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. **python -m database.init_db_embeddings**
3. **python scripts/bulk_import.py scripts/knowledge_data.json**
4. **python scripts/bulk_import.py scripts/knowledge_data2.json**
5. **python.exe -m knowledge.reindex**
6. **python app.py**

Output server

* Serving Flask app 'app'

* Debug mode: on

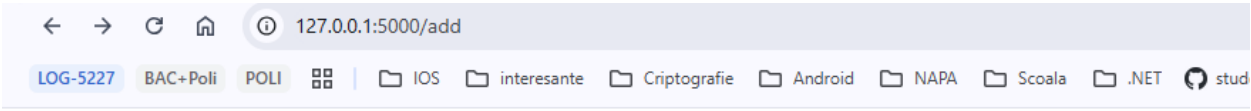
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.

* 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



Adaugă informație în baza de cunoștințe

Materie: Fizica

Clasa: IX

Conținut:

Valoarea accelerației gravitaționale (notată cu g) depinde de locul în care te afli, dar în contextul problemelor de fizică pentru școală, se folosesc de obicei următoarele valori:1. Valoarea standard pe PământLa nivelul mării și la o latitudine de 45°, valoarea convențională este:g egala cu 9,80665 m/s^2. În calculele rapide de la școală (clasele VI-IX), profesorii acceptă de obicei aproximarea:g egala cu 10 \m/s^2.

Adaugă

Sistemul **RAG (Retrieval-Augmented Generation)** este bazat pe două modele diferite, așa cum apar în config.py:

1. **SBERT_MODEL (paraphrase-multilingual-MiniLM-L12-v2)**: Acesta este "căutătorul". El transformă întrebarea ta în numere (vectori) și caută în baza de date FAISS fragmentele relevante (ex: lecția despre Newton).
2. **FLAN_MODEL (google/flan-t5-base)**: Acesta este "scriitorul". El primește rezultatele găsite și încearcă să formuleze un răspuns.

Rezumat (ce e bine deja)

- Arhitectură simplă și clară: **Flask API + SQLite + FAISS + SBERT + LLM**.
- UI-ul este minimalist și corect pentru text: folosești textContent și ai meta charset="utf-8".
- Importul din JSON folosește encoding='utf-8' și log-uri.
- Este folosit SentenceTransformer multilingv (bun pentru RO) și FAISS cu normalizare L2 pentru cos-sim (IndexFlatIP + normalize).

01.02.2026 - Ce trebuie regândit?

În implementarea curentă, pentru fiecare rezultat FAISS faci: connect → SELECT → close, în interiorul buclei.
Asta e overhead mare mai ales la top_k=10 (10 conexiuni + 10 query-uri).

Codul actual

```
def search_semantic(query: str, top_k, subject, grade):
    global _index, _id_map
    if _index is None:
        return []

    q_emb = compute_embedding(query)
    q_emb = q_emb.reshape(1, -1)
    faiss.normalize_L2(q_emb)
    D, I = _index.search(q_emb, top_k)
    results = []
    for score, idx in zip(D[0], I[0]):
        if idx < 0:
            continue

        knowledge_id = int(_id_map[idx])
        con = get_connection()
        cur = con.cursor()
        cur.execute("SELECT content, subject, grade FROM knowledge WHERE id=?",
(knowledge_id,))
        row = cur.fetchone()
        con.close()

        if row:
            content, s, g = row
            if subject and s.lower() != subject.lower():
                continue
            if grade and g.lower() != grade.lower():
                continue
            results.append({"id": knowledge_id, "score": float(score), "content":
content, "subject": s, "grade": g})
    return results
```

Codul îmbunătățit

```python
def search_semantic(query: str, top_k: int, subject: str, grade: str):
    global _index, _id_map
    if _index is None:
        return []

    # 1) Embed + căutare FAISS (rămâne la fel)
    q_emb = compute_embedding(query).reshape(1, -1)
    faiss.normalize_L2(q_emb)
    D, I = _index.search(q_emb, top_k)

    # 2) Transformă index-urile FAISS în knowledge_id-uri
    #    Păstrăm și scorurile în map ca să le atașăm ulterior
    ids_in_order = []
    score_by_id = {}
    for score, idx in zip(D[0], I[0]):
        if idx < 0:
            continue
        knowledge_id = int(_id_map[idx])
        ids_in_order.append(knowledge_id)
        score_by_id[knowledge_id] = float(score)

    if not ids_in_order:
        return []

    # 3) Un singur SELECT cu IN (...)
    #    Construim placeholder-ele (?, ?, ?, ...)
    placeholders = ",".join(["?"] * len(ids_in_order))

    con = get_connection()
    cur = con.cursor()

    # Observație: filtrele subject/grade le putem aplica direct în SQL
    # ca să reducem datele returnate.
    sql = f"""
        SELECT id, content, subject, grade
        FROM knowledge
        WHERE id IN ({placeholders})
    """
    params = list(ids_in_order)

    # Optional: filtre în SQL dacă au fost furnizate
    if subject:
        sql += " AND LOWER(subject) = LOWER(?)"
        params.append(subject)
    if grade:
        sql += " AND LOWER(grade) = LOWER(?)"
        params.append(grade)
```

```
    cur.execute(sql, params)
    rows = cur.fetchall()
    con.close()

    # 4) Mapează rezultatele pe id ca să păstrăm ordinea FAISS
    row_by_id = {}
    for rid, content, s, g in rows:
        row_by_id[int(rid)] = (content, s, g)

    results = []
    for knowledge_id in ids_in_order:
        row = row_by_id.get(knowledge_id)
        if not row:
            continue
        content, s, g = row
        results.append({
            "id": knowledge_id,
            "score": score_by_id.get(knowledge_id, 0.0),
            "content": content,
            "subject": s,
            "grade": g
        })

    return results
```

