheim
```

Das Passwort steckt für immer im Image. Richtig ist, es beim Start aus einem Secret zu übergeben.

IM INTERVIEW

Was ist der Unterschied zwischen `ENV` und `ARG`?

`ARG` gibt es nur beim Bauen, `ENV` auch im laufenden Container. Beide sind ungeeignet für Geheimnisse, weil sie in der Historie oder den Metadaten des Images landen.

DAS NIMMST DU MIT

- `ARG`: Bauzeit. `ENV`: Bauzeit und Laufzeit.
- Beide sind kein Ort für Geheimnisse.
- Geänderte `ARG`s machen den Cache ab ihrer Nutzung ungültig.

10

WORKDIR und USER

Wo läuft es, und als wer?

WORKDIR

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

`WORKDIR` setzt das aktuelle Verzeichnis für alles Folgende: `RUN`, `COPY`, `CMD`. Fehlt der Ordner, legt Docker ihn an. Relative Pfade bauen auf dem vorherigen `WORKDIR` auf.

> **MERKE**
>
> Nutze absolute Pfade. Ein relativer `WORKDIR` hängt vom Basis-Image ab und kann sich mit dessen nächster Version ändern.

USER

Ohne `USER` läuft alles als `root`.

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

`USER` wirkt auf `RUN`, `CMD` und `ENTRYPOINT`, die danach kommen.

> **STOLPERFALLE**
>
> `COPY` kümmert sich nicht um `USER`. Kopierte Dateien gehören standardmäßig `root`. Soll die Anwendung dort schreiben, nutze `--chown` oder ändere die Rechte vorher.

> **STOLPERFALLE**
>
> `USER app` mit Namen statt Nummer scheitert in Kubernetes an `runAsNonRoot`, denn das Kubelet kann den Namen nicht prüfen. Die Nummer `USER 10001` funktioniert.

IM INTERVIEW

Warum sollte ein Container nicht als `root` laufen?

Bricht jemand aus der Anwendung aus, ist er im Container `root`. Das erleichtert Ausbrüche auf den Host. Mit einem eigenen Nutzer ohne Rechte sinkt der Schaden. Details in Kapitel 26.

DAS NIMMST DU MIT

- `WORKDIR` statt `cd`, immer absolut.
- `USER` mit Nummer setzen.
- `COPY` ignoriert `USER`, also `--chown`.

11

EXPOSE, VOLUME, LABEL

Drei Anweisungen, die weniger tun, als viele glauben.

EXPOSE

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

`EXPOSE` ist Dokumentation. Es öffnet keinen Port. Veröffentlichen musst du ihn beim Start mit `-p 8080:8080`. `-P` veröffentlicht alle `EXPOSE`-Ports auf zufälligen Host-Ports.

VOLUME

```
VOLUME /var/lib/daten
```

`VOLUME` markiert einen Pfad als Ort für beständige Daten. Beim Start legt Docker dort ein anonymes Volume an, wenn du nichts anderes mountest.

> **STOLPERFALLE**
>
> Alles, was ein späteres `RUN` in einen `VOLUME`-Pfad schreibt, geht verloren. Bereite den Inhalt vorher vor. Viele verzichten ganz auf `VOLUME` und mounten beim Start ausdrücklich, damit keine vergessenen anonymen Volumes entstehen.

LABEL

Labels sind Metadaten als Schlüssel und Wert. Es gibt Standardnamen:

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

> **MERKE**
>
> `MAINTAINER` ist veraltet. Nutze `LABEL org.opencontainers.image.authors`.

STOPSIGNAL

```
STOPSIGNAL SIGQUIT
```

Legt das Signal fest, das Docker beim Stoppen schickt. Standard ist `SIGTERM`. Das Beispiel passt zu nginx, das bei `SIGQUIT` sanft beendet.

IM INTERVIEW

Öffnet `EXPOSE` einen Port?

Nein. Es dokumentiert nur, auf welchem Port die Anwendung lauscht. Den Port machst du mit `-p` beim Start erreichbar.

DAS NIMMST DU MIT

- `EXPOSE` dokumentiert, `-p` veröffentlicht.
- Nach `VOLUME` geschriebene Daten sind verloren.
- `LABEL` mit `org.opencontainers.image.*`.

12

HEALTHCHECK

Läuft der Prozess, oder funktioniert er auch?

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

Docker führt den Befehl regelmäßig im Container aus. Exit-Code 0 heißt `healthy`, 1 heißt `unhealthy`.

Option	Standard	Bedeutung
`--interval`	30 s	Abstand der Prüfungen.
`--timeout`	30 s	Länger gilt als fehlgeschlagen.
`--start-period`	0 s	Schonfrist beim Start. Fehler zählen nicht.
`--start-interval`	5 s	Abstand während der Schonfrist.
`--retries`	3	So viele Fehlschläge in Folge bis `unhealthy`.

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

Eselsbrücke: Der Arztbericht des Containers: Status und die letzten Prüfungen.

Mit `HEALTHCHECK NONE` schaltest du eine geerbte Prüfung ab.

> **DAS BILD IM KOPF**
>
> *Ein laufender Prozess ist wie ein Licht im Büro. Es brennt, aber das sagt nichts darüber, ob jemand arbeitet. Der Healthcheck ruft an und erwartet eine Antwort.*

> **STOLPERFALLE**
>
> `curl` steckt nicht in jedem Image, in distroless schon gar nicht. Dort brauchst du ein kleines Prüfprogramm oder einen Unterbefehl deiner Anwendung, etwa `CMD ["/app", "healthcheck"]`.

STOLPERFALLE

Kubernetes ignoriert `HEALTHCHECK`. Dort gelten Liveness-, Readiness- und Startup-Probes. Die Anweisung hilft bei Docker und Compose.

IM INTERVIEW

Wozu dient `HEALTHCHECK`, wenn der Prozess doch läuft?

Ein Prozess kann laufen und trotzdem hängen. Der Healthcheck prüft die echte Funktion. Docker meldet den Zustand, und Compose kann mit `depends_on` darauf warten.

DAS NIMMST DU MIT

- 0 ist `healthy`, 1 ist `unhealthy`.
- `--start-period` gibt Zeit zum Starten.
- Kubernetes nutzt Probes, nicht `HEALTHCHECK`.

13

SHELL, ONBUILD, Direktiven

Die seltenen Werkzeuge, nach denen gern gefragt wird.

SHELL

`SHELL` ändert die Shell für die Shell-Form von `RUN`, `CMD` und `ENTRYPOINT`. Der wichtigste Anwendungsfall ist `pipefail`:

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

Jetzt bricht der Build ab, wenn `curl` fehlschlägt. Ohne `pipefail` zählt nur das letzte Glied der Pipe.

> **MERKE**
>
> Ob die Standard-`sh` `pipefail` kennt, hängt vom Image ab. Darum steht hier ausdrücklich `bash`, das im Image installiert sein muss.

ONBUILD

`ONBUILD` legt eine Anweisung für später ab. Sie läuft erst, wenn jemand mit `FROM` auf dieses Image aufbaut.

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

> **DAS BILD IM KOPF**
>
> *Ein Zettel im Bausatz: „Wer das hier aufbaut, muss zuerst Schritt A erledigen.“ Der Hersteller selbst macht es nicht.*

Das ist für firmeneigene Basis-Images gedacht. Es versteckt allerdings Schritte. Viele meiden es deshalb.

Parser-Direktiven

Direktiven stehen ganz oben im Dockerfile, als spezielle Kommentare.

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

Direktive	Wirkung
`syntax`	Wählt die Dockerfile-Version. Neue Funktionen ohne Docker-Update.
`escape`	Ändert das Fluchtzeichen, nützlich unter Windows.
`check`	Steuert Build Checks, siehe Kapitel 31.

IM INTERVIEW

Was macht `# syntax=docker/dockerfile:1`?

Es legt das Frontend fest, das BuildKit lädt, um das Dockerfile zu verstehen. So bekommst du neue Funktionen wie aktuelle Mount-Optionen, ohne Docker zu aktualisieren.

DAS NIMMST DU MIT

- `SHELL` für `pipefail`.
- `ONBUILD` läuft erst im Kind-Image.
- Direktiven stehen ganz oben.

TEIL III

Besser bauen

Cache, BuildKit, Multi-Stage, Secrets, Multi-Arch und kleine Images.

14

Der Build-Cache

Der Unterschied zwischen zehn Sekunden und zehn Minuten.

Docker prüft jede Anweisung von oben nach unten: Gibt es dafür schon einen Layer? Wenn ja, nimmt es ihn. Sobald eine Anweisung neu gebaut wird, sind in der Regel alle folgenden auch neu.

Ein Cache-Miss oben baut alles darunter neu

> **DAS BILD IM KOPF**
>
> *Dominosteine. Stößt du einen an, fallen alle dahinter. Darum gehört alles, was sich oft ändert, ganz nach hinten.*

Wann ist der Cache gültig?

Anweisung	Geprüft wird
`RUN`	Nur der Text der Anweisung. Nicht das Ergebnis!
`COPY`, `ADD`	Prüfsumme des Dateiinhalts. Die Uhrzeit zählt nicht.
`ENV`, `ARG`	Der Wert.

> **STOLPERFALLE**
>
> `RUN apt-get install curl` und `RUN git clone ...` sehen für den Cache gleich aus, egal was sich im Netz geändert hat. Docker holt nichts neu, nur weil es Neues gibt.

Die wichtigste Regel: Reihenfolge

Dockerfile (schlecht)

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

Jede Code-Änderung macht `COPY . .` ungültig, also läuft auch `npm ci` neu.

Dockerfile (gut)

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

Jetzt läuft `npm ci` nur noch, wenn sich die Abhängigkeiten ändern.

> **MERKE**
>
> Die Regel lautet: von selten geändert nach oft geändert. Erst das System, dann die Abhängigkeiten, dann dein Code.

Cache umgehen

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

Eselsbrücke: Alles neu, kein Layer aus dem Cache.

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

Eselsbrücke: Prüfe, ob es eine neuere Version der Basis gibt.

Cache Mounts

Ein Cache Mount ist ein Ordner, der zwischen Builds bestehen bleibt, ohne im Image zu landen. Ideal für Paketmanager:

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

Ändert sich `package-lock.json`, läuft `npm ci` neu, aber die Downloads sind noch da.

> **STOLPERFALLE**
>
> Cache Mounts liegen lokal beim Builder. `--cache-to` exportiert sie nicht. Auf einem frischen CI-Runner sind sie leer. Sie sind eine Beschleunigung, keine Abhängigkeit: Der Build muss auch ohne sie funktionieren.

Cache in CI

Ein frischer Runner kennt keine alten Layer. Du lädst den Cache aus einem 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` speichert auch die Layer der Zwischenstufen. Weitere Backends sind `gha`, `local` und `s3`.

IM INTERVIEW

Wie optimierst du ein Dockerfile für den Cache?

Selten Geändertes nach oben, oft Geändertes nach unten. Erst die Lockfiles kopieren und Abhängigkeiten installieren, dann den Code. Dazu eine gute `.dockerignore`, in CI einen exportierten Cache.

IM INTERVIEW

Wie erkennt Docker, ob sich ein `COPY` geändert hat?

Über eine Prüfsumme des Dateiinhalts und der Rechte. Die Änderungszeit spielt keine Rolle. Bei `RUN` vergleicht es dagegen nur den Text der Anweisung.

DAS NIMMST DU MIT

- Ein Cache-Miss entwertet alles darunter.
- `RUN` wird am Text erkannt, `COPY` am Inhalt.
- Lockfile zuerst, Code zuletzt.
- Cache Mounts für Paketmanager, `--cache-to` für CI.

15

BuildKit

Der Motor hinter `docker build`.

BuildKit ist der Build-Motor von Docker. Seit Docker Engine 23.0 ist er Standard. **Buildx** ist der CLI-Teil dazu. Heute läuft `docker build` also schon über BuildKit.

Was BuildKit anders macht

- **Graph statt Liste:** Es plant den ganzen Build als Abhängigkeitsgraph.
- **Parallel:** Unabhängige Stufen laufen gleichzeitig.
- **Sparsam:** Stufen, die niemand braucht, werden übersprungen.
- **Gezielte Kontexte:** Es überträgt nur Dateien, die wirklich benötigt werden.
- **Neue Funktionen:** Secrets, SSH, Cache Mounts, Heredocs, Multi-Arch.

> **DAS BILD IM KOPF**
>
> *Der alte Builder arbeitet wie jemand, der ein Rezept stur Zeile für Zeile abkocht. BuildKit liest zuerst das ganze Rezept und kocht auf drei Herdplatten zugleich.*

Ausgabe lesen

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

Eselsbrücke: Volle Ausgabe jeder Zeile, statt des zusammengefalteten Fortschritts. Gut zum Debuggen.

