



Warum Docker Images bauen?

-  Apps ausliefern, unabhängig von der Umgebung
-  Vorhandene Images anpassen für Unternehmensanforderungen
 - z. B. interne Tools installieren
 - Eigene **Root-CA-Zertifikate** einfügen

Was ist ein Dockerfile?

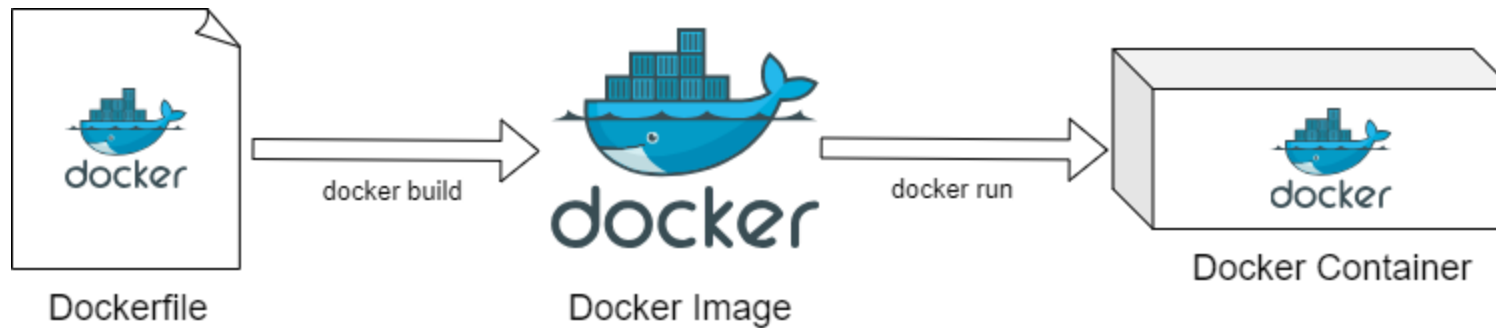
Ein **Dockerfile** ist eine Sammlung von Anweisungen zum automatisierten Erstellen eines Images.

Darin definierst du:

- welches Basis-Image verwendet wird
- welche Dateien kopiert werden
- was installiert werden soll
- wie der Container gestartet wird

Man kann es sich wie ein **Rezept zum Bauen eines Images** vorstellen.

Vom Dockerfile zum Container



Dieser Ablauf beschreibt, wie aus Quellcode und Instruktionen ein laufender Container wird.

Zentrale Dockerfile-Befehle

FROM – Basis-Image wählen

Definiert, auf welchem Image dein neues Image basiert.

Beispiel:

```
FROM alpine:3.20
```

- Muss der **erste Befehl** im Dockerfile sein
- Kann ein offizielles Image oder ein eigenes sein
- Du kannst mehrere `FROM` verwenden (für Multi-Stage Builds)

COPY – Dateien ins Image übernehmen

Kopiert Dateien vom Build-Kontext ins Image.

Beispiel:

```
COPY ./src /app
```

- Relativ zum Pfad, in dem der Build gestartet wird
- Zielpfad muss im Image existieren oder wird erstellt

RUN – Kommandos zur Build-Zeit ausführen

Führt Shell-Kommandos beim Erstellen des Images aus.

Beispiel:

```
RUN apt-get update && apt-get install -y curl
```

- Wird während des Builds ausgeführt

CMD – Standardstartbefehl des Containers

Legt fest, was ausgeführt wird, wenn der Container startet.

Beispiel:

```
CMD ["echo", "Hello World"]
```

- Es kann nur **ein CMD** geben (der letzte zählt)
- Kann beim `docker run` überschrieben werden
- Nutzt entweder **Exec-Form** (`["node", "app.js"]`) oder **Shell-Form** (`CMD echo Hello`)

Kurzübersicht der 4 Dockerfile-Befehle

- `FROM` : Legt das Basis-Image fest, auf dem das neue Image aufbaut.
- `COPY` : Übernimmt Dateien aus dem lokalen Kontext ins Image.
- `RUN` : Führt Befehle während des Builds aus, z. B. zur Installation von Software.
- `CMD` : Definiert den Standardbefehl, der beim Starten des Containers ausgeführt wird.



Übung 1 - Einfaches Docker Image erstellen

- Wechsel in den Ordner `examples/example-1`
- Ergänze die Dockerfile, das:
 - die `index.js` und `packages.json` ins Image kopiert wird
 - im Image `npm install`
 - Beim start des containers die Anwendung `node` mit der `index.js` aufgerufen wird
- Baue und starte das Image

Erwartete Ausgabe:



```
Welcome to your first Dockerized Node.js app!
```

Wie funktionieren Layer & Caching?

Docker baut Images **Schritt für Schritt**, jede Anweisung erzeugt einen **Layer**.

- Jeder Layer wird **zwischengespeichert (Cache)**.
- Wenn sich ein Layer nicht ändert, wird der Cache verwendet.
- Änderungen **brechen das Caching** ab dem ersten veränderten Layer.

Beispiel: Layer-Verhalten

```
COPY package.json .      # Layer 1  
RUN npm install           # Layer 2  
COPY . .                  # Layer 3
```

💡 Ändert sich `package.json`, wird `npm install` neu ausgeführt.

Ändert sich nur der Code, wird Layer 2 gecached.

➡ Reihenfolge der Befehle = Performancefaktor!

Root vs. Non-Root

Standardmäßig läuft der Container als **root**.

Risiken:

- Sicherheitslücken (root inside Container \neq sicher)
- Keine Rechtebeschränkung gegenüber dem Container-Dateisystem

Lösung: Benutzer anlegen

```
RUN useradd -m appuser  
USER appuser
```

- `USER` schaltet auf anderen Benutzer um
- Best Practice für Produktions-Container
- Manche Images haben bereits `non-root` Nutzer vorinstalliert

Was ist der Build-Kontext?

Der **Build-Kontext** ist der Ordner, den du an `docker build` übergibst:

```
docker build -t my-image .
```

➡ Hier: `.` = aktueller Ordner

- Alle Pfade im Dockerfile (z. B. `COPY`) beziehen sich auf diesen Kontext.
- Nur Dateien **im Kontext** können ins Image kopiert werden.
 - ✗ Du kannst NICHT einfach `../..`/irgendwas kopieren – alles außerhalb des Build-Kontexts ist für Docker unsichtbar!

.dockerignore – wie .gitignore für Docker

Du willst keine `.git`, `node_modules`, `secrets.txt`, etc. im Image?

Dann sag Docker: "Ignorier das bitte."

`.dockerignore` :

```
node_modules
.git
secrets.txt
```

Vorteile:

- Build schneller
- Images kleiner
- Keine versehentliche Leaks von Daten



Übung 2a – Diese Dockerfile ist schrecklich!

Das Ding läuft als root. Es wird nicht Gecached. Es übernimmt Debug-Dateien mit rein.

Fix it!

Simuliere Code-Änderungen und erstelle das Image neu.

```
"" >> index.js && docker build -t safe-node-app .
```

```
docker run --rm safe-node-app
```

➡ Erwartete Ausgabe:

✅ App is running!



Übung 3 – Build-Kontext & .dockerignore

- Wechsel in `examples/example-3`
- Erstelle `.dockerignore`, die Folgendes ignoriert:
 - `.git/`
 - `node_modules/`
 - `*.log`
- Baue das Image und inspiziere es:

```
docker build -t clean-image .  
docker run -it clean-image sh
```

➡ Prüfe, dass die ignorierten Dateien wirklich fehlen.