`

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Ανάπτυξη φορητής εφαρμογής για την κοινή χρήση επιβατηγών αυτοκινήτων

Διπλωματική Εργασία

του

ΚΙΤΣΗ ΘΕΟΔΩΡΟΥ-ΙΩΑΝΝΗ

Επιβλέπων: Παναγιώτης Τσανάκας
Καθηγητής ΕΜΠ

Αθήνα, Αύγουστος, 2025

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Ανάπτυξη φορητής εφαρμογής για την κοινή χρήση επιβατηγών αυτοκινήτων

Διπλωματική Εργασία

του

ΚΙΤΣΗ ΘΕΟΔΩΡΟΥ-ΙΩΑΝΝΗ

Επιβλέπων: Παναγιώτης Τσανάκας
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 24η Αυγούστου 2025.

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Παναγιώτης Τσανάκας	Α.Σταφυλοπάτης	Β.Βεσκούκης
Καθηγητής ΕΜΠ	Ομότιμος Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα , Αύγουστος 2025

Κίτσης Θεόδωρος Ιωάννης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
24 Αυγούστου 2025

Δήλωση Μη Λογοκλοπής και Ανάληψης Ατομικής Ευθύνης

Με πλήρη επίγνωση των συνεπειών που απορρέουν από τη νομοθεσία περί πνευματικών δικαιωμάτων, δηλώνω υπεύθυνα ότι είμαι ο αποκλειστικός συγγραφέας της παρούσας διπλωματικής εργασίας. Οποιαδήποτε βοήθεια ή συνεισφορά που έλαβα κατά τη διάρκεια της εκπόνησης αναγνωρίζεται πλήρως και αναφέρεται λεπτομερώς εντός της εργασίας. Αναλαμβάνω πλήρως την προσωπική ευθύνη ότι, σε περίπτωση παραβίασης των ανωτέρω δηλώσεων, θα είμαι υπόλογος για λογοκλοπή, κάτι που συνεπάγεται την απόρριψη της διπλωματικής μου εργασίας και, κατ' επέκταση, την αδυναμία απόκτησης του τίτλου σπουδών, πέραν των λοιπών συνεπειών που προβλέπονται από τη νομοθεσία. Συνεπώς, δηλώνω ότι η παρούσα εργασία συντάχθηκε και ολοκληρώθηκε αποκλειστικά και προσωπικά από εμένα, και αποδέχομαι πλήρως κάθε ευθύνη σε περίπτωση που αποδειχθεί, σε οποιοδήποτε χρονικό σημείο, ότι τμήμα ή το σύνολο αυτής αποτελεί προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

Περίληψη

Η σύγχρονη πραγματικότητα, η οποία χαρακτηρίζεται από την εντεινόμενη αστικοποίηση, την κυκλοφοριακή συμφόρηση και την κλιματική αλλαγή, καθιστά επιτακτική την ανάγκη για υιοθέτηση πιο βιώσιμων και οικονομικών λύσεων μετακίνησης. Στο πλαίσιο αυτό, ο συνεπιβατισμός (carpooling) αναδεικνύεται ως μια πολλά υποσχόμενη πρακτική, ικανή να μειώσει το περιβαλλοντικό αποτύπωμα, το κόστος μετακίνησης και τον αριθμό των οχημάτων στους δρόμους. Η παρούσα διπλωματική εργασία εστιάζει στον σχεδιασμό και την ανάπτυξη μιας ολοκληρωμένης φορητής εφαρμογής που αποσκοπεί στην προώθηση και διευκόλυνση της κοινής χρήσης επιβατηγών αυτοκινήτων για διαδρομές μεταξύ πόλεων. Κύριος στόχος της πλατφόρμας είναι να δημιουργήσει μια αξιόπιστη ψηφιακή κοινότητα, φέρνοντας σε επαφή οδηγούς που διαθέτουν ελεύθερες θέσεις στα οχήματά τους με επιβάτες που αναζητούν ασφαλείς, φθηνές και βολικές λύσεις για τις μετακινήσεις τους. Για την υλοποίηση του συστήματος στο backend, επιλέχθηκε το **Django REST Framework**, ένα ευέλικτο και επεκτάσιμο framework της Python, το οποίο επέτρεψε την ταχεία ανάπτυξη ενός ασφαλούς και καλά τεκμηριωμένου RESTful API. Για το frontend, προτιμήθηκε η βιβλιοθήκη **React** σε συνδυασμό με την **TypeScript**, μια επιλογή που διασφαλίζει τη δημιουργία δυναμικών, αποκριτικών και συντηρήσιμων διεπαφών χρήστη (User Interfaces) χάρη στην αρχιτεκτονική των components και την ασφάλεια τύπων που προσφέρει η TypeScript. Η αρχιτεκτονική του συστήματος ολοκληρώνεται με τη χρήση μιας **σχεσιακής βάσης δεδομένων**, η οποία εγγυάται την ακεραιότητα, τη συνοχή και την αποτελεσματική διαχείριση των δεδομένων που αφορούν χρήστες, διαδρομές, κρατήσεις και αξιολογήσεις. Η εργασία είναι δομημένη ώστε να παρέχει μια σφαιρική εικόνα του έργου. Αρχικά, παρατίθεται το θεωρητικό υπόβαθρο και γίνεται επισκόπηση των υφιστάμενων λύσεων στον χώρο του συνεπιβατισμού. Στη συνέχεια, αναλύονται εκτενώς οι τεχνολογικές επιλογές και παρουσιάζεται η αρχιτεκτονική του συστήματος, συμπεριλαμβανομένου του σχεδιασμού της βάσης δεδομένων και των endpoints του API. Ακολουθεί η λεπτομερής περιγραφή της υλοποίησης των βασικών λειτουργικοτήτων. Τέλος, παρατίθενται τα αποτελέσματα των δοκιμών ελέγχου (unit, integration, and user acceptance tests) που διενεργήθηκαν για την επαλήθευση της ορθής λειτουργίας, της απόδοσης και της ασφάλειας της εφαρμογής, επιβεβαιώνοντας την επιτυχή ολοκλήρωση των αρχικών στόχων του εγχειρήματος.

Λέξεις-κλειδιά — carpooling, mobile app, Django REST Framework, React, TypeScript

Abstract

The modern reality, characterized by intensifying urbanization, traffic congestion, and climate change, makes the adoption of more sustainable and economical transportation solutions imperative. In this context, carpooling emerges as a highly promising practice, capable of reducing the environmental footprint, travel costs, and the number of vehicles on the road. This thesis focuses on the design and development of a comprehensive mobile application aimed at promoting and facilitating the sharing of passenger cars for inter-city journeys. The main objective of the platform is to create a reliable and user-friendly digital community, connecting drivers with available seats in their vehicles with passengers seeking safe, affordable, and convenient travel solutions.

To implement the system, a modern technological approach was adopted, based on established and robust software development tools. For the backend, **Django REST Framework**, a flexible and scalable Python framework, was chosen, which allowed for the rapid development of a secure and well-documented RESTful API. For the frontend, the **React** library was selected in combination with **TypeScript**, a choice that ensures the creation of dynamic, responsive, and maintainable User Interfaces, thanks to its component-based architecture and the type safety offered by TypeScript. The system's architecture is completed by the use of a **relational database**, which guarantees the integrity, consistency, and efficient management of data concerning users, routes, bookings, and reviews.

The thesis is structured to provide a holistic view of the project, from its theoretical foundation to its practical implementation and final evaluation. Initially, the theoretical background is presented, along with a review of existing solutions in the carpooling domain. Subsequently, the technological choices are analyzed in detail, and the system's architecture is presented, including the database schema and API endpoints. This is followed by a detailed description of the implementation of core functionalities, such as user registration and authentication, route creation and search, the booking system, and the mutual rating system for drivers and passengers. Finally, the results of the control tests (unit, integration, and user acceptance tests) that were conducted to verify the application's correct functionality, performance, and security are presented, confirming the successful achievement of the project's initial objectives.

Keywords — carpooling, mobile app, Django REST Framework, React, TypeScript

Table of Contents

Chapter 5 : Development Reflections and Engineering Insights

5.1 - Development Challenges
5.2 - Summary of Lessons Learned and Developer Growth
5.3 - Overall Reflection

Chapter 6: Code Technical Specifications

6.1 - Introduction
6.2 - Frontend in detail
6.3 - Backend in detail

Εκτενής Περίληψη

Η παρούσα διπλωματική εργασία αποτελεί μια ολοκληρωμένη μελέτη, σχεδίαση, υλοποίηση και αποτίμηση μιας σύγχρονης εφαρμογής συνεπιβατισμού (carpooling) για διαδρομές μεταξύ πόλεων. Στόχος της είναι να αντιμετωπίσει τις σύγχρονες προκλήσεις της αστικής και περιαστικής μετακίνησης —όπως η κυκλοφοριακή συμφόρηση, το υψηλό κόστος καυσίμων και το αυξανόμενο περιβαλλοντικό αποτύπωμα— μέσω της δημιουργίας μιας αξιόπιστης και εύχρηστης ψηφιακής πλατφόρμας.

Το **Κεφάλαιο 1** θέτει τα θεμέλια του εγχειρήματος. Αρχικά, αναλύει το πρόβλημα (**1.1**) στις σύγχρονες κοινωνικοοικονομικές του διαστάσεις. Στη συνέχεια, παρουσιάζει τον συνεπιβατισμό (**1.2**) ως μια πρακτική με πολλαπλά οφέλη, οικονομικά, κοινωνικά και οικολογικά. Το κεφάλαιο εξειδικεύει τους στόχους της εφαρμογής (**1.3**), οι οποίοι περιλαμβάνουν τη δημιουργία ασφαλών προφίλ χρηστών, την παροχή ενός ευέλικτου συστήματος αναζήτησης και δημοσίευσης διαδρομών, την ενσωμάτωση ενός μηχανισμού κρατήσεων σε πραγματικό χρόνο, και την ανάπτυξη ενός συστήματος αμφίδρομων αξιολογήσεων για την ενίσχυση της εμπιστοσύνης εντός της κοινότητας. Τέλος, περιγράφεται η βασική λειτουργικότητα (**1.4**), σκιαγραφώντας τη ροή αλληλεπίδρασης μεταξύ οδηγών και επιβατών.

Το **Κεφάλαιο 2** παρέχει τη θεωρητική και τεχνολογική τεκμηρίωση του έργου. Εμβαθύνει στις αρχές της κοινής κινητικότητας (shared mobility) και της οικονομίας του διαμοιρασμού (sharing economy) (**2.1**). Αναλύονται διεξοδικά οι τεχνολογίες που επιλέχθηκαν (**2.2**): για το backend, το **Django REST Framework** για την ταχύτητα ανάπτυξης, την ενσωματωμένη ασφάλεια και την επεκτασιμότητά του· για το frontend, η βιβλιοθήκη **React** για τη δημιουργία δυναμικών διεπαφών μέσω της αρχιτεκτονικής components και η **TypeScript** για την προσθήκη στατικής τυποποίησης, που διασφαλίζει τη στιβαρότητα και τη συντηρησιμότητα του κώδικα. Επιπλέον, εξετάζονται οι θεμελιώδεις αρχές σχεδιασμού λογισμικού (π.χ., SOLID, DRY) που υιοθετήθηκαν για τη διασφάλιση ενός καθαρού και αρθρωτού κώδικα (**2.3**).

Το **Κεφάλαιο 3** είναι αφιερωμένο στην εις βάθος ανάλυση του **Frontend**. Παρουσιάζεται η συνολική αρχιτεκτονική τύπου **Single Page Application (SPA)** (**3.1**). Αναλύεται ο σχεδιασμός των βασικών components (**3.2**), όπως η φόρμα αναζήτησης, η κάρτα εμφάνισης διαδρομής και το προφίλ χρήστη, παραθέτοντας και παραδείγματα κώδικα. Εξηγούνται οι τεχνικές υλοποίησης της δρομολόγησης (routing) με τη χρήση της βιβλιοθήκης `React Router`, η

στρατηγική διαχείρισης της κατάστασης (state management) για τον συγχρονισμό των δεδομένων σε ολόκληρη την εφαρμογή (π.χ., με Context API ή Redux), και η ασύγχρονη επικοινωνία με το backend API (**3.3**). Το κεφάλαιο κορυφώνεται με την παρουσίαση ολοκληρωμένων σεναρίων χρήσης (user journeys), από την εγγραφή και την αναζήτηση μέχρι την κράτηση και την αξιολόγηση μιας διαδρομής (**3.4**).

Το **Κεφάλαιο 4** εστιάζει στην αρχιτεκτονική και την υλοποίηση του **Backend**. Περιγράφεται η αρθρωτή δομή του project σε Django "apps" (π.χ., users, rides, bookings) (**4.1**). Ορίζονται τα **Models** της βάσης δεδομένων (**4.2**), όπως το User, το Ride, και το Booking, μαζί με τις μεταξύ τους σχέσεις (Foreign Keys, Many-to-Many). Επεξηγείται ο ρόλος των **Serializers** (**4.3**) στη μετατροπή των Python objects σε JSON και στην επικύρωση των εισερχόμενων δεδομένων. Αναλύεται η επιχειρησιακή λογική που ενσωματώνεται στα **Views** (**4.4**), τα οποία εκθέτουν τα endpoints του RESTful API (π.χ., GET /api/rides/, POST /api/bookings/). Ιδιαίτερη έμφαση δίνεται στους μηχανισμούς ασφαλείας, όπως το σύστημα **αυθεντικοποίησης** βασισμένο σε tokens (π.χ., JWT) και οι κλάσεις **Permissions** που ορίζουν τα δικαιώματα πρόσβασης των χρηστών στους πόρους του API (**4.5**, **4.6**).

Το **Κεφάλαιο 5** προσδίδει μια διάσταση κριτικού αναστοχασμού. Καταγράφει τις τεχνικές **προκλήσεις** που αντιμετωπίστηκαν (**5.1**), όπως η διαχείριση ζωνών ώρας ή η επίλυση ζητημάτων CORS μεταξύ frontend και backend. Συνοψίζει τα πολύτιμα **διδάγματα** και την **εξέλιξη του προγραμματιστή** (**5.2**) μέσα από την επίλυση προβλημάτων και την εφαρμογή νέων τεχνολογιών. Το κεφάλαιο κλείνει με έναν συνολικό **απολογισμό** του έργου, αξιολογώντας το τελικό αποτέλεσμα σε σχέση με τους αρχικούς στόχους (**5.3**).

Τέλος, το **Κεφάλαιο 6** λειτουργεί ως ένα πλήρες τεχνικό παράρτημα. Παρέχει τις **λεπτομερείς προδιαγραφές του κώδικα**, τόσο για το **Frontend** (**6.2**) όσο και για το **Backend** (**6.3**). Αυτό περιλαμβάνει την τεκμηρίωση των API endpoints (διαδρομές, μέθοδοι HTTP, αναμενόμενα payloads) και μια ανάλυση των κυριότερων components του React και των props τους, καθιστώντας τον κώδικα κατανοητό, επεκτάσιμο και συντηρήσιμο.

Chapter 1: Introduction

The demand for sustainable, efficient, and affordable transportation solutions is a pressing challenge in modern society. Increasing traffic congestion, rising fuel prices, environmental concerns, and limited access to flexible transportation options make it necessary to explore alternative mobility strategies. Carpooling — the practice of sharing a vehicle among multiple passengers traveling in the same direction — has emerged as one of the most promising solutions to address these issues.

This thesis focuses on the design, development, and evaluation of a mobile application for passenger carpooling. The application allows drivers to offer available seats on their trips and enables passengers to search for and book these seats for intercity travel. By facilitating organized carpooling, the application aims to reduce transportation costs, improve travel accessibility, and promote environmentally responsible mobility.

In recent years, technological advancements and the widespread use of smartphones have created new opportunities for implementing and scaling carpooling services. Mobile applications provide a user-friendly platform that enables real-time ride sharing, secure communication between users, and efficient coordination of trips. By leveraging geolocation, data analytics, and modern UI/UX practices, such platforms can enhance user trust and streamline the process of matching drivers with potential passengers. This thesis aims to harness these capabilities to deliver a reliable and scalable solution that supports the growing demand for smarter urban and intercity transportation.

1.1 The Issue

There are several lingering problems in intercity transportation. The high number of individual car owners results in a significant percentage of empty seats, with most vehicles carrying only the driver. This situation is a key contributor to the growing traffic congestion, excessive fuel consumption, and the consequent increase in carbon emissions. On the other hand, public transportation alternatives might not cater to the specific needs of all passengers in terms of scheduling, convenience, and frequency of service, especially on the less popular routes.

Carpooling can be a way to solve these problems by connecting drivers who have vacant seats with people going the same way. However, unofficial carpooling arrangements are challenged by several factors: problems with finding a suitable match, lack of organization tools, and trust issues among participants. A specially designed mobile platform can be a solution to these problems by providing organized matching features, secure user accounts, and an embedded feedback system.

