

# **Todo App**

**Eine Lern-Anwendung für Softwarearchitektur**

**Vom Quick & Dirty zur Clean Architecture**

Robert Bretz

9. Mai 2026

# Inhaltsverzeichnis

<b>1</b>	<b>Server-Absicherung (Ubuntu 24.04 auf Contabo VPS)</b>	<b>5</b>
1.1	Schritt 1: SSH-Verbindung testen . . . . .	5
1.2	Schritt 2: System-Updates . . . . .	6
1.3	Schritt 3: SSH-Key-Authentifizierung . . . . .	6
1.4	Schritt 4: SSH-Client-Konfiguration (Alias) . . . . .	7
1.5	Schritt 5: SSH-Timeout auf 20 Minuten . . . . .	8
1.6	Schritt 6: Fail2Ban (Bruteforce-Schutz) . . . . .	8
1.7	Zusammenfassung . . . . .	9
<b>2</b>	<b>Firewall mit UFW einrichten</b>	<b>9</b>
2.1	Was ist eine Firewall und warum brauchen wir sie? . . . . .	10
2.2	Die 65.535 Ports: Ein kurzer Überblick . . . . .	10
2.3	Die drei Ports, die wir öffnen . . . . .	10
2.4	Warum HTTPS für PWAs Pflicht ist . . . . .	10
2.5	Durchführung . . . . .	11
2.5.1	Standardrichtlinien setzen . . . . .	11
2.5.2	Benötigte Ports öffnen . . . . .	12
2.5.3	Firewall aktivieren . . . . .	12
2.5.4	Konfiguration überprüfen . . . . .	13
2.6	Zusammenfassung . . . . .	13
<b>3</b>	<b>Docker-Images bauen und App deployen</b>	<b>14</b>
3.1	Was ist Docker und warum nutzen wir es? . . . . .	14
3.2	Die drei Dockerfiles im Detail . . . . .	14
3.2.1	Backend-Dockerfile: apps/api/Dockerfile . . . . .	14
3.2.2	Frontend-Dockerfile: apps/web/Dockerfile . . . . .	15
3.2.3	Nginx-Konfiguration: apps/web/nginx.conf . . . . .	16
3.3	Das Backend: Program.cs im Detail . . . . .	17
3.4	Der API-Client: client.ts im Detail . . . . .	18
3.5	Das Frontend: App.tsx im Detail . . . . .	18
3.6	Images bauen . . . . .	19
3.7	Images exportieren und auf den Server kopieren . . . . .	20
3.8	Container auf dem Server starten . . . . .	20
3.9	Aufgetretene Probleme und Lösungen . . . . .	21
3.10	Zusammenfassung . . . . .	22
<b>4</b>	<b>Domain kaufen und DNS konfigurieren</b>	<b>22</b>
4.1	Warum eine eigene Domain? . . . . .	22
4.2	Grundlagen: Wie funktioniert das DNS? . . . . .	22
4.3	Was ist DNS-Propagation? . . . . .	23
4.4	Domain-Kauf bei Contabo . . . . .	23
4.4.1	Schritt 1: Einloggen ins Kundencenter . . . . .	23
4.4.2	Schritt 2: Domain bestellen . . . . .	23
4.4.3	Schritt 3: Domain-Handles konfigurieren . . . . .	24
4.4.4	Schritt 4: Nameserver festlegen . . . . .	24
4.4.5	Schritt 5: IP-Adresse auswählen . . . . .	24
4.4.6	Schritt 6: Bestellung abschließen . . . . .	24

4.5	DNS-Einträge für die Domain einrichten	24
4.5.1	Schritt 1: DNS Zone Management öffnen	24
4.5.2	Schritt 2: Notwendige A-Records anlegen	25
4.6	DNS-Propagation prüfen und beschleunigen	25
4.6.1	Direkt beim Nameserver prüfen	25
4.6.2	Lokalen DNS-Cache leeren	26
4.6.3	Server direkt über IP prüfen	26
4.6.4	Online DNS-Checker verwenden	26
4.7	Zusammenfassung	26
<b>5</b>	<b>HTTPS mit nginx-proxy und Let's Encrypt</b>	<b>27</b>
5.1	Warum brauchen wir HTTPS?	27
5.2	Wie funktioniert das SSL-Zertifikat von Let's Encrypt?	27
5.3	DNS-Propagation: Wie lange dauert es?	28
5.4	Architektur: Wie hängen die Container zusammen?	28
5.5	Schritt-für-Schritt: HTTPS einrichten	29
5.5.1	Schritt 1: Bestehende Container stoppen und löschen	29
5.5.2	Schritt 2: Verzeichnis für docker-compose anlegen	29
5.5.3	Schritt 3: docker-compose.yml erstellen	29
5.5.4	Schritt 4: Container starten	32
5.5.5	Schritt 5: Status prüfen	32
5.5.6	Schritt 6: Logs des acme-companion prüfen	32
5.6	Häufige Fehler und ihre Behebung	32
5.6.1	Fehler 1: "can't get nginx-proxy container ID"	32
5.6.2	Fehler 2: "contact email has forbidden domain"	32
5.6.3	Fehler 3: "DNS problem: NXDOMAIN"	33
5.7	Wie füge ich später weitere Subdomains hinzu?	33
5.8	Wie funktioniert der Reverse Proxy im Detail?	33
5.9	Zusammenfassung	34
<b>6</b>	<b>Gitea Installation und Server-Übersicht</b>	<b>34</b>
6.1	Warum ein eigener Git-Server?	34
6.2	OneDev: Der gescheiterte Versuch	34
6.2.1	Installationsversuch mit Docker (Version 1dev/server:latest)	35
6.3	Gitea: Die schlanke Alternative	35
6.4	Gitea mit Docker installieren	36
6.4.1	Schritt 1: OneDev rückstandslos entfernen	36
6.4.2	Schritt 2: Gitea-Verzeichnis und docker-compose.yml anlegen	36
6.4.3	Schritt 3: Container starten	37
6.4.4	Schritt 4: Firewall öffnen	37
6.4.5	Schritt 5: Gitea im Browser einrichten	37
6.5	Vollständige Server-Übersicht	38
6.5.1	Laufende Docker-Container	38
6.5.2	Docker-Volumes (persistente Datenspeicher)	38
6.5.3	Firewall (nur diese Ports sind offen!)	38
6.5.4	Installierte Systempakete	39
6.5.5	Verzeichnisstruktur unter /opt	39
6.6	Wie Gitea in die Infrastruktur passt	39
6.7	Zusammenfassung	40

<b>7</b>	<b>CI/CD mit Gitea Actions</b>	<b>40</b>
7.1	Docker-Grundlagen: Container verstehen und verwalten . . . . .	40
7.1.1	Container auflisten . . . . .	40
7.1.2	Container-Logs anzeigen . . . . .	41
7.1.3	Container stoppen, starten, neustarten . . . . .	42
7.1.4	In einen Container einsteigen . . . . .	42
7.1.5	Images verwalten . . . . .	42
7.1.6	Netzwerke inspizieren . . . . .	43
7.2	CI/CD-Pipeline mit Gitea Actions einrichten . . . . .	43
7.2.1	Was ist CI/CD? . . . . .	43
7.2.2	Der Gitea Act Runner . . . . .	43
7.2.3	Die Workflow-Datei . . . . .	44
7.3	Alle aufgetretenen Probleme und ihre Lösungen . . . . .	44
7.3.1	Problem 1: Docker-Socket doppelt gemountet . . . . .	44
7.3.2	Problem 2: Pfade zu Dockerfiles . . . . .	45
7.3.3	Problem 3: ERR_PNPM_IGNORED_BUILDS . . . . .	45
7.3.4	Problem 4: Fehlende nginx.conf . . . . .	45
7.3.5	Problem 5: Container ohne Port-Mapping . . . . .	45
7.4	Tutorial: Einfache HTML-Seite deployen . . . . .	45
7.4.1	Schritt 1: Projekt erstellen . . . . .	45
7.4.2	Schritt 2: index.html erstellen . . . . .	46
7.4.3	Schritt 3: Dockerfile erstellen . . . . .	46
7.4.4	Schritt 4: Workflow erstellen . . . . .	46
7.4.5	Schritt 5: In Gitea pushen . . . . .	47
7.4.6	Schritt 6: Firewall öffnen und testen . . . . .	47
7.5	Docker-Befehle Cheat Sheet . . . . .	47
7.6	Zusammenfassung . . . . .	47



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:

```
1 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):**

```
1 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 unterschreibt sie 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:**

```
1 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:

```
1 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:

```
1 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:

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

Listing 8: SSH-Konfiguration bearbeiten

Folgenden Inhalt einfügen:

```
1 Host testserver
2     HostName 185.209.229.167
3     User root
4     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:

```
1 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):**

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

Listing 11: SSH-Server-Konfiguration bearbeiten

Folgende Zeilen suchen oder am Ende der Datei einfügen:

```
1 ClientAliveInterval 120
2 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:  $120 \text{ Sekunden} \times 10 = 1200 \text{ Sekunden} = 20 \text{ Minuten}$ . **SSH-Dienst neustarten:**

```
1 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:



```
1 bantime = 600
2 findtime = 600
3 maxretry = 3
4
5 [sshd]
6 enabled = true
```

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

### Installation auf dem Server:

```
1 apt install -y fail2ban
```

Listing 15: Fail2Ban installieren

### Automatischen Start aktivieren und sofort starten:

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

Listing 16: Fail2Ban aktivieren und starten

### Status prüfen:

```
1 systemctl status fail2ban
```

Listing 17: Fail2Ban-Status abfragen

Die Ausgabe sollte active (running) zeigen.

