# Dokumentation: Webanwendung "Habit Tracker V"

Name: Pascal Speicher Matrikelnummer: 4478197 Datum: 20.11.2025

## Inhaltsverzeichnis



## 1. Thema des Projekts

### 1.1 Projektbeschreibung

Bei meinem Projekt handelt es sich um die Webanwendung **"Habit Tracker"**. Die Grundidee der Anwendung ist die Unterstützung bei der persönlichen Selbstoptimierung durch das Erfassen und Verfolgen von Gewohnheiten.

Die Anwendung ermöglicht es Nutzern, individuelle "Habits" anzulegen. Dabei wird unterschieden zwischen:

- Positiven Zielen (Do's): Gewohnheiten, die man aufbauen möchte (z. B. "Joggen gehen", "2 Liter Wasser trinken").
- Negativen Verzichten (Don'ts): Gewohnheiten, die man ablegen möchte (z. B. "Kein Zucker", "Nicht Rauchen").

Der Kern der Anwendung ist das tägliche Tracking. Der Nutzer kann sich einloggen, seine Liste einsehen und für den aktuellen Tag abhaken, ob er ein Ziel erreicht oder einem Laster widerstanden hat. Eine visuelle Übersicht (Dashboard) soll dabei helfen, die Motivation aufrechtzuerhalten.

### 1.2 Motivation

Die Wahl fiel auf dieses Thema, da Apps zur Produktivitätssteigerung ("Self-Tracking") einen direkten, praktischen Nutzen im studentischen Alltag bieten. Es ging mir darum, eine Anwendung zu schaffen, die ich theoretisch auch selbst nutzen würde. Technisch eignet sich das Thema hervorragend, um einen kompletten CRUD-Zyklus (Create, Read, Update, Delete) sowie eine Benutzerverwaltung in einer modernen Fullstack-Umgebung umzusetzen.

# 2. Ausgangssituation

Vor Beginn des Projekts hatte ich bereits grundlegende Erfahrungen in der Webentwicklung, allerdings lag mein Fokus bisher eher auf statischen Seiten oder kleineren Skripten.

- **Backend:** Die Laufzeitumgebung Deno war für mich komplettes Neuland. Ich wollte sie bewusst als moderne Alternative zu Node.js kennenlernen, insbesondere wegen der nativen TypeScript-Unterstützung.
- **Frontend:** Mit **Vue.js** hatte ich erste Berührungspunkte, aber die Architektur einer kompletten Single-Page-Application (SPA) mit State-Management (Pinia) und Routing war eine neue Herausforderung.
- **Infrastruktur:** Themen wie Deployment (Deno Deploy) und Authentifizierung mittels JWT (JSON Web Tokens) hatte ich theoretisch behandelt, aber noch nie eigenständig von Null an implementiert.

# 3. Vorgehen

Mein Entwicklungsprozess gliederte sich in mehrere Meilensteine, um die Komplexität beherrschbar zu machen:

- **Setup & Basis-Funktionalität:** Zu Beginn habe ich die Entwicklungsumgebung (Deno & Vite) aufgesetzt und die Verbindung zwischen Client und Server hergestellt (CORS). Da die App nutzerspezifisch ist, habe ich direkt die Authentifizierung (`Login` / `Register`) und die grundlegende Verwaltung von Habits (Erstellen und Auflisten) implementiert.
- **Deployment:** Sobald diese Basisfunktionen liefen, habe ich die Anwendung auf **Deno Deploy** veröffentlicht. Dies war ein wichtiger Schritt, um sicherzustellen, dass die Datenbank (Deno KV) und die Authentifizierung auch in der Cloud-Umgebung funktionieren. Hierbei fielen erste Probleme auf (z. B. Session-Verlust), die ich direkt beheben konnte.
- **Feature-Erweiterung:** Nach dem erfolgreichen Deployment habe ich die Anwendung funktional erweitert. Dazu gehörten:
    - Die Unterscheidung zwischen "Positiven" und "Negativen" Habits inkl. UI-Anpassungen.
    - Die Implementierung der komplexeren Tracking-Logik (Datumsspeicherung und täglicher Reset des Status).
    - Visuelles Feedback
- **Refactoring & Stabilisierung:** Zum Schluss habe ich den Code aufgeräumt und optimiert. Ich habe monolithische Views in wiederverwendbare Komponenten zerlegt (z. B. `HabitItem`) und fehlende Typendefinitionen ergänzt, um die Codequalität und Wartbarkeit zu erhöhen.