```
docker buildx ls
```

Eselsbrücke: Welche Builder gibt es, und was können sie?

```
docker buildx du
```

Eselsbrücke: **du** wie auf Linux: Wie viel Platz belegt der Build-Cache?

```
docker builder prune
```

Eselsbrücke: Räume den Cache auf. Mit `-a` auch alles Ungenutzte.

Die Treiber

Treiber	Wofür
`docker`	Standard, im Docker-Daemon.
`docker-container`	Eigener Builder-Container. Für Multi-Arch und mehr Cache-Backends.
`kubernetes`	Builder als Pods im Cluster.
`remote`	Verbindung zu einem bestehenden BuildKit-Daemon.

Mehr Kontrolle in `RUN`

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

Dieser Schritt hat kein Netz. So fällt auf, wenn ein Test heimlich etwas herunterlädt.

Mehrere Kontexte

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

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

Damit bekommst du Dateien außerhalb des Hauptkontexts, aber ausdrücklich und sichtbar.

IM INTERVIEW

Was bringt BuildKit gegenüber dem alten Builder?

Paralleles Bauen über einen Abhängigkeitsgraphen, das Überspringen unnötiger Stufen, besseres Caching und neue Funktionen wie Secrets, SSH-Mounts und Cache Mounts.

DAS NIMMST DU MIT

- BuildKit ist Standard, Buildx das Kommando.
- `--progress=plain` zeigt alles.
- `docker-container`-Treiber für Multi-Arch.
- `docker buildx du` und `builder prune` gegen volle Platten.

16

Multi-Stage-Builds

Die wichtigste Technik für kleine Images.

Bauwerkzeuge gehören nicht ins fertige Image. Ein Multi-Stage-Build nutzt mehrere `FROM`. Nur die letzte Stufe wird zum Image. Den Rest kopierst du gezielt herüber.

Dockerfile

```
# Stufe 1: bauen
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 .

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

Das Go-Image ist groß. Das Ergebnis enthält nur noch die Basis und deine Binärdatei.

> **DAS BILD IM KOPF**
>
> *Die Werkstatt und der Laden. In der Werkstatt liegen Sägen und Späne. In den Laden trägst du nur den fertigen Stuhl.*

Nützliche Tricks

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

Eselsbrücke: `--target`: Baue nur bis zu dieser Stufe. Das Ziel eignet sich für Tests oder Entwicklung.

```
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
```

Eine Stufe kann auf einer anderen aufbauen. Weil BuildKit unnötige Stufen überspringt, läuft `test` nur, wenn du sie als Ziel wählst.

> **STOLPERFALLE**
>
> Eine Stufe, von der die Zielstufe nicht abhängt, baut BuildKit nicht. Wer glaubt, die Tests seien gelaufen, weil sie im Dockerfile stehen, irrt sich. Baue sie mit `--target test` in CI ausdrücklich.

> **MERKE**
>
> `COPY --from` geht auch von einem fremden Image: `COPY --from=nginx:alpine /etc/nginx/nginx.conf .`

Was das bringt

- **Kleiner:** Kein Compiler, kein Quellcode, keine Dev-Abhängigkeiten.
- **Sicherer:** Weniger Pakete, weniger Schwachstellen.
- **Sauber:** Zwischendaten der Bau-Stufen landen nicht im Endergebnis.

> **STOLPERFALLE**
>
> Sicher vor Geheimnissen ist auch die Bau-Stufe nicht von selbst. Ein Token in einem Layer der Bau-Stufe liegt weiter im Build-Cache. Nutze Secret Mounts, siehe Kapitel 17.

> **IM INTERVIEW**
>
> **Was ist ein Multi-Stage-Build, und wozu dient er?**
>
> Mehrere `FROM` in einem Dockerfile. Frühe Stufen bauen, die letzte enthält nur das Ergebnis, das du per `COPY --from` holst. So bleiben Compiler und Quellcode draußen, und das Image wird klein und sicher.

IM INTERVIEW

Was macht `--target`?

Es beendet den Build bei einer bestimmten Stufe. So baust du aus einem Dockerfile eine Test-, Entwicklungs- oder Produktionsvariante.

DAS NIMMST DU MIT

- Nur die letzte Stufe wird zum Image.
- `COPY --from=stufe` holt das Ergebnis.
- `--target` wählt eine Stufe.
- Ungenutzte Stufen laufen nicht.

17

Secrets und SSH

Geheimnisse zum Bauen, die nicht im Image landen.

Ein Build braucht manchmal ein Token, etwa für ein privates Paket. Dieses Token darf weder im Layer noch in der Historie stehen.

Was nicht funktioniert

Versuch	Problem
`ENV TOKEN=...`	Steht in den Metadaten.
`ARG TOKEN`	Steht in der Historie.
`COPY .npmrc` und danach `rm`	Liegt weiter im 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 .
```

Eselsbrücke: Du gibst dem Builder das Geheimnis nur für diesen einen `RUN`.

Das Geheimnis liegt während dieses `RUN` als Datei vor, standardmäßig unter `/run/secrets/`, wenn du kein Ziel nennst. Danach ist es weg, und es landet nicht im Layer.

> **DAS BILD IM KOPF**
>
> *Der Handwerker bekommt den Schlüssel nur für diesen einen Raum und gibt ihn danach zurück. Im fertigen Haus steckt er nirgendwo.*

Als Umgebungsvariable geht es auch:

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

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

Eselsbrücke: Ohne `src` liest Docker den Wert aus der gleichnamigen Umgebungsvariable deiner Shell.

STOLPERFALLE

Das Ergebnis eines Schritts kann das Geheimnis trotzdem enthalten. Schreibt ein Werkzeug das Token in eine Datei oder ins Log, ist es gespeichert. Prüfe, was der Befehl hinterlässt.

SSH-Mounts

Für private Git-Repositories gibst du deinen SSH-Agent weiter, statt einen Schlüssel zu kopieren.

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/intern.git /
src
```

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

Eselsbrücke: `default` ist dein laufender SSH-Agent. Der Schlüssel selbst verlässt ihn nie.

MERKE

Für den Betrieb sind Build-Secrets die falsche Stelle. Laufzeit-Geheimnisse kommen zum Start aus Vault, Kubernetes Secrets oder Docker Secrets.

