UNIVERSIDAD DE GRANADA

BACHELOR'S THESIS

BACHELOR'S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

AIDiaCAR

Design and development of a sustainability-focused app for optimizing personal vehicle usage

Author
Juan Manuel Segura Duarte

Thesis supervisor
Rosa Ana Montes Soldado

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, June 2025

 Juan Manuel Segura Duarte, 2025

Design and development of a sustainability-focused app for optimizing personal vehicle usage

Juan Manuel Segura Duarte

Keywords: Sustainable Mobility, Green IT, Full-Stack Development, React Native, Vehicle Management, Gamification.

Abstract:

This project presents the design and development of *AlDiaCAR*[1], a cross-platform mobile and web application aimed at promoting sustainable and conscious use of a personal fleet of vehicles. The application is designed to support an individual user managing one or more vehicles, offering tools to centralize maintenance tracking and encourage environmentally-aware decisions.

AlDiaCAR provides three core functionalities: (1) unified vehicle maintenance tracking with smart reminders based on upcoming dates and distance-based intervals, (2) a trip logging system that updates vehicle metrics and user statistics, and (3) a recommendation system that suggests the most sustainable vehicle for a given route based on estimated emissions. To further engage the user, the application incorporates gamification elements and statistical feedback, fostering eco-responsible habits.

The system architecture is built with Expo (a React Native framework) for the frontend and Express.js (a Node.js framework) for the backend, supported by a MongoDB database for flexible data management. The project was developed following modular design principles and includes a comprehensive testing suite to ensure quality and reliability.

This report documents the design rationale, implementation details, and validation of the application, which aligns with the broader goals of sustainable software and "Green through IT" initiatives.

[1]The complete source code for this project is available at: https://github.com/jseg380/Trabajo-Fin-Grado

Diseño y desarrollo de una aplicación para la optimización del uso del vehículo privado con enfoque en sostenibilidad

Juan Manuel Segura Duarte

Palabras clave: Movilidad Sostenible, TI Verde, Desarrollo Full-Stack, React Native, Gestión de Vehículos, Gamificación.

Resumen:

Este proyecto presenta el diseño y desarrollo de *AlDiaCAR*[2], una aplicación multi-plataforma (móvil y web) orientada a fomentar un uso sostenible y consciente de una flota de vehículos personal. La aplicación está diseñada para dar soporte a un usuario individual que gestiona uno o más vehículos, ofreciendo herramientas para centralizar el seguimiento del mantenimiento y fomentar decisiones respetuosas con el medio ambiente.

AlDiaCAR ofrece tres funcionalidades principales: (1) seguimiento unificado del mantenimiento del vehículo con recordatorios inteligentes basados en fechas próximas e intervalos de distancia, (2) un sistema de registro de viajes que actualiza las métricas del vehículo y las estadísticas del usuario, y (3) un sistema de recomendaciones que sugiere el vehículo más sostenible para una ruta determinada en función de las emisiones estimadas. Para implicar al usuario, la aplicación incorpora elementos de gamificación y retroalimentación estadística que fomentan hábitos eco-responsables.

La arquitectura del sistema se ha desarrollado con Expo (un framework de React Native) para el frontend y Express.js (un framework de Node.js) para el backend, utilizando una base de datos MongoDB para una gestión de datos flexible. El proyecto se ha desarrollado siguiendo principios de diseño modular e incluye una completa suite de tests para garantizar su calidad y fiabilidad.

Este informe documenta la justificación del diseño, los detalles de implementación y la validación de la aplicación, que se alinea con los objetivos más amplios del software sostenible y las iniciativas de "TI Verde" (Green through IT).

[2]El código fuente completo de este proyecto está disponible en: https://github.com/jseg380/Trabajo-Fin-Grado

Acknowledgements

I would like to take this opportunity to express my heartfelt gratitude to all those who have supported me throughout my journey in completing this thesis.

First and foremost, I would like to thank my family for their unwavering love and encouragement. Your belief in my abilities has been a constant source of motivation, and I am forever grateful for your sacrifices and support.

I am also grateful to my friends and peers who have stood by me during this challenging process. Your camaraderie, late-night study sessions, and shared laughter have made this journey not only bearable but also enjoyable. Thank you for being my sounding board and for always believing in me.

Additionally, I would like to acknowledge the faculty and staff of the University of Granada for providing a nurturing academic environment. Your dedication to teaching and mentorship has profoundly impacted my educational experience.

This thesis is not just a reflection of my efforts but a testament to the collective support of all those around me. Thank you for being part of this journey.

Contents

List of Figures

List of Tables

Glossary

A

API (Application Programming Interface) A set of rules and protocols that allows different software applications to communicate. In this project, it refers to the REST API that connects the frontend mobile app to the backend server..

B

bcryptjs A password-hashing function designed to be slow and computationally intensive, which protects against brute-force search attacks. This library is used in the project to securely hash and store user passwords..

C

CI/CD (Continuous Integration/Continuous Deployment) A software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. In this project, CI is implemented via GitHub Actions to automatically run backend and frontend tests on every pull request..

CORS (Cross-Origin Resource Sharing) A browser security feature that restricts web pages from making requests to a different domain than the one that served the page. The backend uses the `cors` middleware to explicitly allow the frontend (e.g., at `localhost:3000`) to access the API (at `localhost:5000`)..

CRUD An acronym for Create, Read, Update, and Delete, which are the four basic functions of persistent storage. These operations are the foundation of the project's vehicle management API..

D

Docker A platform for developing and running applications in isolated environments called containers. It is used in this project via Docker Compose to create a consistent and reproducible development environment for the backend and database..

dotenv A zero-dependency module that loads environment variables from a `.env` file into `process.env`. It is used in the backend to manage configuration and secrets like database connection strings and JWT keys..

E

E2E (End-to-End) Testing A testing methodology used to verify the workflow of an application from start to finish. This project uses Playwright to conduct E2E tests by simulating real user scenarios in a browser..

Expo A framework and platform for universal React applications. It provides a set of tools and services built around React Native that simplify the development and deployment of the project's mobile app..

Express.js A minimal and flexible Node.js web application framework that provides a robust set of features for building APIs. It is the core framework for the AIDiaCAR backend..

G

Gamification The application of game-design elements (like achievements and stats) in non-game contexts to engage users and motivate specific behaviors, such as sustainable driving..

Green IT The practice of environmentally sustainable computing. It encompasses "Green in IT" (making computing itself more efficient) and "Green through IT" (using IT to enable sustainability in other domains), the latter of which is the focus of this project..

H

HCI (Human-Computer Interaction) A multidisciplinary field of study focusing on the design of computer technology and the interaction between humans (the users) and computers..

J

Jest A JavaScript testing framework with a focus on simplicity. It is used for the unit and integration testing of the backend API..

JSON (JavaScript Object Notation) A lightweight data-interchange format that is the standard for data transfer in this project's REST API..

JWT (JSON Web Token) A compact, URL-safe means of representing claims to be transferred between two parties. Used in this project for managing user authentication sessions via secure cookies..

M

Middleware In Express.js, these are functions that execute during the lifecycle of a request to the server. Used extensively in this project for authentication (`authMiddleware`) and file uploads (`uploadMiddleware`)..

MongoDB A document-oriented NoSQL database program. It was chosen for this project due to its flexible schema, which is ideal for storing the varied data of users and vehicles..

Mongoose An Object Data Modeling (ODM) library for MongoDB and Node.js. It is used throughout the backend to define schemas, validate data, and manage relationships between documents..

Monorepo A software development strategy where code for multiple projects is stored in the same repository. This strategy is used to manage the `frontend`, `backend`, and `tests` codebases together..

N

Node.js A back-end JavaScript runtime environment that executes JavaScript code outside a web browser. It is the foundation of the project's backend server..

O

ODM (Object Data Modeling) A programming technique for converting data between incompatible type systems in object-oriented programming languages. Mongoose is the ODM used in this project to map JavaScript objects to documents in MongoDB..

P

PaaS (Platform-as-a-Service) A category of cloud computing services that provides a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the underlying infrastructure..

Playwright A framework for web testing and automation. It is used for the end-to-end testing of the frontend, simulating user journeys across different browsers and devices..

R

React Native An open-source UI software framework used to develop applications for Android, iOS, and the Web from a single codebase. It is the core technology of the frontend..

REST (Representational State Transfer) An architectural style for designing networked applications. The project's backend is a RESTful API that uses HTTP requests to manage resources..

1 Introduction

1.1 Context and relevance: global emissions and personal vehicle management

The escalating climate crisis represents one of the most significant challenges of the 21st century, demanding urgent and innovative solutions across all sectors of society. Transportation stands out as a critical area for intervention. According to comprehensive data analysis, the transport sector was responsible for approximately 16.2% of global greenhouse gas (GHG) emissions in 2016. Within this figure, road transport—comprising cars, motorcycles, buses, and trucks—was the largest contributor, accounting for 11.9% of total global emissions. A striking 60% of these road transport emissions originate from passenger travel alone, primarily from the use of private vehicles [1].