# 4. Anforderungsliste

| Nr. | Anforderung | Status | Umsetzung im Code |
|---|---|---|---|
| 1 | **Architektur:** Strikte Trennung von Client und Server (REST-API). | Erledigt | Frontend (Vue) kommuniziert nur via `fetch` mit Backend (Oak). |
| 2 | **Registrierung:** Nutzer können ein Konto anlegen. | Erledigt | `POST /api/register` (Passwörter werden gehasht). |
| 3 | **Login:** Authentifizierung mittels Token. | Erledigt | `POST /api/login` (JWT wird generiert und signiert). |
| 4 | **Habits verwalten (CRUD):** Erstellen, Lesen, Löschen. | Erledigt | Endpunkte `/api/habits` (GET, POST, DELETE). |
| 5 | **Tracking:** Status für den aktuellen Tag speichern. | Erledigt | `POST /api/habits/:id/entries`. |
| 6 | **Persistenz:** Dauerhafte Datenspeicherung. | Erledigt | Einsatz von **Deno KV** (Key-Value Store). |
| 7 | **UI-Feedback:** Unterscheidung Ziel/Verzicht & Statusanzeige. | Erledigt | Farbliche Kodierung (Grün/Rot) und dynamische Buttons. |
| 8 | **Sicherheit:** Schutz privater Daten. | Erledigt | Serverseitige Validierung des Users (`getUserIdFromContext`). |

| | | | |
|---|---|---|---|
| 9 | **Deployment:** Anwendung muss online verfügbar sein. | Erledigt | Gehostet auf Deno Deploy. |

# 5. Konzeption

## 5.1 Technologieauswahl

Für die Realisierung habe ich mich für einen modernen, typsicheren Stack entschieden:

- Backend: Deno & Oak
  Ich habe Deno gewählt, da es als Nachfolger von Node.js viele Probleme (wie node_modules) löst und TypeScript nativ unterstützt. Oak dient als Middleware-Framework, was das Routing und CORS-Handling sehr übersichtlich macht. Als Datenbank nutze ich Deno KV, da diese in die Runtime integriert ist ("Zero Config") und für die hierarchischen Daten (User -> Habits) völlig ausreicht.
- Frontend: Vue.js 3 & Pinia
  Vue.js bietet durch die Composition API und Single-File-Components (SFC) eine sehr saubere Struktur. Für das State Management (z.B. Speicherung des Login-Status) habe ich Pinia gewählt, da es der moderne Standard für Vue ist und deutlich intuitiver als Vuex funktioniert.

## 5.2 Entwurf & Datenmodell

Architektur:

Die Anwendung folgt dem Prinzip einer RESTful API. Der Server ist "stateless"; der Zustand der Authentifizierung liegt im JWT (Client).

Datenmodell (Deno KV):

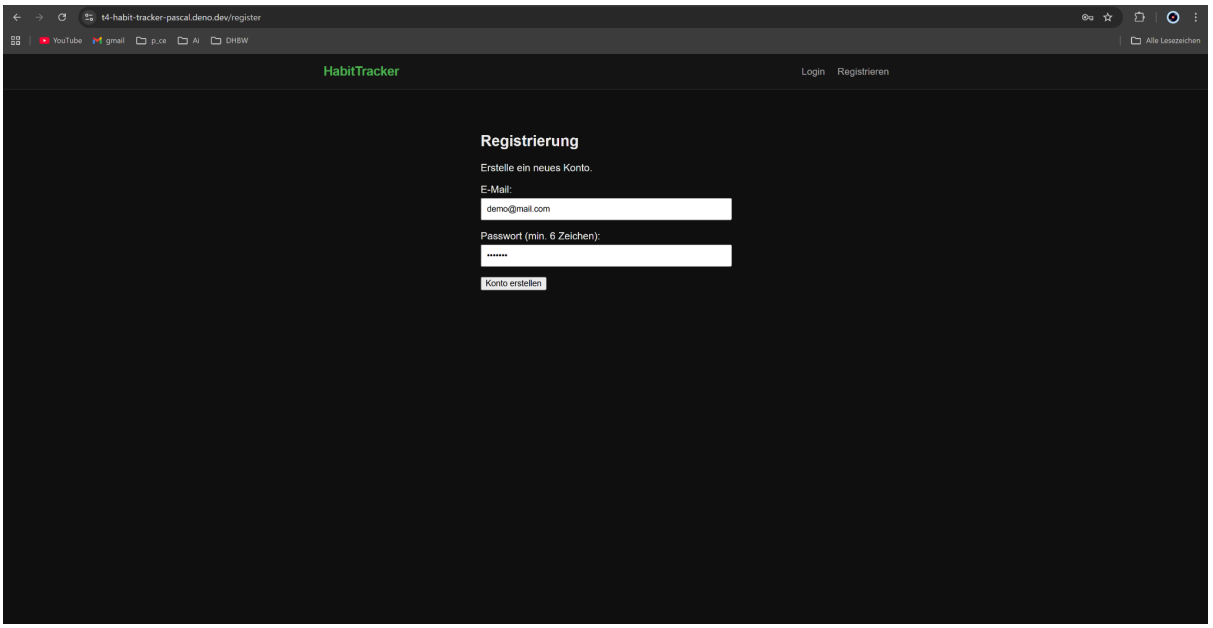
Um die Daten effizient zu speichern, habe ich folgende Schlüssel-Struktur in der Key-Value-Datenbank entworfen:

