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:

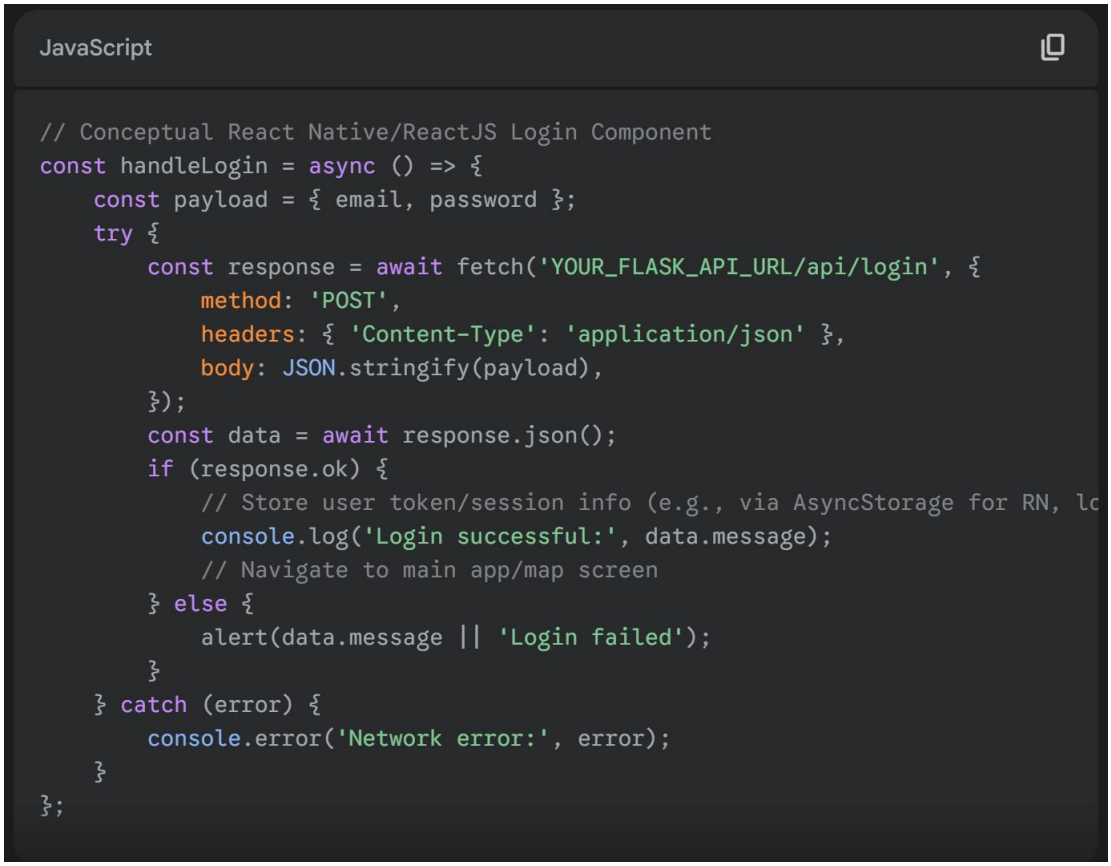
Real-Time Roadside Assistance Website - Conceptual End-to-End Flow

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.

Python

```python
# Conceptual Flask Backend - app.py snippet for login
from flask import Flask, request, jsonify
from werkzeug.security import generate_password_hash, check_password_hash
from pymongo import MongoClient # Assuming MongoDB

app = Flask(__name__)
client = MongoClient('mongodb://localhost:27017/') # MongoDB connection
db = client.roadside_assistance_db
users_collection = db.users # MongoDB collection for users

@app.route('/api/login', methods=['POST'])
def login():
    data = request.get_json()
    email = data.get('email')
    password = data.get('password')

    if not email or not password:
        return jsonify({'message': 'Email and password required'}), 400

    user_data = users_collection.find_one({'email': email}) # Fetch user from M
    if user_data and check_password_hash(user_data['password_hash'], password):
        # In a real app: Generate JWT or set session cookie
        return jsonify({'message': 'Login successful', 'user_id': str(user_data
    else:
        return jsonify({'message': 'Invalid credentials'}), 401

@app.route('/api/signup', methods=['POST'])
def signup():
    data = request.get_json()
    email = data.get('email')
    password = data.get('password')

    if users_collection.find_one({'email': email}):
        return jsonify({'message': 'User already exists'}), 409

    hashed_password = generate_password_hash(password)
    new_user = {'email': email, 'password_hash': hashed_password}
    result = users_collection.insert_one(new_user) # Store user in MongoDB

    return jsonify({'message': 'User created', 'user_id': str(result.inserted_i
```

