Education AI Platform

Project Documentation

1. Project Overview

This is an AI-powered educational platform that provides intelligent assistance and learning capabilities. The project uses modern web technologies, AI models, and database systems to deliver educational content and interactive features.

2. Project Structure

2.1. Directory Layout

```
EducationAiUpdate/
├── app/                    # Main application code
│   ├── models/             # Database models and schemas
│   ├── views/              # Route handlers and controllers (Consider renaming to
routes/ or controllers/)
│   ├── templates/          # HTML templates (Located at root level in create_app)
│   ├── static/             # Static assets (CSS, JS, images)
│   ├── utils/              # Utility functions
│   ├── services/           # Business logic services
│   ├── routes/             # Blueprint routes (auth, chat, files)
│   ├── __init__.py         # Application factory
│   └── config.py           # Configuration class
├── data/                   # Data storage directory (Consider if still needed with
DB/Uploads)
├── docs/                   # Documentation files
├── instance/              # Instance-specific configuration
├── static/                # Static files (CSS, JS, images) - Note: Redundant?
`app/static` is used.
├── templates/             # HTML templates - Note: Actual templates folder used by Flask
├── tests/                 # Test files
├── uploads/               # User upload directory
├── .dockerignore         # Docker ignore rules
├── .gitignore            # Git ignore rules
├── Dockerfile            # Main Docker configuration
├── Dockerfile.nginx      # Nginx Docker configuration
├── docker-compose.yml    # Docker compose configuration
├── nginx.conf            # Nginx server configuration
├── requirements.txt      # Python dependencies
├── run.py                # Application entry point
└── README.md             # Project readme
```

(Note: Added detail to app/ subdirectories based on later sections)
(Note: Highlighted potential redundancy/clarification needed for `static/` and `templates/` at the root vs. app/)

3. Dependencies (`requirements.txt`)

The project uses the following key dependencies:

3.1. Web Framework

- **Flask==3.1.0:** Web application framework
- **flask-cors==5.0.1:** Cross-Origin Resource Sharing support
- **Flask-Mail==0.10.0:** Email functionality
- **Werkzeug==3.1.3:** WSGI utilities

3.2. AI and Machine Learning

- **langchain==0.3.19:** Framework for developing applications powered by language models
- **langchain-community==0.3.17:** Community-contributed components
- **langchain-core==0.3.35:** Core functionality
- **langchain-groq==0.2.4:** Groq integration
- **faiss-cpu==1.8.0:** Vector similarity search
- **fastembed:** Fast embedding generation *(Note: Version missing, consider adding)*

3.3. Data Processing

- **pandas==2.2.3:** Data manipulation and analysis
- **numpy==1.26.4:** Numerical computing
- **docx2txt==0.8:** DOCX to text conversion
- **pdfminer.six==20231228:** PDF processing
- **python-docx==1.1.2:** DOCX file handling

3.4. Database

- **SQLAlchemy==2.0.38:** SQL toolkit and ORM *(Note: Text mentions SQLAlchemy ORM, but `db.py` uses `sqlite3`. Clarify if SQLite is for dev/testing and SQLAlchemy for production, or if `db.py` needs updating)*

3.5. Authentication and Security

- **PyJWT==2.10.1:** JSON Web Token implementation
- **cryptography==44.0.2:** Cryptographic operations
- **google-auth>=2.22.0:** Google authentication *(Note: Mentioned here, but not explicitly shown in usage examples. Verify integration)*

3.6. Utilities

- **python-dotenv==1.0.1:** Environment variable management
- **loguru==0.7.3:** Logging *(Note: `run.py` uses standard `logging`, `loguru` mentioned here. Standardize or clarify usage)*
- **tqdm==4.67.1:** Progress bars
- **requests==2.32.3:** HTTP requests

4. Configuration Files

4.1. Docker Configuration

- **Dockerfile:** Defines the main application container (Python/Flask app).
- **Dockerfile.nginx:** Defines the Nginx web server container.
- **docker-compose.yml:** Orchestrates the multi-container setup (web, nginx, db).
- **.dockerignore:** Specifies files and directories to exclude from Docker build contexts.

4.2. Nginx Configuration

- **nginx.conf:** Web server configuration including server blocks, SSL settings (if applicable), proxy pass directives to the Flask app, static file serving rules, and security headers.

5. Application Structure

5.1. Main Application (`app/`)

The application follows a modular structure:

- **`app/__init__.py` (Application Factory):** Initializes the Flask application, loads configuration, sets up extensions (like Flask-Mail, database), registers blueprints, and creates necessary directories (like uploads).
- **`app/config.py`:** Defines configuration classes (e.g., `Config`) loading settings from environment variables.
- **`app/models/`:** Contains database model definitions (e.g., User, Content, AIInteraction using SQLAlchemy ORM or direct DB interaction classes as shown later).
- **`app/routes/`:** Contains Flask Blueprints defining application routes/endpoints (e.g., `auth.py`, `chat.py`, `files.py`).
- **`app/services/`:** Encapsulates business logic (e.g., `ChatService`, `FileService`, `PromptService`).
- **`app/templates/`:** Holds HTML templates used for rendering web pages (referenced relative to the project root in `create_app`).
- **`app/static/`:** Stores static assets like CSS, JavaScript, and images (served by Flask in dev, Nginx in prod).
- **`app/utils/`:** Contains helper functions and utility modules (e.g., `db.py`, `decorators.py`, `constants.py`, security helpers, AI processing helpers).

5.2. Database

- Utilizes an ORM (SQLAlchemy mentioned) or direct database access (sqlite3 shown in `db.py`). *(Clarification needed)*
- Supports database backends (PostgreSQL shown in `docker-compose.yml`, SQLite shown in `db.py`).
- Includes schema definitions and potential migration support (though no migration tool is listed in dependencies).

5.3. API Endpoints

The application provides RESTful API endpoints organized by module:

- **Authentication (`/api/auth/` or via `auth` blueprint):**
    - User registration (email verification flow, final registration)
    - Login / Logout
    - Session checking / retrieval
    - Password management (forgot/reset)
    - User profile (`/me`)
- **Content Management (`/api/content/` or via `files` blueprint):**
    - CRUD operations for content/files
    - File uploads

- **AI Interaction (`/api/ai/` or via `chat` blueprint):**
    - Chat message processing
    - Conversation management (create, list, get messages, delete)
    - Prompt management (get, update)
    - History retrieval
    - Feedback submission (Survey)
- **File Operations (likely part of Content or separate `/api/files/`):**
    - Upload, download, delete files.

6. Security Features

- **Authentication:** JWT-based (mentioned) or Session-based (implemented in `auth.py` examples). Password hashing (bcrypt mentioned, standard library `hashlib` or `werkzeug.security` likely used). Token refresh (if JWT).
- **Authorization:** Login required decorator (`@login_required`), potentially role-based access control (mentioned but not detailed), resource ownership validation.
- **Transport Security:** SSL/TLS support (handled by Nginx).
- **Web Security:** CORS protection (`flask-cors`), Input validation, XSS/CSRF protection (Flask built-ins/extensions), secure file uploads (validation, size limits).
- **Configuration Security:** Environment variable management (`python-dotenv`), secret key usage.
- **Session Security:** Secure, HTTPOnly, SameSite session cookies configured in `create_app`.

7. Development Tools

- **Testing:** `pytest` (mentioned, structure in `tests/`).
- **Containerization:** `Docker`, `docker-compose`.
- **Version Control:** `Git`.
- **Web Server / Reverse Proxy:** `Nginx`.

8. Deployment

- **Method:** Docker containers orchestrated by `docker-compose`.
- **Architecture:** Nginx as a reverse proxy forwarding requests to the Flask application container(s). A separate database container (e.g., PostgreSQL).
- **Configuration:** Environment-specific configurations managed via environment variables.
- **Automation:** Potential for automated deployment scripts (not detailed).

9. Testing

9.1. Unit Tests

- Located in the `tests/` directory.
- Focus on individual components: Models, utility functions, service logic, individual route functions (mocking dependencies).
- AI integration tests (mocking external AI APIs).

9.2. Integration Tests

- Test interactions between components: API endpoint functionality, database interactions, file operations, authentication flow.

