Project Goal

Building a production-ready web application where users can select two points on a map, visualize the route, simulate vehicle movement at custom speeds with ETA calculations, and later as future task track real-time location from mobile devices with live ETA.

Core Features

Phase 1: Static Route Planning

- Display interactive map where user can select starting and destination by searching or can click to set pickup and drop-off points, show them as markers.
- Calculate and display the shortest route between two points.
- Show total distance in km/meters.
- Allow user to input custom vehicle speed.
- Calculate estimated travel time based on distance and speed.
- Highlight the route on the map clearly.

Phase 2: Route Simulation

- Animate a vehicle marker moving along the calculated route.
- Control/replicate animation speed based on user-input speed.
- Provide play/pause/reset/fast forward animation speed controls.
- Show real-time progress (distance covered, time elapsed, ETA based on current position and speed).

Phase 3: Real-Time Location Tracking (Future)

- Capture GPS location from mobile device.
- Stream location data to the web application.
- Display live position on the map.
- Calculate current speed from GPS data.
- Show dynamic ETA based on current position and speed.
- Handle connection drops and reconnections gracefully.

Technology Decisions

1. Map Rendering

I will use: Leaflet.js

Why:

- Open-source, zero cost
- Lightweight
- Works perfectly with OpenStreetMap tiles

Alternatives considered:

- ❌ Google Maps API (costs money after free tier)

2. Map Tiles Provider

I will use: OpenStreetMap (OSM) tiles

Why:

- Completely free with fair use policy

For future optimization: If I need faster loading or custom styling, I can switch to Mapbox/Maptiler free tier or self-host tiles using Tileserver GL.

Transition difficulty level: Easy- just change the tile URL

2.1 Geocoding Service

I will use: Nominatim (OpenStreetMap's geocoding API)
Why:

- Free and open-source
- Works with OSM data
- Can be self-hosted or use public API (1 req/sec limit)

Returns: latitude, longitude, display name

I will self-host Nominatim in Docker.

3. Routing Engine

I will use: Self-hosted OSRM in Docker

Why:

- Unlimited API calls
- Response time: 10-50ms (vs 200-500ms from external APIs)
- Full control over uptime and performance
- Bangladesh OSM data is ~200MB - very manageable

Alternatives considered:

- ❌ OSRM Demo Server (unreliable, will fail in production)
- ❌ GraphHopper & OpenRouteService (Free Tier) (15k requests/day = ~10 active users, not scalable)

4. Distance & Time Calculations

I will use: Manual calculation with linear interpolation

Why:

- No dependencies
- For distances < 100km (typical city routes), linear interpolation error is < 0.1%

formula:

```
time_seconds = distance_meters / (speed_kmh / 3.6)

# Linear interpolation for animation
interpolated_lat = start_lat + (end_lat - start_lat) * fraction
interpolated_lng = start_lng + (end_lng - start_lng) * fraction
```

For future: If I need to expand intercity routes (>100km), I will need to shift to Turf.js for Great Circle distance calculations.

Transition difficulty level: Medium - requires refactoring interpolation logic, but APIs are similar

5. Frontend-Backend Frameworks: Vanilla JavaScript & Node.js

I will use: Vanilla JavaScript (no framework)
Why:

- Direct DOM manipulation with Leaflet
- Simple & Clean

I will use: Node.js with Express.js

Why:

- built for event-driven & real-time applications (Massive ecosystem)
- Single-threaded event loop handles thousands of concurrent WebSocket connections efficiently
- Lower memory footprint (50-100MB vs 200-300MB for Django)
- Faster response times for I/O operations (routing calculations, database queries)

Difficulty level: Medium

6. Real-Time Communication: WebSockets

I will use: Socket.io

Why:

- Auto-reconnection logic built-in
- Better Scalability
- Can handle thousands of concurrent connections on a minimal server
- Battle-tested (Uber, Slack)

Difficulty level: Medium

6.1 Authentication (for Later)

I will use: JWT (JSON Web Tokens)

Why:

- Works perfectly with REST APIs and WebSockets
- Industry standard

Libraries:

- jsonwebtoken (JWT generation/verification)

- bcrypt (password hashing)

Difficulty level: Medium

7. Database (for later)

I will use: PostgreSQL with PostGIS extension

Why:

- Free and open-source
- PostGIS adds geospatial capabilities (distance calculations, spatial indexing)
- Can query things like "find all routes within 5km radius"
- Industry-standard for location-based apps
- Docker-friendly

What I'll store:

- User accounts
- Route history
- Location snapshots (for analytics)
- Vehicle information

My choice: I'll use **Sequelize** because it's more mature and has better PostGIS integration.

Alternatives I considered:

- ❌ SQLite (no PostGIS support, not production-ready for multi-user)
- ❌ MongoDB (overkill, geospatial queries are harder)

Difficulty level: Medium

8. Caching Layer (for Later)

I will use: Redis

Why:

- Store current vehicle locations (O(1) lookup speed)
- Cache routing calculations
- Pub/Sub for broadcasting location updates to multiple clients

- Handles 100k+ operations/second

For real-time state: I'll use **Redis** as an in-memory cache for current locations (ultra-fast lookups).

Transition difficulty level: Easy

9. Deployment & Hosting (for Later)

I will use: Docker Compose for everything

Why:

- Can run on any cloud provider

Hosting options:

Free tier (for testing):

- Render.com (750 hrs/month free, WebSocket support)

Paid (for production):

- DigitalOcean App Platform
- AWS Lightsail

Difficulty level: Medium

10. Mobile GPS Capture (for Later)

I will use: HTML5 Geolocation API in mobile browser

Why:

- No app installation required
- Works in any modern mobile browser
- Simple API

I'll implement adaptive polling (update every 5 seconds when stationary, every 1 second when moving).

Difficulty level: Easy

11. Location Smoothing- Jitter Handling (for Later)

I will use: Simple moving average filter

Why:

- GPS data jumps around (±5-10 meters)
- Raw data makes the vehicle marker "teleport" erratically
- Smoothing creates fluid movement

For future: If I need more accuracy, I can implement a Kalman filter.

Transition difficulty level: Medium - Kalman filters are mathematically complex

12. Animation Engine

I will use: vanilla JS: `requestAnimationFrame` for smooth 60fps animation

Why:

- Browser-optimized, syncs with screen refresh
- Automatically pauses when tab is inactive

My Stack:

Both OSRM and Nominatim will run in my Docker Compose.

javascript

```
// Core framework
Express.js        // HTTP server & REST API
Socket.io         // WebSocket server (real-time communication)
Sequelize     // ORM for PostgreSQL
```

// Additional libraries

// Core framework

Express.js // HTTP server & REST API

Socket.io // WebSocket server

Sequelize // ORM for PostgreSQL

// Database & Caching

node-postgres (pg) // PostgreSQL driver

ioredis // Redis client

// Security & Validation

helmet // Security headers

cors // CORS handling

express-validator // Input validation

express-rate-limit // Rate limiting

// Utilities

dotenv // Environment variables

axios // HTTP client (for OSRM calls)

winston or pino // Logging

// Development

nodemon // Auto-restart during dev