Final Proposal
Car Tracker

Tiger Yang
017267757
SJSU
San Jose, California
tiger.yang@sjsu.edu

Marsel Abdullin
017115072
SJSU
San Jose, California
marsel.abdullin@sjsu.edu

I. Project Goals and Description

A. Overview

Car Tracker is a centralized, user–friendly web platform that allows users to manage all aspects of vehicle upkeep in one place. It brings together maintenance records, expenses, and parts into a single, accessible system, making it easy to maintain a complete service history and plan future maintenance.

By combining organized record keeping with actionable insights, *Car Tracker* helps users stay on top of maintenance, prevent costly repairs, and make informed, data–driven decisions about their vehicles.

B. Real–World Problem Addressed

Vehicle information is often stored in many different places, such as emails, paper receipts, warranty documents, and personal notes. This lack of organization can result in missed maintenance, limited insight into operating costs, and difficulty identifying performance trends.

Car Tracker addresses these challenges by:

- **Vehicle Management:** Add, edit, and delete vehicles with validation to ensure accurate records.
- **Ownership Management:** Link users to specific vehicles and enforce user-scoped data access for privacy.
- **Service Records:** Track date, mileage, cost, and details of services; link multiple mechanics, parts, and service types; automatically create maintenance expenses when adding a service.
- **Mechanics & Shops:** Manage shop and mechanic profiles, and link service records directly to them.
- **Parts Catalog:** Maintain a list of parts and associate them with related services for accurate tracking.
- **Expenses Tracking:** Record maintenance, registration, insurance, and miscellaneous costs with category-specific fields.
- **Upcoming Services & Reminders:** Schedule future services by date or mileage and receive in-app notifications for pending, completed, or overdue tasks.
- **Analytics:** Generate reports on expenses by category, average service cost per vehicle, service counts by month, and overall cost summaries.
- **Search, Filters, & CSV Export:** Quickly find records using column filters and export expenses, analytics, and detailed data to CSV.

C. Intended Users and Roles

Primary User Groups:

- **Individual Vehicle Owners:** Track maintenance, monitor expenses, and maintain a complete history to preserve reliability and resale value.
- **Small Business Fleet Managers:** Manage multiple vehicles from a single interface, schedule services, and export expense reports for budgeting.
- **Car Enthusiasts:** Keep detailed logs of maintenance, performance upgrades, and cosmetic changes.

System Roles:

- **Regular User:** Add, edit, and view their own vehicles and related records; privacy enforced through SQL scoping.
- **Admin:** Full system visibility and control, including user management, role changes, database tools, and log access.

II. Functional Requirements and Application Architecture

A. Entity Features:

Users

- Regular users view their own profiles; admins manage all users.
- Admins can add, edit, delete users, and change roles.
- Search and filter show expanded details (e.g., owned vehicles, upcoming services).

Vehicles & Ownership

- Add, edit, and delete vehicles with validation.
- Vehicle details include owners, services, and expenses.
- Ownership can be started, paused, or ended; status updates are reflected across sections.

Car Shops & Mechanics

- Manage shops and mechanics, linking them to services.
- View mechanics' work history with search and filter options.

Service Records

- Track maintenance work linked to mechanics, parts, and service types.
- Automatically create related maintenance expense entries.

Parts

- Manage parts and link them to service records.
- Search, filter, and view details by user or admin scope.

Expenses

- Track fuel, insurance, registration, and miscellaneous costs.
- Maintenance expenses generated automatically from service records.
- Category-specific details and CSV export available.

Upcoming Services & Reminders

- Schedule maintenance by date or mileage.
- Send reminders with read/sent status and popup alerts.

Analytics

- Summaries for expenses, service costs, fuel use, and maintenance trends.
- CSV export for reports.

Admin Tools

- Database health checks, raw SQL runner, and data reset options.
- Error logging and recovery features.

B. Application Architecture

Frontend

- Single-page web app with Bootstrap.
- Sections for managing users, vehicles, shops, services, expenses, and reports.
- Role-based access for admins and regular users.

Backend

- Node.js and Express handle database queries, health checks, and error logs.

Database

- MySQL schema with linked tables for users, vehicles, services, and expenses.
- Foreign keys and indexing ensure data integrity and speed.

