Final Year Project Report – B.Sc. in Software Development

CarStop – Car Park Management Mobile App

Authors:
Justine Madriaga

Supervisors:
Evan Quinn

BSc in Software Development – Declaration of Authorship

I, Justine Madriaga, declare that the software development project titled “CarStop" submitted by me as part of my BSc in Software Development is entirely my own work, and I have not used unauthorized external assistance in its creation. I affirm that:

- I have fully and independently developed the software code, design, and documentation contained within this project.
- Any external sources, such as code snippets, libraries, or third-party resources, have been appropriately cited and referenced in accordance with academic citation standards.
- This project is a result of my own creative and intellectual efforts, and I have not copied, borrowed, or utilized work produced by others without proper acknowledgment.
- I have not used automated content generation tools, including AI-based code or text generation services, in the creation of this project.
- Any collaborations or contributions from external individuals are acknowledged and appropriately credited in the project documentation.
- The code and documentation presented here reflect my understanding of the concepts and principles learned during the course.

I understand that any breach of academic integrity, including plagiarism or submitting work that is not my own, is subject to disciplinary actions as outlined in the TUS academic policies.

[Signature] Justine Madriaga

[Date] 28/04/2025

Acknowledgements

I would like to thank my final year project supervisor Evan Quinn for helping with my research for this report.

Abstract

This project presents CarStop, a mobile-based parking management system designed to address common issues faced by drivers and parking operators, such as lack of real-time availability, inefficient reservation systems, and poor user experience. The application allows users to view live parking space availability using data collected from ultrasonic sensors connected to a Raspberry Pi. This data is stored in and retrieved from Firebase Realtime Database, ensuring that updates are reflected in the app almost instantly. Users can reserve spaces, top up their balance, extend bookings, and track reservation history through a clean and responsive interface built with Jetpack Compose.

A separate admin interface enables system administrators to monitor space usage, view live parking data, and remove reservations when necessary. Visual charts and dashboards were developed to provide insights into parking activity. While features such as licence plate recognition and full payment integration were initially planned, they were ultimately removed due to time constraints and hardware limitations. Instead, a simplified payment top-up method was implemented to simulate transactions.

Despite the limitations, CarStop successfully integrates hardware, cloud services, and mobile development into a single, functional system. The project demonstrates the real-world application of Internet of Things (IoT) concepts, Firebase integration, and user role differentiation within a mobile app. It offers a strong foundation for further development and shows the potential of combining embedded systems and mobile technologies to improve everyday challenges like parking.

Table of Contents

Table of Figures

Chapter 1 Introduction

1.1 Objectives

The core aim of this project is to develop a real-time car park management solution through a purpose-built mobile application. With urbanization increasing and space at a premium, the challenge of locating parking is not only frustrating for drivers but also costly in terms of wasted time, fuel, and emissions. This project seeks to solve this problem by designing an Android mobile app that gives users the ability to check real-time parking availability, reserve spaces, manage their bookings, and make payments — all from their smartphones. In tandem, the application integrates a live sensor system using Raspberry Pi hardware and ultrasonic distance sensors to simulate physical parking bays. By establishing this connection between physical input (vehicle presence) and digital output (app updates), the system bridges the gap between modern parking demands and currently outdated infrastructure.

In addition to user functionality, the application includes a built-in administrative interface. This allows an administrator to oversee the live status of all parking bays, remove user reservations, and view system analytics such as active reservations, occupancy rates, and distance readings. The project incorporates real-time Firebase database integration to facilitate these features. To simulate real-world use cases, features such as balance deduction, credit top-up, and reservation time extension are also implemented. Users will be able to add funds to their account and pay for parking based on the number of hours selected, with the system enforcing a five-hour limit. Through this combination of software and hardware, the solution delivers a complete, two-sided system for managing modern car parking scenarios.