IM INTERVIEW

Wie nutzt du ein Token im Build, ohne dass es im Image landet?

Mit einem BuildKit-Secret-Mount: `--secret` beim Build, `RUN --mount=type=secret` im Dockerfile. Das Geheimnis steht nur während dieses Schritts bereit und landet weder in Layer noch Historie.

IM INTERVIEW

Reicht ein Multi-Stage-Build zum Schutz von Geheimnissen?

Nur begrenzt. Das Endimage ist sauber, aber die Bau-Layer können im Cache oder in der Registry des Caches liegen. Sicher ist nur, wenn das Geheimnis nie in einen Layer kommt.

DAS NIMMST DU MIT

- Nie über `ENV`, `ARG` oder `COPY`.
- `--mount=type=secret` für Tokens.
- `--mount=type=ssh` für private Repositories.
- Laufzeit-Geheimnisse kommen beim Start.

18

Multi-Arch mit buildx

Ein Name, viele Architekturen.

Macs und viele Cloud-Server laufen auf `arm64`, klassische Server auf `amd64`. Ein **Multi-Arch-Image** ist ein **Image Index**: eine Liste, die pro Plattform auf ein eigenes Image zeigt.

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

Der Client wählt beim Pull automatisch die passende Variante.

Builder anlegen

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

Eselsbrücke: Ein eigener Builder-Container, der mehrere Plattformen kann.

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

Eselsbrücke: Baue beide Varianten und schiebe sie gemeinsam mit dem Index zur Registry.

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

Eselsbrücke: Schau in den Index: Welche Plattformen sind drin?

Drei Wege zur fremden Architektur

Weg	Wie	Tempo
QEMU	Emuliert die fremde CPU.	Langsam
Nativ	Ein Builder pro Architektur.	Schnell, aber teuer
Cross-Compile	Der Compiler erzeugt fremden Code.	Schnell

QEMU richtest du mit diesem Befehl ein:

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

Eselsbrücke: Bringt dem Kernel bei, fremde Binärdateien per Emulator zu starten. Braucht `--privileged`, also nur auf einer vertrauenswürdigen Maschine.

Cross-Compile im Dockerfile

Der Compiler soll nativ laufen, das Ergebnis aber für das Ziel passen:

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	Bedeutung
`BUILDPLATFORM`	Plattform, auf der gebaut wird.
`TARGETPLATFORM`	Plattform, für die gebaut wird.
`TARGETOS`, `TARGETARCH`	Die Einzelteile des Ziels.

DAS BILD IM KOPF

Ein Dolmetscher (QEMU) übersetzt jeden Satz live: Das klappt, ist aber zäh.

Ein Cross-Compile ist ein Buch, das du gleich in die Zielsprache schreibst.

STOLPERFALLE

Ein `arm64`-Image auf `amd64` ohne Emulator endet mit `exec format error`. Das ist fast immer eine falsche Architektur oder ein Skript ohne Shebang.

MERKE

Mehrere Ziele und Varianten beschreibst du mit `docker buildx bake` in einer Datei `docker-bake.hcl`. Das ersetzt lange Kommandozeilen in CI.

IM INTERVIEW

Wie baust du ein Image für `amd64` und `arm64`?

Mit `docker buildx build --platform linux/amd64,linux/arm64 --push`. Das erzeugt einen Image Index. Für Tempo kompiliere ich per Cross-Compile mit `--platform=$BUILDPLATFORM`, statt QEMU zu nutzen.

IM INTERVIEW

Was ist ein Image Index?

Eine Liste von Images pro Plattform unter einem Tag. Der Client wählt beim Pull das passende Manifest.

DAS NIMMST DU MIT

- Der Index zeigt pro Plattform auf ein Image.
- `buildx build --platform ... --push`.
- QEMU ist langsam, Cross-Compile schnell.
- `exec format error`: Architektur prüfen.

19

Kleine Images

Weniger Größe heißt weniger Pull-Zeit und weniger Angriffsfläche.

Warum die Größe zählt

- **Schneller:** Pull, Push und Skalierung im Cluster.
- **Sicherer:** Jedes Paket weniger ist eine mögliche Schwachstelle weniger.
- **Günstiger:** Weniger Speicher und Transfer.

Die Hebel, nach Wirkung sortiert

Hebel	Wie
Multi-Stage	Compiler und Quellcode bleiben draußen.
Kleine Basis	`slim`, distroless oder `scratch`.
Nur Laufzeit-Abhängigkeiten	`npm ci --omit=dev`, keine Test-Pakete.
Saubere `.dockerignore`	Keine versehentlichen Dateien.
Aufräumen im selben `RUN`	Paket-Caches nicht in Layern lassen.
Keine Empfehlungen	`--no-install-recommends`, `apk --no-cache`.

Der Ablauf

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

Eselsbrücke: Welcher Schritt ist schuld?

```
dive app:1.0
```

Eselsbrücke: Welche Dateien sind es, und was ist doppelt?

```
docker system df
```

Eselsbrücke: **df** wie auf Linux: Wie viel Platz belegen Images, Container und Cache?

STOLPERFALLE

Das Ziel ist kein kleinstes Image um jeden Preis. Ein Image mit Alpine, das wegen musl im Betrieb merkwürdig läuft, ist schlechter als ein etwas größeres `slim`-Image. Erst Korrektheit und Sicherheit, dann Größe.

MERKE

Die Registry speichert Layer komprimiert. Das `docker image ls` zeigt den ausgepackten Wert. Zahlen aus der Registry sind darum kleiner.

IM INTERVIEW

Wie machst du ein Image kleiner?

Ich messe mit `docker history` und `dive`. Dann Multi-Stage, eine kleine Basis, nur Laufzeit-Abhängigkeiten und Aufräumen im selben `RUN`.

DAS NIMMST DU MIT

- Multi-Stage ist der größte Hebel.
- Erst messen, dann kürzen.
- Kleiner heißt auch sicherer, aber Korrektheit geht vor.

TEIL IV

Basis und Sprachen

Die richtige Grundlage und fertige Muster für Node, Python, Go und Java.

20

Basis-Images wählen

Eine Entscheidung, die alles Weitere prägt.

Basis	C-Bibliothek	Shell	Passt für
`debian`, `ubuntu`	glibc	Ja	Bauen, Werkzeuge, Debuggen.
`*-slim`	glibc	Ja	Guter Standard für die Laufzeit.
`alpine`	musl	Ja	Sehr klein, wenn alles mit musl läuft.
distroless	glibc oder keine	Nein	Produktion mit geringer Angriffsfläche.
`scratch`	keine	Nein	Statische Binärdateien.

> **DAS BILD IM KOPF**
>
> *`debian` ist die möblierte Wohnung, `slim` die Wohnung mit Bett und Tisch, `alpine` das winzige Apartment mit eigenen Maßen. Distroless ist das Hotelzimmer ohne Werkzeugkasten. `scratch` ist der leere Raum.*

Alpine und musl

Alpine nutzt **musl** statt **glibc** als C-Bibliothek und BusyBox statt GNU-Werkzeugen. Das macht es klein, bringt aber Unterschiede:

- Für glibc gebaute Binärdateien laufen nicht. Die Fehlermeldung ist oft ein irreführendes `not found`.
- Python-Pakete haben nicht immer fertige `musllinux`-Wheels. Dann wird lange aus den Quellen kompiliert.
- Speicherverwaltung und DNS verhalten sich anders als bei glibc.

> **STOLPERFALLE**
>
> „Alpine ist immer besser, weil kleiner“ ist falsch. Für Python und Java ist `slim` oft die bessere Wahl. Für Go mit `CGO_ENABLED=0` spielt es keine Rolle, weil die Binärdatei statisch ist.

Distroless

Distroless-Images enthalten nur die Laufzeit und ihre Abhängigkeiten: keine Shell, keinen Paketmanager. Das reduziert die Angriffsfläche deutlich.

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

Die Tags verraten etwas: `:nonroot` läuft ohne Root-Rechte, `:debug` enthält eine BusyBox-Shell für die Fehlersuche.

> **STOLPERFALLE**
>
> Ohne Shell funktionieren die Shell-Form, `$VAR` in `CMD` und Entrypoint-Skripte nicht mehr. Es gibt auch kein `docker exec ... sh`. Für die Fehlersuche nutzt du das `:debug`-Tag, `docker debug` oder in Kubernetes `kubectl debug`.

Eine Entscheidungshilfe

1. Statische Binärdatei? `scratch` oder distroless `static`.
2. Node, Python, Java? `slim` oder distroless, wenn es zur Anwendung passt.
3. Winziges Tool, dessen musl-Verhalten du kennst? `alpine`.
4. Nur für die Bau-Stufe? Das normale große Image. Dort stört die Größe nicht.

> **MERKE**
>
> Nutze offizielle Images oder eine vertrauenswürdige Quelle. Viele Firmen spiegeln Basis-Images in eine eigene Registry und pflegen sie zentral. Gehärtete Angebote von Anbietern gibt es ebenfalls.

IM INTERVIEW

Alpine oder slim?

Für die meisten Laufzeiten `slim`, weil glibc weniger Überraschungen bringt. Alpine, wenn die Anwendung sicher mit musl läuft und die Größe wirklich zählt. Für statische Binärdateien gleich distroless oder `scratch`.

IM INTERVIEW

Was ist distroless?

Images ohne Shell und Paketmanager, nur mit der Laufzeit. Weniger Pakete bedeuten weniger Schwachstellen, und ein Angreifer findet keine Werkzeuge. Der Preis ist eine aufwendigere Fehlersuche.