```
1 - fail2ban.service - Fail2Ban Service
2   Active: active (running)
3   ...
4   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 "nacktim 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 <b>einzigster Verwaltungszugang</b> . Ohne Port 22 könnten wir den Server nicht mehr fernsteuern – wir wären ausgesperrt.
80	HTTP	<b>Standard-Webport</b> für alle Browser. Wenn jemand deine Domain aufruft, landet er zuerst hier. Leitet später automatisch auf HTTPS (Port 443) um.
443	HTTPS	<b>Verschlüsselte Webseiten</b> . 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:**

```
1 ufw default deny incoming
2 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, wärest 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

```
1 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:

```
1 Status: active
2 Logging: on (low)
3 Default: deny (incoming), allow (outgoing), disabled (routed)
4 New profiles: skip
5
6 To                Action            From
7 --                -
8 22/tcp            ALLOW IN          Anywhere
9 80/tcp            ALLOW IN          Anywhere
10 443/tcp           ALLOW IN          Anywhere
11 22/tcp (v6)       ALLOW IN          Anywhere (v6)
12 80/tcp (v6)       ALLOW IN          Anywhere (v6)
13 443/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`

```
1 FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
2 WORKDIR /src
3 COPY ["apps/api/Api.csproj", "apps/api/"]
4 RUN dotnet restore "apps/api/Api.csproj"
5 COPY . .
6 WORKDIR /src/apps/api
7 RUN dotnet publish "Api.csproj" -c Release -o /app/publish
8
9 FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS final
10 WORKDIR /app
11 COPY --from=build /app/publish .
12 EXPOSE 5000
13 ENV ASPNETCORE_URLS=http://+:5000
14 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 [apps/api/Api.csproj", apps/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, NS-wag, 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", "Api.dll"]` – Startet die Anwendung beim Container-Start.

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

```

1 FROM node:22-alpine AS build
2 WORKDIR /app
3 COPY pnpm-lock.yaml pnpm-workspace.yaml package.json ./
4 COPY apps/web/package.json apps/web/
5 RUN npm install -g pnpm && pnpm install --no-frozen-lockfile
6 COPY apps/web/ apps/web/
7 WORKDIR /app/apps/web
8 RUN pnpm run build
9
10 FROM nginx:stable-alpine
11 COPY --from=build /app/apps/web/dist /usr/share/nginx/html
12 COPY apps/web/nginx.conf /etc/nginx/conf.d/default.conf
13 EXPOSE 80
14 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`

```
1 server {
2     listen 80;
3     server_name localhost;
4     root /usr/share/nginx/html;
5     index index.html;
6
7     location / {
8         try_files $uri $uri/ /index.html;
9     }
10
11     location /api/ {
12         proxy_pass http://fitness-api:5000;
13         proxy_http_version 1.1;
14         proxy_set_header Upgrade $http_upgrade;
15         proxy_set_header Connection keep-alive;
16         proxy_set_header Host $host;
17         proxy_cache_bypass $http_upgrade;
18     }
19 }
```

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

```
1 using Microsoft.EntityFrameworkCore;
2
3 var builder = WebApplication.CreateBuilder(args);
4
5 var dbPath = Path.Combine("/app/data", "fitness.db");
6 builder.Services.AddDbContext<AppDbContext>(options =>
7     options.UseSqlite($"Data Source={dbPath}"));
8
9 builder.Services.AddEndpointsApiExplorer();
10 builder.Services.AddOpenApiDocument();
11 builder.Services.AddSwaggerGen();
12
13 var app = builder.Build();
14
15 using (var scope = app.Services.CreateScope())
16 {
17     var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
18     db.Database.EnsureCreated();
19 }
20
21 if (app.Environment.IsDevelopment())
22 {
23     app.UseOpenApi();
24     app.UseSwaggerUI();
25 }
26
27 app.UseHttpsRedirection();
28
29 app.MapPost("/api/workouts", async (Workout workout, AppDbContext db) => { ...
30     });
31 app.MapGet("/api/workouts", async (AppDbContext db) => ...);
32 app.MapGet("/api/workouts/{id:guid}", async (Guid id, AppDbContext db) => ...);
33 app.MapPut("/api/workouts/{id:guid}", async (Guid id, Workout input,
34     AppDbContext db) => ...);
35 app.MapDelete("/api/workouts/{id:guid}", async (Guid id, AppDbContext db) =>
36     ...);
37
38 app.MapGet("/", () => "H Fitness API läuft!");
39 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

```

1 const API_BASE = "";
2
3 export interface Workout {
4   id?: string;
5   name: string;
6   date: string;
7   durationMinutes: number;
8   notes?: string;
9 }
10
11 export const fitnessApi = {
12   async getWorkouts(): Promise<Workout[]> {
13     const res = await fetch('/api/workouts');
14     return res.json();
15   },
16
17   async createWorkout(workout: Workout): Promise<Workout> {
18     const res = await fetch('/api/workouts', {
19       method: "POST",
20       headers: { "Content-Type": "application/json" },
21       body: JSON.stringify(workout),
22     });
23     return res.json();
24   },
25
26   // Weitere Methoden: getWorkout, updateWorkout, deleteWorkout...
27 };

```

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

```

1 function App() {
2   const [workouts, setWorkouts] = useState<Workout[]>([]);
3   const [form, setForm] = useState<Workout>({ ... });
4

```

```

5  const loadWorkouts = async () => {
6    const data = await fitnessApi.getWorkouts();
7    setWorkouts(data);
8  };
9
10 useEffect(() => { loadWorkouts(); }, []);
11
12 const handleSubmit = async (e: React.FormEvent) => {
13   e.preventDefault();
14   await fitnessApi.createWorkout({ ...form });
15   loadWorkouts();
16 };
17
18 return (
19   <div className="min-h-screen bg-gray-950 text-white p-4 max-w-md mx-auto">
20     /* Formular */
21     <form onSubmit={handleSubmit}>...</form>
22     /* Workout-Liste */
23     {workouts.map(w => ( ... ))}
24   </div>
25 );
26 }

```

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

```

1 # Images als tar-Datei speichern
2 docker save fitness-api:latest fitness-web:latest -o fitness-images.tar
3
4 # Auf den Server kopieren (scp = Secure Copy über SSH)
5 scp fitness-images.tar testserver:/root/
6
7 # Auf dem Server importieren
8 ssh testserver
9 docker load -i /root/fitness-images.tar
10 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

```

1 # Volume für die Datenbank (überlebt Container-Neustarts)
2 docker volume create fitness-data
3
4 # Netzwerk (damit Backend und Frontend kommunizieren können)
5 docker network create fitness-net
6
7 # Backend starten
8 docker run -d \
9   --name fitness-api \
10  --network fitness-net \
11  -v fitness-data:/app/data \
12  -p 5000:5000 \
13  fitness-api:latest
14
15 # Frontend starten
16 docker run -d \
17   --name fitness-web \
18   --network fitness-net \
19   -p 80:80 \
20   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

## 4 Domain kaufen und DNS konfigurieren

In diesem Schritt kaufen wir eine eigene Domain und verknüpfen sie mit unserem Server, damit die Fitness-App unter einem eigenen Namen (nicht nur der IP-Adresse) weltweit erreichbar ist. Wir machen das direkt bei unserem Server-Anbieter Contabo – das spart Verwaltungsaufwand, weil alles unter einem Dach bleibt.

### 4.1 Warum eine eigene Domain?

Bisher ist unsere App unter `http://185.209.229.167` erreichbar. Das hat mehrere Nachteile:

- **Schwer zu merken:** Niemand kann sich IP-Adressen merken.
- **Kein HTTPS:** Für ein SSL-Zertifikat braucht man eine Domain. Ohne HTTPS keine PWA-Installation!
- **Unprofessionell:** Eine eigene Domain wirkt seriös und vertrauenswürdig.
- **Flexibel:** Wenn du später den Server wechselst, änderst du einfach den DNS-Eintrag. Die Domain bleibt gleich.

### 4.2 Grundlagen: Wie funktioniert das DNS?

DNS steht für **Domain Name System**. Es ist das "Telefonbuch des Internets" und übersetzt menschenlesbare Domain-Namen in maschinenlesbare IP-Adressen.