III. ER Data Model

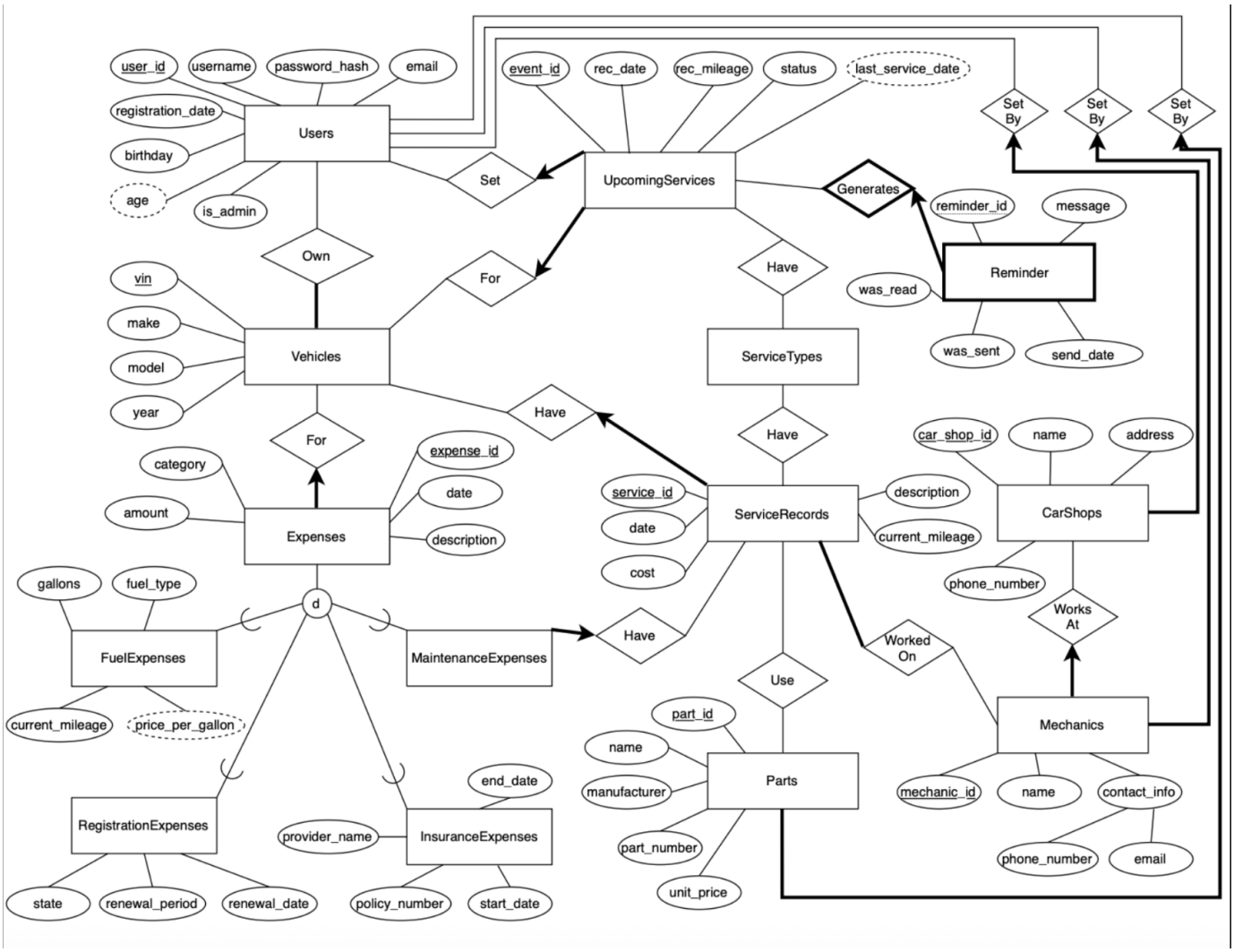


Fig. 1: Entity–Relationship Data Model.

