**Real-Time Roadside Assistance Website**

**This is a full-stack emergency roadside assistance website developed to cater to both 2-wheeler and 4-wheeler users.**

**Key Features & Functionality:**

* **Real-time Location Services: The website integrates the Google Maps API for dynamic navigation, live location detection, and real-time location mapping, enabling users to share their location.**
* **Service Tracking: Users can track the status of their requested services.**
* **Communication: It facilitates user-to-provider communication, including chat and call functions.**
* **Resource Location: Users can contact and locate nearby garages, fuel stations, and towing services.**
* **User Authentication: Login authentication is implemented to secure user access.**

**Technology Stack:**

* **Frontend: Built using ReactJS and React Native.**
* **Backend: Developed with Flask (Python), featuring robust backend logic and RESTful APIs.**
* **Databases: Utilizes both SQL and MongoDB for persistent data handling.**
* **Version Control: The codebase is maintained using GitHub version control.**

**Project Timeline: The development of this website began in January 2025 and is currently ongoing.**

**Sources**

Developer's View:

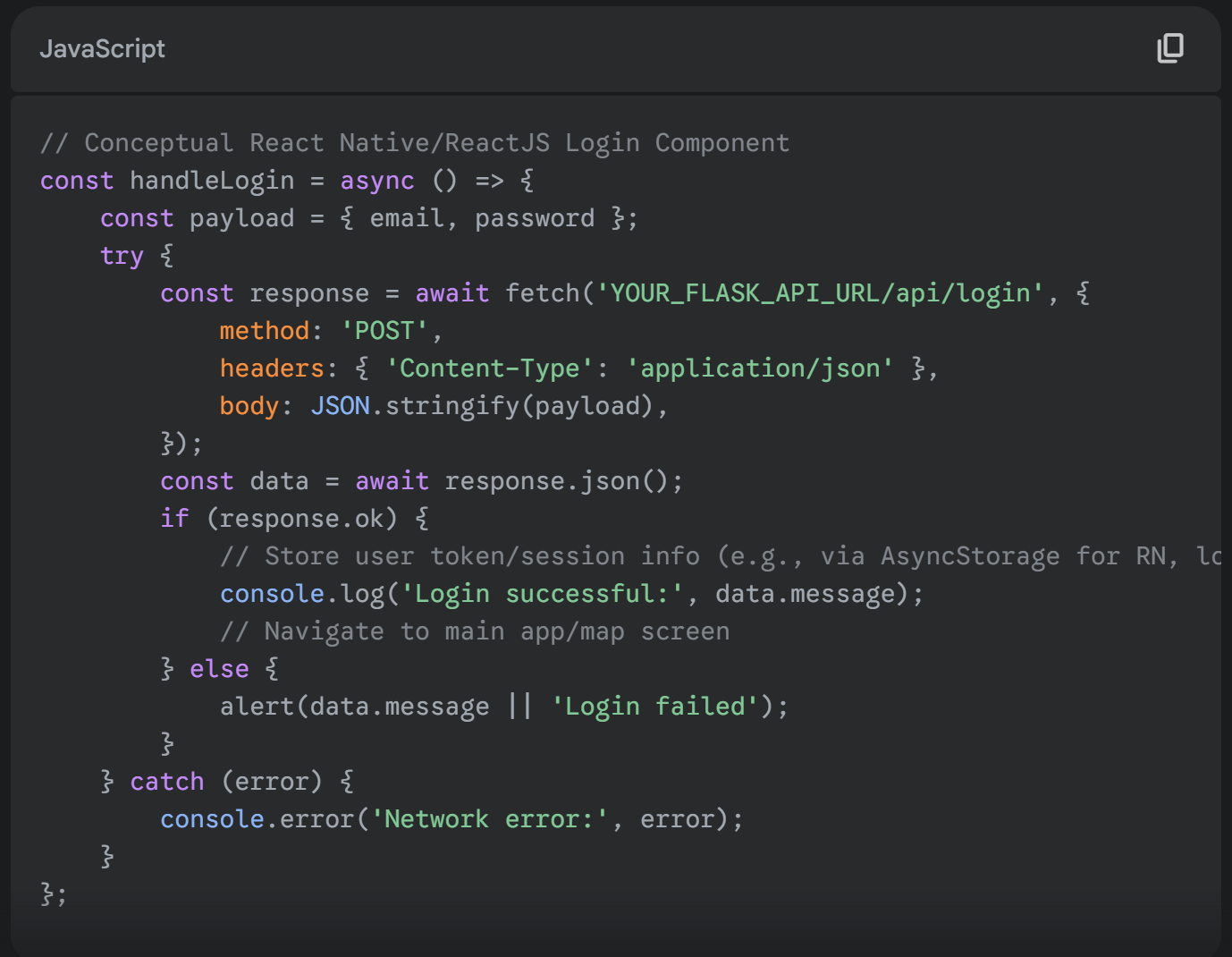
Real-Time Roadside Assistance Website - Conceptual End-to-End Flow