**Die wichtigsten Record-Typen:**

**Wie eine DNS-Auflösung abläuft:**

1. Du gibst `robre.de` in den Browser ein.
2. Dein Rechner fragt seinen DNS-Resolver (meist bei deinem Internet-Provider): "Welche IP hat `robre.de`?"
3. Der Resolver fragt die Root-Nameserver, dann die `.de`-Nameserver, dann Contabos Nameserver.
4. Contabo antwortet: `robre.de = 185.209.229.167`
5. Der Browser stellt eine HTTP-Verbindung zu dieser IP her.

Tabelle 3: DNS-Record-Typen und ihre Funktion

Typ	Name	Funktion
A	Address Record	Verbindet eine Domain mit einer IPv4-Adresse. "robre.de → 185.209.229.167"
AAAA	IPv6 Address Record	Wie A-Record, aber für IPv6-Adressen
CNAME	Canonical Name	Verweist eine Domain auf eine andere Domain (Alias)
MX	Mail Exchange	Legt fest, welcher Server E-Mails für die Domain empfängt
NS	Name Server	Definiert, welche Nameserver für die Domain zuständig sind
TXT	Text Record	Enthält beliebige Textinformationen (z. B. für SPF, DKIM)

6. Deine App erscheint!

### 4.3 Was ist DNS-Propagation?

Wenn du DNS-Einträge änderst, dauert es eine Weile, bis alle DNS-Server weltweit die neuen Informationen haben. Das nennt man **Propagation** (Verbreitung).

- **Dauer:** Meist 15 Minuten bis 2 Stunden, in seltenen Fällen bis zu 48 Stunden.
- **Grund:** Jeder DNS-Resolver hat einen Cache (Zwischenspeicher) mit alten Einträgen. Erst wenn der Cache abläuft (TTL = Time To Live), wird der aktuelle Wert abgefragt.
- **TTL-Wert:** Unsere Einträge haben TTL 86400 = 24 Stunden. Deshalb kann es länger dauern, bis alte Caches verfallen sind.

**Tipp:** Mit dem Befehl `nslookup robre.de ns1.contabo.net` fragst du **direkt** bei Contabos Nameserver an – ohne Cache. So siehst du sofort, ob die Konfiguration stimmt, auch wenn dein lokaler DNS die Domain noch nicht kennt.

## 4.4 Domain-Kauf bei Contabo

### 4.4.1 Schritt 1: Einloggen ins Kundencenter

Unter <https://contabo.com> mit deinen Zugangsdaten anmelden.

### 4.4.2 Schritt 2: Domain bestellen

1. In der linken Seitenleiste auf **"Domains"** klicken.
2. Auf den blauen Button **"Domain bestellen"** klicken.
3. Wunschnamen eingeben (z. B. "robre") und Endung auswählen (.de, .com, .net, etc.).

4. Auf "Weiter" klicken.

#### 4.4.3 Schritt 3: Domain-Handles konfigurieren

Auf der nächsten Seite werden die sogenannten **Domain-Handles** (Kontaktdaten) abgefragt:

- **Owner / Besitzer:** Der rechtmäßige Eigentümer der Domain. Das sollte immer eine Privatperson oder Firma sein – NIEMALS der Provider!
- **Admin:** Der administrative Ansprechpartner. Meist dieselbe Person wie der Owner.
- **Tech:** Der technische Ansprechpartner. Hier kann der Provider stehen (Contabo GmbH).
- **Zone:** Zonenverwaltung. Ebenfalls Contabo GmbH.

**Standardwerte übernehmen:** Für Tech und Zone einfach die vorausgefüllte Contabo GmbH lassen. Das vereinfacht die technische Verwaltung.

#### 4.4.4 Schritt 4: Nameserver festlegen

Auf der nächsten Seite wählst du die Nameserver:

- **Contabo Standard-Nameserver** auswählen (ns1.contabo.net, ns2.contabo.net, ns3.contabo.net).
- Das bedeutet: Contabo verwaltet die DNS-Einträge für dich. Du kannst sie jederzeit im Kundencenter ändern.

#### 4.4.5 Schritt 5: IP-Adresse auswählen

Hier wählst du aus, auf welchen deiner Server die Domain zeigen soll. In unserem Fall:

- Server **"test"** mit IP 185.209.229.167
- Server **"prod"** mit IP 185.209.230.235

**Empfehlung:** Nur die erste IP (185.209.229.167) auswählen, da dort unsere App läuft. Man kann später weitere Einträge hinzufügen.

#### 4.4.6 Schritt 6: Bestellung abschließen

Die Zusammenfassung prüfen und auf **"Bestellen & Bezahlen"** klicken.

### 4.5 DNS-Einträge für die Domain einrichten

Nach dem Kauf muss die Domain noch mit unserer Server-IP verknüpft werden. Das machen wir im **DNS Zone Management**.

#### 4.5.1 Schritt 1: DNS Zone Management öffnen

Im Contabo Control Panel: **Netzwerkdienste** → **DNS-Verwaltung** oder direkter: **Netzwerkdienste** → **"DNS-Zone für [deine Domain] bearbeiten"**.



## 4.5.2 Schritt 2: Notwendige A-Records anlegen

Wir brauchen drei A-Records:

### 1. Hauptdomain ohne Subdomain:

- Name: `robre.de`
- Typ: A
- Data: `185.209.229.167`
- TTL: `86400`

### 2. www-Subdomain:

- Name: `www.robre.de`
- Typ: A
- Data: `185.209.229.167`
- TTL: `86400`

### 3. Wildcard für alle anderen Subdomains (optional, aber nützlich):

- Name: `*.robre.de`
- Typ: A
- Data: `185.209.229.167`
- TTL: `86400`

### Warum alle drei?

- `robre.de` – die Hauptdomain, die die meisten Nutzer eingeben.
- `www.robre.de` – viele Nutzer geben aus Gewohnheit "www.ëin.
- `*.robre.de` – fängt alle zukünftigen Subdomains ab (z. B. `app.robre.de`, `api.robre.de`), ohne dass wir jedes Mal einen neuen Eintrag anlegen müssen.

## 4.6 DNS-Propagation prüfen und beschleunigen

### 4.6.1 Direkt beim Nameserver prüfen

Dieser Befehl fragt **direkt** Contabos Nameserver an – ohne Cache. Wenn hier die richtige IP erscheint, ist die Konfiguration korrekt und wir müssen nur auf die weltweite Verbreitung warten.

```
1 nslookup robre.de ns1.contabo.net
```

Listing 32: DNS direkt bei Contabo abfragen

### Ausgabe bei korrekter Konfiguration:

```
1 Server: ns1.contabo.net
2 Address: 2a02:c207:ff00:1200::1#53
3
4 Name: robre.de
5 Address: 185.209.229.167
```

Listing 33: Erwartete Ausgabe

#### 4.6.2 Lokalen DNS-Cache leeren

Manchmal hat dein Rechner noch alte DNS-Einträge im Cache. So löschst du ihn:

```
1 sudo resolvectl flush-caches
```

Listing 34: DNS-Cache leeren unter Linux mit systemd-resolved

#### 4.6.3 Server direkt über IP prüfen

Um sicherzugehen, dass der Server selbst erreichbar ist (unabhängig von DNS), testen wir mit curl:

```
1 curl -I http://185.209.229.167
```

Listing 35: Server direkt über IP testen

#### Erwartete Ausgabe:

```
1 HTTP/1.1 200 OK
2 Server: nginx/1.30.0
3 Content-Type: text/html
4 ...
```

Listing 36: Server antwortet korrekt

HTTP/1.1 200 OK bedeutet: Der Server läuft, die App ist erreichbar. Das Problem liegt ausschließlich an der DNS-Verbreitung.

#### 4.6.4 Online DNS-Checker verwenden

Websites wie <https://dnschecker.org> oder <https://whatsmydns.net> zeigen an, von welchen Standorten weltweit die Domain bereits aufgelöst wird. Praktisches Werkzeug, um den Fortschritt der Propagation zu verfolgen!

### 4.7 Zusammenfassung

Nach diesem Schritt haben wir:

- Eine eigene Domain `robre.de` für 11,88 € / Jahr
- DNS-Einträge, die auf unsere Server-IP `185.209.229.167` zeigen
- Geprüft, dass Contabo die Domain korrekt registriert hat
- Geprüft, dass der Server direkt über die IP erreichbar ist
- DNS-Propagation abwarten, bis die Domain weltweit funktioniert

Als nächstes folgt die Einrichtung von HTTPS mit einem kostenlosen SSL-Zertifikat – das ist die Voraussetzung für die PWA-Installation!

