Simple Car Maintenance Tracker

Marsel Abdullin, Tiger Yang

Problem Description

- **Scattered Records:** Important vehicle data like service logs and fuel receipts are scattered, causing confusion and lost information.
- **Missed Maintenance:** Without easy tracking of service dates by mileage or time, crucial maintenance is often delayed or missed.
- **No Multi-Vehicle Management:** Managing service schedules and histories for several vehicles wastes time and adds complexity.
- **Poor Cost Tracking:** Lack of a unified system makes it hard to control and analyze spending across multiple vehicles.
- **No Performance Trends:** Vehicle efficiency and maintenance trends remain hidden, preventing informed decisions.

Intended Users

- **Individual Vehicle Owners:** Track maintenance, fuel costs, and service history
- **Small Business Fleet Managers:** Manage multiple vehicles from a centralized system
- **Car Enthusiasts:** Maintain detailed service logs and modification records

Application Overview

- What the Application Does
  - Car Tracker - Web-based vehicle management system that centralizes all vehicle data
  - Provides unified platform for tracking maintenance, expenses, and service history
- Main Features
  - **Vehicle Management** - Register vehicles, track ownership, and manage multi-vehicle fleets
  - **Service Tracking** - Record service history, manage parts, and schedule upcoming service
  - **Expense Management** - Track all vehicle costs with detailed categorization
  - **Analytics Dashboard** - Generate reports on costs, fuel consumption, and maintenance patterns
  - **Data Export** - Export your data about vehicles in the CSV file

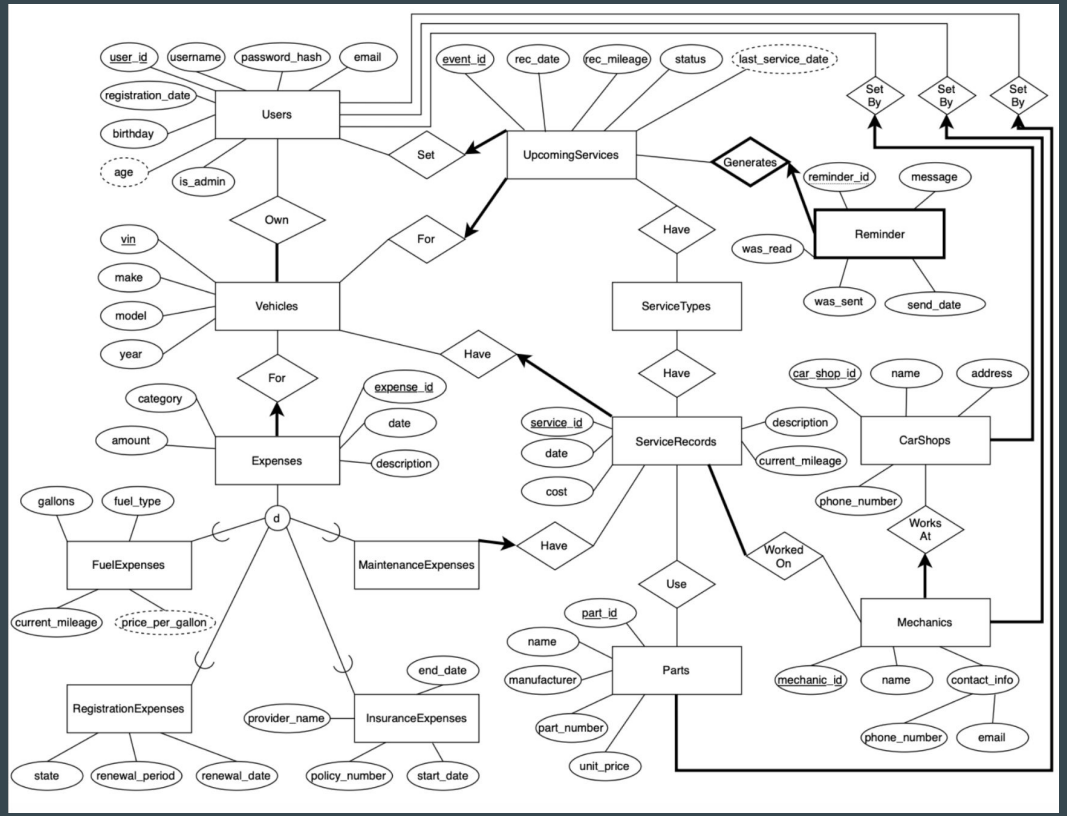
User Roles

- Regular User
  - Can only view and edit their own records
  - Full application functionality within own account
- Admin
  - Can view and edit records of all users
  - Can view and assign additional information to records
  - Can elevate users to admin permissions