This project aims to provide immediate roadside assistance, connecting users in distress with nearby service providers. The core challenge lies in real-time location handling, efficient service matching, and robust communication.

**1. User Onboarding & Authentication (Sign Up / Login)**

Similar to the car rental system, user authentication is fundamental.

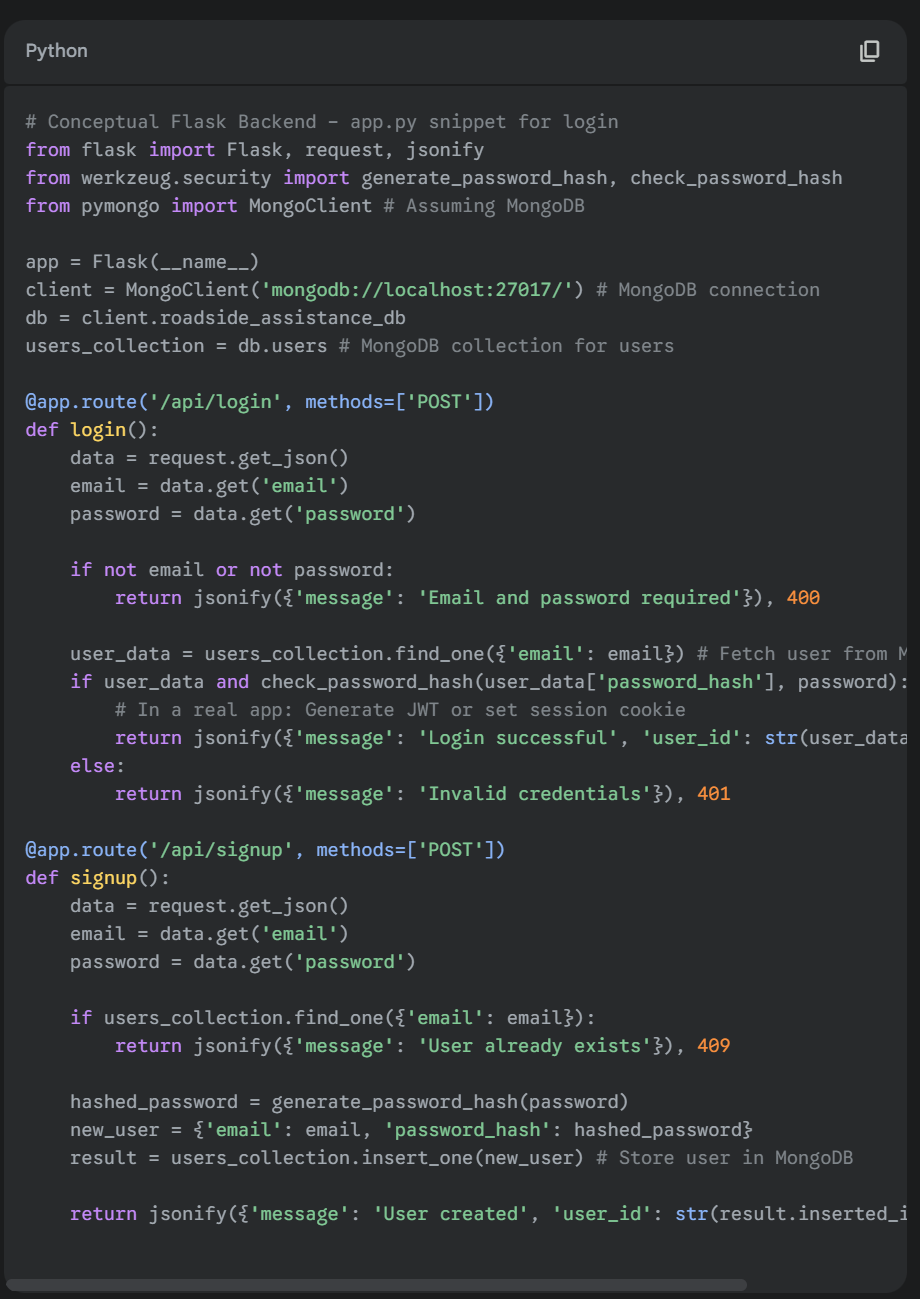
* **Frontend (React Native/ReactJS):**
  + **UI:** User-friendly forms for account creation and login.
  + **Data Capture:** Collects user's email, password, and possibly contact details.
  + **Data Transmission:** Sends credentials (e.g., POST request) to the Flask backend using fetch or a similar HTTP client library.