9.3. Performance Tests

- Load testing endpoints.
- Measuring response times.
- Monitoring resource usage (CPU, memory) under load.
- Assessing scalability.

9.4. Security Tests

- Vulnerability scanning (automated tools).
- Penetration testing (manual or automated).
- Testing authentication and authorization logic rigorously.

10. File Processing

- **Supported Formats:** DOCX (`docx2txt`, `python-docx`), PDF (`pdfminer.six`), Text files, Images (implied, check `ALLOWED_EXTENSIONS`).
- **Operations:** Secure upload, validation (type, size), content extraction, potential processing/indexing for AI context.

11. AI Integration

11.1. Language Model Integration

- **Framework:** `LangChain` used for structuring interactions.
- **Provider:** `Groq` via `langchain-groq` (requires `GROQ_API_KEY`). Model configurable via `MODEL_NAME`.

- **Core Logic:** Implemented in `ChatService` and `ChatModel`, handling prompt formatting, API calls, and response parsing.
- **Prompt Engineering:** Managed by `PromptService`, allowing for dynamic system prompts based on user settings or defaults (like `Ms. Potter's Teaching Framework`).

11.2. Vector Store

- **Library:** `faiss-cpu` for similarity search.
- **Embeddings:** `fastembed` for generating vector embeddings from text content.
- **Implementation:** `VectorStoreModel` provides an interface to create/update the vector store from documents and search for similar context based on user queries. Uses a shared persistent file (`shared_vector_store.pkl`).
- **Use Case:** Retrieving relevant document snippets to provide context to the LLM during chat interactions (`get_document_context` in `ChatService`).

11.3. Text Processing

- **Document Parsing:** Handled by `FileService` using libraries like `docx2txt`, `pdfminer.six`.
- **Text Extraction:** Getting plain text from various file formats.
- **Potential Features:** Content summarization, question answering (leveraging the LLM and vector store).

11.4. Content Generation

- Leveraging the integrated LLM (`Groq`) for:
    - Educational explanations.
    - Answering student questions.
    - Generating examples or quiz questions (potential).

12. Monitoring and Logging

- **Logging:** Configured in `run.py` using standard `logging`. A structured logging system (potentially `loguru` or custom configuration) is mentioned, aiming for different levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) and handlers (console, file, email). Service-level logging implemented via `ServiceLogger`. Audit logging via `AuditLogger`.
- **Error Tracking:** Comprehensive error handling with specific HTTP error handlers and custom exceptions. Errors are logged with details.
- **Performance Monitoring:** Mentioned as a goal, potentially integrated with logging or external tools. Nginx access/error logs provide basic request monitoring.
- **User Activity Logging:** Implied through database records (logins, interactions, surveys). Audit logging can enhance this.

13. Best Practices

- **Code Quality:** Adherence to PEP 8 standards.
- **Architecture:** Modular design (Blueprints, Services, Models).
- **Security:** Secure coding practices applied throughout (input validation, ORM/parameterized queries, password hashing, etc.).
- **Documentation:** Commitment to documentation standards (this document, code comments).
- **Version Control:** Use of Git for source code management.

14. Future Improvements

- Enhanced AI capabilities (more models, fine-tuning).
- Support for additional file formats.
- Performance optimizations (caching, query tuning, asynchronous tasks).
- Extended API features.
- Improved and more comprehensive documentation.

15. Support and Maintenance

- Plan for regular updates, bug fixes, security patches.
- Ongoing performance improvements and feature additions.

16. Core Application Files

16.1. `run.py` (Application Entry Point)

```python
from app import create_app
import logging
import sys
import os


# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
# Note: Consider using Flask's built-in logger or integrating Loguru if preferred.
logger = logging.getLogger(__name__) # Added logger instance


app = create_app()
```

```
if __name__ == '__main__':
    # Get port from environment variable or default to 5000
    # Command line arg is less common for containerized apps
    port = int(os.environ.get('PORT', 5000))

    # Enable debug mode based on FLASK_DEBUG env var
    debug = os.environ.get('FLASK_DEBUG', '0') == '1'

    logger.info(f"Starting application on port {port} with debug={debug}")
    # Use waitress or gunicorn for production instead of Flask dev server
    app.run(debug=debug, host='0.0.0.0', port=port, use_reloader=debug)
```

Key Features:

- Uses the application factory pattern (`create_app`).
- Configures basic logging using the standard `logging` module.
- Reads port from PORT environment variable (default 5000).
- Enables Flask debug mode and reloader based on FLASK_DEBUG environment variable.
- Binds to `0.0.0.0` to be accessible within the container network.
- *(Note: For production, the development server (`app.run`) should be replaced with a production-grade WSGI server like Gunicorn or Waitress, typically managed via the CMD in the `Dockerfile`)*

16.2. Docker Configuration Files

16.2.1. `Dockerfile` (Main Application)

```
# Use a specific Python 3.9 slim image version for reproducibility
FROM python:3.9.18-slim

WORKDIR /app

# Copy only requirements first to leverage Docker cache
COPY requirements.txt .
# Consider using virtual environments for better isolation
# RUN python -m venv /venv && . /venv/bin/activate
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application code
COPY . .
```

```
# Set environment variables (can also be set in docker-compose.yml)
# ENV FLASK_APP=run.py
# ENV FLASK_RUN_HOST=0.0.0.0
# ENV PORT=5000


EXPOSE 5000


# Command to run the application (Use a production server)
# CMD ["gunicorn", "-b", "0.0.0.0:5000", "run:app"]
CMD ["python", "run.py"] # Ok for dev, replace for prod
```

Features:

- Based on `python:3.9-slim`.
- Sets working directory to /app.
- Installs Python dependencies from `requirements.txt`.
- Copies application code into the image.
- Exposes port 5000.
- Specifies the default command to run the application (`python run.py`).

16.2.2. `docker-compose.yml` (Orchestration)

```
version: '3.8'


services:
  web:
    build: . # Uses the Dockerfile in the current directory
    ports:
      # Maps host port 5000 to container port 5000 (direct access, usually only Nginx
port is exposed)
      - "5000:5000" # Consider removing if only accessing via Nginx
    volumes:
      # Mounts current directory to /app for development hot-reloading
      - .:/app
    environment:
      # Environment variables passed to the Flask container
      - FLASK_DEBUG=1 # Enable debug mode for development
      - DATABASE_URL=postgresql://app_user:app_password@db:5432/app_db
      - SECRET_KEY=${SECRET_KEY} # Pass from host env or .env file
      - JWT_SECRET_KEY=${JWT_SECRET_KEY}
      - MAIL_SERVER=${MAIL_SERVER}
      - MAIL_PORT=${MAIL_PORT}
      - MAIL_USE_TLS=${MAIL_USE_TLS}
      - MAIL_USERNAME=${MAIL_USERNAME}
      - MAIL_PASSWORD=${MAIL_PASSWORD}
      - GROQ_API_KEY=${GROQ_API_KEY}
      - MODEL_NAME=${MODEL_NAME}
      # Add other necessary env vars
    depends_on:
      - db # Ensures 'db' service starts before 'web'


  nginx:
    build:
      context: .
      dockerfile: Dockerfile.nginx
    ports:
      # Maps host port 80 to container port 80
      - "80:80"
      # Maps host port 443 to container port 443 (for SSL)
      - "443:443"
    volumes:
      # Consider mounting SSL certificates here if needed
      # - ./certs:/etc/nginx/certs
      - ./static:/app/static:ro # Mount static files for Nginx serving (optional if
served by Flask/copied in Dockerfile)
    depends_on:
      - web # Ensures 'web' service is available for proxying


  db:
    image: postgres:13 # Uses official PostgreSQL 13 image
    environment:
      # Database credentials and name
      - POSTGRES_USER=app_user
      - POSTGRES_PASSWORD=app_password # Use secrets for production
      - POSTGRES_DB=app_db
    volumes:
      # Persists database data on the host machine
      - postgres_data:/var/lib/postgresql/data


volumes:
  # Defines the named volume for data persistence
  postgres_data:
```

Features:

- Defines three services: web (Flask app), `nginx` (reverse proxy), db (PostgreSQL).
- Builds web and `nginx` services from local Dockerfiles.
- Maps ports for external access (80/443 via Nginx, 5000 direct to Flask - optional).
- Uses volumes for code mounting (development) and data persistence (`postgres_data`).
- Configures environment variables for the web service, linking to the db service by hostname (db).
- Defines service dependencies (depends_on).