## 5 HTTPS mit nginx-proxy und Let's Encrypt

In diesem Schritt richten wir **HTTPS** für unsere Domain ein. Dafür nutzen wir zwei Docker-Container: nginx-proxy als Reverse Proxy und acme-companion für automatische SSL-Zertifikate von Let's Encrypt. Nach diesem Schritt ist unsere App unter `https://robre.de` erreichbar und erfüllt alle Voraussetzungen für die PWA-Installation.

### 5.1 Warum brauchen wir HTTPS?

HTTP (Hypertext Transfer Protocol) sendet alle Daten im **Klartext**. Ein Angreifer im gleichen Netzwerk kann mitlesen, manipulieren oder die Verbindung kapern. HTTPS (HTTP Secure) verschlüsselt die gesamte Kommunikation zwischen Browser und Server mittels TLS (Transport Layer Security).

**Für Progressive Web Apps ist HTTPS zwingend vorgeschrieben:**

- Der **Service Worker** – das Herzstück einer PWA – funktioniert nur mit HTTPS
- Browser verweigern die PWA-Installation bei unverschlüsselten Verbindungen
- Moderne Browser-APIs (Geolocation, Kamera, Push-Benachrichtigungen) erfordern HTTPS

### 5.2 Wie funktioniert das SSL-Zertifikat von Let's Encrypt?

**Let's Encrypt** ist eine kostenlose Zertifizierungsstelle (Certificate Authority). Der Ablauf der Zertifikatserstellung läuft automatisch nach dem **ACME-Protokoll** (Automatic Certificate Management Environment):

1. Der acme-companion fordert ein Zertifikat für `robre.de` an
2. Let's Encrypt stellt eine **Challenge** – eine Aufgabe, die beweist, dass du die Domain kontrollierst
3. Der acme-companion legt eine temporäre Datei auf deinem Webserver ab
4. Let's Encrypt prüft, ob diese Datei unter `http://robre.de/.well-known/acme-challenge/...` erreichbar ist
5. Wenn ja: Zertifikat wird ausgestellt und automatisch in den nginx eingebunden
6. Das Zertifikat ist 90 Tage gültig und wird automatisch erneuert

**Wichtige Voraussetzung:** Die Domain muss weltweit über DNS erreichbar sein (**DNS-Propagation**). Ohne funktionierende DNS-Auflösung schlägt die Challenge fehl – Let's Encrypt kann die Domain nicht finden und verweigert die Zertifikatsausstellung.

## 5.3 DNS-Propagation: Wie lange dauert es?

Wenn du eine neue Domain registrierst oder DNS-Einträge änderst, dauert es eine Weile, bis alle DNS-Server weltweit die neuen Informationen kennen. Diesen Prozess nennt man **DNS-Propagation**.

- **Normale Dauer:** 15 Minuten bis 2 Stunden
- **Maximale Dauer:** Bis zu 48 Stunden (in seltenen Fällen)
- **Beeinflussender Faktor:** Der **TTL-Wert** (Time To Live) der DNS-Einträge. Unser Wert von 86400 Sekunden (24 Stunden) erlaubt anderen DNS-Servern, die Information bis zu 24 Stunden zwischenspeichern.

### So prüfst du den Status der Propagation:

```
1 # Über deinen Standard-DNS-Resolver
2 nslookup robre.de
3
4 # Direkt bei Contabos Nameserver (ohne Cache)
5 nslookup robre.de ns1.contabo.net
6
7 # Über Googles öffentlichen DNS (8.8.8.8)
8 nslookup robre.de 8.8.8.8
```

Listing 37: DNS-Auflösung lokal prüfen

Erscheint als Antwort 185.209.229.167, ist die Propagation für diesen Server abgeschlossen. Erscheint NXDOMAIN (Non-Existent Domain), kennt der DNS-Server die Domain noch nicht.

**Online-Tool zur weltweiten Prüfung:** <https://dnschecker.org> zeigt auf einer Weltkarte, an welchen Standorten die Domain bereits aufgelöst wird.

## 5.4 Architektur: Wie hängen die Container zusammen?

Nach diesem Schritt sieht unsere Docker-Infrastruktur so aus:

```
Internet
|
| HTTPS (Port 443)
v
nginx-proxy (Reverse Proxy)
|
+-- robre.de --> fitness-web (Port 80)
|
|                               | /api/* --> fitness-api (Port 5000)
|
+-- SSL-Zertifikate von acme-companion verwaltet
```

### Erklärung der Komponenten:

- `nginx-proxy`: Empfängt alle HTTP/HTTPS-Anfragen und leitet sie an den richtigen Container weiter. Entscheidet anhand des Host-Headers (enthält die Domain), welcher Container die Anfrage bekommt.
- `acme-companion`: Überwacht laufende Container auf die Umgebungsvariable `LETSENCRYPT_HOST`. Wenn eine neue Domain auftaucht, fordert er automatisch ein SSL-Zertifikat an und konfiguriert `nginx`.
- `fitness-web`: Unser React-Frontend mit `Nginx`. Braucht selbst kein SSL – der Proxy übernimmt die Verschlüsselung.
- `fitness-api`: Unser .NET-Backend. Nur über das interne Docker-Netzwerk erreichbar, nicht direkt von außen.

## 5.5 Schritt-für-Schritt: HTTPS einrichten

### 5.5.1 Schritt 1: Bestehende Container stoppen und löschen

Da wir von manuellen `docker run`-Befehlen auf `docker compose` umsteigen, müssen die alten Container entfernt werden.

```
1 ssh testserver
2 docker stop fitness-web fitness-api
3 docker rm fitness-web fitness-api
```

Listing 38: Alte Container stoppen und löschen

### 5.5.2 Schritt 2: Verzeichnis für docker-compose anlegen

```
1 mkdir -p /opt/fitness
2 cd /opt/fitness
```

Listing 39: Projektverzeichnis auf dem Server

### 5.5.3 Schritt 3: docker-compose.yml erstellen

`docker compose` ist ein Tool, mit dem wir mehrere Container als **eine Einheit** definieren und verwalten können. Statt vier einzelner `docker run`-Befehle definieren wir alle Container in einer YAML-Datei.

Erstelle die Datei mit `nano /opt/fitness/docker-compose.yml`:

```
1 services
2   # =====
3   # REVERSE PROXY (nimmt HTTP/HTTPS-Anfragen entgegen)
4   # =====
5   nginx-proxy
6     image nginxproxy/nginx-proxy
7     container_name nginx-proxy
8     restart unless-stopped
9     ports
10      - '8080'
11      - '443443'
12     volumes
13      - /var/run/docker.sock:/tmp/docker.sockro
14      - certs/etc/nginx/certs
15      - vhost/etc/nginx/vhost.d
```

```

16     - html/usr/share/nginx/html
17     networks
18     - proxy-net
19
20 # =====
21 # SSL-ZERTIFIKATE (automatisch via Let's Encrypt)
22 # =====
23 acme-companion
24     image nginxproxy/acme-companion
25     container_name acme-companion
26     restart unless-stopped
27     volumes
28     - /var/run/docker.sock:/var/run/docker.sock:ro
29     - certs/etc/nginx/certs
30     - vhost/etc/nginx/vhost.d
31     - html/usr/share/nginx/html
32     - acme/etc/acme.sh
33     environment
34     - DEFAULT_EMAIL=robert@robre.de
35     - NGINX_PROXY_CONTAINER=nginx-proxy
36     depends_on
37     - nginx-proxy
38     networks
39     - proxy-net
40
41 # =====
42 # BACKEND (.NET 8 API)
43 # =====
44 backend
45     image fitness-apilatest
46     container_name fitness-api
47     restart unless-stopped
48     volumes
49     - fitness-data/app/data
50     networks
51     - proxy-net
52
53 # =====
54 # FRONTEND (React + Nginx)
55 # =====
56 frontend
57     image fitness-weblatest
58     container_name fitness-web
59     restart unless-stopped
60     environment
61     - VIRTUAL_HOST=robre.de,www.robre.de
62     - LETSENCRYPT_HOST=robre.de,www.robre.de
63     - VIRTUAL_PORT=80
64     depends_on
65     - backend
66     networks
67     - proxy-net
68
69 # =====
70 # VOLUMES (persistente Datenspeicher)
71 # =====
72 volumes
73     certs
74     vhost
75     html

```

```

76  acme
77  fitness-data
78
79  # =====
80  # NETZWERKE
81  # =====
82  networks
83    proxy-net
84    driver bridge

```

Listing 40: Vollständige docker-compose.yml

## Zeile für Zeile erklärt:

### nginx-proxy

- `image: nginxproxy/nginx-proxy` – Offizielles Image des nginx-proxy-Projekts.
- `ports: "80:80" / "443:443"` – Nur dieser Container lauscht auf den Standard-Webports. Alle HTTP/HTTPS-Anfragen kommen hier an.
- `/var/run/docker.sock:/tmp/docker.sock:ro` – Bindet den Docker-Socket ein (read-only). Darüber erkennt nginx-proxy, welche Container gerade laufen und welche Domains sie bedienen.
- `certs:/etc/nginx/certs` – Hier speichert der acme-companion die SSL-Zertifikate.

