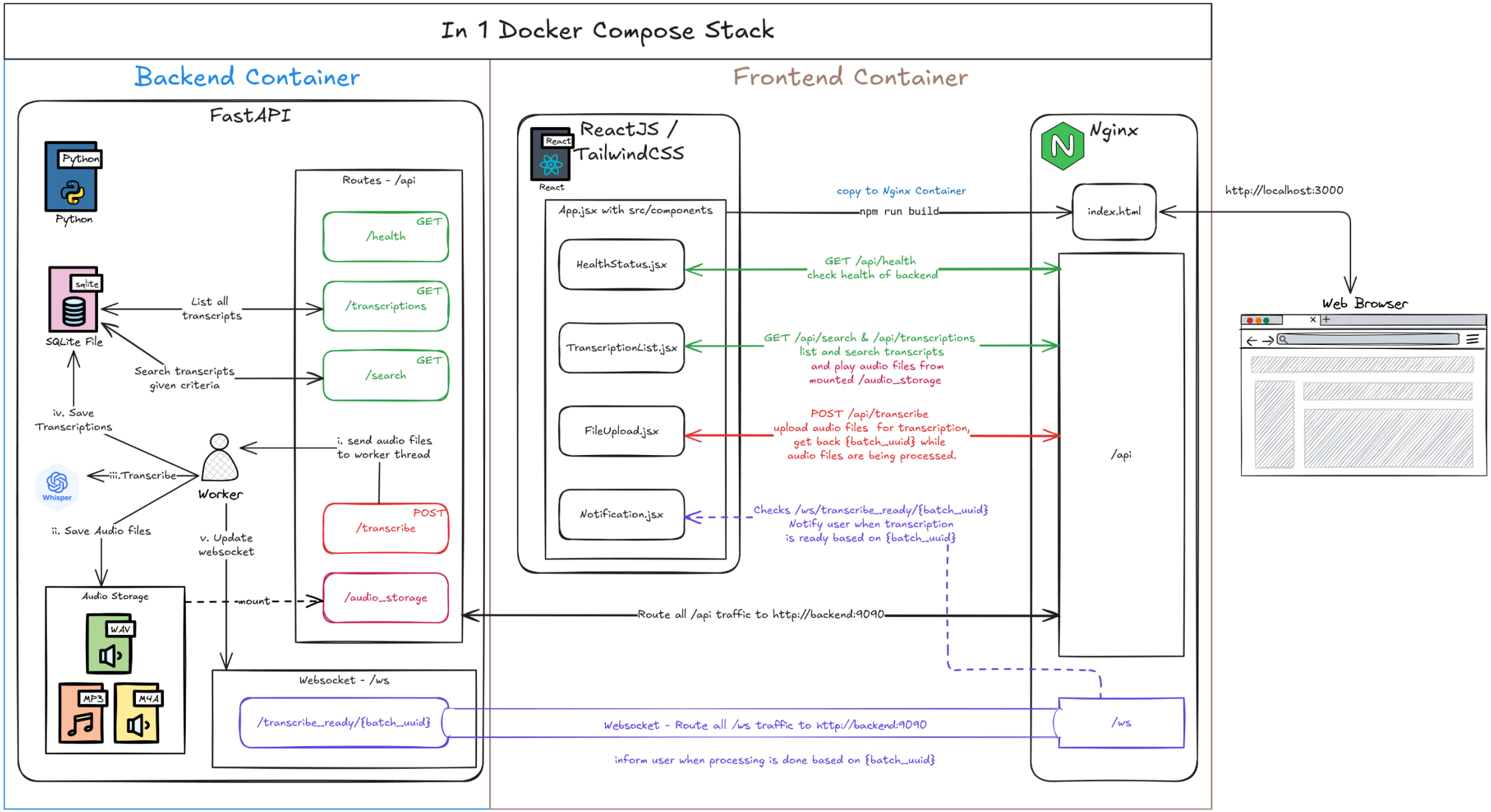
In 1 Docker Compose Stack
Backend Container
FastAPI
Python
Python
sqlite
SQLite File
List all transcripts
Search transcripts given criteria
iv. Save Transcriptions
Routes - /api
GET
/health
GET
/transcriptions
GET
/search
i. send audio files to worker thread
iii.Transcribe
Whisper
Worker
ii. Save Audio files
v. Update websocket
POST
/transcribe
Audio Storage
WAV
MP3
M4A
mount
/audio_storage
Websocket - /ws
/transcribe_ready/{batch_uuid}
Frontend Container
React
ReactJS / TailwindCSS
React
App.jsx with src/components
HealthStatus.jsx
TranscriptionList.jsx
FileUpload.jsx
Notification.jsx
copy to Nginx Container
npm run build
Nginx
index.html
http://localhost:3000
Web Browser
GET /api/health
check health of backend
GET /api/search & /api/transcriptions
list and search transcripts
and play audio files from
mounted /audio_storage
POST /api/transcribe
upload audio files for transcription,
get back {batch_uuid} while
audio files are being processed.
Checks /ws/transcribe_ready/{batch_uuid}
Notify user when transcription
is ready based on {batch_uuid}
/api
Route all /api traffic to http://backend:9090
/ws
Websocket - Route all /ws traffic to http://backend:9090
inform user when processing is done based on {batch_uuid}

Overview

The diagram illustrates a containerized full-stack application for audio transcription built with Docker Compose. The architecture consists of two main containers (frontend and backend) that communicate through RESTful API endpoints, proxied through Nginx, and WebSocket for real time updates.