16.3. Nginx Configuration (`nginx.conf`)

(Format the following as a code block)

```
user nginx;
worker_processes auto; # Adjust based on server cores
error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;

events {
    worker_connections 1024; # Adjust based on expected concurrency
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" "$http_x_forwarded_for"';

    access_log /var/log/nginx/access.log main;

    sendfile on;
    tcp_nopush on; # Optimize sending data packets
    tcp_nodelay on; # Reduce latency for small packets
    keepalive_timeout 65;
    types_hash_max_size 2048;

    # Gzip compression (optional but recommended)
    # gzip on;
    # gzip_vary on;
    # gzip_proxied any;
    # gzip_comp_level 6;
    # gzip_types text/plain text/css application/json application/javascript text/xml
application/xml application/xml+rss text/javascript;

    server {
        listen 80;
        server_name localhost; # Replace with your domain name

        # Redirect HTTP to HTTPS (uncomment if using SSL)
        # location / {
        #     return 301 https://$host$request_uri;
        # }

        # For SSL (uncomment and configure)
        # listen 443 ssl;
        # server_name your_domain.com;
        # ssl_certificate /etc/nginx/certs/your_cert.pem;
        # ssl_certificate_key /etc/nginx/certs/your_key.pem;
        # Include snippets/ssl-params.conf; # For strong SSL settings

        location / {
            proxy_pass http://web:5000; # Forward requests to the 'web' service container
on port 5000
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme; # Important for Flask to know if
it's http/https
            # Add settings for large uploads if needed
            # client_max_body_size 100M;
        }

        # Serve static files directly via Nginx for better performance
        location /static/ {
            alias /app/static/; # Path inside the Nginx container (adjust if volume mount
differs)
            expires 30d; # Add cache control headers
            add_header Cache-Control "public";
        }

        # Optional: Add location block for /uploads/ if needed
        # location /uploads/ {
        #     alias /app/uploads/;
        #     # Add appropriate access control if needed
        # }
    }
}
```

Features:

- Configures Nginx worker processes and connections.
- Sets up logging formats and paths for access and error logs.
- Includes MIME types and default type.
- Optimizes file sending (`sendfile`, `tcp_nopush`).
- Defines an HTTP server block listening on port 80.
- Includes commented-out sections for HTTPS redirection and SSL configuration.
- Proxies requests to the web service (Flask app) running on `web:5000`.
- Sets necessary proxy headers (`Host`, `X-Real-IP`, `X-Forwarded-For`, `X-Forwarded-Proto`).
- Configures efficient serving of static files from `/static/` URL path, mapping to the `/app/static/` directory within the Nginx container (adjust alias path if needed).
- Sets cache expiration headers for static assets.

17. Application Implementation Details

17.1. Application Factory (`app/__init__.py`)

(Format the following as a code block)

```
from flask import Flask
from datetime import timedelta
import os
# Assuming db utilities are in app.utils.db
from app.utils.db import init_db, close_db # Add close_db if using Flask < 2.3 pattern
from flask_mail import Mail
# Assuming Config class is in app.config
from app.config import Config
# Import blueprints
from app.routes.auth import bp as auth_bp
from app.routes.chat import bp as chat_bp
from app.routes.files import bp as files_bp # Assuming files routes exist
import logging # Use standard logging

mail = Mail()
logger = logging.getLogger(__name__) # Get logger for factory

def create_app(config_class=Config): # Allow passing config class
    # Create Flask app instance
    app = Flask(__name__, instance_relative_config=True) # instance_relative_config=True
is often useful

    # Load configuration from config object
    app.config.from_object(config_class)

    # Load instance config if it exists (e.g., instance/config.py)
    # app.config.from_pyfile('config.py', silent=True)

    # Configure session settings
    app.config.setdefault('PERMANENT_SESSION_LIFETIME', timedelta(hours=1))
    app.config.setdefault('SESSION_COOKIE_SECURE', not app.debug) # Secure only if not in
debug mode
    app.config.setdefault('SESSION_COOKIE_HTTPONLY', True)
    app.config.setdefault('SESSION_COOKIE_SAMESITE', 'Lax')

    # --- Initialization of Extensions ---
    # Initialize Flask-Mail
    mail.init_app(app)

    # Initialize Database
    # Using direct sqlite3 from db.py requires app_context management
    # If using Flask-SQLAlchemy, it would be: db.init_app(app)
    try:
        with app.app_context():
            init_db(app) # Pass app if config is needed inside init_db
    except Exception as e:
        logger.error(f"Failed to initialize database: {e}", exc_info=True)
        # Decide if the app should fail to start or continue without DB

    # Register teardown function to close DB connection (Flask < 2.3 pattern for g)
    # Newer Flask handles this better with app_context
    @app.teardown_appcontext
    def teardown_db(exception=None):
        close_db() # Assuming close_db function exists in utils.db


    # --- Register Blueprints ---
    app.register_blueprint(auth_bp, url_prefix='/auth') # Example prefix
    app.register_blueprint(chat_bp, url_prefix='/chat') # Example prefix
    # app.register_blueprint(files_bp, url_prefix='/files') # Example prefix
    # Consider adding a main/index blueprint for routes like '/'


    # --- Create Upload Folder ---
    upload_folder = app.config.get('UPLOAD_FOLDER', os.path.join(app.root_path, '..',
'uploads'))
    os.makedirs(upload_folder, exist_ok=True)
    app.config['UPLOAD_FOLDER'] = upload_folder # Ensure it's set correctly


    logger.info("Flask application created successfully.")
    return app
```

Key Features:

- Implements the Application Factory pattern (`create_app`).
- Loads configuration from `Config` object and optionally `instance/config.py`.
- Configures secure session cookie settings.
- Initializes extensions like Flask-Mail.
- Initializes the database using `utils.db.init_db` within an application context.
- Registers Blueprints (`auth`, `chat`, `files`) with optional URL prefixes.
- Creates the upload folder if it doesn't exist.
- Includes teardown function to close DB connection (important for g-based connections).

17.2. Application Structure (Detailed)

17.2.1. Routes (`app/routes/`)

- **`auth.py`:** Handles user authentication (registration, email verification, login, logout, password reset, session checks).
- **`chat.py`:** Manages chat interactions (main interface, message processing, AI calls, conversation history, prompt management, survey submission).
- **`files.py` (or similar):** Handles file operations (upload, download, processing, potentially content listing/management).

17.2.2. Services (`app/services/`)

- **`ChatService`:** Orchestrates chat logic, interacts with `ChatModel`, `ConversationModel`, `VectorStoreModel`, formats history, manages context.
- **`PromptService`:** Manages loading, retrieving, and updating system prompts for users.
- **`FileService`:** Handles file uploads, validation, storage, content extraction, and potentially interacts with `VectorStoreModel` for indexing.
- **(Implied) `UserService`:** Could encapsulate logic related to user creation, profile updates, API key management (currently spread between `auth.py` routes and `UserModel`).
- **(Implied) `SurveyService`:** Could encapsulate logic for saving and retrieving survey responses (currently in `chat.py` routes and `SurveyModel`).