DAS NIMMST DU MIT

- `slim` ist der gute Standard.
- musl bricht manche glibc-Annahmen.
- distroless hat keine Shell, also keine Shell-Form.
- Die Bau-Stufe darf groß sein.

21

Node.js

Abhängigkeiten trennen, und bitte nicht mit `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"]
```

Das passiert hier:

- `deps` erzeugt nur die Produktions-Abhängigkeiten.
- `build` installiert alles und baut den Code.
- Die letzte Stufe nimmt beide Ergebnisse, ohne Dev-Pakete.
- `USER node` nutzt den Nutzer, den das offizielle Image mitbringt.

Die Regeln

- `npm ci` statt `npm install`: Es hält sich exakt an das Lockfile und bricht bei Abweichungen ab.
- `--omit=dev` lässt Dev-Pakete weg.
- `node_modules` steht in `.dockerignore`. Native Module vom Mac laufen nicht unter Linux.

- `NODE_ENV=production` setzen. Viele Bibliotheken verhalten sich danach anders.

> **STOLPERFALLE**
>
> `CMD ["npm", "start"]` ist eine Falle. `npm` steht dann zwischen Docker und deiner Anwendung und reicht Signale nicht sauber weiter. Starte direkt mit `node`.

Sauber beenden

Als PID 1 muss Node `SIGTERM` selbst behandeln:

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

Mehr dazu in Kapitel 27.

> **IM INTERVIEW**
>
> **Warum `npm ci` statt `npm install` im Dockerfile?**
>
> `npm ci` installiert exakt das Lockfile, löscht vorher `node_modules` und schlägt bei Abweichungen fehl. Das macht Builds reproduzierbar.

DAS NIMMST DU MIT

- Lockfile zuerst kopieren, `npm ci` nutzen.
- Drei Stufen: `deps`, `build`, Laufzeit.
- Direkt `node` starten, nicht `npm`.

22

Python

Ein virtuelles Environment, das umziehen kann.

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"]
```

Das passiert hier:

- Das `venv` liegt an einem festen Pfad, damit es in die letzte Stufe umziehen kann.
- In beiden Stufen steht dieselbe Python-Basis. Sonst passen kompilierte Pakete nicht.
- `PYTHONUNBUFFERED=1` schreibt Logs sofort, statt sie zu puffern.
- `--no-cache-dir` lässt den pip-Cache weg.

> **STOLPERFALLE**
>
> `requirements.txt` mit `requests` ohne Version ist nicht reproduzierbar. Nutze ein gepinntes Lockfile, etwa mit `pip-compile`, Poetry oder `uv.lock`.

Mit uv

`uv` ist ein schneller Paketmanager mit 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` nimmt das Lockfile unverändert, und `--no-dev` lässt Entwicklungspakete weg. `--no-install-project` installiert nur die Abhängigkeiten, denn der Code ist noch nicht im Image. So bleibt dieser Layer im Cache. `--locked` bricht ab, wenn das Lockfile nicht mehr passt.

STOLPERFALLE

Native Pakete brauchen zum Bauen `gcc` und Header. Installiere sie nur in der Bau-Stufe. Laufzeit-Bibliotheken wie `libpq` gehören dagegen in die letzte Stufe. Fehlen sie, gibt es erst beim Start einen `ImportError`.

IM INTERVIEW

Warum kopierst du ein `venv` zwischen Stufen?

Compiler und Header bleiben in der Bau-Stufe, während die installierten Pakete an einem Pfad mitziehen. Beide Stufen brauchen dieselbe Basis, damit die Binär-Erweiterungen passen.

DAS NIMMST DU MIT

- `venv` an festem Pfad, in die letzte Stufe kopieren.
- `PYTHONUNBUFFERED=1` für Logs.
- Lockfile, keine losen Versionen.
- Bauwerkzeuge in die Bau-Stufe, Laufzeit-Bibliotheken ins Endimage.

23

Go

Eine Binärdatei, fast kein Image.

Go erzeugt eine einzelne Datei. Ist sie statisch gelinkt, braucht sie fast nichts.

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"]
```

Schalter	Wirkung
`CGO_ENABLED=0`	Statische Binärdatei ohne C-Bibliothek.
`-trimpath`	Keine lokalen Pfade in der Datei.
`-ldflags="-s -w"`	Symboltabelle und Debug-Infos weglassen. Kleiner.

Mit `scratch`

`scratch` enthält nichts, also auch keine CA-Zertifikate. Ohne sie schlagen HTTPS-Aufrufe fehl.

```
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` bringt Zertifikate, Zeitzonen und einen Nutzer `nonroot` schon mit. Das ist der bequemere Weg.

STOLPERFALLE

Ohne `CGO_ENABLED=0` kann die Binärdatei dynamisch gegen glibc gelinkt sein, etwa bei Paketen wie `net`. In `scratch` meldet der Start dann `no such file or directory`, obwohl die Datei da ist.

IM INTERVIEW

Warum ist Go so gut für kleine Images?

Es baut eine statische Binärdatei. Mit `CGO_ENABLED=0` und Multi-Stage kommt ein Image mit wenigen Megabyte heraus, ohne Compiler und Bibliotheken.

DAS NIMMST DU MIT

- `CGO_ENABLED=0` für statisches Linken.
- `go.mod` und `go.sum` zuerst kopieren.
- `scratch` braucht CA-Zertifikate.
- Distroless `static` ist bequemer.

24

Java

JRE statt JDK, und Layer für den 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` steht vor `COPY src`, damit der Dependency-Download im Cache bleibt.
- Das Endimage enthält nur die JRE.
- `MaxRAMPercentage` richtet den Heap nach dem Container-Limit aus.

> **MERKE**
>
> Moderne JVMs erkennen Container-Limits. Ohne Einstellung nimmt der Heap nur ein Viertel des Speichers. Darum setzt man `MaxRAMPercentage`, etwa auf 75.

Layered JARs mit Spring Boot

Eine einzige große JAR ändert sich bei jedem Build ganz. Spring Boot kann sie in Layer zerlegen. Die Stufen unten ersetzen die letzte Stufe des Dockerfiles von oben. Die JAR kommt also aus der Stufe `build`. Sie braucht den Layer-Index, den die Spring-Boot-Plugins standardmäßig erzeugen.

```
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"]
```

Das `app.jar` am Ende ist nicht mehr die große JAR. Sie enthält nur noch den Anwendungscode und verweist auf die entpackten Bibliotheken. Bei neuem Code ändert sich darum nur der kleine letzte Layer, und Pushes werden deutlich kürzer.

Eigene JRE mit jlink

`jlink` baut eine Laufzeit nur aus den Modulen, die deine Anwendung braucht. `jdeps` findet sie.

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

> **STOLPERFALLE**
>
> `ENTRYPOINT java -jar app.jar` in Shell-Form bedeutet: Die JVM bekommt kein `SIGTERM`, und das geordnete Herunterfahren entfällt.

> **IM INTERVIEW**
>
> **Wie optimierst du ein Java-Image?**
>
> Multi-Stage mit JDK zum Bauen und JRE zur Laufzeit, Layered JARs für den Cache, optional `jlink`. Und `MaxRAMPercentage`, damit der Heap zum Container-Limit passt.

DAS NIMMST DU MIT

- JDK zum Bauen, JRE zum Laufen.
- Layered JARs machen Pushes klein.
- `MaxRAMPercentage` statt fester Heap-Größe.

25

Statische Seiten mit nginx

Eine Single-Page-App ohne Node zur Laufzeit.

Ein Frontend braucht Node nur zum Bauen. Ausgeliefert wird es von nginx.

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` schickt unbekannte Pfade an `index.html`, damit der Client-Router übernimmt.

> **MERKE**
>
> Das Image `nginx-unprivileged` läuft ohne Root-Rechte und lauscht darum auf 8080 statt 80. Das Tag `stable-alpine` dient hier nur als Beispiel. In Produktion pinnst du es.

Warum `daemon off`?

nginx wechselt normalerweise in den Hintergrund. Im Container endet damit PID 1, und der Container stoppt. Darum läuft nginx mit `daemon off;` im Vordergrund. Das offizielle Image regelt das schon im `CMD`.

STOLPERFALLE

`VITE_`-Variablen oder ähnliche Build-Werte stehen fest im ausgelieferten JavaScript. Das Image lässt sich danach nicht einfach umkonfigurieren. Lade Konfiguration zur Laufzeit oder ersetze Platzhalter beim Start.

MERKE

Das offizielle nginx-Image füllt Dateien aus `/etc/nginx/templates/` beim Start per `envsubst` mit Umgebungsvariablen. So brauchst du kein eigenes Entrypoint-Skript.

IM INTERVIEW

Warum läuft nginx im Container mit `daemon off`?

Ein Container lebt so lange wie PID 1. Geht nginx in den Hintergrund, endet der Startprozess, und der Container stoppt. Also bleibt nginx im Vordergrund.

DAS NIMMST DU MIT

- Node nur in der Bau-Stufe.
- `nginx-unprivileged` auf Port 8080.
- `try_files` für den Client-Router.
- Container brauchen einen Vordergrund-Prozess.

TEIL V

Sicherheit und Laufzeit

Non-Root, Signale, Pinning, Scannen und Signieren.

26

Non-Root

Das einfachste große Sicherheitsplus.