### acme-companion

- `DEFAULT_EMAIL` – Für Let's-Encrypt-Benachrichtigungen (z. B. wenn ein Zertifikat abläuft). **Muss eine echte, erreichbare E-Mail-Adresse sein!**
- `NGINX_PROXY_CONTAINER=nginx-proxy` – Sagt dem acme-companion explizit, wie der Proxy-Container heißt. Notwendig, wenn der Container-Name vom Standard abweicht.
- `depends_on: nginx-proxy` – Stellt sicher, dass der Proxy zuerst startet.

### frontend (entscheidende Umgebungsvariablen!)

- `VIRTUAL_HOST=robre.de,www.robre.de` – Sagt nginx-proxy: "Ich bin für diese Domains zuständig". Alle Anfragen an `robre.de` oder `www.robre.de` werden an diesen Container weitergeleitet.
- `LETSencrypt_HOST=robre.de,www.robre.de` – Sagt acme-companion: "Hol SSL-Zertifikate für diese Domains".
- `VIRTUAL_PORT=80` – Der interne Port, auf dem der Container lauscht (unser Nginx im Frontend-Container).

## Docker Volumes

- `certs`, `vhost`, `html`, `acme` – Speichern SSL-Zertifikate, Konfiguration und Challenge-Dateien. Diese Volumes werden von `nginx-proxy` und `acme-companion` geteilt.
- `fitness-data` – Persistente Speicherung der SQLite-Datenbank (siehe Schritt 03).

### 5.5.4 Schritt 4: Container starten

```
1 cd /opt/fitness
2 docker compose up -d
```

Listing 41: Alle Container im Hintergrund starten

`-d` steht für **detached** – die Container laufen im Hintergrund und blockieren nicht das Terminal.

### 5.5.5 Schritt 5: Status prüfen

```
1 docker ps
```

Listing 42: Laufende Container anzeigen

Es sollten vier Container erscheinen: `nginx-proxy`, `acme-companion`, `fitness-api`, `fitness-web`.

### 5.5.6 Schritt 6: Logs des acme-companion prüfen

```
1 docker logs -f acme-companion
```

Listing 43: SSL-Zertifikatserstellung verfolgen

Bei erfolgreicher Zertifikatserstellung erscheinen diese Meldungen:

```
1 [Thu May  7 09:56:11 UTC 2026] Registered
2 [Thu May  7 09:56:14 UTC 2026] Account update success
3 Creating/renewal robre.de certificates... (robre.de www.robre.de)
```

Listing 44: Erfolgreiche Ausgabe

## 5.6 Häufige Fehler und ihre Behebung

### 5.6.1 Fehler 1: "can't get nginx-proxy container ID"

**Ursache:** Der `acme-companion` findet den Proxy-Container nicht, weil dieser einen individuellen Namen hat (`nginx-proxy` statt dem Standard).

**Lösung:** Die Umgebungsvariable `NGINX_PROXY_CONTAINER=nginx-proxy` im `acme-companion` setzen (wie oben bereits eingebaut).

### 5.6.2 Fehler 2: "contact email has forbidden domain"

**Ursache:** Die `DEFAULT_EMAIL` enthält eine ungültige Domain wie `@example.com` – Let's Encrypt lehnt reservierte Domains ab.

**Lösung:** Eine echte, erreichbare E-Mail-Adresse verwenden (z. B. `robert@robre.de`).



### 5.6.3 Fehler 3: "DNS problem: NXDOMAIN"

**Ursache:** Die DNS-Propagation ist noch nicht abgeschlossen. Let's Encrypt kann die Domain nicht auflösen und verweigert die Zertifikatsausstellung.

**Lösung:** Warten, bis die Domain weltweit erreichbar ist (bis zu 24 Stunden). Dann den acme-companion neustarten:

```
1 docker restart acme-companion
```

Listing 45: Zertifikatserstellung erneut anstoßen

## 5.7 Wie füge ich später weitere Subdomains hinzu?

Das System ist extrem flexibel. Angenommen, du willst später eine Todo-App unter `todo.robre.de` hosten:

1. **DNS-Eintrag beim Hoster:** Einen neuen A-Record für `todo.robre.de` anlegen, der auf dieselbe Server-IP `185.209.229.167` zeigt. Oder: Der Wildcard-Eintrag `*.robre.de` fängt automatisch alle nicht explizit definierten Subdomains ab!
2. **Docker-Container:** Einen neuen Container mit den Umgebungsvariablen `VIRTUAL_HOST=todo.robre.de` und `LETSencrypt_HOST=todo.robre.de` starten.
3. **Fertig!** nginx-proxy erkennt den neuen Container automatisch und leitet Anfragen weiter. acme-companion holt automatisch ein SSL-Zertifikat.

**Du musst auf dem Server nichts weiter konfigurieren – alles läuft automatisch!**

## 5.8 Wie funktioniert der Reverse Proxy im Detail?

Wenn eine Anfrage an `https://robre.de/api/workouts` eingeht:

1. Die Anfrage kommt an Port 443 des Servers an (HTTPS).
2. nginx-proxy nimmt die Anfrage entgegen und entschlüsselt sie mit dem SSL-Zertifikat.
3. nginx-proxy schaut in den Host-Header: `robre.de`.
4. In seiner Konfiguration findet er: `robre.de` → Container `fitness-web`, Port 80.
5. Er leitet die Anfrage an `fitness-web:80` weiter (internes Docker-Netzwerk).
6. Der Nginx im Frontend-Container empfängt die Anfrage an `/api/workouts`.
7. Seine Konfiguration sagt: Alles mit `/api/` → weiterleiten an `fitness-api:5000`.
8. Das Backend verarbeitet die Anfrage und schickt die Antwort zurück durch die gesamte Kette.

**Die gesamte Kommunikation innerhalb des Docker-Netzwerks (Schritte 5-8) läuft unverschlüsselt – das ist okay, weil sie den Server nie verlässt.**

## 5.9 Zusammenfassung

Nach diesem Schritt haben wir:

- Einen automatischen Reverse Proxy (nginx-proxy), der Anfragen an die richtigen Container verteilt
- Kostenlose SSL-Zertifikate von Let's Encrypt, die sich automatisch erneuern
- Eine Infrastruktur, die beliebig viele weitere Domains und Subdomains aufnehmen kann – ohne manuelle Konfiguration
- Die App ist unter `https://robre.de` verschlüsselt erreichbar
- Alle Voraussetzungen für die PWA-Installation sind erfüllt

**Wartezeit beachten:** Nach der Einrichtung kann es bis zu 24 Stunden dauern, bis die DNS-Propagation abgeschlossen ist und Let's Encrypt die Zertifikate ausstellen kann. Der acme-companion versucht es automatisch jede Stunde erneut.

## 6 Gitea Installation und Server-Übersicht

In diesem Schritt installieren wir **Gitea** – einen selbst gehosteten Git-Server mit Web-Oberfläche – als Ersatz für OneDev, das sich leider als fehlerhaft erwies. Am Ende dieses Kapitels hast du einen vollständigen Überblick über deinen Server: welche Container laufen, welche Ports offen sind, wo die Daten liegen und wie alles zusammenhängt.

### 6.1 Warum ein eigener Git-Server?

Bisher haben wir den Code manuell gebaut, als Docker-Image exportiert und per scp auf den Server kopiert – ein umständlicher Prozess, der bei jedem Update wiederholt werden muss. Ein eigener Git-Server mit CI/CD-Pipeline automatisiert diesen Ablauf:

- **Push = Deploy:** Code auf den Server pushen, Pipeline baut automatisch Docker-Images und startet Container neu
- **Versionierung:** Alle Änderungen sind nachvollziehbar, kein „das ging gestern noch“
- **Teamarbeit:** Mehrere Entwickler können gleichzeitig am Projekt arbeiten
- **Backup:** Das Repository liegt auf deinem Server, nicht bei GitHub/GitLab

### 6.2 OneDev: Der gescheiterte Versuch

Ursprünglich wollten wir **OneDev** installieren – eine All-in-One-Lösung aus Git-Server, CI/CD-Pipeline, Issue-Tracker und Paket-Registry. OneDev bietet deutlich mehr Funktionen als Gitea, erwies sich jedoch als problematisch:

## 6.2.1 Installationsversuch mit Docker (Version 1dev/server:latest)

```
1 services
2   onedev
3     image 1dev/serverlatest
4     container_name onedev
5     restart unless-stopped
6     ports
7       - "6610:6610"
8       - "6611:6611"
9     volumes
10      - /var/run/docker.sock:/var/run/docker.sock
11      - onedev_data:/opt/onedev
12
13 volumes
14   onedev_data
```

Listing 46: docker-compose.yml für OneDev

### Aufgetretene Probleme:

1. **Container stoppt sofort:** Nach dem Start erschien in den Logs nur „INFO - Stopping application“. Der Container beendete sich ohne Fehlermeldung.
2. **Workaround mit fester Version:** Statt latest nutzten wir 1dev/server:11.6.13 – gleiches Problem.
3. **Datenbank-Fehler:** Die entscheidende Fehlermeldung in /opt/onedev/logs/server.log:

```
1 java.lang.IllegalStateException: EntityManagerFactory is closed
2   at org.hibernate.internal.SessionFactoryImpl.validateNotClosed(...)
3
```

Listing 47: Kritischer Fehler beim OneDev-Start

**Ursache:** OneDevs interne H2-Datenbank wurde beim ersten Start korrupt. Der Befehl `upgrade /opt/onedev` im Start-Script versuchte, eine nicht existierende Datenbank zu aktualisieren statt sie neu anzulegen. Selbst komplettes Löschen des Volumes und Neustart behoben das Problem nicht – die Logs wurden nach dem ersten Crash nicht mehr aktualisiert, der Container hing fest.