IV. Database Design, Normalization, and Indexing

A. Database Relational Schema

- 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, street: String, city: String, state: String, zip_code: 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)
- ServiceTypes(service_type: String)
- ServiceRecords_ServiceTypes(service_id: Integer, service_type: String)
  - FK (service_id: Integer) references ServiceRecords(service_id: Integer)
  - FK (service_type: String) references ServiceTypes(service_type: String)
- Parts(part_id: Integer, name: String, manufacturer: String, part_number: String, unit_price: Float, user_id: Integer)
  - FK (user_id: Integer) references Users(user_id: Integer)
- ServiceRecords_Parts(service_id: Integer, part_id: Integer)
  - FK (service_id: Integer) references ServiceRecords(service_id: Integer)
  - FK (part_id: Integer) references Parts(part_id: Integer)
- Expenses(expense_id: Integer, vin: String, date: Date, category: Enum, amount: Float, description: String)
  - FK (vin: String) references Vehicles(vin: String)
- MaintenanceExpenses(expense_id: Integer, service_id: Integer)
  - FK (expense_id: Integer) references Expenses(expense_id: Integer)
  - FK (service_id: Integer) references ServiceRecords(service_id: Integer)
- RegistrationExpenses(expense_id: Integer, renewal_date: Date, renewal_period: String, state: String)
  - FK (expense_id: Integer) references Expenses(expense_id: Integer)
- InsuranceExpenses(expense_id: Integer, policy_number: String, start_date: Date, end_date: Date, provider_name: String)
  - FK (expense_id: Integer) references Expenses(expense_id: Integer)
- FuelExpenses(expense_id: Integer, gallons: Float, current_mileage: Integer, fuel_type: String)
  - FK (expense_id: Integer) references Expenses(expense_id: Integer)
- UpcomingServices(event_id: Integer, user_id: Integer, vin: String, rec_date: Date, rec_mileage: Integer, status: String)
  - FK (user_id: Integer) references Users(user_id: Integer)
  - FK (vin: String) references Vehicles(vin: String)
- UpcomingServices_ServiceTypes(event_id: Integer, service_type: String)
  - FK (event_id: Integer) references UpcomingServices(event_id: Integer)
  - FK (service_type: String) references ServiceTypes(service_type: String)
- Reminder(reminder_id: Integer, event_id: Integer, message: String, send_date: Date, was_sent: Boolean, was_read: Boolean)
  - FK (event_id: Integer) references UpcomingServices(event_id: Integer)

B. Normalization Process (Up to BCNF)

- **First Normal Form (1NF):** Ensure that all attributes have atomic values and eliminate repeating groups.
  - Ensure that attributes are atomic and relations in RDBMS are in 1st Normal Form by default
- **Second Normal Form (2NF):** Remove partial dependencies by ensuring that every non-key attribute is fully functionally dependent on a candidate key.
  - Almost all entities rely on auto-indexed, simple primary keys. This key functionally determines all non-key attributes.
  - The only composite keys are in connector tables for many-to-many relationships. These keys require all their attributes to functionally determine non-key attributes.
- **Third Normal Form (3NF):** Eliminate transitive dependencies so that non-key attributes depend only on a candidate key.
  - Users have a unique username and email, making them candidate keys. They are identifiers, able to replace userid in functionally determining the remaining attributes. Since userid, email, and username all technically functionally determine each other, we chose to keep them in the same table to simplify queries.
  - Our DB is designed to give the user maximum flexibility when populating forms with data. This means that attributes are all isolated from each other and do not lead to strict functional dependencies. For example, in CarShops, zip code could potentially functionally determine state and city. However, the address fields are all optional, and users may enter inconsistent or incomplete data, so this dependency is not strictly enforced. This design choice prioritizes flexibility over rigid normalization, allowing for varied real-world scenarios where zip codes may cover multiple cities or where user-entered address data might be partial.
  - As a result, most tables only have a single functional dependency, with the primary key functionally determining all attributes.
- **Boyce-Codd Normal Form (BCNF):** Strengthen 3NF by ensuring that every determinant is a candidate key.
  - All relations are already dependent on a candidiate key. No change needed in this step.

