Dokumentation: Webanwendung Habit Tracker

Name: Pascal Speicher Matrikelnummer: 4478197 Datum: 23.11.2025

Inhaltsverzeichnis

1. Thema des Projekts

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.

1.1 Projektbeschreibung

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	Erledigt	Farbliche Kodierung (Grün/Rot) und dynamische Buttons.

	Ziel/Verzicht & Statusanzeige.		
8	Sicherheit: Schutz privater Daten.	Erledigt	Serverseitige Validierung des Users (getUserIdFromContext).
9	Deployment: Anwendung muss online verfügbar sein.	Erledigt	Gehostet auf Deno Deploy.
10	Kalenderansicht	Das Dashboard zeigt den aktuellen Tag und eine Historie der letzten 5 Tage an, jedoch keinen erweiterten wochen oder Monatsfunktitonen	Eine umfassende Kalenderansicht oder detaillierte Historie ist nicht implementiert. Nutzer können den Fortschritt nicht über Wochen oder Monate hinweg einsehen oder historische Daten eintragen/korrigieren.

5. Konzeption

5.1 Technologieauswahl

Für die Realisierung habe ich mich für einen modernen, typsicheren Stack entschieden, der einen effizienten Entwicklungsprozess ohne aufwendiges Tooling ermöglicht.

Backend: Deno & Oak Obwohl ich bereits Erfahrungen mit TypeScript und Node.js gesammelt habe, entschied ich mich in diesem Projekt bewusst für Deno. Mein Ziel war es, die Vorteile einer Laufzeitumgebung zu evaluieren, die TypeScript nativ unterstützt. Im Gegensatz zu Node.js entfällt hier die komplexe Konfiguration von Build-Tools oder tsconfig.json-Dateien („Zero Config“), was den Fokus direkt auf die

Programmierlogik lenkt. Das Framework Oak wurde als Middleware gewählt, da es das Routing und die Verarbeitung von Anfragen sehr übersichtlich gestaltet. Als Datenbank kommt Deno KV zum Einsatz, da diese Key-Value-Lösung direkt in die Runtime integriert ist und keine externe Datenbank-Installation erfordert.

Frontend: Vue.js 3 & Pinia Im Frontend setze ich auf Vue.js 3. Die Composition API in Verbindung mit Single-File-Components (SFC) erlaubt eine sehr saubere Trennung von Logik und Darstellung. Für das State Management (z. B. zur globalen Speicherung des Login-Status) habe ich Pinia gewählt. Es gilt als der moderne Standard für Vue.js und bietet eine deutlich intuitivere und typsichere API als ältere Alternativen wie Vuex.

5.2. Entwurf

Aufbauend auf der Technologieauswahl wurde eine Client-Server-Architektur nach dem REST-Prinzip entworfen. Diese Architektur trennt die Verantwortlichkeiten (Separation of Concerns) strikt: Der Server verwaltet Daten und Logik, während der Client für die Darstellung und Interaktion zuständig ist.

5.2.1 Architektur und Komponenten

Backend (Server-Struktur): Der Server fungiert als zentrale Schnittstelle für alle Datenoperationen.

- Middleware: Das Framework Oak bildet das Rückgrat der API. Es verarbeitet eingehende HTTP-Requests, steuert das Routing zu den Endpunkten (z. B. /api/habits) und verwaltet die CORS-Header für die Kommunikation mit dem Client.
- Authentifizierung: Die Sicherheit wird zustandslos (stateless) über JWT (JSON Web Tokens) gewährleistet. Anstatt Sessions auf dem Server zu speichern, signiert das Backend beim Login einen Token. Dieser wird vom Client bei jeder Anfrage im Header mitgesendet und serverseitig validiert.

Frontend (Client-Struktur): Die Client-Anwendung ist als Single Page Application (SPA) konzipiert.