1.1.1 Project Objectives

One of the primary objectives of this project is to deliver a functional mobile application that communicates seamlessly with a real-time cloud database and receives data from physical sensors. The app will distinguish between standard users and administrative users, showing different features depending on the login credentials. For users, the app will offer features such as real-time space tracking, reservation management, and simulated payments. Admins, on the other hand, will have access to detailed dashboards, data visualizations, and control mechanisms to monitor and manage the car park. All sensors input from the Raspberry Pi will be pushed to Firebase, from which the app will update the visual status of each space — showing whether a bay is available, occupied, or reserved.

The system aims to simulate a live car park using three ultrasonic sensors, each representing a parking space. These sensors will be wired to a Raspberry Pi, running Python scripts that continuously measure distances and determine whether a vehicle is present. If the space is occupied, the app will reflect this immediately. When a user reserves a space, it will appear yellow in the app, and turn red when occupied. The solution is designed to be as close to a real-world system as possible, albeit within the limits of simulation and available resources. The app also supports additional functionality such as cancelling reservations, handling situations with insufficient funds, and tracking reservation history.

1.1.2 Academic Objectives

Academically, this project offers the opportunity to apply skills learned across several modules throughout the course of the degree, particularly in areas like mobile development, networking, databases, and embedded systems. It demands a full software development lifecycle, from initial research and requirement analysis to system design, coding, testing, and documentation. Moreover, it demonstrates the student’s ability to integrate third-party services like Firebase, as well as physical components such as sensors, into a cohesive and working system. This fusion of technologies is particularly relevant in today’s software industry, where cross-platform and Internet of Things (IoT) solutions are increasingly common.

Another academic objective is to show proficiency in Android application development using Kotlin and Jetpack Compose, ensuring that the user interface is not only functional but also modern and accessible. Firebase is used for user authentication, real-time data updates, and cloud storage, reinforcing knowledge of backend services and mobile-first architecture. The project will also show understanding of asynchronous programming, API integration, real-time updates, and data-driven user interfaces. In addition, it enables practical experience in problem-solving, time management, and adapting project requirements during development, which are all key academic and professional skills.

1.2 Problem Domain

In many urban centres and university campuses, drivers often face the same dilemma — circling for extended periods in the hope of finding a parking space. The inefficiency of this system leads to not just driver frustration, but also increased congestion, CO_2 emissions, and missed appointments. Traditional parking setups are often unmanaged, rely on static signage, and offer no indication of real-time space availability. Furthermore, there is a growing demand for digital solutions that provide both visibility and control over parking space usage. Despite these challenges being well-known, there is a noticeable lack of smart parking systems in many localities, particularly solutions that are both affordable and adaptable for small-scale use, such as in university campuses or private lots.

From the perspective of parking operators, managing a car park manually is inefficient and leaves room for errors or underutilisation. There is often no easy way to enforce bookings, track usage statistics, or streamline payments. This project recognises the gap in the market and attempts to provide a dual-sided system where both users and operators benefit from digital oversight. With increasing smartphone usage and cloud-based technologies, the timing is right for a simple, affordable, and functional solution to emerge. The combination of sensor technology, real-time databases, and mobile apps provides an ideal framework to explore this problem domain.

1.3 CarStop: A Solution

The solution proposed in this project is an Android-based mobile application called CarStop. This app acts as the user's interface for checking parking availability, reserving bays, managing bookings, viewing transaction history, and managing account credit. The app also integrates a dashboard for admin use, displaying detailed information about the car park, including current status of all parking spaces, time remaining on reservations, and distance values from sensors. The app is connected to a Firebase Realtime Database, which stores all relevant information including user accounts, parking status, and payment records. It communicates with a Raspberry Pi that is programmed to monitor three ultrasonic sensors placed over a simulated parking layout.

Through this setup, CarStop aims to replicate the features of a modern parking solution without requiring complex or expensive infrastructure. The app also includes support for dark mode, a splash screen, and a clean navigation structure. In addition, the admin can manually override reservations or remove them entirely, and users will receive feedback such as pop-up messages and timers for their bookings. All changes are reflected live across the app, meaning if a reservation is removed by the admin, the user will see that update immediately in their "My Reservations" section. Overall, CarStop is built as a real-world prototype that can be adapted for commercial or educational use.

