```python
# docuchat_backend/app/routers/health.py
from __future__ import annotations

import os, inspect
from fastapi import APIRouter
from sqlalchemy import text

from ..database import engine
from .. import rag  # only for Chroma ensure; NOT used for OpenAI
from openai import OpenAI
import openai as openai_pkg
import httpx

router = APIRouter(prefix="/health", tags=["health"])

def _make_openai_client() -> OpenAI:
    api_key = os.getenv("OPENAI_API_KEY")
    if not api_key:
        raise RuntimeError("missing_api_key")

    # optional proxy via httpx (new SDK does NOT support 'proxies=' kwarg)
    proxy = os.getenv("HTTPS_PROXY") or os.getenv("HTTP_PROXY")
    if os.getenv("DOCUCHAT_DISABLE_PROXY") == "1" or not proxy:
        return OpenAI(api_key=api_key)

    http_client = httpx.Client(proxies=proxy, timeout=30.0)
```

```
27      return OpenAI(api_key=api_key,
    http_client=http_client)
28
29  @router.get("/")
30  def health():
31      status = {
32          "database": "unknown",
33          "chroma": "unknown",
34          "openai": "unknown",
35          "openai_module": getattr(
    openai_pkg, "__file__", "n/a"),
36          "openai_version": getattr(
    openai_pkg, "__version__", "n/a"),
37          "collection": os.getenv("
    CHROMA_COLLECTION", "docuchat"),
38          "model": os.getenv("OPENAI_MODEL"
    , "gpt-4o-mini"),
39          "impl": "routers.health.v2",  # so
     we know THIS file is live
40      }
41
42      # 1) DB check
43      try:
44          with engine.connect() as conn:
45              conn.execute(text("SELECT 1"))
46          status["database"] = "ok"
47      except Exception as e:
48          status["database"] = f"error: {e!r
    }"
49
50      # 2) Chroma check
51      try:
52          coll = rag._ensure_collection()
```

```
53          _ = coll.count()
54          status["chroma"] = "ok"
55      except Exception as e:
56          status["chroma"] = f"error: {e!r}"
57
58      # 3) OpenAI check (new SDK only, NO
    proxies kwarg anywhere)
59      try:
60          client = _make_openai_client()
61          resp = client.chat.completions.
    create(
62              model=os.getenv("OPENAI_MODEL"
    , "gpt-4o-mini"),
63              messages=[{"role": "user", "
    content": "Say OK"}],
64              max_tokens=1,
65              temperature=0,
66          )
67          _ = resp.choices[0].message.
    content
68          status["openai"] = "ok"
69      except Exception as e:
70          status["openai"] = f"error: {e!r}"
71
72      return status
73
```

```python
# reset_user.py
# one-time helper to (re)create a user with a properly hashed password

from docuchat_backend.app.database import SessionLocal
from docuchat_backend.app import models
from docuchat_backend.app.auth import get_password_hash

EMAIL = "umit.ipek@gmail.com"
PLAINTEXT_PASSWORD = "12345678"  # this is what you'll log in with

db = SessionLocal()

try:
    user = db.query(models.User).filter(models.User.email == EMAIL).first()
    if user:
        print(f"User {EMAIL} exists. Updating password hash.")
        user.password_hash = get_password_hash(PLAINTEXT_PASSWORD)
    else:
        print(f"User {EMAIL} not found. Creating new user.")
        user = models.User(
            email=EMAIL,
            password_hash=get_password_hash(PLAINTEXT_PASSWORD),
        )
        db.add(user)
```

```
25
26      db.commit()
27      print("Done. You can now log in with
    that email + password.")
28  finally:
29      db.close()
30
```

```python
import streamlit as st
import requests
import json
import os
from typing import Optional, List

API_BASE = "http://127.0.0.1:8000"

st.set_page_config(page_title="DocuChat", layout="centered")


# --- session state helpers ---
def init_session():
    if "token" not in st.session_state:
        st.session_state.token = None
    if "documents" not in st.session_state:
        st.session_state.documents = []
    if "selected_doc_ids" not in st.session_state:
        st.session_state.selected_doc_ids = []
    if "health" not in st.session_state:
        st.session_state.health = None


def do_login(email: str, password: str) -> Optional[str]:
    """
    Call /auth/login and return the JWT access token if successful, else None.
    """
```

```
28      try:
29          resp = requests.post(
30              f"{API_BASE}/auth/login",
31              data={"username": email, "
   password": password},
32              headers={"Content-Type": "
   application/x-www-form-urlencoded"},
33              timeout=30,
34          )
35          if resp.status_code == 200:
36              data = resp.json()
37              return data.get("access_token"
   )
38          else:
39              st.error(f"Login failed ({resp
   .status_code}): {resp.text}")
40              return None
41      except Exception as e:
42          st.error(f"Login error: {e}")
43          return None
44
45
46  def fetch_documents(token: str) -> List[
   dict]:
47      """
48      Fetch the current user's documents
   from /documents/
49      """
50      try:
51          resp = requests.get(
52              f"{API_BASE}/documents/",
53              headers={"Authorization": f"
   Bearer {token}"},
```

```
54              timeout=30,
55          )
56          if resp.status_code == 200:
57              return resp.json()
58          else:
59              st.error(f"Failed to load
   documents ({resp.status_code}): {resp.text
   }")
60              return []
61      except Exception as e:
62          st.error(f"Error fetching
   documents: {e}")
63          return []
64
65
66  def fetch_health(token: str) -> dict:
67      """
68      Get backend health (/health/) to show
   model/chroma/db status.
69      """
70      try:
71          resp = requests.get(
72              f"{API_BASE}/health/",
73              headers={"Authorization": f"
   Bearer {token}"},
74              timeout=10,
75          )
76          if resp.status_code == 200:
77              return resp.json()
78          else:
79              return {"error": f"{resp.
   status_code} {resp.text}"}
80      except Exception as e:
```

```
        return {"error": str(e)}


def upload_file(token: str, uploaded_file) -> Optional[dict]:
    """
    Send file -> /documents/upload with long timeout for big PDFs.