Logica de căutare – esenta metodei

Se preia întrebarea utilizatorului (query), se transforma într-un **vector numeric** (embedding), apoi se găsește în FAISS cei mai apropiați vectori salvați (din lecțiile introduse) — adică acele lecții care au sens apropiat de întrebare.

```
1.   q_emb = compute_embedding(query)
2.   q_emb = q_emb.reshape(1, -1)
3.   faiss.normalize_L2(q_emb)
4.    D, I = _index.search(q_emb, top_k)
5.   results = []
```

1. q_emb = compute_embedding(query)

compute_embedding folosește *SentenceTransformer(MODEL_NAME)* și model.encode([text]), apoi returnează un numpy.ndarray de tip float32.

q_emb este un de forma:

- (dim,) — de exemplu (384,) sau (768,) depinde de model

2. q_emb = q_emb.reshape(1, -1)

Transformă vectorul 1D într-o **matrice 2D cu un singur rând**.

- Înainte: q_emb.shape == (dim,)
- După: q_emb.shape == (1, dim)

-1 înseamnă „deduce automat dimensiunea potrivită”. În cazul tău devine dim. (Deci e un mod comod de a spune „fă-l 2D cu 1 rând”.)

FAISS .search() se așteaptă la o matrice de query-uri cu forma:

- (n_queries, dim)

Chiar dacă ai doar o întrebare, tot trebuie să fie 2D: (1, dim).

3 faiss.normalize_L2(q_emb)

Normalizează vectorul astfel încât norma lui L2 să fie 1:

Pe scurt: **vectori normalizați** sunt vectori care au fost **scalați astfel încât lungimea lor (norma) să fie 1**, fără să li se schimbe direcția.\ Mai jos îți explic **intuitiv**, **matematic** și **de ce sunt esențiali** exact în cazul tău (SBERT + FAISS + cosine similarity).

---

1) Ce este „lungimea” (norma) unui vector

Un vector este, matematic, o listă de numere:

v = [v1, v2, v3, ..., vD]

``

👉 **Norma L2** (lungimea vectorului) este:

$$\|v\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_D^2}$$

Intuitiv:

- norma = „cât de mare” este vectorul
- nu spune nimic despre **direcție**, doar despre **mărime**

---

2) Ce înseamnă „normalizare L2”

Un vector **normalizat L2** este vectorul original împărțit la norma sa:

$$v_{norm} = \frac{v}{\|v\|}$$

Rezultat:

- **direcția rămâne aceeași**
- **lungimea devine exact 1**

Exemplu simplu:

v = [3, 4]

norma = sqrt($3^2 + 4^2$) = 5

v_normalizat = [3/5, 4/5] = [0.6, 0.8]
norma(v_normalizat) = 1

---

3) Ce face faiss.normalize_L2(vectors) concret

Când scrii:

faiss.normalize_L2(arr)

FAISS:

- ia **fiecare vector**
- calculează norma lui
- împarte toate valorile la norma respectivă
- modifică vectorul **in-place**

✅ După acest pas:

- fiecare vector se află pe „sfera unitate”
- ||vector|| = 1

Același lucru îl faci și pentru query:

faiss.normalize*L2(q*emb)

---

4) De ce e CRUCIAL în proiectul tău (SBERT + FAISS)

In proiect este folosit:

idx = faiss.IndexFlatIP(dim)

IndexFlatIP = **Inner Product** (produs scalar).

Dar:

Produsul scalar „pur” depinde de:

- **direcție**
- **și lungime**

Formula:

$$v \cdot w = \|v\| \cdot \|w\| \cdot \cos(\theta)$$

Dacă NU normalizezi vectorii:

- vectori mai „lungi” (cu norme mai mari) pot părea mai similari

5) Ce se întâmplă DUPĂ normalizare

Dacă ambii vectori sunt normalizați:

$$\|v\| = 1,\ \|w\| = 1$$

Produsul scalar devine:

$$v \cdot w = \cos(\theta)$$

Asta înseamnă:

- scorul depinde DOAR de unghi (direcție)
- adică DOAR de **sensul semantic**
- nu de mărimea vectorului

De aceea combinația:

faiss.normalize*L2(arr)*

IndexFlatIP

*faiss.normalize*L2(q_emb)

este **echivalentă cu cosine similarity**, care este standardul de facto în semantic search cu SBERT.

6) Interpretarea scorurilor după normalizare

După normalizare + IP:

- score ≈ 1.0 → aproape identic semantic
- score ~ 0.7–0.8 → foarte similar
- score ~ 0.4–0.6 → parțial relevant
- score < 0.2 → probabil irelevant

De aceea e foarte logic să folosești ulterior un **threshold**:

```
if score < 0.25:
    continue
```

---

7) Intuiție vizuală (cea mai importantă)

Imaginează-ți:

- fiecare text = o săgeată din centru
- toate săgețile sunt normalizate → lungime 1
- diferența dintre texte = **unghiul dintre săgeți**

Căutarea semantică devine:

„care săgeți sunt orientate în direcția cea mai apropiată de săgeata întrebării?”

---

8) Ce s-ar întâmpla dacă NU ai normaliza

Fără normalizare:

- texte mai lungi sau mai „dense” semantic
- ar avea embedding-uri cu norme mai mari
- și ar fi returnate chiar dacă sensul e mai slab

Rezultatul:

- căutare INSTABILĂ
- scoruri greu de interpretat

- context zgomotos trimis LLM-ului

Produs scalar → „cât de mult merge una în direcția celeilalte”

Cosine similarity → „cât de asemănătoare este direcția”

Normalizare → „ignor diferența de mărime”