4. **Versuch mit direkter Java-Installation:** Nachdem Docker scheiterte, luden wir die JAR-Datei herunter (57 Bytes – defekter Download) und stellten fest, dass Java nicht installiert war. Zu diesem Zeitpunkt war klar: Der Aufwand für OneDev steht in keinem Verhältnis zum Nutzen.

**Fazit:** OneDev ist ein mächtiges Tool, aber die Docker-Images sind instabil. Für einen produktiven Einsatz wäre eine native Installation mit externer PostgreSQL-Datenbank empfehlenswert – für unsere Zwecke überdimensioniert.

## 6.3 Gitea: Die schlanke Alternative

**Gitea** ist ein in Go geschriebener Git-Server, der sich durch Einfachheit und geringe Ressourcenanforderungen auszeichnet. Im Vergleich zu OneDev:

Tabelle 4: Vergleich OneDev vs. Gitea

Merkmal	OneDev	Gitea
Größe Image	~500 MB	~100 MB
Startzeit	2–5 Minuten	~5 Sekunden
CI/CD	h Eingebaut	x Extern (z.B. Woodpecker CI)
Datenbank	H2 (intern, fehleranfällig)	SQLite3, PostgreSQL, MySQL
Komplexität	Hoch	Gering
Installation	Fehlgeschlagen	h In 10 Sekunden

## 6.4 Gitea mit Docker installieren

### 6.4.1 Schritt 1: OneDev rückstandslos entfernen

```

1 cd /opt/onedev
2 docker compose down 2>/dev/null      # Container stoppen
3 cd /
4 rm -rf /opt/onedev                    # Verzeichnis löschen
5 docker system prune -af                # Ungenutzte Images/Volumes entfernen

```

Listing 48: OneDev komplett löschen

docker system prune -af löscht:

- Alle gestoppten Container
- Alle ungenutzten Netzwerke
- Alle ungenutzten Images
- Alle ungenutzten Build-Caches

Insgesamt wurden 806 MB Speicher freigegeben!

### 6.4.2 Schritt 2: Gitea-Verzeichnis und docker-compose.yml anlegen

```

1 mkdir -p /opt/gitea
2 nano /opt/gitea/docker-compose.yml

```

Listing 49: Gitea-Verzeichnis vorbereiten

```

1 services
2   gitea
3     image gitea/gitealatest
4     container_name gitea
5     restart unless-stopped
6     ports
7       - "3000:3000"      # Web-UI
8       - "2222:22"        # SSH für Git-Operationen
9     volumes
10      - gitea_data/data
11      - /etc/timezone:/etc/timezoner0
12      - /etc/localtime:/etc/localtimer0
13
14 volumes
15   gitea_data

```

---

## Listing 50: docker-compose.yml für Gitea

### Erklärung der Konfiguration:

- ports: "3000:3000" – Die Web-Oberfläche ist auf Port 3000 erreichbar
- ports: "2222:22" – Git-Operationen per SSH laufen auf Port 2222 (weil Port 22 bereits vom Host-SSH belegt ist)
- gitea\_data:/data – Alle Gitea-Daten (Repositories, Konfiguration, Datenbank) liegen in diesem Volume
- /etc/timezone:/etc/timezone:ro – Übergibt die Zeitzone des Hosts an den Container (ro = read-only)
- /etc/localtime:/etc/localtime:ro – Übergibt die lokale Zeit

### 6.4.3 Schritt 3: Container starten

```
1 cd /opt/gitea
2 docker compose up -d
```

Listing 51: Gitea starten

### Ausgabe:

```
1 [+] up 9/9
2 h Image gitea/gitea:latest Pulled
3 h Network gitea_default Created
4 h Volume gitea_gitea_data Created
5 h Container gitea Started
```

Listing 52: Erfolgreicher Start

### 6.4.4 Schritt 4: Firewall öffnen

```
1 ufw allow 3000/tcp
```

Listing 53: Port 3000 freigeben

### 6.4.5 Schritt 5: Gitea im Browser einrichten

**URL:** `http://185.209.229.167:3000`

Beim ersten Aufruf erscheint die Installationsseite. Folgende Einstellungen sind vorausgefüllt und können übernommen werden:

- **Datenbanktyp:** SQLite3 (einfachste Option, perfekt für kleine Teams)
- **Pfad:** `/data/gitea/gitea.db`
- **Server-Domain:** `185.209.229.167`
- **Gitea-Basis-URL:** `http://185.209.229.167:3000/`
- **SSH-Server-Port:** 22
- **Gitea-HTTP-Listen-Port:** 3000

**Wichtig:** Unter „Administratoreneinstellungen“(ganz unten aufklappen) musst du einen Admin-Account anlegen:

- **Benutzername:** robert
- **Passwort:** [dein sicheres Passwort]
- **E-Mail:** robert@robre.de

Dann auf „**Gitea installieren**“ klicken. Nach wenigen Sekunden ist Gitea einsatzbereit.

## 6.5 Vollständige Server-Übersicht

Nach diesem Schritt ergibt sich folgendes Gesamtbild deines Servers:

### 6.5.1 Laufende Docker-Container

Tabelle 5: Alle laufenden Container auf dem Server

Container	Image	Ports	Aufgabe
gitea	gitea/gitea:latest	3000, 2222→22	Git-Server mit Web-UI
nginx-proxy	nginxproxy/nginx-proxy	80, 443	Reverse Proxy
acme-companion	nginxproxy/acme-companion	–	SSL-Zertifikate (Let’s Encrypt)
fitness-web	fitness-web:latest	80 (intern)	React-Frontend
fitness-api	fitness-api:latest	5000 (intern)	.NET 8 Backend

### 6.5.2 Docker-Volumes (persistente Datenspeicher)

Tabelle 6: Volumes und ihre Inhalte

Volume	Inhalt
gitea_gitea_data	Gitea: Repositories, Datenbank, Konfiguration
fitness_fitness-data	Fitness-App: SQLite-Datenbank (/app/data)
fitness_certs	SSL-Zertifikate von Let’s Encrypt
fitness_html	Challenge-Dateien für Let’s Encrypt
fitness_vhost	Nginx-Konfiguration pro Virtual Host
fitness_acme	ACME-Kontodaten

### 6.5.3 Firewall (nur diese Ports sind offen!)

Tabelle 7: Geöffnete Ports

Port	Dienst	Begründung
22	SSH	Server-Verwaltung
80	HTTP	Fitness-App, leitet auf HTTPS um
443	HTTPS	Fitness-App (verschlüsselt), PWA-Pflicht!
3000	Gitea Web	Git-Server-Oberfläche

## 6.5.4 Installierte Systempakete

- docker-ce, docker-ce-cli, docker-compose-plugin – Docker-Plattform
- fail2ban – Brute-force-Schutz für SSH
- git – Git (von Gitea genutzt)

Der Server ist bewusst schlank gehalten – keine überflüssigen Dienste, keine unnötigen Pakete.