1.4 The Scope of the Solution

The scope of this project is clearly defined to cover Android development, real-time database integration, and hardware simulation using Raspberry Pi. The mobile app is developed entirely using Kotlin and Jetpack Compose, which allows for modern UI design and reactive data handling. The Firebase platform is used for user authentication, real-time database storage, and secure communication between the app and sensor system. The physical simulation includes three HC-SR04 sensors connected to the Raspberry Pi's GPIO pins, each representing one parking space. These sensors constantly send distance measurements, which are processed to determine the status of each bay available, occupied, or reserved.

The solution also covers user management, including sign-up, login, and different home screens depending on user type (admin or standard). The admin has additional tools, such as graphs and pie charts for visualizing space usage and can remove reservations at will. For users, the app supports all parking-related tasks, from finding a space to paying and cancelling. However, some features are intentionally scoped out of the current solution, such as integration with live payment gateways or larger-scale deployments. These limitations are acknowledged in the documentation, and recommendations for future work are provided. The focus remains on building a robust, modular, and scalable prototype that functions effectively within a simulated environment, while laying the groundwork for potential real-world applications.

Chapter 2 Literature Review

2.1 Introduction

Parking management has become a critical issue in urban areas, particularly as cities grow and vehicle ownership increases. Theres is often more cars than parking spaces, which leads to traffic, pollution and time wasted for drivers searching for available spots to park. Traditional parking systems rely on manual processes or static signage, which is proving to be inadequate in dealing with these modern challenges. As cities look to implement more efficient sustainable solutions, technology has emerged as a vital tool to address the problem.

In recent years, there has been a rise in car park mobile apps. In 2023, the market for smart parking had a projected value of $8.1 billion (Bamboo Apps, 2024). Mobile apps have transformed how parking is managed, providing drivers with real-time information on parking availability, accept mobile payments, send reminders when your time is low, and extending your allotted time (Carroll, 2016). These apps not only help drivers but also assist parking operators in monitoring and managing their facilities more effectively. However, while many solutions exist, there are still significant gaps in their implementation, such as inconsistent real-time data, cumbersome payment processes, and limited integration of user and admin functionalities within a single platform.

The objective of this project is to develop CarStop, an Android-based mobile application that integrates both user and admin functionalities. Drivers will be able to locate and reserve parking spots, while operators can manage spaces, track revenue, and view real-time data within the same app. This literature review will explore the technologies, methodologies, and existing solutions that will inform the development of this integrated system, providing a comprehensive foundation for the project’s design and implementation.

2.2 Background

The development of mobile applications has revolutionized the way car parks are managed and utilized in urban environments (Nizio, 2024). As cities grow bigger and the number of drivers on the road increase, the demand for more efficient parking solutions has become paramount. Traditional parking, management involved manual processes, paper-based systems, or static signage, which were inefficient and prone to human error. The introduction of digital and mobile solutions has brought significant improvements to this sector, allowing real-time updates, better space management, and easier access for drivers. Mobile applications have become central to this shift, offering a seamless experience for both users and parking operators.

Early iterations of parking management apps primarily focused on providing basic services, like using sensors for discovering available spots and simple payment functionality (Zheng, 2015). These early apps however, faced challenges with integrating real-time data, leading to issues like inaccurate availability updates. Over time, advancements in technologies such as GPS, the Internet of Things (IoT), and cloud computing have allowed parking apps to evolve into sophisticated platforms that provide real-time information on parking availability, route optimization, reservation systems, and seamless payment gateways.

Applications such as Q-Park, SpotHero, and JustPark have become market leaders by incorporating features like real-time space tracking, advanced booking, and dynamic pricing. However, these apps often still struggle with issues such as inconsistent data from parking lots, fragmented user interfaces, and a lack of comprehensive solutions for parking lot operators. While users benefit from features like ease of payments and real-time updates, operators need better tools to manage their spaces efficiently, monitor usage, and handle revenue tracking.

