AILLMChat/ 📁 **STRUCTURA DIRECTOARELOR**

```
│
├── app.py              # aplicația principală Flask
├── config.py           # configurări (DB path, chei API)
│
├── database/
│   ├── __init__.py
│   ├── db.py           # conexiunea SQLite
│   └── init_db_embeddings.py       # script de creare tabele
│
├── chatbot/
│   ├── __init__.py
│   ├── logic.py        # Orchestratorul. El face legătura între baza de date, motorul de căutare semantică și modelul AI (LLM) pentru a oferi un răspuns inteligent.
│   └── llm_client.py   # Când utilizatorul trimite un, textul ajunge aici, generate_answer procesează cererea folosind modelul FLAN, iar rezultatul este trimis înapoi către interfață.
│
├── knowledge/
│   ├── __init__.py
│   ├── routes.py       # endpoint API + formular web
│   └── service.py      # operațiuni CRUD pe baza SQLite
│   └── embeddings.py       # MODEL_NAME = SBERT_MODEL
├── templates/
│   ├── chat.html
│   └── add_knowledge.html
│
└── requirements.txt
```

**Cum funcționează căutarea semantică? (search_semantic)**

Când un elev pune o întrebare, se întâmplă următoarele:

1. **Transformare**: Întrebarea elevului este transformată într-un vector folosind același model SBERT.
2. **Căutare în Index**: FAISS compară vectorul întrebării cu toți vectorii din baza de date și îi găsește pe cei mai apropiați (cei mai „similari”).
3. **Recuperare**: Codul ia ID-urile găsite de FAISS, merge în baza de date SQL și extrage textul real (materia, clasa, conținutul).
4.

| **Componentă** | **Rol** |
|---|---|
| **SentenceTransformer** | „Traducătorul” care transformă textul în coordonate matematice (vectori). |
| **FAISS Index** | „Harta” care permite găsirea rapidă a coordonatelor apropiate. |
| **id_map** | Un dicționar care face legătura între poziția din harta FAISS și ID-ul din baza de date SQL. |
| **all-mpnet-base-v2** | Modelul de AI folosit. (Sfat: Pentru română, paraphrase-multilingual-MiniLM-L12-v2 ar putea fi mai precis). |

Dacă sunt 10.000 de lecții salvate, o căutare clasică prin

**SELECT * FROM knowledge WHERE content LIKE '%text%'**

ar fi foarte lentă și ar rata răspunsurile care nu conțin exact acele cuvinte. Codul tău găsește răspunsul corect chiar dacă elevul scrie cu greșeli sau folosește sinonime.

**Embedding** este traducerea înțelesului unui text într-un șir de numere (un vector). Dacă textul ar fi o locație pe o hartă, embedding-ul ar fi coordonatele GPS ale acelei locații.

### Ce înseamnă Embeddings aici?

În mod normal, un calculator vede cuvântul „măr” și „fruct” ca fiind total diferite (pentru că literele diferă). Un model de **Embeddings** (cum este SBERT-ul pe care l-ai configurat) „citește” textul și îl plasează într-un spațiu cu sute de dimensiuni.

- Dacă două texte vorbesc despre același subiect (ex: „Gravitația ne atrage spre Pământ” și „Forța gravitațională acționează asupra corpurilor”), vectorii lor vor fi **foarte apropiați** matematic.
- Acest lucru îi permite chatbot-ului tău să găsească informații relevante chiar dacă elevul nu folosește exact cuvintele din manual.

```
def add_knowledge(subject, grade, content):
    con = get_connection()
    cur = con.cursor()
    cur.execute(
        "INSERT INTO knowledge (subject, grade, content) VALUES (?, ?, ?)",
        (subject, grade, content)
    )
    knowledge_id = cur.lastrowid
    con.commit()
    con.close()

    emb = compute_embedding(content)
    save_embedding(knowledge_id, emb)
    # Pentru set mic: rebuild index. Pentru dataset mare: append logic.
    build_faiss_index()
    ensure_index()
```

Mai întâi, salvezi textul „lizibil” (materia, clasa, conținutul) în baza de date clasică. Ai nevoie de **knowledge_id** (cheia primară) pentru a ști mai târziu căruia text îi aparține vectorul pe care urmează să-l creezi.

```
cur.execute(
        "INSERT INTO knowledge (subject, grade, content) VALUES (?, ?, ?)",
        (subject, grade, content)
    )
    knowledge_id = cur.lastrowid
```

**compute_embedding**: Textul lecției trece prin modelul AI și iese o listă de numere (vectorul).

**save_embedding**: Acest vector este salvat într-un tabel separat (sau coloană specială), legat de **knowledge_id**. Astfel, ai o bază de date cu "coordonatele GPS" ale tuturor lecțiilor tale.

```
    emb = compute_embedding(content)
    save_embedding(knowledge_id, emb)
```

Chiar dacă ai salvat vectorul în baza de date SQL, căutarea prin mii de vectori în SQL este lentă.

- **build_faiss_index()**: Reconstruiește structura de date FAISS (indexul). Imaginează-ți că FAISS este un bibliotecar care organizează cărțile pe rafturi în funcție de subiect. De fiecare dată când adaugi o „carte” nouă, bibliotecarul trebuie să își actualizeze registrul pentru a o găsi instantaneu.
- **ensure_index()**: Se asigură că indexul nou creat este cel folosit de aplicație în memoria RAM pentru următoarele întrebări ale utilizatorilor.

```
    build_faiss_index()
    ensure_index()
```

**De ce se reconstruiește indexul la fiecare adăugare?**

În acest stadiu (pentru un volum mic de date), este cea mai sigură metodă de a te asigura că noua informație este imediat „căutabilă”.

Dacă aplicația ta ar avea **milioane** de rânduri, nu ai mai reconstrui tot indexul (ar dura minute), ci ai folosi o funcție de tip index.add(new_vector), care doar adaugă elementul nou la structura existentă.

**Rezumatul procesului:**

1. **Omul** adaugă o lecție (Text).
2. **SQL** reține textul pentru a-l afișa.
3. **AI-ul** transformă textul în numere (Embedding).
4. **FAISS** organizează aceste numere pentru o căutare ultra-rapidă.

**Pași finali de setup și rulare**

1. **pip install -r requirements.txt**
2. PS C:\student\2025-2026\AILLMChat> **python -m database.init_db_embeddings**
3. PS C:\student\2025-2026\AILLMChat> **python app.py**

* Serving Flask app 'app'

* Debug mode: on

WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.

* Running on all addresses (0.0.0.0)

* Running on **http://127.0.0.1:5000**

* Running on http://192.168.1.135:5000

**http://127.0.0.1:5000/add**