Docker und Projekt-Architektur Erklärung

Dieses Dokument dient dazu, Einsteigern zu erklären, wie unsere Fitness-Tracker-Anwendung mit Docker "verpackt" und gestartet wird. Wir nutzen Docker, um sicherzustellen, dass die Software auf jedem Computer (egal ob Windows, Mac oder Linux) genau gleich läuft, ohne dass man Java, Node.js oder Datenbanken manuell installieren muss.

1. Das Backend (`fitness-tracker-service/Dockerfile`)

Für das Java-Backend nutzen wir einen sogenannten **Multi-Stage Build** (Mehrstufigen Bauprozess). Das hilft uns, das finale Programm klein und effizient zu halten.

Aufbau der Datei

1. **Stage 1: Der "Bauarbeiter" (Build Stage)**

```
FROM maven:3.9.9-eclipse-temurin-21-alpine AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests
```

   - `FROM maven...`: Wir starten mit einem Image, das alle Werkzeuge hat, um Java-Code zu kompilieren (Maven & JDK). Das ist wie ein voll ausgestatteter Werkzeugkoffer.
   - `COPY ...`: Wir kopieren unseren Quellcode in diesen Container.
   - `RUN mvn ...`: Wir führen den Befehl aus, um aus dem Code eine fertige `.jar` Datei zu bauen. Das ist das eigentliche Programm.

2. **Stage 2: Der "Lieferant" (Run Stage)**

```
FROM eclipse-temurin:21-jre-alpine
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

   - `FROM eclipse-temurin...`: Hier nehmen wir ein neues, frisches Image. Wichtig: Diesmal nur das **JRE** (Java Runtime Environment), nicht das JDK. Wir brauchen keine Compiler-Werkzeuge mehr, nur noch etwas, das Java ausführen kann. Das spart extrem viel Speicherplatz.
   - `COPY --from=build`: Wir greifen in die erste Stage ("build") zurück und holen uns **nur** die fertige `.jar` Datei. Der ganze Quellcode und die Maven-Tools werden weggeworfen.
   - `ENTRYPOINT`: Das ist der Befehl, der ausgeführt wird, wenn der Container startet: "Starte die Java-Anwendung".

2. Das Frontend (`fitness-tracker-web/Dockerfile`)

Auch hier nutzen wir **Multi-Stage Builds**, aber mit anderen Technologien.

1. **Stage 1: Bauen mit Node.js**

```
FROM node:22-alpine AS build
...
RUN npm run build
```

   - Wir nutzen ein Node.js Image, um die Angular-Anwendung zu "bauen". Angular-Code (TypeScript) kann vom Browser nicht direkt verstanden werden. Er muss erst in normales JavaScript, HTML und CSS übersetzt werden. Das Ergebnis landet im Ordner `dist/`.

2. **Stage 2: Ausliefern mit Nginx**

```
FROM nginx:alpine
COPY nginx.conf /etc/nginx/nginx.conf
COPY --from=build /app/dist/fitness-tracker-app/browser
/usr/share/nginx/html
```

   - `FROM nginx...`: Angular-Apps sind nach dem Bauen nur noch statische Dateien (HTML, JS, CSS). Wir brauchen kein Node.js mehr, um sie laufen zu lassen. Ein Webserver ist viel effizienter. **Nginx** (ausgesprochen "Engine X") ist einer der schnellsten und beliebtesten Webserver der Welt.
   - Wir kopieren die fertigen Dateien aus der Build-Stage in den Ordner, aus dem Nginx Webseiten ausliefert (`/usr/share/nginx/html`).

---

3. Der Webserver: Nginx (`nginx.conf`)

Nginx ist der "Pförtner". Er nimmt Anfragen vom Browser entgegen (z.B. "Zeig mir die Startseite") und gibt die passenden Dateien zurück.

Warum brauchen wir eine extra Konfiguration?

Angular ist eine **Single Page Application (SPA)**. Das bedeutet, es gibt technisch gesehen nur eine einzige echte Seite: die `index.html`. Alles andere (wie `/users` oder `/exercises`) wird nur durch JavaScript simuliert, das den Inhalt austauscht.

Das Problem: Wenn du im Browser direkt `http://localhost/users` aufrufst, fragt der Browser den Server nach einer Datei oder einem Ordner namens "users". Den gibt es aber nicht wirklich auf der Festplatte. Der Server würde normalerweise "404 Not Found" sagen.

Die Lösung in der Config

```
location / {
    try_files $uri $uri/ /index.html;
}
```

- `try_files`: Das ist der wichtigste Befehl. Er sagt Nginx:
  1. `$uri`: Schau erst, ob es die angeforderte Datei wirklich gibt (z.B. ein Bild `logo.png`). Wenn ja, liefere sie aus.
  2. `$uri/`: Wenn nicht, schau, ob es ein Ordner ist.
  3. `/index.html`: **Wenn beides nicht zutrifft**, gib einfach die `index.html` zurück.
- Dadurch wird immer die Angular-App geladen, und Angular merkt dann selbst: "Oh, der Benutzer wollte eigentlich zu `/users`, ich zeige ihm mal die Benutzerliste an."

---

4. Der Dirigent: Docker Compose (`docker-compose.yml`)

Wenn wir Dockerfiles haben, müssten wir normalerweise jeden Container einzeln bauen und mit langen Befehlen starten. **Docker Compose** automatisiert das. Es ist wie ein Skript, das eine ganze Umgebung beschreibt.

- **Services**: Wir definieren zwei Dienste: `backend` und `frontend`.
- **Build**: Wir sagen Compose, wo die Dockerfiles liegen (`context: ./...`), damit es die Images automatisch bauen kann.
- **Ports**:
  - `"8080:8080"` beim Backend bedeutet: Leite den Port 8080 meines Laptops an den Port 8080 im Container weiter.
  - `"8081:80"` beim Frontend: Die Webseite ist unter dem Port 8081 erreichbar (z.B. `http://localhost:8081`), da Port 80 oft schon belegt ist.
- **Networks**: Wir stecken beide Container in ein gemeinsames virtuelles Netzwerk (`fitness-network`).
- **Depends_on**: Das Frontend wartet mit dem Start, bis der Backend-Container gestartet wurde (Achtung: Das heißt nur, der Container läuft, nicht unbedingt, dass Java schon komplett hochgefahren ist).

Zusammenfassung für den Start

Mit einem einzigen Befehl bauen wir alles und fahren es hoch: `docker-compose up --build`