1.2 Carpooling Overview

Carpooling means using a private car by the driver and one or more passengers. Besides the ecological and money-saving aspects, carpooling has social benefits as well, as it helps people to communicate and establishes a shared sense of responsibility for reducing the negative impact of traffic.

The carpooling applications of today have revolutionized the old carpooling concept into a new, digital version. With these apps, the drivers are able to publish offers for their trips, specifying the starting point, destination, and time of the journey, the number of available seats, and any other relevant data. The passengers, in turn, can check the available trips, set filters, make bookings, and communicate with the drivers. To ensure the security of the participants and to foster trust among them, such systems usually employ user verification, profile management tools, and review functions.

1.3 Application Goals and Features

The following features are proposed to be implemented in the mobile application for this thesis: User Sign Up and User Sign In: Protected registration and login, creation of a profile and sign in/log out of data in a user's space.

Driver Module: Drivers can offer trips by adding sets of information such as specific routes, the number of available seats, and time slots. Passenger Module- To let your passengers able to search for rides, filter by date and location, then reserve seat.

Vehicle Management: Offers drivers the capability to input vehicle information like make, model, year, license plate, and seating.

Review and Ratings: This feature utilizes the feedback and ratings component, so users can share their experience and makes the platform trustworthy.

This first chapter on structures the chapters to come, in range of background literature, system design and implementation, and reaches a concluding Application. Overall, our endeavor is to provide a functional and scalable answer to the nuance of shared mobility and the promise it holds for long distance carpooling.

1.4 Application Functionality

The app operates around the aim of connecting drivers and riders as efficiently as possible. Like all gig-economy jobs, drivers sign on and enter their car's specifications. They will then be able to post available trips and include the route, departure and arrival times and number of seats available. Riders, meanwhile, can find a ride by browsing using a number of filters (like date, location, or how many seats are available), where they can see they can–see the detail of a trip before making a booking. Once there is a match, the system manages also the process for booking so that drivers and passengers can contact each other confidentially. Once the trip is over, users can review and rate one another, which become part of the platform's reputation system, creating trust for future transactions. It also has automated reminders like reminding people of their next trips and a dashboard for users to keep track of and manage trips, bookings, and feedback. This chapter forms the basis of subsequent sections, which will detail some of the background research, system design, development methodology, and performance evaluation. The aim will be to provide a practical, scalable and user centric solution to stimulate intercity carpooling and sustainable mobility.

Chapter 2: Theoretical Background and Technological Foundations

This chapter introduces the theory and technology based techniques for the development of carpooling mobile application as elaborated in this thesis. It explains the car pooling systems and the social and environmental justifications for shared mobility before discussing the architectural frameworks, programming languages, and design tools used in production, and namely the Shadcn component library.

2.1 Carpooling and the Shared Mobility Principles

Carpool is a mode of travel in which organizes people use private vehicles when travelling close to each other. The system is intended to promote ridesharing (reducing the number of vehicles on the road), and more efficient use of transportation capacity. The concept of carpooling and shared mobility are based on sustainability, efficiency, and community involvement.

Eco Friendliness: Carpooling greatly reduces the CO_2 emissions, fuel consumption and air pollution. Research demonstrates that increasing the average vehicle occupancy could reduce the carbon emission rate per capita by a significant gap, which can help to reach the global climate targets while improving the urban air quality problems.

Cost Savings: By sharing direct travel expenses (fuel, tolls, and vehicle maintenance) among a larger group of passengers, the overall cost per person drops for drivers and passengers. This sharing-of-cost element is ideally suited for students, commuters, and those with restricted travel budgets.

Reduced congestion: Raising the average occupancy of cars reduces the total number of cars on the roads, alleviating congestion, shortens travel time and makes transportation networks more efficient at peak times. Community and Social You and Me Benefits: Besides eco and economic advantages, carpooling promotes social behavior, and forms the basis of community for everyone providing and needing travel. It has potential to enhance network and cooperation and enhance social networking.

Resource Utilisation: A city-level approach will be able to more efficiently utilise the available-infra such that it is not over-exerted, relieving pressure from public transportation systems, the roads and the parking spaces. However, research

emphasises potential barriers to the introduction of the programme, including schedule matching, safety concerns, and trust between strangers. To overcome these challenges and fully harness the benefits of carpooling, the magic of digital applications unfold. Apps like these modernize the process of identifying, booking, and assembling shared rides. To achieve this, they use such functionality as real-time changes, automatic alerts, user library, scorecards, user rankings and feedback to improve visibility as well as trust and experience. This work is based on these ideas and intends to provide a sound, scalable implementation which is friendly to the environment and convenient for future users.

2.2 Technological Foundations of the Application

The technological stack chosen for this application is carefully designed to balance performance, scalability, security, and maintainability:

- **Backend (Django REST Framework):** DRF provides a structured framework for creating RESTful APIs in Python. It handles serialization (converting database objects to JSON and back), authentication, and request handling. By using DRF, the backend can implement robust APIs that support complex data relationships, manage user sessions securely, and integrate permission systems to control access.

- **Frontend (React + TypeScript):** React enables the construction of interactive UIs through reusable components, while TypeScript adds type safety, making the codebase more predictable and easier to debug. Together, they improve maintainability and reduce runtime errors. React's ecosystem also offers a rich set of libraries for state management, routing, and hooks, enhancing development speed.

- **Tailwind CSS for Styling:** Tailwind CSS brings a utility-first approach to styling. Instead of writing custom CSS, developers apply predefined utility classes directly in the markup, creating highly customizable and responsive layouts. Tailwind's flexibility allows for theming and consistent design across the entire app.

- **Shadcn Component Library:** Shadcn offers high-quality, customizable React components such as drawers, modals, forms, buttons, and dropdowns, saving developers time and ensuring visual consistency. These components integrate seamlessly with Tailwind CSS, enabling a polished, professional UI without the need to design every element from scratch.

- **Relational Database (PostgreSQL/MySQL/SQLite):** The system uses a relational database to store structured data — including user profiles, vehicle details, ride listings, passenger bookings, and reviews. Django's ORM (Object-Relational Mapping) provides an abstraction layer, making it easier to query, update, and migrate data without writing raw SQL.

- **API Communication:** The frontend and backend communicate over HTTP using RESTful API principles. This approach ensures a clean separation of concerns, with the frontend focused on user interaction and the backend responsible for data processing and business logic.

- **Development Workflow and Tools:** Modern tools such as Git (for version control), GitHub (for collaboration), and Visual Studio Code (for coding) are essential for efficient teamwork, code management, and debugging. These tools also support continuous integration practices, allowing for rapid deployment and testing.

Django

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It provides built-in features for authentication, database management, and security, making it an excellent choice for building robust backend systems.

Django REST Framework (DRF) builds on Django, adding a layer for creating RESTful APIs. DRF simplifies serialization, viewsets, authentication, and permission handling, letting us create a powerful API backend efficiently.

Key advantages

- Built-in admin interface.
- Secure and scalable.
- Strong community support and extensive documentation.
- Excellent integration with databases like PostgreSQL.

React and TypeScript

React is a JavaScript library for building interactive user interfaces. Its component-based architecture enables modular, reusable UI elements, speeding up development and improving maintainability.

TypeScript extends JavaScript by adding static typing, which reduces runtime errors and improves developer productivity. By combining React with TypeScript, we ensure a safer and more scalable frontend codebase.

Key advantages

- Declarative and flexible UI building.
- Strong ecosystem with thousands of libraries.
- Enhanced code safety with TypeScript's type checking.
- Easier refactoring and collaboration in larger teams.

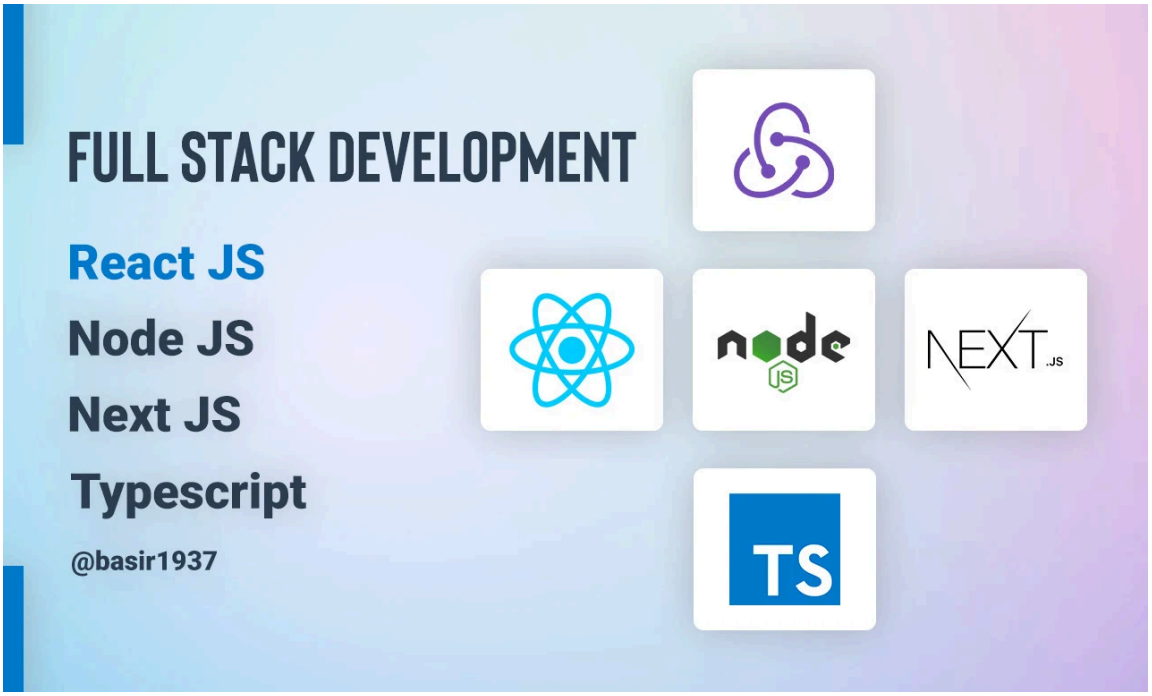
Tailwind CSS and Shadcn UI

Tailwind CSS is a utility-first CSS framework that allows rapid, responsive design without writing custom styles. By using predefined utility classes, developers can quickly compose complex layouts.

Shadcn UI provides a set of accessible, beautifully designed React components (like buttons, drawers, modals) that integrate smoothly with Tailwind. This combination accelerates UI development while maintaining design consistency.

Key advantages

- Rapid prototyping and iteration.
- Fully customizable styles.
- Built-in accessibility features.
- Seamless integration with React projects.



2.3 Software Design Principles

Principles of Software Design Technology is at the center of this project, but it is for people — the students, drivers and ride customers who will use the app. That's why the software design principles go above and beyond simple technical considerations – they represent a fundamental investment in crafting something meaningful, dependable, and human.

Separation of Concerns: We separate the system into 3 obvious pieces — backend, frontend, and database — so that each can evolve without taking down the rest. It's why maintenance is now easier, and why the team can concentrate on making one piece better at a time.

Scalability: Platform needs to scale up effortlessly while more and more users are moving onto the platform. Here, the modular design means that we can add new features (like notifications or payment systems) without having to re-architect the entire application. Scalability here is not just a matter of size — it is about keeping the user experience smooth, even at large scales.

Security: We care about the trust users place in us. All the sensitive information, such as passwords and personal information, are taken care of securely using encryption and authentication. The app was created "specifically to defend against ordinary cyber threats" and "to give people peace of mind when sharing rides with others."

User-Friendly Layout: The app is laid out to be intuitive, particularly for those who will access it on mobile devices the most. By adhering to established UI/UX patterns as well as to Shadcn's beautiful and consistent components, we ensure the experience is intuitive and enjoyable — because people should notice the journey but don't fight the app.

Maintainability: Code is not only for machines, but for humans — developers of the future that will read, improve, and extend it. Ultimately using TypeScript and Django's patterned structure for clarity keeps the system easy to understand and maintain as it grows. Unit tests and code reviews keep the quality high and the WTF/min low.

Extensibility: There is a lot of future in the system. New features, integrations, or enhancements can be built upon without rebuilding everything from scratch, because of the modular backend and frontend architecture. In other words, our design principles seek a balance between technical power and human empathy, and hope this balance results in not just powerful and efficient but also friendly and trustworthy end product for all users.

2.4 Summary

In conclusion, Chapter 2 set the stage to understand the dual foundation of this project: the human reasons that under the need of a carpooling application and the technical decisions to make it real. We examined the environmental, economic, and social impact of shared mobility, and the modular, nested architecture that forms the app's foundation. We underscored the ways in which every design decision — from the infrastructure to the tiniest button in the frontend — is underpinned by a dedication to technical fortitude and human empathy. This set of tools acts as an operational model, guaranteed to make the application practical, relevant and viable. In the following chapter, we further explore how these systems work together, profiling the architecture of the full system along with diagrams and real-world examples. the same seat. It's a space in which technical precision can mean the difference between a user experience that feels stable and trustworthy and one that doesn't — because when it comes to sharing a car with strangers, trust is everything.

Chapter 3: Frontend System Design and Implementation

When you load the carpooling app, what you see — the screens, buttons, forms and interactions — is the result of hundreds of design and development choices. And the frontend is where the UX magic really happens. It is the layer that transforms raw data and complex backend logic into something meaningful, beautiful and easy to use. In this chapter, you'll get acquainted with how we built the frontend, not as code, but as a human-centered system that fit the needs of drivers and passengers. We'll take a look at the high level architecture, real pieces of code, reasons why we made specific technology choices and touch on the user experience decisions that guided the final result. We don't just want to throw technical terms at you, we want to keep you entertained, tell you the story of the frontend: why React and TypeScript, and of course, how did Shadcn and Tailwind CSS give us a one heck beautiful and consistent design system, how did we structured the app in simple modular and reusable components and why every click, swipe, or tap you made felt smooth and chill, Because feeling reliable isn't just sex, it's good sh*t! In this chapter, we will illustrate with some real code snippets of the project, the decisions that were made and why, during this choice. Let's just not only explain the frontend of the system, but provide you insight how to think about building web applications in a modern, maintainable and user-friendly way!

3.1 Overall Frontend Architecture

The frontend of the Carpool Application was developed with a singular objective: To offer an intuitive, seamless user experience, while significantly reduced the response time. At its core, there are some main principles making it: modular, maintainable, scalable, and user oriented. We chose React as our main framework due to its flexible component-based architecture. With React, we can decompose the user interface into small, composable building blocks — components like buttons, ride cards, booking forms, or profile drawers. For instance, in our PastBookings. tsx component we have created a reusable Card component that can be used to show old rides in a consistent, good looking way:

```
<Card key={ride.id} className="min-w-[250px] flex-shrink-0 shadow-md border rounded-lg">
    <CardHeader className="flex flex-col items-center text-center">
    <CardTitle className="text-lg font-semibold">
        {ride.start_location} → {ride.end_location}
    </CardTitle>
    <p className="text-sm text-gray-500">
        {new Date(ride.departure_time).toLocaleDateString()}{" "}
        |{" "}
        {new Date(ride.departure_time).toLocaleTimeString([], { hour: '2-digit', minute: '2-digit' })}
    </p>
    </CardHeader>
```

We started with React and then added TypeScript on top to provide strong typing and clarity in our code. We may define TypeScript interfaces like these, to keep a data structure consistent:

```typescript
export interface Ride {
    id: number;
    vehicle: number;
    start_location: string;
    end_location: string;
    date: string;
    departure_time: string;
    arrival_time: string;
    driver: number;
    passengers: number[];
    available_seats: number;
    created_by: number;
}
```

For styling, we used Tailwind CSS paired with Shadcn, a modern React components library. With Tailwind's utility-first approach we have the freedom to style elements in the markup, ensuring that our design tokens stay fast and flexible. Shadcn adds to this with pre-styled UI components – drawers, modals, tooltips and the like, ready to drop into your app's visual identity. I.e., our drawer elements that are in pastBookings. tsx look like this:

```tsx
<Drawer open={isOpen} onOpenChange={(open) => { setIsOpen(open); if (!open) setSelectedRide(null); }}>
  <DrawerContent>
    <div className="mx-auto w-full max-w-sm">
      <DrawerHeader>
        <DrawerTitle>
          Review the rides' users
        </DrawerTitle>
        <DrawerDescription>It will be anonymous</DrawerDescription>
      </DrawerHeader>
      <ReviewForm ride={selectedRide} userId={userData?.id} closeDrawer={() => { setIsOpen(false); setS
    </div>
  </DrawerContent>
</Drawer>
```

Structurally, the frontend has a number of nice layers: Components Layer: All visual objects with which we construct UI, starting from simple buttons up to principally sized layout containers.