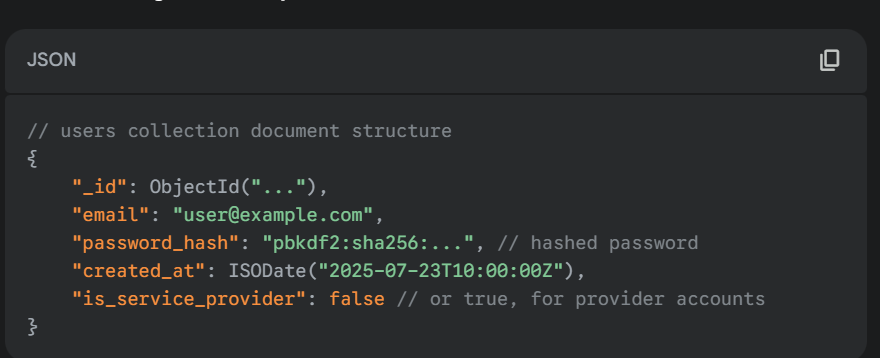
*Explanation:* Frontend captures user input and sends it as a JSON payload to the backend.

**Backend (Python Flask):**

* **Route:** Defines POST /api/signup and POST /api/login endpoints.
* **Data Reception:** request.get\_json() to parse incoming JSON.
* **Signup Logic:** Hashes passwords using werkzeug.security.generate\_password\_hash before storing. Checks for email uniqueness.
* **Login Logic:** Retrieves user by email from **MongoDB** (or SQL, if a hybrid approach). Uses werkzeug.security.check\_password\_hash to verify the password. Upon successful login, typically generates a JWT (JSON Web Token) or establishes a session ID to maintain user state across requests.

 *Explanation:* Flask receives login/signup requests. For signup, passwords are hashed before storing in the users\_collection in MongoDB. For login, it retrieves the user document from MongoDB and uses check\_password\_hash for verification.

**Database (MongoDB / SQL):**

* **Schema (MongoDB Conceptual):**
* *Explanation:* MongoDB documents (users collection) would store user details, including the securely hashed password.