## 6.5.5 Verzeichnisstruktur unter /opt

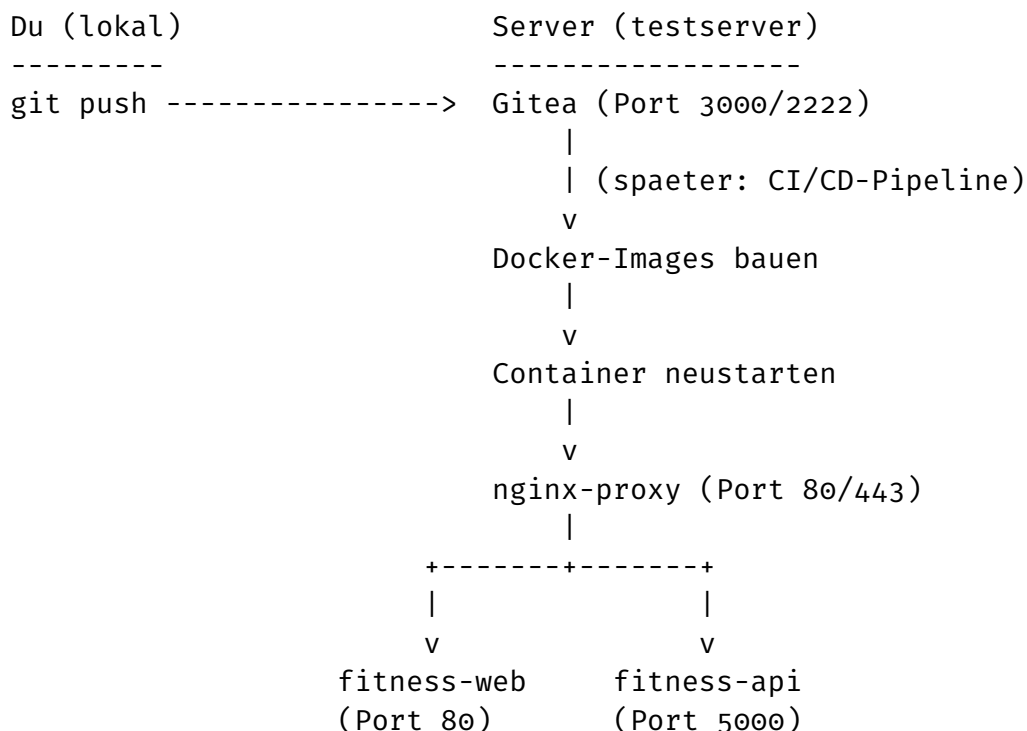
```
1 /opt/  
2 containerd/      # Docker-Laufzeitumgebung  
3 fitness/         # docker-compose.yml fuer Fitness-App  
4   docker-compose.yml  
5 gitea/           # docker-compose.yml fuer Gitea  
6   docker-compose.yml
```

Listing 54: Projektverzeichnisse auf dem Server

## 6.6 Wie Gitea in die Infrastruktur passt

**Aktueller Datenfluss:**

**Aktueller Datenfluss:**



**Was noch fehlt:** Die CI/CD-Pipeline in Gitea (z.B. mit Gitea Actions oder Woodpecker CI), die automatisch bei jedem Push Docker-Images baut und die Container aktualisiert. Das folgt in einem späteren Schritt.

## 6.7 Zusammenfassung

Nach diesem Schritt haben wir:

- OneDev aufgrund von Docker-Inkompatibilitäten verworfen
- Gitea erfolgreich mit Docker installiert (in unter 1 Minute!)
- Einen vollständigen Überblick über alle Container, Volumes, Ports und Verzeichnisse
- Die Grundlage für eine spätere CI/CD-Pipeline geschaffen

### Nützliche Befehle für die tägliche Verwaltung:

```
1 # Alle laufenden Container
2 docker ps
3
4 # Alle Volumes
5 docker volume ls
6
7 # Firewall-Status
8 ufw status verbose
9
10 # Installierte Pakete
11 apt list --installed | grep -E "docker|fail2ban"
12
13 # Verzeichnisse unter /opt
14 ls -la /opt/
```

Listing 55: Server-Cockpit – wichtige Befehle

## 7 CI/CD mit Gitea Actions

In diesem Schritt haben wir eine vollständige CI/CD-Pipeline mit Gitea Actions eingerichtet. Bei jedem `git push` auf den `master`-Branch wird unsere Anwendung automatisch gebaut und auf dem Server deployed. Dieses Kapitel dokumentiert den kompletten Prozess, alle aufgetretenen Probleme und deren Lösungen, sowie ein ausführliches Docker-Tutorial für die tägliche Arbeit.

### 7.1 Docker-Grundlagen: Container verstehen und verwalten

Bevor wir in die CI/CD-Pipeline einsteigen, ist es wichtig, die Docker-Befehle zu verstehen, mit denen wir täglich arbeiten.

#### 7.1.1 Container auflisten

```
1 docker ps
```

Listing 56: Alle laufenden Container anzeigen

**Ausgabe verstehen:**



CONTAINER ID	IMAGE	PORTS	NAMES
98f3eff6a77f	gitea/gitea:latest	0.0.0.0:3000->3000	gitea
e8e823a7beea	nginxproxy/nginx-proxy	0.0.0.0:80->80	nginx-proxy
596b718e9c16	fitness-api:latest	5000/tcp	fitness-api

### Spalten erklärt:

- CONTAINER ID – Eindeutige 12-stellige Hexadezimal-ID des Containers. Kann abgekürzt verwendet werden (z.B. 98f3).
- IMAGE – Das Image, aus dem der Container gestartet wurde. latest ist der Tag (Version).
- PORTS – Port-Weiterleitungen. 0.0.0.0:3000->3000 bedeutet: Port 3000 des Hosts ist auf Port 3000 des Containers weitergeleitet. Steht nur 5000/tcp, ist der Port nur container-intern erreichbar.
- NAMES – Der Name, den wir dem Container gegeben haben. Wird für `docker exec` und `docker logs` verwendet.

### Alle Container (auch gestoppte):

```
1 docker ps -a
```

Listing 57: Auch gestoppte Container anzeigen

### 7.1.2 Container-Logs anzeigen

```
1 # Letzte 50 Zeilen
2 docker logs fitness-api --tail 50
3
4 # Logs live verfolgen (Strg+C zum Beenden)
5 docker logs -f nginx-proxy
6
7 # Logs seit 10 Minuten
8 docker logs fitness-web --since 10m
```

Listing 58: Logs eines Containers anzeigen

### Typische Log-Einträge verstehen:

- info: Microsoft.Hosting.Lifetime[14] Now listening on: http://[::]:5000  
– Backend läuft und wartet auf Anfragen
- [notice] 19#19: signal 1 (SIGHUP) received, reconfiguring-nginx wurde neu geladen (nach Konfigurationsänderung)
- ERROR: failed to build: failed to solve-Docker-Build-Fehler (häufig falsche Pfade)

### 7.1.3 Container stoppen, starten, neustarten

```
1 # Container stoppen (bleibt existent, kann neugestartet werden)
2 docker stop fitness-web
3
4 # Gestoppten Container wieder starten
5 docker start fitness-web
6
7 # Container neustarten (stop + start)
8 docker restart nginx-proxy
9
10 # Container stoppen UND löschen (Wegwerfen!)
11 docker rm fitness-web
12
13 # Erzwingen: stoppen + löschen in einem Befehl
14 docker rm -f fitness-web
```

Listing 59: Container-Lebenszyklus

### 7.1.4 In einen Container einsteigen

```
1 # In einen laufenden Container einsteigen und Bash öffnen
2 docker exec -it fitness-api bash
3
4 # Einzelnen Befehl im Container ausführen
5 docker exec nginx-proxy nginx -s reload
6
7 # Mit sh (falls bash nicht verfügbar)
8 docker exec -it fitness-web sh
```

Listing 60: Shell im Container öffnen

#### Erklärung:

- `exec` – Führt einen Befehl in einem laufenden Container aus
- `-it` – Interaktiv + Terminal (damit du tippen kannst)
- `bash` / `sh` – Die zu öffnende Shell

### 7.1.5 Images verwalten

```
1 # Alle lokal gespeicherten Images anzeigen
2 docker images
3
4 # Nicht mehr verwendete Images löschen
5 docker image prune -a
6
7 # Ein bestimmtes Image löschen
8 docker rmi fitness-api:latest
```

Listing 61: Images anzeigen und löschen

## 7.1.6 Netzwerke inspizieren

```
1 # Alle Docker-Netzwerke
2 docker network ls
3
4 # Details eines Netzwerks (welche Container sind drin?)
5 docker network inspect fitness_proxy-net
6
7 # Nur die Container-Namen im Netzwerk anzeigen
8 docker network inspect fitness_proxy-net --format='{{range .Containers}}{{.Name
  }} {{end}}'
```

Listing 62: Netzwerke untersuchen

## 7.2 CI/CD-Pipeline mit Gitea Actions einrichten

### 7.2.1 Was ist CI/CD?

**CI (Continuous Integration):** Bei jedem Push wird der Code automatisch gebaut und getestet. Fehler werden sofort erkannt.