Pages and Views Layer: Primary application screens such as the home screen, search results, ride details, booking process and user profile. State Management Layer – For local state we are using hooks in React (such as useState and useEffect) and context to share state (for example, whether the user is logged in).

API Integration Layer: Makes requests to the backend using fetch() or by using libraries, such as Axios, to bring in ride, user, and booking data.

Routing Layer: Manages page navigation via React Router, noting that transitions between pages are seamless with no full reload. This architecture was not just a technical strategy for the app, but also an experience we wanted for our users — one in which actions feel immediate and the interface feels clean and coherent, where every interaction — from searching for a ride to booking a seat — feels smooth and reliable. The rest of this guide dissects such specific pieces, demonstrates additional use-cases and explains how each piece helps in creating a frontend system that is not only technically sound but deeply user-oriented.

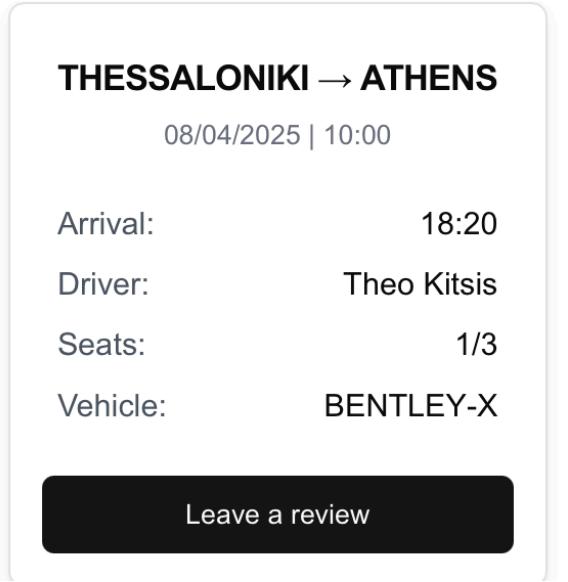
3.2 Component Design and Code Examples

In this chapter, we further explain the design and implementation details of some of the frontend components of the carpooling application. One benefit of React's modular approach is that it lets us divide our UI into little, reusable bits, and each one is responsible for a single function or visual. One of the advantages of this approach is that it is reusable. For instance, the Button component from Shadcn is applied across pages seamlessly, be it for form submits, drawer opens, or other actions — promoting visual conformance and behavioral expectations.

Example: Ride Card Component Search results and past and upcoming bookings use the ride cards; they're an implementation of the shared Card component. Here's an example code snippet:

```
<Card key={ride.id} className="min-w-[250px] flex-shrink-0 shadow-md border rounded-lg">
    <CardHeader className="flex flex-col items-center text-center">
    <CardTitle className="text-lg font-semibold">
        {ride.start_location} → {ride.end_location}
    </CardTitle>
    <p className="text-sm text-gray-500">
        {new Date(ride.departure_time).toLocaleDateString()}{" "}
        |{" "}
        {new Date(ride.departure_time).toLocaleTimeString([], { hour: '2-digit', minute: '2-digit' })}
    </p>
    </CardHeader>
    <CardContent className="space-y-2">
    <div className="flex justify-between items-center">
        <span className="font-medium text-gray-600">Arrival:</span>
        <span className="text-base">
        {new Date(ride.arrival_time).toLocaleTimeString([], { hour: '2-digit', minute: '2-digit' })}
        </span>
```

Card

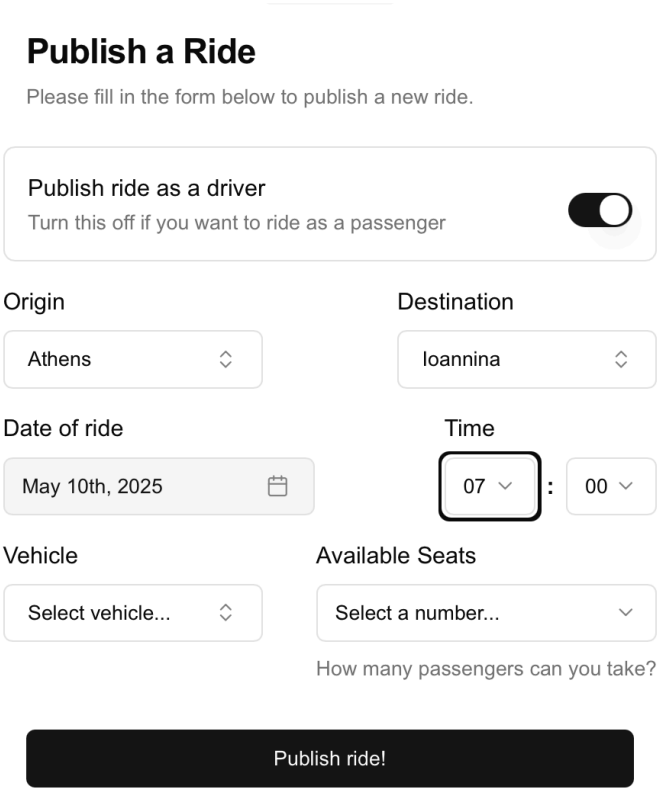


Example: TimePicker Component TimePicker It is used to select ride departure/arrival times. The following is an example of this that uses the React State and, as such, uses controlled inputs:

```tsx
interface TimePickerProps {
  value: string;
  onChange: (value: string) => void;
  minuteStep?: number;
}

export const TimePicker: React.FC<TimePickerProps> = ({
  value,
  onChange,
  minuteStep = 5,
}) => {

  const [hour, setHour] = React.useState<string>("00");
  const [minute, setMinute] = React.useState<string>("00");

  React.useEffect(() => {
    if (value) {
      const [h, m] = value.split(":");
      setHour(h);
      setMinute(m);
    }
  }, [value]);
```

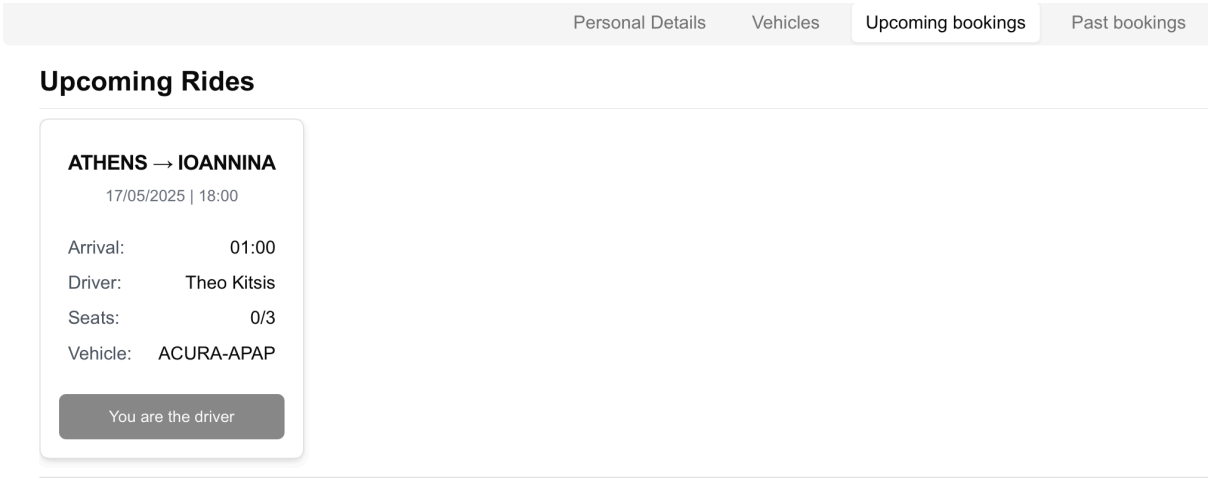
```tsx
React.useEffect(() => {
  onChange(`${hour}:${minute}`);
}, [hour, minute, onChange]);

const hours = Array.from({ length: 24 }, (_, i) =>
  i < 10 ? `0${i}` : `${i}`
);

const minutes = Array.from({ length: Math.ceil(60 / minuteStep) }, (_, i) => {
  const m = i * minuteStep;
  return m < 10 ? `0${m}` : `${m}`;
});
```

Briefly, the “Publish a Ride” form offers to drivers a user-friendly tool that will allow easy formulation and immediate sharing of new ride offers with prospective passengers. There is a toggle at the top for the user to indicate if they are posting the ride as a Driver or a Passenger. The form will gather all info related to the trip, such as departure and arrival addresses, departure date and time, selected vehicle and number of total seats available. Dropdowns, pickers Date picker and time picker Material Design time picker and date picker help to prevent errors Provides a more intuitive user experience by enabling faster entry. After filling out the form, you can click the "Publish ride! button, you can make the ride available for reservation. This feature is at the core of the app’s mission to create efficient, eco-friendly and organized shared transportation.



This piece kind of demonstrates where we start meshing UI elements with logic to bring everything together in a smooth, easy to understand experience. All over the frontend, this component-based system means we can scale the app, add features, and keep things consistent. In next article we will see how these things work together in an real-application with state management and api integration.

Upcoming Bookings



The Upcoming Bookings interface is a critical piece of the application's frontend architecture and is important because it will help improve user experience overall. This component shows a high-level view of all the scheduled rides for the logged-in user.

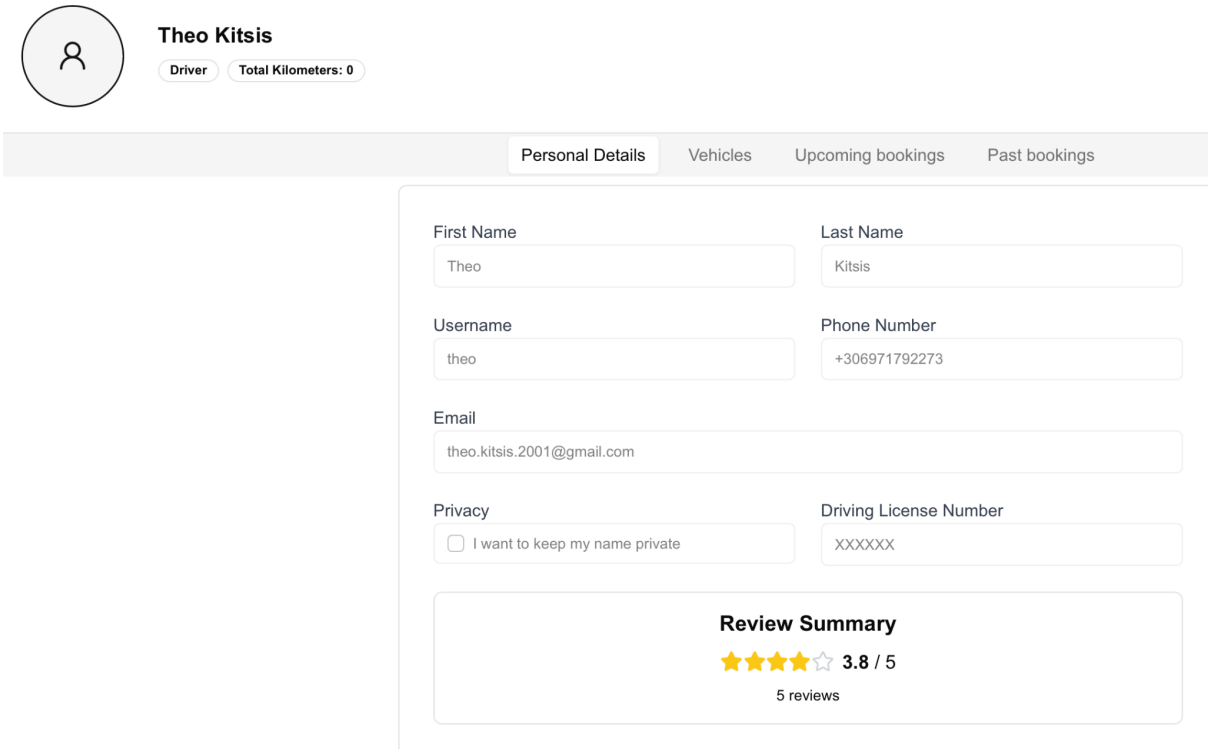
The layout is based on cards – great for an instant visual identification where each ride is separate, and provides all necessary information: journey (e.g. Athens → Ioannina), departure date, time and expected arrival time, driver's name, available seats, and a type of car. A status text like "You are the driver" is dynamically displayed when a user is the creator of the ride and thus, the knowledge and use of the system would increase. Such a format is not only aesthetically tidy but is also operationally efficient, focusing on ease of use and instant access to information. Technically, the component is receiving booking data as a JSONAPI response from the backend and rendering that using React's rendering, such as. map() method in order to process the ride entries. Conditional rendering is used to display alternative views based on user role (driver or passenger) and booking state.

This implementation also encourages reusability and maintainability, since we enclose each ride card inside of its own subcomponent. Athough they are sleek and responsive on tablets and desktop, the theme still maintains super quick load times on smaller screens and has been designed with 'touch' in mind. This is a great example of what we're aiming for with the application's interface strategy: straightforward frontend logic, and a clean UI design.

An example of the code for Upcoming Bookings :

```
<div className="flex space-x-4 overflow-x-auto pb-2 min-h-[200px]">
  {upcomingRides !== undefined && upcomingRides.length === 0 && (
      <div className="flex justify-center items-center w-full min-h-[200px] text-xl">
          No upcoming rides found.
      </div>
  )}
  {upcomingRides !== undefined && upcomingRides.length > 0 && upcomingRides.map((ride) => (
  <Card key={ride.id} className="min-w-[250px] flex-shrink-0 shadow-md border rounded-lg">
      <CardHeader className="flex flex-col items-center text-center">
      <CardTitle className="text-lg font-semibold">
          {ride.start_location} → {ride.end_location}
      </CardTitle>
      <p className="text-sm text-gray-500">
          {new Date(ride.departure_time).toLocaleDateString()}{" "}
          |{" "}
          {new Date(ride.departure_time).toLocaleTimeString([], { hour: '2-digit', minute: '2-digit' })}
      </p>
      </CardHeader>
      <CardContent className="space-y-2">
      <div className="flex justify-between items-center">
          <span className="font-medium text-gray-600">Arrival:</span>
          <span className="text-base">
          {new Date(ride.arrival_time).toLocaleTimeString([], { hour: '2-digit', minute: '2-digit' })}
          </span>
      </div>
```

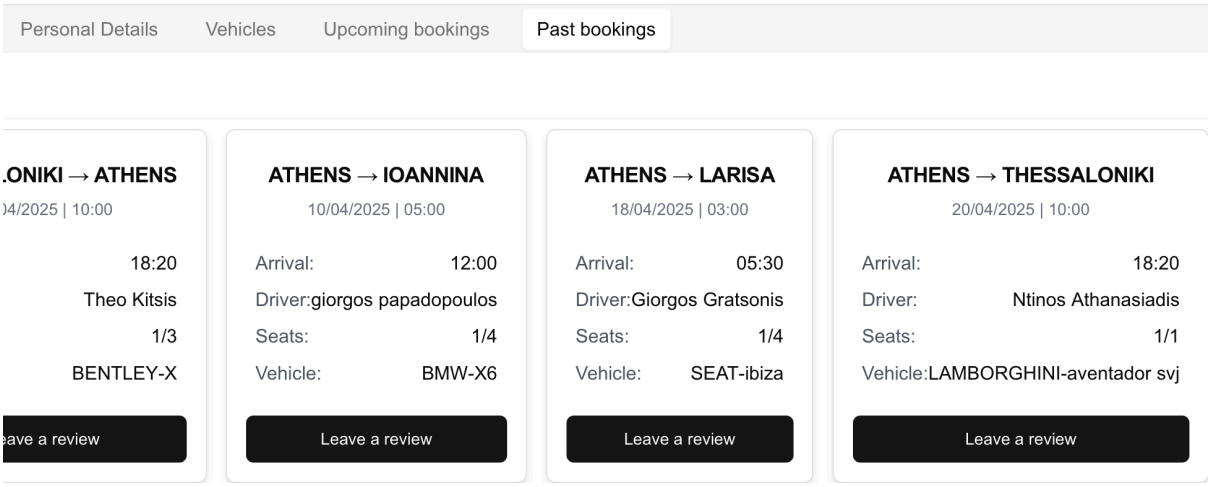
Personal Details



On the driver profile of Theo Kitsis, the Personal Details tab gives an organized overview of Theo's most important user info, such as first name, last name, username, phone number, email and a filled with love just for his driving license number. The interface is pretty neat and easy to use with the driver able to easily access his/her personal information as well as update it. And there's a privacy checkbox so the author can be private if he/she wants, allowing more control over personal visibility.

The thing that makes this tab unique is it has a section for Review Summary too which serves as a proof and authority. Theo overall rating 3.8 of 5 5 reviews The brief is star rated and quantitized, providing quick-witted feedback for other users or potential passengers somehow. Displaying members' reviews on their public profile page provides users with more transparency and confidence in the platform this is really important in ride share environments. This makes the profile more than just a static page and more of a living, breathing view of a driver's activity and reputation.