Standardmäßig läuft ein Container als `root`. Der Container-`root` ist zwar eingeschränkt, aber es ist derselbe Kernel. Bei einer Kernel-Lücke oder einer Fehlkonfiguration wird daraus schnell ein Problem für den Host.

Einen Nutzer anlegen

Debian und 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
```

Danach:

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

> **MERKE**
>
> Nimm eine hohe UID wie 10001. So kollidiert sie nicht mit Nutzern auf dem Host.

> **DAS BILD IM KOPF**
>
> *Selbst im eigenen Hotelzimmer lässt du nicht den Generalschlüssel im Schloss stecken. Wer einbricht, soll nur dieses eine Zimmer haben.*

Typische Hürden

- **Niedrige Ports:** Ports unter 1024 sind für Nicht-Root nicht überall erlaubt. Lausche auf 8080.

- **Schreibrechte:** Die Anwendung darf nur in ausgewählte Ordner schreiben.
- **Volumes:** Ein gemountetes Volume kann `root` gehören. Plane die Rechte ein.

Noch enger beim Start

Das Image legt den Grundstein, der Start macht den Rest:

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

Schalter	Wirkung
`--read-only`	Das Dateisystem des Containers wird schreibgeschützt.
`--tmpfs /tmp`	Ein beschreibbarer Bereich im Speicher.
`--cap-drop=ALL`	Alle Linux-Capabilities weg.
`no-new-privileges`	Prozesse können keine Rechte hinzugewinnen, etwa über `setuid`.

STOLPERFALLE

`USER` im Dockerfile lässt sich beim Start überschreiben: `docker run --user 0` macht wieder `root` daraus. In Kubernetes erzwingst du es mit `runAsNonRoot` und passenden Richtlinien.

IM INTERVIEW

Wie machst du ein Image non-root?

Ich lege einen Nutzer mit fester UID an, setze mit `--chown` die Rechte und nenne ihn mit `USER 10001`. Die Anwendung lauscht auf einem hohen Port. In Kubernetes prüft `runAsNonRoot` das.

DAS NIMMST DU MIT

- Eigene UID, `USER` als Nummer.
- Hohe Ports, enge Schreibrechte.
- `--read-only`, `--cap-drop=ALL`, `no-new-privileges` beim Start.

27

PID 1 und Signale

Warum dein Container zehn Sekunden zum Sterben braucht.

Was bei `docker stop` passiert

```
docker stop  -->  SIGTERM an PID 1
             -->  bis zu 10 Sekunden warten
             -->  SIGKILL, Exit-Code 137
```

Reagiert der Prozess nicht, stirbt er hart: keine Verbindungen geschlossen, keine Daten gesichert.

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

Eselsbrücke: `-t` wie **t**imeout: Gib ihm 30 Sekunden zum Aufräumen.

Warum PID 1 besonders ist

Der erste Prozess im Container hat zwei Sonderrollen:

1. **Signale:** Der Kernel ignoriert Signale für PID 1, wenn der Prozess keinen eigenen Handler dafür hat.
2. **Waisen:** PID 1 übernimmt verwaiste Prozesse und muss ihr Ende abholen. Sonst bleiben **Zombies**.

> **DAS BILD IM KOPF**
>
> *PID 1 ist der Hausmeister des Containers. Er nimmt die Kündigung entgegen und räumt auf, was andere liegen lassen. Eine Shell, die nicht für ihn arbeitet, lässt die Post liegen.*

Die drei häufigen Fehler

Fehler	Lösung
Shell-Form: `CMD node app.js`	Exec-Form: `CMD ["node", "app.js"]`
Skript ohne `exec`	Letzte Zeile: `exec "$@"`
`npm start` als Vermittler	Direkt `node` starten

Ein Init-Prozess

Startet deine Anwendung Kindprozesse, oder hat sie keinen Signal-Handler, schaltest du ein kleines Init davor:

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

Eselsbrücke: Docker setzt ein winziges Init als PID 1 davor.

Im Image selbst geht es mit `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` startet dein Programm als Kind, reicht Signale weiter und räumt Zombies weg.

> **MERKE**
>
> In Kubernetes gibt es kein `--init`. Dort gehört das Init ins Image, oder die Anwendung macht es selbst richtig. Dazu kommt `terminationGracePeriodSeconds` statt `-t`.

> **STOLPERFALLE**
>
> Eine Anwendung mit langem Abschluss braucht mehr Zeit als die Standardfrist. Der Exit-Code 137 bei jedem Deployment ist ein Zeichen, dass sie hart getötet wird. Prüfe Signale und Fristen.

> **IM INTERVIEW**
>
> **Warum dauert `docker stop` manchmal genau zehn Sekunden?**
>
> Das `SIGTERM` kommt nicht an oder wird ignoriert. Docker wartet zehn Sekunden und schickt `SIGKILL`. Typische Ursache ist die Shell-Form, denn die Shell ist dann PID 1 und reicht das Signal nicht weiter.

IM INTERVIEW

Wozu braucht man `tini`?

Als kleiner PID-1-Prozess. Es leitet Signale weiter und räumt verwaiste Prozesse ab. Sonst sammeln sich Zombies, wenn die Anwendung selbst das nicht tut.

DAS NIMMST DU MIT

- `SIGTERM`, zehn Sekunden, `SIGKILL`.
- Exec-Form, `exec "$@"`, kein `npm start`.
- `--init` oder `tini` bei Kindprozessen.

28

Pinning

Gleiches Dockerfile, gleiches Ergebnis. Fast.

Ein Tag ist ein bewegliches Etikett. `node:24-slim` kann morgen auf ein anderes Image zeigen. Ein **Digest** ist der Hash des Inhalts und ändert sich nie.

```
FROM node:24-slim                       # bewegt sich
FROM node:24.22.2-slim                  # besser
FROM node:24-slim@sha256:9f86d081...    # fest
```

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

Eselsbrücke: Zeigt den Digest hinter dem Tag.

Stufe	Vorteil	Nachteil
`:latest`	Immer neu.	Unvorhersehbar. Für Produktion ungeeignet.
`:24-slim`	Bekommt Sicherheitsupdates.	Nicht reproduzierbar.
`@sha256:...`	Absolut fest.	Updates kommen nicht von selbst.

> **DAS BILD IM KOPF**
>
> *Ein Tag ist wie „das Restaurant an der Ecke“: Morgen kann es ein anderes sein. Ein Digest ist der Fingerabdruck des Gerichts.*

Das Problem mit festen Digests

Ein fester Digest bekommt keine Sicherheitsupdates. Darum gehört ein Bot dazu: **Renovate** oder **Dependabot** öffnen automatisch einen Pull Request, wenn es einen neuen Digest gibt. So bleibst du fest und trotzdem aktuell.

Auch die anderen Quellen

- **Pakete:** `npm ci`, `go.sum`, `uv.lock`, gepinnte `requirements`.
- **Downloads:** `ADD --checksum` oder `sha256sum -c`.

- **Systempakete:** Exakte `apt`-Versionen verschwinden aus den Repositories. Viele pinnen stattdessen die Basis und bauen regelmäßig neu.

> **STOLPERFALLE**
>
> Ein gleiches Dockerfile ergibt nicht automatisch das gleiche Image. Zeitstempel und frische Pakete ändern den Digest. Bit-genau reproduzierbare Builds brauchen mehr Aufwand, zum Beispiel `SOURCE_DATE_EPOCH`.

IM INTERVIEW

Tag oder Digest?

In Produktion ein Digest oder mindestens ein fester Versions-Tag, dazu ein Update-Bot. Nur so ist klar, was läuft. Der Bot sorgt dafür, dass Sicherheitsupdates trotzdem kommen.

DAS NIMMST DU MIT

- Tags bewegen sich, Digests nicht.
- Digest plus Renovate ist fest und aktuell.
- Lockfiles und Prüfsummen für alle Quellen.

29

Scannen und SBOM

Was steckt drin, und was davon ist verwundbar?

Scannen

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

Eselsbrücke: Suche nach schweren Lücken. `--exit-code 1` lässt die Pipeline scheitern.

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

Eselsbrücke: Zeige nur Lücken, für die es schon eine Lösung gibt.

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

Eselsbrücke: Die Docker-eigene Alternative. Prüft Abhängigkeiten gegen CVE-Datenbanken.

