Todo App

Eine Lern-Anwendung für Softwarearchitektur

Vom Quick & Dirty zur Clean Architecture

Robert Bretz

6. Mai 2026

Inhaltsverzeichnis

1 Server-Absicherung (Ubuntu 24.04 auf Contabo VPS)

In diesem Schritt richten wir einen frisch installierten Ubuntu 24.04 Server bei Contabo ein und härten ihn gegen Angriffe. Folgende Maßnahmen werden durchgeführt:

1. SSH-Verbindung mit Passwort testen
2. System-Updates einspielen
3. SSH-Key-Authentifizierung einrichten (Login ohne Passwort)
4. SSH-Client-Konfiguration mit Alias (Kurzbefehl)
5. SSH-Timeout auf 20 Minuten verlängern
6. Fail2Ban installieren (Bruteforce-Schutz)

Server-Daten:

- Hostname: `vmd147914`
- IP: `185.209.229.167`
- OS: Ubuntu 24.04.4 LTS
- Speicher: 386 GB SSD

1.1 Schritt 1: SSH-Verbindung testen

Die erste Verbindung zum Server erfolgt per SSH (Secure Shell) mit Benutzername und Passwort. SSH ist ein verschlüsseltes Netzwerkprotokoll, mit dem du sicher auf entfernte Server zugreifen kannst.

Ausgeführt auf deinem lokalen PC:

```
ssh root@185.209.229.167
```

Listing 1: SSH-Verbindung zum Server aufbauen

Erklärung des Befehls:

- `ssh` – der Befehl zum Starten einer SSH-Verbindung
- `root` – der Benutzername (root ist der Administrator unter Linux)
- `@` – trennt Benutzername und Server-Adresse
- `185.209.229.167` – die öffentliche IPv4-Adresse deines Servers

Nach Eingabe des Passworts erscheint der Ubuntu-Willkommensbildschirm mit Systeminformationen.

Hinweis: Bei einer neu installierten Maschine kann folgende Warnung erscheinen:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Listing 2: WARNUNG: REMOTE HOST IDENTIFICATION HAS CHANGED

Das passiert, weil der Server einen neuen SSH-Fingerabdruck hat (durch die Neuinstallation). Dein PC erinnert sich an den alten Fingerabdruck und warnt dich vor einem möglichen Man-in-the-Middle-Angriff. Da du den Server selbst neu installiert hast, ist das harmlos.

Lösung: Den alten Eintrag löschen mit:

```
ssh-keygen -f '/home/computer/.ssh/known_hosts' -R '185.209.229.167'
```

Listing 3: Alten SSH-Fingerabdruck entfernen

Danach erneut verbinden und den neuen Fingerabdruck mit `yes` bestätigen.

1.2 Schritt 2: System-Updates

Nach dem ersten Login wird das System auf den neuesten Stand gebracht.

Ausgeführt auf dem Server (root@vmd147914):

```
apt update && apt upgrade -y
```

Listing 4: System-Updates ausführen

Erklärung des Befehls:

- `apt` – Advanced Package Tool, der Paketmanager von Ubuntu/Debian
- `update` – holt die neuesten Paketlisten von den Ubuntu-Servern
- `&&` – führt den zweiten Befehl nur aus, wenn der erste erfolgreich war
- `upgrade` – installiert alle verfügbaren Aktualisierungen
- `-y` – beantwortet alle Rückfragen automatisch mit "Yes"

1.3 Schritt 3: SSH-Key-Authentifizierung

Ein SSH-Key ist sicherer als ein Passwort, da er nicht durch Ausprobieren (Bruteforce) erraten werden kann. Er besteht aus zwei Teilen:

- **Private Key** (`id_ed25519`) – bleibt auf deinem PC, niemals weitergeben!
- **Public Key** (`id_ed25519.pub`) – wird auf den Server kopiert

Das Verfahren nennt sich **asymmetrische Kryptographie**: Der Server schickt eine zufällige Nachricht, dein PC ünterschreibtßie mit dem privaten Schlüssel, der Server prüft die Unterschrift mit dem öffentlichen Schlüssel. Stimmt sie überein, bist du eingeloggt – ohne Passwort.