17.2.3. Models (`app/models/`)

(These appear to be custom classes interacting directly with the DB via `db.py` utilities, not SQLAlchemy models as initially suggested. Format the following Python snippets as code blocks)

- **`UserModel`:** Manages user data (create, get by email, update API key).

```python
# Example Structure (adapt from doc)
class UserModel:
    def __init__(self, user_id: Optional[int] = None):
        self.user_id = user_id
        # Potentially load user data if user_id provided


    @staticmethod
    def create_user(...) -> int: ...
    @staticmethod
    def get_user_by_email(...) -> Dict[str, Any]: ...
    def update_api_key(self, ...) -> bool: ...
    # Other user-related db operations
```

- **`ChatModel`:** Interfaces with the Groq LLM API.

```python
# Example Structure (adapt from doc)
class ChatModel:
    def __init__(self, api_key: str, user_id: Optional[int] = None):
        self.api_key = api_key
        self.user_id = user_id # May not be needed here
        # Initialize Groq client


    def generate_response(...) -> str: ...
    # Logic to call Groq API with prompt, history, etc.
```

- `VectorStoreModel`**:** Manages the FAISS vector store (singleton pattern).

```
# Example Structure (adapt from doc)
class VectorStoreModel:
    _instance = None
    VECTOR_STORE_PATH = "shared_vector_store.pkl"
    # ... (Singleton __new__)


    def __init__(self): # Initialization logic (loading/creating store)
        if not hasattr(self, 'initialized'): # Avoid re-init
            # Load FAISS index, embeddings, etc.
            self.initialized = True


    def create_vectorstore(self, documents: List) -> None: ...
    def search_similar(self, query: str, k: int = 3) -> List: ...
    # Add methods for saving/loading the store
```

- `ConversationModel`**:** Manages conversation metadata and messages in the database.

```
# Example Structure (adapt from doc)
class ConversationModel:
    def __init__(self, user_id: int):
        self.user_id = user_id


    def create_conversation(self, title: str) -> int: ...
    def get_conversations(self, limit: int = 4) -> List[Dict]: ...
    def save_message(self, conversation_id: int, message: str, role: str) -> int: ...
    def get_messages(self, conversation_id: int) -> List[Dict]: ... # Added based on
chat.py
    def delete_conversation(self, conversation_id: int) -> None: ... # Added based on
chat.py
    # Other conversation/message db operations
```

- `SurveyModel`**:** Manages survey responses in the database.

```
# Example Structure (adapt from doc)
class SurveyModel:
    def __init__(self, user_id: int):
        self.user_id = user_id


    def save_survey_response(self, rating: int, message: Optional[str] = None) ->
int: ...
    def get_user_survey_responses(self) -> List[Dict]: ...
    def check_survey_status(self) -> bool: ... # Added based on chat.py
```

17.2.4. Utils (`app/utils/`)

- **`db.py`:** Contains database connection logic (`get_db`, `close_db`), initialization (`init_db`), and potentially raw SQL execution helpers or basic CRUD functions.
- **`constants.py`:** Defines application-wide constants (e.g., `MAX_CONVERSATIONS`, `DEFAULT_PROMPT`, `ALLOWED_EXTENSIONS`).
- **`decorators.py`:** Contains custom decorators, notably `@login_required` for authentication checks.
- **(Implied) `security.py`:** Could hold password hashing functions, token generation/verification logic (if using JWT).
- **(Implied) `ai_helpers.py`:** Could contain utility functions for interacting with LangChain, formatting prompts/history, or processing AI responses.
- **(Implied) `file_helpers.py`:** Could contain functions for validating filenames, extracting text, etc.

17.3. Configuration (`app/config.py`)

```
import os
from dotenv import load_dotenv

# Load environment variables from .env file if it exists
basedir = os.path.abspath(os.path.dirname(__file__))
load_dotenv(os.path.join(basedir, '..', '.env')) # Load .env from project root

class Config:
    # Secret Keys
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'you-should-set-a-real-secret-key'
    JWT_SECRET_KEY = os.environ.get('JWT_SECRET_KEY') or 'another-secret-key-for-jwt' #
If using JWT

    # Database Configuration
    # Option 1: SQLite (matches db.py)
    DATABASE = os.environ.get('DATABASE_URL') or \
        os.path.join(basedir, '..', 'instance', 'app.db') # Path to SQLite file
    # Option 2: SQLAlchemy style (if using SQLAlchemy)
    # SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
    #     'sqlite:///' + os.path.join(basedir, '..', 'instance', 'app.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False # Generally set to False

    # File Upload Configuration
    UPLOAD_FOLDER = os.environ.get('UPLOAD_FOLDER') or \
        os.path.join(basedir, '..', 'uploads')
    MAX_CONTENT_LENGTH = int(os.environ.get('MAX_CONTENT_LENGTH') or 16 * 1024 * 1024) #
16MB default
    ALLOWED_EXTENSIONS = {'txt', 'pdf', 'doc', 'docx'} # Keep in sync with constants.py?

    # Email Configuration (using Flask-Mail)
    MAIL_SERVER = os.environ.get('MAIL_SERVER')
    MAIL_PORT = int(os.environ.get('MAIL_PORT', 587))
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS', 'true').lower() in ['true', '1', 't']
    MAIL_USE_SSL = os.environ.get('MAIL_USE_SSL', 'false').lower() in ['true', '1', 't']
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD') # Use App Password for Gmail
    MAIL_DEFAULT_SENDER = os.environ.get('MAIL_DEFAULT_SENDER') or MAIL_USERNAME

    # AI Configuration
    GROQ_API_KEY = os.environ.get('GROQ_API_KEY')
    MODEL_NAME = os.environ.get('MODEL_NAME', 'llama3-8b-8192') # Example Groq model

    # Add other configurations as needed
    # e.g., Vector Store Path
    VECTOR_STORE_PATH = os.environ.get('VECTOR_STORE_PATH') or \
        os.path.join(basedir, '..', 'instance', 'shared_vector_store.pkl')

# You might have different config classes for development, testing, production
# class DevelopmentConfig(Config):
#     DEBUG = True
#     SQLALCHEMY_DATABASE_URI = os.environ.get('DEV_DATABASE_URL') or ...

# class TestingConfig(Config):
#     TESTING = True
#     SQLALCHEMY_DATABASE_URI = os.environ.get('TEST_DATABASE_URL') or
'sqlite:///:memory:'

# class ProductionConfig(Config):
#     DEBUG = False
#     SESSION_COOKIE_SECURE = True
#     # etc.
```

17.4. Environment Variables

Required environment variables (typically set in a `.env` file for local development or system environment variables in production):

(Format as a code block or list)

```
# Flask Core & Security
SECRET_KEY=your-very-strong-random-secret-key
FLASK_DEBUG=1 # 1 for development, 0 for production

# Database (Example for PostgreSQL in docker-compose)
DATABASE_URL=postgresql://app_user:app_password@db:5432/app_db
# Or for SQLite (if using file path in Config)
# DATABASE_URL=instance/app.db

# Email (Flask-Mail)
MAIL_SERVER=smtp.example.com
MAIL_PORT=587
MAIL_USE_TLS=true
MAIL_USERNAME=your-email@example.com
MAIL_PASSWORD=your-email-app-password-or-key
MAIL_DEFAULT_SENDER=your-email@example.com

# AI (Groq)
GROQ_API_KEY=your-groq-api-key-here
MODEL_NAME=llama3-8b-8192 # Or your preferred Groq model

# Optional / Other
# JWT_SECRET_KEY=your-jwt-secret-if-using-jwt
# UPLOAD_FOLDER=/path/to/uploads # Optional, defaults set in Config
# MAX_CONTENT_LENGTH=16777216 # Optional, defaults set in Config (bytes)
# VECTOR_STORE_PATH=instance/shared_vector_store.pkl # Optional, defaults set in Config
PORT=5000 # Port the Flask app runs on inside the container
```

17.5. Error Handling

- **HTTP Errors:** Standard Flask error handlers (e.g., `@app.errorhandler(404)`) for common codes (400, 401, 403, 404, 500), returning JSON responses for API routes and HTML pages for web routes.

- **Custom Exceptions:** Defined specific exception classes (e.g., `AuthenticationError`, `ValidationError`, `FileError`, `AIError`, `ServiceError`) possibly inheriting from a base `AppError` class, allowing for more granular error catching and reporting.
- **Error Logging:** All significant errors are logged with stack traces, context (like user ID, request data), using the configured logging system.
- **User Feedback:** Clear error messages returned to the user (API responses or flashed messages on web pages), avoiding exposure of sensitive internal details.