    """
    headers = {
        "Authorization": f"Bearer {token}",
    }

    files = {
        "file": (
            uploaded_file.name,
            uploaded_file.getvalue(),
            uploaded_file.type or "application/octet-stream",
        )
    }

    try:
        resp = requests.post(
            f"{API_BASE}/documents/upload",
            headers=headers,
            files=files,
            timeout=300,  # bumped so 10-30MB PDFs won't time out
        )
        if resp.status_code == 201:
```

```
108             return resp.json()
109         else:
110             st.error(f"Upload failed ({
    resp.status_code}): {resp.text}")
111             return None
112     except requests.exceptions.
    ReadTimeout:
113         st.error("Upload timed out while
    waiting for the server (still processing
    ?). Try again or use a smaller file.")
114         return None
115     except Exception as e:
116         st.error(f"Upload error: {e}")
117         return None
118 
119 
120 def ask_question(token: str, question:
    str, doc_ids: List[int], top_k: int = 3
    ) -> Optional[dict]:
121     """
122     Call /query (POST) with question,
    selected document IDs, top_k.
123     """
124     headers = {
125         "Authorization": f"Bearer {token}
    ",
126         "Content-Type": "application/json
    ",
127     }
128 
129     body = {
130         "question": question,
131         "document_ids": doc_ids,
```

```
132             "top_k": top_k,
133         }
134
135         try:
136             resp = requests.post(
137                 f"{API_BASE}/query/",
138                 headers=headers,
139                 data=json.dumps(body),
140                 timeout=60,
141             )
142             if resp.status_code == 200:
143                 return resp.json()
144             else:
145                 st.error(f"Query failed ({
    resp.status_code}): {resp.text}")
146                 return None
147         except Exception as e:
148             st.error(f"Query error: {e}")
149             return None
150
151
152 # -------------- UI starts here
     --------------
153 init_session()
154
155 st.title("DocuChat ")
156
157 # If not logged in: show login form only
158 if not st.session_state.token:
159     st.subheader("Login")
160
161     email = st.text_input("Email", value=
    "umit.ipek@gmail.com", autocomplete="
```

```
username")
    password = st.text_input("Password", type="password", value="12345678", autocomplete="current-password")

    if st.button("Sign in"):
        token = do_login(email, password)
        if token:
            st.session_state.token = token
            # fetch docs + health once on login
            st.session_state.documents = fetch_documents(token)
            st.session_state.health = fetch_health(token)
            st.success("Logged in ☑")
            st.rerun()

    st.caption("Enter your credentials and click Sign in to continue.")
    st.stop()

# If logged in: main app
tab_files, tab_ask, tab_status = st.tabs(["▯ Files", "❓Ask", "▯ Status"])

# --- Tab 1: Files ---
with tab_files:
    st.header("Upload & Manage Files")

    uploaded_file = st.file_uploader(
        "Choose a file to upload (PDF,
```

```
185 TXT, DOCX). Up to ~30MB.",
186         type=["pdf", "txt", "docx", "md"
    ],
187     )
188 
189     if st.button("Upload file"):
190         if not uploaded_file:
191             st.warning("Please select a
    file first.")
192         else:
193             with st.spinner("Uploading &
    indexing... this can take a minute for
    big PDFs ⏳"):
194                 result = upload_file(st.
    session_state.token, uploaded_file)
195             if result:
196                 st.success(f"Uploaded: {
    result.get('filename')} (id={result.get('
    id')})")
197                 # Refresh doc list
198                 st.session_state.
    documents = fetch_documents(st.
    session_state.token)
199 
200     st.subheader("Your documents")
201     if not st.session_state.documents:
202         st.info("No documents yet.")
203     else:
204         # make a lookup dict for display
205         doc_label_map = {
206             d["id"]: f"{d['id']} - {d['
    filename']} ({d['status']})"
207             for d in st.session_state.
```

```
207 documents
208         }
209 
210         # multiselect for which docs to
    use in questions
211         current_selection_labels = [
212             doc_label_map.get(doc_id, f"{
    doc_id}")
213             for doc_id in st.
    session_state.selected_doc_ids
214         ]
215 
216         chosen = st.multiselect(
217             "Select documents to use for
    Q&A",
218             options=list(doc_label_map.
    values()),
219             default=
    current_selection_labels,
220         )
221 
222         # reverse map labels -> ids
223         chosen_ids = []
224         for label in chosen:
225             # label format: "18 - file.
    pdf (ready)"
226             first_part = label.split(
    " - ")[0].strip()
227             try:
228                 chosen_ids.append(int(
    first_part))
229             except:
230                 pass
```

```
231
232         st.session_state.selected_doc_ids
     = chosen_ids
233
234         st.write("Currently selected doc
    IDs:", st.session_state.selected_doc_ids)
235
236 # --- Tab 2: Ask ---
237 with tab_ask:
238     st.header("Ask Your Documents")
239
240     if not st.session_state.
    selected_doc_ids:
241         st.info("Select one or more
    documents in the Files tab first.")
242     else:
243         question = st.text_area("Your
    question:", "Summarize in two sentences."
    )
244         top_k = st.slider("How many
    relevant chunks to retrieve (top_k)", 1,
    8, 3)
245
246         if st.button("Ask"):
247             with st.spinner("Thinking..."
    ):
248                 answer = ask_question(
249                     st.session_state.
    token,
250                     question,
251                     st.session_state.
    selected_doc_ids,
252                     top_k=top_k,
```

```
                )
            if answer:
                st.subheader("Answer")
                st.write(answer.get("answer", ""))

                st.subheader("Citations")
                st.json(answer.get("citations", []))

# --- Tab 3: Status ---
with tab_status:
    st.header("Backend status")
    st.write("This checks `/health/` on the FastAPI server.")

    if st.button("Refresh status"):
        st.session_state.health = fetch_health(st.session_state.token)

    if st.session_state.health:
        st.json(st.session_state.health)
    else:
        st.info("No status yet. Click Refresh status.")