```
trivy config .
```

Eselsbrücke: Prüft nicht das Image, sondern Dockerfile und Konfiguration.

Weitere bekannte Scanner sind Grype, Snyk und der Scanner deiner Registry.

> **STOLPERFALLE**
>
> Ein Scan ist eine Momentaufnahme. Ein heute sauberes Image ist in drei Monaten verwundbar, obwohl sich nichts geändert hat. Scanne auch gespeicherte Images, und baue regelmäßig neu.

> **MERKE**
>
> Nicht jede CVE ist ausnutzbar. Schau, ob der betroffene Code läuft und ob es einen Fix gibt. Ein Image voller irrelevanter Funde beheben ist Arbeit an der falschen Stelle. Die beste Abhilfe ist oft eine kleinere Basis.

SBOM

Eine **SBOM** (Software Bill of Materials) ist die Zutatenliste des Images. Taucht morgen eine neue Lücke auf, siehst du sofort, welche Images betroffen sind, ohne neu zu scannen.

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

Eselsbrücke: Erzeuge die Zutatenliste im SPDX-Format.

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

Eselsbrücke: BuildKit erzeugt SBOM und Herkunftsnachweis direkt beim Build und hängt sie ans Image.

> **DAS BILD IM KOPF**
>
> *Die SBOM ist die Zutatenliste auf der Lebensmittelverpackung. Wird ein Rückruf gemeldet, musst du nicht jede Packung öffnen.*

Format	Bekannt für
SPDX	ISO-Standard, stark bei Lizenzen.
CycloneDX	Aus dem OWASP-Umfeld, stark bei Sicherheit.

IM INTERVIEW

Wie gehst du mit Schwachstellen in Images um?

Scan in der Pipeline mit Abbruch bei `HIGH` und `CRITICAL`, sofern ein Fix existiert. Dazu regelmäßige Scans der gespeicherten Images und automatische Rebuilds. Kleine Basis-Images reduzieren die Menge an Funden.

IM INTERVIEW

Wozu dient eine SBOM?

Sie listet alle Komponenten eines Images auf. Bei einer neuen Lücke wie Log4Shell weißt du sofort, wo du betroffen bist. Auch Lizenzprüfungen und Compliance nutzen sie.

DAS NIMMST DU MIT

- `trivy image` in die Pipeline.
- Auch gespeicherte Images erneut scannen.
- SBOM ist die Zutatenliste.
- Rebuilds sind Pflege, kein Sonderfall.

30

Signieren und Provenance

Beweise, dass dieses Image wirklich von dir stammt.

Eine Registry sagt dir nicht, wer ein Image gebaut hat. Zwei Nachweise schließen die Lücke:

- **Signatur:** Dieses Image wurde von dieser Identität freigegeben.
- **Provenance:** So, wann und woraus wurde es gebaut.

Signieren mit cosign

Signiert wird immer der Digest, nie ein Tag.

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

Eselsbrücke: Keyless: Du meldest dich per OIDC an, und cosign holt ein kurzlebiges Zertifikat. In CI nutzt es die Identität der Pipeline, und `--yes` überspringt die Rückfrage.

```
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...
```

Beim Prüfen sagst du also, **wer** signiert haben muss. Verankere das Muster mit `^` und nenne das Repository. Ein Muster wie `acme/.*` würde jedem Workflow der Organisation vertrauen.

> **DAS BILD IM KOPF**
>
> *Die Signatur ist das Siegel auf dem Brief. Es nützt nur, wenn der Empfänger auch prüft, ob es unbeschädigt ist und von der richtigen Person stammt.*

Provenance

Provenance beschreibt die Herkunft: Quelle, Commit, Builder, Parameter. Das Rahmenwerk dazu heißt **SLSA**.

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

Am Ende muss jemand prüfen

Eine Signatur, die niemand prüft, ist nutzlos. In Kubernetes prüft ein Admission Controller beim Deployment, zum Beispiel Kyverno oder Sigstore Policy Controller: Unsignierte Images werden abgewiesen.

> **STOLPERFALLE**
>
> Mit `mode=max` kann die Provenance die Build-Argumente enthalten. Ein weiterer Grund, Geheimnisse nie als `--build-arg` zu übergeben.

IM INTERVIEW

Wie stellst du sicher, dass nur vertrauenswürdige Images laufen?

Ich signiere in CI den Digest mit cosign und erzeuge Provenance und SBOM. Beim Deployment prüft ein Admission Controller die Signatur und die Identität. Unsignierte Images kommen nicht in den Cluster.

DAS NIMMST DU MIT

- Immer den Digest signieren.
- Keyless cosign nutzt OIDC.
- Provenance erklärt die Herkunft.
- Prüfen gehört zum Deployment.

TEIL VI

Alltag

Lint, CI, Fehlersuche und drei Dockerfiles aus dem echten Leben.

31

Lint: hadolint und Build Checks

Ein Roboter findet die Hälfte der Fehler vor dem Review.

hadolint

```
hadolint Dockerfile
```

Eselsbrücke: **Lint** ist der Fussel im Trockner: Das Werkzeug zupft die kleinen Fehler heraus.

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

Eselsbrücke: Dasselbe ohne Installation, per Image.

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

Die Regeln, die du kennen solltest:

Regel	Bedeutung
`DL3002`	Der letzte `USER` sollte nicht `root` sein.
`DL3003`	`WORKDIR` statt `cd`.
`DL3006`	Immer einen Tag für `FROM` nennen.
`DL3007`	Nicht `latest` nutzen.
`DL3008`	Versionen bei `apt-get install` pinnen.
`DL3009`	Die `apt`-Listen löschen.
`DL3015`	`--no-install-recommends` nutzen.
`DL3020`	`COPY` statt `ADD` für Dateien.
`DL3025`	JSON-Form für `CMD` und `ENTRYPOINT`.
`DL4006`	`pipefail` bei Pipes setzen.

Eine Regel schaltest du gezielt ab, und zwar mit Begründung:

```
# Basis ist per Digest gepinnt, Pakete kommen mit ihr
# hadolint ignore=DL3008
RUN apt-get install -y --no-install-recommends curl
```

Für das ganze Projekt gibt es die Datei `.hadolint.yaml`:

`.hadolint.yaml`

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

Build Checks

BuildKit prüft seit einiger Zeit selbst. Bei jedem Build erscheinen Warnungen, und mit `--check` läuft nur die Prüfung, ohne zu bauen.

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

Eselsbrücke: Prüfe nur, baue nicht.

Beispiele für Regeln:

Regel	Findet
`JSONArgsRecommended`	`CMD` oder `ENTRYPOINT` in Shell-Form.
`SecretsUsedInArgOrEnv`	Verdächtige Namen in `ARG` oder `ENV`.
`CopyIgnoredFile`	`COPY` einer Datei, die `.dockerignore` ausschließt.
`UndefinedVar`	Variablen, die nicht definiert sind.
`FromAsCasing`	Unterschiedliche Schreibweise von `FROM` und `AS`.

Als Pflicht im Dockerfile:

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

> **STOLPERFALLE**
>
> Mit `error=true` und `syntax=docker/dockerfile:1` können neue Regeln alte Builds brechen. Pinne die Syntax-Version genauer, wenn du das setzt. Die Docker-Doku empfiehlt das ausdrücklich.

IM INTERVIEW

Wie sicherst du die Qualität von Dockerfiles?

hadolint und `docker build --check` laufen in der Pipeline, dazu `trivy config`. Das fängt Standardfehler ab, bevor ein Mensch das Review macht.

DAS NIMMST DU MIT

- `hadolint` und `docker build --check`.
- Ausnahmen nur mit Begründung.
- `# check=error=true` macht Warnungen zu Fehlern.

32

CI

Vom Commit bis zum signierten Image.

Die Reihenfolge

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

Erst wenn alles grün ist, kommt das Image in die 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
```

MERKE

Die Versionen der Actions sind Beispiele. Prüfe die aktuellen, und pinne Actions in sicherheitskritischen Pipelines auf einen Commit-Hash.

Tags

- **Commit-SHA** für jedes Image. Eindeutig und nachvollziehbar.
- **Version**, etwa `1.4.2`, für Releases.
- Tags in der Registry möglichst unveränderlich einstellen.

Bauen ohne Docker-Socket

Den Docker-Socket in einen CI-Job zu mounten, gibt dem Job praktisch Root-Rechte auf dem Host. Das sind die Alternativen:

Werkzeug	Idee
BuildKit rootless	BuildKit als eigener Dienst ohne Root-Rechte.
Buildah	Baut Images ohne Daemon.
Kaniko	Baute Images in einem Pod. Das Google-Projekt wurde 2025 archiviert, prüfe daher Forks.

STOLPERFALLE

Baue einmal und befördere dasselbe Image durch die Umgebungen. Wer für jede Stufe neu baut, testet nicht das, was in Produktion läuft. Konfiguration kommt beim Start, nicht beim Build.

IM INTERVIEW

Wie sieht eine gute Image-Pipeline aus?

Lint, Build mit exportiertem Cache, Tests, Scan, Push und Signatur. Getaggt wird mit dem Commit-SHA. Dasselbe Image wandert dann durch alle Umgebungen, und Konfiguration kommt von außen.

IM INTERVIEW

Warum nicht den Docker-Socket in CI mounten?

Wer den Socket hat, kann privilegierte Container starten und damit den Host übernehmen. Besser sind rootless BuildKit oder ein getrennter Builder.

DAS NIMMST DU MIT

- Lint, Build, Test, Scan, Push, Signieren.
- `cache-to` und `cache-from` auf frischen Runnern.
- Einmal bauen, überall dasselbe Image.
- Kein Docker-Socket in CI-Jobs.

33

Wenn der Build scheitert

Die häufigsten Fehlermeldungen und was sie bedeuten.

Die Methode

1. Ausgabe vollständig lesen: `--progress=plain`.
2. Letzte funktionierende Stufe bauen und hineinschauen.
3. Cache als Ursache ausschließen: `--no-cache`.

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

Eselsbrücke: Baue bis zur Stufe, die gerade noch klappt.

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

Eselsbrücke: Geh hinein und führe den kaputten Befehl von Hand aus.

Häufige Meldungen

Meldung	Ursache
`failed to compute cache key: ... not found`	Datei liegt nicht im Kontext oder steht in `.dockerignore`.
`pull access denied`	Name falsch, oder das Image ist privat. `docker login` nötig.
`toomanyrequests`	Rate Limit der Registry. Anmelden oder einen Spiegel nutzen.
`no space left on device`	Der Cache füllt die Platte. `docker builder prune`.
`Unable to locate package`	`apt-get update` fehlt im selben `RUN`.
`404 Not Found` bei `apt`	Alte Paketlisten aus dem Cache. `--no-cache` oder `update` im selben `RUN`.
`exit code: 127`	Befehl gibt es im Image nicht.
`exec format error`	Fremde Architektur ohne Emulator.