Schritt 3a: Key-Paar erstellen – auf deinem lokalen PC:

```
ssh-keygen -t ed25519 -C "robert@local"
```

Listing 5: SSH-Key generieren

Erklärung des Befehls:

- `ssh-keygen` – Programm zum Erstellen von SSH-Schlüsselpaaren

- `-t ed25519` – verwendet den modernen Ed25519-Algorithmus (kurz, schnell, sicher)
- `-C "robert@local"` – Kommentar, damit du später erkennst, wofür der Key ist

Bei den Rückfragen einfach Enter drücken – der Key wird im Standardverzeichnis `~/.ssh/` gespeichert.

Schritt 3b: Public Key auf den Server kopieren – auf deinem lokalen PC:

```
ssh-copy-id root@185.209.229.167
```

Listing 6: Public Key auf den Server übertragen

Einmal das Server-Passwort eingeben. Der Befehl kopiert deinen Public Key in die Datei `~/.ssh/authorized_keys` auf dem Server.

Schritt 3c: Testen – auf deinem lokalen PC:

```
ssh root@185.209.229.167
```

Listing 7: Login ohne Passwort testen

Du wirst jetzt ohne Passwort-Abfrage eingeloggt.

1.4 Schritt 4: SSH-Client-Konfiguration (Alias)

Damit du nicht jedes Mal die IP-Adresse eintippen musst, wird ein Alias in der lokalen SSH-Konfiguration eingerichtet.

Auf deinem lokalen PC:

```
nano ~/.ssh/config
```

Listing 8: SSH-Konfiguration bearbeiten

Folgenden Inhalt einfügen:

```
Host testserver
    HostName 185.209.229.167
    User root
    IdentityFile ~/.ssh/id_ed25519
```

Listing 9: Inhalt von ~/.ssh/config

Erklärung der Konfiguration:

- `Host testserver` – der Alias, unter dem du den Server ansprichst
- `HostName 185.209.229.167` – die tatsächliche Server-Adresse
- `User root` – Benutzername für die Verbindung
- `IdentityFile ~/.ssh/id_ed25519` – Pfad zum privaten Schlüssel

Testen:

```
ssh testserver
```

Listing 10: Mit Alias verbinden

Ab jetzt reicht dieser kurze Befehl.

1.5 Schritt 5: SSH-Timeout auf 20 Minuten

Standardmäßig trennt Ubuntu inaktive SSH-Verbindungen nach etwa 5 Minuten. Das wird nun auf 20 Minuten erhöht.

Auf dem Server (als root):

```
nano /etc/ssh/sshd_config
```

Listing 11: SSH-Server-Konfiguration bearbeiten

Folgende Zeilen suchen oder am Ende der Datei einfügen:

```
ClientAliveInterval 120
ClientAliveCountMax 10
```

Listing 12: Timeout-Konfiguration

Erklärung der Werte:

- `ClientAliveInterval 120` – Der Server sendet alle 120 Sekunden (2 Minuten) ein Signal an den Client
- `ClientAliveCountMax 10` – Nach 10 unbeantworteten Signalen wird die Verbindung getrennt

Die gesamte Timeout-Zeit berechnet sich: 120 Sekunden $\times$ 10 = 1200 Sekunden = 20 Minuten. **SSH-Dienst neustarten:**

```
systemctl restart ssh
```

Listing 13: SSH-Dienst neustarten

Wichtig: Auf Ubuntu heißt der Dienst `ssh`, nicht `sshd` (im Gegensatz zu anderen Distributionen). Die aktuelle Verbindung bleibt beim Neustart bestehen. Die neue Einstellung gilt für alle zukünftigen Verbindungen.

1.6 Schritt 6: Fail2Ban (Bruteforce-Schutz)

Fail2Ban ist ein Dienst, der Logdateien überwacht und IP-Adressen automatisch sperrt, wenn zu viele fehlgeschlagene Login-Versuche erkannt werden.

Was ist Bruteforce? Ein Angreifer probiert tausende Passwörter durch, bis er das richtige findet. Fail2Ban unterbindet das, indem es die IP des Angreifers nach einer bestimmten Anzahl Fehlversuche temporär sperrt.

Standard-Konfiguration (ab Werk):

- 5 Fehlversuche in 10 Minuten
- Sperrdauer: 10 Minuten
- Überwacht wird der SSH-Dienst

Wo wird installiert? Die Programmdateien liegen unter `/usr/bin/`, die Konfiguration unter `/etc/fail2ban/`.