Despite their success, there is still considerable room for improvement in this market. Many parking apps fail to fully integrate the needs of both drivers and parking lot operators into a single, cohesive platform. Furthermore, the reliability of real-time data, user-friendly design, and security in payment processes remain ongoing challenges. This project, through the development of CarStop, aims to create an Android-based solution that addresses these gaps by offering an integrated platform for both users and operators. By leveraging existing technologies and improving the user experience, CarStop seeks to build on the advancements in this field while resolving common pain points faced by both drivers and parking lot managers.

2.3 Mobile Application Development

Developing the CarStop app for Android involves defining core functionalities such as real-time availability, reservations, and payments to create an efficient, user-friendly experience. The initial planning phase will focus on identifying these essential features, allowing the team to build a clear and practical service aimed at a seamless parking experience.

In the design phase, CarStop will utilize tools like Figma or Adobe XD to create prototypes that map out the user journey, prioritizing an intuitive UI and UX. Quick access to key features like maps, booking, and payment options will be central to the layout, and early feedback on prototypes will refine the design to meet user expectations.

The development process will use Android Studio, along with Kotlin and Java, to build a robust backend and responsive front end. Integrating external libraries for specific functionalities will ensure security and reliability while accelerating development.

Finally, a comprehensive testing phase including unit, integration, and security testing will verify the app's functionality and safeguard user data, especially in payment processing. Once launched on the Google Play Store, CarStop will continuously evolve, updating features based on user feedback to deliver an increasingly refined parking experience.

2.4 Artificial Intelligence

Artificial Intelligence (AI) has become a transformative force in the development of mobile applications across various industries, and parking management is no exception. AI-powered car park management systems offer compatibility with various platforms including mobile apps and smart city initiatives. This ensures accessibility for users, allowing them to easily locate available parking spaces, make reservations, and even pay seamlessly through digital platforms (Parker, 2024).

2.4.1 License Plate Text Recognition

License Plate Recognition (LPR) is one of the most practical applications of AI in the realm of parking management (Chang, 2004). LPR systems, often powered by Optical Character Recognition (OCR) technologies and enhanced through AI, enable the automatic detection and reading of vehicle license plates. This capability is crucial for improving the efficiency of parking facilities, especially in automated or unmanned systems. By recognizing and recording vehicle information without the need for human intervention, license plate text recognition can streamline entry and exit processes, improve security, and facilitate payment automation in parking lots.

2.5 Real-Time Parking Availability

Real-time parking availability is an important feature in modern parking applications, providing users with up-to-date information on open parking spaces. This functionality is essential for drivers navigating congested urban areas, where the search for available parking often leads to stress, traffic congestion, and wasted time. By offering real-time data, parking applications can significantly enhance the user experience, reduce unnecessary driving, and optimize space utilization.

There is a list of solutions that could be used to track parking availability in real-time. One solution uses the GPS and/or accelerometer sensors in a traveler's mobile phone to automatically detect when and where the traveler parked their car, and when they're released a parking slot (Xu, 2013). Another solution that might be easier to work with is the use of sensors on parking bays to detect cars parked on that spot.

On the operator side, real-time availability helps car park owners monitor how full their car parks are and adjust their strategies accordingly. They can track peak times, adjust pricing, and even use the data to optimize how their spaces are used, possibly by introducing dynamic pricing where costs fluctuate based on demand. This ensures that parking is well-managed and that operators get the most out of their spaces.

For drivers, this feature will save time and reduce the stress of parking, especially in areas that are usually packed. The option to reserve spaces ahead of time could also be added to CarStop, giving users peace of mind knowing they've secured a spot before they even arrive. By providing users with real-time information and predictive capabilities, CarStop aims to streamline the parking process, making it more efficient and less stressful for drivers, while also helping parking operators optimize their facilities.