Past Bookings



Past Bookings' gives users a structured view of their past rides. Here, every past ride is presented in a card view which includes details such as route, date/time of departure, arrival time, driver's, report time, no of seats occupied and the car used.

This is not only a trip history log, but it also includes a 'user touch' via the "Leave a review" button which allows the passenger to give feedback. This feature promotes transparency and trust in the platform by promoting community-based rating. It is styled to match the rest of the

interface, and adds to a smooth and informative user experience—one that doesn't need you to rack your brain or dive into deep historical stats to remember or reproduce an old ride.

3.3 Routing, State Management and API Integration

Components make up the appearance of the app, but the power of the app is in the way they work together — the way they handle data, respond to user input, and interact with the backend. This part dives deep into routing, state management, and API integration, with real code taken from the project—this part explains how it all fits together under the hood.

Routing with React Router React Router is used for navigation so I can keep pages from having to force full page reloads. For instance, when a user clicks on a ride card in the SearchResults. tsx file, the app is dynamically pushing them to /rides/:id with useParams:

```
// See this in SearchResults.tsx
import { useParams } from 'react-router-dom';

function RideDetail() {
  const { id } = useParams();

  useEffect(() => {
    fetchRideDetails(id);
  }, [id]);

  return <div>/* Ride details */</div>;
}
```

In the project, nested routes organize subpages (e.g., showing booking details inside a ride page) and protected routes restrict access to pages like the user profile.

State Management with React Hooks and Context

The app relies on React's useState and useEffect hooks, as seen in components like UpcomingBookings.tsx, where we manage the loading state:

```
// From UpcomingBookings.tsx
const [loading, setLoading] = useState(true);

useEffect(() => {
  async function fetchUserData() {
    setLoading(true);
    try {
      const data = await me();
      setUserData(data);
    } catch (error) {
      console.error(error);
    } finally {
      setLoading(false);
    }
  }
  fetchUserData();
}, []);
```

For **global state** (like the logged-in user), we use React Context, typically set up in UserProvider.tsx:

```
const UserContext = React.createContext();

function UserProvider({ children }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    async function loadUser() {
      const data = await fetchCurrentUser();
      setUser(data);
    }
    loadUser();
  }, []);

  return (
    <UserContext.Provider value={{ user, setUser }}>
      {children}
    </UserContext.Provider>
  );
}
```

Any child component can then consume the context using useContext.

API Integration: Communicating with the Backend

Our app communicates with the Django backend via REST API calls, typically handled in utility files like me.ts or vehicles.ts.

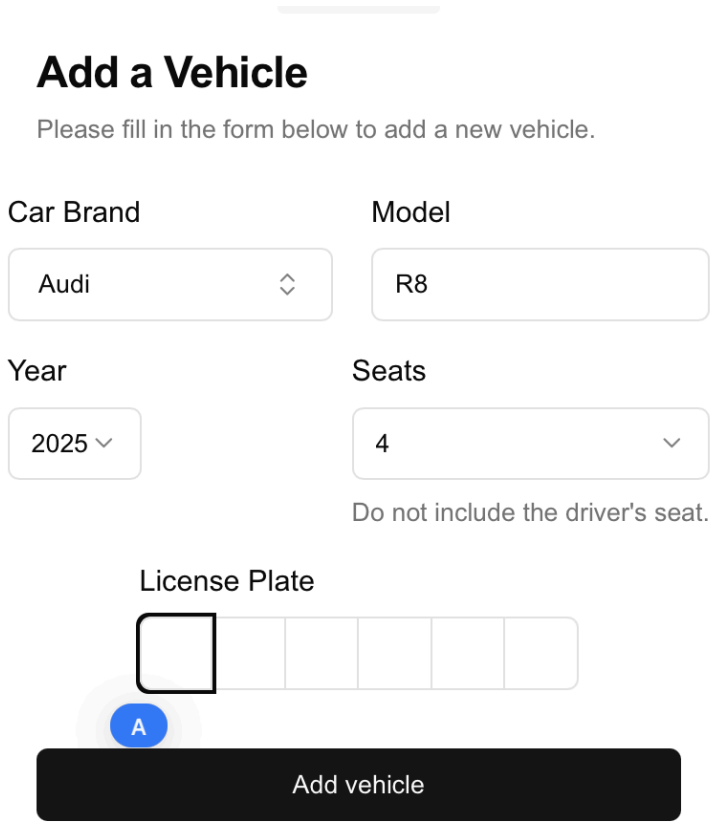
Example from **me.ts**:

```
export async function me() {
  const token = localStorage.getItem("token");
  const response = await fetch("http://localhost:8000/user
    headers: {
      Authorization: `Token ${token}`,
    },
  });

  if (!response.ok) {
    throw new Error("Failed to fetch user info");
  }

  return await response.json();
}
```

Example POST request from vehicleForm.tsx:

```
async function addVehicle(vehicleData) {
  const response = await fetch('/api/vehicles/', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      Authorization: `Token ${localStorage.getItem('token'
    },
    body: JSON.stringify(vehicleData),
  });

  if (response.ok) {
    const newVehicle = await response.json();
    setVehicles(prev => [...prev, newVehicle]);
  } else {
    console.error('Failed to add vehicle');
  }
}
```

Vehicle



Error Handling and Improvements

In the app generally, we catch errors, and are optimistic using book, and review actions where we show the result to the user on the UI before the server sends a confirmation. From this you can observe this pattern in inside joinRide function in SearchResults. tsx. We also debounce calls so we don't hit the API every time we type in search, use pagination to deal with large search results, do local caching when possible (e.g. save fetched user details) to make the app faster.

Summary

A Fully Connected Frontend With React Router, React Hooks, React Context and direct API integration, the frontend becomes a powerful, smooth and interactive application. You can feel safe letting users take a spin through trips, book travel, edit their profiles, and engage with the app without feeling like it'll come back to haunt you in a coupla years with code that's both maintainable and follows best practice patterns. In the following, we will discuss full user journeys and demonstrate how these technical components serve practical use cases.

3.4 Entire User Journeys, and Well-connected Web Flow

In this file, let's get out of individual components and learn how our app ties everything up into real, actual user journeys! By this, I mean critical thinking about how the user clicks around and uses the app on different pages, clicks and navigates, how the data flow between different components and server and how the whole thing feels cohesive. Time to unpack this by walking through the important user actions and hopping between the code and the structures.

Trip 1: Rides Quest

Upon arrival to the search page, a user enters their origin, available on this upcoming web page, and destination and date. We will take care of this in SearchPage. tsx, with a search form that takes the input and sends a search request to the backend:

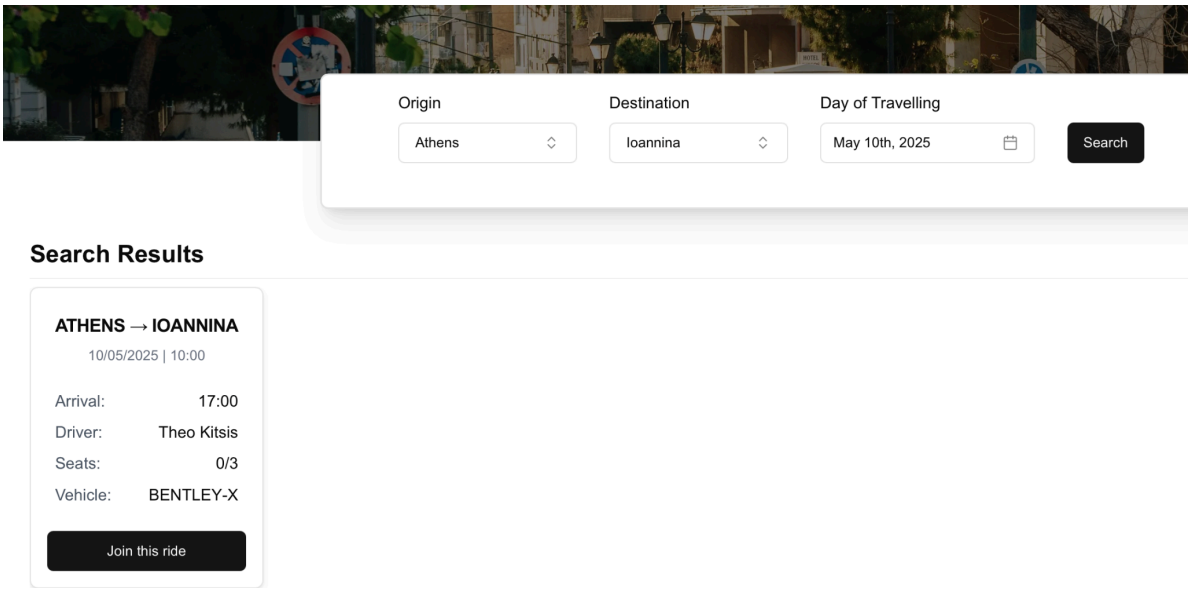
```
// In SearchPage.tsx
async function handleSearch() {
  const response = await fetch(`/api/rides?from=${origin}&
  const results = await response.json();
  setSearchResults(results);
}
```

The results are passed down to the SearchResults.tsx component, which displays a list of ride cards:

<SearchResults searchResults={searchResults} />

Each card is clickable, taking the user to the detailed ride page using React Router.

SearchResults



Search Results

The Search Results is a primary component of the rides discovery flow, allowing for filtering and browsing through the available rides according to pre-determined criteria. The three fields Origin, Destination and Day of Travelling in top search bar are created using controlled dropdowns and date picker component. When the user selects these values and clicks on the Search button, a request is sent to the backend to retrieve data using these values. The reply is used to dynamically display cards of matching rides below. The details of the trip, like the departure and arrival times, the driver's itinerary and rates, the number of seats available and the vehicle model are printed on each card, as is a large "Join this ride" button to begin booking. The component uses useState to handle search form inputs and useEffect to update results when a change is made.

Trip 2: Booking a Ride

On the ride detail page (RideDetail. tsx), users can book a seat. When they press the Join Ride button, we trigger the joinRide function like so:

```
// From RideDetail.tsx
async function joinRide(rideId) {
  const token = localStorage.getItem('token');
  await fetch(`/api/rides/${rideId}/`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
      Authorization: `Token ${token}`,
    },
    body: JSON.stringify({ passengers: [...currentPassenge
  });
}
```

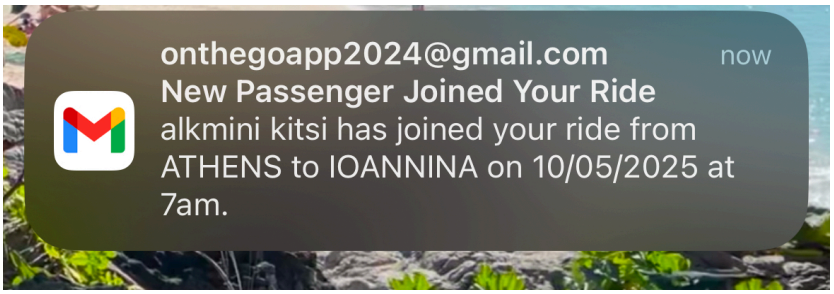
After a successful booking, we optimistically update the UI by updating the local state so the user sees themselves as part of the ride immediately.

Join

You joined this ride!

You can view the ride in your profile.

When the user join the ride an email is sent to the driver



Trip 3: Adding a Vehicle

For drivers, adding a new vehicle is done via the **Add Vehicle** drawer, which you can find in Vehicles.tsx. This drawer opens a form from VehicleForm.tsx, which on submission triggers:

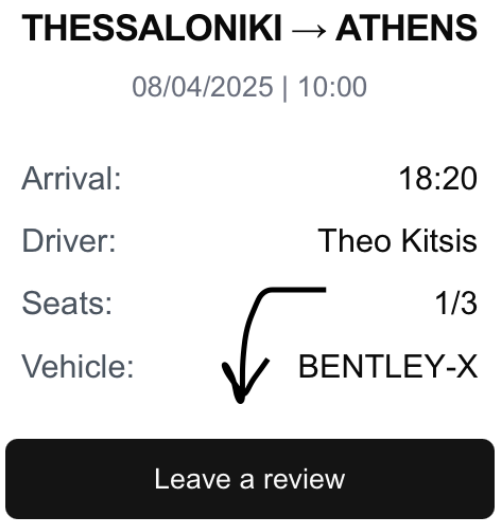
```
async function addVehicle(vehicleData) {
  const response = await fetch('/api/vehicles/', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      Authorization: `Token ${localStorage.getItem('token'
    },
    body: JSON.stringify(vehicleData),
  });

  if (response.ok) {
    const newVehicle = await response.json();
    setVehicles(prev => [...prev, newVehicle]);
  }
}
```

This keeps the driver's vehicle list updated without needing to reload the entire page.

Trip 4: Leaving a Review

After completing a ride, users can leave a review from the **Past Bookings** page (PastBookings.tsx). Clicking the **Leave a Review** button opens a drawer where the ReviewForm.tsx component is loaded. Submitting the review triggers:

```
async function submitReview(reviewData) {
  await fetch('/api/reviews/', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      Authorization: `Token ${localStorage.getItem('token'
    },
    body: JSON.stringify(reviewData),
  });
}
```



Upon submission, the app refreshes the local list of reviews so the user can easily sees their feedback appear.

Connecting the Pieces Every trip is enabled by: React Router managing the flow of pages. Manage local and global data with react state and context. Tailwind and Shadcn having the relatively same UI elements. API calls that keep the frontend and backend in sync. By partitioning the app into distinct pieces with unidirectional data flow, we've built a frontend system that is modular, maintainable, and very easily extended. This architecture allows the addition or modification of features to other parts of the application with no side-effects.

Developers can work on separate parts of the application with little overlap, and debugging is simplified with encapsulated logic. Additionally the collection of reusable components and centralised data fetching leads to a coherent and seamless user experience across the full platform. Also, the separation of concern for UI rendering, state control and backend communication leads to cleaner development flow. This structure not only speeds up development process, but also allows the future contributors to work on the system with understanding. In practice, this results in a more stable and scalable offering that can gracefully mature as the needs of the user evolve.

Chapter 4: Backend Architecture and Core Logic

The back end of the carpoolings app is more than just a server, it is the essential engine powering every aspect of the experience — transactions, business logic, data security, connections with the front end and more. Based on Django and Django REST Framework (DRF), the backend is modularized, maintainable, and sturdy — serving as a secure base for the entire platform. In this chapter, we will start by presenting what the backend is supposed to do, how it is organized inside and why each pieces of it is relevant, before diving into its flows and logic.

4.1 Django Project Structure Overview.

The backend is broken up into a few crucial files and folders, each with a unique and crucial role:

models. py → contains the database tables and object relations. That's where User, Ride, Vehicle, and Review models live, and dictate how the data is stored and related to one another.

serializers. py → Django models transform into a json, to communicate over an API. It's the one that determines which fields are exposed, and how they are formatted.

views. py → Everything related to core API logic to search rides, update bookings, or manage user profile.

urls. py → maps APIs routes (such as /rides/ or /users/me/) to specific view functions, and defines the entire API structure.

permissions. py → defines custom access rules, so only authorized users can edit rides or submit reviews.

utils. py → homes Ditto for helper functions such as the custom email system being used when a new passenger joins a ride. This modular and that re-usable nature design pattern conforms to Django's principles of separation of concerns (each components are specialized), whereby the backend architecture is a snap to develop, maintain and extend.

For example: models. py which describes the format of the data in the system. serializers. py to see what it is sending or receiving data through the API. views. py decides how business logic and user actions work against the data. It all works together seamlessly.

Let's trace a simple flow: A user searches for rides: The frontend makes an API call to the endpoint /rides/ (in urls. py). The request is sent to RideViewSet at views. py. The view looks up the Ride objects with the given ID in the model. py. The serializers.py is where the data is serialized using RideSerializer. py. The JSON response is sent back to the frontend. This chain leads to clean, organized back-end flow that is the basis for the scalable solution we're creating. Each is inextricably linked. when a frontend component asks for a user's previous bookings, the system works through these links: The frontend calls the api route defined in urls. py. The route directs to a view in views. py. The view pulls in data from the models in models. py. Here the view is passing the data through a serialiser in serializers. py. The answer is then returned to the frontend in the form of a json. Below we will dissect these files, discussing what their main contents are, and display real code examples of how they work together to support the system.

