Project information

Created on
February 25, 2026

Name	Last update
backend	17 minutes ago
docker	48 minutes ago
frontend	17 minutes ago
k6	6 days ago
loadtest	6 days ago
.env.example	1 week ago
.gitignore	1 week ago
README.md	17 minutes ago
docker-compose.yml	6 days ago

README.md

SwipeLearning — Challenge Task FS 2026

Autor: Yves Fricker
Studiengang: Informatik, OST – Ostschweizer Fachhochschule
Modul: Distributed Systems
Abgabe: 18. Mai 2026

---

1. Projektidee

SwipeLearning ist eine Karteikarten-Lernapplikation, bei der Benutzer eigene Decks mit Lernkarten erstellen und diese im Swipe-Format durcharbeiten können. Die Applikation ist als verteiltes System konzipiert: Statt eines einzelnen Servers laufen zwei identische Backend-Instanzen hinter einem Load Balancer. Damit lassen sich die zentralen Konzepte verteilter Systeme praktisch demonstrieren:

- **Zustandslose Replikation** – jede Anfrage kann von jeder Instanz beantwortet werden
- **Geteilter persistenter Zustand** – eine gemeinsame PostgreSQL-Datenbank als Single Source of Truth
- **Verteilte Authentifizierung** – JWT-Tokens werden von jeder Instanz unabhängig verifiziert
- **Horizontale Skalierbarkeit** – eine dritte Instanz hinzufügen erfordert null Codeänderungen
- **Ausfallsicherheit** – fällt eine Instanz aus, übernimmt die andere nahtlos

---

2. Architektur

```
Browser
   |
   | HTTP :80
   v
+---------------------------------+
|           Nginx 1.27            |
|  Load Balancer + Static Server  |
```

```
+-----------------------------------+
   |    round-robin  /api/*
   +------------------+
   |                  |
   v                  v
+----------+    +----------+
| backend1 |    | backend2 |   FastAPI (Python 3.13)
|  :8000   |    |  :8000   |   Uvicorn ASGI
+----------+    +----------+
       \              /
        \            /
         v          v
      +------------------+
      |   PostgreSQL 18  |
      |   swipelearning  |
      +------------------+
```

Komponentenübersicht

Schicht	Komponente	Aufgabe
Client	Vanilla JS SPA	Login/Register, Deck-Verwaltung, Karten-Swipe-Interface
Reverse Proxy	Nginx 1.27	Statische Dateien ausliefern, `/api/*` zum Backend weiterleiten, Round-Robin
Backend	FastAPI × 2	REST API, Geschäftslogik, JWT-Validierung, ORM-Zugriff auf DB
Authentifizierung	JWT + Refresh-Tokens (python-jose)	Kurzlebige Access-Tokens (15 min) + rotierbare Refresh-Tokens (7 Tage, in PostgreSQL gespeichert)
Datenbank	PostgreSQL 18	Persistente Speicherung aller Daten (Users, Decks, Cards, Lernfortschritt, Refresh-Tokens)

Request-Flow

```
1. Browser öffnen  →  Nginx liefert index.html + statische Dateien

2. Registrieren / Einloggen
   POST /api/auth/register oder /api/auth/login
   →  Nginx  →  backend1 oder backend2 (Round-Robin)
   →  Passwort mit bcrypt geprüft / gehashed
   →  Access-Token (15 min) + Refresh-Token (7 Tage) zurück an Browser
   →  Refresh-Token wird als SHA-256-Hash in PostgreSQL gespeichert

2b. Token erneuern (automatisch im Hintergrund, nach 15 min)
   POST /api/auth/refresh (Refresh-Token im Request-Body)
   →  Nginx  →  backend1 oder backend2
   →  Alter Refresh-Token wird sofort ungültig (Rotation)
   →  Neues Access-Token + neues Refresh-Token zurück

3. Decks laden
   GET /api/decks/  (JWT im Authorization-Header)
   →  Nginx  →  backend1 oder backend2
   →  Instanz prüft JWT lokal (kein Session-Store nötig)
   →  SQLAlchemy-Query an PostgreSQL
   →  JSON-Antwort zurück

4. Karte swipen
   POST /api/decks/{id}/study
   →  gleicher Flow wie 3.
   →  Lernfortschritt wird in DB gespeichert
```