1. **User:** ["users", email] -> Speichert ID und Passwort-Hash.
2. **Habits:** ["habits", userId, habitId] -> Speichert den Habit. Durch die userId im Key können sehr schnell alle Habits eines Nutzers geladen werden.
3. **Einträge:** ["entries", habitId, date] -> Speichert das eigentliche Tracking (Datum + Status). Diese Trennung verhindert, dass das Habit-Objekt unendlich groß wird.

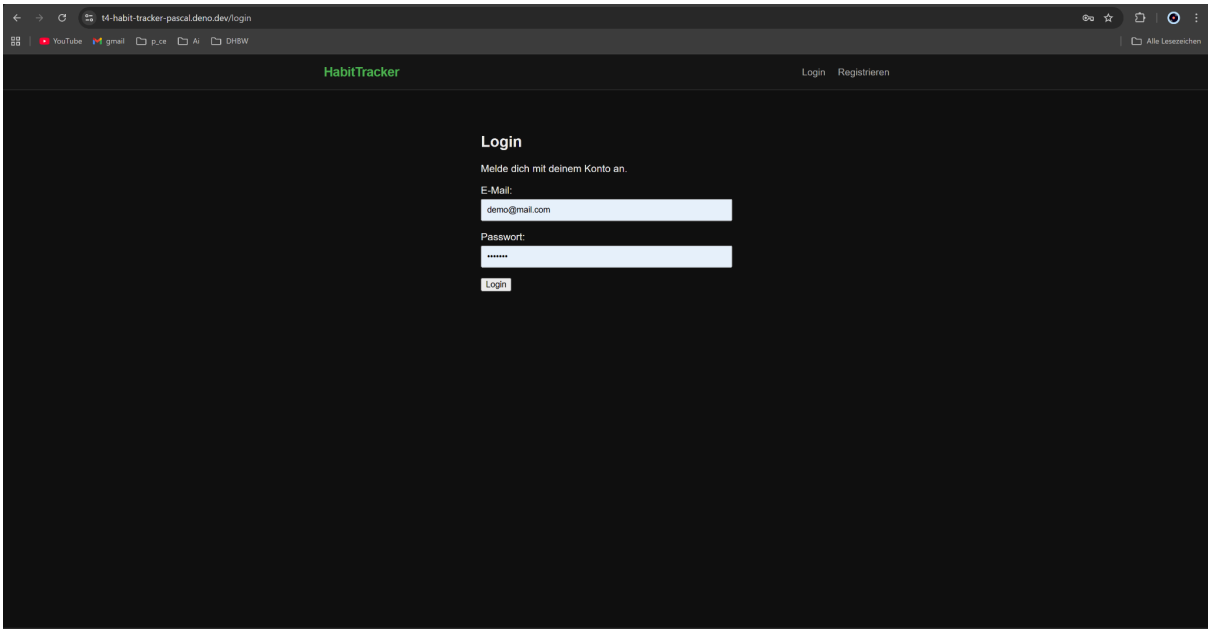
# 6. Ergebnis des Projekts

Die Webanwendung "Habit Tracker V" ist vollständig funktionsfähig und online verfügbar.