4.2 Models: The Data Blueprint

For the modeling of the data a blueprint is used. The models. py is the heart of the backend data layer. This is where we'll establish, how the app is going to store and organize all of its important information through users and rides, to vehicles and reviews. Each model corresponds to a table in the database, and Django provides us with a great facility for interacting with these tables as though they were Python objects, allowing us to easily read, write, and manipulate the data. For instance, the Ride model captures a trip that users can start, join or comment:

```python
class Ride(models.Model):
    created_by = models.ForeignKey(User, on_delete=models.CASCADE, related_name='rides_created')
    driver = models.ForeignKey(User, on_delete=models.CASCADE, related_name='rides_driven', null=True, blank=Tr
    vehicle = models.ForeignKey(Vehicle, on_delete=models.CASCADE, null=True, blank=True)
    start_location = models.CharField(max_length=255)
    end_location = models.CharField(max_length=255)
    departure_time = models.DateTimeField()
    arrival_time = models.DateTimeField()
    available_seats = models.IntegerField(null=True, blank=True)
    passengers = models.ManyToManyField(User, related_name="joined_rides", blank=True)

    def __str__(self):
        return f"{self.start_location} to {self.end_location} on {self.departure_time.strftime('%Y-%m-%d %H:%M'
```

Key points

We associate the ride to its creator (once it's put up), along with its driver and vehicle using ForeignKey fields. Passengers is the passengers field as we had a passengers methid to the Ride model. We save trip information such as start and end point, time and number of available seats.

The User model likewise inherits from Django's AbstractUser and supplements the following: phone_number privacy_enabled is_driver total_kilometers driving_license_number Those fields customize the user object itself to the needs of any ride sharing platform, such as verifying drivers or caring about user privacy.

Why it matters:

1. Modeling the entire structure and consistency of all of the app's data.
2. Models allow you to query powerfully, such as finding rides on a given date or counting passengers. 3) Models are the beating heart serializers, views, and permissions are built on. On top of all of this, Django's ORM gives us the ability to write easy-to-read, easy-to-maintain queries that don't run the risk of injection attacks, as they don't rely on raw sql. For example:

```python
# Find all rides where the user is a passenger
my_rides = Ride.objects.filter(passengers=user)

# Count how many rides a driver has completed
completed_rides = Ride.objects.filter(driver=user).count()
```

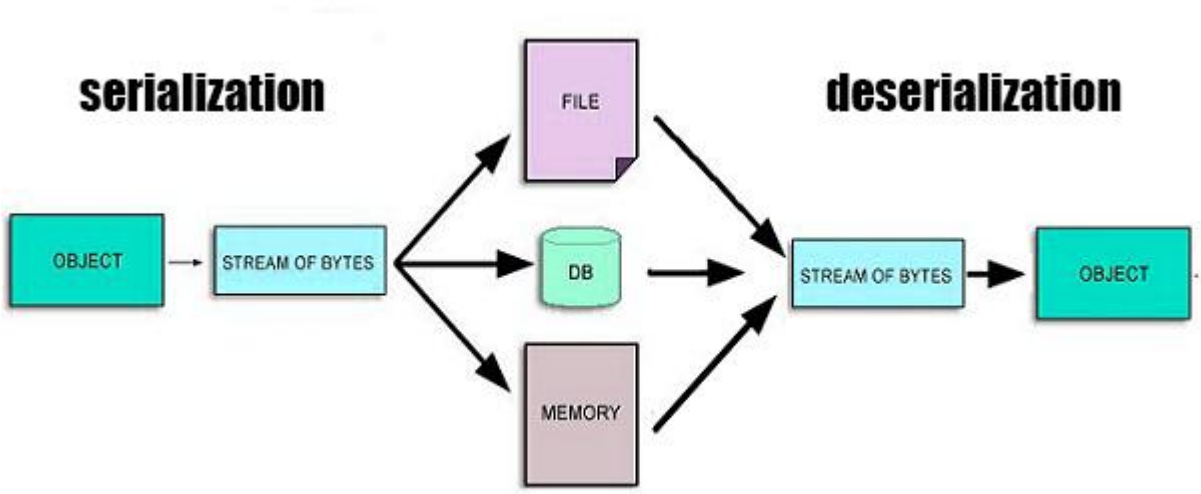
This means the app can evolve without major rewrites — adding new fields or relationships is straightforward thanks to Django's migration system.

4.3 Serializers: The Data Translators

Serializers in Django REST Framework are the interface through which the database models are exposed to the outside world, that is, the frontend and the other systems interacting with the backend API. The serializers reside in the serializers. py files, and do two things:

Serialize → A complex Django model instance (Python object) would then be converted into JSON, which can then be very simply sent to the front end over HTTP.

Deserialize → The incoming JSON data that comes from the frontend is validated, and it is then converted back to Django models, which can be saved to the database.



This dual role makes serializers one of the most important components in the system, as they control exactly what data flows in and out of the backend.

Let's look at some concrete examples from the project.

Example 1

```
class RideSerializer(serializers.ModelSerializer):
    class Meta:
        model = Ride
        fields = [
            'id', 'created_by', 'driver', 'vehicle',
            'start_location', 'end_location',
            'departure_time', 'arrival_time',
            'available_seats', 'passengers'
        ]
```

This serializer:

- Exposes only selected fields to the frontend — keeping sensitive or internal data hidden.
- Automatically handles nested relationships (like users and vehicles) thanks to DRF ModelSerializer features.
- Enables the frontend to create or update rides by sending a JSON payload that the backend will validate.

Example 2 is the **UserSerializer**:

```
class UserSerializer(serializers.ModelSerializer):
    password = serializers.CharField(write_only=True)

    def create(self, validated_data):
        password = validated_data.pop("password")
        user = User(**validated_data)
        user.set_password(password)
        user.save()
        return user
```

The password is write-only, and will never be returned in API responses (for security reasons). What is done here is that the password is hashed when saving through the create method so it doesn't store the password in plain-text.

Serializers :

They are the contract between frontend and backend. They also blockade user data by adding controls to what is exposed. They enable you to create or update complex data (such as many to many relationships) cleanly. You can centralize validation; so the backend can remain strong against bad or nefarious input. In reality, when the frontend wants to 'pull down data' (in this example, past rides),

the backend: Queries the models. Passes the data to the proper serializer. Returns a neat and tidy JSON response that you can easily visualize on your front-end application.

And when a user sends a form (e.g. adding a new vehicle),

the frontend: Sends JSON data. The data is validated by the serializer. The serializer will, if valid, turn it into a Django object and then save it. If there is failure, the serializer provides clear error messages. This combination of transformation, validation and security makes serializers one of the genuine backbones of a backend system.

4.4 Views : The Business Logic Engine

The Business Logic Engine The views. py file is the one where the logic of the backend resides. It's how we specify what the backend should do when a user makes an action in our system, like searching for rides, creating a booking, updating user information, or leaving a review. In Django REST Framework, we make use of viewsets, a concept that groups all related actions (like list, retrieve, create, update, delete) in a concise manner. It makes the code a lot clearer and it "feels" nice and resty to me. Let's break it down. Example from views.py:

```python
class RideViewSet(viewsets.ModelViewSet):
    queryset = Ride.objects.all()
    serializer_class = RideSerializer
    permission_classes = [IsAuthenticated]

    def get_queryset(self):
        queryset = super().get_queryset()
        origin = self.request.query_params.get('from')
        destination = self.request.query_params.get('to')
        date_str = self.request.query_params.get('date')

        if origin:
            queryset = queryset.filter(start_location=origin)
        if destination:
            queryset = queryset.filter(end_location=destination)
        if date_str:
            date_obj = parse_date(date_str)
            if date_obj:
                queryset = queryset.filter(departure_time__date=date_obj)

        user_id = self.request.query_params.get('user_id')
        if user_id:
            queryset = queryset.filter(Q(driver__id=user_id) | Q(passengers__id=user_id)).distinct()


        return queryset
```

What this does:

1) It's responsible for incoming requests to the /rides/ endpoint.

2) Filters the rides using query params, like origin, destination or date of the ride.

3) Returns the correct rides list to the frontend (via the RideSerializer) Another essential piece are custom actions, which are RESTful endpoints that go beyond the basic CRUD methods. Like for instance, the /users/me/ endpoint:

```python
@action(detail=False, methods=['get'], url_path='me', perm
def me(self, request):
    serializer = self.get_serializer(request.user)
    return Response(serializer.data)
```

This lets the frontend make a term for the current user and fetch their information in a neat single call. We also specify custom update logic:

```python
def update(self, request, *args, **kwargs):

    instance = self.get_object()
    original_passengers = list(instance.passengers.all().values_list('id', flat=True))
    original_driver = instance.driver.id if instance.driver else None

    response = super().update(request, *args, **kwargs)
    instance.refresh_from_db()

    new_passengers = list(instance.passengers.all().values_list('id', flat=True))
    new_driver = instance.driver.id if instance.driver else None

    departure_time = instance.departure_time
    date_str = departure_time.strftime("%d/%m/%Y")

    time_str = f"{int(departure_time.strftime('%I'))}{departure_time.strftime('%p').lower()}"
```

This both refreshes the ride and infuses a bit of behavior, like sending email notices when new passengers hop on.

Views They are a way to connect a user request with the business logic that operates on the data — when a user clicks a button, or submits a form, a view handles that. They handle data flow between models, serializers, and the frontend, and make sure everything is in sync and coherent. They are the enforces of permissions and business rules, only allowing allowed things to happen. They allow custom behavior on a feature-by-feature basis, such as sending email, running analytics, or adding specialized filters. In other words, views aren't just plumbing — they actively influence the behavior of the app, what the users' experience, and the seamless integration of backend and frontend.

4.5 Permission, Utilities and Supporting Logic

Beyond models, serializers, and views, the backend has a set of important supporting layers to keep the application secure, flexible, and operational. These are a combination of permission classes, utility methods, helper modules, and routing definitions all working together to enforce the app's business logic and ensure that the entire application is maintainable.

Permissions: Regulating Access And Limiting Rules Permissions are described in permissions. py and attached directly to viewsets. They specify the groups of individuals who are allowed to read or change some set of data, and what types of users are permitted to access or change it. This is the layer where we did things like we must only allow sensitive operations to be performed by privileged users.

For example, by using: permission_classes = [IsAuthenticated]

We make sure unauthenticated users have no access to the protected API endpoints like creating rides, updating profiles or posting reviews. If you need more fine grained control, we'll need to create custom permission classes.

For example: Only the driver can accept new passengers. A ride can only be updated by the user that created the lift. Users can only leave a review on a ride they participated in. These rules of the road are critical in order to keep a level playing field and build a community that can be trusted for both consistency and fairness. Permissions

1. To secure the sensitive operation from unauthorized users.
2. Implement role-based access control (drivers and riders). Implement vital business rules on the backend.
3. Push security checks that are to be strictly enforced to the server to minimize front-end superuser privileges. Utilities – Helpers to keep your logic clean. The utils. When I give the above pstools code, the entire code is well over 1000 lines, but, if you haven't already, you could try refactoring myapp/util. To illustrate, at a fundamental level, you could understand the email notification system as:

```
from django.core.mail import send_mail

def send_email(to, subject, message):
    send_mail(subject, message, 'noreply@carpoolapp.com',
```

This helper is used in response views when certain types of actions have been performed. A passenger requesting a ride (the driver is notified). The driver confirming a job (the passenger is alerted). Putting that logic in a utility function, and the main bussiness logic in views. py keeps extremely focused, readable, and maintainable.

Other potential utilities are: Date/Time format helpers (normalised display of date/time). Logging or tracking (e.g. for analytics, monitoring). Setup hooks for third-party APIs (eg. maps or payment services).

As always keep your core code DRY. Collect popular commands into one place. Ease the ability to scale the system by factoring out special cases.

The Backing Logic: Routing and API Structure Supporting files like urls. py are responsible for configuring how the API is organised, and to determine how requests subsist their way through the system. For example:

```
router.register(r'rides', RideViewSet)
router.register(r'vehicles', VehicleViewSet)
router.register(r'users', UserViewSet)
```

This maps predictable, RESTful API endpoints such as /api/rides/ and /api/users/ to the appropriate viewsets, so the frontend always knows where to direct its requests. The nice URL syntax also allows you to be able to:

Aging View the structure of the system complete. Extend functionalty by creating new routes. Connect the backend to other tools or services.

Routings

1) The outwardly facing principle-part of the API.

2) Wires everything (views, models, serializers) together into a functioning system.

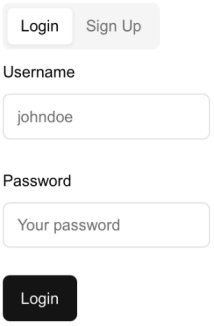
3) Greater ease of maintenance and clarity.

4) It gives a clean contract for the frontend to deal with. We then get to the backend's authentication system, the data relationships that's gluing our data together and we finished with a detailed step-by-step guide of what happens when a user interacts with our backend.

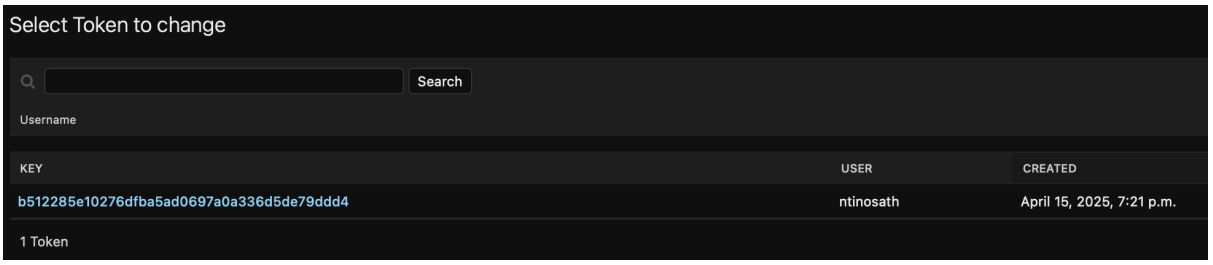
4.6 Authentication, Database Relationships, and Data Flow

In this part, we concentrate on how the backend enforces secure access, governs the relations between different data and orchestrates the step-by-step workflow of data among components. These are important elements which form the app's trustworthiness and reliability.

How Authentication Works: What Does It Mean To Have Permission? The app utilizes Django's builtin authentication system along with Django REST Framework's token authentication. On the login, the user is given an access token, that they add into the header of every API call they make thereafter.



Example :



The above approach, which is token-based, lets the backend to fetch the user issuing a request, and apply necessary permission policies.

Authentication

1. The verification process, so that private/sensitive endpoints can only be access by verified users.

2. Provides user-related functionality (accessible booking details or writing reviews).

3. Prevents unauthorized operations, information disclosure or a malicious attack. In the project that I have the authentication is handled by Django's REST framework token system — it generates a token when a user logs in and automatically checks it on each API request. Database Relationships: One To Many And Many To Many Relationships With Data Joining together data with Database Relationships To join two database relations (a table is a relation, remember from the definition of a database above) you need to have a relationship from one to many or many to many between the relations in you database.

The value of Django's ORM is in the way it models the relationships between the models.

Key relationships in the app: A User can have multiple Vehicle entries (one-to-many). A Ride is made by a User, has one driver (a User), and can have many passengers (many-to-many). A Review ties two users together (the reviewer and the reviewed) and points to a specific Ride. These connections are described by models. py You can use Django fields such as ForeignKey and ManyToManyField and they are essential for representing real-world connections.

Example: passengers = models.

ManyToManyField(User, related_name="joined_rides", blank = True)

This corresponds to the fact that one ride can be shared by multiple passengers and one user can join multiple rides, reflecting the nature of carpooling.

Relationships

1) Make the system extensible and capable enough to model the complex real world scenarios.
2) Support fast querying (e.g. get all rides a user participated in).
3) Secure data integrity by using the relational constraints in Django.

Flow of Data: A full Backend Travelogue

So, let me go through how this works when a person joins a ride:

1. Frontend Action → The user presses "Join Ride" in the mobile app.
2. API Request → The frontend sends a PATCH request to /rides/{ride_id}/ with the user's token.
3. View Logic → RideViewSet. update() Validates the request, user and updates the passengers.
4. Database Update → DUpdated passenger list is updated in the database.
5. Utility Trigger → The system invokes the send_email() method to notify the driver.
6. API Response → Backend returns a JSON response acknolwedging the successful update.
7. Frontend Update → Mobile app reflects new UI state for joined ride.
   This multi-stage flow illustrates one of the examples of just how closely coordinated the different backend components are — every piece has its role, from implementing access management, through data mutations and up to user tracking.

Chapter 5 : Development Reflections and Engineering Insights

5.1 Development Challenges

During the development of the application, a series of technical and architecture challenges appeared. These difficulties were actually teaching moments on the road to creating a full-stack web app with Next. js, Tailwind CSS, ShadCN UI for frontend and Django REST Framework for backend. The aim was to build a responsive modern carpooling solution.