2.6 Parking Reservations

The ability to reserve parking spaces in advance has become an increasingly popular feature in modern parking apps over the last 10 years (O'Donovan, 2020), offering users the convenience of securing a spot before arriving at their destination. This is particularly beneficial in busy urban areas or during peak times when parking can be hard to find. By integrating a reservation system into CarStop, users will have the peace of mind that a space is guaranteed, reducing the stress and uncertainty that often comes with parking in high-demand locations.

Parking reservations work by allowing users to select a location, date, and time for their parking needs. The app would show available spaces in real-time, and once a user reserves a spot, that space is marked as unavailable to others for the chosen time period. This system is especially useful for commuters or people who are visiting popular city centers where parking is scarce. For users, it eliminates the need to circle around looking for parking and provides a more organized, predictable experience.

Furthermore, this system could integrate seamlessly with other features such as license plate recognition. When a user arrives at the car park, the system could automatically verify their reservation by matching their license plate, allowing for a smooth entry without needing physical tickets or manual intervention. This would make the process even more convenient for users, as they can simply drive in and park, knowing their spot has been secured in advance.

By offering parking reservations, CarStop will provide a valuable service that improves the overall parking experience for both users and operators, ensuring convenience, efficiency, and better management of parking resources.

2.7 Payment Integration

Payment integration can be a vital feature for any mobile app, including CarStop as it can make it easier for users to pay (Stax Payments, 2022). This functionality not only streamlines the user experience by removing the need for physical payment methods but also supports faster, more efficient operations for parking providers. With payment integration, users can pay directly through the app, choose from various payment methods, and even receive digital receipts, making the entire parking process smoother and more modern.

Integrating payment capabilities into CarStop will allow users to handle all aspects of their parking journey within a single app. Users will be able to select a payment method such as credit card, debit card, mobile wallets like Google Pay, or even subscription options for frequent parkers at the point of booking or upon leaving the parking facility. By offering multiple payment options, the app can cater to a broader user base and meet the varied needs of drivers, whether they park occasionally or on a regular basis.

For the back end of the app, payment integration will be facilitated through a secure payment gateway like Stripe or PayPal (Stax Payments, 2022), both of which are compatible with mobile platforms and known for their robust security features. These platforms support PCI compliance, which ensures that all transaction data is encrypted and secure, protecting users from potential fraud. Additionally, payment gateways handle the complexities of currency conversion and tax calculation, enabling CarStop to operate smoothly across different regions if the app expands beyond its initial launch area.

With secure, efficient payment integration, CarStop will provide a comprehensive parking solution, making it easy for users to book, park, and pay all from their phones. This feature will support CarStop in offering a seamless, end-to-end parking experience that prioritizes convenience, security, and flexibility.

2.8 Conclusion

In conclusion, this literature review highlights the essential components and innovative technologies shaping the modern car park management landscape, with a focus on real-time availability, reservation systems, payment integration, and mobile application development. The integration of these features into a single mobile platform, like CarStop, represents a significant step forward in simplifying the parking experience, minimizing congestion, and addressing user frustrations commonly associated with traditional parking. By leveraging advancements in IoT, machine learning, and mobile payment solutions, CarStop aims to provide users with a more streamlined, reliable, and accessible way to locate, reserve, and pay for parking spaces in real-time.

The research underscores the importance of a well-structured mobile application that prioritises user experience and data security, particularly when handling sensitive payment information. The review also reveals how predictive analytics and machine learning algorithms can enhance functionality by anticipating parking availability, further improving convenience and efficiency for users. As cities grow more congested, the demand for efficient and tech-driven parking solutions will only increase, indicating a valuable opportunity for applications like CarStop.

Future developments in mobile parking applications will likely explore even deeper integrations with AI, enhanced data analytics, and expanded IoT capabilities. These trends point toward a more connected and user-centred approach, where real-time and predictive features are seamlessly woven into the user journey. Through a cohesive integration of these features, CarStop aspires to meet the growing demand for accessible, efficient, and tech-enabled parking solutions, ultimately contributing to a smarter, more efficient urban environment.