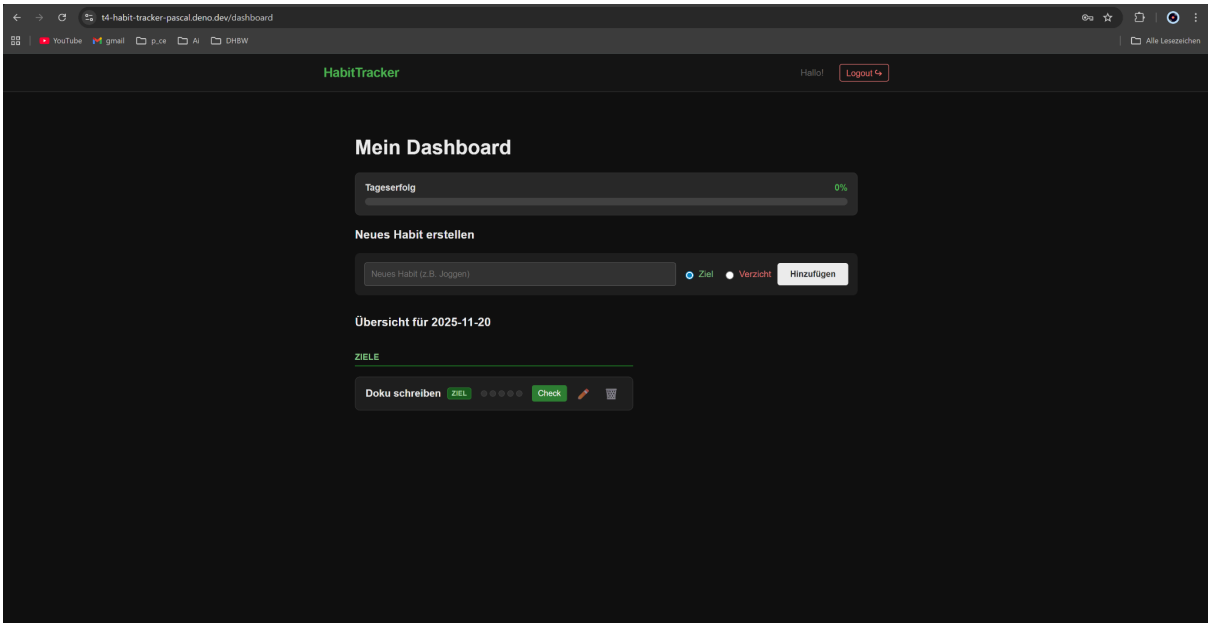
Registrierung:



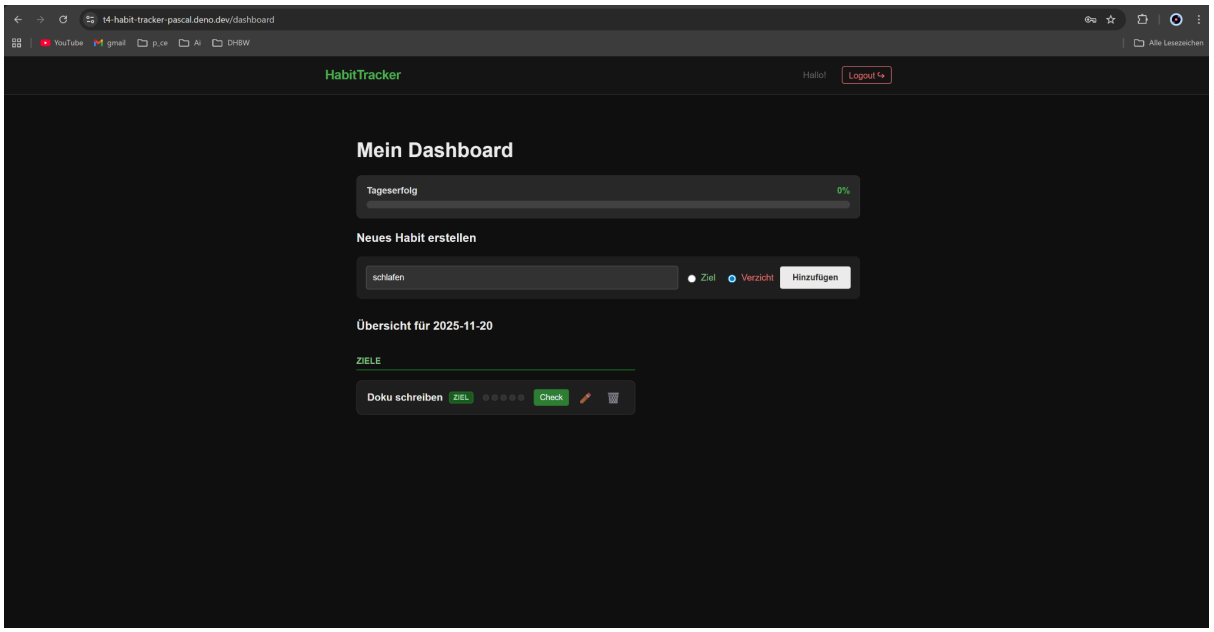

Login:

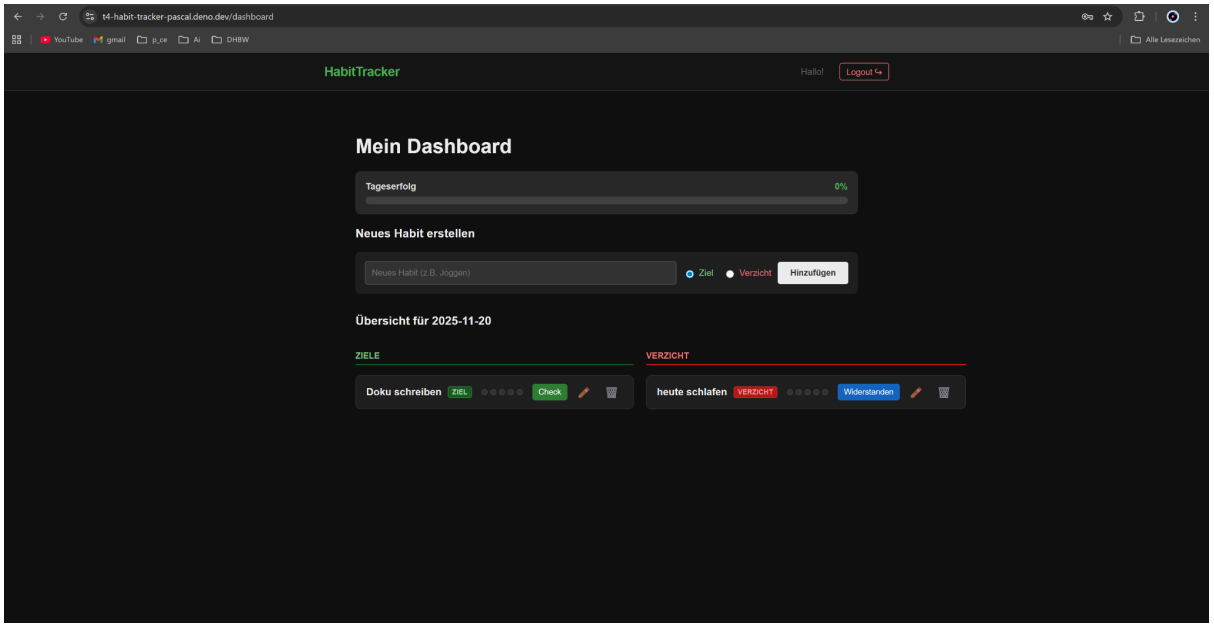



Dashboard Ansicht:

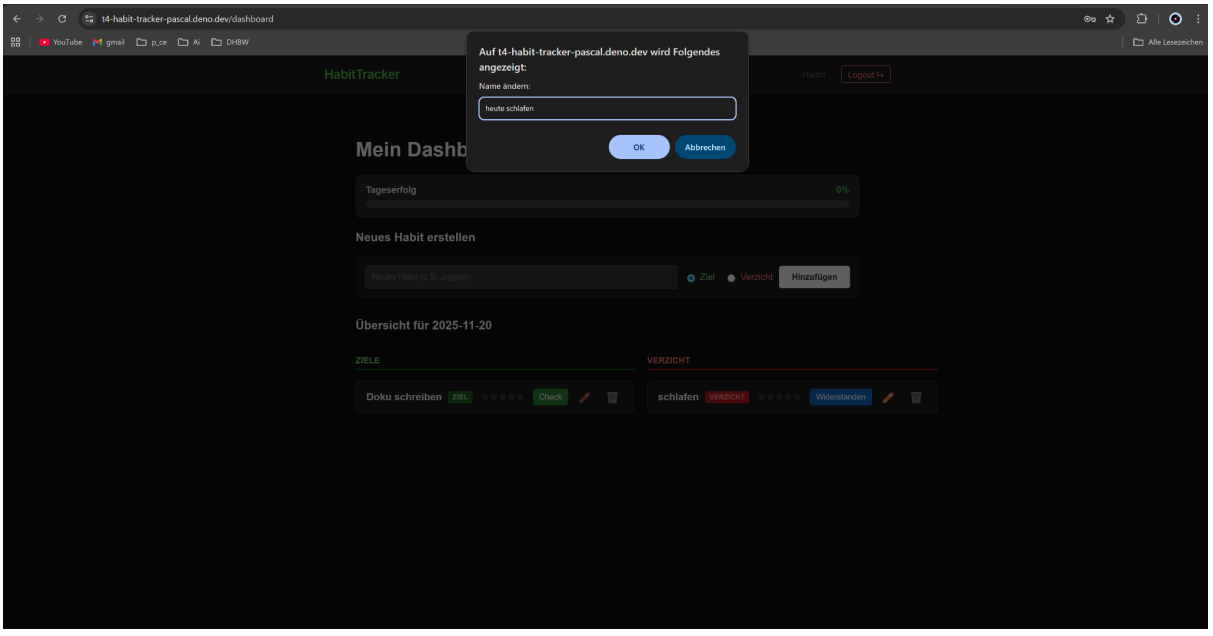



Bad Habit/Verzicht anlegen, mit Auswahl der radio buttons und dem Klick auf Hinzufügen:

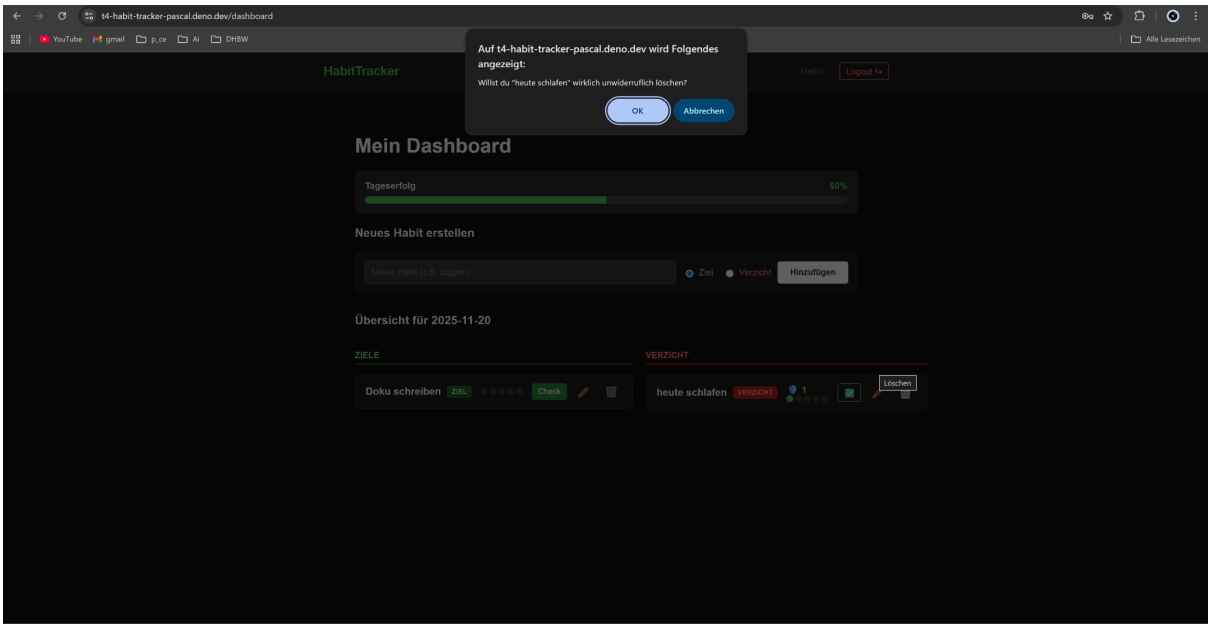





Erstelltes Habit bearbeiten, durch einen Klick auf den Stift:




Habit löschen, mit Sicherheitsfrage, um Missklicks vorzubeugen:




## 7.1 Herausforderungen

Während der Entwicklung stieß ich auf mehrere technische Hürden, die ich lösen musste:

1. **Deno KV Instabilität:** Lokal stürzte der Server anfangs ab, da die KV-Datenbank ein neues Feature ist. Ich lernte, dass ich den Server mit dem Flag --unstable-kv starten muss.
2. **Session-Verlust beim Deployment:** Nach dem Deployment wurden Nutzer ständig ausgeloggt. Die Ursache war, dass Deno Deploy Serverinstanzen oft neustartet und ich den JWT-Schlüssel dynamisch generiert hatte. Die Lösung war die Implementierung eines festen, statischen Secret-Keys.
3. **Deployment Crash (bcrypt):** Die Bibliothek bcrypt funktionierte in der Serverless-Umgebung von Deno Deploy nicht. Ich musste auf die native **Web Crypto API** (crypto.subtle) umsteigen, was letztlich sogar performanter ist.
4. **Daten-Inkonsistenz:** Beim Löschen eines Habits blieben anfangs die Tracking-Einträge als "Datenmüll" zurück. Ich habe dies gelöst, indem ich kv.atomic() Transaktionen im Backend eingeführt habe, die Habit und Einträge gleichzeitig löschen.

## 7.2 Unterstützung

Bei spezifischen Fehlermeldungen und für das Refactoring habe ich KI-Tools als "Pair-Programmer" genutzt. Besonders intensiv habe ich diese Hilfe beim Deployment auf Deno Deploy in Anspruch genommen, da mir hier die Erfahrung fehlte, um spezifische Cloud-Fehler (wie Session-Verluste oder fehlende Native Bindings) schnell zu deuten. Dies hat mir geholfen, Best Practices schneller zu verstehen und umzusetzen. Zudem habe ich die offizielle Dokumentation von Deno und Vue.js genutzt.

## 7.3 Lernerfolge & Fazit

Dieses Projekt war mein erster Schritt in die Fullstack-Entwicklung mit Deno. Ich habe gelernt:

- Wie wichtig Typensicherheit (TypeScript) für die Wartbarkeit von Code ist.
- Wie man eine REST-API von Grund auf plant und absichert.
- Dass Deployment oft eigene Probleme (Serverless Environment) mit sich bringt, die lokal nicht auftreten, weshalb frühes Testen essenziell ist.

Insgesamt bin ich mit dem Ergebnis sehr zufrieden. Die App ist stabil, schnell und erfüllt ihren Zweck.

Für die Weiterentwicklung habe ich bereits konkrete Ideen: Aktuell liegt der Fokus auf der täglichen Liste. Eine Art Monatskalender wäre eine sinnvolle Ergänzung, um den Verlauf über einen längeren Zeitraum besser zu visualisieren. Zudem könnte man die Habits feiner kategorisieren (z. B. in "Gesundheit", "Studium", "Finanzen"), um bei vielen Einträgen eine bessere Übersicht zu gewährleisten, als es die reine Unterscheidung in "Ziel" und "Verzicht" aktuell zulässt.

## A. Installationsanleitung (Docker-Variante)

## 1. Systemvoraussetzungen (Pre-Requisites)

Zur Inbetriebnahme ist lediglich die installierte und lauffähige **Docker-Plattform** erforderlich.

| **Software** | **Anmerkung** |
|---|---|
| **Docker Engine** | Muss auf dem Host-System installiert sein (z. B. Docker Desktop). |

## 2. Container-Image erstellen (Build-Prozess)

Der Build-Prozess führt automatisch die folgenden Schritte durch: Abhängigkeiten installieren, das Frontend bauen, den Deno-Server vorbereiten und die statischen Dateien in den korrekten Pfad (`frontend/Habit-Tracker-V/dist`) kopieren.

1. **Repository entpacken und wechseln:** Entpacken Sie die Abgabedatei (`.zip`) und wechseln Sie in das Stammverzeichnis des Monorepos (`Habit-Tracker`).
    a. cd Habit-Tracker
2. **Image bauen:** Führen Sie den `docker build`-Befehl aus.
    a. docker build -t habittracker-t4 .