Synchronize Frontend and Backend

In the beginning, one of the biggest challenges was to make them (frontend and backend) talk to each other. Manipulating user authentication tokens properly — especially on login and retrieving data (/users/me/) — required a bit of work. And in many of those cases it was an API-related problem like CORS errors or "401 Unauthorized" errors caused by wrong headers and / or fail to store tokens in localStorage etc. There would have to be a reliable, secure way of sending requests to the authenticated API.

State Management & Data Refresh

Even though there is no great state management libraries involved such as Redux, it employs React‘s useState, useEffect and context API to manage the local and global state. But things got complicated to keep the frontend state in sync with backend changes. For instance, when a ride was updated (PATCH /rides/:id/), the changes were sometimes not shown right away in the UI. This involved manually re-fetching data after those actions and being very careful with something like the dependency array within useEffect.

Reuseable and structured component

Since the early days, the application has strived for a modular and component-based frontend architecture. But how to correctly define props for dynamic components was not so easy – especially when you start to reuse UI components such as a Card component in different contexts (upcoming vs. past rides). A few other parts needed to be reconfigured to prevent code-duplication while keeping everything clear and maintainable.

UI Consistency with ShadCN and Tailwind

The UI component library that I used was ShadCN UI, developed with Radix UI alongside Tailwind CSS, and it allowed for rapid prototyping with visual coherence. It did however take a while to figure out how to modify components such as Tabs, Badge and Drawer for use with dynamic content. Being able to adhere to accessibility and responsiveness, while embracing the utility-first mindset of Tailwind, was very important for a consistent user-interface experience.

Debugging and Testing

Because there were no automated tests (neither unit nor integration) debugging was rather manual and subjective. Many of the bugs originated in nuanced factors—like not accounting for null drivers or improperly formatted date strings. Browser dev tools, console logs, and server-side logging proved invaluable in debugging and resolving these types of bugs during development.

5.2 Summary of Lessons Learned and Developer Growth

The learning process was more than just programming during the development of the carpooling application. It was a holistic exercise which challenged my knowledge of full-stack application development, made me more adaptable as a developer, and bolstered my capability to translate real-world problems to software engineering solutions.

One of the main learnings was the necessity to write maintainable and modular code. The methodology of Extracting reusable & well-isolated components was the key to developing a scalable app (especially in frontend). Libraries such as ShadCN UI made development speedy without sacrificing flexibility, and Reacts component-based design pattern taught me to build with structure and reusability in mind.

On the back end, Django's Model-View-Serializer pattern helped me understand a clean way to do RESTful design. Serializers' role as a middleman between Python objects and JSON-formatted API responses taught me about data abstraction and verification. Also, while working with custom views, permissions, and actions I got a better sense of the tradeoff between backend control and frontend flexibility.

I faced difficulties along the way such as fetching data asynchronously, token based authentication, conditional UI rendering according to user role. These problems were solved by using React hooks (useEffect,state), dynamic routing and authenticating protected API calls using headers. So I learned more out of necessity, which not everyone would expect of me: JavaScript, TypeScript, and HTTP protocol logic.

From a program management standpoint, I understood the importance of planning and testing in increments. Writing and testing each feature, ride requests, user authentication and profile viewing, piece by piece, helped to avoid unstable, un-testable code. In addition, separately testing each module before combining the code into an entire flow meant fewer bugs, and a higher chance of being able to deploy the system.

Finally, this study reiterated the importance of UX thinking. By structuring tabs (personal details, upcoming/ridden rides, etc) and conditional views (if you are a driver or no) I provided user friendly but clear navigation, with attention to the overall feel.

Altogether, this project actually helped me to grow as a student to be more patient and solve a hard challenge in the most ideal way. It was a "hands on" class where theory and engineering decisions were blended, this is exactly what I needed to prepare for real world development processes and team based software design.

5.3 Overall Reflection

Throughout the development of this carpooling application, the process has been both technically challenging and deeply rewarding. What began as a concept — connecting drivers and passengers to promote shared transportation — evolved into a full-stack solution, complete with authentication, dynamic frontend views, backend logic, and real-time data flow.

Technical Growth

From a technical standpoint, this project offered an invaluable opportunity to work with a modern tech stack. On the **frontend**, using **React**, **TypeScript**, and **Tailwind CSS** (with **ShadCN UI**) taught me how to build responsive and scalable user interfaces. Managing state and routing, especially through tools like React Context and React Router, gave me hands-on experience with modular

application design. On the **backend**, I gained deep exposure to **Django** and the **Django REST Framework**, learning how to model data with precision using serializers and how to enforce permissions and roles securely. I also learned to integrate third-party tools, handle asynchronous actions, and set up safe API endpoints. Connecting the frontend and backend seamlessly via token-based authentication and structured RESTful APIs was one of the most gratifying accomplishments of the project.

Design and Usability Insights

Beyond code, I also gained a stronger appreciation for **user experience (UX)** and **interface design (UI)**. It wasn't enough for the app to simply function — it had to be intuitive, accessible, and pleasant to use. This led to several design iterations and usability refinements, including cleaner layout structures, visual feedback during loading, and a clear separation between driver and passenger roles.

Moreover, testing the app under realistic conditions helped reveal user flows that needed improvement. Implementing features like "Upcoming Bookings," "Past Rides," and "Leave a Review" added depth and completeness to the user experience.

Project Management and Discipline

One of the biggest takeaways from this thesis was the discipline of managing a long-term project: organizing tasks, debugging under pressure, and breaking large goals into smaller, actionable items. The experience of documenting, maintaining clean code, and testing features thoroughly offered a clear sense of how software is built professionally.

Personal Takeaways

This project reinforced the importance of end-to-end thinking — understanding not just how a component works but also how it integrates into the broader system. It also reminded me that every problem has multiple solutions, but clarity, maintainability, and user-centric design should always guide the final decision. In conclusion, this application is more than just a functional piece of software — it represents a journey of learning, building, and refining. It stands as a solid foundation that could be extended and deployed to support real users in real cities, solving real transportation challenges.

Chapter 6 :

Code Technical Specifications

6.1 Introduction

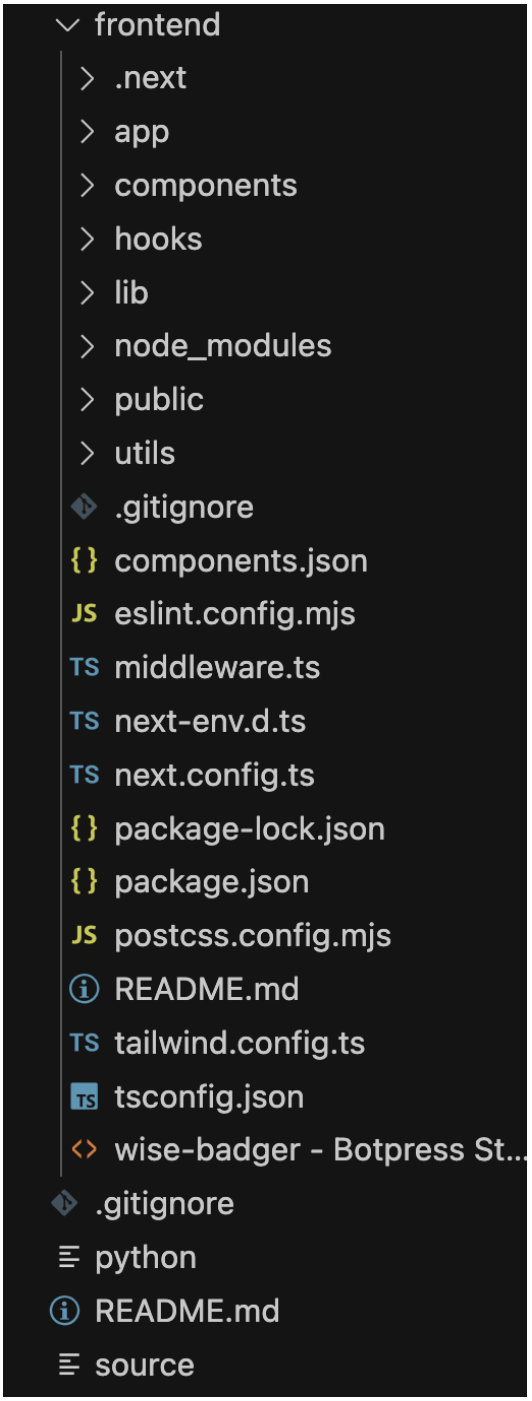
In this section we further analyze the codebase for the carpooling application. We dissect the architecture, components, modules and high level logic that spans through backend and frontend on a road to understand how things work under the hood. The purpose of this section is not to provide code snippets alone, but to expatiate on their end, meaning, and even reason for existence — and this is what I believe is sorely missing from most tutorial-like materials. In doing so, we bring to the fore the technical depth, difficulties, and solutions involved in the project. This chapter is organized into two main sections:

Frontend Code: This includes all user facing react and typescript components, hooks and different styling System (Tailwind css) etc and how they integrated with the shadcn- ui components.

Backend Code: Including Django models, serializers, viewsets, permissions, custom logic, and in general the structure of the Rest API.

We're going to show the interesting parts of the code here.

6.2 Frontend in detail



‘ app→ login → page.tsx ‘

```
import LoginForm from "@/components/forms/loginForm";
import SignUpForm from "@/components/forms/signupForm";
import { Tabs, TabsContent, TabsList, TabsTrigger } from
"@/components/ui/tabs"
import Image from "next/image";

export default function Login() {
   return (
   <div className="flex h-screen">
     <div className="w-1/2 h-full bg-[#009ADC] items-center justify-center
flex">
       <Image
         src="/onthegoblue.svg"
         alt="On The Go logo"
         width={500}
         height={500}
         priority
       />
     </div>
     <div className="w-1/2 items-center justify-center flex p-8">
       <Tabs defaultValue="login" className="">
         <TabsList>
           <TabsTrigger value="login">Login</TabsTrigger>
           <TabsTrigger value="signup">Sign Up</TabsTrigger>
         </TabsList>
         <TabsContent value="login">
           <LoginForm />
         </TabsContent>
         <TabsContent value="signup" className="">
           <SignUpForm />
         </TabsContent>
       </Tabs>
     </div>
   </div>
   );
}
```

Explanation:

The react component (Login. tsx) that performs the initial authentication interface.

Imports: We import the LoginForm and SignUpForm components, which contain the actual login and registration logic, respectively. We import Shadcn's Tabs modules for the UI.

Layout: This components splits the screen in two halves - left side displays the project logo using the Next. js's Image component, and right-hand side, with the login/signup forms inside a tabbed interface.

Tabs: With the use of TabsList, TabsTrigger and TabsContent, we offer a nice switch between login and sign-up views without the need for a page-load (better for users).

' app→ profile → page.tsx '

```
"use client";

import { useEffect, useState } from "react";
import Loading from "@/components/loading";
import { me } from "@/utils/me";
import { Avatar, AvatarFallback, AvatarImage } from
"@/components/ui/avatar";
import { UserRound } from "lucide-react";
import { Badge } from "@/components/ui/badge";
import { Tabs, TabsContent, TabsList, TabsTrigger } from
"@/components/ui/tabs"
import PersonalDetails from "@/components/personalDetails";
import Vehicles from "@/components/vehicles";
import UpcomingBookings from "@/components/upcomingBookings";
import PastBookings from "@/components/pastBookings";

export default function Profile() {
 const [loading, setLoading] = useState(true);
 const [firstName, setFirstName] = useState("");
 const [lastName, setLastName] = useState("");
 const [isDriver, setIsDriver] = useState<boolean | undefined>(undefined);
 const [totalKilometers, setTotalKilometers] = useState("");

 useEffect(() => {
   async function fetchMe() {
     try {
```

```
      const data = await me();
      setFirstName(data.first_name);
      setLastName(data.last_name);
      setIsDriver(data.is_driver);
      setTotalKilometers(data.total_kilometers);
    } catch (error) {
      console.error(error);
    } finally {
      setLoading(false);
    }
  }
  fetchMe();
}, []);

if (loading) {
  return <Loading />;
}

return (
  <div className="min-h-screen relative">
    <section className="relative w-full h-[15vh] flex gap-2 items-center
border-b gap-8 p-8">
      <Avatar className="w-24 h-24 border-2 border-primary">
          <AvatarImage src="" alt={`${firstName} ${lastName}`} />
          <AvatarFallback>
            <UserRound className="w-8 h-8 text-primary" />
          </AvatarFallback>
        </Avatar>
      <div>
          <h2 className="text-xl font-bold text-base text-left">
              {firstName} {lastName}
          </h2>
          <div className="flex space-x-2 my-2">
              <Badge variant="outline">
              {isDriver ? "Driver" : "Passenger"}
              </Badge>
              {isDriver && (
              <Badge variant="outline">
                  Total Kilometers:{" "}
                  {totalKilometers === null ? "0" : totalKilometers}
              </Badge>
              )}
          </div>
      </div>
    </section>
    <Tabs defaultValue="personal-details" className="">
      <TabsList className="flex justify-center w-full gap-4">
          <TabsTrigger value="personal-details"
className="text-md">Personal Details</TabsTrigger>
          {isDriver && (
```

```
              <TabsTrigger value="vehicles"
className="text-md">Vehicles</TabsTrigger>
           )}
           <TabsTrigger value="upcoming-bookings"
className="text-md">Upcoming bookings</TabsTrigger>
           <TabsTrigger value="past-bookings" className="text-md">Past
bookings</TabsTrigger>
       </TabsList>
       <TabsContent value="personal-details" className="flex justify-center
items-center">
         <PersonalDetails/>
       </TabsContent>
       {isDriver && (
           <TabsContent value="vehicles"><Vehicles/></TabsContent>
       )}
       <TabsContent value="upcoming-bookings">
         <UpcomingBookings/>
       </TabsContent>
       <TabsContent value="past-bookings">
         <PastBookings/>
       </TabsContent>
     </Tabs>
   </div>
 );
}
```

Explanation:

This React Profile component (Profile. tsx) is the primary user dashboard where users can see details about his account, the cars and the bookings.

State management: We fetched user (first name, last name, driver status, total kilometers) details with the help of React useState and useEffect, and we also called the me() utility which hits the backend API.

Loading state: When fetching data we show Loading component to provide a better user feedback. Depending on the user's role (Driver/Passenger) and if they already traveled or not, we display badges to show that so the passenger knows.

Components: The dashboard is a composition of reusable and configurable components such as PersonalDetails, Vehicles, UpcomingBookings and PastBookings components, and each component is responsible for rendering its own data and layout. This being gateways the user has to everything in order to manage their profile and activity within the platform.

‘ app→ utils → me.ts ‘

```
export async function me() {
   const token = localStorage.getItem("token");
   const response = await fetch("http://localhost:8000/users/me/", {
     headers: {
       "Content-Type": "application/json",
       "Authorization": `Token ${token}`,
     },
   });
    if (!response.ok) {
     throw new Error("Failed to fetch user data");
   }
    const data = await response.json();
   return data;
 }

export async function getUserById(id: number) {
 const token = localStorage.getItem("token");
 const response = await fetch(`http://localhost:8000/users/${id}/`, {
   headers: {
     "Content-Type": "application/json",
     "Authorization": `Token ${token}`,
   },
 });
 if (!response.ok) {
   throw new Error("Failed to fetch user info");
 }
 const data = await response.json();
 return data;
}
```

Explanation:

These two are some basic async functions in typescript that we need to grab user data securely from backend_api.

me(): This method returns the user profile of the currently logged in user. It takes the token that is stored in localStorage (after login) and adds it to the Authorization header. It queries the /users/me/ endpoint with a GET request and receives the name, driver status and profile data. If the request is rejected (occupied token, server error), it is rejected.

getUserById(id): This method returns information about any user, by their ID. Like me(), it relies on token in localStorage, and performs a GET to /user/{id}/. This comes handy when client wants us to showcase public profiles or retrieve details of other kind of users (say drivers, passengers in a ride).