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):**

```JSON
// users collection document structure
{
    "_id": ObjectId("..."),
    "email": "user@example.com",
    "password_hash": "pbkdf2:sha256:...", // hashed password
    "created_at": ISODate("2025-07-23T10:00:00Z"),
    "is_service_provider": false // or true, for provider accounts
}
```

- *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.

JavaScript

```javascript
// Conceptual React Native/ReactJS - Get location and send request
import Geolocation from '@react-native-community/geolocation'; // or browser Ge

const requestAssistance = async (issueType, vehicleType) => {
    Geolocation.getCurrentPosition(
        async (position) => {
            const { latitude, longitude } = position.coords; // How data is tak
            const payload = {
                user_id: 'current_user_id', // From authenticated session
                location: { latitude, longitude },
                issue_type: issueType,
                vehicle_type: vehicleType,
                timestamp: new Date().toISOString()
            };
            try {
                const response = await fetch('YOUR_FLASK_API_URL/api/requests',
                    method: 'POST',
                    headers: { 'Content-Type': 'application/json' },
                    body: JSON.stringify(payload),
                });
                const data = await response.json();
                if (response.ok) {
                    console.log('Assistance requested:', data.message);
                    // Display service tracking on map
                } else {
                    alert(data.message || 'Failed to request assistance');
                }
            } catch (error) {
                console.error('Network error requesting assistance:', error);
            }
        },
        (error) => console.error(error.message),
        { enableHighAccuracy: true, timeout: 15000, maximumAge: 10000 }
    );
};
```

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.

Python

```python
# Conceptual Flask Backend - app.py snippet for assistance request
# ... (imports for Flask, MongoDB, etc.)
# from flask_socketio import SocketIO, emit # If using WebSockets

# socketio = SocketIO(app)

service_providers_collection = db.service_providers
requests_collection = db.assistance_requests

@app.route('/api/requests', methods=['POST'])
def create_assistance_request():
    data = request.get_json()
    user_id = data.get('user_id') # From validated token/session
    location = data.get('location') # {'latitude': ..., 'longitude': ...}
    issue_type = data.get('issue_type')
    vehicle_type = data.get('vehicle_type')

    # Basic validation
    if not all([user_id, location, issue_type, vehicle_type]):
        return jsonify({'message': 'Missing request data'}), 400

    # Geospatial Query: Find nearby service providers
    # This is a conceptual query, actual implementation depends on MongoDB geos
    nearby_providers = service_providers_collection.find({
        "location": {
            "$near": {
                "$geometry": {
                    "type": "Point",
                    "coordinates": [location['longitude'], location['latitude']
                },
                "$maxDistance": 5000 # 5 km radius, for example
            }
        },
        "services_offered": issue_type # E.g., ['flat_tire', 'fuel_delivery']
        # And also check for provider availability status
    }).limit(5) # Get top 5 nearby

    provider_ids = [str(p['_id']) for p in nearby_providers]
    if not provider_ids:
        return jsonify({'message': 'No service providers found nearby for this

    new_request = {
        'user_id': user_id,
        'user_location': location,
        'issue_type': issue_type,
        'vehicle_type': vehicle_type,
        'status': 'pending', # 'pending', 'assigned', 'en_route', 'completed',
        'assigned_provider_id': None,
        'nearby_providers': provider_ids, # For potential manual assignment or
        'created_at': datetime.utcnow()
    }
    result = requests_collection.insert_one(new_request) # Store request in Mor

    # Conceptual: Push notification/WebSocket event to relevant providers
    # for provider_id in provider_ids:
    #     socketio.emit('new_request', {'request_id': str(result.inserted_id),

    return jsonify({'message': 'Assistance request created', 'request_id': str(
```