C. Per-Table Normalization Notes

- **Users:** Normalized to BCNF with unique constraints on username and email as candidate keys, ensuring no partial or transitive dependencies.
- **Vehicles:** In BCNF, with VIN as the primary key uniquely determining all vehicle attributes.
- **Owns:** Composite primary key (user_id, vin) with all attributes depending on the full key, satisfying 2NF and beyond.
- **CarShops:** Address fields decomposed into atomic attributes (street, city, state, zip code) to comply with 1NF; no partial or transitive dependencies present.
- **Mechanics:** Normalized to BCNF with mechanic_id as the primary key and email/phone number set to optional (not a User so no need to uniquely identify) to remove dependencies
- **ServiceRecords:** In BCNF with service_id uniquely identifying all service details.
- **WorkedOn:** Junction table with composite key and no non-key attributes, inherently normalized.
- **ServiceTypes:** Single attribute primary key, inherently normalized.
- **ServiceRecords_ServiceTypes:** Junction table associating services and types, inherently normalized.
- **Parts:** In BCNF with part_id as primary key.
- **ServiceRecords_Parts:** Normalized many-to-many association table with composite primary key.
- **Expenses:** BCNF with expense_id uniquely identifying all expense attributes.
- **FuelLog:** BCNF, uniquely identified by fuel_log_id.
- **MaintenanceEvents:** BCNF with event_id as primary key; no transitive dependencies.
- **MaintenanceEvents_ServiceTypes:** Composite key junction table, normalized.
- **Reminder:** Primary key adjusted to reminder_id for 2NF compliance; fully normalized afterward.

D. Indexing Strategy

To optimize query performance, several indexes have been created across key tables in the database. These indexes focus primarily on attributes frequently used in search conditions, join operations, and filtering. The strategy includes:

- Composite indexes on foreign keys to speed up join operations, such as the index on `Owns(user_id, vin)` and the index on `ServiceRecords(vin, service_date)`.
- Indexes on categorical and frequently filtered columns, for example, the index on `Expenses(category)` and the index on `ServiceRecords_ServiceTypes(service_type)`.
- Indexes to support queries filtering by location and association, such as the index on `CarShops(city, state)` and the index on `Mechanics(car_shop_id)`.
- Indexes to improve lookup on name and descriptive attributes, for instance, the index on `Parts(name)`.
- Indexes to accelerate operations on reminder and upcoming service scheduling, including the index on `Reminder(send_date, was_sent)` and the index on `UpcomingServices(user_id, status)`.
- Indexes on junction tables to optimize many-to-many relationship queries, such as the index on `WorkedOn(service_id)` and the index on `ServiceRecords_Parts(part_id)`.

E. Index Performance

The following figure illustrates the performance impact of the indexing strategy on some query operations and the most expensive operation within the database.

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
ownership_user_vehicle	0.000886	0.001799
expenses_by_category	0.002038	0.001706
service_records_user	0.002472	0.003321
upcoming_services_status	0.004396	0.004071

Fig. 2: Comparison of query execution times with and without indexing

As shown in Figure 2, the introduction of indexes significantly reduces query execution time, especially for the most complex operations. The operation that join the Expenses table with its sub-entities is by far the most expensive and sees the largest benefit. The improvement is around 50%. Some slightly less complicated operations see smaller improvements. The simplest operations at the bottom of Figure 2 see marginal improvements or variations in system timing due to how fast they are.

V. Major Design Decisions

- **Handling one-to-many relationships:** Used foreign keys to reduce entity clutter of having junction/connector tables when unnecessary.
- **Handling many-to-many relationships:** Resolved by introducing junction/connector tables to maintain normalization and data integrity. Used to intentionally allow each ServiceRecord to have multiple Parts, Mechanics, and ServiceTypes.
- **Enforcement of business rules:** Business rules applied at the 2nd tier (backend level) of a 3-tier DB system. Users never directly interact with the DB. All CRUD operations occur though forms, fields, and buttons at the 1st tier (website UI/UX level).
- **Choice of primary keys:** Used auto-increment key IDs for simplicity and performance, despite natural keys potentially providing better domain meaning.
- **Linking Entities to User:** Parts, Mechanics, and CarShops have a foreign key link to Users. Although admins can see all relations, the BD is designed so that regular users can only see the parts, mechanics, and car shops they personally add to the DB system. This FK is necessary to control which rows users can view.
- **Sub-Entities of Expenses:** Rather than have a single expenses table and using flags to differentiate attributes, we chose to create sub-entities. The category attribute in Expenses is a required field that is used to determine which sub-entity to join with to access their specific attributes.
- **User Flexibility vs Strict Normalization:** Our DB prioritizes user flexibility when entering information into the system. It can be tedious for users to always fill out every field of a form, especially when that information may be hard to find or not available. To that end, whenever possible (while maintaining DB integrity), fields are allowed to be optional. This means that restrictions on fields that usually create FDs during normalization are relaxed (example in 3NF explanation above).