17.6. Logging System

- **Framework:** Standard `logging` (or potentially `loguru`).
- **Levels:** Utilizes standard levels (DEBUG, INFO, WARNING, ERROR, CRITICAL).
- **Format:** Configured format including timestamp, level, logger name (module), and message. Structured logging (key-value pairs) might be used for easier parsing.
- **Handlers:** Configured handlers direct logs to appropriate outputs:
    - Console Handler (useful for development and container logs).
    - File Handler (for persistent logs, possibly rotating).
    - Email Handler (e.g., `SMTPHandler` for critical errors).
- **Context:** Logs include relevant context (user ID, request ID, etc.) where possible.

17.7. Performance Optimization

- **Database:**
    - Connection management (using g or Flask-SQLAlchemy's session scoping).
    - Query optimization (efficient SQL, use of ORM features if applicable).
    - Indexing critical table columns (e.g., foreign keys, frequently queried fields - see `db.py` schema).
    - Potential database-level caching (less common with SQLite/Postgres unless using specific tools).
- **File Handling:**
    - Chunked uploads (if handling very large files, possibly via JS library).
    - Background processing for time-consuming tasks like content extraction/indexing (using Celery, RQ, or Flask background tasks - *Note: No dependency listed*).
    - Compression (e.g., Gzip via Nginx).
    - Client-side caching for static assets (via Nginx headers).
- **AI Processing:**
    - Response caching (e.g., caching LLM responses for identical prompts/context for a short period).
    - Context optimization (retrieving only the most relevant context from the vector store).
    - Rate limiting API calls (if needed, externally or within the app).

- **Web Server / Application:**
    - Efficient static file serving via Nginx.
    - Response compression (Gzip via Nginx).
    - Caching headers set by Nginx/Flask.
    - Using a production-grade WSGI server (Gunicorn, Waitress).
    - Potential for load balancing across multiple application instances (via Nginx Plus, HAProxy, or cloud load balancer).

18. API Endpoints Documentation

18.1. Authentication Module (`auth.py`)

- `POST /auth/register_email`
    - **Description:** Handles initial email submission for registration. Sends a verification email with a token.
    - **Request Body:** `{ "email": "user@example.com" }`
    - **Response:** Success message or error (e.g., email already exists, invalid format).
    - **Features:** Email validation, duplicate check, secure token generation (e.g., `itsdangerous`), email sending via Flask-Mail. Token typically has an expiration time (e.g., 24 hours).
- `GET /auth/verify_email/<token>`
    - **Description:** Verifies the email address using the token from the email link. Redirects to the main registration form upon success.
    - **URL Parameter:** `token` (the verification token).
    - **Response:** Redirect to `/auth/register` (passing verified email) or error page/message (invalid/expired token).
    - **Security:** Token validation (signature, expiration).
- `GET /auth/register`
    - **Description:** Renders the final user registration form (expects verified email, possibly passed via session or query param after verification).
    - **Response:** HTML registration page.
- `POST /auth/register`
    - **Description:** Creates the new user account after email verification. Logs the user in.
    - **Request Body:** `{ "username": "...", "password": "...", "class_standard": "...", "medium": "...", "groq_api_key": "...", "verified_email": "..." }`
    - **Response:** Redirect to main app page (e.g., chat) or JSON success/error message.
    - **Processing:** Username/password validation, password hashing, store user data in DB, initialize user session.
- `GET /auth/login`

    - **Description:** Renders the login form.
    - **Response:** HTML login page.
- `POST /auth/login`
    - **Description:** Authenticates the user using email and password.
    - **Request Body:** `{ "email": "...", "password": "..." }`
    - **Response:** Redirect to main app page or JSON success/error message.
    - **Security:** Password verification against hash, session creation/management.
- `GET /auth/logout`
    - **Description:** Logs the user out by clearing the session.
    - **Response:** Redirect to login page.
- `GET /auth/check_session`
    - **Description:** API endpoint to verify if the current user has an active session.
    - **Response:** `{ "authenticated": true/false }`
- `GET /auth/session`
    - **Description:** API endpoint to retrieve basic data about the logged-in user from the session.
    - **Response:** `{ "user_id": ..., "username": ... }` or error if not logged in.
- `GET /auth/forgot_password`
    - **Description:** Renders the forgot password form (email input).
    - **Response:** HTML forgot password page.
- `POST /auth/forgot_password`
    - **Description:** Initiates the password reset process. Sends a password reset link/token via email.
    - **Request Body:** `{ "email": "..." }`
    - **Response:** Success message (even if email doesn't exist, to prevent enumeration).
    - **Security:** Token generation, email sending.
- `GET /auth/reset_password/<token>`
    - **Description:** Renders the password reset form, validating the token first.
    - **URL Parameter:** `token` (the reset token).
    - **Response:** HTML password reset page or error page.
- `POST /auth/reset_password/<token>`
    - **Description:** Sets the new password for the user associated with the valid token.
    - **Request Body:** `{ "password": "...", "confirm_password": "..." }`
    - **Response:** Redirect to login page with success message or error message.
    - **Security:** Token validation, password validation/hashing, update user record.

18.2. Chat Module (`chat.py`)

(Assuming routes are under `/chat` blueprint prefix and require login)

- `GET /chat/health`
    - **Description:** Simple health check endpoint for monitoring (e.g., by Docker).
    - **Response:** `{ "status": "healthy" }`, HTTP 200.
- `GET /chat/` **(or `/chat/index`)**
    - **Description:** Renders the main chat interface page.
    - **Response:** HTML chat page (`chat.html`). Requires `@login_required`.
- `GET /chat/get_prompt`
    - **Description:** Retrieves the current system prompt for the logged-in user.
    - **Response:** `{ "prompt": "current system prompt text..." }`
    - **Logic:** Uses `PromptService`.
- `POST /chat/update_prompt`
    - **Description:** Updates the system prompt for the logged-in user.
    - **Request Body:** `{ "prompt": "new system prompt text..." }`
    - **Response:** `{ "status": "success" }` or error.
    - **Logic:** Uses `PromptService`, validates input.
- `POST /chat/chat`
    - **Description:** Processes a user's chat message, gets a response from the AI, and saves the interaction.
    - **Request Body:** `{ "message": "User's message", "conversation_id": (optional) id }`
    - **Response:** `{ "response": "AI's response text...", "conversation_id": id }` or error.
    - **Logic:** Uses `ChatService.process_message`, handles context, AI calls, saving via `ConversationModel`. Creates a new conversation if `conversation_id` is null/missing.
- `POST /chat/create_conversation`
    - **Description:** Explicitly creates a new, empty conversation for the user.
    - **Request Body:** `{ "title": "Optional conversation title" }`
    - **Response:** `{ "conversation_id": new_id, "title": "..." }` or error.
    - **Logic:** Uses `ConversationModel.create_conversation`.
- `GET /chat/get_conversations`
    - **Description:** Retrieves a list of the user's recent conversations.
    - **Response:** `[ { "id": ..., "title": "...", "last_updated": "..." }, ... ]`
    - **Logic:** Uses `ConversationModel.get_conversations`.
- `GET /chat/get_messages/<int:conversation_id>`
    - **Description:** Retrieves the message history for a specific conversation.
    - **URL Parameter:** `conversation_id`.
    - **Response:** `[ { "role": "user/bot", "content": "...", "timestamp": "..." }, ... ]`
    - **Logic:** Uses `ConversationModel.get_messages`. Access control check needed.

- `DELETE /chat/delete_conversation/<int:conversation_id>`
    - **Description:** Deletes a specific conversation and its messages.
    - **URL Parameter:** `conversation_id`.
    - **Response:** `{ "status": "success" }` or error.
    - **Logic:** Uses `ConversationModel.delete_conversation`. Access control check needed.
- `DELETE /chat/delete_all_conversations`
    - **Description:** Deletes all conversations for the logged-in user.
    - **Response:** `{ "status": "success" }` or error.
    - **Logic:** Needs implementation in `ConversationModel` or service layer. Use with caution.
- `GET /chat/download_chat/<int:conversation_id>`
    - **Description:** Exports a specific conversation history as a downloadable file (e.g., TXT, JSON).
    - **URL Parameter:** `conversation_id`.
    - **Response:** File download response.
    - **Logic:** Retrieves messages, formats them, creates file response. Access control check needed.
- `POST /chat/submit_survey`
    - **Description:** Submits the user's feedback/rating survey response.
    - **Request Body:** `{ "rating": number (1-10), "message": "Optional feedback text" }`
    - **Response:** `{ "status": "success" }` or error.
    - **Logic:** Uses `SurveyModel.save_survey_response`.
- `GET /chat/check_survey_status`
    - **Description:** Checks if the user has already submitted the survey.
    - **Response:** `{ "submitted": true/false }`
    - **Logic:** Uses `SurveyModel.check_survey_status`.
- `GET /chat/test_survey_db` *(Likely for debugging)*
    - **Description:** Endpoint to test survey database connectivity/functionality.
    - **Response:** Success/error message.