```

```
fastapi==0.115.0
uvicorn==0.32.0
sqlalchemy==2.0.36
python-jose==3.3.0
passlib[bcrypt]==1.7.4
chromadb==0.4.24
sentence-transformers==3.2.1
pdfplumber==0.11.3
python-docx==1.1.2
openai==1.52.2
pydantic==2.9.2
python-multipart==0.0.12
tqdm==4.67.1
anyio==4.11.0
typing-extensions>=4.11

```

```python
"""DocuChat FastAPI application entry point.

This module creates the FastAPI app, includes routers and sets up
middleware.  To run locally, execute ``uvicorn app.main:app --reload``.
"""
from fastapi import FastAPI

from .routers import auth as auth_router
from .routers import documents as documents_router
from .routers import query as query_router
from .routers import health as health_router
app = FastAPI(title="DocuChat API", version="0.1.0")

# Include API routers
app.include_router(auth_router.router)
app.include_router(documents_router.router)
app.include_router(query_router.router)
app.include_router(health_router.router)

@app.get("/", tags=["health"])
def read_root():
    """Health check endpoint."""
    return {"status": "ok"}

```

```python
# docuchat_backend/app/auth.py
from __future__ import annotations

import os
import logging
from datetime import datetime, timedelta, timezone
from typing import Optional

from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from jose import jwt, JWTError
from passlib.context import CryptContext
from sqlalchemy.orm import Session

from .database import SessionLocal
from . import models

logger = logging.getLogger("docuchat.auth")
logger.setLevel(logging.INFO)

# ------------------------------
# Password hashing
# ------------------------------
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def verify_password(plain_password: str, hashed_password: str) -> bool:
    return pwd_context.verify(
```

```
plain_password, hashed_password)

def get_password_hash(password: str) -> str:
    return pwd_context.hash(password)

# ------------------------------
# DB dependency
# ------------------------------
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# ------------------------------
# JWT config
# ------------------------------
SECRET_KEY = os.getenv("JWT_SECRET", "change-this-in-production")
ALGORITHM = os.getenv("JWT_ALG", "HS256")
ACCESS_TOKEN_EXPIRE_MINUTES = 60  # 1 hour

def create_access_token(data: dict, expires_delta: Optional[timedelta] = None) -> str:
    to_encode = data.copy()
    expire = datetime.now(timezone.utc) + (
        expires_delta if expires_delta else timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
```

```
    )
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

# ------------------------------
# Authenticate user for /auth/login
# ------------------------------
def authenticate_user(db: Session, email: str, password: str) -> Optional[models.User]:
    user = db.query(models.User).filter(models.User.email == email).first()
    if not user:
        return None
    if not verify_password(password, user.password_hash):
        return None
    return user

# ------------------------------
# Bearer token dependency for protected routes
# ------------------------------
bearer_scheme = HTTPBearer(auto_error=False)

def get_current_user(
    db: Session = Depends(get_db),
    credentials: HTTPAuthorizationCredentials = Depends(bearer_scheme),
```

```
77 ) -> models.User:
78     """
79     - Pulls Authorization header ("Bearer
    <token>")
80     - Decodes the JWT
81     - Loads that user from DB
82     - Raises 401 if anything fails
83     """
84     if credentials is None:
85         logger.warning("get_current_user
   : no credentials provided")
86         raise HTTPException(
87             status_code=status.
   HTTP_401_UNAUTHORIZED,
88             detail="Not authenticated",
89         )
90 
91     token = credentials.credentials
92     logger.info(f"get_current_user: got
   token (len={len(token)})")
93 
94     try:
95         payload = jwt.decode(token,
   SECRET_KEY, algorithms=[ALGORITHM])
96         logger.info(f"get_current_user:
   decoded payload {payload}")
97     except JWTError as e:
98         logger.warning(f"get_current_user
   : JWT decode failed: {e}")
99         raise HTTPException(
100             status_code=status.
    HTTP_401_UNAUTHORIZED,
101             detail="Invalid token",
```

```
        )

    email: str = payload.get("sub")
    if email is None:
        logger.warning("get_current_user: 'sub' missing in JWT payload")
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid token payload",
        )

    user = db.query(models.User).filter(models.User.email == email).first()
    if not user:
        logger.warning(f"get_current_user: user {email} not found in DB")
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="User not found",
        )

    # success ✅
    return user

```

```python
"""SQLAlchemy models for DocuChat.

Defines the database schema for users, documents, collections and chat
history.  Uses SQLAlchemy's declarative base from ``app.database``.