- Routing & State: Der Vue Router übernimmt die clientseitige Navigation und schützt sensible Routen (Navigation Guards). Der globale Anwendungszustand (z. B. Authentifizierungsstatus) wird zentral im Pinia-Store gehalten, um Redundanzen bei API-Aufrufen zu vermeiden.
- Build-Prozess: Vite bündelt den Quellcode für die Produktion. Im Docker-Container wird dieser Build-Schritt vorgeschaltet (Multi-Stage Build), sodass der Deno-Server im Betrieb lediglich die optimierten statischen Assets ausliefern muss.

5.2.2 UI-Entwurf und Designentscheidungen

Das User Interface (UI) folgt den Prinzipien Fokussierung und Reduktion. Um die tägliche Nutzung so reibungsarm wie möglich zu gestalten, wurde auf komplexe Menüstrukturen verzichtet.

Design- und Interaktionskonzept:

- Dark Mode: Die Anwendung nutzt durchgängig ein dunkles Farbschema (#121212), um bei regelmäßiger Nutzung die Augen zu schonen.
- Signalwirkung: Die Unterscheidung der Habit-Typen erfolgt primär über Farben:
    - Grün: Positive Ziele (Bestätigung).
    - Rot: Negative Verzichte (Warnung/Stopp).
- Interaktionsmodell (Entscheidung): Anstelle von aufwendigen modalen Fenstern zur Bearbeitung von Einträgen kommen native Browser-Prompts (prompt, confirm) zum Einsatz.
    - Abwägung: Obwohl visuell weniger anpassbar, reduziert dies die technische Komplexität drastisch und garantiert auf mobilen Geräten ein natives, vertrautes Bediengefühl.
- Styling-Strategie (Entscheidung): Es wurde auf CSS-Frameworks (wie Tailwind) verzichtet. Stattdessen kommt Scoped CSS zum Einsatz, um Stile strikt an ihre Komponenten zu binden und den "Overhead" ungenutzter CSS-Klassen zu vermeiden.

5.2.3 Datenmodell und Persistenz

Das Datenmodell wurde spezifisch für den Deno KV Key-Value-Store entwickelt. Da hier keine relationalen Tabellen wie in SQL zur Verfügung stehen, basiert das Design auf hierarchischen Schlüsseln, die gezielt für schnelle Lesezugriffe optimiert sind.

Struktur der Datensätze: Das Schema umfasst drei primäre Entitäten:

1. User: ["users", email]
    - Speichert die User-ID und den Passwort-Hash. Dies ermöglicht einen direkten und performanten Lookup beim Login über die E-Mail-Adresse.
2. Habits: ["habits", userId, habitId]
    - Speichert die Metadaten der Gewohnheit (Name, Typ, Erstellungsdatum).
    - Optimierung: Durch den Prefix userId können alle Habits eines Nutzers mit einer einzigen Datenbankabfrage (kv.list) geladen werden, ohne über den gesamten Datenbestand iterieren zu müssen.
3. Entries: ["entries", habitId, date]
    - Speichert den täglichen Status (done | failed) für ein spezifisches Datum.

- Normalisierung: Die Tracking-Daten werden separat vom Habit gespeichert. Würden diese Einträge als Array innerhalb des Habit-Objekts gehalten, würde dieses Objekt im Laufe der Zeit unverhältnismäßig groß werden, was die Performance beeinträchtigen würde.

Transaktionssicherheit: Ein besonderer Fokus lag auf der Datenintegrität beim Löschen ("Cascading Delete"). Hierfür wird die atomare Operation kv.atomic() genutzt. Sie stellt sicher, dass beim Löschen eines Habits auch zwingend alle zugehörigen Historien-Einträge entfernt werden. Schlägt ein Teil der Operation fehl, wird die gesamte Transaktion abgebrochen, um inkonsistente Datenbestände zu verhindern.

5.3 Sicherheitsbetrachtung und Konfiguration

Im Rahmen des Sicherheitskonzepts wurde eine bewusste Abwägung zwischen operativer Sicherheit und Deployment-Komplexität getroffen.