Wo kann ich es konfigurieren? Die Datei `/etc/fail2ban/jail.local` wird bei Updates nicht überschrieben und ist für eigene Anpassungen gedacht. Beispiel:

```
bantime = 600
findtime = 600
maxretry = 3

[sshd]
enabled = true
```

Listing 14: Beispiel: /etc/fail2ban/jail.local

Installation auf dem Server:

```
apt install -y fail2ban
```

Listing 15: Fail2Ban installieren

Automatischen Start aktivieren und sofort starten:

```
systemctl enable fail2ban && systemctl start fail2ban
```

Listing 16: Fail2Ban aktivieren und starten

Status prüfen:

```
systemctl status fail2ban
```

Listing 17: Fail2Ban-Status abfragen

Die Ausgabe sollte `active (running)` zeigen.

```
- fail2ban.service - Fail2Ban Service
     Active: active (running)
     ...
     Server ready
```

Listing 18: Erfolgreiche Ausgabe

1.7 Zusammenfassung

Nach Abschluss dieses Schritts ist der Server grundlegend abgesichert:

- Passwort-Login funktioniert weiterhin (als Backup)
- SSH-Key-Login ist eingerichtet (bequem & sicher)
- Alias `ssh testserver` ist konfiguriert
- Verbindung trennt nach 20 Minuten Inaktivität
- Fail2Ban sperrt Angreifer nach 5 Fehlversuchen

Als nächstes folgt die Firewall-Konfiguration mit ufw.

2 Firewall mit UFW einrichten

In diesem Schritt konfigurieren wir die Firewall des Servers mit **UFW** (Uncomplicated Firewall). UFW ist eine benutzerfreundliche Schnittstelle für `iptables`, die seit Ubuntu 8.04 standardmäßig installiert ist.

2.1 Was ist eine Firewall und warum brauchen wir sie?

Eine Firewall ist wie ein **Türsteher vor einem Club**: Sie entscheidet, welche Datenpakete (Gäste) hereinkommen und welche draußen bleiben. Ohne Firewall steht der Server "nacktïm Internet und jeder kann an jede Tür (Port) klopfen.

Das Prinzip der minimalen Angriffsfläche:

- Jeder offene Port ist eine potenzielle **Eintrittspforte** für Angreifer
- Je weniger Ports offen sind, desto weniger Möglichkeiten gibt es für einen Angriff
- Standard-Dienste haben oft **bekannte Sicherheitslücken** – selbst wenn wir sie nicht aktiv nutzen

2.2 Die 65.535 Ports: Ein kurzer Überblick

Ein Server hat 65.535 TCP-Ports und 65.535 UDP-Ports. Jeder Netzwerkdienst lauscht auf einem bestimmten Port. Hier sind die bekanntesten:

Tabelle 1: Bekannte Ports und ihre Dienste

Port	Dienst	Risiko/Bemerkung
21	FTP	Uralt, Passwörter im Klartext – niemals offen lassen
22	SSH	Unser Verwaltungszugang – MUSS offen sein
23	Telnet	Wie SSH, aber unverschlüsselt – Todesurteil für Server
25	SMTP	Mail-Versand – Angreifer könnten Spam verschicken
53	DNS	Namensauflösung – Ziel für DDoS-Angriffe
80	HTTP	Standard-Webport – für unsere Fitness-App
110	POP3	E-Mail-Abruf – veraltet, unsicher
143	IMAP	E-Mail-Abruf – veraltet
443	HTTPS	Verschlüsselter Webport – PFLICHT für PWAs!
3306	MySQL	Datenbank – beliebtes Bruteforce-Ziel
5432	PostgreSQL	Datenbank – ebenso populär bei Angreifern
6379	Redis	Oft ohne Passwort vorkonfiguriert – sehr gefährlich
8080	HTTP-Alt	Häufig für Entwicklertools mit schwacher Absicherung
27017	MongoDB	Bekannt für katastrophale Standardkonfigurationen

Merksatz: Alles, was du nicht explizit brauchst, wird blockiert. Das ist keine Paranoia, sondern Best Practice im Server-Management.

2.3 Die drei Ports, die wir öffnen

Wir öffnen nur drei Ports – das absolute Minimum für einen Webserver:

2.4 Warum HTTPS für PWAs Pflicht ist

Eine Progressive Web App (PWA) **kann ohne HTTPS nicht installiert werden**. Das ist eine Sicherheitsanforderung von Google und Apple:

- Der **Service Worker** (das Herzstück einer PWA) benötigt zwingend HTTPS

Tabelle 2: Geöffnete Ports und ihre Begründung

Port	Dienst	Warum offen?
22	SSH	Unser **einziger Verwaltungszugang**. Ohne Port 22 könnten wir den Server nicht mehr fernsteuern – wir wären ausgesperrt.
80	HTTP	**Standard-Webport** für alle Browser. Wenn jemand deine Domain aufruft, landet er zuerst hier. Leitet später automatisch auf HTTPS (Port 443) um.
443	HTTPS	**Verschlüsselte Webseiten**. Seit 2018 Pflicht für moderne Web-Apps! Ohne HTTPS verweigern Browser Funktionen wie PWA-Installation, Kamera-Zugriff oder Standort.

