NYC Subway Chatbot Assistant - Developer Documentation

1. Project Overview

The **NYC Subway Assistant** is an intelligent chatbot capable of planning trips, providing real-time train arrivals, and reporting service alerts for the New York City Subway system. It leverages **FastAPI** for the backend, **NetworkX** for routing logic, **SQLite** for data storage, and **OpenAI GPT-4** for natural language understanding and orchestration.

2. System Architecture

The project consists of the following key layers:

1. **Client Layer**: A lightweight HTML/JS frontend that communicates with the backend via REST API.
2. **API Layer (`backend/api`)**: A FastAPI application that handles chat requests, manages session history, and forwards prompts to the LLM.
3. **LLM Orchestration (`backend/llm`)**: Uses OpenAI's Function Calling (Tools) to determind when to query the subway data vs. just chatting.
4. **Service Layer (`backend/services`)**:
   - **GraphBuilder**: Constructs a directed weighted graph (NetworkX) from static GTFS data.
   - **StationService**: Provides fuzzy string matching to map user input (e.g., "Grand Central") to GTFS `stop_ids`.
   - **Routing**: Implements Dijkstra's algorithm (via NetworkX) to find the shortest path.
5. **Data Layer (`backend/data`)**:
   - **SQLite Database**: Stores static GTFS data (stops, routes, trips) and real-time buffer (updates, alerts).
   - **Ingestion Scripts**: Parsers to load raw `.txt` GTFS files into the database.
   - **Real-time Worker**: A background process (not fully detailed here but implied) that fetches updates from the MTA API.

---

3. Getting Started

Prerequisites

- Python 3.9+
- MTA API Key (for real-time data)
- OpenAI API Key (for the chatbot)

Installation

1. **Clone the Repository**: `bash git clone <repo_url> cd Prototype`
2. **Install Dependencies**: `bash pip install fastapi uvicorn sqlalchemy networkx pandas fuzzywuzzy python-multipart openai python-dotenv # Note: You might need 'python-Levenshtein' for faster fuzzy matching`
3. **Environment Variables**: Create a `.env` file in the root directory: `env OPENAI_API_KEY=sk-... MTA_API_KEY=...`

Database Initialization

Before running the app, you must populate the SQLite database with static GTFS data.

1. Ensure you have the unzipped GTFS files in `gtfs_subway/`.
2. Run the ingestion script: `bash python backend/data/ingest_gtfs.py` This creates `backend/data/subway.db`.

Running the Application

1. **Start the Backend Server**: `bash python backend/api/main.py` The server will start on `http://0.0.0.0:8000`.
2. **Launch the Frontend**: Simply open `frontend/index.html` in your browser. It is configured to talk to `localhost:8000`.

---

4. Backend Documentation

Core API (`backend/api/main.py`)

- **POST /chat**:
  - Input: `{"message": "How do I get to Times Square?", "session_id": "abc-123"}`
  - Output: `{"response": "Take the 1 train..."}`
  - Logic: Maintains conversation history in memory. Uses GPT-4 to decide whether to call a tool (`plan_trip`, `get_next_trains`) or answer directly.

LLM Tools (`backend/llm/tools.py`)

These Python functions are exposed to GPT-4:

- `tool_plan_trip(origin, dest)`:
  - Resolves station names using `StationService`.
  - Calculates shortest path using `NetworkX`.

    - Returns a human-readable itinerary.
    - Handles ambiguity (e.g., "Did you mean 86th St (1) or 86th St (Q)?").
- `tool_get_next_trains(station, route_filter)`:
    - Queries `realtime_updates` table for incoming trains.
- `tool_get_alerts()`:
    - Fetches active service alerts from `realtime_alerts` table.

Services (`backend/services`)

- **StationService**:
    - Loads all station names from the DB into memory on startup.
    - Uses `fuzzywuzzy` to match user queries (e.g., "Main St") to the best candidate.
    - Returns a confidence score to prevent hallucinations.
- **GraphBuilder**:
    - Builds the routing graph.
    - **Nodes**: Stops (Child platforms) and Parents (Complexes).
    - **Edges**:
        * `TRACK`: Actual train movement (weighted by median travel time from scheduled trips).
        * `TRANSFER`: Walking connections between lines (weighted by `min_transfer_time`).
        * `STATION_PATH`: Logic link between Parent and Child nodes.

Data Models (`backend/data/schema.py`)

- **Stop**: Maps to `stops.txt`. Contains `stop_id`, `stop_name`, `location_type`.
- **Trip**: Maps to `trips.txt`. Links routes to specific physical trips.
- **StopTime**: Maps to `stop_times.txt`. The schedule sequence.
- **RealtimeUpdate**: Stores live arrival times (transient data).

---

5. Frontend Structure

The frontend is a single-file application (`index.html`): * **Tech Stack**: Vanilla HTML/CSS/JS. * **Markdown Support**: Uses `marked.js` to render rich text responses from the bot. * **State**: Generates a random `session_id` on load to maintain context with the backend.

---

6. Real-Time Data Flow

1. **Ingestion**: A separate worker script (e.g., `inspect_rt.py` or a dedicated cron job) fetches Protocol Buffers from the MTA API.
2. **Storage**: Parses `TripUpdate` and `Alert` entities and upserts them into `realtime_updates` and `realtime_alerts` tables.

3. **Query**: When `tool_get_next_trains` is called, it queries the DB for arrivals `> now()` for the specific `stop_id`.

---

7. Troubleshooting

- **"Ambiguous Station"**: The system asks for clarification if multiple stations match a name with high confidence (e.g., "Spring St" exists on both the 6 and C/E lines).
- **"No Route Found"**: Usually means the graph is disconnected. Check if `transfers` were loaded correctly.
- **Database Locks**: SQLite handles limited concurrency. For production, switch to PostgreSQL.

8. Development Roadmap

- **Phase 1 (Complete)**: Static Graph, Basic Routing, LLM Interface.
- **Phase 2 (In Progress)**: Robust Real-time integration (Feed consumption).
- **Phase 3**: Graph updates based on delays (Dynamic Edge Weights).
- **Phase 4**: User personalization (Favorite routes).