"""
from datetime import datetime
from sqlalchemy import Column, Integer, String, ForeignKey, Text, DateTime
from sqlalchemy.orm import relationship
from .database import Base

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    email = Column(String(255), unique=True, nullable=False, index=True)
    password_hash = Column(String(255), nullable=False)
    created_at = Column(DateTime, default=datetime.utcnow)
    last_login = Column(DateTime, nullable=True)
    documents = relationship("Document", back_populates="owner", cascade="all, delete-orphan")
    collections = relationship("Collection", back_populates="owner", cascade="all, delete-orphan")
    chat_history = relationship("ChatHistory", back_populates="owner",
```

```
20 cascade="all, delete-orphan")
21
22 class Collection(Base):
23     __tablename__ = "collections"
24     id = Column(Integer, primary_key=True
   , index=True)
25     user_id = Column(Integer, ForeignKey("
   users.id", ondelete="CASCADE"), nullable=
   False)
26     name = Column(String(255), nullable=
   False)
27     created_at = Column(DateTime, default=
   datetime.utcnow)
28     owner = relationship("User",
   back_populates="collections")
29     documents = relationship("Document",
   back_populates="collection", cascade="all
   , delete-orphan")
30
31 class Document(Base):
32     __tablename__ = "documents"
33     id = Column(Integer, primary_key=True
   , index=True)
34     user_id = Column(Integer, ForeignKey("
   users.id", ondelete="CASCADE"), nullable=
   False)
35     filename = Column(String(255),
   nullable=False)
36     file_path = Column(String(500),
   nullable=False)
37     file_size = Column(Integer, nullable=
   True)
38     file_type = Column(String(50),
```

```
38 nullable=True)
39     status = Column(String(50), default="
   processing")
40     summary = Column(Text, nullable=True)
41     upload_date = Column(DateTime, default
   =datetime.utcnow)
42     collection_id = Column(Integer,
   ForeignKey("collections.id", ondelete="SET
    NULL"), nullable=True)
43     owner = relationship("User",
   back_populates="documents")
44     collection = relationship("Collection"
   , back_populates="documents")
45     chat_history = relationship("
   ChatHistory", back_populates="document",
   cascade="all, delete-orphan")
46
47 class ChatHistory(Base):
48     __tablename__ = "chat_history"
49     id = Column(Integer, primary_key=True
   , index=True)
50     user_id = Column(Integer, ForeignKey("
   users.id", ondelete="CASCADE"), nullable=
   False)
51     document_id = Column(Integer,
   ForeignKey("documents.id", ondelete="
   CASCADE"), nullable=False)
52     question = Column(Text, nullable=False
   )
53     answer = Column(Text, nullable=False)
54     sources = Column(Text, nullable=True)
55     rating = Column(Integer, nullable=True
   )
```

```
56      created_at = Column(DateTime, default=
    datetime.utcnow)
57      owner = relationship("User",
    back_populates="chat_history")
58      document = relationship("Document",
    back_populates="chat_history")
59
```

```
1 """Core configuration and utility
    functions for DocuChat."""
2
```

```python
# docuchat_backend/app/routers/query.py
from __future__ import annotations

from fastapi import APIRouter, Depends, HTTPException
from pydantic import BaseModel, Field
from typing import List, Optional

from .. import rag, auth, models

router = APIRouter(prefix="/query", tags=["query"])

class QueryRequest(BaseModel):
    question: str = Field(..., description="Natural language question to ask your docs")
    document_ids: Optional[List[int]] = Field(None, description="List of document IDs to restrict search")
    top_k: int = Field(4, description="How many chunks to retrieve internally")

class QueryResponse(BaseModel):
    answer: str
    citations: Optional[List[dict]]

@router.post("/", response_model=QueryResponse)
def ask_question(
    req: QueryRequest,
    current_user: models.User = Depends(auth.get_current_user),
```

```
):
    """
    Auth required. Runs semantic search + LLM answer (if API key is set).
    """
    try:
        result = rag.query_documents(
            question=req.question,
            document_ids=req.document_ids,
            top_k=req.top_k,
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

    # result should be {"answer": ..., "citations": ...}
    return QueryResponse(
        answer=result["answer"],
        citations=result["citations"],
    )

```

```python
"""
DocuChat Backend Package

This package contains the FastAPI application and supporting modules for
user authentication, database models and API routes.
"""

from .main import app  # noqa: F401

```

```python
"""Database configuration and session management for DocuChat.

This module defines the SQLAlchemy engine and session maker.  It reads configuration from environment variables with sensible defaults, and exports a dependency that yields a database session per request.
"""
import os
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, DeclarativeBase

def get_database_url() -> str:
    """Construct the database URL from environment variables.

    For local development, fall back to an SQLite database so that the backend can start without external dependencies.  In production, override `DATABASE_URL` with a PostgreSQL connection string.
    """
    return os.getenv("DATABASE_URL", "sqlite:///./docuchat.db")

# Create the SQLAlchemy engine.  For SQLite, we need check_same_thread=False
# because FastAPI runs each request in a separate thread.
```

```
22 DATABASE_URL = get_database_url()
23 connect_args = {}  # extra arguments for
   create_engine
24 if DATABASE_URL.startswith("sqlite"):  #
   pragma: no cover
25     connect_args = {"check_same_thread":
   False}
26
27 engine = create_engine(DATABASE_URL,
   connect_args=connect_args)
28
29 class Base(DeclarativeBase):
30     """Base class for SQLAlchemy models
   ."""
31
32 # Create a configured "Session" class
33 SessionLocal = sessionmaker(autocommit=
   False, autoflush=False, bind=engine)
34
35 def get_db():
36     """FastAPI dependency that provides a
   database session per request.
37
38     Yields a SQLAlchemy session and
   ensures it is closed after the request
39     is complete, even if an exception
   occurs.
40     """
41     db = SessionLocal()
42     try:
43         yield db
44     finally:
45         db.close()
```

```
1 """API routers for DocuChat.
2
3 Routers are grouped by feature area.  They
   are included in the main
4 application in ``app.main``.
5 """
6
```

```python
"""File storage utilities.

This module abstracts saving uploaded files to the local filesystem.  In production you might replace this with an S3 or Google Cloud Storage client.  Filenames are randomised to prevent collisions and to avoid disclosing the original filenames on disk.
"""
import os
import uuid
from fastapi import UploadFile

UPLOAD_DIR = os.getenv("UPLOAD_DIR", "uploaded_files")

def ensure_upload_dir() -> None:
    """Ensure that the upload directory exists."""
    os.makedirs(UPLOAD_DIR, exist_ok=True)

def save_upload_file(upload_file: UploadFile) -> str:
    """Save an uploaded file and return the file path."""
    ensure_upload_dir()
    _, ext = os.path.splitext(upload_file.filename)
    unique_name = f"{uuid.uuid4().hex}{ext}"
    file_path = os.path.join(UPLOAD_DIR, unique_name)
```

```
    with open(file_path, "wb") as buffer:
        while True:
            contents = upload_file.file.read(1024 * 1024)
            if not contents:
                break
            buffer.write(contents)
    return file_path

def delete_file(path: str) -> None:
    """Delete a file from the filesystem."""
    try:
        os.remove(path)
    except FileNotFoundError:
        pass

```

```python
"""Application configuration settings.

