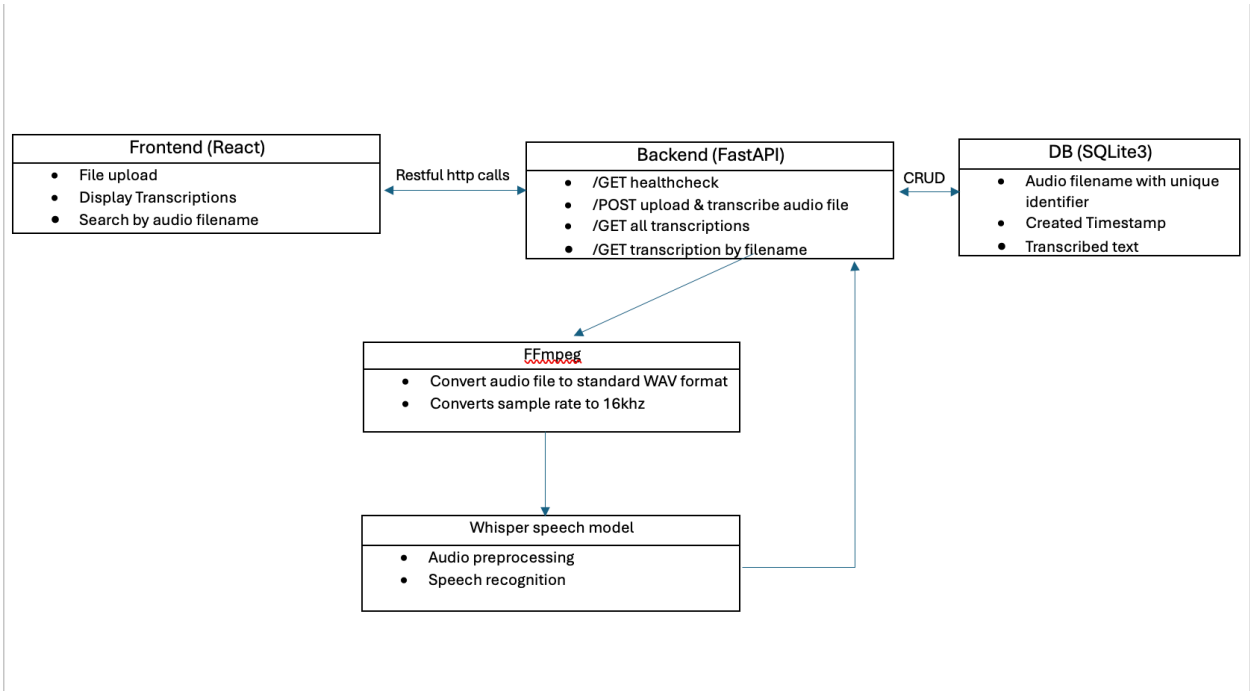
Architecture Diagram



Overview

This system provides an end-to-end solution for **audio transcription** using **FastAPI, Whisper-Tiny, SQLite, FFmpeg and React Framework.** The architecture consists of the following components:

1. **Frontend** – Provides an UI for audio file uploads, transcription display, and search using React framework.
2. **Backend** – Exposes REST API endpoints for audio processing and querying transcriptions using FastAPI.
3. **FFmpeg** – Preprocesses uploaded audio files (format conversion, resampling, and normalization).
4. **Whisper-Tiny Model** – Performs speech-to-text transcription.
5. **Database (SQLite)** – Stores transcription metadata.

2. Backend (FastAPI Application)

- Implements the following **REST API endpoints**:
- **GET /health** : Returns service status.
- **POST /transcribe** : Accepts audio files, preprocesses them using **FFmpeg**, transcribes them using **Whisper-Tiny**, and stores the results.
- **GET /transcriptions** : Retrieves all transcriptions from the database.
- **GET /search** : Searches transcriptions by filename.
- Manages **audio preprocessing, transcription, database interactions, and storage**.

3. FFmpeg (Audio Preprocessing)

Ensures that all uploaded audio files are in a **compatible format** before being processed by Whisper.

Tasks performed:

- **Format Conversion**: Converts audio files (MP3, WAV, M4A, etc.) into a standard **WAV** format.
- **Resampling**: Converts the sample rate to **16kHz**, which is required by Whisper.

4. Whisper-Tiny (Speech-to-Text Model)

- Receives preprocessed audio from FFmpeg.
- Runs speech recognition to generate transcriptions.
- Returns the transcribed text to the backend.

5. Database (SQLite)

- Stores transcription metadata:
- Audio file name.
- Transcribed text.
- Timestamp of creation.
- Optimized for fast querying via the search API.

6. File Storage (Local)

- Stores uploaded audio files.
- The backend retrieves files from storage for processing.

How the System Works (Workflow)

1. User uploads an audio file via the frontend.
2. The file is sent to the backend (FastAPI) through the /transcribe API.
3. FFmpeg preprocesses the audio:
    i. Converts the file to WAV (if necessary).
    ii. Resamples it to 16kHz.
4. The Whisper-Tiny model transcribes the processed audio and returns the text transcription.
5. The backend stores the transcription in the SQLite database.
6. The user can view transcriptions via the /transcriptions API or search for specific files using /search.
7. The frontend displays the results to the user.

Assumptions

1. File Upload & Preprocessing

- Audio files may be uploaded in **different formats** (MP3, WAV, M4A, etc.), so **FFmpeg must be installed** on the backend to handle conversions.
- Audio files must be resampled to 16kHz since Whisper-Tiny requires this for accurate transcription.

2. Whisper Model & Performance

- The system uses **Whisper-Tiny** due to its small size and efficiency, but it may not perform as well on long or low-quality recordings.
- Inference time depends on hardware (CPU vs. GPU). If GPU acceleration is available, transcription speed will be significantly faster.

3. Database & Storage

- SQLite is chosen for simplicity for this short technical assessment project, but a more scalable option like **PostgreSQL** may be required in production.
- Audio files and transcriptions must be linked using a **unique identifier** in the database.
- The storage system can be local (disk-based) or cloud-based (e.g., AWS S3)

4. API & Request Handling

- All API requests must be validated to prevent file corruption or security vulnerabilities.

5. Security & Authentication

- The system does not currently include authentication, but in production, API access should be restricted (e.g., JWT tokens, OAuth).
- File uploads should be scanned to prevent malicious payloads.

Considerations

1. Scalability & Performance

- **Whisper-Tiny is a small model**, but larger Whisper versions (e.g.Whisper Large) could improve accuracy at the cost of more processing power.
- For **high-traffic use cases**, deploying the system with asynchronous processing **(**Celery, Redis queues**)** can help handle multiple transcriptions simultaneously.
- **Database indexing and caching** should be considered to speed up search queries.

2. User Experience (UX) Considerations

- Frontend should show real-time progress updates during transcription.
- Users should receive error messages if their file format is unsupported or transcription fails.

3. Deployment & Containerization

- The backend should be containerized with Docker, ensuring consistency across environments.
- A multi-container setup (Docker Compose or Kubernetes) should be used in production for better scalability.
- Environment variables should be used to manage database credentials, API keys, and file storage configurations.