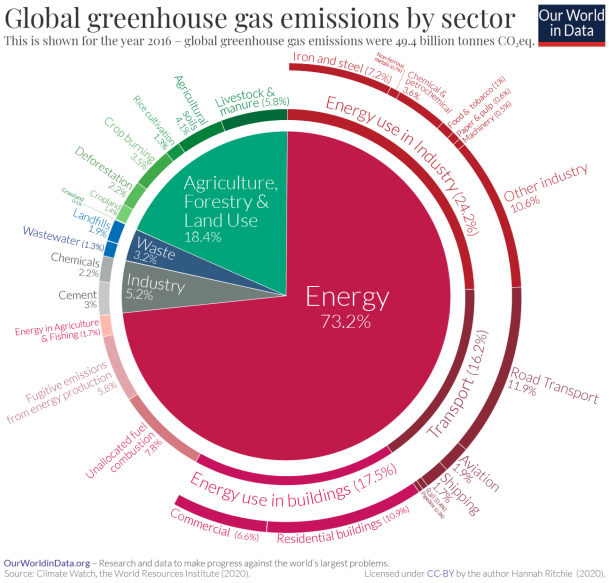

Figure 1.1: Breakdown of global greenhouse gas emissions by sector.

This reliance on private cars is particularly pronounced in developed nations. In the European Union, for instance, the number of passenger cars per thousand inhabitants reached 560 in 2022, signaling a persistent trend towards private vehicle ownership [2]. This trend, combined with longer vehicle lifecycles, presents a new challenge for the individual owner: managing a personal fleet that often consists of multiple vehicles, each with a distinct age, fuel type, and emissions profile. This complexity introduces significant inefficiencies, as daily transportation choices are often made based on convenience rather than optimal, sustainable selection.

This thesis addresses these inefficiencies by focusing on holistic sustainability in private vehicle use. The concept extends beyond simply owning an electric vehicle; it encompasses diligent maintenance to ensure peak operational efficiency, as poorly maintained vehicles can see a significant increase in fuel consumption and pollutant emissions [3]. It also involves conscious vehicle selection for each journey and the adoption of eco-driving habits.

Furthermore, this project is grounded in the principles of Green Computing, specifically "ICT for Sustainability." This pillar of Green IT focuses on applying software to influence and improve real-world processes. By developing intelligent software solutions, we can empower users to make more informed, environmentally conscious decisions. This project posits that a well-designed mobile application can serve as a persuasive technology, a concept defined as technology intentionally designed to change a person's attitudes or behaviors [4], thereby nudging users towards sustainable mobility patterns.

1.2 The problem statement: the management gap for the modern vehicle owner

1.2.1 Core problem

Individuals managing multiple personal vehicles lack integrated tools to:

- Centralize maintenance schedules (e.g., technical inspections, oil changes) across their fleet.
- Optimize vehicle selection for trips based on sustainability metrics.
- Track and be incentivized for eco-conscious usage patterns.

1.2.2 User persona

Alejandro Martínez

Scenario: Alejandro is a tech-savvy professional who owns three vehicles: a 2015 diesel SUV, a 2020 hybrid sedan, and a 2022 electric hatchback. As the primary manager of this personal fleet, he struggles to keep track of the different maintenance needs and timelines for each car and often makes suboptimal choices for short trips out of habit.

Pain points:

- **Maintenance Overlooks:** He recently missed the SUV's oil change deadline, leading to reduced engine efficiency and potential for premature component wear.
- **Suboptimal Selection:** He used the high-emission SUV for a 18km urban trip despite the fully charged electric car being available, simply because it was parked more conveniently.
- **Usage Tracking:** He has no centralized system to compare the total trip emissions or running costs across his different vehicles.
- **Eco-Awareness Gap:** He lacks clear, actionable feedback on how his daily vehicle choices impact his carbon footprint.

This results in higher operational costs, accelerated vehicle wear, and avoidable emissions—challenges directly addressable through a personal fleet management application, as illustrated in Figure 1.2.

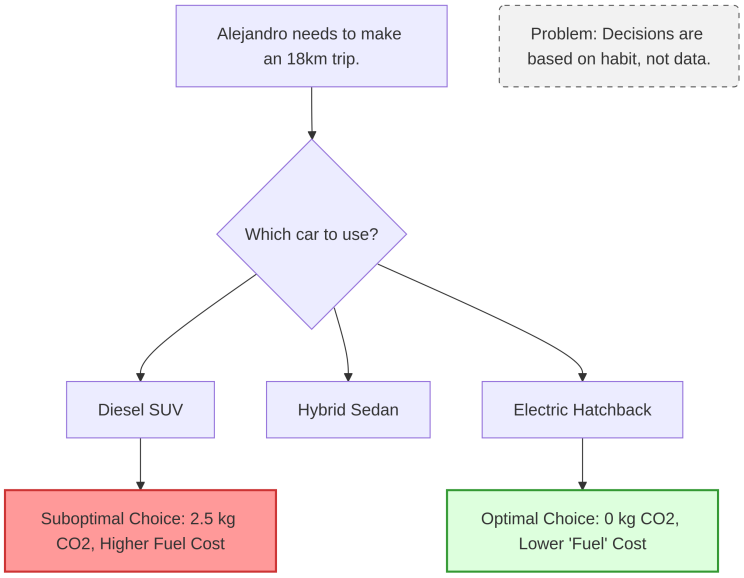

Figure 1.2: Conceptual diagram of the user's suboptimal vehicle selection problem.

1.3 Objectives and scope

1.3.1 Primary objective

Design and develop a cross-platform application (Android/web/iOS) to help an individual optimize the use of their multiple vehicles through:

- Centralized vehicle and maintenance management.
- Sustainability-driven trip recommendations.
- Behavioral reinforcement via gamification.

1.3.2 Specific objectives

Table 1.1: Project objectives and technical approaches

Objective	Technical approach
1. Cross-platform accessibility	Develop using React Native (Expo) for universal device access.
2. Maintenance automation	Implement alerts based on time and mileage triggers.
3. Vehicle selection optimization	Create a heuristic algorithm to recommend the optimal vehicle per trip based on CO_2 output.
4. Emissions tracking	Calculate CO_2 output using vehicle-specific emission factors.
5. Gamification system	Design an achievement and badge system for sustainable usage.

1.3.3 Scope boundaries

Included:

- Private vehicles (cars) managed by a single user account.
- Support for multiple vehicles per user.
- Rule-based recommendation heuristics.
- Integration with a mock third-party API for technical vehicle specifications.

Excluded:

- Multi-user/household coordination features.
- Real-time traffic data or public transport integration.
- Machine learning components for predictive analysis.

1.4 Methodology overview

Development approach:

- **Cross-platform strategy:**
  - Frontend: Expo (React Native) for reusable UI components.
  - Backend: Express.js (Node.js) REST API with MongoDB.
- **Testing protocol:**
  - Backend Testing: Unit and integration tests using Jest.
  - Frontend Testing: End-to-end tests for core user journeys using Playwright.
- **Deployment environment:**
  - Docker and Docker Compose for a consistent and reproducible development environment.

2 | Background and related work

To contextualize the contribution of this thesis, this chapter provides a review of the existing landscape of software solutions related to vehicle management and sustainable mobility. We will examine the state of the art in three distinct categories of applications: commercial fleet management systems, personal car maintenance trackers, and eco-routing tools. Furthermore, we will touch upon the theoretical foundations of gamification and Green IT that underpin the project's design philosophy. The chapter concludes with a gap analysis that identifies the unique niche this project aims to fill.

2.1 State of the art in vehicle management applications

The market for vehicle-related software is mature but highly segmented, with tools typically targeting either commercial enterprises or individual users with a narrow focus.

2.1.1 Commercial fleet management systems

Fleet management platforms such as *Fleetio* and *Samsara* offer powerful, comprehensive solutions for businesses that manage a large number of vehicles. Their core features include real-time GPS tracking, advanced telematics, fuel management, maintenance scheduling, and detailed operational analytics.

While highly effective for their target audience, these systems are ill-suited for the context of a private individual for several reasons:

- **Complexity and cost:** They are enterprise-grade platforms with pricing models and feature sets that are excessive for an individual managing a small number of personal cars.
- **User experience:** The user interface is designed for professional fleet managers and administrators, not for a casual, non-technical user.

- **Focus:** Their primary goal is operational efficiency and cost reduction for a business, not promoting personal sustainable habits.

2.1.2 Personal car maintenance applications

Applications like *Drivvo*, *Fuelly*, and the formerly popular *aCar* are designed for individual car enthusiasts and owners who want to diligently track their vehicle's health. These apps excel at logging fuel-ups, tracking expenses, and setting reminders for routine maintenance like oil changes or technical inspections (e.g., the Spanish ITV).

However, their utility is limited in the multi-vehicle scenario that this project addresses. Their main drawbacks are:

- **Single-vehicle focus:** While designed for a single user, they are often optimized for tracking just one or two vehicles. They lack the dashboarding and comparative features needed to efficiently manage a diverse personal fleet.
- **Lack of integrated sustainability:** While they may calculate fuel economy, their purpose is generally financial tracking rather than environmental impact. They do not offer intelligent recommendations on which vehicle is the most eco-friendly for a given trip.
- **Limited engagement model:** Most of these apps are functional utilities. They do not incorporate gamification to actively encourage better maintenance or driving habits.