Values can be overridden via environment variables.  For example,
``SECRET_KEY`` controls JWT token signing, and ``ACCESS_TOKEN_EXPIRE_MINUTES``
defines how long login tokens remain valid.
"""
import os

class Settings:
    # Cryptographic key used to sign JWT tokens.  In production, set this to a
    # long, random string via an environment variable.
    SECRET_KEY: str = os.getenv("SECRET_KEY", "CHANGEME_SUPER_SECRET")
    ALGORITHM: str = os.getenv("JWT_ALGORITHM", "HS256")
    ACCESS_TOKEN_EXPIRE_MINUTES: int = int(os.getenv("ACCESS_TOKEN_EXPIRE_MINUTES", "30"))

settings = Settings()

```

```python
# docuchat_backend/app/rag.py
from __future__ import annotations

import os
import io
import logging
from typing import List, Optional, Dict, Any

logger = logging.getLogger("docuchat.rag")
logger.setLevel(logging.INFO)

# ---------- external libs (optional imports with graceful fallback) ----------

# text PDFs
try:
    import pdfplumber
except Exception:
    pdfplumber = None  # type: ignore

# docx
try:
    import docx  # python-docx
except Exception:
    docx = None  # type: ignore

# PyPDF2 as a second text extractor
try:
    import PyPDF2
except Exception:
    PyPDF2 = None  # type: ignore

```

```
32 # OCR stack
33 try:
34     import pytesseract
35 except Exception:
36     pytesseract = None  # type: ignore
37
38 try:
39     from pdf2image import
   convert_from_path
40 except Exception:
41     convert_from_path = None  # type:
   ignore
42
43 try:
44     from PIL import Image  # pillow
45 except Exception:
46     Image = None  # type: ignore
47
48 # chroma
49 try:
50     import chromadb
51 except Exception as e:
52     chromadb = None  # type: ignore
53     logger.error("Chroma import failed: %s
   ", e)
54
55 # embeddings
56 try:
57     from sentence_transformers import
   SentenceTransformer
58 except Exception as e:
59     SentenceTransformer = None  # type:
   ignore
```

```
60      logger.error("sentence-transformers
    import failed: %s", e)
61
62  # OpenAI chat model
63  _OPENAI_AVAILABLE = False
64  try:
65      from openai import OpenAI
66      _OPENAI_AVAILABLE = True
67  except Exception:
68      _OPENAI_AVAILABLE = False
69
70
71  # ---------- env / config ----------
72
73  CHROMA_DB_DIR = os.getenv("CHROMA_DB_DIR"
    , "./docuchat_backend/chroma_db")
74  CHROMA_COLLECTION = os.getenv("
    CHROMA_COLLECTION", "docuchat")
75  EMBEDDING_MODEL_NAME = os.getenv("
    EMBEDDING_MODEL", "sentence-transformers/
    all-MiniLM-L6-v2")
76
77  OPENAI_API_KEY = os.getenv("OPENAI_API_KEY
    ")
78  OPENAI_MODEL = os.getenv("OPENAI_MODEL", "
    gpt-4o-mini")
79
80  # IMPORTANT: update these two for your
    Windows install paths.
81  # If you add tesseract/poppler to PATH
    later, you can set them to None.
82  TESSERACT_EXE = r"C:\Program Files\
    Tesseract-OCR\tesseract.exe"
```

```
 83 POPLER_BIN_DIR = r"C:\Tools\poppler-25.07.0\Library\bin"  # <- example; update to your actual poppler bin folder
 84
 85 # ---------- globals ----------
 86 _COLLECTION = None
 87 _S_BERT = None
 88
 89
 90 # ---------- helpers: embeddings ----------
 91
 92 def _load_sbert() -> SentenceTransformer | None:
 93     """lazy-load embedding model"""
 94     global _S_BERT
 95     if _S_BERT is not None:
 96         return _S_BERT
 97     if SentenceTransformer is None:
 98         logger.warning("sentence-transformers not available; RAG will degrade.")
 99         return None
100     logger.info(f"Loading embeddings model: {EMBEDDING_MODEL_NAME}")
101     _S_BERT = SentenceTransformer(EMBEDDING_MODEL_NAME)
102     return _S_BERT
103
104
105 def _embed_texts(texts: List[str]) -> List[List[float]]:
106     """
```

```
107     Turn text chunks into embeddings via sentence-transformers.
108     If unavailable, fallback to dummy vectors.
109     """
110     model = _load_sbert()
111     if model is None:
112         logger.warning("Embeddings model missing; using zero vectors.")
113         return [[0.0] * 384 for _ in texts]
114 
115     vecs = model.encode(
116         texts,
117         convert_to_tensor=False,
118         normalize_embeddings=True,
119     )
120     return vecs.tolist()  # type: ignore
121 
122 
123 # ---------- helpers: chroma ----------
124 
125 def _ensure_collection():
126     """Return / create persistent Chroma collection."""
127     if chromadb is None:
128         raise RuntimeError("ChromaDB is not installed.")
129 
130     global _COLLECTION
131     if _COLLECTION is not None:
132         return _COLLECTION
133 
```

```
134     os.makedirs(CHROMA_DB_DIR, exist_ok=
    True)
135     logger.info(f"Opening Chroma at {
    CHROMA_DB_DIR!r}, collection={
    CHROMA_COLLECTION!r}")
136     client = chromadb.PersistentClient(
    path=CHROMA_DB_DIR)
137
138     try:
139         collection = client.
    get_collection(CHROMA_COLLECTION)
140     except ValueError:
141         collection = client.
    create_collection(CHROMA_COLLECTION)
142
143     _COLLECTION = collection
144     return _COLLECTION
145
146
147 # ---------- helpers: text splitting
     ----------
148
149 def _split_text(text: str, chunk_size:
    int = 900, chunk_overlap: int = 150) ->
    List[str]:
150     text = text.strip().replace("\r\n", "
    \n")
151     if not text:
152         return []
153     out: List[str] = []
154     start = 0
155     n = len(text)
156     while start < n:
```

```
157         end = min(n, start + chunk_size)
158         chunk = text[start:end]
159         out.append(chunk)
160         if end == n:
161             break
162         start = max(end - chunk_overlap,
    start + 1)
163     return out
164
165
166 # ---------- helpers: PDF extraction
    paths ----------
167
168 def _extract_pdf_text_pdfplumber(path:
    str) -> str:
169     """Try pdfplumber to pull machine-
    readable text."""