- Nur so kann der Browser garantieren, dass die App nicht manipuliert wurde
- HTTP-Verbindungen können von Angreifern verändert werden (Man-in-the-Middle)

Praxis-Beispiel: Wenn du `http://deine-domain.de` aufrufst und dort die PWA installieren willst, verweigert Chrome die Installation. Erst mit `https://deine-domain.de` und einem gültigen SSL-Zertifikat funktioniert es.

2.5 Durchführung

2.5.1 Standardrichtlinien setzen

Zuerst definieren wir die grundlegenden Regeln: Alles Eingehende wird blockiert, alles Ausgehende erlaubt.

Auf dem Server:

```
ufw default deny incoming
ufw default allow outgoing
```

Listing 19: Firewall-Standardregeln definieren

Erklärung der Befehle:

- `ufw` – das Firewall-Programm (Uncomplicated Firewall)
- `default` – setzt die Standardregel für alle Ports, die nicht explizit konfiguriert sind
- `deny incoming` – alle eingehenden Verbindungen werden **abgelehnt** (geblockt)

- `allow outgoing` – alle ausgehenden Verbindungen sind **erlaubt** (Server kann selbst ins Internet)

Warum outgoing erlauben? Der Server muss Updates herunterladen können (`apt update`), auf externe APIs zugreifen und im Internet kommunizieren. Das sind alles ausgehende Verbindungen – die der Server selbst initiiert.

2.5.2 Benötigte Ports öffnen

Jetzt geben wir gezielt die drei Ports frei, die von außen erreichbar sein sollen.

```
1 ufw allow 22/tcp
2 ufw allow 80/tcp
3 ufw allow 443/tcp
```

Listing 20: Ports 22, 80, 443 für TCP freigeben

Erklärung der Befehle:

- `allow` – dieser Port wird geöffnet
- `22/tcp` – Port 22, nur TCP-Protokoll (nicht UDP)
- `80/tcp` – Port 80 (HTTP), nur TCP
- `443/tcp` – Port 443 (HTTPS), nur TCP

Warum nur TCP? SSH, HTTP und HTTPS verwenden ausschließlich das TCP-Protokoll. UDP wird von diesen Diensten nicht benötigt. Würden wir nur `ufw allow 80` (ohne `/tcp`) schreiben, wäre auch UDP offen – unnötige Angriffsfläche.

2.5.3 Firewall aktivieren

Die Firewall wurde bisher nur konfiguriert, ist aber noch nicht aktiv. Erst mit dem Enable-Befehl greifen die Regeln.

```
1 ufw --force enable
```

Listing 21: Firewall aktivieren

Erklärung:

- `enable` – schaltet die Firewall ein
- `–force` – überspringt die Sicherheitsabfrage ("Bist du sicher?") und führt den Befehl direkt aus

Achtung: Wenn du Port 22 vergessen hättest, wärst du jetzt vom Server ausgesperrt! Die Firewall würde deine aktuelle SSH-Verbindung zwar nicht sofort trennen, aber ein erneuter Login wäre unmöglich. Deshalb prüfen wir im nächsten Schritt die Konfiguration.

2.5.4 Konfiguration überprüfen

```
ufw status verbose
```

Listing 22: Firewall-Status mit Details anzeigen

Erklärung:

- `status` – zeigt an, ob die Firewall aktiv ist und welche Regeln gelten
- `verbose` – erweiterte Ausgabe mit zusätzlichen Details wie Logging-Einstellungen und Standardrichtlinien

Die erwartete Ausgabe:

```
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip

To                         Action      From
--                         ------      ----
22/tcp                     ALLOW IN    Anywhere
80/tcp                     ALLOW IN    Anywhere
443/tcp                    ALLOW IN    Anywhere
22/tcp (v6)                ALLOW IN    Anywhere (v6)
80/tcp (v6)                ALLOW IN    Anywhere (v6)
443/tcp (v6)               ALLOW IN    Anywhere (v6)
```

Listing 23: Erwartete Firewall-Ausgabe (gekürzt)