2.1.3 Eco-routing and navigation tools

Mainstream navigation applications have started to incorporate sustainability features. A prominent example is *Google Maps*, which now offers "eco-friendly routing" that suggests a route optimized for lower fuel consumption. Other specialized apps also focus exclusively on calculating the most efficient route.

The limitation of these tools is their lack of integration with the user's specific context. They operate in isolation and cannot answer more complex questions relevant to an owner of multiple vehicles, such as:

- They are unaware of the specific vehicles a user owns or their relative efficiencies.
- They cannot factor in a vehicle's upcoming maintenance needs when making a recommendation.
- They are disconnected from the user's broader vehicle management goals.

Essentially, they can optimize the route for a given car, but cannot help the user choose the optimal car for that route from their personal fleet.

2.2 Theoretical foundations

The design of this project is informed by established principles from human-computer interaction and sustainable computing.

2.2.1 Gamification for behavior change

Gamification is defined as "the use of game design elements in non-game contexts" [5]. It is a powerful technique for increasing user engagement and motivating specific behaviors. In the context of this project, gamification elements such as badges and progress statistics are core mechanics designed to provide positive reinforcement for sustainable choices, such as selecting the lowest-emission vehicle or performing maintenance on time. By making sustainable actions rewarding and visible, the application aims to foster long-term habit formation, a claim supported by a large body of empirical studies [6].

2.2.2 Green IT and sustainable HCI

This project aligns with the principles of Green Information Technology (Green IT), also known as Sustainable Human-Computer Interaction (HCI). While some Green IT initiatives focus on reducing the energy consumption of computing hardware itself ("Green in IT"), this project exemplifies the "Green through IT" concept. It uses software as a persuasive technology to influence user behavior in the physical world, leading to tangible environmental benefits. This aligns with research by Berkhout and Hertin, who analyzed how digital technologies can both "de-materialise and re-materialise" environmental impacts, highlighting the potential for software to guide more sustainable outcomes [7].

2.3 Gap analysis and niche identification

A review of the state of the art reveals a clear gap in the market. While specialized tools exist for commercial fleets, single-car maintenance, and generic eco-routing, no existing solution integrates these functionalities into a single, cohesive platform designed specifically for the individual managing a personal, multi-vehicle fleet. This synthesis is the core contribution of this project, as illustrated in Figure 2.1.

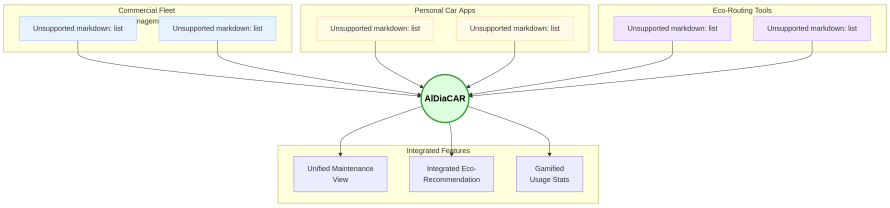

Figure 2.1: Visual representation of the market gap and AIDiaCAR's position as an integrated solution.

Table 2.1 further summarizes the limitations of existing application categories in relation to this project's goals.

Table 2.1: Summary of gaps in existing application categories

Application category	**Primary focus**	**Identified gaps for this project**
Fleet management systems	B2B operational efficiency and large-scale logistics.	Not designed for personal use; too complex and expensive; lacks focus on individual habit formation.
Personal maintenance apps	Tracking expenses and maintenance for a single vehicle.	Lacks robust multi-vehicle management features and integrated sustainability recommendations.
Eco-routing tools	Calculating a fuel-efficient route for a single, generic journey.	Not integrated with a user's specific vehicle data; cannot recommend the best vehicle, only the best route.

Therefore, this thesis project, *AIDiaCAR*, is positioned to fill this distinct niche. It aims to be the first application that holistically addresses the needs of an individual

managing multiple vehicles by unifying:

1. **Personal fleet management:** Centralized, user-friendly tracking of maintenance for multiple vehicles.
2. **Integrated sustainability recommendations:** A system that recommends not only an eco-friendly route but also the most appropriate, low-emission vehicle from the user's fleet for that route.
3. **Behavioral reinforcement:** A gamification layer designed to motivate and reward sustainable choices and diligent vehicle care.

By combining these three pillars, the project provides a novel contribution to the field of sustainable HCI and offers a practical tool to help individuals reduce their personal transportation footprint.

3 System design and architecture

This chapter details the architectural design of the application, named *AlDiaCAR*[1]. The design is guided by the project's core objectives: to provide a cross-platform, scalable, and maintainable solution for optimizing an individual's personal vehicle use with a focus on sustainability. We will discuss the high-level architecture, break down the system into its primary components, define the data flow, justify the technology stack, and outline the logic for the sustainability and gamification features.

3.1 High-level architecture

The system is designed following a classic client-server architecture, which decouples the user interface from the core business logic and data storage. This model enhances security, scalability, and allows for multiple types of clients to be developed in the future without altering the backend.

The main components are:

- **Frontend client:** A cross-platform mobile application built with React Native and the Expo framework. It is responsible for all user interactions, data presentation, and communication with the backend.
- **Backend server:** A Node.js application using the Express.js framework. It serves as the central hub, handling business logic, user authentication, data processing, and communication with the database.
- **Database:** A MongoDB NoSQL database that stores all persistent data, including user profiles, vehicle details, and trip logs.
- **Mock API:** A secondary Node.js server that simulates a third-party API for retrieving vehicle specifications, ensuring development and testing can occur offline and with deterministic data.

[1] A placeholder name for the application, derived from "Al Día" (up-to-date) and "Car".

Figure 3.1 provides a visual representation of this architecture and the data flow between components within the development environment.

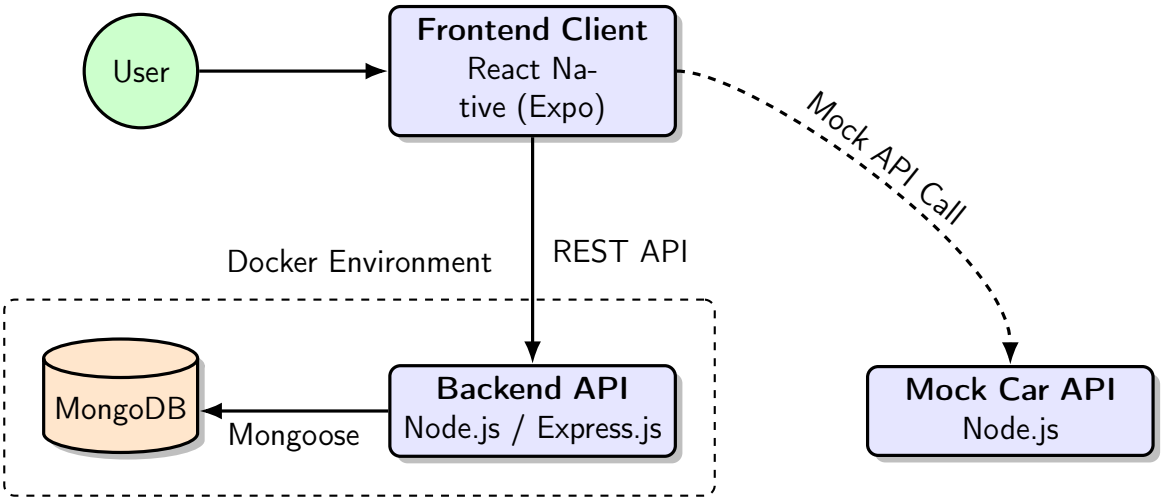

Figure 3.1: High-level system architecture, including the development environment.

3.2 Component overview

The system is logically divided into frontend and backend components, each with distinct responsibilities that reflect the project's source code organization.

3.2.1 Frontend components (React Native)

The frontend is structured as a mobile application using Expo, with a file-based routing system. The key components, located in `src/frontend/app/`, are:

- **Authentication (‘/app/(auth)‘):** Screens for user registration and login.
- **Main application tabs (‘/app/(tabs)‘):** The core user-facing part of the app:
  - **Home (‘/home‘):** A dashboard showing upcoming maintenance alerts.
  - **Vehicles (‘/vehicles‘):** A module for a user to manage their personal fleet, including adding, viewing, and editing vehicles.
  - **Routes (‘/routes‘):** The interface for the recommendation system.
  - **Statistics (‘/stats‘):** Implements gamification and data visualization, showing user stats and earned achievements.

- **State management (‘/context/AuthContext.tsx‘):** A React Context that manages the global authentication state, making user session data available throughout the component tree.
- **Localization (‘/localization‘):** Supports internationalization with dedicated folders for English and Spanish, allowing for easy translation of all UI text.