VI. Implementation Summary

A. Technology Stack

- **Languages:** JavaScript (Node.js backend, vanilla JS frontend), MySQL, Python.
- **Frameworks/Libraries:** Express, mysql2, CORS, dotenv, nodemon, Bootstrap 5, Font Awesome, Faker.

- **Tools:** Node.js/npm, MySQL Server, shell scripts for setup/reset/start.

B. MySQL Connection

- Credentials read from `installation/mysql_credentials.conf`
- Connected via `mysql2.createConnection`
- Static site served by Express.

C. CRUD Operations

- Frontend sends SQL to backend via `/api/query`.
- `js/database.js` provides wrappers: `select`, `insertRecord`, `updateRecord`, `deleteRecords`.
- Used by feature managers for Users, Vehicles, Services, Expenses, and other entities.

D. Schema Constraints

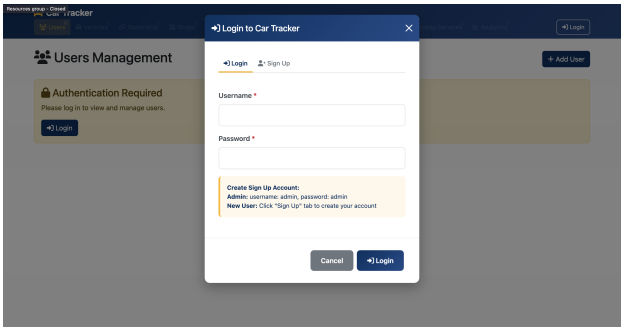
- Primary keys and composite keys for many-to-many relationships.
- Foreign keys with cascades for referential integrity.
- UNIQUE constraints on usernames and emails; ENUM for expense categories.
- Seed scripts create admin user and initialize service types.

E. Indexing and Query Structure

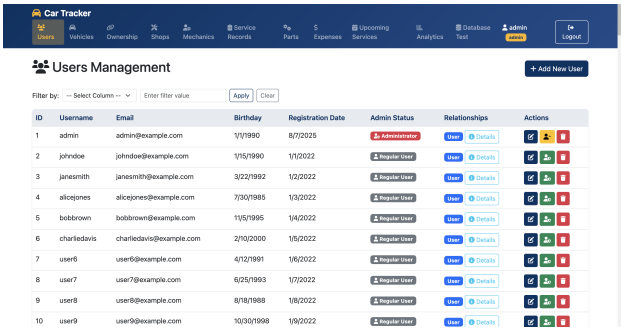
- Indexed PKs, FKs, and frequently filtered columns (VIN, category, date).
- Composite indexes (e.g., `(user_id, vin)`, `(vehicle_id, service_date)`) improve joins and filters.
- Analytics and dashboard queries aligned to indexes; performance verified with `EXPLAIN` and benchmarks.

