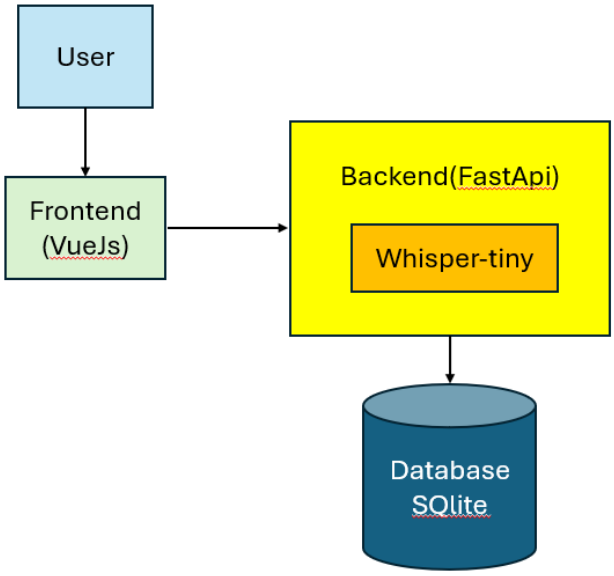

Diagram Explaination

1. **Frontend (Vue.js)**:
    - The frontend is still built with **Vue.js**, providing the user interface for the application where users can upload audio files and view transcription results.
    - It communicates with the **Backend** through HTTP requests (using Axios or Fetch) to interact with the endpoints (/upload, /status, /transcript).
2. **Backend (Python/Flask)**:
    - The **backend** is still implemented using **Python** with the **FaskAPI** framework. It handles all the logic for processing the uploaded audio files, interacting with the transcription service, and managing status updates.
    - The backend exposes RESTful API endpoints for the frontend to consume.
    - The backend now uses **SQLite** for local, file-based data storage. It stores metadata like file details, transcription status, and the resulting text.
3. **Database (SQLite)**:
    - The database used here is **SQLite**, which is a lightweight, file-based database. Since it's embedded, there's no need for an external database service.
    - SQLite is suitable for applications that don’t require heavy scaling or distributed databases, which works well for smaller apps or apps running in local environments.

- The backend interacts with SQLite through SQLAlchemy connection to store and retrieve data related to audio file uploads and transcription results.

4. **Docker and Docker Compose**:
    - The **Docker** containers for both the **frontend** and **backend** will encapsulate the entire application. Since the database is SQLite (file-based), no separate database container is needed.
    - The Docker Compose file ensures that the **frontend** and **backend** containers can communicate with each other on a shared network, making it easy to deploy the entire application.

Key Considerations

1. **Audio Preprocessing:** The audio input must be preprocessed and normalized to a range of -1.0 to 1.0. If this step is skipped, Whisper-Tiny may fail to transcribe the audio correctly or produce inaccurate results, as it expects input audio within this normalized range for optimal performance.

2. **Database Usage:** The SQLite database stores metadata about the uploaded audio files, transcription statuses, and the resulting transcriptions. Additionally, audio files are saved to a folder (e.g., uploads) for future retrieval, in case a feature is added to allow users to download their files again. SQLite is lightweight and suitable for local development and small-scale applications.

3. **Dockerization:** Both the frontend and backend are containerized using Docker, and the application can be run using Docker Compose. This simplifies deployment and ensures consistent development environments across different machines.

4. **Error Handling and Response Time:** The system should gracefully handle errors such as unsupported audio formats, failed transcription processes, and network issues. To ensure smooth operation, the frontend validates the file type before uploading, and the backend performs additional validation to confirm the file is in the correct format. Transcription might take time depending on the audio length, so response time must be optimized, especially for larger files.

Assumptions:

1. **Environment Setup:** The user is expected to have Docker and Docker Compose installed for an easy setup. Alternatively, local setups for the frontend (Vue.js) and backend (FastAPI) can be used.
2. **Audio File Format:** The application assumes the audio files provided for transcription are in a supported format (e.g., MP3, WAV). The system will not handle unsupported formats.
3. **Whisper Model Limitations:** The Whisper-Tiny model has limitations in transcription quality compared to larger Whisper models (e.g., Whisper-Base, Whisper-Large). This smaller model may struggle with achieving high transcription accuracy. Additionally, for cases involving Singlish, terms like "yong tau foo" may be incorrectly transcribed as "young too foo."