3.2.2 Backend components (Node.js)

The backend, located in `src/backend/`, is a RESTful API built with Node.js and Express.js. Its structure promotes separation of concerns:

- **Routes (‘/routes‘):** Defines the API endpoints. The project uses dedicated files for each resource (e.g., `authRoutes.js`, `vehicleRoutes.js`, `maintenanceRoutes.js`).
- **Controllers (‘/controllers‘):** Contains the business logic for each route. For example, `authController.js` handles registration and login, while `maintenanceController.js` contains the logic to calculate and sort vehicle alerts.
- **Models (‘/models‘):** Defines the data schemas for MongoDB using Mongoose. The core models are `User.js`, `Vehicle.js`, and `Trip.js`.
- **Middleware (‘/middleware‘):** Contains functions that process requests before they reach the controller. `authMiddleware.js` is crucial for protecting routes by verifying the user's JWT.

3.2.3 Data model

The NoSQL data model, implemented in MongoDB, was chosen for its flexibility. The primary collections are `Users`, `Vehicles`, and `Trips`. Relationships are managed through object references (e.g., a `Trip` document contains a reference to the `_id` of the `User` and `Vehicle`). This structure is well-suited for the application's needs, as vehicle attributes and maintenance requirements can vary significantly. A diagram of the data model is presented in Figure 3.2.

3.3 Data flow and API design

Communication between the frontend and backend occurs over HTTPS via a RESTful API using standard HTTP methods and JSON. A typical data flow for a core feature, such as getting a vehicle recommendation, is as follows:

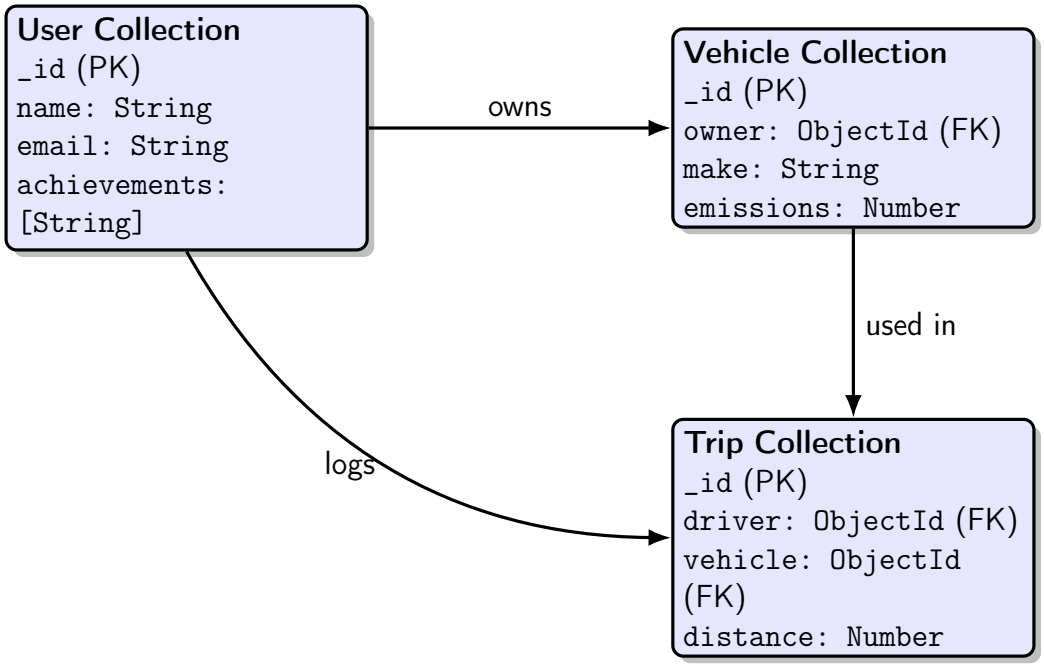

Figure 3.2: Simplified Data Model illustrating the relationships between the primary collections.

1. The user enters a trip distance on the "Routes" screen and requests a recommendation.
2. The React Native client sends a POST request to the backend endpoint `/api/recommendations`. The browser automatically includes the JWT cookie with the request.
3. The backend's authentication middleware intercepts the request and verifies the JWT.
4. The recommendation controller fetches all vehicles associated with that user from the database.
5. The controller's algorithm calculates the estimated emissions for each vehicle and responds with a sorted JSON array.
6. The frontend parses the response and displays the recommendations to the user.

A complete specification of all API endpoints is available in Appendix A.

3.4 Non-functional requirements

Beyond the core features, the system was designed with several non-functional requirements in mind:

- **Cross-platform compatibility:** React Native and Expo ensure a single codebase can be deployed to both iOS and Android, as well as the web.
- **Scalability:** The backend is stateless, allowing for horizontal scaling. MongoDB also offers robust scaling capabilities.
- **Security:** User authentication is handled using JWTs stored in secure, HTTP-only cookies. Passwords are never stored in plain text, only as bcrypt hashes.
- **Maintainability:** The project is structured into logical modules with a clear separation of concerns. This modularity, combined with TypeScript, improves code quality and makes future maintenance easier.
- **Usability:** The application provides internationalization support and aims for a clean, intuitive user interface.

3.5 Technology choices justification

- **React Native (with Expo):** Chosen for its ability to build native-quality applications from a single JavaScript/TypeScript codebase. Expo further accelerates development by simplifying the build process and managing native project configuration.
- **Node.js with Express.js:** Selected for the backend due to its high performance with asynchronous I/O, making it ideal for an API-driven application. Its use of JavaScript creates language synergy with the frontend.
- **MongoDB:** A NoSQL database was preferred over a relational one due to its flexible schema, which is advantageous for storing vehicle data where attributes and maintenance items can vary.
- **Docker:** The inclusion of a `docker-compose.yaml` file facilitates a consistent and isolated development environment, eliminating "it works on my machine" issues.
- **Jest & Playwright:** Selected for testing. Jest is a standard for unit testing Node.js applications, while Playwright is a powerful tool for E2E testing that simulates real user interactions.

3.6 Gamification and sustainability logic

The logic for these features is implemented on the backend to ensure consistency and security.

- **Sustainability recommendation:** The algorithm calculates total CO_2 emissions for a trip using the formula:

$$\text{Emissions } (gCO_2) = \text{Distance (km)} \times \text{Emission Factor } (gCO_2/km) \tag{3.1}$$

  The system recommends the vehicle with the lowest calculated emissions.

- **Gamification:** The backend tracks user actions to award achievements. For example, upon creating the first vehicle, the 'FIRST_VEHICLE' key is added to the user's `achievements` array in the database. When a trip is logged that surpasses a distance milestone (e.g., 1000 km), the corresponding achievement key is granted.

4 | Implementation details

This chapter transitions from the abstract design of Chapter 3 to the concrete technical implementation of the *AIDiaCAR* system. It provides a detailed walkthrough of the project's source code, explaining the structure, key modules, and programming patterns used in both the backend and frontend.

4.1 Project structure overview

The project is organized as a monorepo, a single repository containing all the code for the system. This approach simplifies dependency management and streamlines development across the different parts of the application. The high-level directory structure is shown in Figure 4.1.

The `src` directory cleanly separates the `backend` and `frontend` concerns. The `docker` directory ensures a reproducible development environment, while the `tests` directory houses the quality assurance framework.

4.2 Backend implementation (Node.js)

The backend is a RESTful API built with Node.js and the Express.js framework. It is responsible for all business logic, data persistence, and security.

4.2.1 Data models (Mongoose)

Data schemas are defined in `src/backend/models/` using Mongoose. The principal models are `User.js`, `Vehicle.js`, and `Trip.js`. A key feature is the use of Mongoose's `timestamps` option in the User schema to automatically manage `createdAt` and `updatedAt` fields. The password field uses `select: false` to prevent it from ever being sent in an API response. The full schema definitions are detailed in Appendix B.

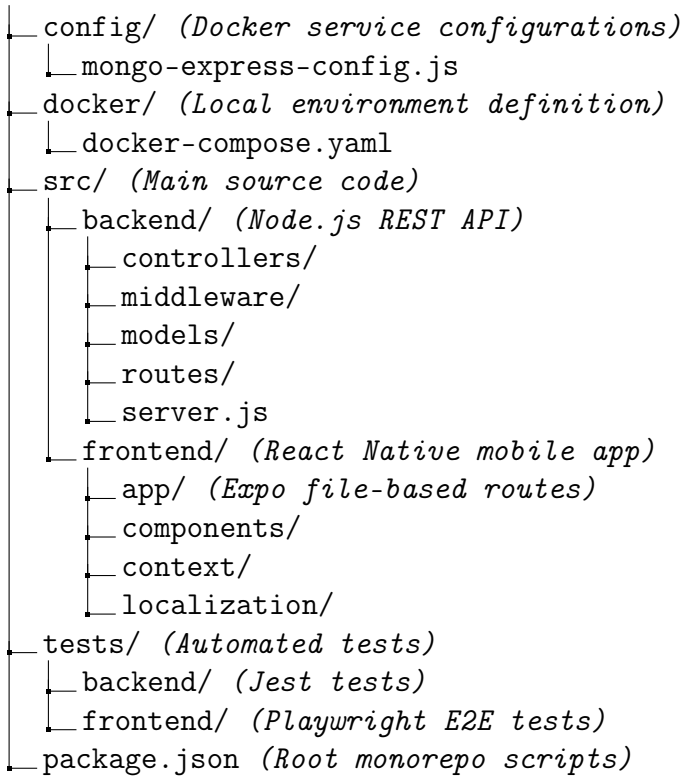

Figure 4.1: High-level project directory structure.