---

3. Technologie-Stack

Bereich	Technologie	Begründung
Backend	Python 3.13 + FastAPI	Async-fähig, automatische OpenAPI-Dokumentation, starke Typisierung via Pydantic
ASGI-Server	Uvicorn	Produktionsreifer ASGI-Server, direkt im Docker-Container
ORM	SQLAlchemy 2.0	Deklarative Modelle, DB-agnostisch, zuverlässige Session-Verwaltung
Authentifizierung	JWT (python-jose) + passlib/ bcrypt	Zustandslos – keine Sticky Sessions, ideal für replizierte Dienste
Datenbank	PostgreSQL 18	ACID-konform, hervorragende Concurrency unter Last
Frontend	Vanilla HTML + CSS + JavaScript	Kein Build-Schritt, direkt als statische Dateien von Nginx ausgeliefert
Reverse Proxy / LB	Nginx 1.27	Round-Robin Upstream-Pool, pfadbasiertes Routing (`/api/*`)
Containerisierung	Docker + Docker Compose	Ein-Befehl-Start des gesamten Clusters
Lasttests	k6 + Apache Bench	Zwei unabhängige Tools: k6 für Szenarien, ab für Rohdurchsatz

4. Projektstruktur

```
swipelearning/
├── backend/
│   ├── app/
│   │   ├── api/
│   │   │   ├── endpoints/
│   │   │   │   ├── auth.py       # Register, Login, Refresh, Logout, Me
│   │   │   │   ├── decks.py      # CRUD Decks
│   │   │   │   ├── cards.py      # CRUD Cards + Study-Modus
│   │   │   │   └── health.py     # Health-Endpoint (zeigt Instanzname)
│   │   │   ├── deps.py           # JWT-Validierung als Dependency
│   │   │   └── routes.py         # Router-Aggregation
│   │   ├── core/
│   │   │   ├── config.py         # Konfiguration via Env-Variablen
│   │   │   └── security.py       # Passwort-Hashing, JWT-Erstellung
│   │   ├── db/
│   │   │   ├── database.py       # SQLAlchemy Engine + Session
│   │   │   └── init_db.py        # DB-Initialisierung mit Advisory Lock
│   │   ├── models/               # SQLAlchemy ORM-Modelle (user, deck, card, refresh_token)
│   │   ├── schemas/              # Pydantic Request/Response-Schemas
│   │   ├── services/             # Business-Logic (auth_service)
│   │   └── main.py               # FastAPI App-Factory
│   ├── requirements.txt
│   └── run.py
├── frontend/
│   ├── index.html                # Single Page Application
│   ├── css/style.css
│   └── js/
│       ├── api.js                # Fetch-Wrapper + Token-Verwaltung + Auto-Refresh
│       └── app.js                # UI-Logik, Swipe, Card-Flip
├── docker/
│   ├── backend.Dockerfile
│   ├── frontend.Dockerfile
│   └── nginx.conf                # Upstream-Pool + Location-Regeln
├── k6/
│   ├── load_test.js              # k6 Lasttest-Szenario
│   └── k6_test.md                # Dokumentierte Ergebnisse
├── loadtest/
│   ├── ab_test.sh                # Apache Bench via Docker
│   └── ab_test.md                # Dokumentierte Ergebnisse
├── .env.example                  # Template für SECRET_KEY
```

```
└── docker-compose.yml            # Vollständige Cluster-Definition
```

---

5. Inbetriebnahme

Voraussetzungen

- Docker Desktop (enthält Docker Compose)
- k6 (nur für k6-Lasttests, optional)

Schritt 1 — Secrets konfigurieren

```
cp .env.example .env
```

`.env` öffnen und `SECRET_KEY` mit einem sicheren Zufallswert befüllen:

```
python -c "import secrets; print(secrets.token_hex(32))"
```

> Dies ist der einzige manuelle Schritt — Secrets werden bewusst nicht im Repository gespeichert.

Schritt 2 — System starten

```
docker compose up --build
```

Die Applikation ist danach unter **http://localhost** erreichbar.