**CD (Continuous Deployment):** Nach erfolgreichem Build wird die Anwendung automatisch auf dem Server deployed.

#### Unser Workflow:

1. Developer macht `git push` auf den master-Branch
2. Gitea erkennt den Push und sucht nach Workflow-Dateien (`.gitea/workflows/*.yaml`)
3. Gitea weist den Job dem `gitea_runner` zu
4. Der Runner checkt den Code aus, baut Docker-Images und startet die Container neu

### 7.2.2 Der Gitea Act Runner

Der Runner ist der "Arbeiter", der die CI/CD-Jobs ausführt. Er läuft als Docker-Container und hat Zugriff auf den Docker-Socket des Hosts, um selbst Container zu bauen.

```
1 docker run -d --name gitea_runner \
2   -e GITEA_INSTANCE_URL=http://185.209.229.167:3000 \
3   -e GITEA_RUNNER_REGISTRATION_TOKEN=DEIN_TOKEN \
4   -v /var/run/docker.sock:/var/run/docker.sock \
5   gitea/act_runner:nightly
```

Listing 63: Runner starten und mit Gitea verbinden

#### Wichtige Details:

- `-v /var/run/docker.sock:/var/run/docker.sock` – Gibt dem Runner Zugriff auf Docker. **Ohne diese Zeile kann der Runner keine Images bauen!**
- Der Token ist ein Einmal-Passwort. Nach erfolgreicher Registrierung ist er verbraucht. Der Container stoppt dann mit Exit-Code 0 – das ist normal! Einfach `docker start gitea_runner` ausführen.

### 7.2.3 Die Workflow-Datei

Die Datei `.gitea/workflows/deploy.yaml` definiert den Ablauf:

```
1 name Deploy Fitness App
2 on
3   push
4     branches [ "master" ]
5
6 jobs
7   deploy
8     runs-on ubuntu-latest
9     container
10      image catthehacker/ubuntu:act-latest
11      volumes
12        - /opt/fitness:/opt/fitness
13
14    steps
15      - name Checkout
16        uses actions/checkout@v4
17
18      - name Build & Deploy
19        run |
20          docker build -f Dockerfile.api -t fitness-apilatest .
21          docker build -f Dockerfile.web -t fitness-weblatest .
22          docker stop fitness-api fitness-web 2>/dev/null || true
23          docker rm fitness-api fitness-web 2>/dev/null || true
24          docker run -d --name fitness-api --network fitness_proxy-net \
25            -v fitness_fitness-data/app/data fitness-apilatest
26          docker run -d --name fitness-web --network fitness_proxy-net \
27            -p 8080 \
28            -e VIRTUAL_HOST=robre.de,www.robre.de \
29            -e LETSENCRYPT_HOST=robre.de,www.robre.de \
30            -e VIRTUAL_PORT=80 \
31            fitness-weblatest
```

Listing 64: Vollständige deploy.yaml (finale Version)

#### Zeile für Zeile erklärt:

- `on: push: branches: [ "master" ]` – Der Workflow startet nur bei Pushes auf den master-Branch.
- `runs-on: ubuntu-latest` – Die virtuelle Maschine, auf der der Job läuft.
- `container: image: catthehacker/ubuntu:act-latest` – Ein Docker-Image mit Ubuntu + Docker CLI.
- `volumes: /opt/fitness:/opt/fitness` – Bindet das Projektverzeichnis ein.
- `2>/dev/null || true` – Unterdrückt Fehlermeldungen, wenn die Container nicht existieren.

## 7.3 Alle aufgetretenen Probleme und ihre Lösungen

### 7.3.1 Problem 1: Docker-Socket doppelt gemountet

**Fehlermeldung:** Duplicate mount point: `/var/run/docker.sock`

**Ursache:** Der Gitea-Runner mountet den Docker-Socket automatisch. Wir hatten ihn zusätzlich im Workflow definiert.

**Lösung:** Die Zeile - `/var/run/docker.sock:/var/run/docker.sock` aus dem Workflow entfernen.

### 7.3.2 Problem 2: Pfade zu Dockerfiles

**Fehlermeldung:** `open Dockerfile: no such file or directory`

**Ursache:** Die Dockerfiles lagen in `apps/api/` und `apps/web/`, aber der Build-Kontext war das Repository-Root. Relative Pfade funktionierten nicht.

**Lösung:** Kopien der Dockerfiles (`Dockerfile.api`, `Dockerfile.web`) im Root-Verzeichnis angelegt.

### 7.3.3 Problem 3: ERR\_PNPM\_IGNORED\_BUILDS

**Fehlermeldung:** `[ERR_PNPM_IGNORED_BUILDS] Ignored build scripts: @swc/core`

**Ursache:** pnpm verweigert standardmäßig Build-Skripts für Sicherheitsüberprüfung.

**Lösung:** `@vitejs/plugin-react-swc` durch `@vitejs/plugin-react` ersetzt. Keine Build-Skripts mehr nötig.

### 7.3.4 Problem 4: Fehlende nginx.conf

**Fehlermeldung:** `/apps/web/nginx.conf: not found`

**Ursache:** Die `nginx.conf` war nicht im Docker-Build-Kontext.

**Lösung:** `nginx.conf` ins Root kopiert und `Dockerfile.web` entsprechend angepasst.

### 7.3.5 Problem 5: Container ohne Port-Mapping

**Symptom:** `docker ps` zeigt `80/tcp` ohne `0.0.0.0:80->80/tcp`.

**Ursache:** Die CI/CD-Pipeline startete Container ohne `-p 80:80`.

**Lösung:** In der Workflow-Datei explizit `-p 80:80` zum `docker run`-Befehl hinzugefügt.

## 7.4 Tutorial: Einfache HTML-Seite deployen

Um den kompletten Prozess von A bis Z zu verstehen, deployen wir eine minimale HTML-Seite.

### 7.4.1 Schritt 1: Projekt erstellen

```
1 cd ~/projects
2 mkdir hello-ci
3 cd hello-ci
4 git init
```

Listing 65: Neues Projekt lokal anlegen

## 7.4.2 Schritt 2: index.html erstellen

```
1 <!DOCTYPE html>
2 <html lang="de">
3 <head>
4   <meta charset="UTF-8">
5   <title>Hello CI/CD</title>
6   <style>
7     body { font-family: sans-serif; text-align: center; padding: 50px; }
8     h1 { color: #059669; }
9   </style>
10 </head>
11 <body>
12   <h1>Hello CI/CD!</h1>
13   <p>Diese Seite wurde automatisch deployed.</p>
14 </body>
15 </html>
```

Listing 66: Minimale HTML-Seite

## 7.4.3 Schritt 3: Dockerfile erstellen

```
1 FROM nginx:stable-alpine
2 COPY index.html /usr/share/nginx/html/index.html
3 EXPOSE 80
4 CMD ["nginx", "-g", "daemon off;"]
```

Listing 67: Dockerfile für statische HTML-Seite

## 7.4.4 Schritt 4: Workflow erstellen

```
1 mkdir -p .gitea/workflows
```

Listing 68: Verzeichnis anlegen

```
1 name Deploy Hello Page
2 on
3   push
4     branches [ "master" ]
5
6 jobs
7   deploy
8     runs-on ubuntu-latest
9     container
10       image catthehacker/ubuntuct-latest
11     steps
12       - name Checkout
13         uses actions/checkout@v4
14       - name Build and Deploy
15         run |
16           docker build -t hello-cilatest .
17           docker stop hello-ci 2>/dev/null || true
18           docker rm hello-ci 2>/dev/null || true
19           docker run -d --name hello-ci -p 808080 hello-cilatest
```

Listing 69: .gitea/workflows/deploy.yaml

## 7.4.5 Schritt 5: In Gitea pushen

```
1 # In Gitea ein neues Repository "hello-ci" anlegen, dann:
2 git remote add gitea http://185.209.229.167:3000/robre/hello-ci.git
3 git add .
4 git commit -m "Initial commit"
5 git push gitea master
```

Listing 70: Push zu Gitea

## 7.4.6 Schritt 6: Firewall öffnen und testen

```
1 ssh testserver "ufw allow 8080/tcp"
```

Listing 71: Port 8080 freigeben

Im Browser: <http://185.209.229.167:8080>

## 7.5 Docker-Befehle Cheat Sheet

Tabelle 8: Häufig verwendete Docker-Befehle

Befehl	Beschreibung
<code>docker ps</code>	Laufende Container anzeigen
<code>docker ps -a</code>	Alle Container (auch gestoppte)
<code>docker logs NAME</code>	Logs eines Containers anzeigen
<code>docker logs -f NAME</code>	Logs live verfolgen
<code>docker stop NAME</code>	Container stoppen
<code>docker start NAME</code>	Container starten
<code>docker restart NAME</code>	Container neustarten
<code>docker rm NAME</code>	Container löschen
<code>docker rm -f NAME</code>	Container erzwingen löschen
<code>docker exec -it NAME bash</code>	Shell im Container öffnen
<code>docker images</code>	Alle Images anzeigen
<code>docker network ls</code>	Netzwerke anzeigen
<code>docker network inspect NETZ</code>	Netzwerk-Details
<code>docker system prune -a</code>	Ungenutzte Daten löschen

## 7.6 Zusammenfassung

Nach diesem Schritt haben wir:

- Eine vollständige CI/CD-Pipeline mit Gitea Actions
- Automatisches Build und Deployment bei jedem Push auf master
- Verständnis aller Docker-Befehle für die tägliche Arbeit
- Ein komplettes Tutorial zum Nachvollziehen des Prozesses
- Alle Fehler dokumentiert mit Ursachen und Lösungen