19. API Best Practices

19.1. Error Handling

- **Consistent Format:** Use a standard JSON structure for API error responses:

```
{
    "status": "error",
    "error": {
        "code": "UNIQUE_ERROR_CODE", // e.g., "INVALID_INPUT", "AUTH_REQUIRED"
        "message": "User-friendly error description.",
        "details": { /* Optional: More specific details, field errors */ }
    }
}
```

- **Appropriate HTTP Status Codes:** Use correct codes (400 for bad request, 401 for unauthorized, 403 for forbidden, 404 for not found, 500 for server error, etc.).
- **Logging:** Log all errors, especially 5xx server errors, with sufficient detail for debugging.

19.2. Performance Optimization

- **Caching:** Implement caching where appropriate (e.g., frequently accessed read-only data, potentially AI responses) using Flask-Caching or similar. Use ETags or Last-Modified headers for client-side caching.
- **Database:** Optimize queries, use indexes, avoid N+1 problems. Use connection pooling effectively.
- **Payload Size:** Keep request/response payloads reasonably sized. Use pagination for lists (`/get_conversations`, `/get_messages`). Consider partial responses if applicable.
- **Asynchronous Operations:** Use background task queues (Celery, RQ) for long-running operations (complex file processing, bulk emails) to avoid blocking web requests. *(Note: Requires adding dependencies)*

19.3. Documentation

- **API Specification:** Maintain an API specification using tools like OpenAPI (Swagger). This can be generated from code comments (e.g., with `apispec`) or written manually.
- **Code Documentation:** Write clear docstrings for functions, classes, and modules explaining their purpose, parameters, and return values. Use type hints.

20. Services Implementation

(Details based on `## Services Implementation`)

20.1. Chat Service (`chat_service.py`)

- **Initialization:** Takes `user_id` and `api_key`. Initializes instances of `ChatModel`, `ConversationModel`, `VectorStoreModel`.
- **`process_message`:** Core method.
    1. Retrieves/creates conversation context (`ConversationModel`).
    2. Formats chat history (`format_chat_history`).

3. Retrieves relevant document context using `VectorStoreModel.search_similar` (`get_document_context`).
4. Combines system prompt, document context, history, and new message.
5. Calls `ChatModel.generate_response` to get AI output.
6. Saves user message and AI response to history using `ConversationModel.save_message`.
7. Returns AI response and conversation ID. Handles errors throughout.

- **`format_chat_history`:** Converts database message history into the format expected by the LLM (e.g., list of dicts with 'role' and 'content').
- **`get_document_context`:** Takes user message, queries `VectorStoreModel`, formats retrieved chunks into a context string.
- **Conversation Methods:** Wraps `ConversationModel` methods (`create_conversation`, `get_conversation_messages`, `delete_conversation`) adding service-level logic or validation if needed.

20.2. Prompt Service (`prompt_service.py`)

- **Initialization:** Takes `user_id`. Loads the user's specific prompt from the database (`user_prompts` table via a hypothetical `PromptModel` or direct DB query) or falls back to `constants.DEFAULT_PROMPT`.
- **`get_prompt`:** Returns the loaded prompt string.
- **`update_prompt`:** Validates the `new_prompt` string (length, content constraints?) and saves it to the database for the user, updating the service instance's prompt.

20.3. File Service (`file_service.py`)

- **Initialization:** Takes `user_id`. Determines user-specific storage path (or uses the global `UPLOAD_FOLDER`).
- **`upload_file`:**
    1. Validates the `FileStorage` object (allowed extension via `constants.ALLOWED_EXTENSIONS`, size via `Config.MAX_CONTENT_LENGTH`).
    2. Generates a secure filename (e.g., using `werkzeug.utils.secure_filename` plus a unique identifier).
    3. Saves the file to the configured `UPLOAD_FOLDER`.
    4. (Crucially) Records file metadata (original name, stored path, user ID, timestamp, content type) in a database table (e.g., a `files` or `documents` table - *Note: This table seems missing from the schema examples*).
    5. Optionally triggers background processing (`process_file`).
    6. Returns file metadata (ID, name).
- **`download_file`:**
    1. Retrieves file metadata from the database using `file_id`.

    2. Checks if the current user has permission to access the file.
    3. Returns the file content using `send_from_directory`.
- **`process_file`:** *(Likely called asynchronously)*
    1. Takes file path/ID.
    2. Uses appropriate libraries (`pdfminer.six`, `docx2txt`) to extract text content based on file type.
    3. Splits extracted text into manageable chunks.
    4. Generates embeddings for chunks using `fastembed`.
    5. Adds chunks and embeddings to the `VectorStoreModel`.
    6. Updates file status in the database (e.g., 'processed', 'error').

20.4. Service Integration

- **Coordination:** Services can call methods on other services where needed (e.g., `ChatService` uses `VectorStoreModel` which might be populated by `FileService`). A `ServiceManager` class is mentioned but might be overly complex; direct instantiation or dependency injection is common.
- **Dependency Injection:** Consider using a framework or pattern for injecting dependencies (like database connections or other services) into service classes, rather than hardcoding instantiation within them.
- **State Management:** Services are generally stateless, operating based on input parameters and data retrieved from models/database. User-specific state is tied to `user_id`.

20.5. Error Handling (Service Level)

- Define custom exceptions (e.g., `ServiceError`, `ChatServiceError`) inheriting from a base application error.
- Catch specific errors from models or external calls (e.g., `sqlite3.Error`, `requests.exceptions.RequestException`, LLM API errors) and re-raise them as custom service exceptions with appropriate context.
- Return consistent error structures or raise exceptions to be handled by the route layer (which then generates the API response).

20.6. Logging (Service Level)

- Each service should have its own logger instance (`logging.getLogger(__name__)`).
- Log key operations, decisions, and especially errors encountered within the service methods. Include relevant IDs (`user_id`, `conversation_id`, `file_id`).

20.7. Performance Optimization (Service Level)

- **Caching:** Implement caching within services for results that are expensive to compute and don't change often (e.g., `PromptService.get_prompt` if prompts rarely change, potentially results from `VectorStoreModel.search_similar`). Use a proper caching library (e.g., `cachetools`, `Flask-Caching`).
- **Resource Management:** Ensure resources like database connections or file handles are properly managed (acquired and released). Use context managers (`with`).
- **Connection Pooling:** Database connection pooling is handled by the underlying library/driver or Flask extension (e.g., Flask-SQLAlchemy's session management).

20.8. Security Implementation (Service Level)

- **Access Control:** Services should always operate within the context of a `user_id`. Methods interacting with user-specific data must validate that the operation is permitted for that user (e.g., checking ownership before deleting a conversation).
- **Data Protection:** Services should handle data securely, ensuring sensitive information (like API keys) isn't logged inappropriately. Input data received by service methods should ideally be validated (though primary validation often happens at the route/form level).

21. Database Models Documentation

21.1. User Model (`UserModel`)

- **Purpose:** Interface for user-related data in the `users` table.
- **Operations:**
    - `create_user`: Inserts a new record into `users` table (hashes password).
    - `get_user_by_email`: Selects user data based on email.
    - `get_user_by_id`: (Implied) Selects user data based on ID.
    - `update_api_key`: Updates the `groq_api_key` for a given user ID.
    - `update_last_login`: (Implied) Updates `last_login` timestamp.
    - `verify_password`: (Implied, potentially in `auth.py` or `UserService`) Compares provided password with stored hash.