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:**

```
JSON
{
    "_id": ObjectId("..."),
    "user_id": "...", // Reference to user
    "user_location": { "latitude": 12.9716, "longitude": 77.5946 },
    "issue_type": "flat_tire",
    "vehicle_type": "4-wheeler",
    "status": "pending",
    "assigned_provider_id": null, // Will be ObjectId when assigned
    "nearby_providers": ["provider_id_1", "provider_id_2"],
    "created_at": ISODate("2025-07-23T10:05:00Z"),
    "last_updated": ISODate("2025-07-23T10:05:00Z")
}
```

service_providers collection:

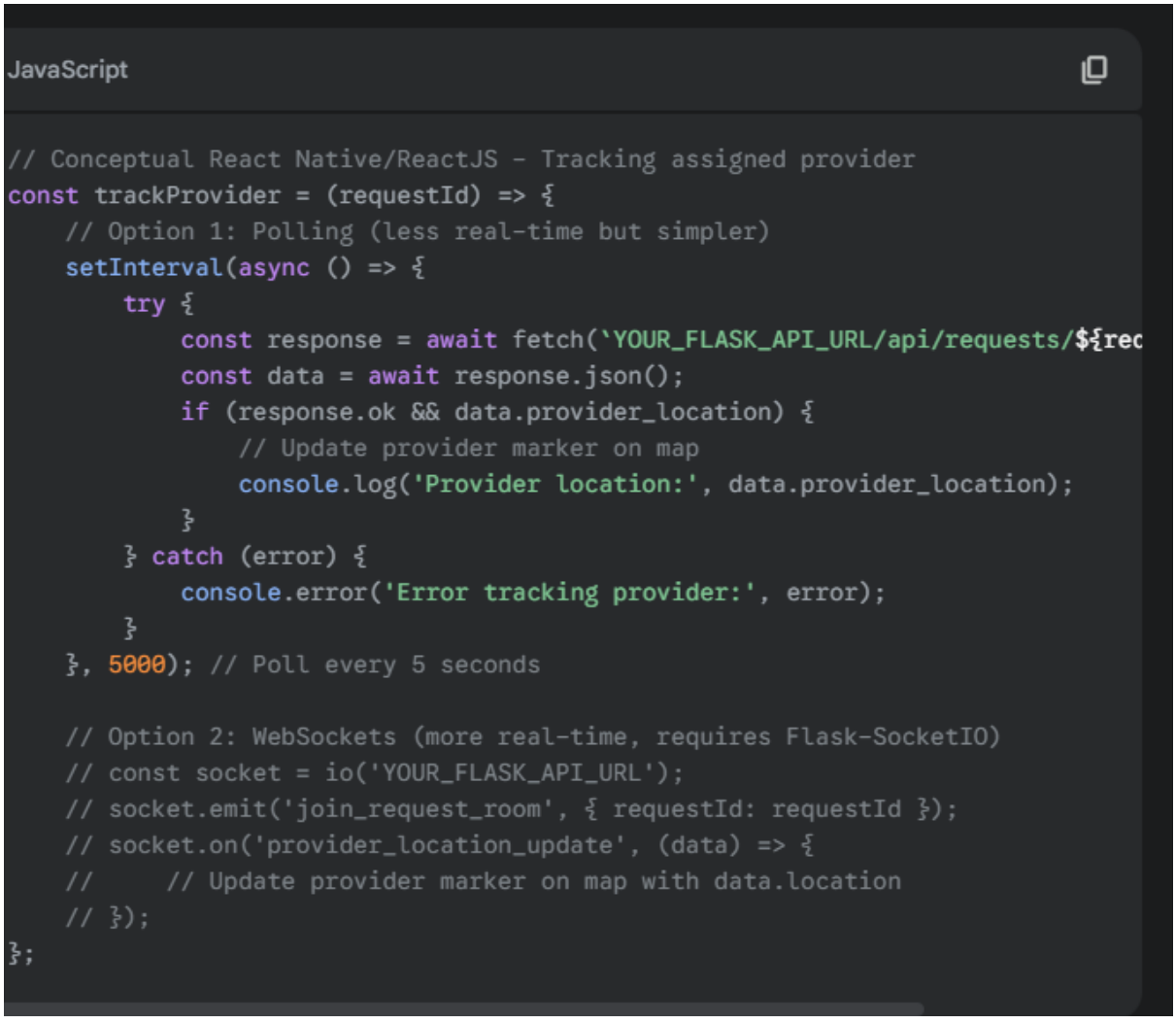
```
JSON
{
    "_id": ObjectId("..."),
    "name": "Krishna's Garage",
    "type": "garage", // 'garage', 'fuel_station', 'towing_service'
    "location": { "type": "Point", "coordinates": [77.6000, 12.9800] }, //
    "contact_phone": "9876543210",
    "services_offered": ["flat_tire", "engine_repair"],
    "availability_status": "online" // 'online', 'offline', 'busy'
}
```

- *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.