**2. Real-Time Location Detection & Service Request**

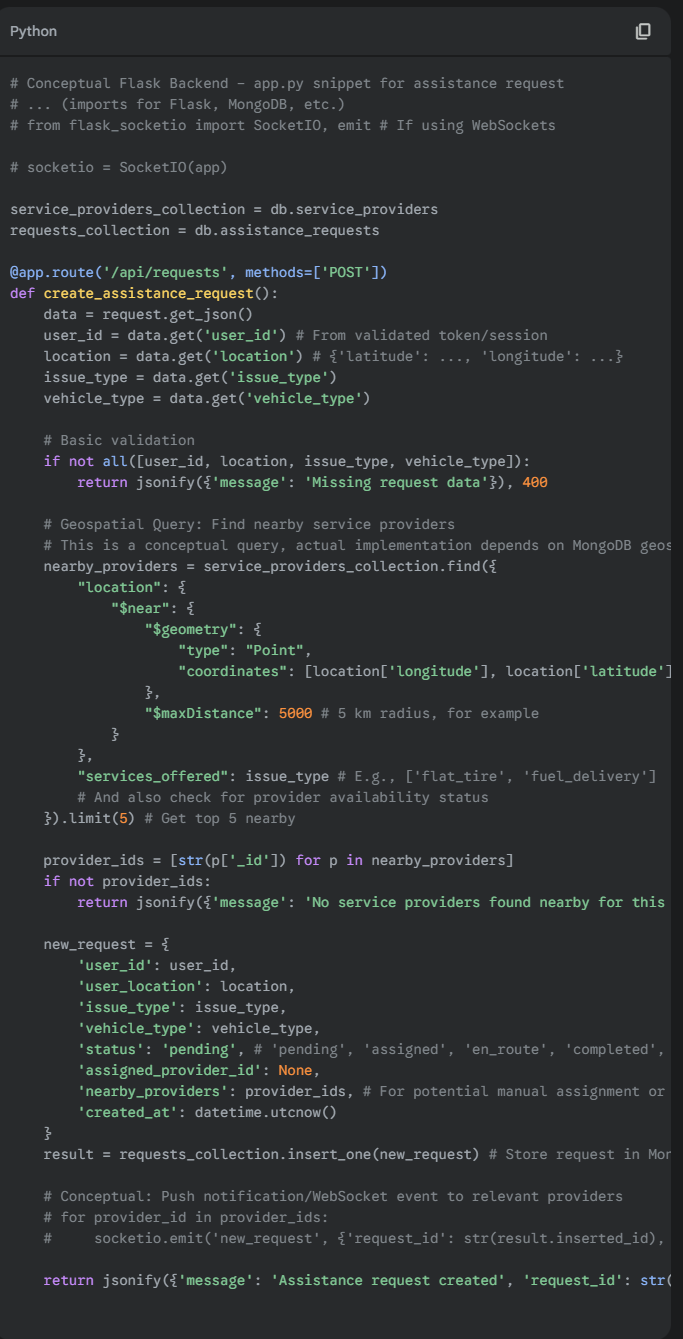
This is the core functionality involving user location, API integration, and initiating a service request.

* **Frontend (React Native/ReactJS):**
  + **Geolocation API:** Uses the device's built-in Geolocation API (for mobile via React Native, for web via browser API) to get the user's current latitude and longitude.
  + **Map Display:** Integrates a map component (e.g., react-native-maps for RN, Google Maps JavaScript API for web) to display the user's location.
  + **Sending Request:** When the user requests assistance, current location (lat/lng), issue type (e.g., "flat tire", "out of fuel"), and vehicle type (2-wheeler/4-wheeler) are sent to the backend.

*Explanation:* The frontend uses the device's geolocation capabilities to get the user's coordinates and then dispatches a POST request to the backend with the location and request details.