Wichtige Details der Ausgabe:

- `Status: active` – die Firewall läuft und blockt unerwünschten Traffic
- `Default: deny (incoming)` – alle nicht explizit erlaubten eingehenden Verbindungen werden geblockt
- `Anywhere` – diese Ports sind von **jeder** IP-Adresse aus erreichbar (für Webseiten notwendig)
- `(v6)` – die Regeln gelten identisch für IPv6, sodass auch moderne Netzwerke geschützt sind

2.6 Zusammenfassung

Nach diesem Schritt ist die Firewall aktiv und schützt den Server:

- **65.532 Ports sind dicht** – nur 3 sind offen
- **SSH** (22) bleibt als einziger Verwaltungszugang offen
- **HTTP/HTTPS** (80/443) sind für die spätere Web-App vorbereitet
- **IPv4 und IPv6** werden beide geschützt
- Die Firewall startet automatisch bei jedem Server-Neustart

Praxis-Tipp: Mit dem Befehl `ufw status numbered` kannst du jederzeit alle Regeln mit Nummern anzeigen. Eine einzelne Regel löschst du dann mit `ufw delete [Nummer]`.

3 Docker-Images bauen und App deployen

In diesem Schritt bringen wir unsere Fitness-App vom lokalen Entwicklungsrechner auf den Server und machen sie weltweit erreichbar. Dafür nutzen wir **Docker** – eine Container-Plattform, die Anwendungen in standardisierten, isolierten Umgebungen verpackt und ausführt.

3.1 Was ist Docker und warum nutzen wir es?

Docker funktioniert wie ein **Versandkarton für Software**. Stell dir vor, du verschickst ein zerbrechliches Paket: Du packst es in einen genormten Karton, der überall auf der Welt gleich behandelt wird – egal ob in Deutschland, Japan oder Brasilien. Docker macht dasselbe mit Software:

- **Image** = Der Bauplan des Kartons (inkl. Inhalt). Ein Image enthält das Betriebssystem, alle Abhängigkeiten und die Anwendung selbst.
- **Container** = Der tatsächliche laufende Karton. Ein Container ist eine laufende Instanz eines Images.
- **Volume** = Ein separater Speicherort, der den Container überlebt. Wie ein externer USB-Stick, den man an den Karton anschließt.

Konkret für unser Projekt:

- `fitness-api:latest` – Image mit .NET 8 Backend
- `fitness-web:latest` – Image mit React Frontend + Nginx
- `fitness-data` – Volume für die SQLite-Datenbank (überlebt Container-Neustarts)

3.2 Die drei Dockerfiles im Detail

3.2.1 Backend-Dockerfile: `apps/api/Dockerfile`

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ["apps/api/Api.csproj", "apps/api/"]
RUN dotnet restore "apps/api/Api.csproj"
COPY . .
WORKDIR /src/apps/api
RUN dotnet publish "Api.csproj" -c Release -o /app/publish

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS final
WORKDIR /app
COPY --from=build /app/publish .
EXPOSE 5000
ENV ASPNETCORE_URLS=http://+:5000
ENTRYPOINT ["dotnet", "Api.dll"]
```

Listing 24: Dockerfile für das .NET Backend

Zeile für Zeile erklärt:

1. `FROM ... AS build` – Startet mit dem .NET 8 SDK Image (enthält Compiler, Tools). Der Alias `build` erlaubt später darauf zuzugreifen.

2. `WORKDIR /src` – Setzt das Arbeitsverzeichnis im Container auf `/src`.
3. `COPY [äpps/api/Api.csproj", äpps/api/"]` – Kopiert NUR die Projektdatei. Dadurch cached Docker diesen Schritt: Solange sich `Api.csproj` nicht ändert, wird der Cache verwendet → schnellere Builds!
4. `RUN dotnet restore` – Lädt alle NuGet-Pakete herunter (Entity Framework, NSwag, Swagger usw.).
5. `COPY . .` – Kopiert den gesamten restlichen Quellcode.
6. `WORKDIR /src/apps/api` – Wechselt ins Backend-Verzeichnis.
7. `RUN dotnet publish` – Kompiliert die Anwendung im Release-Modus in den Ordner `/app/publish`.
8. `FROM ... AS final` – Startet ein NEUES, schlankeres Image (nur ASP.NET Runtime, kein SDK). Das spart Speicher!
9. `COPY –from=build ...` – Kopiert die kompilierte Anwendung aus dem Build-Image.
10. `EXPOSE 5000` – Dokumentiert, dass der Container auf Port 5000 lauscht.
11. `ENV ASPNETCORE_URLS=http://+:5000` – Sagt .NET, es soll auf Port 5000 auf ALLEN Netzwerkschnittstellen lauschen.
12. `ENTRYPOINT ["dotnet", Äpi.dll"]` – Startet die Anwendung beim Container-Start.

3.2.2 Frontend-Dockerfile: `apps/web/Dockerfile`