21.2. Chat Model (`ChatModel`)

- **Purpose:** Encapsulates interaction logic with the external Groq LLM API. *Not a database model.*
- **Initialization:** Requires Groq `api_key`. Initializes the Groq client.
- **Operations:**

    - `generate_response`: Takes formatted prompt, history, system message, context; constructs the API request; calls the Groq API; parses and returns the text response. Handles API errors.

21.3. Vector Store Model (`VectorStoreModel`)

- **Purpose:** Manages the FAISS vector index for document similarity search. *Not a database model, interacts with a file (`.pkl`) and in-memory index*.
- **Implementation:** Singleton pattern ensures a single instance manages the shared store.
- **Operations:**
    - `__init__` / `_load_store`: Loads the FAISS index and associated data from `VECTOR_STORE_PATH` if it exists.
    - `create_vectorstore` / `add_documents`: Takes processed document chunks, generates embeddings (using `fastembed`), adds them to the FAISS index, and potentially stores mappings/metadata. Persists changes (`_save_store`).
    - `search_similar`: Takes a query string, generates its embedding, performs a similarity search on the FAISS index, retrieves relevant document chunks/metadata.
    - `_save_store`: Saves the current state of the index and associated data to `VECTOR_STORE_PATH`.

21.4. Conversation Model (`ConversationModel`)

- **Purpose:** Interface for conversation and message data in `conversations` and `chat_history` tables.
- **Initialization:** Requires `user_id` to scope operations.
- **Operations:**
    - `create_conversation`: Inserts a new record into `conversations` for the user.
    - `get_conversations`: Selects recent conversations for the user from `conversations`.
    - `save_message`: Inserts a new record into `chat_history`, linked to a `conversation_id`.
    - `get_messages`: Selects all messages for a given `conversation_id` from `chat_history`, ordered by timestamp.
    - `delete_conversation`: Deletes a conversation record from `conversations` and all associated messages from `chat_history` (using `ON DELETE CASCADE` or explicit deletes).
    - `delete_all_conversations`: Deletes all conversations and associated messages for the user.

21.5. Survey Model (`SurveyModel`)

- **Purpose:** Interface for user survey feedback in the `survey_responses` table.

- **Initialization:** Requires `user_id`.
- **Operations:**
    - `save_survey_response`: Inserts a new record into `survey_responses`.
    - `get_user_survey_responses`: Selects all survey responses for the user.
    - `check_survey_status`: Checks if any records exist in `survey_responses` for the user.

21.6. Database Schema (`db.py`)

- **`users` Table:** Stores user account information.

```
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    useremail TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL, -- Stores hash
    class_standard TEXT NOT NULL,
    medium TEXT NOT NULL,
    groq_api_key TEXT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    last_login DATETIME
);
```

- **`conversations` Table:** Stores conversation metadata.

```
CREATE TABLE IF NOT EXISTS conversations (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    title TEXT NOT NULL,
    is_active BOOLEAN DEFAULT TRUE, -- Assuming active/archived status
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP, -- Needs trigger or app logic to
update
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

- **`chat_history` Table:** Stores individual messages within conversations.

```
CREATE TABLE IF NOT EXISTS chat_history (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    conversation_id INTEGER NOT NULL,
    role TEXT NOT NULL CHECK(role IN ('user', 'bot', 'system')), -- Added 'system' role
possibility
    message TEXT NOT NULL, -- Renamed from 'content' for clarity
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
```

```sql
    FOREIGN KEY (conversation_id) REFERENCES conversations(id) ON DELETE CASCADE
);
```

- `survey_responses` **Table:** Stores user feedback.

```sql
CREATE TABLE IF NOT EXISTS survey_responses (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    rating INTEGER NOT NULL CHECK(rating BETWEEN 1 AND 10),
    message TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

- `user_prompts` **Table:** Stores custom system prompts per user.

```sql
CREATE TABLE IF NOT EXISTS user_prompts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL UNIQUE, -- Assuming one prompt per user
    prompt TEXT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP, -- Needs trigger or app logic
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

- `files` / `documents` **Table:** Needed to store metadata about uploaded files.

```sql
-- Example Schema (Needs to be added)
CREATE TABLE IF NOT EXISTS documents (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    original_filename TEXT NOT NULL,
    stored_filename TEXT NOT NULL UNIQUE, -- Secure name on disk
    file_path TEXT NOT NULL, -- Relative path in UPLOAD_FOLDER
    content_type TEXT, -- MIME type
    size INTEGER, -- File size in bytes
    status TEXT DEFAULT 'uploaded', -- e.g., uploaded, processing, processed, error
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    processed_at DATETIME,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

```
```

- **Indexes:**

```sql
-- For faster lookups based on foreign keys / common queries
CREATE INDEX IF NOT EXISTS idx_conversations_user_id ON conversations(user_id);
CREATE INDEX IF NOT EXISTS idx_chat_history_conversation_id ON
chat_history(conversation_id);
-- CREATE INDEX IF NOT EXISTS idx_user_prompts_user_id ON user_prompts(user_id); --
redundant if UNIQUE
-- Add index for documents table if added:
-- CREATE INDEX IF NOT EXISTS idx_documents_user_id ON documents(user_id);
```

21.7. Model Relationships

- **User <-> Conversations:** One-to-Many (One user has many conversations).
- **User <-> Survey Responses:** One-to-Many (One user can give many responses, though UI might limit to one).
- **User <-> User Prompts:** One-to-One (One user has one custom prompt).
- **User <-> Documents:** (If added) One-to-Many (One user uploads many documents).
- **Conversation <-> Chat History:** One-to-Many (One conversation contains many messages).

21.8. Database Operations (`db.py` helpers)

(Assuming these are implemented in `utils/db.py`)

- **`get_db()`:** Returns a thread-local database connection (`g.db`). Creates connection if none exists for the current context.
- **`close_db()`:** Closes the thread-local database connection. Called automatically via `@app.teardown_appcontext`.
- **`init_db()`:** Executes `CREATE TABLE IF NOT EXISTS` statements and `CREATE INDEX IF NOT EXISTS` statements.
- **CRUD Helpers (Conceptual):** The Model classes likely use helper functions within `db.py` or execute SQL directly using `get_db()`:
  - `execute_query(query, params)`: Executes SELECT statements, returns results (e.g., list of dicts).
  - `execute_update(query, params)`: Executes INSERT, UPDATE, DELETE statements, returns row count or last row ID. Handles commits and potential rollbacks.

21.9. Database Security

- **Injection Prevention:** Use parameterized queries exclusively. Never format SQL strings with user input directly. The `execute_query(query, params)` pattern supports this.
- **Access Control:** Database access credentials should be stored securely (environment variables, secrets management) and not hardcoded. The application logic (services, routes) enforces user-based access to data.
- **Data Protection:** Sensitive data like passwords must be hashed before storing. Consider encryption for other sensitive fields if necessary (e.g., API keys, though hashing might be sufficient if only verification is needed). *(Note: `groq_api_key` is currently stored directly)*.
- **Audit Logging:** Log significant database operations (creates, updates, deletes) potentially using triggers or application-level logging, including user ID and timestamp.