**Backend (Python Flask):**

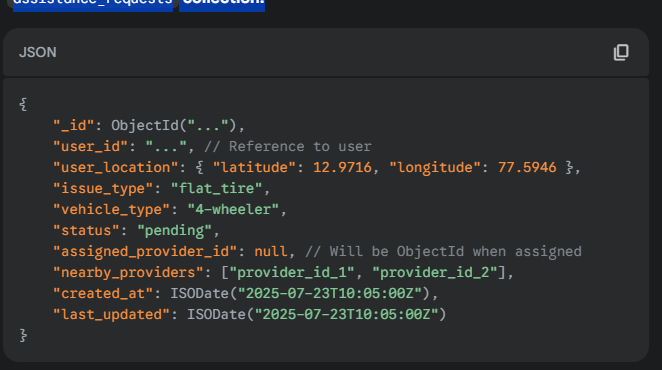
* **Route:** POST /api/requests endpoint.
* **Data Reception:** Receives user's location, issue type, etc.
* **Service Matching Logic:**
  + Queries a service\_providers collection/table in **MongoDB** (or SQL) to find nearby relevant service providers (garages, fuel stations, towing services) based on the user's location and the service\_type they offer.
  + This typically involves geospatial queries.
  + Determines availability of service providers.
  + Creates a new assistance\_request record.
  + Could use WebSockets (e.g., Flask-SocketIO) to push the request to relevant service providers' dashboards in real-time.



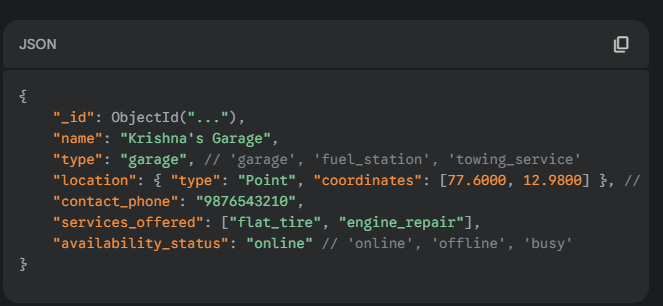
*Explanation:* The backend receives the request, performs a geospatial query on the service\_providers collection to find relevant providers within a certain radius. It then creates an assistance\_request document and potentially notifies service providers in real-time.

 **Database (MongoDB):**

* **assistance\_requests collection:**



service\_providers collection:

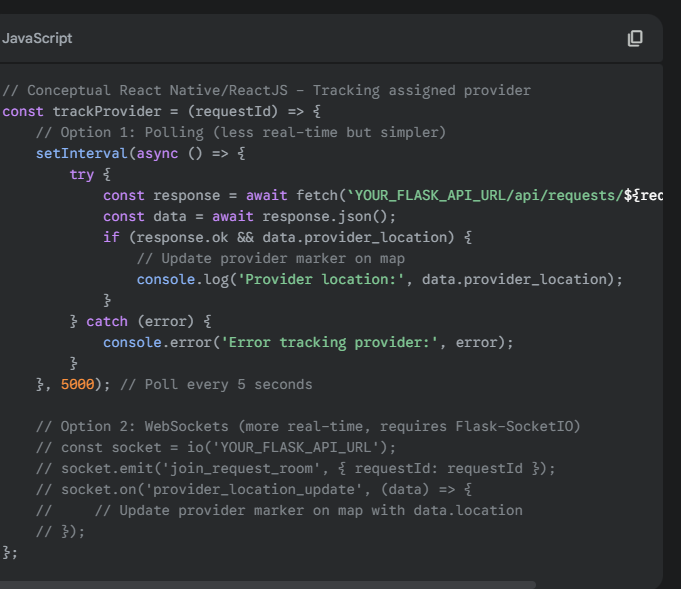


* *Explanation:* MongoDB would store user assistance requests and details of service providers, including their location in a GeoJSON format for efficient geospatial querying.

**3. Service Tracking & Navigation**

Once a service provider is assigned, the user needs to track their arrival.