```dockerfile
FROM node:22-alpine AS build
WORKDIR /app
COPY pnpm-lock.yaml pnpm-workspace.yaml package.json ./
COPY apps/web/package.json apps/web/
RUN npm install -g pnpm && pnpm install --no-frozen-lockfile
COPY apps/web/ apps/web/
WORKDIR /app/apps/web
RUN pnpm run build

FROM nginx:stable-alpine
COPY --from=build /app/apps/web/dist /usr/share/nginx/html
COPY apps/web/nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Listing 25: Dockerfile für das React Frontend

Zeile für Zeile erklärt:

1. `FROM node:22-alpine` – Leichtgewichtiges Node.js 22 Image (Alpine Linux = nur ∼5 MB statt ∼180 MB bei Ubuntu).
2. `COPY pnpm-lock.yaml pnpm-workspace.yaml package.json ./` – Kopiert die Monorepo-Konfiguration aus dem Root. `pnpm-workspace.yaml` ist nötig, damit pnpm die Workspace-Struktur erkennt.
3. `COPY apps/web/package.json apps/web/` – Kopiert die Frontend-Paketliste an ihren Platz.

4. `RUN npm install -g pnpm && pnpm install` – Installiert pnpm global und dann alle Abhängigkeiten.
5. `COPY apps/web/ apps/web/` – Kopiert den restlichen Frontend-Code.
6. `WORKDIR /app/apps/web` – Wechselt ins Frontend-Verzeichnis.
7. `RUN pnpm run build` – Baut das Frontend mit Vite (erzeugt `dist/`).
8. `FROM nginx:stable-alpine` – NEUES schlankes Image mit Nginx (Webserver).
9. `COPY –from=build ... /usr/share/nginx/html` – Kopiert den Build-Output in Nginx's Standard-Webverzeichnis.
10. `COPY apps/web/nginx.conf ...` – Unsere eigene Nginx-Konfiguration.
11. `EXPOSE 80` – HTTP-Port.
12. `CMD ["nginx", g", "daemon off;"]` – Startet Nginx im Vordergrund (Container bleibt am Leben).

3.2.3 Nginx-Konfiguration: `apps/web/nginx.conf`

```
server {
    listen 80;
    server_name localhost;
    root /usr/share/nginx/html;
    index index.html;

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

    location /api/ {
        proxy_pass http://fitness-api:5000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection keep-alive;
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Listing 26: Nginx-Konfiguration mit Reverse Proxy

Erklärung: Dies ist ein **Reverse Proxy**. Nginx nimmt alle Anfragen entgegen und entscheidet, wohin sie weitergeleitet werden:

- `location /` – Anfragen an die Hauptseite → liefert React-Dateien aus `/usr/share/nginx/html`
- `location /api/` – Anfragen an `/api/*` → leitet sie an das Backend (`fitness-api:5000`) weiter
- `try_files $uri $uri/ /index.html` – Sorgt dafür, dass Reacts Client-Side-Routing funktioniert (z.B. `/workouts/123` wird an React weitergegeben, nicht als 404 beantwortet)

3.3 Das Backend: Program.cs im Detail

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

var dbPath = Path.Combine("/app/data", "fitness.db");
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlite($"Data Source={dbPath}"));

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddOpenApiDocument();
builder.Services.AddSwaggerGen();

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    db.Database.EnsureCreated();
}

if (app.Environment.IsDevelopment())
{
    app.UseOpenApi();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.MapPost("/api/workouts", async (Workout workout, AppDbContext db) => { ...
    });
app.MapGet("/api/workouts", async (AppDbContext db) => ...);
app.MapGet("/api/workouts/{id:guid}", async (Guid id, AppDbContext db) => ...);
app.MapPut("/api/workouts/{id:guid}", async (Guid id, Workout input,
    AppDbContext db) => ...);
app.MapDelete("/api/workouts/{id:guid}", async (Guid id, AppDbContext db) =>
    ...);

app.MapGet("/", () => "H Fitness API läuft!");
app.Run();
```