4.2.2 Controllers and business logic

Controllers, located in `src/backend/controllers/`, contain the core logic for each API endpoint. While many controllers perform standard CRUD operations, some, like `tripController.js`, encapsulate more complex business logic. The `logSimulatedTrip` function demonstrates how a single API call can trigger multiple state changes across the system.

Listing 4.1: Core logic from `tripController.js`

```
// 1. UPDATE VEHICLE MAINTENANCE METRICS
if (vehicle.upcomingMaintenance) {
  if (vehicle.upcomingMaintenance.brakes?.distance) {
    vehicle.upcomingMaintenance.brakes.distance -= distance;
  }
  // ... other distance-based counters are reduced
}
```

```
// 2. CREATE THE TRIP RECORD
const calculatedEmissions = distance * (vehicle.emissions || 150);
await Trip.create({ driver: userId, vehicle: vehicleId, distance, ... })

// 3. UPDATE USER STATS & GRANT ACHIEVEMENTS
user.stats.distanceTraveled += distance;
const achievementsToGrant = [];
if (!user.achievements.includes('FIRST_TRIP')) {
  achievementsToGrant.push('FIRST_TRIP');
}
if (oldDistance < 1000 && user.stats.distanceTraveled >= 1000) {
  achievementsToGrant.push('DIST_1000');
}
if (achievementsToGrant.length > 0) {
  user.achievements.push(...achievementsToGrant);
}

// 4. SAVE ALL CHANGES ATOMICALLY
await Promise.all([vehicle.save(), user.save()]);
```

This function shows how logging a trip correctly decrements maintenance counters on the vehicle, creates a new trip record, and updates the user's statistics and achievements, all within a single transaction. Figure 4.2 illustrates this interaction.

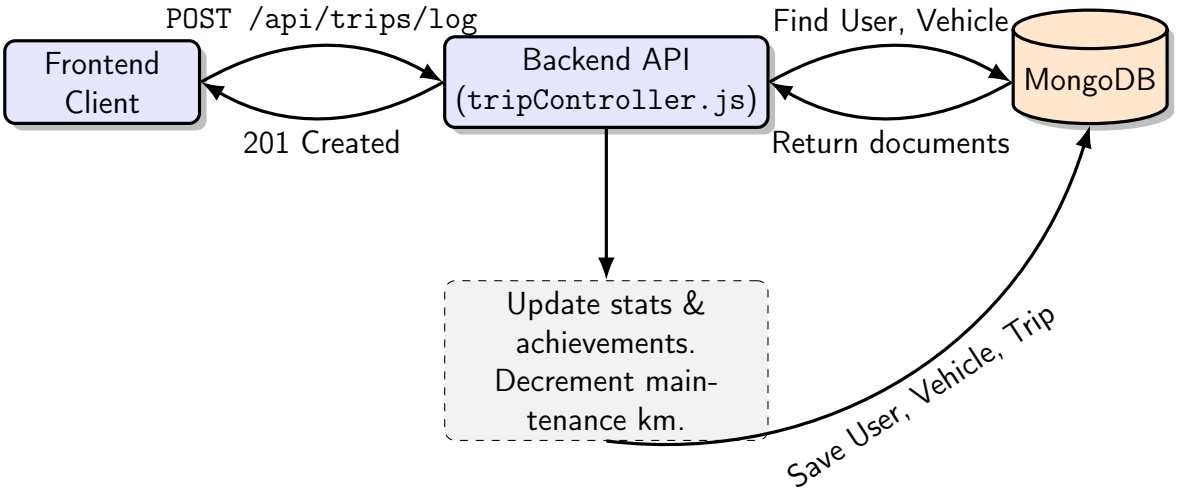

Figure 4.2: Sequence diagram for the trip logging data flow.

4.2.3 API routes and middleware

Routes defined in `src/backend/routes/` map the API endpoints to controller functions. The most critical middleware, `authMiddleware.js`, protects these routes. It extracts the JWT from an `HttpOnly` cookie, verifies it, and attaches the user's ID to the request object. This makes the user's identity available to any protected controller.

4.3 Frontend implementation (React Native)

The frontend is a cross-platform mobile application developed using React Native with the Expo framework and TypeScript for static typing.

4.3.1 Navigation and screen structure

The application uses Expo's file-based router. Directories and files within `src/frontend/app/` automatically become routes.

- **Layouts (‘_layout.tsx‘):** These files define the shell UI, such as the main tab bar defined in `app/(tabs)/_layout.tsx`.
- **Authentication flow (‘app/(auth)‘):** A route group for the login and register screens, active when a user is not authenticated.
- **Main screens (‘app/(tabs)‘):** Represent the core features. A key pattern used is `useFocusEffect` from Expo Router to re-fetch data whenever a screen comes into view, ensuring data is always fresh after an update (e.g., after adding a new vehicle).

4.3.2 State management and API communication

Global state for user authentication is managed via React's Context API in `context/AuthContext.tsx`. This provider stores the user's authentication status and profile information. Communication with the backend is handled using the `axios` library, which is configured with `withCredentials: true` to automatically handle the sending and receiving of authentication cookies.

4.4 Integration and deployment

4.4.1 Local development with Docker

To ensure a consistent development environment, the project utilizes Docker and Docker Compose. The `docker/docker-compose.yaml` file defines all services re-

quired to run the application stack locally, including the backend, frontend, database, and the Mongo Express GUI. A developer can start the entire stack with a single command: `docker-compose up`.

4.4.2 Deployment strategy

While the current focus is a robust prototype, a potential production deployment strategy would involve:

- **Backend:** Containerizing the Node.js application and deploying it to a Platform-as-a-Service (PaaS) like Heroku or a container orchestrator.
- **Database:** Using a managed database service like MongoDB Atlas.
- **Frontend:** Building the mobile application for production using Expo Application Services (EAS) and submitting the binaries to the app stores.

4.5 Testing strategy and implementation

The `tests/` directory houses a multi-layered testing strategy. For full details on the specific test cases, see Appendix D.

- **Backend unit & Integration tests:** The `tests/backend/` directory uses Jest to test modules in isolation and Supertest to test API endpoint integration.
- **Frontend E2E tests:** The `tests/frontend/` directory is configured for Playwright, which runs automated tests that simulate key user journeys in a real browser environment.

4.6 Security and privacy considerations

Security is implemented through several mechanisms:

- **Password hashing:** The `bcryptjs` library is used to securely hash and salt user passwords.
- **Authentication:** JWTs are stored in secure, `HttpOnly` cookies, which helps mitigate cross-site scripting (XSS) attacks.
- **Route protection:** The `authMiddleware` ensures that only authenticated users can access protected endpoints.

- **Environment variables:** Sensitive information like database connection strings and JWT secrets are managed via `.env` files and are not committed to version control.

5 | Results and validation

This chapter presents the results of the implementation, validating the functionality of the AlDiaCAR prototype against the objectives defined in Chapter 1. The validation process confirms that the system's key features are operational and effectively address the problem statement outlined for our user persona, Alejandro Martínez. This chapter provides visual evidence of the working frontend, complemented by a summary of the automated tests that verify the backend logic.

5.1 Frontend validation via core user workflows

The most direct way to validate the application is to walk through the essential user journeys. The following sections use screenshots from the running application to demonstrate that the primary features have been successfully implemented.

5.1.1 User registration and authentication

The first step for any user is to create an account and log in. The system's authentication flow, managed by the components in `/app/(auth)`, was tested manually and via Playwright's E2E tests. Figure 5.1 shows the application's entry point, and Figure 5.2 shows the user's profile after a successful login, confirming the authentication and data retrieval process is working.

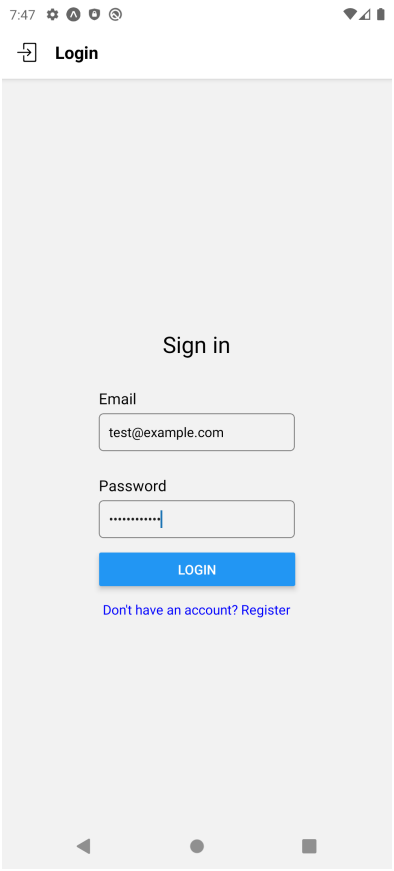

Figure 5.1: The user login screen.