170     if pdfplumber is None:
171         return ""
172     try:
173         parts: List[str] = []
174         with pdfplumber.open(path) as pdf
    :
175             for page in pdf.pages:
176                 try:
177                     t = page.extract_text
    () or ""
178                 except Exception:
179                     t = ""
180                 if t.strip():
181                     parts.append(t)
182         return "\n\n".join(parts).strip()
183     except Exception as e:
```

```
184         logger.warning("pdfplumber failed
    on %s: %s", path, e)
185         return ""
186
187
188 def _extract_pdf_text_pypdf2(path: str
    ) -> str:
189     """Fallback using PyPDF2 .
    extract_text(). Good for some digital
    PDFs."""
190     if PyPDF2 is None:
191         return ""
192     try:
193         parts: List[str] = []
194         with open(path, "rb") as f:
195             reader = PyPDF2.PdfReader(f)
196             for page in reader.pages:
197                 try:
198                     t = page.extract_text
    () or ""
199                 except Exception:
200                     t = ""
201                 if t.strip():
202                     parts.append(t)
203         return "\n\n".join(parts).strip()
204     except Exception as e:
205         logger.warning("PyPDF2 failed on
    %s: %s", path, e)
206         return ""
207
208
209 def _ocr_pdf_pages(path: str) -> str:
210     """
```

```
211     LAST RESORT: OCR every page image with Tesseract.
212
213     Requires:
214     - pdf2image.convert_from_path()
215     - pytesseract
216     - Tesseract binary
217     - Poppler (for pdf2image on Windows)
218     """
219     if convert_from_path is None or pytesseract is None:
220         return ""
221
222     # configure external binaries if needed
223     # (if you added them to PATH already, you can skip this)
224     if TESSERACT_EXE and hasattr(pytesseract, "pytesseract"):
225         pytesseract.pytesseract.tesseract_cmd = TESSERACT_EXE  # point to tesseract.exe
226
227     poppler_kw = {}
228     if POPLER_BIN_DIR and convert_from_path is not None:
229         poppler_kw["poppler_path"] = POPLER_BIN_DIR
230
231     try:
232         # convert PDF pages -> list of PIL Images
233         images = convert_from_path(path
```

```
233 , **poppler_kw)  # this can be slow for
    huge PDFs
234     except Exception as e:
235         logger.warning("pdf2image failed
    to rasterize %s: %s", path, e)
236         return ""
237 
238     ocr_parts: List[str] = []
239     for i, img in enumerate(images):
240         try:
241             text = pytesseract.
    image_to_string(img, lang="eng") or ""
242         except Exception as e:
243             logger.warning("pytesseract
    failed on page %s of %s: %s", i, path, e)
244             text = ""
245         if text.strip():
246             ocr_parts.append(text.strip
    ())
247 
248     return "\n\n".join(ocr_parts).strip()
249 
250 
251 # ---------- helpers: DOCX / fallback
     ----------
252 
253 def _read_docx(path: str) -> str:
254     if docx is None:
255         raise RuntimeError("python-docx
    not installed; run: pip install python-
    docx")
256     try:
257         d = docx.Document(path)
```

```
258         txt = "\n".join(p.text for p in d
    .paragraphs)
259         return txt.strip() or "[
    NO_EXTRACTED_TEXT_FROM_DOCX]"
260     except Exception as e:
261         logger.warning("DOCX read failed
    for %s: %s", path, e)
262         return "[DOCX_READ_ERROR]"
263
264
265 # ---------- master extractor ----------
266
267 def _read_file_text(file_path: str) ->
    str:
268     """
269     Unified extractor with OCR fallback.
270     1. direct text (.txt, .md, .log)
271     2. pdfplumber -> PyPDF2 -> OCR
272     3. docx
273     4. best-effort bytes decode
274     """
275     ext = os.path.splitext(file_path)[1].
    lower()
276
277     # plain text-ish
278     if ext in [".txt", ".md", ".log"]:
279         with open(file_path, "r",
    encoding="utf-8", errors="ignore") as f:
280             return f.read().strip()
281
282     # pdf
283     if ext == ".pdf":
284         # step 1: machine-readable
```

```
285         text =
    _extract_pdf_text_pdfplumber(file_path)
286         if not text.strip():
287             text =
    _extract_pdf_text_pypdf2(file_path)
288 
289         # step 2: OCR fallback
290         if (not text.strip()) or text.
    strip().startswith("[NO_EXTRACTED_TEXT"):
291             logger.info("Attempting OCR
    for PDF with little/no text: %s",
    file_path)
292             ocr_text = _ocr_pdf_pages(
    file_path)
293             if ocr_text.strip():
294                 text = ocr_text
295 
296         if not text.strip():
297             text = "[
    NO_EXTRACTED_TEXT_FROM_PDF]"
298         return text
299 
300     # word
301     if ext in [".docx", ".doc"]:
302         return _read_docx(file_path)
303 
304     # fallback raw bytes
305     with open(file_path, "rb") as f:
306         raw = f.read()
307     for enc in ("utf-8", "latin-1"):
308         try:
309             decoded = raw.decode(enc)
310             if decoded.strip():
```

```
311                 return decoded
312         except Exception:
313             pass
314
315     return "[BINARY_OR_EMPTY]"
316
317
318 # ---------- answer helpers ----------
319
320 def _build_prompt(question: str, contexts
    : List[str]) -> str:
321     ctx = "\n\n---\n\n".join(contexts)
322     return (
323         "You are a helpful assistant.