Listing 27: Vollständige Program.cs

Die NuGet-Pakete in der .csproj-Datei:

- `Microsoft.EntityFrameworkCore.Sqlite` (8.0.14) – EF Core für SQLite
- `Microsoft.EntityFrameworkCore.Design` (8.0.14) – EF Core Tools für Migrationen
- `NSwag.AspNetCore` (14.1.0) – OpenAPI/Swagger-Generator
- `Swashbuckle.AspNetCore` (6.6.2) – Swagger UI (die schöne Oberfläche)

Wichtige Details:

- `Path.Combine("/app/data", "fitness.db")` – Im Docker-Container liegt die Datenbank im Volume-Ordner `/app/data`. Das stellt sicher, dass die Daten erhalten bleiben, auch wenn der Container gelöscht und neu erstellt wird.

- `db.Database.EnsureCreated()` – Erstellt die Datenbank-Tabellen automatisch beim Start, falls sie noch nicht existieren. Erspart uns manuelle Migrationen im Produktivbetrieb.
- Die CRUD-Endpunkte sind **Minimal API Endpoints** – .NET 8's leichtgewichtige Alternative zu Controllern.

3.4 Der API-Client: client.ts im Detail

```
const API_BASE = "";

export interface Workout {
  id?: string;
  name: string;
  date: string;
  durationMinutes: number;
  notes?: string;
}

export const fitnessApi = {
  async getWorkouts(): Promise<Workout[]> {
    const res = await fetch(`/api/workouts`);
    return res.json();
  },

  async createWorkout(workout: Workout): Promise<Workout> {
    const res = await fetch(`/api/workouts`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(workout),
    });
    return res.json();
  },

  // Weitere Methoden: getWorkout, updateWorkout, deleteWorkout...
};
```

Listing 28: Manueller API-Client

Erklärung:

- `API_BASE = ""` – Keine absolute URL! Stattdessen relative Pfade wie `/api/workouts`. Der Browser sendet die Anfrage dann an dieselbe Domain, auf der die Seite gehostet ist. Nginx leitet sie an das Backend weiter.
- `interface Workout` – TypeScript-Interface für Typsicherheit. Stellt sicher, dass wir keine falschen Felder an die API senden.
- `fetch()` – Native Browser-API für HTTP-Anfragen. Kein Axios, kein jQuery nötig!
- Die Methoden geben direkt das geparste JSON zurück.

3.5 Das Frontend: App.tsx im Detail

```
function App() {
  const [workouts, setWorkouts] = useState<Workout[]>([]);
  const [form, setForm] = useState<Workout>({ ... });

```

```
  const loadWorkouts = async () => {
    const data = await fitnessApi.getWorkouts();
    setWorkouts(data);
  };

  useEffect(() => { loadWorkouts(); }, []);

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    await fitnessApi.createWorkout({ ...form });
    loadWorkouts();
  };

  return (
    <div className="min-h-screen bg-gray-950 text-white p-4 max-w-md mx-auto">
      {/* Formular */}
      <form onSubmit={handleSubmit}>...</form>
      {/* Workout-Liste */}
      {workouts.map(w => ( ... ))}
    </div>
  );
}
```

Listing 29: Hauptkomponente mit Workout-Liste und Formular

Erklärung:

- `useState` – Reacts State-Management für die Workout-Liste und das Formular.
- `useEffect` – Lädt die Workouts einmalig beim ersten Rendern der Komponente.
- `handleSubmit` – Wird beim Absenden des Formulars aufgerufen. Verhindert den Standard-Seiten-Reload (`e.preventDefault()`), sendet das Workout an die API und lädt die Liste neu.
- Tailwind CSS-Klassen wie `bg-gray-950`, `text-white`, `p-4` gestalten die App im Dark-Mode.

3.6 Images bauen

Am Anfang war das Frontend-Image fehlerhaft. Hier die drei wichtigsten Fixes:

Fehler 1: `pnpm-lock.yaml` nicht gefunden

- Ursache: Dockerfile suchte in `apps/web/`, aber die Datei liegt im Root.
- Lösung: `COPY pnpm-lock.yaml ./` (vom Root kopieren).

Fehler 2: `–frozen-lockfile` schlug fehl

- Ursache: pnpm-Lockfile war nicht aktuell mit der Root-`package.json`.
- Lösung: `–no-frozen-lockfile` verwenden, damit pnpm fehlende Pakete nachinstalliert.