As far as **components** concerned we'll show the 2 basic :
upcomingBookings , pastBookings

upcomingBookings

```
import { useEffect, useState } from "react";
import { Button } from "./ui/button";
import { Card, CardContent, CardHeader, CardTitle } from "./ui/card";
import { Separator } from "./ui/separator";
import { getVehicles } from "@/utils/vehicles";
import Loading from "./loading";
import { me } from "@/utils/me";
import VehicleInfo from "./vehicleInfo";
import UserInfo from "./userInfo";
import { Drawer, DrawerContent, DrawerDescription, DrawerHeader,
DrawerTitle, DrawerTrigger } from "./ui/drawer";
import DriverRequestForm from "./forms/driverRequestForm";
import { useToast } from "@/hooks/use-toast"
import { Ride } from "./searchResults";

export default function UpcomingBookings() {

   const { toast } = useToast();
   const [rides, setRides] = useState<Ride[] | undefined>(undefined);
   const [userData, setUserData] = useState<{ id: number; is_driver:
boolean } | undefined>(undefined);
   const [loading, setLoading] = useState(true);

   useEffect(() => {
```

```
      async function fetchMeData() {
        setLoading(true);
        try {
          const data = await me();
          setUserData(data);
        } catch (error) {
          console.error(error);
        } finally {
          setLoading(false);
        }
      }
      fetchMeData();
    }, []);

    useEffect(() => {
      if (!userData?.id) return;
      async function fetchUserRidesData(userId: number) {
        setLoading(true);
        try {
          const token = localStorage.getItem("token");
          const response = await
fetch(`http://localhost:8000/rides/?user_id=${userId}`, {
            method: "GET",
            headers: {
              "Content-Type": "application/json",
              "Authorization": `Token ${token}`,
            },
          });
          if (!response.ok) {
            throw new Error("Failed to fetch user rides");
          }

          const ridesData: Ride[] = await response.json();
          setRides(ridesData);
        } catch (error) {
          console.error(error);
        } finally {
          setLoading(false);
        }
      }
      fetchUserRidesData(userData.id);
    }, [userData]);

    if (loading) {
      return <Loading />;
    }

    const upcomingRides = rides?.filter(
      (ride) => new Date(ride.departure_time).getTime() > Date.now()
    );
```

```
    return (
      <div className="mx-8">
        <div className="flex justify-between items-center">
          <h1 className="text-2xl font-bold">Upcoming Rides</h1>
      </div>

        <Separator className="my-2"/>

        <div className="flex space-x-4 overflow-x-auto pb-2
min-h-[200px]">
          {upcomingRides !== undefined && upcomingRides.length === 0 && (
              <div className="flex justify-center items-center w-full
min-h-[200px] text-xl">
                  No upcoming rides found.
              </div>
          )}
          {upcomingRides !== undefined && upcomingRides.length > 0 &&
upcomingRides.map((ride) => (
          <Card key={ride.id} className="min-w-[250px] flex-shrink-0
shadow-md border rounded-lg">
              <CardHeader className="flex flex-col items-center
text-center">
              <CardTitle className="text-lg font-semibold">
                  {ride.start_location} → {ride.end_location}
              </CardTitle>
              <p className="text-sm text-gray-500">
                  {new Date(ride.departure_time).toLocaleDateString()}{"
"}
                  |{" "}
                  {new Date(ride.departure_time).toLocaleTimeString([], {
hour: '2-digit', minute: '2-digit' })}
              </p>
              </CardHeader>
              <CardContent className="space-y-2">
              <div className="flex justify-between items-center">
                  <span className="font-medium
text-gray-600">Arrival:</span>
                  <span className="text-base">
                  {new Date(ride.arrival_time).toLocaleTimeString([], {
hour: '2-digit', minute: '2-digit' })}
                  </span>
              </div>
              <div className="flex justify-between items-center">
                  <span className="font-medium
text-gray-600">Driver:</span>
                  <span className="text-base">{ride.driver === null ? "Not
found yet" : <UserInfo userId={ride.driver}/>}</span>
              </div>
              <div className="flex justify-between items-center">
```

```
                    <span className="font-medium
text-gray-600">Seats:</span>
                    <span className="text-base">{ride.available_seats ===
null ? "N/A" : ride.passengers.length + "/" + ride.available_seats}</span>
                </div>
                <div className="flex justify-between items-center">
                    <span className="font-medium
text-gray-600">Vehicle:</span>
                    <span className="text-base">{ride.vehicle === null ?
"Not defined yet" : <VehicleInfo vehicleId={ride.vehicle} />}</span>
                </div>
                </CardContent>
                <div className="p-4 pt-0">
                    <Button className="w-full" disabled>
                        {ride.driver === userData?.id ? "You are the driver"
: "You are a passenger"}
                    </Button>
                </div>
            </Card>
            ))}
          </div>
          <Separator className="my-2"/>
        </div>
      );
    };
```

Explanation:

This UpcomingBookings component shows all of the future rides that have been created by the logged-in user. State management: It manages rides and user data and fires off backend API calls (me()and /rides/).

Fetching Data: After the user has loaded, it fetches their rides associated with them, and filters out for only those rides with a departure_time in the future.

Display: It uses components created by Shadcn Card for showing details of each ride like origin and destination, date, time, driver etc with empty seats.

Conditional rendering: If you don't have any upcoming rides it should display a nice message, otherwise, it maps over the rides array and render each ride as a card. UI components: Employs custom subcomponents (UserInfo, VehicleInfo) to retrieve and display counterpart information.

pastBookings

```
import { useEffect, useState } from "react";
import { Button } from "./ui/button";
import { Drawer, DrawerClose, DrawerContent, DrawerDescription,
DrawerFooter, DrawerHeader, DrawerTitle, DrawerTrigger } from
"./ui/drawer";
import UserInfo, { User } from "./userInfo";
import { me } from "@/utils/me";
import { Separator } from "./ui/separator";
import { Card, CardContent, CardHeader, CardTitle } from "./ui/card";
import VehicleInfo from "./vehicleInfo";
import { Ride } from "./searchResults";
import ReviewForm from "./forms/reviewForm";

export default function PastBookings() {

   const [loading, setLoading] = useState(true);
   const [userData, setUserData] = useState<{id: number; is_driver:
boolean} | undefined>(undefined);
   const [rides, setRides] = useState<Ride[] | undefined>(undefined);
   const [isOpen, setIsOpen] = useState(false);
   const closeDrawer = () => setIsOpen(false);
   const [selectedRide, setSelectedRide] = useState<Ride | null>(null);

   useEffect(() => {
       async function fetchMe() {
           const data = await me();
           setUserData(data);
       }
       fetchMe();
       setLoading(false);
   }, []);

   useEffect(() => {
       if (!userData?.id) return;
       async function fetchUserRidesData(userId: number) {
           setLoading(true);
           try {
           const token = localStorage.getItem("token");
           const response = await
fetch(`http://localhost:8000/rides/?user_id=${userId}`, {
               method: "GET",
               headers: {
               "Content-Type": "application/json",
               "Authorization": `Token ${token}`,
               },
           });
           if (!response.ok) {
               throw new Error("Failed to fetch user rides");
```

```
            }

            const ridesData: Ride[] = await response.json();
            setRides(ridesData);
            } catch (error) {
            console.error(error);
            } finally {
            setLoading(false);
            }
        }
        fetchUserRidesData(userData.id);
    }, [userData]);

// palia rides
    const filteredRides = rides?.filter(
        (ride) => new Date(ride.departure_time).getTime() < Date.now()
    );

    return (
        <div className="mx-8">
          <div className="flex justify-between items-center">
            <h1 className="text-2xl font-bold">Search Results</h1>
        </div>

          <Separator className="my-2"/>

          <div className="flex space-x-4 overflow-x-auto pb-2
min-h-[200px]">
            {rides === undefined && (
                <div className="flex justify-center items-center w-full
min-h-[200px] text-xl">Your search results will appear here!</div>
            )}
            {filteredRides !== undefined && filteredRides.length === 0 && (
                <div className="flex justify-center items-center w-full
min-h-[200px] text-xl">
                    No rides found. Try another search.
                </div>
            )}
            {filteredRides !== undefined && filteredRides.length > 0 &&
filteredRides.map((ride) => (
            <Card key={ride.id} className="min-w-[250px] flex-shrink-0
shadow-md border rounded-lg">
                <CardHeader className="flex flex-col items-center
text-center">
                <CardTitle className="text-lg font-semibold">
                    {ride.start_location} → {ride.end_location}
                </CardTitle>
                <p className="text-sm text-gray-500">
                    {new Date(ride.departure_time).toLocaleDateString()}{"
"}
```

```
                    |{" "}
                    {new Date(ride.departure_time).toLocaleTimeString([], {
hour: '2-digit', minute: '2-digit' })}
                </p>
                </CardHeader>
                <CardContent className="space-y-2">
                <div className="flex justify-between items-center">
                    <span className="font-medium
text-gray-600">Arrival:</span>
                    <span className="text-base">
                    {new Date(ride.arrival_time).toLocaleTimeString([], {
hour: '2-digit', minute: '2-digit' })}
                    </span>
                </div>
                <div className="flex justify-between items-center">
                    <span className="font-medium
text-gray-600">Driver:</span>
                    <span className="text-base">{ride.driver === null ? "Not
found yet" : <UserInfo userId={ride.driver}/>}</span>
                </div>
                <div className="flex justify-between items-center">
                    <span className="font-medium
text-gray-600">Seats:</span>
                    <span className="text-base">{ride.available_seats ===
null ? "N/A" : ride.passengers.length + "/" + ride.available_seats}</span>
                </div>
                <div className="flex justify-between items-center">
                    <span className="font-medium
text-gray-600">Vehicle:</span>
                    <span className="text-base">{ride.vehicle === null ?
"Not defined yet" : <VehicleInfo vehicleId={ride.vehicle} />}</span>
                </div>
                </CardContent>
                <div className="p-4 pt-0">
                <div>
                <Button
                  className="w-full"
                  onClick={() => {
                    setSelectedRide(ride);
                    setIsOpen(true);
                  }}
                >
                  Leave a review
                </Button>
                </div>
                </div>
            </Card>
            ))}
          </div>
```

```
          <Separator className="my-2"/>


     {selectedRide && (
       //review apo ta rides
       <Drawer open={isOpen} onOpenChange={(open) => { setIsOpen(open); if
(!open) setSelectedRide(null); }}>
         <DrawerContent>
           <div className="mx-auto w-full max-w-sm">
             <DrawerHeader>
               <DrawerTitle>
                 Review the rides' users
               </DrawerTitle>
               <DrawerDescription>It will be anonymous</DrawerDescription>
             </DrawerHeader>
             <ReviewForm ride={selectedRide} userId={userData?.id}
closeDrawer={() => { setIsOpen(false); setSelectedRide(null); }} />
           </div>
         </DrawerContent>
       </Drawer>
     )}
       </div>
   );
}
```

Breaking down The PastBookings Segment

This component will show all rides that have occurred and for the user to leave a review. State management: React useState and useEffect hooks to manage user data, ride data, loading state and drawer state (for the review form).

Data fetching: It calls me() in a first place to get some details about the current authenticated user (user ID, driver status). When the user is established, you make API request (/rides/? user_id={id}) to gather all rides, which belong to this particular user.

Filtering logic: Only trips where the departure time is in the past(out of Date. now()) ensuring the user only sees finished trips.

Display: Displays the details of each ride (start/end location, departure, and arrival time, driver information with the UserInfo component, vehicle information with the VehicleInfo component, seat details) using the Shadcn CardWithImg component. If there are no past rides, a smiley face is drawn (with messages like Bruno hasn't given any rides yet or Flora has no past rides and very little ego) otherwise past rides are listed in a horizontally scrollable fashion.

Review system integration: There's a "Leave a review" button on each ride card. When clicked it opens a component Shadcn Drawer and it has the component ReviewForm. That allows the user to submit feedback for that ride specifically, and improves accountability and quality.

Styling: Styling is consistent and responsive (Tailwind CSS), the scrollable Nature of the layout allows for smooth scan through previous rides. This aspect is important for engaging users as it promotes reminiscence and user-generated reviews on which the platform becomes more interactive and credible.

Explanation : Forms

What Are Forms?

LoginForm: It is the form by which the old user can login by user name or password. It does input validation, presents an error message in case of login failure, and saves the auth token upon successful login in manner that the auth token can be utilized in a secure way to access protected routes.

SignUpForm : Registration form, where the new users can sign up for an account by submitting basic info like their username, email, password, and if they are driver or not. It does input validation for strong password and data format before the sending the data to backend.

ReviewForm: Users can fill this form after they finish a trip to provide reviews on the other members. The form usually has one or more fields: one for a rating, and an optional field for a comment, which helps build trust and accountability on the platform.

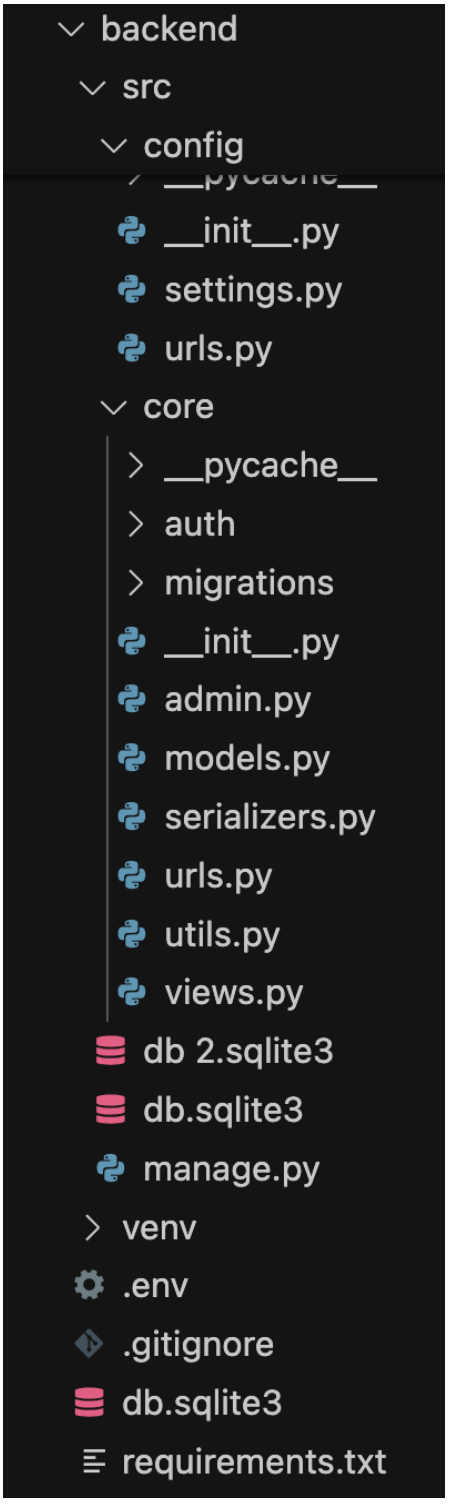
DriverRequestForm: This form is for users who wish to volunteer to be a driver for a ride that does not currently have one. It gives the user a way to verify that they are in fact driving and that their vehicle is assigned to the requested ride.

This allows for adaptable and on-the-fly matching of the drivers and the ride requests.

VehicleForm: Drivers enter or update information about their vehicles such as the brand, model, year, license plate, and the number of available seats. "Users being able to trust that the vehicle information is correct is of utmost importance for a secure carpooling experience."

Common Features Across Forms: All the shapes use the Shadcn UI components for consistent UI templates and behaviour. They have validation errors, ensuring the user is told there is an error with their input! Forms speak to the backend through AJAX requests so they both update in real-time, with no full page reloads. It uses TailwindCss across all aspects of the forms to keep them responsive, accessible and looking great.

6.3 Backend in detail



‘ core → models.py ‘

```
from django.contrib.auth.models import AbstractUser
from django.db import models
from django.contrib.auth.models import Group, Permission


class User(AbstractUser):
   phone_number = models.CharField(max_length=15, blank=True, null=True)
   privacy_enabled = models.BooleanField(default=False)
   is_driver = models.BooleanField(default=False)
   total_kilometers = models.DecimalField(max_digits=10, decimal_places=2,
default=None, blank=True, null=True)
   driving_license_number = models.CharField(max_length=20, blank=True,
null=True)

   groups = models.ManyToManyField(
       Group,
       verbose_name='groups',
       blank=True,
       help_text='The groups this user belongs to. A user will get all
permissions granted to each of their groups.',
       related_name="%(app_label)s_%(class)s_related",
       related_query_name="%(app_label)s_users",
   )

   user_permissions = models.ManyToManyField(
       Permission,
       verbose_name='user permissions',
       blank=True,
       help_text='Specific permissions for this user.',
       related_name="%(app_label)s_%(class)s_related",
       related_query_name="%(app_label)s_users",
   )

   def __str__(self):
       return self.username


class Vehicle(models.Model):
   owner = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='vehicles')
   make = models.CharField(max_length=255)
   model = models.CharField(max_length=255)
   year = models.IntegerField()
   license_plate = models.CharField(max_length=20)
   seats_available = models.IntegerField()

   def __str__(self):
       return f"{self.make} {self.model} - {self.license_plate}"
```

```
class Ride(models.Model):
   created_by = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='rides_created')
   driver = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='rides_driven', null=True, blank=True)
   vehicle = models.ForeignKey(Vehicle, on_delete=models.CASCADE,
null=True, blank=True)
   start_location = models.CharField(max_length=255)
   end_location = models.CharField(max_length=255)
   departure_time = models.DateTimeField()
   arrival_time = models.DateTimeField()
   available_seats = models.IntegerField(null=True, blank=True)
   passengers = models.ManyToManyField(User, related_name="joined_rides",
blank=True)

   def __str__(self):
       return f"{self.start_location} to {self.end_location} on
{self.departure_time.strftime('%Y-%m-%d %H:%M')}"


class Review(models.Model):
   ride = models.ForeignKey('Ride', on_delete=models.CASCADE,
related_name='reviews', null=True, blank=True)
   reviewer = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='given_reviews')
   reviewed_user = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='received_reviews', null=True, blank=True)
   rating = models.IntegerField()
   comment = models.TextField(blank=True, null=True)
   created_at = models.DateTimeField(auto_now_add=True)

   def __str__(self):
       return f'Review by {self.reviewer.username} on
{self.created_at.strftime("%Y-%m-%d")}'
```

Explanation : Models

What Are Models?