    Answer ONLY from the provided context. "
324         "If the answer is not in the
    context, say you don't know.\n\n"
325         f"Context:\n{ctx}\n\n"
326         f"Question: {question}\n\n"
327         "Answer:"
328     )
329
330
331 def _answer_with_openai(question: str,
    contexts: List[str]) -> str:
332     prompt = _build_prompt(question,
    contexts)
333
334     client = OpenAI(api_key=
    OPENAI_API_KEY)
335     resp = client.chat.completions.create
    (
336         model=OPENAI_MODEL,
```

```
337         messages=[{"role": "user", "
    content": prompt}],
338         temperature=0.2,
339     )
340     return resp.choices[0].message.
    content.strip()
341
342
343 def _answer_extractive(hits: List[Dict[
    str, Any]]) -> str:
344     parts = []
345     for h in hits:
346         meta = h.get("metadata", {})
347         idx = meta.get("chunk_index", "?"
    )
348         did = meta.get("doc_id", "?")
349
350         txt = h.get("document", "").strip
    ()
351         if len(txt) > 600:
352             txt = txt[:600].rstrip() +
    "…"
353
354         parts.append(f"{txt}\n[cite: doc=
    {did} chunk={idx}]")
355
356     if not parts:
357         return "I couldn't find relevant
    context in the selected documents."
358
359     return "\n\n---\n\n".join(parts)
360
361
```

```
362 # ---------- public API ----------
363
364 def store_document(doc_id: int, file_path
    : str) -> None:
365     """
366     Read file, split text, embed, upsert
    into Chroma.
367     Always index something (even OCR text
     or placeholder) so status can be 'ready
    '.
368     """
369     collection = _ensure_collection()
370
371     full_text = _read_file_text(file_path
    )
372
373     chunks = _split_text(full_text)
374     if not chunks:
375         chunks = [full_text]
376
377     embeddings = _embed_texts(chunks)
378
379     ids = [f"doc:{doc_id}:chunk:{i}" for
    i in range(len(chunks))]
380     metadatas = [
381         {
382             "doc_id": doc_id,
383             "chunk_index": i,
384             "filename": os.path.basename(
    file_path),
385         }
386         for i in range(len(chunks))
387     ]
```

```
388
389     logger.info(f"Upserting {len(chunks)} chunks for doc_id={doc_id}")
390     collection.upsert(
391         ids=ids,
392         documents=chunks,
393         metadatas=metadatas,
394         embeddings=embeddings,
395     )
396
397
398 def query_documents(question: str, document_ids: Optional[List[int]] = None, top_k: int = 4) -> Dict[str, Any]:
399     """
400     Retrieve relevant chunks from Chroma, then answer with OpenAI if available,
401     else fall back to stitched extractive answer.
402     """
403     collection = _ensure_collection()
404
405     where: Dict[str, Any] = {}
406     if document_ids:
407         where = {"doc_id": {"$in": document_ids}}
408
409     q_emb = _embed_texts([question])[0]
410
411     res = collection.query(
412         query_embeddings=[q_emb],
413         n_results=max(1, top_k),
414         where=where or None,
```

```
415         include=["documents", "metadatas"
    , "distances"],
416     )
417
418     documents: List[str] = res.get("
    documents", [[]])[0] if res.get("
    documents") else []
419     metadatas: List[Dict[str, Any]] = res
    .get("metadatas", [[]])[0] if res.get("
    metadatas") else []
420     distances: List[float] = res.get("
    distances", [[]])[0] if res.get("
    distances") else []
421
422     hits = []
423     for doc, meta, dist in zip(documents
    , metadatas, distances):
424         hits.append({"document": doc, "
    metadata": meta, "distance": float(dist
    )})
425
426     contexts = [h["document"] for h in
    hits]
427
428     if OPENAI_API_KEY and
    _OPENAI_AVAILABLE and contexts:
429         try:
430             answer_text =
    _answer_with_openai(question, contexts)
431         except Exception as e:
432             logger.warning("OpenAI call
    failed (%s); falling back.", e)
433             answer_text =
```

```
433 _answer_extractive(hits)
434     else:
435         answer_text = _answer_extractive(
    hits)
436 
437     citations = [
438         {
439             "doc_id": h["metadata"].get("
    doc_id"),
440             "chunk_index": h["metadata"].
    get("chunk_index"),
441             "filename": h["metadata"].get
    ("filename"),
442             "distance": h["distance"],
443         }
444         for h in hits
445     ]
446 
447     return {"answer": answer_text, "
    citations": citations}
448 
```

```python
"""Authentication endpoints.

Provides routes for user registration and login, and a dependency for
retrieving the currently authenticated user from a JWT token.
"""
from datetime import timedelta
from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from sqlalchemy.orm import Session
from .. import models, schemas, auth
from ..database import get_db
from ..core.config import settings

router = APIRouter(prefix="/auth", tags=["auth"])
# OAuth2PasswordBearer expects token in Authorization header as: Bearer <token>
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/auth/login")

@router.post("/register", response_model=schemas.UserRead, status_code=status.HTTP_201_CREATED)
def register(user_in: schemas.UserCreate, db: Session = Depends(get_db)):
    existing = db.query(models.User).filter(models.User.email == user_in.email).first()
```

```
21      if existing:
22          raise HTTPException(status_code=
    status.HTTP_400_BAD_REQUEST, detail="Email
     already registered")
23      hashed_password = auth.
    get_password_hash(user_in.password)
24      user = models.User(email=user_in.email
    , password_hash=hashed_password)
25      db.add(user)
26      db.commit()
27      db.refresh(user)
28      return user
29
30  @router.post("/login", response_model=
    schemas.Token)
31  def login(form_data:
    OAuth2PasswordRequestForm = Depends(), db
    : Session = Depends(get_db)):
32      """Authenticate a user and return a
    JWT token."""
33      user = auth.authenticate_user(db,
    email=form_data.username, password=
    form_data.password)
34      if not user:
35          raise HTTPException(status_code=
    status.HTTP_401_UNAUTHORIZED, detail="
    Invalid email or password")
36      access_token_expires = timedelta(
    minutes=settings.