Fehler 3: TypeScript Compiler (`tsc`) nicht gefunden

- Ursache: `tsc -b` benötigt TypeScript als Abhängigkeit, die im Container fehlte.

- Lösung: Build-Script von `"tsc -b && vite build"` auf `"vite build"` geändert. Vite führt den TypeScript-Check beim Dev-Server durch – im Produktions-Build reicht die reine Vite-Kompilierung.

3.7 Images exportieren und auf den Server kopieren

```
# Images als tar-Datei speichern
docker save fitness-api:latest fitness-web:latest -o fitness-images.tar

# Auf den Server kopieren (scp = Secure Copy über SSH)
scp fitness-images.tar testserver:/root/

# Auf dem Server importieren
ssh testserver
docker load -i /root/fitness-images.tar
docker images | grep fitness
```

Listing 30: Images exportieren und kopieren

Befehle erklärt:

- `docker save` – Exportiert Docker-Images in eine portable tar-Datei.
- `scp` – Secure Copy: Kopiert Dateien verschlüsselt über SSH.
- `testserver:/root/` – Der Alias aus unserer `~/.ssh/config`. Die Datei landet im `/root/`-Verzeichnis des Servers.
- `docker load -i` – Importiert Images aus einer tar-Datei in Docker.
- `docker images` – Listet alle lokal verfügbaren Docker-Images auf.

3.8 Container auf dem Server starten

```
# Volume für die Datenbank (überlebt Container-Neustarts)
docker volume create fitness-data

# Netzwerk (damit Backend und Frontend kommunizieren können)
docker network create fitness-net

# Backend starten
docker run -d \
  --name fitness-api \
  --network fitness-net \
  -v fitness-data:/app/data \
  -p 5000:5000 \
  fitness-api:latest

# Frontend starten
docker run -d \
  --name fitness-web \
  --network fitness-net \
  -p 80:80 \
  fitness-web:latest
```

Listing 31: Docker-Netzwerk, Volume und Container anlegen

Optionen erklärt:

- `-d` – Detached Mode: Container läuft im Hintergrund (gibt die Konsole frei).
- `–name fitness-api` – Gibt dem Container einen festen Namen (sonst vergibt Docker Zufallsnamen).
- `–network fitness-net` – Bindet den Container in unser Docker-Netzwerk ein. Container im selben Netzwerk können sich über ihren Namen erreichen (z.B. `fitness-api`).
- `-v fitness-data:/app/data` – Bindet das Volume `fitness-data` in den Container-Pfad `/app/data` ein. Alles, was im Container unter `/app/data` gespeichert wird, landet tatsächlich im Volume und überlebt.
- `-p 5000:5000` – Port-Mapping: Leitet Port 5000 des Hosts (Server) an Port 5000 des Containers weiter.

3.9 Aufgetretene Probleme und Lösungen

Problem 1: Nginx konnte Host "backend"nicht auflösen

- Ursache: In `nginx.conf` stand `proxy_pass http://backend:5000`, aber der Backend-Container heißt `fitness-api`.
- Lösung: `backend` → `fitness-api` in `nginx.conf` ändern, Image neu bauen.

Problem 2: Frontend lud Assets mit Pfad `/app/...`

- Ursache: In `vite.config.ts` war `base: /app/"` für PWA-Zwecke gesetzt.
- Lösung: Geändert auf `base: /"`.

Problem 3: API-Anfragen gingen an lokale IP `192.168.178.189`

- Ursache: Im Client stand `const API_BASE = "http://192.168.178.189:5107"`.
- Lösung: Geändert auf relative Pfade (`/api/workouts`), sodass Nginx die Anfragen per Reverse Proxy ans Backend weiterleitet.

Problem 4: Doppeltes `/api` im Pfad

- Ursache: Client hatte `API_BASE = /api"` und Endpunkte begannen ebenfalls mit `/api`.
- Ergebnis: Anfragen gingen an `/api/api/workouts`.
- Lösung: API-Base auf leeren String gesetzt und Endpunkte mit `/api/` beginnen lassen.

3.10 Zusammenfassung

Nach diesem Schritt ist die Fitness-App produktiv auf dem VPS im Einsatz:

- Die App ist unter `http://185.209.229.167` weltweit erreichbar
- Workouts werden persistent in einer SQLite-Datenbank gespeichert
- Docker-Volumes stellen sicher, dass Daten Container-Neustarts überleben
- Das Backend läuft auf .NET 8, das Frontend auf Nginx
- Ein Reverse Proxy (Nginx) leitet API-Anfragen intern an das Backend weiter