Was automatisch passiert:

1. Docker baut alle Images (Backend Python 3.13, Frontend Nginx)
2. PostgreSQL 18 startet und besteht den Health-Check
3. Beide Backend-Instanzen starten, sobald die DB bereit ist
4. Nginx startet und nimmt Verbindungen auf Port 80 an
5. Beim FastAPI-Start ruft jede Instanz `init_db()` auf — ein PostgreSQL Advisory Lock stellt sicher, dass die Tabellen nur einmal angelegt werden (kein Race Condition)

Container	Rolle	Interne Adresse
`swipelearning-db`	PostgreSQL 18	`db:5432`
`swipelearning-backend1`	FastAPI Instanz 1	`backend1:8000`
`swipelearning-backend2`	FastAPI Instanz 2	`backend2:8000`
`swipelearning-nginx`	Load Balancer + Frontend	`localhost:80`

Schritt 3 — Failover demonstrieren

```
# Eine Instanz abschalten – System läuft weiter
docker stop swipelearning-backend1-1

# Instanz wieder starten
docker start swipelearning-backend1-1
```

---

6. Lasttests

k6 — Skriptbasierter Lasttest (20 VUs, 50 Sekunden)

Getesteter Endpoint: `GET /api/decks/` (JWT-geschützt)

```
k6 run k6/load_test.js
```

Metrik	Wert

Metrik	Wert
Gesamtanfragen	1 853
Durchsatz	35.8 req/s
Latenz p(95)	36.6 ms
Fehlerrate	0.00 %
Alle Checks	✓ 100 %

Apache Bench — Rohdurchsatz (100 concurrent, 1 000 Anfragen)

```
docker run --rm httpd:alpine ab -n 1000 -c 100 http://host.docker.internal/api/health
```

Metrik	Wert
Gesamtanfragen	1 000
Durchsatz	1 225 req/s
Mittlere Latenz	77 ms
Latenz p(95)	113 ms
Fehlgeschlagene Anfragen	0

Beobachtung: Beide Tools zeigen 0% Fehlerrate unter gleichzeitiger Last. Der Health-Endpoint gibt den Container-Hostnamen zurück und bestätigt, dass beide Instanzen Anfragen erhalten (Round-Robin aktiv).

Bottleneck-Analyse: Bei diesem Setup ist die PostgreSQL-Instanz der potenzielle Engpass, da beide Backend-Instanzen sich eine DB teilen. Die Backends selbst sind zustandslos und horizontal unbegrenzt skalierbar — die DB ist der Single Point of Failure.

---

7. Sicherheit (OWASP JWT Guidelines)

Massnahme	Implementierung
Passwort-Speicherung	bcrypt-Hash via passlib — Klartext wird nie gespeichert
Secret Management	`SECRET_KEY` ausschliesslich aus Umgebungsvariable, kein hartcodierter Default
Access-Token-Laufzeit	15 Minuten — kurze Lebensdauer begrenzt das Zeitfenster bei Token-Diebstahl
Refresh-Token-Rotation	Bei jeder Nutzung des Refresh-Tokens wird der alte sofort widerrufen und ein neues Paar ausgestellt
Reuse-Detection	Wird ein bereits widerrufener Refresh-Token erneut verwendet, werden alle Tokens des Users gesperrt (Token-Family-Invalidierung gegen Replay-Angriffe)
Server-seitige Revozierung	Refresh-Tokens in PostgreSQL gespeichert (nur als SHA-256-Hash) — sofortiger Logout via `POST /auth/logout` möglich
Token-Transport	Bearer-Token im `Authorization`-Header, nie in der URL
Zustandslose Validierung	Jede Instanz prüft die JWT-Signatur unabhängig — kein geteilter Session-Store
Minimaler Token-Inhalt	Nur `sub` (User-ID) und `exp` (Ablaufzeit) im Payload
XSS-Prävention	`escHtml()`-Funktion im Frontend verhindert Injection in die DOM

---

8. Verteilte Systeme — Designentscheidungen

Zustandslosigkeit der Backend-Instanzen

Beide Instanzen teilen keinerlei In-Memory-Zustand. Jede Anfrage trägt ein selbst-beschreibendes JWT, das von jeder Instanz unabhängig verifiziert werden kann. Es wird kein gemeinsamer Session-Store (z.B. Redis) benötigt.