    ACCESS_TOKEN_EXPIRE_MINUTES)
37      access_token = auth.
    create_access_token(data={"sub": user.
    email}, expires_delta=access_token_expires
```

```
37 )
38     return schemas.Token(access_token=
   access_token, token_type="bearer")
39
40 def get_current_user(db: Session = Depends
   (get_db), token: str = Depends(
   oauth2_scheme)) -> models.User:
41     user = auth.get_user_from_token(db,
   token)
42     if not user:
43         raise HTTPException(status_code=
   status.HTTP_401_UNAUTHORIZED, detail="
   Invalid or expired token")
44     return user
45
```

```python
"""Background job definitions.

In a production deployment you would use Celery, RQ or another task
queue to run these functions asynchronously.  For demonstration
purposes, they are synchronous functions called directly from the API
handlers.
"""
import logging
from sqlalchemy.orm import Session
from . import models, rag

logger = logging.getLogger(__name__)

def process_document_job(db: Session, document: models.Document) -> None:
    """Process a document: extract text, generate embeddings and update status."""
    try:
        rag.process_document(doc_id=document.id, file_path=document.file_path)
        document.status = "ready"
        db.add(document)
        db.commit()
    except Exception as exc:
        logger.error("Processing failed for document %s: %s", document.id, exc)
        document.status = "error"
        db.add(document)
        db.commit()

```

```python
from __future__ import annotations

from datetime import datetime
from typing import List, Optional

from pydantic import BaseModel, EmailStr, Field


# ============================
# Auth / Tokens
# ============================
class Token(BaseModel):
    access_token: str
    token_type: str = "bearer"
    # Optional: seconds until expiry if you choose to return it
    expires_in: Optional[int] = None


# ============================
# Users
# ============================
class UserBase(BaseModel):
    email: EmailStr


class UserCreate(UserBase):
    password: str


class UserRead(UserBase):
    id: int
```

```
32
33      class Config:
34          # Pydantic v2 replacement for orm_mode
35          from_attributes = True
36
37
38  # ============================
39  # Documents
40  # ============================
41  class DocumentOut(BaseModel):
42      id: int
43      filename: str
44      file_size: Optional[int] = None
45      file_type: Optional[str] = None
46      status: str
47      summary: Optional[str] = None
48      upload_date: datetime
49      collection_id: Optional[int] = None
50
51      class Config:
52          from_attributes = True
53
54
55  # ============================
56  # Query (RAG)
57  # ============================
58  class QueryRequest(BaseModel):
59      question: str
60      document_ids: Optional[List[int]] = None
61      # how many chunks to retrieve from the vector store
```

```
62      top_k: int = Field(default=4, ge=1, le
    =20)
63
64
65  class Citation(BaseModel):
66      doc_id: int
67      chunk_index: int
68      filename: Optional[str] = None
69      distance: Optional[float] = None
70
71
72  class QueryResponse(BaseModel):
73      answer: str
74      citations: Optional[List[Citation]] =
    None
75
```

```python
from __future__ import annotations

import logging
from pathlib import Path
from typing import List

from fastapi import APIRouter, Depends, File, HTTPException, UploadFile, status
from sqlalchemy.orm import Session

from .. import models, schemas, auth, rag
from ..database import get_db

logger = logging.getLogger("docuchat.documents")

router = APIRouter(prefix="/documents", tags=["documents"])


@router.get("/", response_model=List[schemas.DocumentOut])
def list_documents(
    db: Session = Depends(get_db),
    current_user: models.User = Depends(auth.get_current_user),
):
    """
    Return all documents for the logged-in user.
    """
    docs = (
        db.query(models.Document)
```

```
28          .filter(models.Document.user_id
    == current_user.id)
29          .order_by(models.Document.
   upload_date.desc())
30          .all()
31      )
32      return docs
33
34
35  @router.post("/upload", response_model=
   schemas.DocumentOut, status_code=status.
   HTTP_201_CREATED)
36  def upload_document(
37      file: UploadFile = File(...),
38      db: Session = Depends(get_db),
39      current_user: models.User = Depends(
   auth.get_current_user),
40  ):
41      """
42      Stream the uploaded file to disk,
   enforce size limit (30MB),
43      insert DB row, then run RAG indexing.
44      """
45      MAX_BYTES = 30 * 1024 * 1024  # 30 MB
46
47      upload_dir = Path(__file__).parent / "
   uploads"
48      upload_dir.mkdir(parents=True,
   exist_ok=True)
49
50      safe_name = Path(file.filename).name
   # strip any path tricks
51      saved_path = upload_dir / safe_name
```

```
52
53      file_size = 0
54      try:
55          with open(saved_path, "wb") as out
    :
56              while True:
57                  chunk = file.file.read(
    1024 * 1024)  # 1MB chunk
58                  if not chunk:
59                      break
60                  file_size += len(chunk)
61                  if file_size > MAX_BYTES:
62                      out.close()
63                      saved_path.unlink(
    missing_ok=True)
64                      raise HTTPException(
65                          status_code=413,
66                          detail=f"File too
    large (> {MAX_BYTES} bytes)",
67                      )
68                  out.write(chunk)
69      finally:
70          # make sure file cursor reset just
     in case (not strictly needed now)
71          try:
72              file.file.seek(0)
73          except Exception:
74              pass
75
76      # Create DB row with status=processing
77      doc = models.Document(
78          user_id=current_user.id,
79          filename=safe_name,
```

```
80          file_path=str(saved_path),
81          file_size=file_size,
82          file_type=file.content_type or "
    application/octet-stream",
83          status="processing",
84      )
85      db.add(doc)
86      db.commit()
87      db.refresh(doc)
88
89      # Now try to process & embed
90      try:
91          rag.store_document(doc.id, str(
    saved_path))
92          doc.status = "ready"
93      except Exception as e:
94          logger.exception("Processing
    failed for document %s: %s", doc.id, e)
95          doc.status = "error"
96
97      db.commit()
98      db.refresh(doc)
99
100     return doc
101
```