🧪 Fullstack Coding Assignment (Senior)

🚘 Project Title:

Connected Fleet Health & Diagnostics Console

📘 Scenario

You are part of a platform team building a **fleet-wide diagnostics console** for connected vehicles.

The console is used by operations engineers to:

- Monitor the **health of many vehicles** in near-real time
- Investigate and search **diagnostic events and error patterns**
- Get a quick overview of **critical issues** and affected vehicles
- Explore data in different ways (by vehicle, by error code, by time window, etc.)

You will build a **minimal but realistic fullstack application** that supports this workflow.

🎯 Objective

Build a system that:

1. Ingests and processes **vehicle diagnostic events** (from a log file and/or simulated stream)
2. Stores events in a way that allows flexible querying and aggregation
3. Exposes a **REST API** to:
    - Query **raw events** with multiple filters
    - Retrieve **aggregated views** (e.g., errors per vehicle, per code, per time range)
4. Provides a **frontend dashboard** that:
    - Uses **observable-based patterns** (RxJS / Signals) for data and state
    - Supports **search, filtering, and live updates**
    - Offers at least **two different views** on the same data (e.g., table + summary/metrics)

You are free to make assumptions and design choices. Please document them.

🗃️ Input Data (Example)

Diagnostic events come in a structured log format, e.g.:

```
[2025-07-24 14:21:08] [VEHICLE_ID:1234] [ERROR] [CODE:U0420] [Steering angle sensor malfunction]
[2025-07-24 14:21:10] [VEHICLE_ID:5678] [WARN] [CODE:P0300] [Random misfire detected]
[2025-07-24 14:22:05] [VEHICLE_ID:1234] [INFO] [CODE:EVT01] [Diagnostic check completed]
```

You can extend this format if you need to (e.g. add mileage, location, subsystem, etc.) and you can decide how events get into your system (upload, seed script, simple generator, etc.).

🗄️ Backend (Node.js + TypeScript)

Use **NestJS** or **Express.js**.

Expectations

- Parse the diagnostic event format into a structured model (timestamp, vehicle, level, code, message, etc.).
- Store data in **memory or a simple DB** (e.g., SQLite, lowdb, or similar).
- Design a **query API** that supports combinations of filters, for example:
  - `GET /events?vehicle=1234`
  - `GET /events?code=U0420&level=ERROR`
  - `GET /events?from=2025-07-24T14:00:00Z&to=2025-07-24T15:00:00Z`
  - Any **reasonable combination** of vehicle, code, severity, time range.
- Provide at least one **aggregated endpoint**, e.g.:
  - Errors per vehicle in a time range
  - Most frequent error codes
  - Vehicles currently in “critical” state (you decide what that means and document it)

Non-functional aspects

- Input validation and error handling (invalid params, bad date ranges, etc.).
- Clear separation of concerns (e.g., parsing, storage, domain logic, API layer).
- Basic API documentation (OpenAPI/Swagger or similar).

You are free to choose the exact shape of the API and the data model. Explain your choices.

🖥️ Frontend (Angular)

Use **Angular 15+**.

The frontend should consume your backend API and demonstrate **senior-level usage of observables and state management**.

Core features

- A **search / filter panel** with fields such as (you can extend/adapt):
  - Vehicle ID (single or multiple)
  - Error code
  - Severity/level
  - Time range
- A **results view** for raw events:
  - Typically a table or list with pagination or infinite scroll
  - Columns like timestamp, vehicle, code, severity, message
- At least one **additional view** of the same data, e.g.:
  - Aggregated summary (cards, charts, or grouped lists)
  - “Per vehicle” view
  - “Per error code” view

State & RxJS

Use **observable-based patterns** for state and async operations. For example:

- A dedicated service and/or store that exposes **observables** for:
  - Current filters
  - Loaded events
  - Aggregated data
  - Loading/error states
- Use operators like e.g. **map, switchMap, combineLatest, debounceTime, shareReplay** where appropriate.
- Handle **in-flight requests** correctly (e.g. when filters change quickly).
- Use either:
  - **NgRx**, or
  - **Signals with RxJS**, or
  - Another **structured state-management approach** built on observables.

Explain your chosen approach briefly in your concept.

UI/UX

- Reasonably clean layout and responsive behavior.
- Basic accessibility (semantic HTML, keyboard-friendly where it makes sense).
- Reusable components (e.g. search panel, table, filter controls).

You do **not** need to build a polished production UI, but we want to see thoughtful structure.

What We Expect From You

We are interested not only in the code, but also in how you **think** about the problem.

Please provide:

1. **Source Code**
   - Frontend and backend via Github link (e.g. in a single repo).
2. **Requirements Description**
   - Your own short list of **customer/business requirements** derived from the scenario.
   - Include any **assumptions** you made.
3. **Concept / Architecture Description**
   - A **backend concept** (how you structured the backend, main components, data model, key decisions).
   - A **frontend concept** (state management approach, main building blocks, data flow).
   - Format is up to you (short document, markdown, diagrams, etc.).
4. **README**
   - How to run backend and frontend.
   - Any prerequisites.
   - Brief overview of what works and what you would do next with more time.
5. **API Documentation**
   - Preferably OpenAPI/Swagger or similar, generated or written by you.
6. **(Optional) Containerization**
   - Dockerfile or docker-compose if you want to.

Good luck!