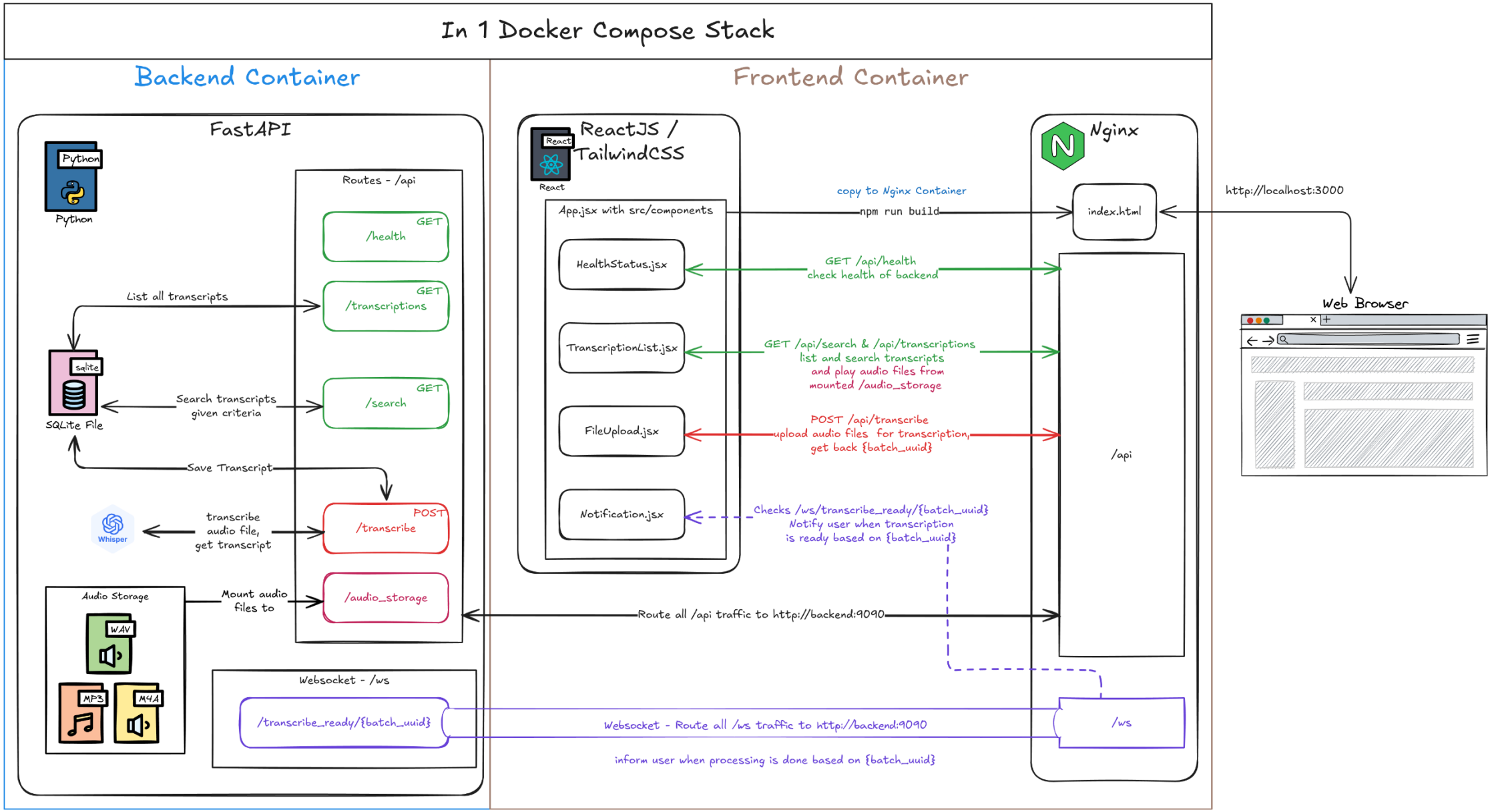
In 1 Docker Compose Stack
Backend Container
Frontend Container
FastAPI
Python
Routes - /api
GET
/health
GET
/transcriptions
GET
/search
POST
/transcribe
/audio_storage
List all transcripts
Search transcripts
given criteria
Save Transcript
transcribe
audio file,
get transcript
Mount audio
files to
sqlite
SQLite File
Whisper
Audio Storage
WAV
MP3
M4A
Websocket - /ws
/transcribe_ready/{batch_uuid}
ReactJS /
TailwindCSS
React
App.jsx with src/components
HealthStatus.jsx
TranscriptionList.jsx
FileUpload.jsx
Notification.jsx
copy to Nginx Container
npm run build
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}
Checks /ws/transcribe_ready/{batch_uuid}
Notify user when transcription
is ready based on {batch_uuid}
Route all /api traffic to http://backend:9090
Nginx
index.html
/api
/ws
Websocket - Route all /ws traffic to http://backend:9090
inform user when processing is done based on {batch_uuid}
http://localhost:3000
Web Browser

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. 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 begins processing
        - Batch UUID is unique to each upload batch, be it single file or batch files (multiple files in one upload)
    - Frontend establishes WebSocket connection based on the returned batch UUID to receive updates
    - Notifications appear as files are processed
3. 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
4. WebSocket Notification Flow:
    - Backend sends status updates through WebSocket connections
        - For every single file processed, backend would send out “completed” status to the particular batch UUID that the audio file that was processed belongs to
        - 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.
    - Frontend receives and displays appropriate notifications
    - When processing completes, 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