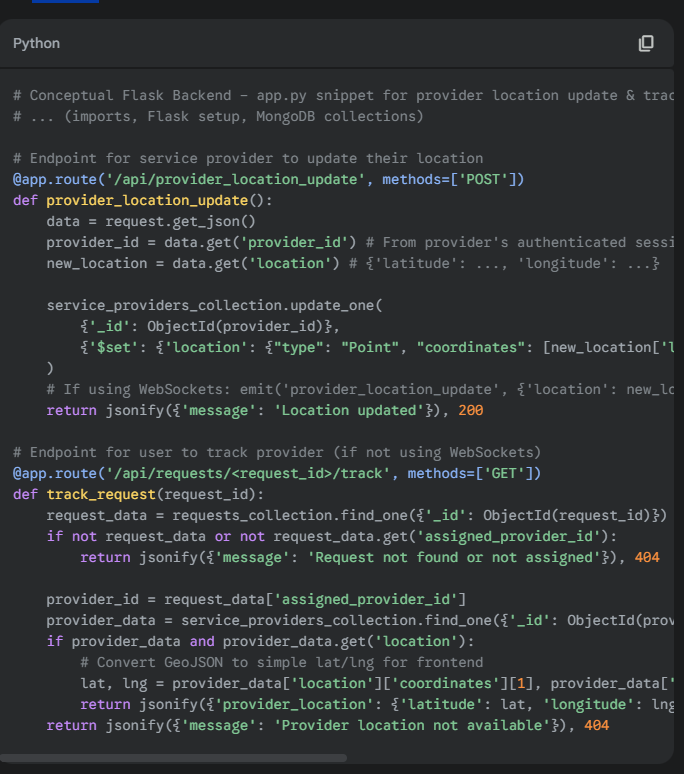
* **Frontend (React Native/ReactJS):**
  + **Real-time Updates:** Continuously polls the backend (or subscribes via WebSockets) for the assigned service provider's live location.
  + **Map Display:** Updates the service provider's marker on the map as their location changes.
  + **Navigation Display:** Potentially shows the route from the service provider to the user using Google Maps directions.



*Explanation:* The frontend periodically fetches or receives real-time updates for the service provider's location and updates the map accordingly.

**Backend (Python Flask):**

* **Provider Location Update:** A separate API endpoint for service providers to send their current location updates.
* **Tracking Endpoint:** GET /api/requests/<request\_id>/track or a WebSocket event listener.
* **Data Retrieval:** Fetches the assigned service provider's latest location from the service\_providers collection (which they would be updating).
* **Pushing Updates:** If using WebSockets, pushes location updates to the relevant user's frontend.



 *Explanation:* Service providers send their live location updates to an endpoint, which updates their document in MongoDB. The user tracking endpoint (or WebSocket) retrieves this updated location.

 **Database (MongoDB):**

* **service\_providers collection (updated):**

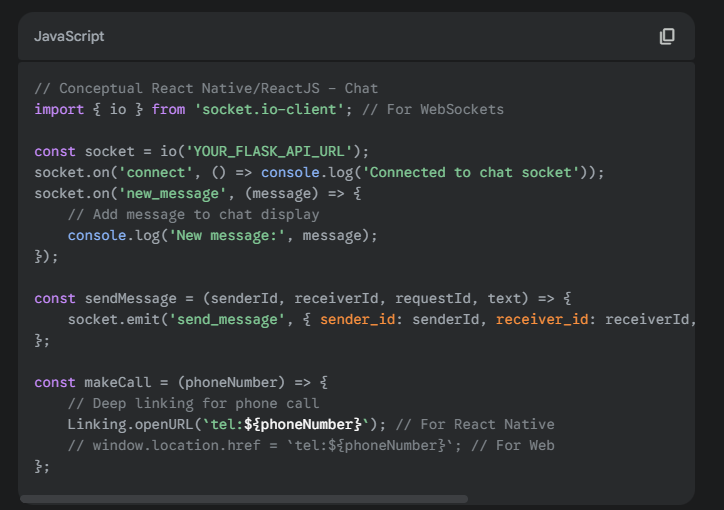


* *Explanation:* The service\_providers collection is dynamically updated with the latest coordinates of the active providers.

**4. Communication Features (Chat / Calling)**

Seamless communication between users and providers is critical.