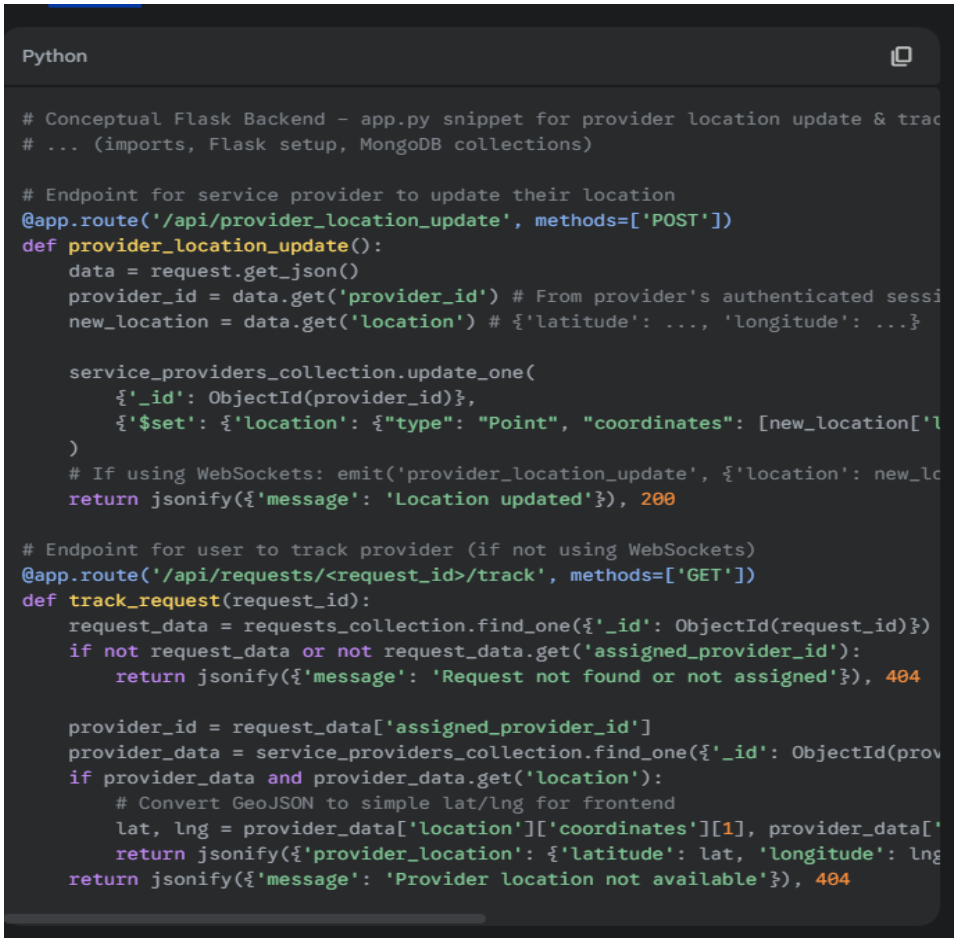
- **Frontend (React Native/ReactJS):**
    - **Real-time Updates:** Continuously polls the backend (or subscribes via WebSockets) for the assigned service provider's live location.
    - **Map Display:** Updates the service provider's marker on the map as their location changes.
    - **Navigation Display:** 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):

