Reginald – API Documentation Whitepaper

Root Route

Endpoint: GET `/`

Purpose: Serves the frontend in production or a simple API message in development.

Workflow:
1. If ENVIRONMENT is production and the frontend distribution path exists, attempts to serve index.html.
2. If index.html is missing, returns an error instructing the developer to build the frontend.
3. If not in production, simply returns `{ "message": "Hello from FastAPI!" }`.

Footnotes:
- Depends on global constants ENVIRONMENT and FRONTEND_DIST_PATH.
- Uses FastAPI's FileResponse for serving static frontend files.

Authentication Endpoints

Register
Endpoint: POST `/api/auth/register`
Function: register(user: UserCreate)

Purpose: Register a new user.

Workflow:
1. Receives a UserCreate object from the request body.
2. Calls create_user(user) to persist the new user in the database.
3. On success, returns a User model instance with user details.
4. If validation fails or user already exists, raises HTTP 400.

Footnotes:
- create_user handles persistence and uniqueness checks.
- Response is validated through Pydantic User model.

Login
Endpoint: POST `/api/auth/login`
Function: login(form_data: OAuth2PasswordRequestForm)

Purpose: Authenticates user credentials and returns a JWT access token.

Workflow:
1. Calls authenticate_user(username, password).
2. If authentication fails, raises HTTP 401 with WWW-Authenticate: Bearer.
3. On success, generates a JWT via create_access_token, expiring after a configured interval.
4. Returns `{ "access_token": token, "token_type": "bearer" }`.

Footnotes:
- authenticate_user validates credentials against stored hashes.
- create_access_token encodes JWT with expiry, embedding username in claims.

Current User
Endpoint: GET `/api/auth/me`
Purpose: Retrieves details of the currently authenticated user.

Footnotes:
- Depends on FastAPI Depends(get_current_active_user) which validates JWT and ensures user is active.

Chat Endpoints

Create Chat
Endpoint: POST `/api/chats`
Purpose: Creates a new chat for the current user.

Workflow:
1. Validates ownership using get_current_active_user.
2. Calls create_chat(user_id, title).
3. Fetches updated user chats with get_user_chats and returns the created chat object.

Footnotes:
- Relies on persistence helpers create_chat and get_user_chats.
- Errors trigger HTTP 500 if chat creation fails.

Get Chats
Endpoint: GET `/api/chats`
Purpose: Retrieves all chats for the logged-in user.

Workflow:
1. Fetches chat list via get_user_chats(user_id).
2. Wraps results in Pydantic Chat models.

Footnotes:
- Depends entirely on persistence layer.

Send Message (Non-streaming)
Endpoint: POST `/api/chats/{chat_id}/messages`
Purpose: Sends a message and generates AI response synchronously.

Workflow:
1. Validates chat ownership with get_user_chats.
2. Adds user message via add_message.
3. Fetches full chat history with get_chat_messages.
4. Calls get_chat_response_sync(conversation) for AI inference.
5. Stores AI response with add_message.

Footnotes:
- Tightly integrates storage (for persistence) and AI service call.
- Raises HTTP 500 on persistence or AI service errors.

Send Message (Streaming)
Endpoint: POST `/api/chats/{chat_id}/messages/stream`
Purpose: Streams AI response back to frontend word-by-word.

Workflow:
1. Same validation and storage of user message as above.
2. Collects conversation history for context.
3. Calls generate_chat_response asynchronously to yield chunks.
4. Yields Server-Sent Events (SSE) chunks to frontend.
5. On completion, persists the full AI response in DB.

Footnotes:
- Implements async generator for streaming with FastAPI StreamingResponse.
- Logs streaming progress for debugging and monitoring.

Document Endpoints

Upload Document
Endpoint: POST `/api/documents/upload`
Purpose: Uploads a PDF, validates it, extracts text, and generates embeddings.

Workflow:
1. Validates MIME type (application/pdf) and file size (<10MB).
2. Calls validate_pdf_file(content) for structural validation.
3. Saves file to disk using save_uploaded_file.
4. Extracts text with extract_text_from_pdf.
5. Creates DB record via create_document.
6. Generates embeddings using store_document_embeddings.

Footnotes:
- Cleans up file on error (rollback mechanism).
- Stores metadata (user_id, filename, size) with embeddings.

Get Documents
Endpoint: GET `/api/documents`
Purpose: Retrieves all uploaded documents for current user.

Footnotes:
- Relies on get_user_documents(user_id).
- Wraps results in Pydantic Document model.

Delete Document
Endpoint: DELETE `/api/documents/{document_id}`
Purpose: Deletes both database record and file from disk.

Workflow:
1. Retrieves document list for user and checks ownership.
2. Calls delete_document in DB and delete_file on disk.
3. Attempts to remove embeddings with delete_document_embeddings.

Footnotes:
- Defensive programming with warnings if embedding cleanup fails.