VII. Demonstration of Example System Run

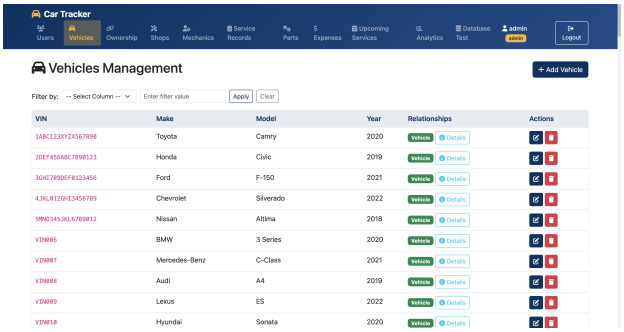
A. Login Page



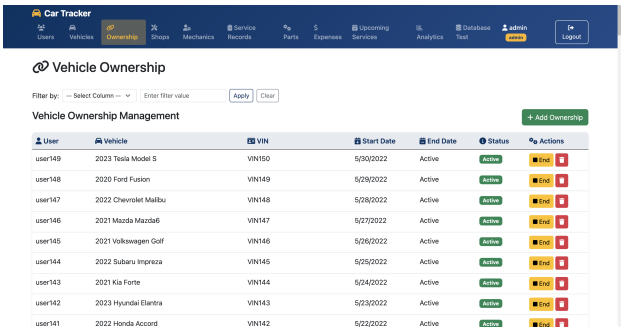
B. Users Page



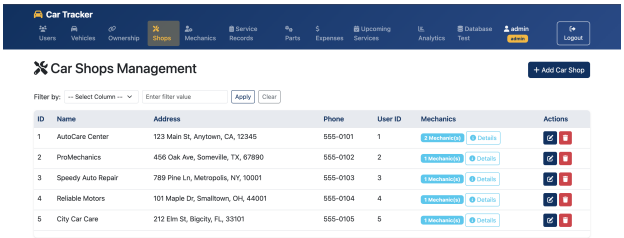
C. Vehicles Page



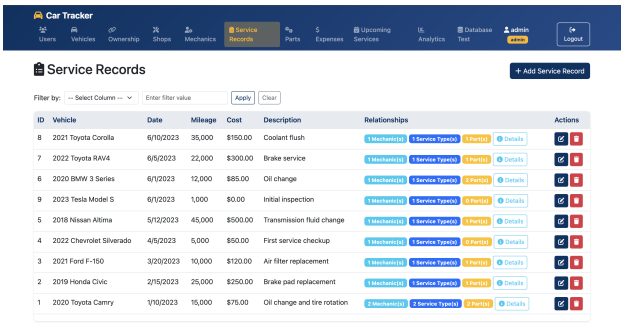
D. Ownership Page



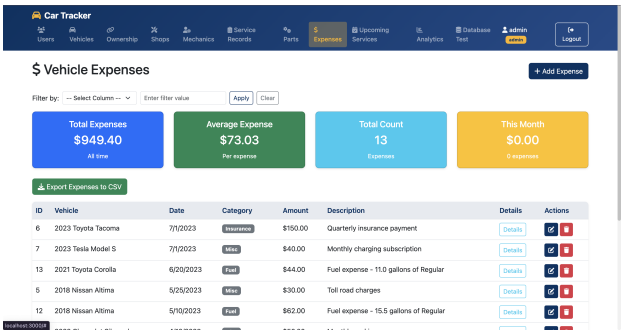
E. Shops Page (Only Admins see User_ID)



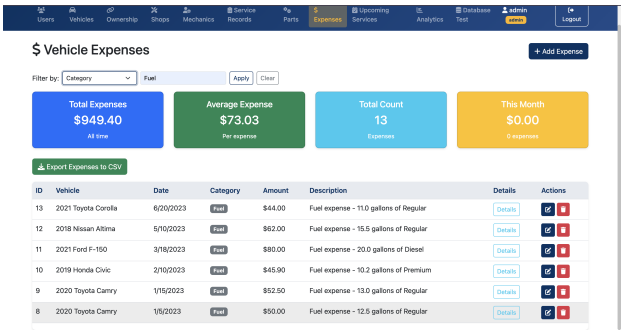
F. Service Records Page



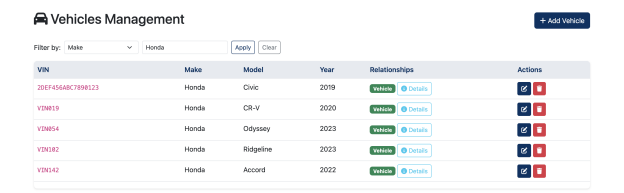
G. Expenses Page



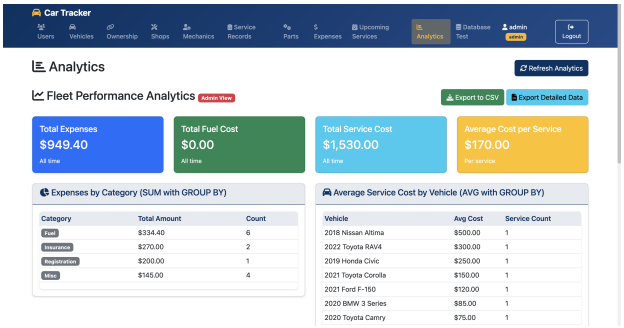
H. Expenses — Filter by Expense Category