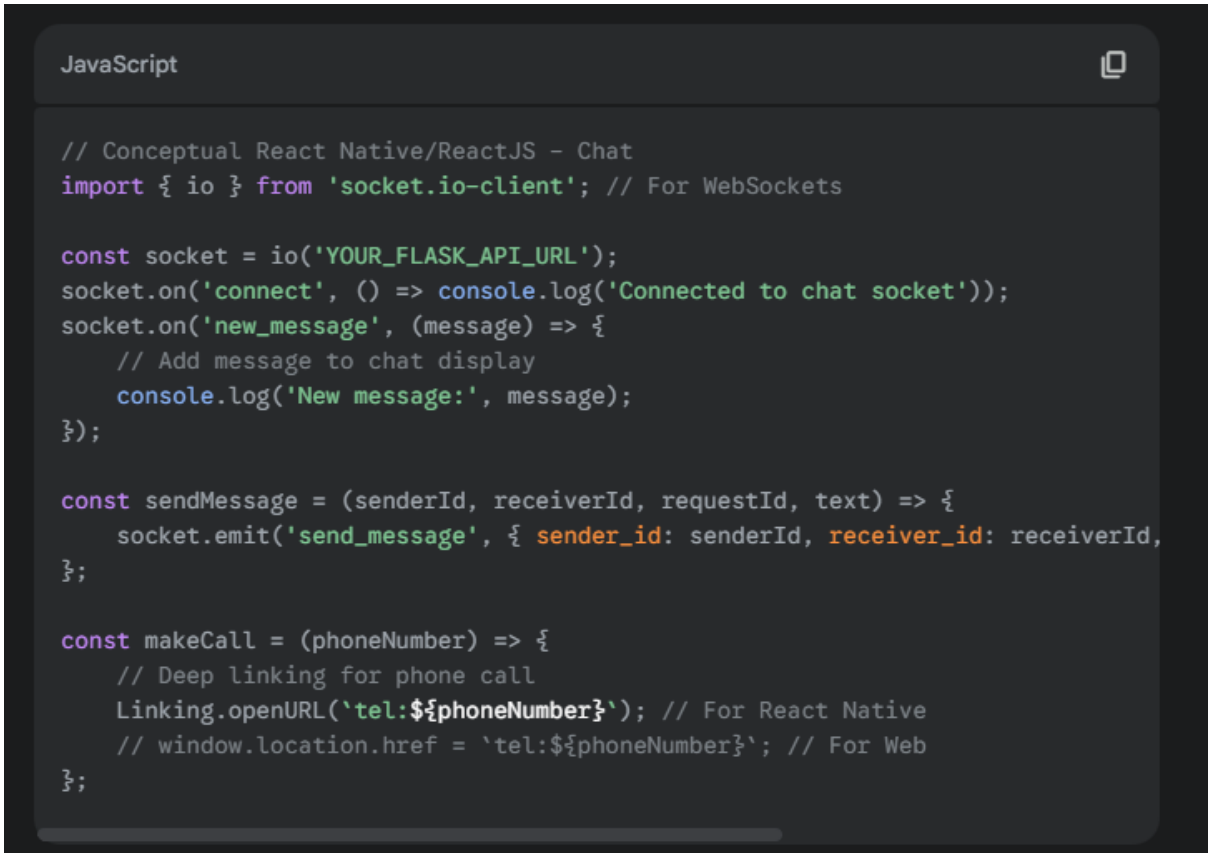
- **service_providers collection (updated):**

- *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).