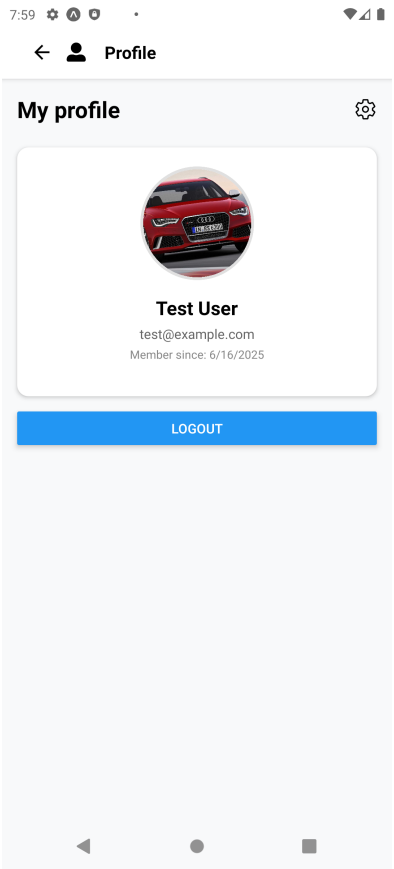

Figure 5.2: The user profile screen after successful authentication.

5.1.2 Vehicle fleet management

A core requirement for Alejandro is to manage his personal fleet. The application provides a seamless interface to add and view his cars. Figure 5.3 shows the form used to add a vehicle, and Figure 5.4 shows the central list of all his registered vehicles, providing a complete overview of his fleet. This validates the vehicle CRUD (Create, Read, Update, Delete) functionality.

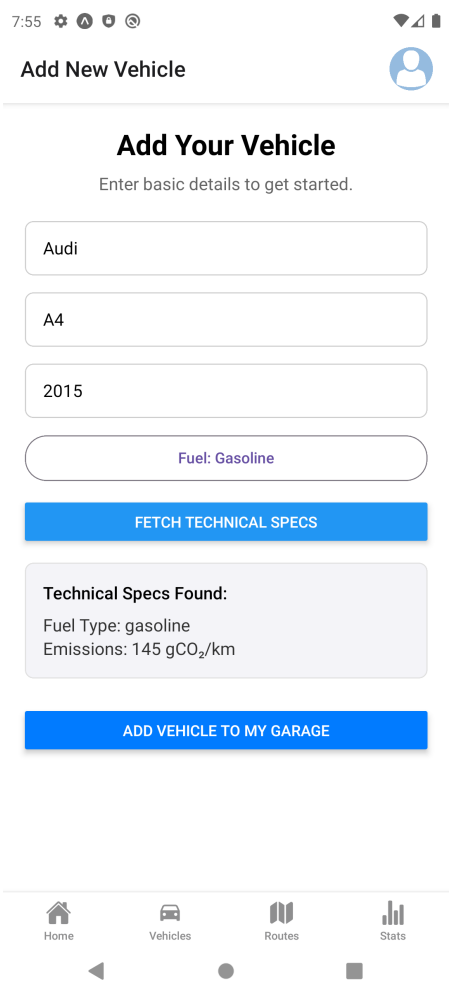

Figure 5.3: Adding a new vehicle to the user's fleet.

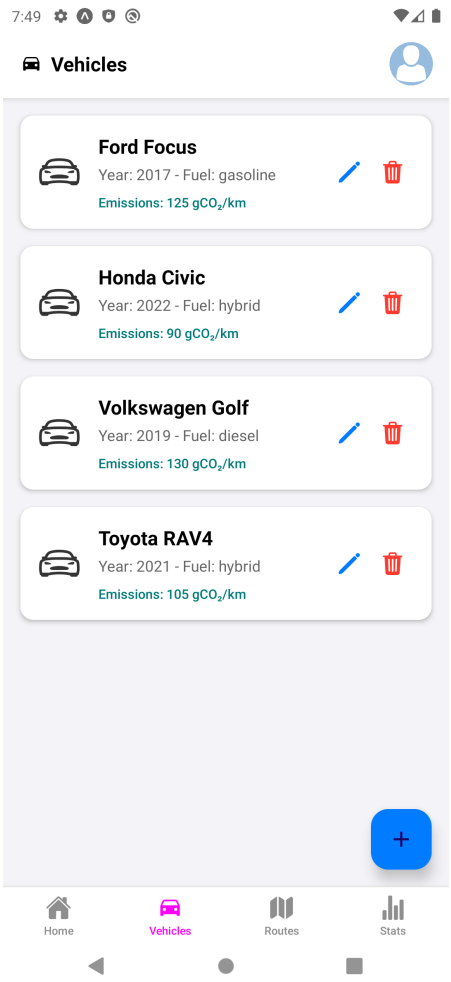

Figure 5.4: The user's list of registered vehicles.

5.1.3 Sustainability and maintenance recommendations

The application's primary goal is to provide actionable, data-driven advice. Figure 5.5 demonstrates the recommendation system in action, suggesting the most eco-friendly vehicle for a trip. Figure 5.6 shows the main dashboard, which displays sorted maintenance alerts, directly addressing Alejandro's pain point of overlooking important tasks.

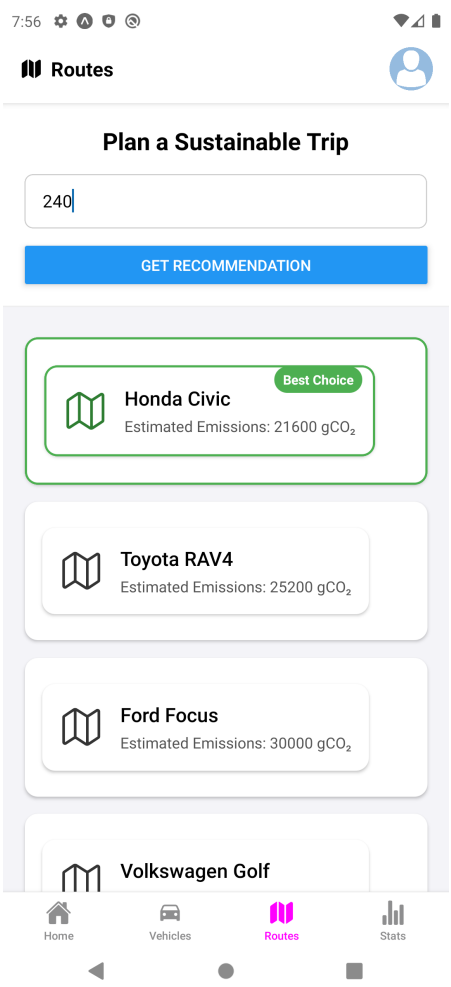

Figure 5.5: The vehicle recommendation interface.

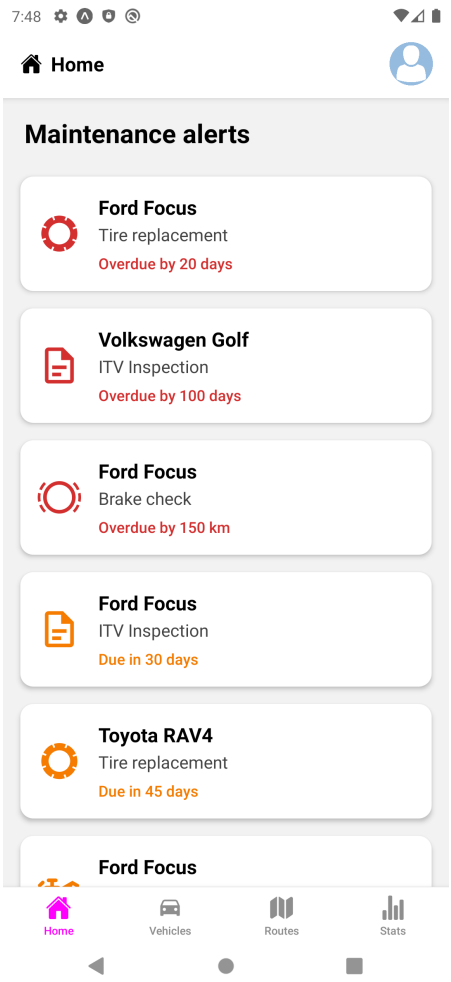

Figure 5.6: The main dashboard displaying maintenance alerts.

5.2 Backend logic and API validation

To validate the backend logic independently of the frontend, a comprehensive suite of automated integration tests was developed using Jest and Supertest. These tests verify the correctness of the API endpoints, business logic, and database interactions in a controlled environment.

A key example from this test suite is the validation of the sustainability recommendation endpoint. The test case, shown below, verifies that the API correctly processes an authenticated request and returns a properly calculated and sorted list of vehicle recommendations, which is a core piece of the application's business logic.