System Structure

- 3 tier system
  - MySQL DB
  - Node JS Backend
  - JavaScript, HTML, CSS Frontend
- Users do not interact with DB directly
  - User input is handled through forms, fields, and buttons
  - Backend validates the input and queries the DB
  - DB returns desired data

Design and Notable Relationships



Expenses

- Expenses **For** Vehicles
  - 4 specific sub-Expense entities
  - Generic Expense

ServiceRecords

- ServiceRecords **Use** Parts
- Mechanics **WorkedOn** ServiceRecords
- ServiceRecords **Have** ServiceTypes

Parts, Mechanics, Carshops

- Parts **SetBy** User
- Mechanics **SetBy** User
- Carshops **SetBy** User

Schema and Normalization

Users(user_id: Integer, username: String, password_hash: String, email: String, birthday: Date, registration_date: Date, is_admin: Integer)

Vehicles(vin: String, make: String, model: String, year: Integer)

Owns(user_id: Integer, vin: String, start_date: Date, end_date: Date)

FK (user_id: Integer) references Users(user_id: Integer)

FK (vin: String) references Vehicles(vin: String)

CarShops(car_shop_id: Integer, name: String, address: String, phone_number: String, user_id: Integer)

FK (user_id: Integer) references Users(user_id: Integer)

Mechanics(mechanic_id: Integer, car_shop_id: Integer, name: String, phone_number: String, email: String, user_id: Integer)

FK (user_id: Integer) references Users(user_id: Integer)

FK (car_shop_id: Integer) references CarShops(car_shop_id: Integer)

ServiceRecords(service_id: Integer, vin: String, service_date: Date, current_mileage: Integer, cost: Float, description: String)

FK (vin: String) references Vehicles(vin: String)

WorkedOn(mechanic_id: Integer, service_id: Integer)

FK (mechanic_id: Integer) references Mechanics(mechanic_id: Integer)

FK (service_id: Integer) references ServiceRecords(service_id: Integer)

1st Normal

- Decompose to atomic attributes

2nd Normal

- Most primary keys for tables are auto incremented simple keys
- Composite keys only present in connect tables, so no partial dependency

3rd Normal

- No transitive dependencies

BNCF

- All attributes depend on superkeys
- Users has unique username and email
- Changed Mechanic email to non-unique

Indexing and Validation

```
"expenses_complex_join": """
    SELECT
        e.*,
        v.make, v.model, v.year,
        me.service_id,
        re.renewal_date, re.renewal_period, re.state,
        ie.policy_number, ie.start_date, ie.end_date, ie.provider_name,
        fe.gallons, fe.current_mileage, fe.fuel_type
    FROM Expenses e
    LEFT JOIN Vehicles v ON e.vin = v.vin
    LEFT JOIN MaintenanceExpenses me ON e.expense_id = me.expense_id
    LEFT JOIN RegistrationExpenses re ON e.expense_id = re.expense_id
    LEFT JOIN InsuranceExpenses ie ON e.expense_id = ie.expense_id
    LEFT JOIN FuelExpenses fe ON e.expense_id = fe.expense_id
    JOIN Owns o ON e.vin = o.vin
    ORDER BY e.date DESC
    LIMIT 100
""",
```

Query	Before (s)	After (s)
expenses_complex_join	0.010587	0.005100
fuel_analytics_groupby	0.002095	0.001368
service_cost_by_vehicle	0.003851	0.002884
service_count_by_month	0.002297	0.001573

Indexing

- Performed with 5000 rows in each table
- Halved processing time for the most expensive operation
- Smaller improvements in simpler operations

Validation

- Forms limit visibility of certain fields (based on regular user or admin)
- Regex validation
- SQL validation using LIKE
- Required fields clearly marked
- Ensures required fields are filled out before form submission

Challenges and Solution

- Frontend
  - **Admin UI Separation Challenge:** Required complex role-based rendering was solved by implementing a centralized AuthManager for consistent permission checks across all components.
  - **Real-time Data Sync:** Race conditions from cascading data updates were solved by a targeted refresh system that uses sequential async/await calls to update only related sections.
- Backend
  - **Page Reload State:** The need for complex state persistence after a page reload was solved by creating session management using sessionStorage, localStorage, and database validation.
- Data Base
  - **Stability on Large Datasets:** Server crashes caused by fetching large datasets were solved by implementing server-side pagination, ensuring data is always fetched in smaller, manageable pages to prevent memory overload.

What's Next

- **AWS EC2 Deployment:** Deploy to ensure scalable and worldwide access.
- **RESTful API Development:** enable seamless third-party integrations with insurance companies, service shops, and financial institutions.
- **AI Integration:** to provide fast, actionable insights from vehicle and maintenance data.

Demo

Show the working application in action
Demonstrate key features and workflows (e.g., data entry, queries, updates, search)
Highlight how the database supports these features

Thank you!