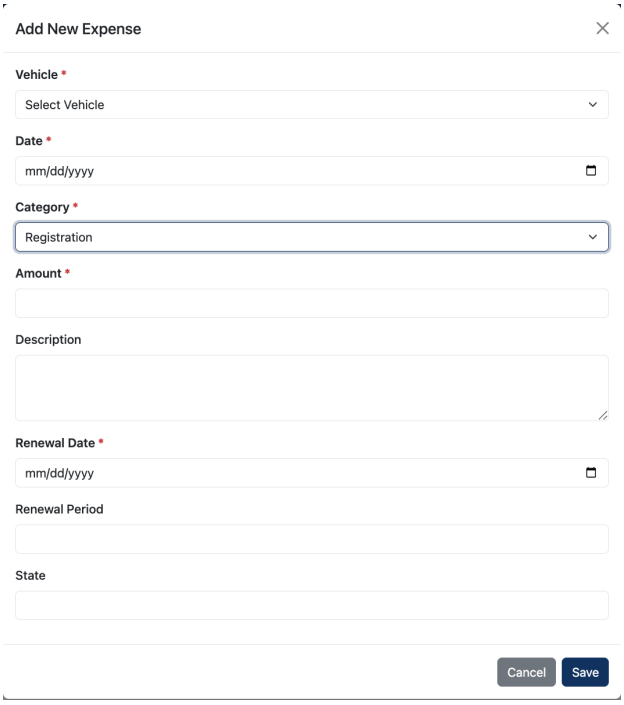
I. Vehicles — Filter by Car Make



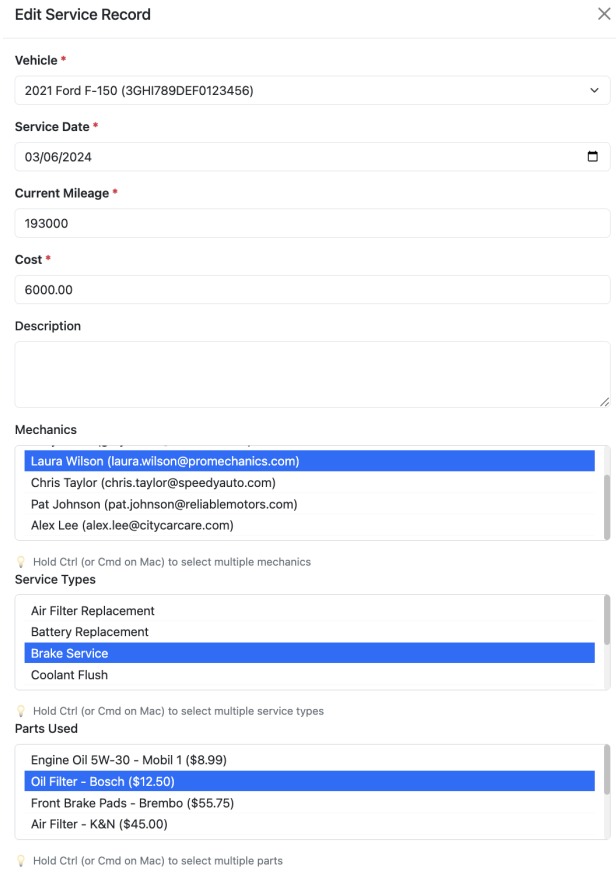
J. Analytics Page



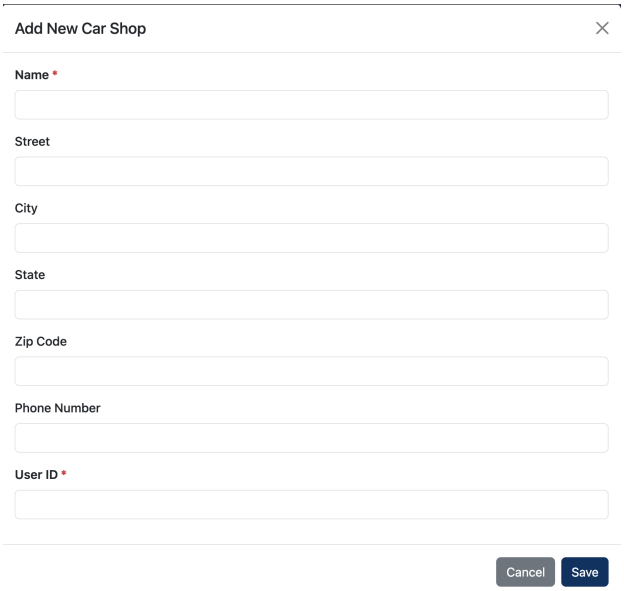
K. Add Expense Form (Category Reveals Specific Fields)