22. Utility Functions and Helpers Documentation

22.1. Database Utilities (`utils/db.py`)

(Format code as blocks)

- **`get_db()`:**

```python
import sqlite3
from flask import current_app, g
import logging

logger = logging.getLogger(__name__)

def get_db():
    """Opens a new database connection if there is none yet for the
    current application context.
    """
    if 'db' not in g:
        try:
            db_path = current_app.config['DATABASE']
            g.db = sqlite3.connect(
                db_path,
                detect_types=sqlite3.PARSE_DECLTYPES
            )
            g.db.row_factory = sqlite3.Row
            # Enable foreign key constraint enforcement for SQLite
            g.db.execute('PRAGMA foreign_keys = ON')
            logger.debug(f"Database connection opened to {db_path}")
        except sqlite3.Error as e:
            logger.error(f"Failed to connect to database at {db_path}: {e}",
exc_info=True)
            # Decide if you want to raise the error or return None/handle differently
            raise # Re-raise to potentially stop app startup or indicate critical failure
        except KeyError:
            logger.error("DATABASE configuration key not found in Flask app config.")
            raise # Config error is critical
    return g.db
```

- **close_db(e=None):**

```
def close_db(e=None):
    """Closes the database again at the end of the request."""
    db = g.pop('db', None)
    if db is not None:
        try:
            db.close()
            logger.debug("Database connection closed.")
        except sqlite3.Error as e:
            logger.error(f"Error closing database connection: {e}", exc_info=True)
```

- **init_db(app):**

```
def init_db(app):
    """Clears existing data and creates new tables."""
    try:
        # No need for app_context here if called from create_app's context
        db = get_db()
        logger.info("Initializing database schema...")
        # --- Execute CREATE TABLE statements (as shown in Schema section) ---
        db.execute("""
        CREATE TABLE IF NOT EXISTS users (...)
        """)
        db.execute("""
        CREATE TABLE IF NOT EXISTS conversations (...)
        """)
        # ... other tables ...
        db.execute("""
        CREATE TABLE IF NOT EXISTS user_prompts (...)
        """)
        # --- Execute CREATE INDEX statements ---
        db.execute("CREATE INDEX IF NOT EXISTS idx_conversations_user_id ON 
conversations(user_id);")
        # ... other indexes ...


        # Commit changes
        db.commit()
        logger.info("Database schema initialized successfully.")
    except sqlite3.Error as e:
        logger.error(f"Database initialization failed: {e}", exc_info=True)
        db.rollback() # Rollback any partial changes
        raise
    except Exception as e: # Catch other potential errors
         logger.error(f"An unexpected error occurred during DB initialization: {e}", 
exc_info=True)
         if 'db' in g and g.db:
             g.db.rollback()
         raise
```

- **Helper Functions (Conceptual):**

```
# Example query helper
def query_db(query, args=(), one=False):
    """Helper to execute SELECT queries."""
    cur = None # Define cur outside try
    try:
        cur = get_db().execute(query, args)
        rv = cur.fetchall()
        cur.close()
        return (rv[0] if rv else None) if one else rv
    except sqlite3.Error as e:
        logger.error(f"Database query failed: {query} | Args: {args} | Error: {e}", 
exc_info=True)
        if cur: cur.close() # Ensure cursor closed on error
        # Decide on error handling: return None, [], or raise exception
        raise # Re-raising is often best for callers to handle


# Example update helper
def update_db(query, args=()):
    """Helper to execute INSERT/UPDATE/DELETE queries."""
    conn = get_db()
    cur = None
    try:
        cur = conn.execute(query, args)
        conn.commit()
        last_row_id = cur.lastrowid # Get last inserted ID if applicable
        cur.close()
        return last_row_id # Or return True/rowcount based on need
    except sqlite3.Error as e:
        logger.error(f"Database update failed: {query} | Args: {args} | Error: {e}",
exc_info=True)
        conn.rollback() # Rollback on error
        if cur: cur.close()
        raise # Re-raise exception
```

22.2. Constants (`utils/constants.py`)

(Format as code block)

```
# Application Limits & Settings
MAX_CONVERSATIONS_PER_USER = 4 # Maximum conversations shown/stored?
MAX_FILE_UPLOAD_SIZE = 100 * 1024 * 1024  # 100 MB (Keep in sync with
Config.MAX_CONTENT_LENGTH)
ALLOWED_UPLOAD_EXTENSIONS = {'txt', 'pdf', 'doc', 'docx'}




# Other Constants...
# Example: Roles for chat messages
CHAT_ROLE_USER = 'user'
CHAT_ROLE_BOT = 'bot'
CHAT_ROLE_SYSTEM = 'system'
```

22.3. Decorators (`utils/decorators.py`)

```
from functools import wraps
from flask import session, redirect, url_for, flash, current_app # Added flash,
current_app
import logging


logger = logging.getLogger(__name__)


def login_required(f):
    """
    Decorator ensures the user is logged in before accessing a view.
    Redirects to the login page if the user is not authenticated.
    Relies on 'user_id' being present in the Flask session.
    """
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'user_id' not in session:
            logger.warning(f"Unauthorized access attempt to '{f.__name__}'. Redirecting
to login.")
            # Flash a message to the user (optional)
            flash('Please log in to access this page.', 'warning')
            # Use url_for with the blueprint name if registered that way
            # Assumes auth blueprint is named 'auth' and has a 'login' route
            return redirect(url_for('auth.login')) # Adjust 'auth.login' if needed
        return f(*args, **kwargs)
    return decorated_function


# Example: Admin required decorator (if roles are implemented)
# def admin_required(f):
#     @wraps(f)
#     @login_required # Ensure user is logged in first
#     def decorated_function(*args, **kwargs):
#         if not session.get('is_admin'): # Check for an admin flag in session
#             logger.warning(f"Non-admin user (ID: {session.get('user_id')}) attempted
admin access to '{f.__name__}'.")
#             flash('You do not have permission to access this resource.', 'danger')
#             return redirect(url_for('main.index')) # Redirect to a safe page
#         return f(*args, **kwargs)
#     return decorated_function
```

22.4. Utility Functions Usage Examples

- **Database Operations (in routes or services):**

```
from .utils import db


# Fetch a single user
user = db.query_db('SELECT * FROM users WHERE id = ?', (user_id,), one=True)
if not user:
    # Handle user not found
    pass


# Insert a new record
new_id = db.update_db('INSERT INTO some_table (col1, col2) VALUES (?, ?)', (val1, val2))
```

- **Authentication Decorator (in routes):**

```
from .utils.decorators import login_required


@chat_bp.route('/my_conversations')
@login_required
def list_my_conversations():
    # This code only runs if user is logged in
    user_id = session['user_id']
    # ... fetch conversations for user_id ...
    return render_template(...)
```

- **File Validation (in file upload route):**

```
from flask import request, current_app
from werkzeug.utils import secure_filename
import os
# Assuming constants are imported
from .utils.constants import ALLOWED_UPLOAD_EXTENSIONS, MAX_FILE_UPLOAD_SIZE


def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in ALLOWED_UPLOAD_EXTENSIONS
```

```
@files_bp.route('/upload', methods=['POST'])
@login_required
def upload_file_route():
    if 'file' not in request.files:
        return jsonify({"error": "No file part"}), 400
    file = request.files['file']
    if file.filename == '':
        return jsonify({"error": "No selected file"}), 400

    # Check file size (using request.content_length might be more reliable)
    if request.content_length > current_app.config['MAX_CONTENT_LENGTH']:
         return jsonify({"error": f"File exceeds maximum size of {MAX_FILE_UPLOAD_SIZE / 
1024 / 1024} MB"}), 413 # Payload Too Large

    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename) # Secure the filename
        # Add unique prefix/suffix to filename before saving to avoid collisions
        unique_filename = f"{session['user_id']}_{uuid.uuid4().hex[:8]}_{filename}"
        save_path = os.path.join(current_app.config['UPLOAD_FOLDER'], unique_filename)
        try:
            file.save(save_path)
            # Trigger file processing / save metadata to DB via FileService
            # ...
            return jsonify({"message": "File uploaded successfully", "filename": 
unique_filename}), 201
        except Exception as e:
            logger.error(f"Failed to save uploaded file {unique_filename}: {e}", 
exc_info=True)
            return jsonify({"error": "Failed to save file"}), 500
    else:
        return jsonify({"error": "File type not allowed"}), 400
```

22.5. Error Handling (Utility Level)

- Utilities that can fail (like database operations, file I/O) should use `try...except` blocks.
- Log errors encountered within the utility function itself.

- Decide whether to handle the error internally (e.g., return `None` or a default value) or re-raise the exception (often preferred) to let the calling code handle the failure appropriately.
- Use specific exception types where possible (e.g., `sqlite3.Error`, `FileNotFoundError`, `ValueError`).

22.6. Logging Configuration (Utility Level)

- Each utility module (`db.py`, `decorators.py`, etc.) should get its own logger instance using `logging.getLogger(__name__)`.
- The main application setup (in `run.py` or `app/__init__.py`) configures the overall logging level, format, and handlers. Utilities will inherit this configuration.
- Use appropriate log levels (e.g., `DEBUG` for connection opening/closing, `INFO` for successful initialization, `WARNING` for recoverable issues, `ERROR` for failures).