STOLPERFALLE

„Bei mir geht es, in CI nicht“ liegt oft an einem Cache, der lokal etwas Veraltetes liefert. Baue lokal einmal mit `--no-cache --pull`. Das zeigt den echten Zustand.

Warum wird nichts neu gebaut?

- Geänderte Datei steht in `.dockerignore`.
- `RUN` wurde nicht verändert, also greift der Cache. Der Text zählt, nicht das Netz.

Warum wird immer alles neu gebaut?

- `COPY . .` steht zu weit oben.
- Eine Datei ändert sich bei jedem Build, etwa `.git` im Kontext oder ein Zeitstempel.
- Ein wechselnder `--build-arg` steht weit oben.

IM INTERVIEW

Ein Build läuft lokal, aber nicht in CI. Wie gehst du vor?

Ich baue lokal mit `--no-cache --pull` und vergleiche Basis, Plattform und Kontext. Oft liefert der lokale Cache etwas Altes, oder die Architektur unterscheidet sich.

DAS NIMMST DU MIT

- `--progress=plain`, dann `--target`, dann hineinschauen.
- `compute cache key`: Kontext und `.dockerignore`.
- `--no-cache --pull` zeigt den wahren Zustand.

34

Wenn der Container scheitert

Er baut, startet aber nicht. Oder nicht lange.

Die erste Minute

```
docker ps -a
```

Eselsbrücke: Auch beendete Container, mit Exit-Code.

```
docker logs web
```

Eselsbrücke: Was hat er noch gesagt?

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

Eselsbrücke: Exit-Code, und ob der Kernel ihn wegen Speicher getötet hat.

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

Eselsbrücke: Starte statt der App eine Shell, und schau dich um.

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

Eselsbrücke: Geh in einen laufenden Container.

```
docker diff web
```

Eselsbrücke: Welche Dateien hat der Container geändert?

Exit-Codes

Code	Bedeutung
`0`	Sauber beendet. Vielleicht war nichts zu tun?
`1`	Die Anwendung meldet einen Fehler.
`126`	Datei gefunden, aber nicht ausführbar. Rechte prüfen.
`127`	Befehl nicht gefunden.
`137`	`SIGKILL` (128 + 9). OOM oder `docker stop` nach Fristablauf.
`139`	Segfault (128 + 11). Oft falsche Bibliothek.
`143`	`SIGTERM` (128 + 15). Ein sauberer Stopp von außen.

Häufige Meldungen

Meldung	Ursache
`exec format error`	Falsche Architektur, oder dem Skript fehlt das Shebang.
`no such file or directory`, obwohl die Datei da ist	Windows-Zeilenenden im Skript, oder der Lader fehlt: glibc-Binärdatei auf Alpine oder `scratch`.
`permission denied`	Datei nicht ausführbar oder falscher Nutzer. `COPY --chmod=755`.
Container endet sofort	Prozess ging in den Hintergrund, oder es gab nichts zu tun.
Port nicht erreichbar	Anwendung lauscht auf `127.0.0.1` statt `0.0.0.0`.

DAS BILD IM KOPF

Der Exit-Code ist der Totenschein. Über 128 heißt: Ein Signal war's. Zieh 128 ab, dann hast du die Signalnummer.

STOLPERFALLE

Lauscht eine Anwendung nur auf `127.0.0.1`, ist sie aus dem Container nicht erreichbar. `-p` zeigt auf die Container-Schnittstelle, nicht auf dessen Loopback. Binde an `0.0.0.0`.

STOLPERFALLE

Zeilenenden sind ein Klassiker unter Windows. Ein Skript mit CRLF sucht `/bin/sh\r`, und die Meldung sagt nur `no such file or directory`. Abhilfe: `.gitattributes` oder `dos2unix`.

IM INTERVIEW

Ein Container endet sofort mit Exit-Code 137. Was prüfst du?

137 bedeutet `SIGKILL`. Ich prüfe `OOMKilled` in `docker inspect`. Ist das `true`, passt das Speicherlimit nicht. Sonst wurde er von außen getötet, etwa von `docker stop` nach Ablauf der Frist.

DAS NIMMST DU MIT

- `docker logs`, `inspect`, `--entrypoint sh`.
- Über 128: Signal, minus 128.
- `no such file` kann der Lader oder CRLF sein.
- `0.0.0.0` statt `127.0.0.1`.

35

Drei Dockerfiles verbessern

Echte Muster aus dem Alltag. Finde die Fehler zuerst selbst.

Fall 1: nginx

Vorher:

```
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"]
```

Was stört?

- `debian:latest` bewegt sich und ist unnötig groß.
- Das `;` lässt den Build weiterlaufen, selbst wenn `apt-get` scheitert.
- Die `apt`-Listen bleiben im Layer.
- Es läuft als `root` auf Port 80.

Nachher:

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

Das fertige Image bringt schon alles mit. Eigene Pakete sind gar nicht nötig.

Fall 2: Node

Vorher:

```
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"]
```

Was stört?

- Node 16 bekommt längst keine Updates mehr.
- Die Vollversion des Images ist groß.
- Kein `WORKDIR`, die Dateien landen in `/`.
- `npm install` hält sich nicht streng ans Lockfile.
- Es läuft als `root` auf Port 80.

Lob: Das Kopieren der Lockfiles vor dem Code stimmt schon.

Nachher:

```
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"]
```

Die Anwendung muss dafür auf Port 8080 lauschen.

Fall 3: Datenbanktest mit Passwort

Vorher:

```
FROM postgres:15-alpine
RUN apk --no-cache add bash
ENV POSTGRES_HOST=meine-db.example.com
ENV POSTGRES_USER=admin
ENV POSTGRES_PASSWORD=geheim
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"]
```

Was stört?

- Das Passwort steckt für immer im Image.
- Auch `HOST` und `USER` gehören zur Umgebung, nicht ins Image.
- Der ganze Postgres-Server ist an Bord, obwohl nur `psql` gebraucht wird.
- `chmod` macht einen zweiten Layer. `COPY --chmod` reicht.

Nachher:

```
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"]
```

Der Start nennt die Werte von außen:

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

> **MERKE**
>
> Wenn ein Geheimnis je in einem Image war, reicht ein neuer Build nicht. Das Passwort gilt als verbrannt und muss getauscht werden.

DAS NIMMST DU MIT

- Fertige Images ersetzen viele `RUN`-Zeilen.
- Konfiguration und Geheimnisse kommen von außen.
- Prüfliste: Basis, Layer, Cache, Nutzer, Geheimnisse.

TEIL VII

Das Interview

Fragen, Live-Aufgaben und ein Spickzettel für den letzten Abend.

36

Die Fragen

Kurze Fragen, kurze Antworten. Zum Wiederholen vor dem Gespräch.

Grundlagen

Image oder Container?

Das Image ist die unveränderliche Vorlage, der Container der laufende Prozess mit beschreibbarer Schicht.

Was ist ein Layer?

Eine unveränderliche Schicht mit den Dateiänderungen eines Schritts.

Welche Anweisungen erzeugen Datei-Layer?

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

Warum macht `rm` im späteren `RUN` das Image nicht kleiner?

Die Datei bleibt im früheren Layer.

Was ist der Build-Kontext?

Die Dateien, die `COPY` und `ADD` sehen dürfen.

Wozu `.dockerignore`?

Weniger Daten, keine Geheimnisse, stabilerer Cache.

Was ist Copy-on-Write?

Eine Datei wird erst beim Ändern in den Container-Layer kopiert.

Was bedeutet `latest`?

Nur der Standard-Tag. Keine Garantie für Aktualität.

Tag oder Digest?

Der Tag ist ein bewegliches Etikett, der Digest ein fester Inhalts-Hash.

Anweisungen

`CMD` oder `ENTRYPOINT`?

`ENTRYPOINT` ist das Programm, `CMD` sind die Standard-Argumente.

Shell-Form oder Exec-Form?

Exec-Form startet direkt, Shell-Form über `/bin/sh -c`.

Warum Exec-Form?

Das Programm wird PID 1 und bekommt Signale.

`COPY` oder `ADD`?

`COPY`. `ADD` nur für Tar-Auspacken oder URLs mit Prüfsumme.

`ENV` oder `ARG`?

`ARG` nur beim Bauen, `ENV` auch zur Laufzeit.

Darf ein Geheimnis in `ENV`?

Nein. Es steht dann dauerhaft in den Metadaten.

Öffnet `EXPOSE` einen Port?

Nein, es dokumentiert. Veröffentlicht wird mit `-p`.

Warum `WORKDIR` statt `cd`?

`cd` wirkt nur in einem einzigen `RUN`.

Wirkt `USER` auf `COPY`?

Nein. Dafür gibt es `--chown`.

Was passiert mit Dateien nach `VOLUME`?

Spätere Änderungen an diesem Pfad gehen beim Build verloren.

Was macht `ONBUILD`?

Es verschiebt Anweisungen in das Image, das darauf aufbaut.

Was macht `SHELL`?

Es ändert die Shell für die Shell-Form, oft für `pipefail`.

Wofür `HEALTHCHECK`?

Er prüft, ob die Anwendung funktioniert, nicht nur läuft.

Nutzt Kubernetes `HEALTHCHECK`?

Nein, dort gelten Probes.

Cache und Build

Wie funktioniert der Cache?

Anweisung für Anweisung. Ein Miss baut alles Folgende neu.

Woran erkennt Docker `RUN`?