Chapter 3 Analysis and Design

3.1 Analysis

The analysis phase focuses on identifying user needs, system requirements, and potential challenges that could arise during the development and implementation of the CarStop mobile application. This comprehensive evaluation ensures that the final product meets the needs of both drivers and parking lot operators, providing a seamless experience with essential functionalities.

To begin with, a user needs analysis was conducted to determine the most common frustrations faced by drivers when searching for parking, such as lack of availability, inefficient payment methods, and difficulties finding spots in busy areas. Similarly, input from parking operators highlighted challenges in monitoring space usage, managing reservations, and maximising revenue. Based on these findings, CarStop aims to integrate solutions such as real-time space availability tracking, intuitive booking features, and secure payment processing.

3.1.0 Budget

As a single developer working on the project, the costs will primarily involve software tools, basic hardware for simulation, and minimal deployment expenses. The following breakdown outlines the estimated budget:

1. Development Costs

o Software Tools: Open-source or free tools will be prioritised, such as Android Studio for development and Figma (free version) for UI/UX design.

Estimated Cost: €0–€50 for any additional plugins or premium tools.

2. Infrastructure Costs

o Cloud Hosting: A minimal-tier hosting solution (e.g., Firebase or AWS free tier) can be used to support the backend and real-time database for testing and deployment.

Estimated Cost: €0–€50 (depending on scale and trial limits).

o IoT Sensor Simulation: Since actual sensors and large-scale IoT devices may be costly, a simulated setup or inexpensive components (e.g., Arduino or Raspberry Pi with basic sensors) will be used for proof of concept.

Estimated Cost: €50–€150.

3. Licensing and Integration

o Payment Gateway Fees: Integration with payment systems like Stripe often incurs no upfront cost but includes small fees per simulated transaction for testing.

Estimated Cost: €10–€20 for testing a few mock transactions.

o API Access: Google Maps and Firebase provide free tiers for student projects, sufficient for basic testing and deployment.

Estimated Cost: €0–€10.

4. Testing and Deployment

o Testing Tools: Free tools such as Firebase Test Lab or Android emulators will be used for QA. Physical device testing (using personal or borrowed Android devices) will be sufficient for small-scale validation.

Estimated Cost: €0.

o Deployment Fees: Publishing the app on the Google Play Store involves a one-time developer account fee.

Estimated Cost: €25.

5. Hardware Simulation Costs

To simulate a parking lot environment, physical props (e.g., paper labels or small markers) can be combined with basic hardware components like Raspberry Pi, Arduino boards, or inexpensive sensors.

Estimated Cost: €50–€100.

Total Estimated Budget

- Development Tools and Software: €0–€50
- Cloud Hosting and API Fees: €0–€60

- IoT Simulation and Hardware: €50–€150
- Testing and Deployment: €25
- Miscellaneous Expenses: €0–€50

Total Estimated Budget Range: €75–€275

3.2 Use Cases

3.2.1 User Login & Registration

- **Description**: This use case allows new users to register for an account and existing users to log in.
- **Actors**: User (Driver), Admin (Operator)
- **Preconditions**: The app must be installed on the user's device, and an internet connection must be available.
- **Basic Flow**:
    1. The user opens the app.
    2. If the user is new, they select "Register" and provide the necessary details (e.g., name, email, password, vehicle info).
    3. Existing users select "Login" and enter their credentials.
    4. The system verifies credentials.
    5. Upon successful login or registration, the user is redirected to their respective dashboard (User or Admin view).
- **Postconditions**: The user is authenticated and can access app functionalities.
- **Alternate Flows**:
    - Invalid credentials: Display error message prompting the user to try again.

Forgotten password: User selects "Forgot Password" to initiate password recovery.

3.2.2 Finding and Viewing Available Parking Spots

- **Description**: Users can search for available parking spots based on location, availability, and preferences.