Key Components

Backend Container

The backend is implemented using FastAPI, a modern Python web framework, with the following components:

1. **API Routes**: Four main RESTful endpoints as required in the assignment:
   - `GET /health`: Simple health check endpoint that returns `{"status": "OK"}`
   - `GET /transcriptions`: Retrieves all transcriptions from the database
   - `GET /search`: Searches transcriptions based on file name with options for exact matching and case sensitivity
   - `POST /transcribe`: Accepts audio file uploads for transcription
2. **Database**: SQLite for storing transcription data
   - The database stores audio file paths, original filenames, transcribed text, and timestamps
   - Mounted as a volume to persist data between container restarts
3. **Audio Storage**: A dedicated volume for storing uploaded audio files
   - Files are mounted and made accessible through FastAPI's StaticFiles middleware
4. **WebSocket Interface**: Real-time updates for transcription status
   - Clients connect to `/ws/transcript_ready/{batch_uuid}` to receive updates
   - Notifications for completed batch processing
5. **Whisper Integration**: Background processing of audio files
   - Uses OpenAI's Whisper model for speech recognition
   - Processing occurs asynchronously in background tasks

Frontend Container

The frontend is a React application with TailwindCSS, structured with the following components:

1. Nginx Server: Serves the built React application
   - Routes API requests to the backend container
   - Exposed on port 3000 for external access
2. React Components:
   - `HealthStatus`: Monitors backend availability with periodic checks
   - `FileUpload`: Interface for uploading single or multiple audio files
   - `TranscriptionList`: Displays all transcriptions with search capabilities
   - `Notification`: Real-time feedback system for transcription status
3. WebSocket Client: Connects to backend for real-time updates
   - Listens for transcription completion events
   - Updates UI with notification system

Communication Flow

1. Health Check Flow:
    - Frontend periodically polls `GET /api/health` to ensure backend availability
    - UI updates to show connection status
2. Transcription Listing Flow:
    - TranscriptionList requests data from `GET /api/transcriptions`
    - Audio files are played directly from the mounted storage path
    - Search functionality sends queries to `GET /api/search` with parameters for matching options
3. File Upload Flow:
    - User uploads audio files through FileUpload component
    - Files are POSTed to `/api/transcribe` endpoint
    - Backend assigns a batch UUID to the batch of files that the user uploads and sends it to a worker thread in the background for processing
        - Batch UUID is unique to each upload batch, be it single file or batch files (multiple files in one upload)
        - Sends transcription tasks to background worker threads asynchronously
        - Update the frontend that files are processed successfully, updating them with the batch UUID
    - Frontend establishes WebSocket connection based on the returned batch UUID to receive updates
    - Meanwhile, the background worker thread would then:
        - Save all audio files uniquely by appending the batch UUID at the front of the file names
        - For all files uploaded in this batch:
            - Send each audio file for transcription by Whisper
            - Save the transcription outputted from Whisper into SQLite DB
            - Update the WebSocket connection every time a file is finished processing with status “completed”
            - Repeat for the rest of the files
        - Finally, once all files in a batch is completed,
            - For single file uploads, “job_completed” status will be sent out when processing of the uploaded file is completed
            - For batch uploads, “batch_completed” status will be sent out when processing of all files that were uploaded in the same batch is completed.
4. WebSocket Notification Flow:
    - Frontend use the batch UUID received from backend and use it to establish a unique WebSocket connection to the backend to receive status updates on the audio files being processed
    - Different status received will display the appropriate notifications
    - When processing completes, transcription list is automatically refreshed

Assumptions and Considerations

1. Docker Networking:
   - Backend and frontend containers communicate through an internal Docker network called "app-network"
   - Only the frontend container exposes a port (3000) to the host machine
   - Backend container is not directly accessible from outside the Docker network
2. Data Persistence:
   - Both the database and audio files are stored in Docker volumes for persistence
   - The application can be restarted without losing data
3. Synchronization and Dependency:
   - Frontend container waits for backend health check to pass before starting
   - This ensures the application is only available when the entire system is operational
4. Audio Format Support:
   - The application handles .wav, .mp3, and .m4a audio formats as specified in the code
   - In terms of audio pre-processing such as: Voice Activity Detection (VAD), speaker detection, word level timestamps output, use project: https://github.com/SYSTRAN/faster-whisper
5. Search Capabilities: Search can be performed with case sensitivity and whole-word matching options
   - The search is implemented on the backend, as confirmed through email.
6. Timezone Configuration: Backend container is configured with Singapore timezone (Asia/Singapore)
7. Notification System for File Uploads:
   - Different notification types (success, warning, error) provide visibility to the user on what went wrong with the file upload
   - Temporary notifications with progress bars that auto-dismiss
8. Scalability and User Experience Considerations:
   - Background tasks handle processing to avoid blocking the API
   - WebSockets provide efficient real-time updates without polling
9. Container Optimization:
   - Multi-stage builds for frontend to minimize container size
   - Python dependency management with UV for backend efficiency
   - Health checks ensure container readiness