Listing 5.1: Integration test for the recommendation API from `tests/backend/features.test.js`

```
it('should return a sorted list of vehicle recommendations', async () =>
  const res = await request(app)
    .post('/api/recommendations')
    .set('Cookie', `jwt=${token}`) // Manually set auth cookie
    .send({ distance: 100 });

  expect(res.statusCode).toEqual(200);
  expect(Array.isArray(res.body)).toBe(true);

  // Verify the business logic: is the list sorted by emissions?
  expect(res.body[0].totalEmissions).toBeLessThanOrEqual(res.body[1].tot
});
```

This test confirms not only that the endpoint is reachable and secure, but also that its complex logic—calculating emissions for multiple vehicles and sorting them—is functioning as designed. A full summary of the test strategy is available in Appendix D.

5.3 Validation summary

The combination of visual confirmation from the frontend and successful execution of the automated test suites confirms that the prototype successfully implements its core objectives. Table 5.1 maps the project objectives to their implemented and validated status.

Table 5.1: Summary of Validated Project Objectives

Objective	**Validation Method**	**Status**
Cross-platform accessibility	Manual testing & Playwright E2E tests on web browser	Implemented
Vehicle management (CRUD)	Frontend Screenshots, Backend tests (Jest), E2E test (Playwright)	Implemented
Authentication system	Frontend Screenshots, Backend tests (Jest), E2E test (Playwright)	Implemented
Maintenance automation	Frontend Screenshots, Backend integration test (Jest)	Implemented
Sustainability recommendation	Frontend Screenshots, Backend integration test (Jest)	Implemented
Gamification system	Backend integration test for achievements (Jest)	Implemented

Bibliography

[1] Hannah Ritchie. Sector by sector: where do global greenhouse gas emissions come from? *Our World in Data*, 2020. https://ourworldindata.org/ghg-emissions-by-sector.

[2] Eurostat. Passenger cars per 1 000 inhabitants reached 560 in 2022. 2024. https://ec.europa.eu/eurostat/web/products-eurostat-news/w/ddn-20240117-1.

[3] International Energy Agency. Fuel economy in major car markets: Technology and policy drivers 2005-2019. Technical report, IEA, 2021. https://www.iea.org/reports/fuel-economy-in-major-car-markets.

[4] B.J. Fogg. *Persuasive Technology: Using Computers to Change What We Think and Do*. Morgan Kaufmann, 2002.

[5] Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart Nacke. Gamification: Toward a definition. *CHI 2011 Workshop*, 2011.

[6] Juho Hamari, Jonna Koivisto, and Harri Sarsa. Does gamification work?–a literature review of empirical studies on gamification. *HICSS*, 2014.

[7] Frans Berkhout and Julia Hertin. De-materialising and re-materialising: digital technologies and the environment. *Futures*, 36(8):903–920, 2004.

A | Appendix: API Reference

This appendix provides a detailed specification for the REST API endpoints implemented in the AIDiaCAR backend. All endpoints requiring authentication are protected by a middleware that verifies a JWT sent as an HTTP-only cookie.

A.1 Authentication Endpoints

Base Path: `/api/auth`

A.1.1 POST /register

- **Description:** Registers a new user account.
- **Protection:** Public.
- **Request Body:**

```
{
    "name": "string",
    "email": "string (unique)",
    "password": "string"
}
```

- **Success Response (201 Created):** Returns the new user's profile and sets a JWT cookie.

```
{
    "_id": "string (ObjectId)",
    "name": "string",
    "email": "string",
    "avatar": "string (path)"
```

```
}
```

- **Error Response (400 Bad Request):** If validation fails or email already exists.

A.1.2 POST /login

- **Description:** Authenticates an existing user.
- **Protection:** Public.
- **Request Body:**

```
{
    "email": "string",
    "password": "string"
}
```

- **Success Response (200 OK):** Returns the user's profile and sets a JWT cookie.

```
{
    "_id": "string (ObjectId)",
    "name": "string",
    "email": "string",
    "avatar": "string (path)"
}
```

- **Error Response (401 Unauthorized):** If credentials are invalid.

A.1.3 POST /logout

- **Description:** Logs the user out by clearing the JWT cookie.
- **Protection:** Public.
- **Success Response (200 OK):**

```
{
    "message": "Logged out successfully"
}
```

A.2 User Profile Endpoints

Base Path: `/api/users`

A.2.1 GET /profile

- **Description:** Retrieves the complete profile of the currently authenticated user.
- **Protection:** JWT Required.
- **Success Response (200 OK):**

```
{
    "_id": "string (ObjectId)",
    "name": "string",
    "email": "string",
    "avatar": "string (relative path)",
    "avatarUrl": "string (full URL)",
    "stats": { "distanceTraveled": "number", ... },
    "joinDate": "Date",
    "achievements": ["string", ...]
}
```

- **Error Response (404 Not Found):** If the user associated with the token cannot be found.

A.2.2 PUT /profile/avatar

- **Description:** Updates the user's profile picture.
- **Protection:** JWT Required.
- **Request Body:** `multipart/form-data` with an image file under the field name `avatar`.
- **Success Response (200 OK):**

```
{
    "message": "Avatar updated successfully",
    "avatar": "string (new relative path)",
    "avatarUrl": "string (new full URL)"
}
```

- **Error Response (400 Bad Request):** If no file is uploaded or the file is not an image.

A.3 Vehicle Endpoints

Base Path: `/api/vehicles`

A.3.1 POST /

- **Description:** Adds a new vehicle for the authenticated user. Awards the 'FIRST_VEHICLE' achievement if it's the user's first vehicle.
- **Protection:** JWT Required.
- **Request Body:**

```
{
    "make": "string",
    "model": "string",
    "year": "number",
    "fuelType": "string",
    "emissionFactor": "number"
}
```

- **Success Response (201 Created):** The newly created vehicle object, including default maintenance schedules.

A.3.2 GET /

- **Description:** Retrieves all vehicles associated with the authenticated user.
- **Protection:** JWT Required.
- **Success Response (200 OK):** An array of vehicle objects.

```
[
    {
        "_id": "string",
        "owner": "string (userId)",
        "make": "string",
```

```
            "model": "string",
            "emissions": "number",
            "upcomingMaintenance": { ... },
            ...
        }
    ]
```

A.3.3 PUT /:id

- **Description:** Updates the details of a specific vehicle.
- **Protection:** JWT Required. User must own the vehicle.
- **Request Body:** A partial or full vehicle object with fields to update.
- **Success Response (200 OK):** The updated vehicle object.
- **Error Responses:** 401 Unauthorized (if not owner), 404 Not Found.

A.3.4 DELETE /:id

- **Description:** Deletes a specific vehicle.
- **Protection:** JWT Required. User must own the vehicle.
- **Success Response (200 OK):**

```
{ "message": "Vehicle removed" }
```

- **Error Responses:** 401 Unauthorized (if not owner), 404 Not Found.

A.4 Trip & recommendation endpoints

Base Path: `/api`

A.4.1 POST /trips/log

- **Description:** Logs a simulated trip. This action updates the vehicle's distance-based maintenance counters and the user's statistics, and may grant new achievements.
- **Protection:** JWT Required.
- **Request Body:**

```
{
    "vehicleId": "string (ObjectId)",
    "distance": "number"
}
```

- **Success Response (201 Created):**

```
{
    "message": "Trip logged successfully!",
    "granted": ["ACHIEVEMENT_KEY", ...]
}
```

A.4.2 POST /recommendations

- **Description:** Calculates estimated emissions for a given distance across all of the user's vehicles and returns them sorted by sustainability (lowest emissions first).
- **Protection:** JWT Required.
- **Request Body:**

```
{ "distance": "number" }
```

- **Success Response (200 OK):** A sorted array of recommendation objects.

```
[
    {
        "_id": "string", "make": "string", "model": "string",
        "emissionFactor": "number", "totalEmissions": "number"
```

```
    },
    ...
]
```

- **Error Responses:** 400 Bad Request (invalid distance), 404 Not Found (no vehicles).

A.5 Data & Logic Endpoints

Base Path: `/api`

A.5.1 GET /maintenance/summary

- **Description:** Returns a dynamic list of all upcoming or overdue maintenance tasks for the user's entire fleet, sorted by urgency (overdue items first).
- **Protection:** JWT Required.
- **Success Response (200 OK):** An array of maintenance task objects.

```
[
  {
    "id": "string (stable_id)",
    "vehicle": "string (name)",
    "taskType": "string (e.g., 'itv', 'tires')",
    "isOverdue": "boolean",
    "unit": "string ('days' or 'km')",
    "value": "number (remaining days/km)"
  },
  ...
]
```

A.5.2 GET /stats

- **Description:** Retrieves aggregated statistics for the authenticated user.
- **Protection:** JWT Required.
- **Success Response (200 OK):**

```
{
    "totalDistance": "number",
    "totalEmissions": "number",
    "tripCount": "number",
    "vehicleCount": "number",
    "averageEmissions": "number"
}
```

A.6 Development Endpoints

Base Path: `/api`

A.6.1 POST /init-db

- **Description:** A development-only endpoint to clear and seed the database with test data (a test user, vehicles with varied maintenance states, and trips).
- **Protection:** Public, but disabled in production environments.
- **Success Response (201 Created):**

  ```
  { "message": "Database initialized..." }
  ```

- **Error Response (403 Forbidden):** If `NODE_ENV` is set to 'production'.