Geteilter SECRET_KEY

Beide Backend-Instanzen lesen denselben `SECRET_KEY` aus der Umgebungsvariable. Ein JWT, das von `backend1` ausgestellt wurde, kann von `backend2` genauso validiert werden — das ist die Grundvoraussetzung für zustandslose Authentifizierung in replizierten Diensten.

Advisory Lock bei DB-Initialisierung

Starten beide Instanzen gleichzeitig (was Docker Compose tut), würden beide versuchen, die Tabellen anzulegen. Der PostgreSQL Advisory Lock (`pg_advisory_xact_lock`) stellt sicher, dass nur eine Instanz die `CREATE TABLE`-Anweisungen ausführt, während die andere wartet. SQLAlchemy's `create_all()` ist idempotent — bestehende Tabellen werden nicht überschrieben.

Round-Robin Load Balancing mit Failover

Nginx verteilt Anfragen standardmässig im Round-Robin-Verfahren. Mit `proxy_next_upstream error timeout` leitet Nginx Anfragen automatisch an die andere Instanz weiter, falls eine nicht antwortet. Damit ist ein einfaches Failover ohne zusätzliche Konfiguration gegeben.

Refresh-Tokens in einem replizierten System

Refresh-Tokens sind serverseitig gespeichert (PostgreSQL) und daher per Definition zustandsbehaftet. Dennoch bleibt das System vollständig horizontal skalierbar: Beide Backend-Instanzen greifen auf dieselbe PostgreSQL-Instanz zu — ein Refresh-Token, das bei `backend1` ausgestellt wurde, kann von `backend2` genauso validiert und rotiert werden. Access-Tokens bleiben weiterhin komplett zustandslos (JWT-Signaturprüfung lokal, ohne DB-Abfrage).

Skalierbarkeit

Eine dritte Backend-Instanz hinzuzufügen erfordert ausschliesslich:

1. Einen neuen Service in `docker-compose.yml`
2. Eine neue Zeile im `upstream backend_pool` in `nginx.conf`

Kein Anwendungscode muss verändert werden.

---

9. Bekannte Einschränkungen

Thema	Beschreibung
HTTPS	System läuft auf HTTP Port 80 (Development). In Produktion wäre TLS via Nginx oder Traefik nötig.
DB Single Point of Failure	PostgreSQL ist nicht repliziert. Bei DB-Ausfall fällt die gesamte Applikation aus.
Version Pinning	`requirements.txt` nutzt `>=`-Constraints. In Produktion würde man exakte Versionen mit `pip freeze` fixieren.
Token-Cleanup	Abgelaufene und widerrufene Refresh-Token-Einträge verbleiben in der `refresh_tokens`-Tabelle — kein automatischer Cleanup-Job (Cronjob oder Datenbankpartitionierung fehlt).

---

10. Verwendete Open-Source-Bibliotheken

Bibliothek	Lizenz	Verwendung
FastAPI	MIT	Web-Framework
Uvicorn	BSD	ASGI-Server
SQLAlchemy	MIT	ORM
psycopg2	LGPL	PostgreSQL-Treiber
python-jose	MIT	JWT
passlib	BSD	Passwort-Hashing
bcrypt	Apache 2.0	Hashing-Algorithmus
Pydantic	MIT	Datenvalidierung

Bibliothek	Lizenz	Verwendung
Nginx	BSD	Reverse Proxy / LB
PostgreSQL	PostgreSQL License	Datenbank
k6	AGPL	Lasttest

Alle Bibliotheken sind Open-Source-Software gemäss den Anforderungen des Challenge Tasks.

---

11. KI-Unterstützung

Dieses Projekt wurde mit Unterstützung von **Claude Code** (Anthropic) entwickelt. Gemäss den Anforderungen des Challenge Tasks verstehe und erkläre ich den gesamten Code vollständig, unabhängig davon, wie er entstanden ist.

Yves Fricker — OST Distributed Systems, FS 2026