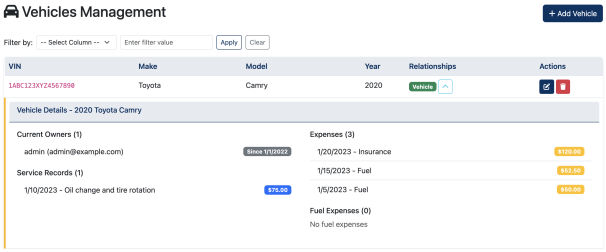
L. Edit Service Records Form



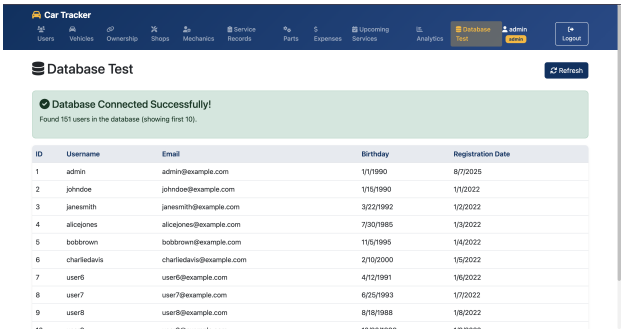
M. Add Car Shop (Admins can assign to User)



N. Vehicle Expanded Details



O. Database Test Page (Admin feature)



P. Example SQL Query and Output

```
SELECT CONCAT(v.year, ' ',
 v.make, ' ', v.model) AS vehicle_info,
       AVG(sr.cost) AS avg_cost,
       COUNT(*) AS service_count
FROM ServiceRecords sr
LEFT JOIN Vehicles v ON sr.vin = v.vin
GROUP BY sr.vin, v.make, v.model, v.year
ORDER BY avg_cost DESC;
```

Average Service Cost by Vehicle (AVG with GROUP BY)

Vehicle	Avg Cost	Service Count
2021 Ford F-150	$3,060.00	2
2022 Toyota RAV4	$300.00	1
2019 Honda Civic	$250.00	1
2018 Nissan Altima	$240.00	3
2021 Toyota Corolla	$150.00	1
2020 BMW 3 Series	$85.00	1
2020 Toyota Camry	$75.00	1
2022 Chevrolet Silverado	$50.00	1
2023 Tesla Model S	$0.00	1

VIII. Conclusions and Reflections

A. What We Learned

From a technical perspective, we learned the importance of building a solid foundation before beginning development. Initially, starting with the frontend gave us some progress, but connecting it to the backend and database proved difficult.

Restarting with a database–first approach provided a clear structure for how every component should connect, making future steps smoother. Along the way, we implemented solutions such as a centralized `AuthManager` for consistent role–based rendering, targeted refresh systems to handle race conditions, session management with `sessionStorage` for persistence after page reloads, and server–side pagination to handle large datasets without crashes.

Collaboratively, we realized the value of identifying team members' strengths and weaknesses early. Some were stronger in frontend work but less familiar with databases, while others excelled at backend development but had limited experience with full–stack integration. Understanding these differences allowed us to assign tasks more strategically and work toward our shared goal efficiently.

B. Challenges and How We Overcame Them

A major challenge was uncertainty about where to begin. Our early frontend work lacked a strong backend and database structure, which made integration difficult. This led us to rebuild from the ground up using a database–first approach, ensuring each feature was properly connected. Another challenge was the steep learning curve of linking the frontend, backend, and database, especially for members unfamiliar with parts of the stack. We overcame this by pairing less experienced member with those who had experience developing fullstack application.

The most significant collaborative challenge was scheduling during the summer semester. Coordinating meetings was difficult and sometimes slowed progress. Once we established a clear division of labor based on strengths, we were able to work asynchronously more effectively, keep momentum, and meet deadlines.

C. Improvements with More Time or Resources

With more time and resources, we would aim to make the application production–ready and expand its capabilities. We would deploy to AWS EC2 for scalability and global access, develop a fully documented RESTful API for third–party integrations with insurance companies, service shops, and financial institutions, and integrate AI–powered analytics to provide predictive maintenance suggestions, cost analysis, and actionable insights.

On the collaborative side, we would refine our workflow by implementing structured sprint planning and regular stand–up meetings to maintain communication and alignment, even during challenging scheduling periods. These improvements would make our development process more efficient and scalable for future projects.