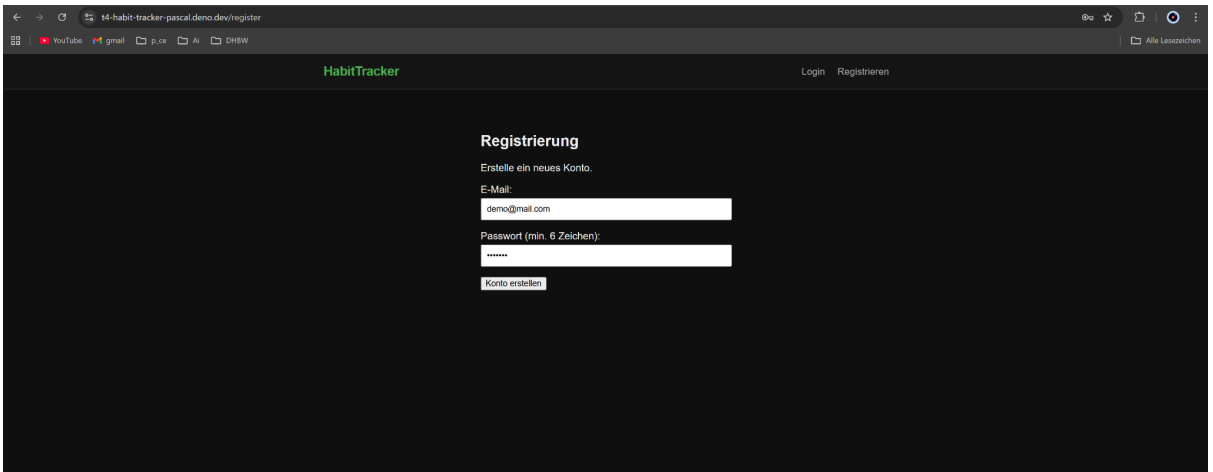
Handhabung von Secrets: Für die Signierung der JSON Web Tokens (JWT) wird im Quellcode ein statischer Schlüssel (STATIC_SECRET) verwendet.

- Begründung: Diese Implementierung wurde gewählt, um die Komplexität des Deployments im Rahmen dieser Studienarbeit zu reduzieren und Fehlerquellen durch fehlende Umgebungsvariablen im Docker-Container zu vermeiden.
- Produktionsszenario: Es ist bekannt, dass dieser Ansatz nicht für Produktivumgebungen geeignet ist. In einem realen Szenario würde dieser Schlüssel niemals im Quellcode verbleiben, sondern dynamisch zur Laufzeit über gesicherte Umgebungsvariablen (Environment Variables, z. B. via Deno.env.get) injiziert werden.