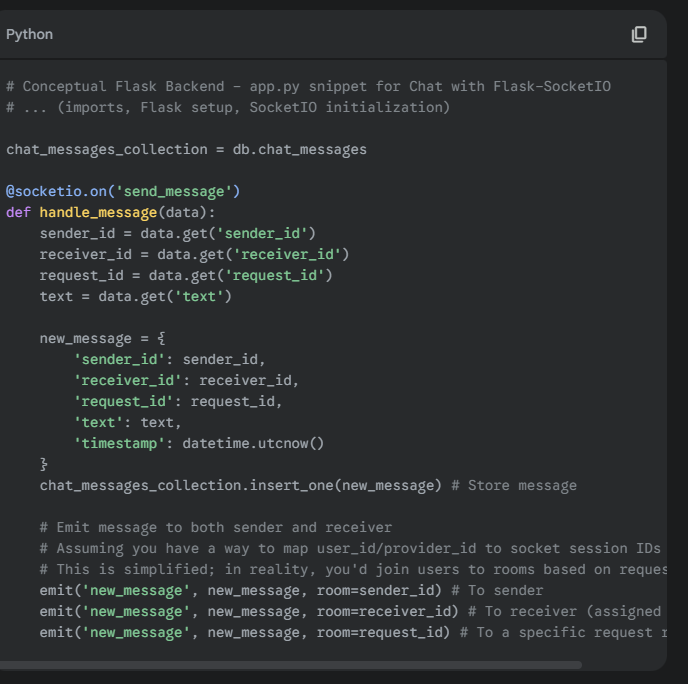
* **Frontend (React Native/ReactJS):**
  + **Chat UI:** Integrates a chat interface (e.g., using a library or custom components).
  + **Calling Integration:** Uses deep linking for phone calls or WebRTC for in-app calls.
  + **Sending Messages:** Dispatches messages to the backend.
  + **Real-time Message Display:** Renders incoming messages in real-time.



 *Explanation:* WebSockets are used for real-time chat, allowing instant message exchange. Deep linking is a straightforward way to enable phone calls.

 **Backend (Python Flask with WebSockets - Flask-SocketIO):**

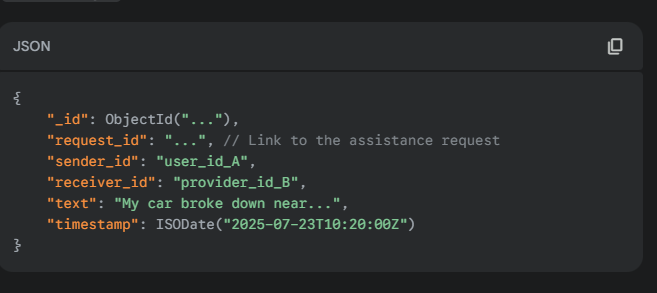
* **WebSocket Events:** Defines events like send\_message, new\_message.
* **Message Storage:** Saves chat messages to **MongoDB** (e.g., in a chat\_messages collection, linked to assistance\_requests).
* **Message Broadcasting:** Broadcasts messages to the specific users involved in a conversation (user and assigned provider).



 *Explanation:* Flask-SocketIO handles WebSocket connections. Incoming messages are stored in MongoDB and then broadcasted to the relevant participants in the conversation, allowing real-time chat.

 **Database (MongoDB):**

* **chat\_messages collection:**



* *Explanation:* All chat messages for each assistance request would be stored in a dedicated MongoDB collection.

**5. Locating Nearby Services (Garages, Fuel Stations, Towing Services)**

This relies heavily on the service\_providers data and Google Maps.

