RAG-based Legal & Policy Document Assistant

A source-grounded Retrieval-Augmented Generation assistant for legal and policy documents — web UI (React + Supabase auth) and a FastAPI backend with local FAISS vector store and OpenAI for answer synthesis.

Problem

Legal and policy documents are large, domain-specific, and frequently updated. Generic LLM answers lack provenance and precise grounding. This project provides short, source-grounded answers with document-level context and per-conversation history to produce reliable responses for non-expert users.

Key features

- RAG pipeline: FAISS vector store for global legal corpus + temporary per-PDF FAISS for on-the-fly PDF queries.
- Conversation history: stores per-user conversations and messages; prompts built from last N messages.
- Source-grounded PDF Q&A: upload PDF, combine local vector search and PDF chunking, return concise answers.
- Authentication: JWT-based login/signup with protected conversation and ask endpoints.
- Local-first vector store: FAISS index persisted under `backend/vectorstore/index.faiss`.
- Embeddings: sentence-transformers (`all-MiniLM-L6-v2`) for semantic retrieval.

Tech stack

- Frontend: React + TypeScript (Vite), Tailwind, Supabase client for auth/session.
- Backend: Python 3.11+ (FastAPI, Uvicorn), SQLAlchemy, PyPDF2, python-dotenv.
- Auth: JWT (HS256) implemented with `python-jose`; `bcrypt` via `passlib` for password hashing.
- Vector DB: FAISS (local, file-backed index).
- Embeddings: `sentence-transformers` (all-MiniLM-L6-v2).
- LLM: OpenAI chat completions (model: `gpt-4o-mini` used in code).

System architecture (high-level flow)

1. Frontend authenticates user via `/authenticate/login` -> receives `access_token` (JWT). Token saved in `localStorage`.
2. User selects/creates a conversation. Frontend calls `/conversations/*` with `Authorization: Bearer <token>`.
3. For a question: frontend POSTs `/ask/{conversation_id}` with `{ prompt }`. Backend:
    - builds prompt from conversation history + RAG context (global FAISS search) + optional PDF context
    - calls OpenAI, stores assistant + user messages, returns concise response
4. For PDF queries: frontend uploads multipart/form-data to `/ask_pdf/{conversation_id}` with file + `question`; backend creates a temporary FAISS index from PDF chunks, merges results with global

context, and queries the model.
5. Vector index is persisted on disk; preload script builds it from `backend/docs`.

API overview

Auth

- POST /authenticate/signup
  - Body: { full_name, email, password }
  - Response: { message }
- POST /authenticate/login
  - Body: { email, password }
  - Response: { access_token, token_type: 'bearer', full_name, new_conversation_id }

Conversations (protected — `Authorization: Bearer <token>`)

- GET /conversations/ -> list user's conversations (returns array of ConversationOut)
- POST /conversations/ -> create conversation
  - Body: { title? }
- GET /conversations/{conversation_id} -> conversation + messages
- GET /conversations/{conversation_id}/messages -> list messages
- POST /conversations/{conversation_id}/messages -> add message
  - Body: { role, content }

Model & RAG endpoints (protected)

- POST /ask/{conversation_id}
  - Body: { prompt }
  - Returns: { response }
- POST /ask_pdf/{conversation_id}
  - Multipart: `file` (PDF), `question` (string), `top_k` (int)
  - Returns: { answer }

Utility

- POST /embed_text -> { embedding }
- POST /embed_texts -> { embeddings }
- POST /query_docs -> query the global FAISS index (returns matched chunks)

Authentication flow

- Signup stores `password_hash` (bcrypt via passlib).
- Login validates password and returns a JWT signed with `JWT_SECRET` (HS256). Token TTL: 24 hours (config in `authenticate/auth.py`).
- Protected routes use FastAPI `HTTPBearer` dependency (`authenticate/dependencies.py`) which extracts and validates the token, then resolves `user.id` for request authorization.

Environment variables

Backend (.env)

- `OPENAI_API_KEY` — OpenAI API key used by `openai.OpenAI(api_key=...)`.
- `DATABASE_URL` — SQLAlchemy connection URL (e.g., `sqlite:///./db.sqlite` or Postgres URL).
- `JWT_SECRET` — HMAC secret for JWT signing (required).

Frontend (.env local / Vite)

- `VITE_API_BASE_URL` — backend base URL (default: `http://localhost:8000`).
- `VITE_SUPABASE_URL` — Supabase instance URL (used by client, optional if not using Supabase auth).
- `VITE_SUPABASE_PUBLISHABLE_KEY` — Supabase anon/public key.

Setup (backend)

1. Create virtual environment and install

```
python -m venv .venv
.venv\\Scripts\\activate   # Windows
pip install -r backend/requirements.txt
```

2. Create `.env` in `backend/` with the three required variables above.
3. Initialize DB and FAISS index

```
python backend/preload_docs.py   # builds vectorstore from backend/docs
```

4. Start dev server

```
uvicorn backend.main:app --reload --host 0.0.0.0 --port 8000
```

Setup (frontend)

1. From `frontend/` install and run

```
cd frontend
npm install
# create .env.local with VITE_API_BASE_URL and optional VITE_SUPABASE_*
values
npm run dev
```

How to run locally (quick steps)

1. Prepare backend `.env` with `OPENAI_API_KEY`, `DATABASE_URL`, `JWT_SECRET`.

2. Build vectorstore: `python backend/preload_docs.py` (optional if `backend/vectorstore/index.faiss` already present).
3. Start backend: `uvicorn backend.main:app --reload`.
4. Start frontend: `cd frontend && npm run dev`.
5. Visit frontend URL (Vite dev server) and sign up / login to test flows.

Folder structure

- backend/
  - main.py # FastAPI app + endpoints
  - preload_docs.py # build persistent FAISS index from `backend/docs`
  - embeddings.py # sentence-transformers wrapper
  - vectorstore.py # FAISS index wrapper (persist/load/search)
  - database.py # SQLAlchemy engine & session
  - authenticate/ # auth, JWT, dependencies, models, schemas
  - conversation/ # conversation routes + schemas
  - docs/ # source documents for vectorstore
  - vectorstore/ # persisted FAISS files (index.faiss)
- frontend/
  - src/ # React app, API client in `src/lib/api.ts`
  - integrations/supabase # Supabase client configuration

Future improvements / roadmap

- Replace OpenAI API calls with an adapter layer to support multiple LLM backends and local LLMs.
- Add vector store replication/backup and optional cloud vector DB (Pinecone/Weaviate) adapter.
- Add role-based access control and per-document permissions.
- Improve prompt engineering: token-aware summarization and chunk scoring.
- CI: tests for prompt/response pipeline, contract tests for front/back API compatibility.

Not boilerplate

This repository implements a focused RAG pipeline with conversation state, PDF-on-the-fly FAISS indexing, and JWT-protected endpoints wired end-to-end (frontend <> backend <> FAISS <> OpenAI). The code is application-specific — not a generic scaffold.