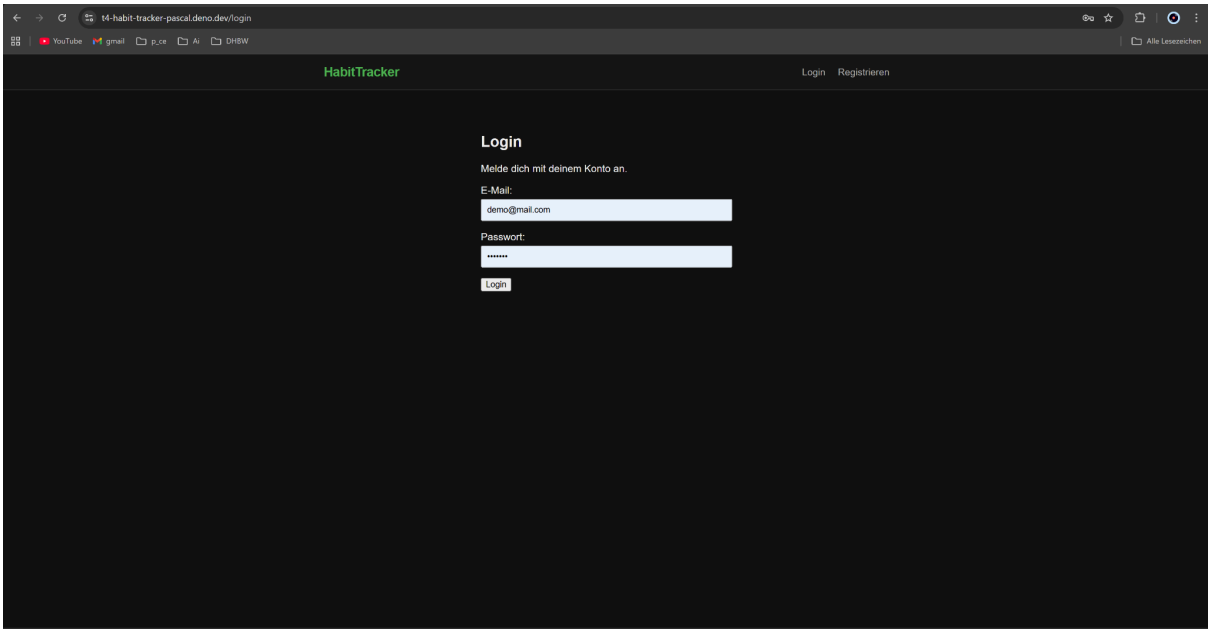
6. Ergebnis des Projekts

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

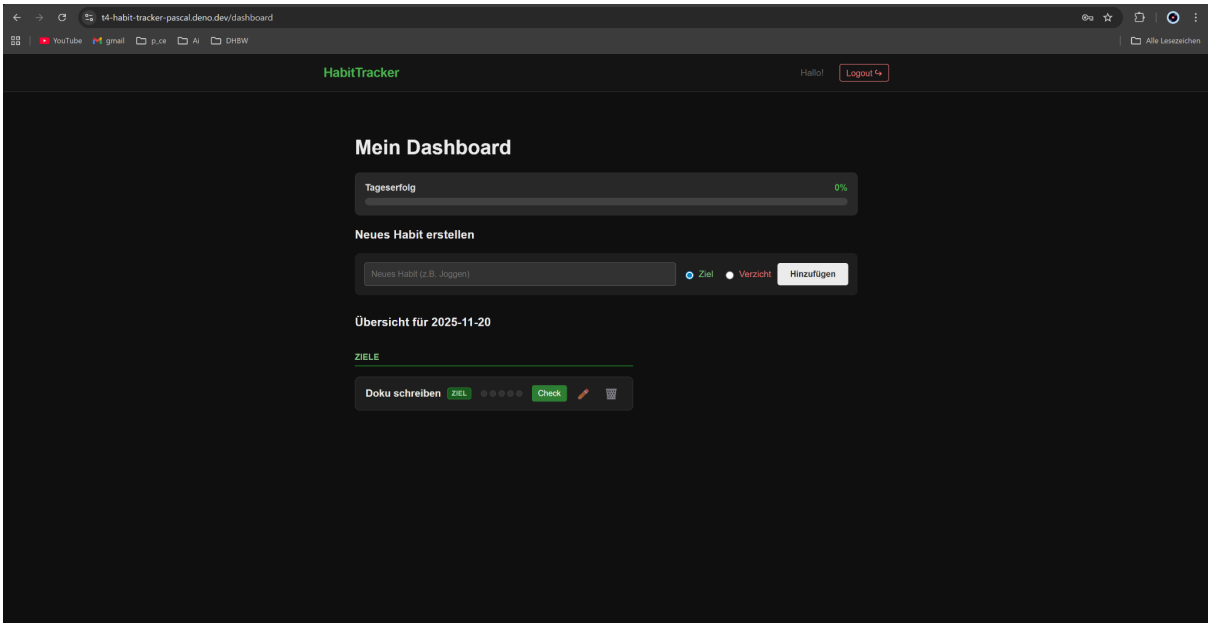
Registrierung:



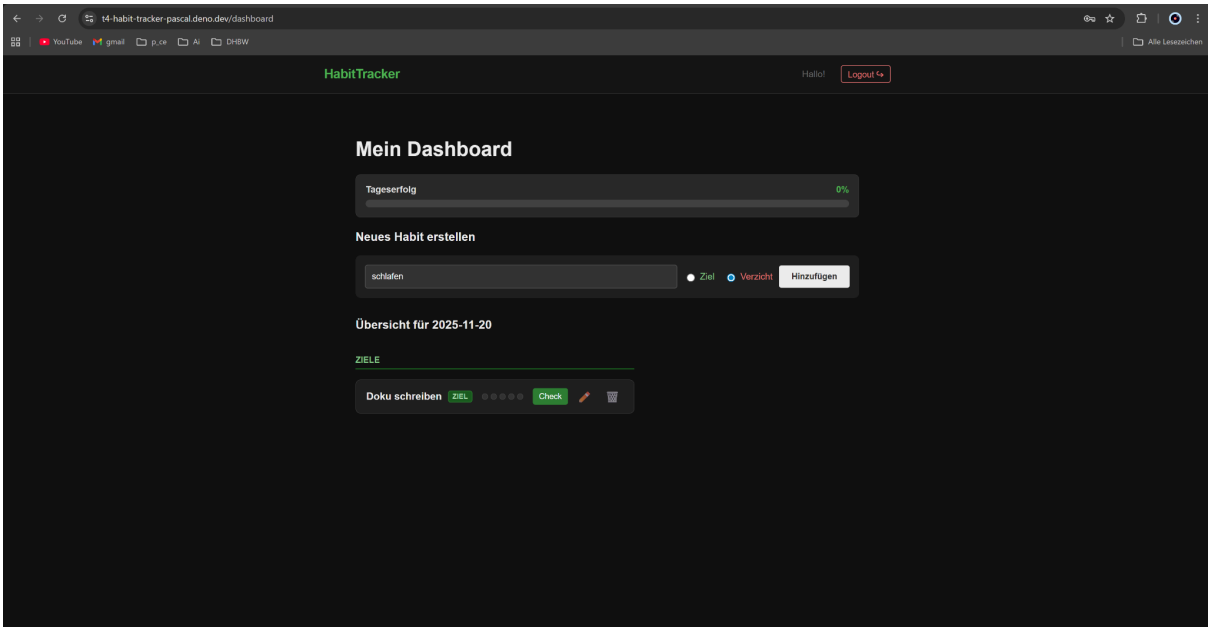
Login:



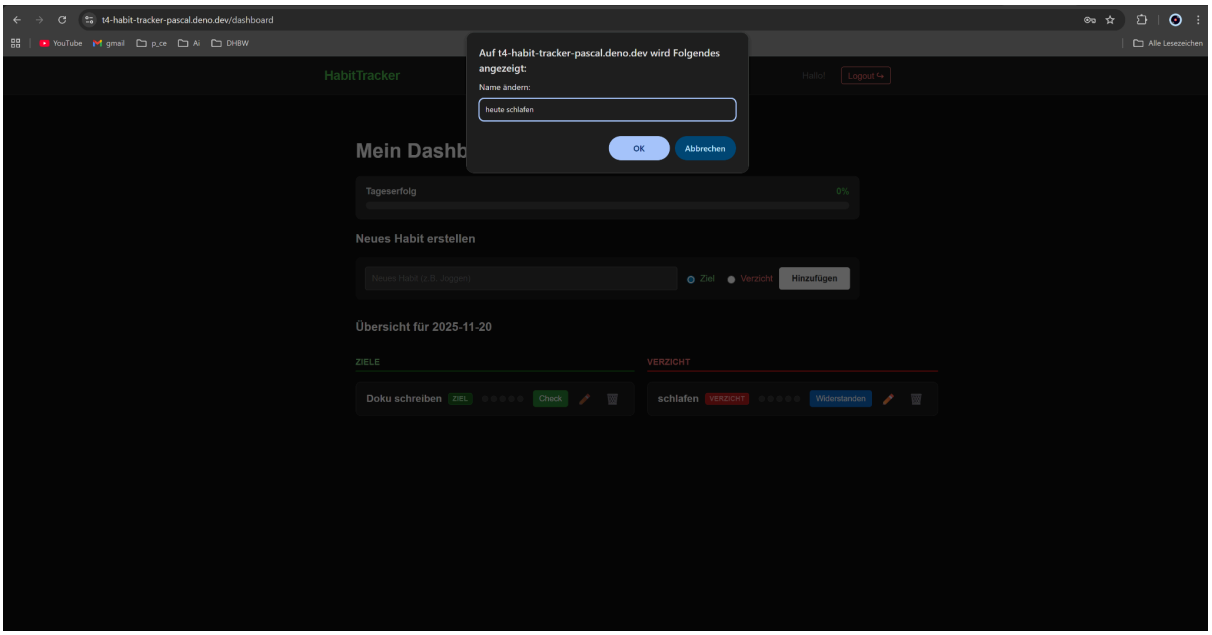
Dashboard Ansicht:



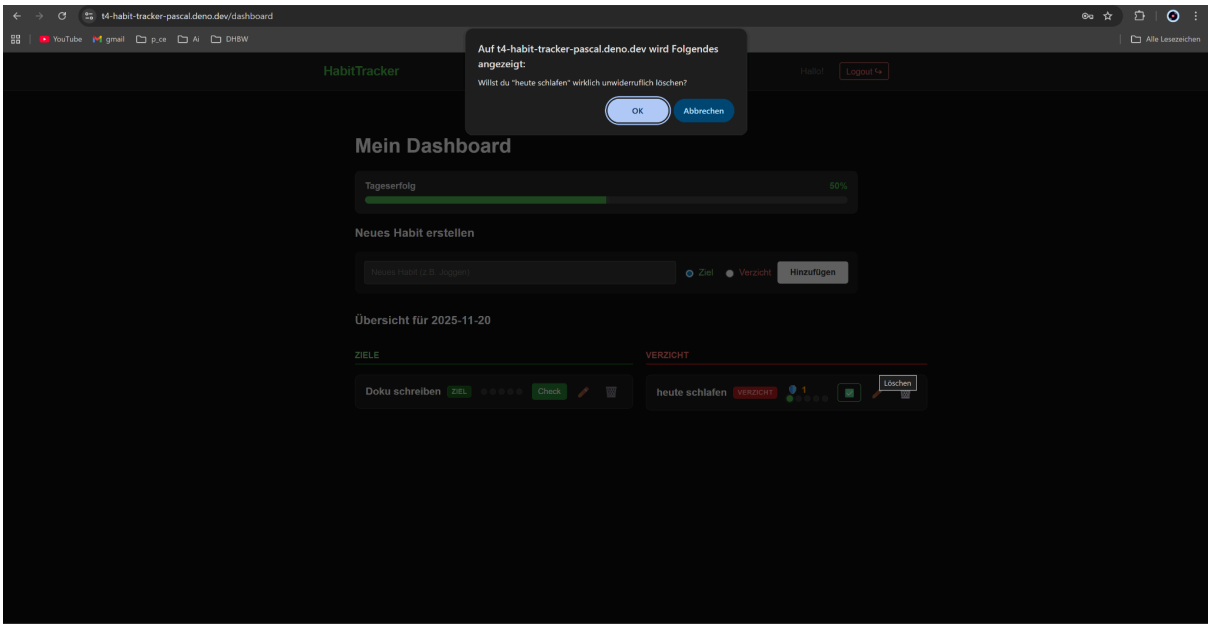
Bad Habit/Verzicht anlegen, mit Auswahl der radio buttons und dem Klick auf Hinzufügen:



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 Google Gemin 2.5 und 3 pro als „Pair-Programmer“ genutzt.

- **Frontend & Styling:** Das CSS-Styling und das responsive Layout (insbesondere die Grid-Struktur des Dashboards) wurden überwiegend durch die KI generiert, was die Entwicklungszeit für das UI erheblich verkürzte.
- **Backend & Deployment:** Besonders wertvoll war die Unterstützung beim Deployment auf Deno Deploy. Da mir hier die Erfahrung mit Serverless-Umgebungen fehlte, half die KI dabei, spezifische Cloud-Fehler (wie die Inkompatibilität von bcrypt oder Session-Verluste) zu analysieren und Best Practices (wie die Nutzung der Web Crypto API) umzusetzen.

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.