B | Appendix: Database Schemas

This appendix details the Mongoose schemas used to structure data in the MongoDB database. Each schema corresponds to a collection and defines the shape and constraints of its documents.

B.1 User Schema (`users` collection)

Defines the structure for user accounts, including authentication details, profile information, and gamification state.

- `_id: ObjectId` *(Primary key, auto-generated by MongoDB)*
- `name: String` *(Required)*
- `email: String` *(Required, unique, indexed)*
- `password: String` *(Required. Hashed via bcrypt and never returned in queries due to `select: false`)*
- `avatar: String` *(Path to the user's profile image, with a generic default)*
- `stats: Object` *(A sub-document tracking user statistics)*
  - `distanceTraveled: Number`
  - `co2Saved: Number`
  - `totalVehicles: Number`
- `achievements: [String]` *(An array of unique keys for earned achievements, e.g., 'FIRST_TRIP')*
- `createdAt: Date` *(Auto-managed by Mongoose's timestamps option)*
- `updatedAt: Date` *(Auto-managed by Mongoose's timestamps option)*

B.2 Vehicle Schema (`vehicles` collection)

Defines the structure for user-owned vehicles, including their specifications and maintenance schedules.

- `_id: ObjectId` *(Primary key)*
- `owner: ObjectId` *(Foreign key referencing the `User` collection, required)*
- `make: String` *(Required)*
- `model: String` *(Required)*
- `year: Number` *(Required)*
- `fuelType: String` *(Required, constrained by an enum: 'gasoline', 'diesel', 'electric', 'hybrid')*
- `emissions: Number` *(CO_2 emission factor in g/km)*
- `upcomingMaintenance: Object` *(A sub-document with default maintenance intervals set upon creation)*
  - `tires: { date: Date, distance: Number }`
  - `brakes: { distance: Number }`
  - `oilChange: { distance: Number }`
  - `itv: Date`

B.3 Trip Schema (`trips` collection)

Defines the structure for a single logged journey, linking a user and a vehicle to performance metrics.

- `_id: ObjectId` *(Primary key)*
- `driver: ObjectId` *(Foreign key referencing the `User` collection, required)*
- `vehicle: ObjectId` *(Foreign key referencing the `Vehicle` collection, required)*
- `locations: Object` *(GeoJSON sub-document for location data)*
  - `start: { type: 'Point', coordinates: [Number, Number] }`
  - `end: { type: 'Point', coordinates: [Number, Number] }`

- `distance: Number` *(Total trip distance in km, required)*
- `date: Date` *(Timestamp for when the trip occurred, defaults to `Date.now`)*
- `calculatedEmissions: Number` *(Total emissions in gCO_2, required)*

C | Appendix: Mock Vehicle Specification API

To facilitate development and testing without relying on a live, and potentially rate-limited or paid, third-party vehicle data service, a mock API was created. This standalone Node.js server, located at `src/backend/mock-api/mock-car-api.js`, simulates the functionality required for the "Add Vehicle" user workflow. It provides endpoints to fetch lists of car makes and models, and to get technical specifications for a specific vehicle.

This approach provides several key advantages:

- **Decoupling:** The frontend is developed against a stable API contract, regardless of the availability or cost of an external service.
- **Offline Development:** The entire system can be run and tested without an internet connection.
- **Deterministic Testing:** End-to-end tests can rely on predictable data from the mock API, making them more robust and reliable.

The mock API runs on a separate port (7500) and provides the following endpoints.

C.1 Endpoint: GET /api/makes

- **Description:** Returns a comprehensive list of all available car makes. This is used to populate a dropdown menu in the "Add Vehicle" screen.
- **Success Response (200 OK):** An array of make objects.

```
[
    { "make_id": 1, "make": "AC" },
    { "make_id": 2, "make": "Acura" },
    ...
```

```
]
```

C.2 Endpoint: GET /api/makes/:make/models

- **Description:** Returns a list of models for a specific car make, identified by its name. This allows for dynamic, dependent dropdown menus in the UI.
- **Example Request URL:**

  ```
  http://localhost:7500/api/makes/Toyota/models
  ```

- **Success Response (200 OK):** An array of model objects corresponding to the given make.

  ```
  [
      { "model_id": 872, "model": "Camry", "make_id": 140 },
      { "model_id": 876, "model": "Corolla", "make_id": 140 },
      ...
  ]
  ```

- **Error Response (404 Not Found):** If the make name does not exist in the mock data.

C.3 Endpoint: GET /api/specs

- **Description:** This is the core endpoint for fetching technical data. It simulates retrieving the CO_2 emission factor based on the user's final vehicle selection.
- **Request Query Parameters:**

  ```
  ?make=<string>&model=<string>&year=<string>&fuelType=<string>
  ```

- **Example Request URL:**

  ```
  http://localhost:7500/api/specs?make=Toyota&model=Corolla&year=2021&fuelType=h
  ```

- **Example Success Response (200 OK):** The API uses simple internal logic to return a plausible emission factor. It also simulates a short network delay to feel more realistic.

```
{
    "emissionFactor": 98,
    "source": "OEM Hybrid Data"
}
```

- **Error Response (400 Bad Request):** If any of the required query parameters are missing.

D | Appendix: Test plans and strategy

This appendix outlines the testing strategy employed to ensure the quality, correctness, and reliability of the AIDiaCAR application. The strategy incorporates multiple levels of testing, from backend unit and integration tests to full end-to-end (E2E) user journey simulations. This section details both the tests that were implemented as part of the project's validation and those planned for future development cycles.

D.1 Backend testing (Jest & Supertest)

The backend was tested using the Jest framework, with Supertest for HTTP assertions and MongoDB Memory Server to provide a clean, isolated database for each test run.

D.1.1 Implemented backend test cases

The following table describes the key test cases that were implemented to validate the core server-side functionality.

D.1.2 Planned future backend tests

- **Input Validation Edge Cases:** Write dedicated tests for invalid inputs (e.g., short passwords, malformed emails, non-numeric trip distances) to ensure robust error handling.
- **Authorization Logic:** Add specific tests to confirm that a user cannot access or modify resources (vehicles, trips) belonging to another user.
- **Achievement Logic:** Test the achievement-granting logic more thoroughly, for example, by simulating the exact trip distance that crosses a milestone (e.g., from 999km to 1001km).

Table D.1: Implemented Backend Unit and Integration Test Cases

Test Case ID	Description	Expected Result
BE-U-01	A new user registers via the `/api/auth/register` endpoint.	A 201 status is returned, the user is created in the database, and a JWT cookie is set.
BE-U-02	An existing user attempts to log in with valid credentials.	A 200 status is returned, and a valid JWT cookie is set.
BE-I-01	An authenticated user performs a full CRUD cycle on a vehicle (Create, Read, Update, Delete).	All operations succeed with the correct HTTP status codes, and database changes are verified.
BE-I-02	An authenticated user requests their maintenance summary from `/api/maintenance/summary`.	A 200 status is returned with a correctly sorted array of upcoming and overdue tasks based on seeded data.
BE-I-03	An authenticated user requests a vehicle recommendation from `/api/recommendations`.	A 200 status is returned with a vehicle array sorted by lowest `totalEmissions`.

D.2 Frontend End-to-End testing (Playwright)

E2E tests were written using Playwright to simulate complete user workflows in a browser, validating the integration between the frontend and backend. All E2E tests utilize a `beforeEach` hook to call the backend's `/api/init-db` endpoint, ensuring each test runs with a fresh, predictable dataset.

D.2.1 Implemented E2E test cases

The following table describes the core user journeys validated with Playwright.

Table D.2: Implemented End-to-End Test Cases

Test Case ID	Description	Expected Result
E2E-01	A user successfully registers a new account and logs in.	The user is redirected from the auth pages to the main application's home screen, and the UI reflects a logged-in state.
E2E-02	An authenticated user adds, edits, and deletes a vehicle.	The vehicle list UI updates correctly after each operation. The user can navigate to the respective forms, and confirmation dialogs for deletion are handled.
E2E-03	A user attempts to log in with invalid credentials.	An error message is displayed, and the user remains on the login page without being redirected.

D.2.2 Planned future E2E tests

- **Recommendation and Trip Logging Flow:** A test to simulate a user entering a distance on the "Routes" screen, selecting the recommended vehicle, and successfully logging the trip.
- **Profile and Settings Update:** A test where a user navigates to their profile, changes their profile picture, and verifies that the new image is displayed in the UI (e.g., in the header).
- **Responsive Design Validation:** Utilize Playwright's device emulation to run the test suite on various viewports (e.g., desktop and mobile sizes) to ensure the layout remains functional.
- **Localization Check:** A test that changes the application's language in the settings and verifies that key UI elements (like page headers) are updated to the selected language.