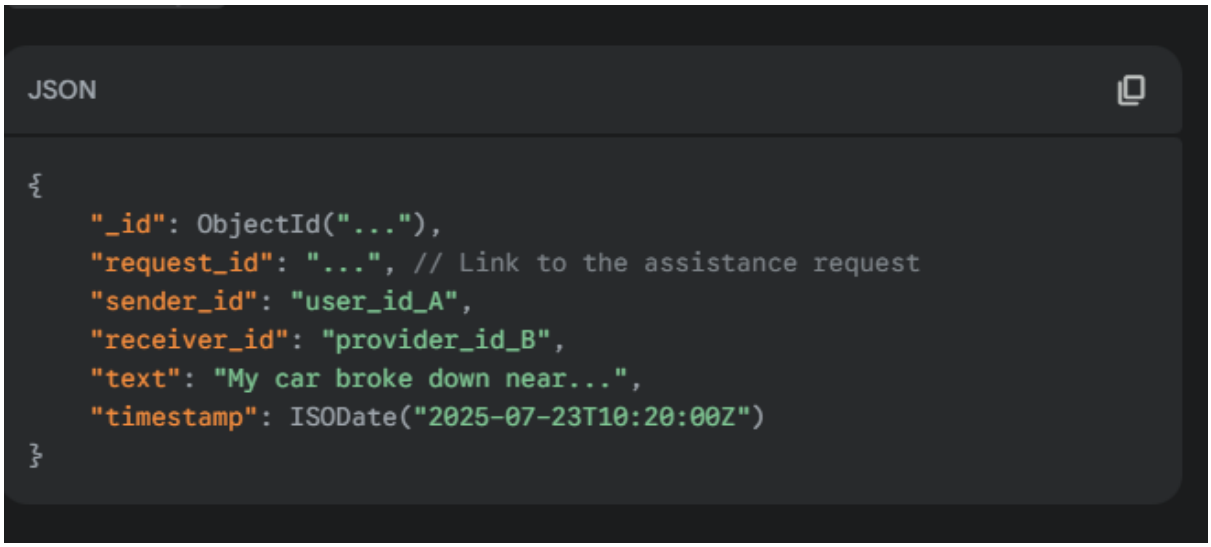
Python

```python
# Conceptual Flask Backend - app.py snippet for Chat with Flask-SocketIO
# ... (imports, Flask setup, SocketIO initialization)

chat_messages_collection = db.chat_messages

@socketio.on('send_message')
def handle_message(data):
    sender_id = data.get('sender_id')
    receiver_id = data.get('receiver_id')
    request_id = data.get('request_id')
    text = data.get('text')

    new_message = {
        'sender_id': sender_id,
        'receiver_id': receiver_id,
        'request_id': request_id,
        'text': text,
        'timestamp': datetime.utcnow()
    }
    chat_messages_collection.insert_one(new_message) # Store message

    # Emit message to both sender and receiver
    # Assuming you have a way to map user_id/provider_id to socket session IDs
    # This is simplified; in reality, you'd join users to rooms based on reques
    emit('new_message', new_message, room=sender_id) # To sender
    emit('new_message', new_message, room=receiver_id) # To receiver (assigned
    emit('new_message', new_message, room=request_id) # To a specific request r
```

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.

JavaScript

```javascript
// Conceptual React Native/ReactJS - Displaying nearby services
import MapView, { Marker } from 'react-native-maps'; // Or Google Maps JS API f

const [nearbyServices, setNearbyServices] = useState([]);
const userLocation = { latitude: 12.9716, longitude: 77.5946 }; // From geoloca

useEffect(() => {
    const fetchNearbyServices = async () => {
        try {
            // Fetch from your backend, which in turn queries MongoDB / Google
            const response = await fetch(`YOUR_FLASK_API_URL/api/nearby_service
            const data = await response.json();
            if (response.ok) {
                setNearbyServices(data.services);
            }
        } catch (error) {
            console.error('Error fetching nearby services:', error);
        }
    };
    fetchNearbyServices();
}, [userLocation]);

// ... inside render/return method
<MapView
    initialRegion={{
        latitude: userLocation.latitude,
        longitude: userLocation.longitude,
        latitudeDelta: 0.0922,
        longitudeDelta: 0.0421,
    }}
>
    {nearbyServices.map(service => (
        <Marker
            key={service._id}
            coordinate={{ latitude: service.location.latitude, longitude: servi
            title={service.name}
            description={service.type}
            onPress={() => console.log('Service selected:', service.name)}
        />
    ))}
</MapView>
```

 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.

Python

```python
# Conceptual Flask Backend - app.py snippet for nearby services
# ... (imports, Flask setup, MongoDB service_providers_collection)

@app.route('/api/nearby_services', methods=['GET'])
def get_nearby_services():
    lat = request.args.get('lat', type=float)
    lng = request.args.get('lng', type=float)
    radius_meters = request.args.get('radius', default=10000, type=int) # Defau

    if not lat or not lng:
        return jsonify({'message': 'Latitude and longitude required'}), 400

    # MongoDB Geospatial Query for nearby service providers
    # Coordinates for GeoJSON Point are [longitude, latitude]
    nearby_docs = service_providers_collection.find({
        "location": {
            "$nearSphere": { # Use $nearSphere for geographic coordinates
                "$geometry": {
                    "type": "Point",
                    "coordinates": [lng, lat]
                },
                "$maxDistance": radius_meters
            }
        },
        "availability_status": "online" # Only show active providers
    })

    services_list = []
    for doc in nearby_docs:
        services_list.append({
            '_id': str(doc['_id']),
            'name': doc['name'],
            'type': doc['type'],
            'contact_phone': doc['contact_phone'],
            'location': {'latitude': doc['location']['coordinates'][1], 'longit
        })

    return jsonify({'services': services_list}), 200
```

 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.