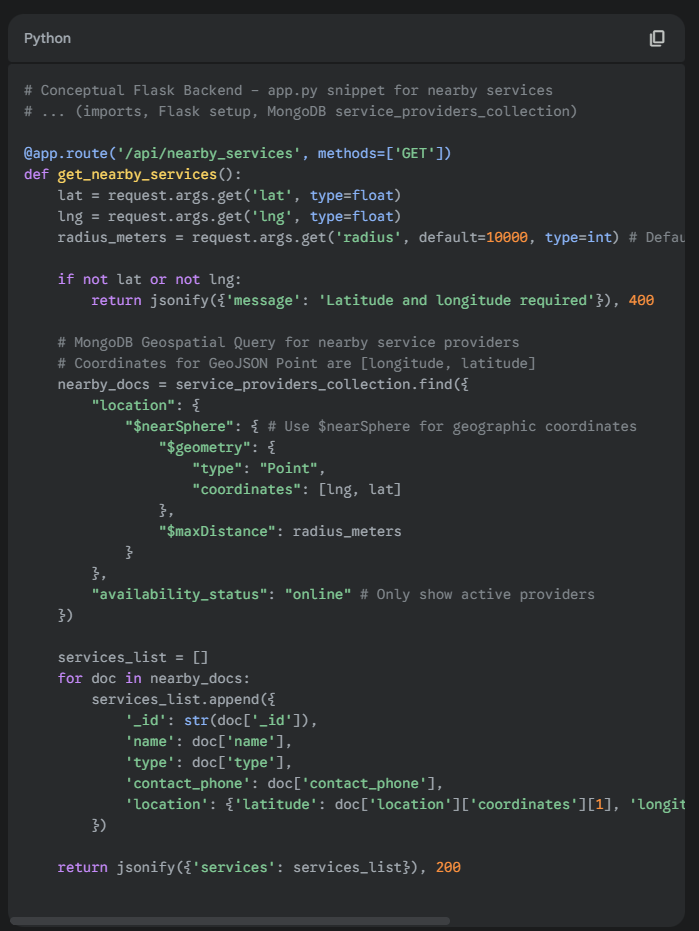
* **Frontend (React Native/ReactJS):**
  + **Map Rendering:** Displays an interactive map.
  + **POI Markers:** Plots markers for nearby service providers retrieved from the backend.
  + **Filtering:** Allows users to filter by service type (e.g., "only show garages").
  + **Interaction:** Clicking a marker shows details and options to contact (call, get directions).
  + **Google Maps API Integration:**
    - Displays map tiles.
    - Renders custom markers.
    - Potentially uses Google Places API (via backend proxy) for general points of interest or relies on pre-populated database.



 *Explanation:* The frontend fetches nearby service providers from the backend, then renders them as markers on the map using a map library or Google Maps API.

 **Backend (Python Flask):**

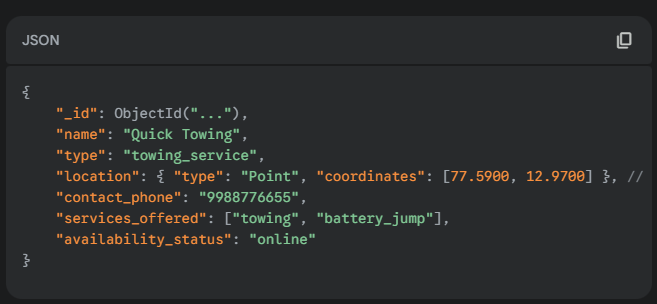
* **Route:** GET /api/nearby\_services?lat=<latitude>&lng=<longitude>&radius=<radius>
* **Data Retrieval:** Performs geospatial queries on the service\_providers collection (MongoDB) to find all registered service providers within a given radius of the user's location.
* **Filtering/Categorization:** Filters by type (garage, fuel station, towing service) if requested by the frontend.
* **Google Places API (Optional/Alternative):** If not relying solely on pre-registered providers, the backend could act as a proxy to Google Places API to search for commercial businesses of specific types (garages, petrol pumps, etc.) around the given coordinates and return them.



 *Explanation:* The backend receives the user's location and radius. It then performs a geospatial query (e.g., $nearSphere in MongoDB) on the service\_providers collection to find and return nearby registered services.

 **Database (MongoDB):**

* **service\_providers collection (GeoJSON for location):**



* *Explanation:* The service\_providers collection is critical, storing the exact location of each service provider in a GeoJSON Point format, enabling efficient spatial queries.

This detailed conceptual breakdown illustrates the developer's perspective on implementing the features of the Real-Time Roadside Assistance Website, utilizing the specified technologies and demonstrating the typical flow of data and logic.