- **Actors**: User (Driver)
- **Preconditions**: The user must be logged in, and location services must be enabled.
- **Basic Flow**:
    1. The user opens the "Find Parking" feature.
    2. The user selects filters such as location, type of spot (e.g., standard, reserved), and availability timeframe.
    3. The system retrieves real-time data from IoT sensors and displays available spots on a map or list view.
    4. The user selects a parking spot to view additional details.
- **Postconditions**: The user has access to detailed information about the selected parking spot and can proceed to reserve it.
- **Alternate Flows**:
    - No spots available: The app displays a message suggesting alternative options.

3.2.3 Parking Reservations

- **Description**: Users can reserve parking spots for a specific period.
- **Actors**: User (Driver)
- **Preconditions**: The user must be logged in and have a valid payment method linked to their account.
- **Basic Flow**:
    1. The user selects an available parking spot.
    2. The user chooses a reservation timeframe.
    3. The system confirms availability and displays the total cost.
    4. The user confirms the reservation and proceeds to payment.
    5. The system processes the payment and confirms the reservation.
- **Postconditions**: The spot is reserved, and the user receives a confirmation notification.
- **Alternate Flows**:
    - Spot no longer available: Display an error message and suggest other spots.

3.2.4 View Parking Reservations

- **Description**: Users can view their reservations.
- **Actors**: User (Driver)
- **Preconditions**: The user must be logged in.
- **Basic Flow**:
    1. The user selects my reservations button.
    2. The system displays a list of the users current and previous reservations.
- **Postconditions**: The user can now see their reservations.

3.2.5 Cancel Parking Reservations

- **Description**: Users can cancel their reservations.
- **Actors**: User (Driver)
- **Preconditions**: The user must be logged in.
- **Basic Flow**:
    1. The user selects my reservations button.
    2. The system displays a list of the users current and previous reservations.
    3. The user selects the cancel button on their active reservation.
    4. The system displays a message asking the user to confirm.
    5. The user selects confirm.
- **Postconditions**: The user's reservation has now been cancelled.

3.2.6 View Payment History

- **Description**: Users can view their previous transactions.
- **Actors**: User (Driver)
- **Preconditions**: The user must be logged in.
- **Basic Flow**:
    1. The user selects payment history button.

2. The system displays a list of the user's previous transactions

- **Postconditions**: The user's can now see their payment history.

3.2.7 Admin Parking Space Management

- **Description**: Admins can view and cancel parking reservations.
- **Actors**: Admin (Operator)
- **Preconditions**: The admin must be logged in.
- **Basic Flow**:
    1. The admin navigates to the "Manage Parking" section.
    2. The admin selects a reservation to be removed.
    3. The system processes the changes and updates the database.
- **Postconditions**: The changes are reflected in real-time for users.
- **Alternate Flows**:
    - Invalid input: Display an error message and prompt for correction.

3.3 System Architecture Design

System Architecture Overview

The system consists of four layers: the IoT layer (for sensor and LPR data), the backend layer (for processing and managing data), the data layer (for storage and synchronisation), and the client layer (the CarStop mobile app). These layers communicate seamlessly to provide a robust and user-friendly parking experience.

IoT Layer

The IoT layer integrates parking sensors and LPR systems to monitor parking spaces and verify vehicle identity.

1. **Sensors for Parking Space Monitoring**:
   Sensors such as ultrasonic, infrared, or magnetic devices are installed in each parking spot to detect the presence of a vehicle. Each sensor is assigned a unique

ID corresponding to a parking spot in the database. The sensors communicate their status (available or occupied) to a local IoT gateway.

2. **License Plate Recognition Cameras**:
   LPR cameras are installed at parking lot entrances and exits. These cameras capture license plate images of vehicles entering or leaving. The images are processed by onboard software or sent to the backend for Optical Character Recognition (OCR). Once the license plate is identified, the system matches it with active reservations or registered vehicles in the database.
3. **IoT Gateway**:
   Both sensors and LPR systems transmit data to the IoT gateway. The gateway aggregates this data and sends it to the backend via secure communication protocols like MQTT or HTTPS.