Am Text der Anweisung.

Woran `COPY`?

An der Prüfsumme der Dateien.

Wie ordnest du ein Dockerfile?

Selten Geändertes oben, oft Geändertes unten.

Warum Lockfile vor Code kopieren?

So bleibt der Abhängigkeits-Layer im Cache.

Warum `update` und `install` im selben `RUN`?

Sonst nutzt `install` alte Paketlisten.

Was ist BuildKit?

Der Standard-Build-Motor mit Graph, Parallelität und Secrets.

Was ist ein Multi-Stage-Build?

Mehrere `FROM`, nur die letzte Stufe wird zum Image.

Was macht `--target`?

Es baut bis zu einer bestimmten Stufe.

Was ist ein Cache Mount?

Ein Ordner für Paket-Caches, der bestehen bleibt, aber nicht im Image landet.

Wie cachst du in CI?

`--cache-to` und `--cache-from` mit Registry oder `gha`.

Wie nutzt du Tokens beim Build?

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

Wie klonst du ein privates Repo?

Mit `RUN --mount=type=ssh` und `--ssh default`.

Wie baust du für mehrere Architekturen?

`buildx build --platform ... --push`, am besten mit Cross-Compile.

Was ist ein Image Index?

Eine Liste, die pro Plattform auf ein Image zeigt.

Basis und Sprachen

Alpine oder slim?

`slim` als Standard, Alpine wenn musl sicher passt.

Was ist musl?

Die kleine C-Bibliothek von Alpine, nicht glibc-kompatibel.

Was ist distroless?

Laufzeit ohne Shell und Paketmanager.

Was ist `scratch`?

Ein leeres Image.

Warum `npm ci`?

Es hält sich exakt ans Lockfile.

Warum nicht `npm start`?

Es steht zwischen Docker und Node und stört Signale.

Warum `PYTHONUNBUFFERED`?

Logs erscheinen sofort.

Warum `CGO_ENABLED=0`?

Für eine statische Go-Binärdatei.

Warum JRE statt JDK?

Der Compiler ist zur Laufzeit überflüssig.

Sicherheit und Betrieb

Warum nicht `root`?

Bei einem Ausbruch wäre der Schaden größer.

Warum `USER` als Nummer?

`runAsNonRoot` kann nur Nummern prüfen.

Was passiert bei `docker stop`?

`SIGTERM`, zehn Sekunden warten, dann `SIGKILL`.

Warum ist PID 1 besonders?

Ohne Handler ignoriert er Signale, und er muss Waisen abräumen.

Was macht `tini`?

Es ist ein kleiner PID 1, der Signale weiterleitet und Zombies abräumt.

Was bedeutet Exit-Code 137?

`SIGKILL`, oft durch OOM.

Was ist eine SBOM?

Die Zutatenliste eines Images.

Was ist Provenance?

Der Nachweis, woraus und wie ein Image gebaut wurde.

Warum den Digest signieren?

Ein Tag kann sich ändern, ein Digest nicht.

Warum regelmäßig neu bauen?

Damit Sicherheitsupdates der Basis ankommen.

Warum ist der Docker-Socket gefährlich?

Sein Besitzer hat praktisch Root auf dem Host.

Wenn du etwas nicht weißt

Sag es. Dann denk laut: „Das habe ich so nicht benutzt. Ich vermute X, weil Y. Ich würde es in der Doku unter Z nachschlagen.“ Das zeigt mehr als geratenes Wissen.

DAS NIMMST DU MIT

- Kurze Antwort zuerst. Details nur, wenn jemand nachfragt.
- Denke immer an Layer, Cache und Sicherheit.

37

Live-Aufgaben

Fünf Aufgaben, die im Interview oft verlangt werden.

> **MERKE**
>
> Ablauf im Interview: Nachfragen, laut denken, klein anfangen, verbessern. Fragen zuerst: Welche Sprache? Welche Zielplattform? Wie läuft das Image? Sprich beim Tippen.

Aufgabe 1: Schreibe ein gutes Dockerfile für einen Go-Dienst

Das Gerüst aus Kapitel 23 zeigt alles: zwei Stufen, Abhängigkeiten zuerst, `CGO_ENABLED=0`, distroless `nonroot`. Sag dazu, warum: Größe, Sicherheit, Cache.

Aufgabe 2: Warum ist dieses Dockerfile langsam?

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

- `COPY . /app` steht vor `pip install`. Jede Code-Änderung installiert alles neu.
- Es gibt keine `.dockerignore`.
- Die Vollversion der Basis ist groß.
- `CMD` steht in Shell-Form.
- Es läuft als `root`.

Die Lösung:

```
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"]
```

Aufgabe 3: Das Image ist 1,2 GB groß. Was tust du?

1. Messen mit `docker history` und `dive`.
2. Basis wechseln, von voll auf `slim` oder distroless.
3. Multi-Stage einführen.
4. Dev-Abhängigkeiten und Caches entfernen.
5. `.dockerignore` prüfen.

Aufgabe 4: `docker stop` dauert immer zehn Sekunden

- Dockerfile prüfen: Shell-Form? Skript ohne `exec`? `npm start`?
- Mit `docker exec web ps` schauen, was PID 1 ist.
- Auf Exec-Form umstellen und `SIGTERM` behandeln, oder `--init` ergänzen.

Aufgabe 5: Im Image wurde ein Token gefunden

1. Das Token sofort widerrufen. Das ist der wichtigste Schritt.
2. Das Image aus der Registry entfernen.
3. Ursache beheben: `ENV`, `ARG` oder `COPY` durch Secret Mounts ersetzen.
4. Neu bauen und mit einem Secret-Scanner prüfen.
5. Die Pipeline ergänzen, damit das nicht wieder passiert.

IM INTERVIEW

Wie erkennst du Geheimnisse in Images?

Ich schaue in `docker history` und in die Layer, zum Beispiel mit `dive`. Zusätzlich laufen in CI Secret-Scanner wie der von Trivy über Image und Repository.

DAS NIMMST DU MIT

- Erst fragen, dann schreiben.
- Immer drei Sätze: Größe, Cache, Sicherheit.
- Geheimnis gefunden? Zuerst widerrufen.

38

Spickzettel

Alles Wichtige auf ein paar Seiten, für den letzten Abend.

Das Muster für fast alles

```
# syntax=docker/dockerfile:1
FROM basis:version AS build
WORKDIR /src
COPY lockfile ./
RUN abhaengigkeiten-installieren
COPY . .
RUN bauen

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

Der eine Satz zu allem

Begriff	Merksatz
Layer	Unveränderlich, und Löschen spart nichts.
Cache	Selten oben, oft unten.
Multi-Stage	Werkstatt und Laden.
`CMD`	Standard-Argumente.
`ENTRYPOINT`	Das Programm.
`ARG`	Nur beim Bauen.
`ENV`	Auch zur Laufzeit.
`EXPOSE`	Nur Doku.
PID 1	Exec-Form oder `exec "$@"`.
Digest	Der feste Fingerabdruck.

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 .
```

Untersuchen

```
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
```

Qualität und Sicherheit

```
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:...
```

Zahlen, die du kennen musst

Wert	Bedeutung
10 s	Frist von `docker stop` bis `SIGKILL`.
137	`SIGKILL`, also 128 + 9. Oft OOM.
143	`SIGTERM`, also 128 + 15.
126, 127	Nicht ausführbar, nicht gefunden.
30 s, 30 s, 3	`HEALTHCHECK`: Intervall, Timeout, Versuche.
10001	Eine typische UID ohne Rechte.

Checkliste für jedes Dockerfile

- Basis gepinnt, klein und aus vertrauenswürdiger Quelle?
- Multi-Stage, wenn etwas gebaut wird?

- Lockfile vor Code kopiert?
- `.dockerignore` vorhanden?
- `CMD` und `ENTRYPOINT` in Exec-Form?
- `USER` als Nummer gesetzt?
- Keine Geheimnisse in `ENV`, `ARG` oder Layern?
- hadolint und Scan grün?

DAS NIMMST DU MIT

- Layer verstehen heißt Dockerfiles verstehen.
- Selten oben, oft unten.
- Klein, non-root, gepinnt, gescannt.

Stichwörter

ADD 23
Alpine 58
ARG 28
Build Checks 86
Build-Cache 40
Build-Kontext 15
BuildKit 43
buildx 43
Cache Mount 41
CGO_ENABLED 65
CI 89
CMD 25
COPY 23
COPY --link 24
Copy-on-Write 10
cosign 82
Digest 77
distroless 58
docker build 13
Docker Scout 79
Dockerfile 8
.dockerignore 15
ENTRYPOINT 25
ENV 28
exec format error 94
Exit-Code 94
EXPOSE 32
FROM 18
glibc 58
hadolint 86
HEALTHCHECK 34
Heredoc 21
Image Index 52
Image-Größe 55
jlink 67
Kaniko 89
LABEL 32
Layer 10
MaxRAMPercentage 67
Multi-Arch 52
Multi-Stage 46
musl 58
nginx 69
Non-Root 72
npm ci 61
OCI 11
ONBUILD 36
PID 1 74
pipefail 36
Provenance 82
Renovate 77
RUN 20
runAsNonRoot 72
SBOM 79
scratch 18
Secret Mount 49
SHELL 36
SIGTERM 74
slim 58
SLSA 82
SSH-Mount 49
STOPSIGNAL 33
syntax 37
Tag 13
tini 74
Trivy 79
USER 30
uv 63
venv 63
VOLUME 32
WORKDIR 30
Zombie 74