User Model: This model is derived from Django's AbstractUser to ensure the fields in the user model suit a carpooling system. It logs information such as the user's phone number, privacy settings (whether they want to show their real name), whether they are a driver, how many kilometers in total they have driven, and their driving license number. It further tailors the group and permission relationships to control the access and roles of the users.

Vehicle Model: This model depicts every registered vehicle in in the system. It's related to a user (the owner) and has some important information such as make, model, year, license plate number, and how many seats are available. It makes sure every vehicle in the system can be matched with a real driver and shown on ride offerings.

Ride Model: The carpool ride is modelled in the Ride model. It keeps track of the creator of the ride, its assigned driver, the vehicle used, its starting and ending locations, departure and arrival times, the number of available vacant positions, and a list of passengers. This design supports many-to-many relationship (for passengers) and all ride details are recorded for matching and booking.

Review Model: This model makes the review and rating system to work. It links a reviewer (the user who left the review) with a reviewed user (driver or rider), attaches the review to a ride, and record a numeric rating, an optional comment, and a time when the created the review. It is a trust-building and the overall platform quality-advancing measure by promoting responsibility.

‘ core → serializers.py ‘

```
from rest_framework import serializers
from .models import User, Vehicle, Ride, Review


class UserSerializer(serializers.ModelSerializer):
   password = serializers.CharField(write_only=True)

   class Meta:
       model = User
       fields = [
           'id', 'username', 'password', 'first_name', 'last_name',
'email', 'phone_number',
           'privacy_enabled', 'is_driver', 'total_kilometers',
           'driving_license_number', 'groups', 'user_permissions'
       ]

   def create(self, validated_data):
       password = validated_data.pop("password")
       user = User(**validated_data)
       user.set_password(password)
       user.save()
       return user


class VehicleSerializer(serializers.ModelSerializer):
   class Meta:
       model = Vehicle
       fields = ['id', 'owner', 'make', 'model', 'year', 'license_plate',
'seats_available']


class RideSerializer(serializers.ModelSerializer):
   class Meta:
       model = Ride
       fields = [
           'id',
           'created_by',
           'driver',
           'vehicle',
           'start_location',
           'end_location',
           'departure_time',
           'arrival_time',
           'available_seats',
           'passengers',
       ]
       read_only_fields = ['created_by']
```

```
    def create(self, validated_data):
        validated_data['created_by'] = self.context['request'].user
        return super().create(validated_data)


class ReviewSerializer(serializers.ModelSerializer):
    class Meta:
        model = Review
        fields = ['id', 'ride', 'reviewer', 'reviewed_user', 'rating',
'comment', 'created_at']
```

Explanation : Serializers

What Are Serializers?

Serializers do the job of translating model instances (Python objects) into a format (such as JSON, but not necessarily) that can be sent to the frontend in an API response, and of validating and converting incoming data from JSON into Python objects, where they can be saved to the database.

UserSerializer: this serializer deals with the user model. It also makes sure, that sensitive inputs (like passwords) can´t be read, but they can be written, that way one can set a password, but not retrieve it in plaintext. It contains all of the basic user information (username, email, name, phone, driver status, etc.) and has a custom create method to hash and save the password.

VehicleSerializer : This manages the vehicle model and it maps fields such as the owner of the vehicle, the type of vehicle (i.e. make), model, year and license plate as well as the available seats. This makes it possible for the system to serialize vehicle data and the ability to deserialize it conveniently while communicating with the frontend.

RideSerializer: This serializer is bound to the ride model and contains the main ride attributes: creator, the driver, the vehicle, the location, the time, the number of seats and the passengers. It sets the created_by field by default as it should be read-only and clients can't decide who created the ride and overrides the create method to automatically set this field from the request context.
ReviewSerializer: Helps in handling the review model, by including various fields such as the ride with which the review is associated, reviewer and the reviewed user, rating, comment and created. It makes sure that the review data is transferred in and out of the backend and frontend cleanly.

Serializers

They promote integrity and security of data (hiding sensitive fields, preventing what can be written). They make sure only valid data gets through to the database. Here they take care of an API response pattern, so the frontend integration easier. They offer the ability for data transformation logic to be customized for special cases (e.g., automagically populating fields, nested serialization).

‘ core → views.py ‘

```
from .utils import send_email
from rest_framework import viewsets, permissions
from rest_framework.permissions import IsAuthenticated, AllowAny
from rest_framework.decorators import action
from rest_framework.response import Response
from django.utils.dateparse import parse_date
from django.shortcuts import get_object_or_404
from .models import User, Vehicle, Ride, Review
from .serializers import UserSerializer, VehicleSerializer, RideSerializer,
ReviewSerializer
from django.db.models import Q


class UserViewSet(viewsets.ModelViewSet):
   queryset = User.objects.all()
   serializer_class = UserSerializer

   def get_permissions(self):
       """
       Instantiates and returns the list of permissions that this view
requires.
       """
       if self.action == 'create':
           permission_classes = [AllowAny]
       else:
           permission_classes = [IsAuthenticated]
       return [permission() for permission in permission_classes]

   @action(detail=False, methods=['get'], url_path='me',
permission_classes=[permissions.IsAuthenticated])
   def me(self, request):
       serializer = self.get_serializer(request.user)
       return Response(serializer.data)
```

```
    @action(detail=True, methods=['get'], url_path='public',
permission_classes=[permissions.AllowAny])
    def public(self, request, pk=None):
        """
        Returns public information for a given user.
        Only returns: isPrivacy, username, first_name, and last_name.
        """
        user = self.get_object()
        data = {
            'isPrivacy': user.isPrivacy,
            'username': user.username,
            'first_name': user.first_name,
            'last_name': user.last_name,
        }
        return Response(data)


class VehicleViewSet(viewsets.ModelViewSet):
    serializer_class = VehicleSerializer
    permission_classes = [IsAuthenticated]

    def get_queryset(self):
        return Vehicle.objects.filter(owner=self.request.user)

    @action(detail=True, methods=["GET"], permission_classes=[AllowAny])
    def public(self, request, pk=None):
        # Bypass the owner filter for public info
        vehicle = get_object_or_404(Vehicle, pk=pk)
        serializer = self.get_serializer(vehicle)
        return Response(serializer.data)


class RideViewSet(viewsets.ModelViewSet):
    queryset = Ride.objects.all()
    serializer_class = RideSerializer
    permission_classes = [IsAuthenticated]

    def get_queryset(self):
        queryset = super().get_queryset()
        origin = self.request.query_params.get('from')
        destination = self.request.query_params.get('to')
        date_str = self.request.query_params.get('date')

        if origin:
            queryset = queryset.filter(start_location=origin)
        if destination:
            queryset = queryset.filter(end_location=destination)
        if date_str:
            date_obj = parse_date(date_str)
            if date_obj:
```

```
                queryset = queryset.filter(departure_time__date=date_obj)

        user_id = self.request.query_params.get('user_id')
        if user_id:
            queryset = queryset.filter(Q(driver__id=user_id) |
Q(passengers__id=user_id)).distinct()


        return queryset

    def update(self, request, *args, **kwargs):

        instance = self.get_object()
        original_passengers =
list(instance.passengers.all().values_list('id', flat=True))
        original_driver = instance.driver.id if instance.driver else None

        response = super().update(request, *args, **kwargs)
        instance.refresh_from_db()

        new_passengers = list(instance.passengers.all().values_list('id',
flat=True))
        new_driver = instance.driver.id if instance.driver else None

        departure_time = instance.departure_time
        date_str = departure_time.strftime("%d/%m/%Y")

        time_str =
f"{int(departure_time.strftime('%I'))}{departure_time.strftime('%p').lower(
)}"


        if getattr(request.user, 'privacyEnabled', False):
            display_name = request.user.username
        else:
            display_name = f"{request.user.first_name}
{request.user.last_name}"


        if (request.user.id not in original_passengers and request.user.id
in new_passengers) and (new_driver != request.user.id):
            if instance.driver and instance.driver.email:
                message = (
                    f"{display_name} has joined your ride from
{instance.start_location} "
                    f"to {instance.end_location} on {date_str} at
{time_str}."
                )
                subject = "New Passenger Joined Your Ride"
                send_email(instance.driver.email, subject, message)
```

```
        if (original_driver != request.user.id) and (new_driver ==
request.user.id):
            message = (
                f"Driver {display_name} has joined your ride from
{instance.start_location} "
                f"to {instance.end_location} on {date_str} at {time_str}."
            )
            subject = "Driver Joined Your Ride"

            for passenger in instance.passengers.all():
                if passenger.email:
                    send_email(passenger.email, subject, message)

        return response

class ReviewViewSet(viewsets.ModelViewSet):
    queryset = Review.objects.all()
    serializer_class = ReviewSerializer

    def get_queryset(self):
        queryset = super().get_queryset()
        reviewed_user = self.request.query_params.get('reviewed_user')
        if reviewed_user:
            queryset = queryset.filter(reviewed_user=reviewed_user)
        return queryset

    def get_permissions(self):
        if self.action in ['list', 'retrieve']:
            return [AllowAny()]
        else:
            return [IsAuthenticated()]
```

Explanation: Views

What Are Views?

In Django REST Framework, views are the places which deal with the HTTP methods (GET, POST, PATCH, …). They manage the retrieval, processing, and returning of data as a response. Views are an embodiment of the control conventions for models, serializers, and permissions in the backend for what kind of request is allowed and how it is interpreted. Our Project's ViewSets We combine ModelViewSet classes (which implement the entire set of read and write operations - list, retrieve, create, update, delete following RESTful pattern).

UserViewSet: Manages user accounts. Includes my custom actions such as me (me returns the current logged in user data) and public (to get limited public user data). Applies permissions: anyone can sign up (AllowAny), but other actions must be signed (IsAuthenticated).

VehicleViewSet: Manages vehicles of all users. Filters queries the user see only their own vehicles. Its public action to obtain anyone's vehicle details.

RideViewSet: Manages carpool rides. Filters by origin, destination, date or user. Overrides the update method to include more logic, like notify users of the system (passengers and driver) by email when new passenger joins or the driver is informed about the new route.

ReviewViewSet: Handles user reviews of drivers or passengers. Option to filter reviews by reviewed user. Grants read access to everybody but write access only to authenticate users.

Views

They bridge requests from the frontend to the backend. They describe how user actions should be processed safely and efficiently. They link together models, serializers and permissions to piece together all the bits of creating a complete workflow. They enable behavior to be extended, such as by adding email notifications or filtering results on the fly.

‘ core→manage.py ‘

```
#!/usr/bin/env python
"""Django's command-line utility for administrative tasks."""
import os
import sys


def main():
   """Run administrative tasks."""
   os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'config.settings')
   try:
       from django.core.management import execute_from_command_line
   except ImportError as exc:
       raise ImportError(
           "Couldn't import Django. Are you sure it's installed and "
           "available on your PYTHONPATH environment variable? Did you "
           "forget to activate a virtual environment?"
       ) from exc
   execute_from_command_line(sys.argv)


if __name__ == '__main__':
   main()
```

The file manage. py is quintessential for every Django project. It’s a tiny Python script that is created during the creation of a Django project and is used as a command-line tool for interacting with and managing the project.

This script enables developers to perform administrative functions including: Run the dev server Applying database migrations Creating new apps Running the test suite Making a super user for your admin panel .

Without manage. py, it was not so easy to work with Django project or automate backend tasks. It’s a centralized application to manage all of these: database operations, testing, server operations and app scaffolding. this script is responsible for setting up your local settings, environment vars, project configuration in a correct way, each time you run anything from the backend;

it’s a basic tool necessary for working on Django.

‘ core → admin.py ‘

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from .models import User, Vehicle, Ride, Review

class CustomUserAdmin(UserAdmin):
   model = User
   # This defines the layout for editing existing users (no need to repeat
username and password here)
   fieldsets = (
       (None, {'fields': ('username', 'email', 'phone_number',
'privacy_enabled', 'is_driver', 'total_kilometers',
'driving_license_number')}),
       ('Permissions', {'fields': ('is_active', 'is_staff', 'is_superuser',
'groups', 'user_permissions')}),
       ('Important dates', {'fields': ('last_login', 'date_joined')}),
   )
   # This defines the layout for adding new users
   add_fieldsets = (
       (None, {
           'classes': ('wide',),
           'fields': ('username', 'email', 'password1', 'password2',
'phone_number', 'privacy_enabled', 'is_driver', 'total_kilometers',
'driving_license_number'),
       }),
   )

# Register your models with the admin site
admin.site.register(User, CustomUserAdmin)
admin.site.register(Vehicle)
admin.site.register(Ride)
admin.site.register(Review)
```

Explanation: Django Admin registering the model for the App.

The code snippet is part of the Django backend admin system. It explains how the project’s data models (User, Vehicle, Ride, Review) are handled in the Django Admin interface — a tool that is pre-built into Django and allows you to interact with database data using a web interface.

CustomUserAdmin Class

This class is a subclass of Django’s provided UserAdmin class and it provides you a way to customize how user objects are displayed and edited in the admin system. The sections' fieldsets are the sections we see while editing the existing

users. It groups together fields such as username, email, driver status, phone number, license number, and total kilometers in an organized fashion. add_fieldsets controls the layout of the fields when adding a user, it is also used for the password reset form, and the user change form, this sets which fields are displayed when creating a user this way including password fields and extra profile fields.

Model Registration

admin. site. register(User, CustomUserAdmin): It registers the User model with the Django admin with the custom layout defined in CustomUserAdmin.

admin. site. register(Vehicle), admin. site. register(Ride), admin. site. register(Review): Here you are telling Django that your Vehicle, Ride and Review model can be listed, added, modified (in the next step) and deleted directly from the Django admin interface.

' auth–>auth_views.py '

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticated
from rest_framework.authentication import TokenAuthentication
from rest_framework import status

class LogoutAPIView(APIView):
  authentication_classes = [TokenAuthentication]
  permission_classes = [IsAuthenticated]

  def post(self, request):
    request.user.auth_token.delete()
    return Response(status=status.HTTP_204_NO_CONTENT)
```

Explanation: LogoutAPIView (API Endpoint)

The LogoutAPIView is a Django REST Framework (DRF) view class which creates a secure API endpoint for authenticating →

users. authentication_classes: This tells us that this endpoint itself employs TokenAuthentication, i.e. the request that arrive to it must contain an authentication token in the →

header. permission_classes: Makes sure that only registered users (with token) can access this API. post : when a client makes a POST request to this endpoint, the server invalidates the user's token (request. user. auth_token. delete()) which in turn would logout the user.

Response: 204 No Content (in the case of a successful logout with no other data). This endpoint is a crucial part of the authentication flow, because it supervises that users' sessions are terminated in a secure manner. Old tokens are made inactive, which decreases the chances of unauthorized access. Promotes good security practice by allowing users to control their login state.

Βιβλιογραφία

Django. (n.d.). Django Project. https://docs.djangoproject.com

Home - Django REST framework. (n.d.). Www.django-Rest-Framework.org. https://www.django-rest-framework.org

Mozilla. (2019, August 21). *JavaScript Guide*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

React – A JavaScript library for building user interfaces. (2019). Reactjs.org. https://reactjs.org

Vercel. (2024). *Getting Started | Next.js*. Nextjs.org. https://nextjs.org/docs

Documentation - Tailwind CSS. (n.d.). Tailwindcss.com. https://tailwindcss.com/docs

The PostgreSQL Global Development Group. (2025). *PostgreSQL: Documentation*. Www.postgresql.org. https://www.postgresql.org/docs/

Fireship. (2020, May 21). *Node.js Ultimate Beginner's Guide in 7 Easy Steps*. YouTube. https://www.youtube.com/watch?v=ENrzD9HAZK4

freeCodeCamp.org. (2019). Python Django Web Framework - Full Course for Beginners. In *YouTube*. https://www.youtube.com/watch?v=F5mRW0jo-U4

Documentation - Tailwind CSS. (n.d.). Tailwindcss.com. https://tailwindcss.com/docs

shadcn. (2024). *shadcn/ui*. Shadcn/Ui. https://ui.shadcn.com/

MDN Web Docs. (n.d.). MDN Web Docs. https://developer.mozilla.org

Christie, T. (2011). *Django REST Framework*. Django-Rest-Framework.org. https://www.django-rest-framework.org/