Project Documentation: Smart Ambulance Routing

This document outlines the step-by-step development process of the EMsync Smart Ambulance Routing simulation feature.

routing_env : Anaconda

Phase 1: Initial Proof of Concept (Python Script)

The initial goal was to establish a connection with the routing API and visualize a static route.

- **Step 1: Environment & API Setup** ⚙️
  - A Conda environment (routing_env) was created to manage project dependencies.
  - **OpenRouteService (ORS)** was chosen as the primary routing API due to its generous free tier.
  - Libraries installed: requests (for API calls), folium (for mapping), and polyline (for decoding).
- **Step 2: Core API Connection** 🌐
  - A Python script (routing_test.py) was created to perform a basic API call.
  - The script successfully fetched a route between two hardcoded points in Kerala.
  - **Debugged KeyError:** We analyzed the API's JSON response to correct the path for accessing the route data (from features to routes), establishing the correct data structure.
- **Step 3: Static Visualization** 🗺️
  - The folium library was used to parse the route's geometry.
  - The script was updated to generate a static route_map.html file, successfully displaying the start point, endpoint, and the calculated route on an interactive map.

Phase 2: Pivot to a Modern Web Architecture

It was determined that the animation quality using a script-based or Streamlit "flipbook" method was insufficient. A decision was made to pivot to a more professional and scalable architecture.

- **Architecture:** A **client-server model** was adopted.
  - **Backend:** A Python API using the **Flask** framework to handle all routing logic and

communication with the ORS API.
    - **Frontend:** A modern web application using **React** to provide a smooth, interactive user experience and handle all visualizations.

Phase 3: Backend Development (Flask API)

The backend was built to serve route data to the frontend.

- **Step 4: Flask Server Setup** 🖥️
    - The **Flask** and **Flask-CORS** libraries were installed. CORS was crucial to allow the React app (on a different port) to communicate with the backend.
    - A basic Flask server was created in backend_api.py, configured to run on port 5000.
- **Step 5: Secure API Key Management** 🔒
    - To avoid hardcoding the API key, we implemented a secure secret management solution.
    - The python-dotenv library was installed.
    - The ORS API key was moved into a **.env** file, which was then added to **.gitignore** to prevent it from being shared. The Flask app now securely loads the key from this file.
- **Step 6: API Endpoint Creation & Logic** 🧠
    - A single endpoint, **/api/get_route**, was created to handle POST requests.
    - **Logic Simplification:** We made the architectural decision to have the backend handle the **polyline decoding**. The endpoint now receives start/end coordinates and returns a complete JSON object that includes the simple, ready-to-use list of decoded [lat, lon] coordinates.

Phase 4: Frontend Development (React App)

A dynamic frontend was built from scratch to consume the backend API and provide the user interface.

- **Step 7: React Project Setup** ⚛️
    - A new project was created using npx create-react-app emsync-frontend.
    - Libraries installed: axios (for API calls), leaflet, and react-leaflet (for mapping).
- **Step 8: UI/UX Refactoring** ✨
    - The UI was structured into a clean layout with a **sidebar** for controls and a main panel for the map.
    - The code was refactored into components (App.js, Controls.js) for better organization.

    - Interactive **dropdown menus** were added to the sidebar, allowing the user to dynamically select start and end locations.
- **Step 9: Smooth Client-Side Animation** 🚑
    - The core challenge of the "flipbook" effect was solved.
    - The animation is now handled entirely on the client-side using React state (useState) and browser timing events (setInterval within a useEffect hook).
    - This results in a **smooth animation** of the ambulance marker moving along the route, as it no longer requires a full page reload for each frame.

Current Status

The Smart Ambulance Routing feature is a functional prototype with a secure Python backend and a polished React frontend. It successfully demonstrates dynamic route calculation and a smooth, interactive simulation.