## 3. Container starten und Anwendung ausführen

Nach dem Bauen starten Sie den Container und stellen den HTTP-Port 8000 bereit.

1. **Container starten:** Starten Sie den Container im Hintergrund (`-d`) und mappen Sie den internen Port 8000 auf den Host-Port 8000.
    a. docker run -d -p 8000:8000 --name t4-habittracker habittracker-t4
2. **Anwendung im Browser öffnen:** Die Anwendung ist nun einsatzbereit.
    a. http://localhost:8000

### 4. Initialer Login und Zugangsdaten

- Da der Container eine neue, leere Instanz des Deno.Kv-Speichers startet, sind keine vordefinierten Benutzerkonten vorhanden.
- Bitte navigieren Sie zur Registrierungsseite und erstellen Sie ein neues Konto, um die Anwendung zu nutzen.

# A. Installationsanleitung B (Ohne Docker)

## 1. Systemvoraussetzungen (Pre-Requisites)

Zur erfolgreichen Inbetriebnahme der Anwendung sind folgende Softwarekomponenten erforderlich

| Software | Version | Anmerkung |
|---|---|---|
| **Deno Runtime** | Mindestens v1.x | Wird zur Ausführung des Backend-Servers (server.ts) benötigt. |
| **Node.js und npm** | Mindestens v18.x | Wird zur Installation der Frontend-Abhängigkeiten und zum Erstellen des Vue.js-Builds benötigt. |
| **Git** | Beliebig | Wird zum Auspacken des Repositorys benötigt. |

## 2. Projekt-Setup und Build-Prozess

Das Projekt ist als Monorepo konfiguriert, wobei der Deno-Server die statischen Frontend-Dateien aus einem Unterverzeichnis ausliefert. Das Frontend-Projekt ist in diesem Beispiel im Ordner frontend/Habit-Tracker-V/ abgelegt.

1. Repository entpacken: Entpacken Sie die Abgabedatei (.zip) in einem beliebigen Verzeichnis. Dies ist das Hauptverzeichnis (MONOREPO_ROOT) Ihres Projekts.

2. **Frontend-Verzeichnis wechseln:** Wechseln Sie in das Vue.js-Projektverzeichnis, um die Abhängigkeiten zu installieren und den Build zu erstellen.
   Bash
   cd frontend/Habit-Tracker-V/
3. **Abhängigkeiten installieren:** Installieren Sie alle notwendigen Node.js-Abhängigkeiten (Vue.js, Pinia, Vue Router, etc.).
   Bash
   npm install
4. **Frontend-Build erstellen:** Erstellen Sie den Produktions-Build des Vue.js-Clients. Dieser erzeugt den Ordner dist/, der die statischen Dateien für den Server enthält.
   Bash
   npm run build
5. **Zurück zum Stammverzeichnis:** Wechseln Sie zurück in das Hauptverzeichnis, um den Server zu starten.
   Bash
   cd ../../

## 3. Server-Start und Anwendung

Der Backend-Server (server.ts) wird mit Deno ausgeführt und nutzt den eingebauten Key-Value Store (Deno.Kv) zur Persistenz der Daten.

1. **Deno Server starten:** Führen Sie den Server mit den erforderlichen Permissions-Flags aus.
   - --allow-net: Erforderlich, um auf Port 8000 zu lauschen.
   - --allow-read: Erforderlich, um die statischen Dateien aus dem dist-Ordner zu lesen.

   Bash
   deno run --allow-net --allow-read server.ts

   *Der Server läuft, sobald die Meldung Server running... in der Konsole erscheint.*

2. **Anwendung im Browser öffnen:** Rufen Sie die Anwendung im Browser auf.

   http://localhost:8000

## 4. Initialer Login und Zugangsdaten

Es sind **keine vordefinierten Test-Zugangsdaten** hinterlegt.

- Die Anwendung nutzt einen Registrierungsprozess.
- Bitte navigieren Sie zur Registrierungsseite (http://localhost:8000/register) und erstellen Sie ein neues Konto, um die Anwendung zu nutzen.
- Das Passwort muss aus Sicherheitsgründen mindestens 6 Zeichen lang sein.

Sie können die Anwendung auch unter: https://t4-habit-tracker-pascal.deno.dev/dashboard finden

# B. Benutzerdokumentation

Siehe Punkt 6