Erweiterung: Cloud-Deployment Obwohl die Aufgabenstellung primär eine lokale Container-Lösung forderte, habe ich mich entschieden, die Anwendung zusätzlich produktiv auf Deno Deploy zu veröffentlichen. Dies diente dazu, den kompletten DevOps-Zyklus einer modernen Serverless-Anwendung kennenzulernen. Diese „Kür“ war besonders lehrreich, da sie Herausforderungen aufzeigte (z. B. flüchtige Dateisysteme, Session-Handling in verteilten Systemen), die in einer rein lokalen Docker-Umgebung verborgen geblieben wären. Die Anwendung ist somit nicht nur lokal, sondern auch weltweit verfügbar unter:
https://t4-habit-tracker-pascal.deno.dev/dashboard

A. Installationsanleitung (Docker-Variante)

1. Systemvoraussetzungen

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)

Repository entpacken und wechseln:

- Entpacken Sie die Abgabedatei (.zip) und wechseln Sie in das Stammverzeichnis des Monorepos (Habit-Tracker), welches das Dockerfile enthält.
    - cd Habit-Tracker
- Terminal-Status: Sie befinden sich nun im Verzeichnis, das backend/, frontend/ und das Dockerfile enthält.

Image bauen:

- Terminal-Status: Sie befinden sich weiterhin im Stammverzeichnis
- Führen Sie den docker build-Befehl aus:
    - 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.
2. Terminal-Status: Sie befinden sich weiterhin im Stammverzeichnis.
    a. docker run -d -p 8000:8000 --name t4-habittracker habittracker-t4
3. 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.
   cd frontend/Habit-Tracker-V/
3. Abhängigkeiten installieren: Installieren Sie alle notwendigen Node.js-Abhängigkeiten (Vue.js, Pinia, Vue Router, etc.).
   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.
npm run build

5. Zurück zum Stammverzeichnis: Wechseln Sie zurück in das Hauptverzeichnis, um den Server zu starten.
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.
      deno run --allow-net --allow-read backend/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

1. Registrierung & Login Der Nutzer ruft die Startseite auf. Ist er nicht eingeloggt, wählt er „Registrieren“. Nach Eingabe einer E-Mail und eines Passworts wird das Benutzerkonto in der Datenbank angelegt. Anschließend kann sich der Nutzer mit diesen Daten einloggen und erhält einen JWT-Token für die Sitzung.

2. Dashboard & Habit erstellen Nach dem Login gelangt der Nutzer auf das Dashboard. Im oberen Bereich „Neues Habit erstellen“ gibt er den Namen einer Gewohnheit ein (z. B. „Joggen“) und wählt den Typ:

- Ziel (Grün): Für positive Gewohnheiten, die aufgebaut werden sollen.
- Verzicht (Rot): Für negative Gewohnheiten, die abgelegt werden sollen. Ein Klick auf „Hinzufügen“ speichert das Habit und listet es sofort unten auf.

3. Tägliches Tracking In der Übersicht sieht der Nutzer alle seine Habits.

- Bei Zielen klickt er auf den grünen „Check“-Button, wenn er die Tätigkeit ausgeführt hat.
- Bei Verzichten klickt er auf den roten „Widerstanden“-Button, wenn er standhaft geblieben ist. Das System speichert den Erfolg für das heutige Datum (YYYY-MM-DD). Der Status ändert sich visuell zu einem Haken (✅) und der „Streak-Counter“ (die Strähne) erhöht sich.

4. Verwaltung (Bearbeiten & Löschen) Über das Stift-Icon kann der Name eines Habits nachträglich korrigiert werden. Über das Mülleimer-Icon kann ein Habit entfernt werden. Vor dem Löschen erscheint eine Sicherheitsabfrage (confirm), um versehentlichen Datenverlust zu verhindern. Wird bestätigt, löscht das System das Habit und alle zugehörigen Historien-Einträge vollständig und rückstandsfrei aus der Datenbank.

Quellen

https://docs.deno.com/runtime/
https://vuejs.org/guide/introduction
Google Gemini
Eigene Notizen und Unterlagen des Studiengangs