Backend Layer

The backend implemented using Node.js with Express.js, processes data from the IoT layer and manages user interactions. Key responsibilities of the backend include:

1. **Real-Time Data Processing:**
   The backend receives parking status updates from sensors and updates the database. For example, when a sensor detects that Spot 101 is occupied, the backend updates its status in real time and pushes this update to the app using WebSocket connections.
2. **License Plate Verification:**
   LPR data from cameras is processed to extract license plate numbers using OCR. The backend matches the extracted license plate against the database to:
   - Confirm if the vehicle has a valid reservation.
   - Automatically update the status of the associated parking spot to “Occupied.”
   - Trigger automated entry or exit by lifting the barrier gate if required.
3. **Reservation Management:**
   Users can reserve parking spaces via the app, and the backend ensures the reserved spots are marked as unavailable in the database. If the user’s license plate matches the reservation, the LPR system grants entry.
4. **Payment Processing:**
   The backend integrates with payment gateways like Stripe or PayPal to handle secure transactions. LPR simplifies payment workflows by automatically identifying the vehicle and charging the associated account upon exit.
5. **Notification Service**:
   Real-time notifications, such as reservation confirmations, availability updates, or payment receipts, are sent via Firebase Cloud Messaging (FCM).

Data Layer

The data layer manages all information related to parking spaces, reservations, users, and transactions.

1. **Database**:
    - PostgreSQL stores structured data, including user profiles, reservations, payment records, and parking lot configurations.
    - Firebase Firestore or a similar NoSQL database handles dynamic, real-time updates from sensors and LPR systems.
2. **LPR Data Storage**:
   License plate images and associated metadata are stored temporarily for processing, ensuring compliance with privacy regulations by automatically deleting the data after a predefined period.

Client Layer (Frontend)

The CarStop Android app serves as the primary interface for users and parking operators. The app adapts its UI based on the user role (driver or admin).

1. **Driver Features**:
    - View available parking spaces in real time with color-coded markers.
    - Make reservations and pay for parking through the app.
    - Receive automatic entry and exit via LPR without manual intervention.

2. **Admin Dashboard**:
    - Monitor parking lot occupancy in real time, with data from sensors and LPR cameras.
    - Manage parking spaces, update statuses, and resolve errors (e.g., faulty sensors or mismatched license plates).
    - View analytics and reports, including revenue and usage patterns.

Data Flow and Synchronisation

The system follows a structured data flow to ensure real-time accuracy and user convenience.

1. **Entry Process with LPR**:
    - As a vehicle approaches the parking lot, the LPR camera captures its license plate.

    - The backend verifies if the vehicle has an active reservation or is registered for automated entry.
    - If verified, the system marks the spot as “Occupied” and triggers entry (e.g., opens the gate).
2. **Sensor Data Updates**:
    - Parking sensors detect vehicle presence and send updates to the IoT gateway.
    - The gateway relays this information to the backend, which updates the database and pushes changes to the app.
3. **Exit Process with LPR**:
    - Upon exit, the LPR camera captures the license plate and verifies the parking session.
    - If payment is due, the system automatically charges the user’s account and marks the spot as “Available.”

Security Considerations

The architecture includes robust security measures:

- **Data Encryption**: All communication between sensors, the backend, and the app is encrypted using SSL/TLS protocols.
- **Privacy Compliance**: License plate data is handled in compliance with GDPR or other applicable regulations, with strict access controls and automatic deletion after processing.
- **Role-Based Access Control (RBAC)**: Sensitive features are restricted to authorised users, ensuring system integrity.

Scalability and Future Expansion

The system is designed to scale easily. Additional sensors and LPR cameras can be integrated into the IoT layer with minimal modifications to the backend. Cloud hosting ensures the system can handle increased user demand and expanded parking facilities.

By incorporating both sensors and license plate recognition, CarStop delivers a modern, efficient, and automated parking management system. This architecture not only enhances the user experience but also optimises operations for parking lot managers, ensuring accuracy, security, and scalability.

3.3.1 System Architecture Design Diagram

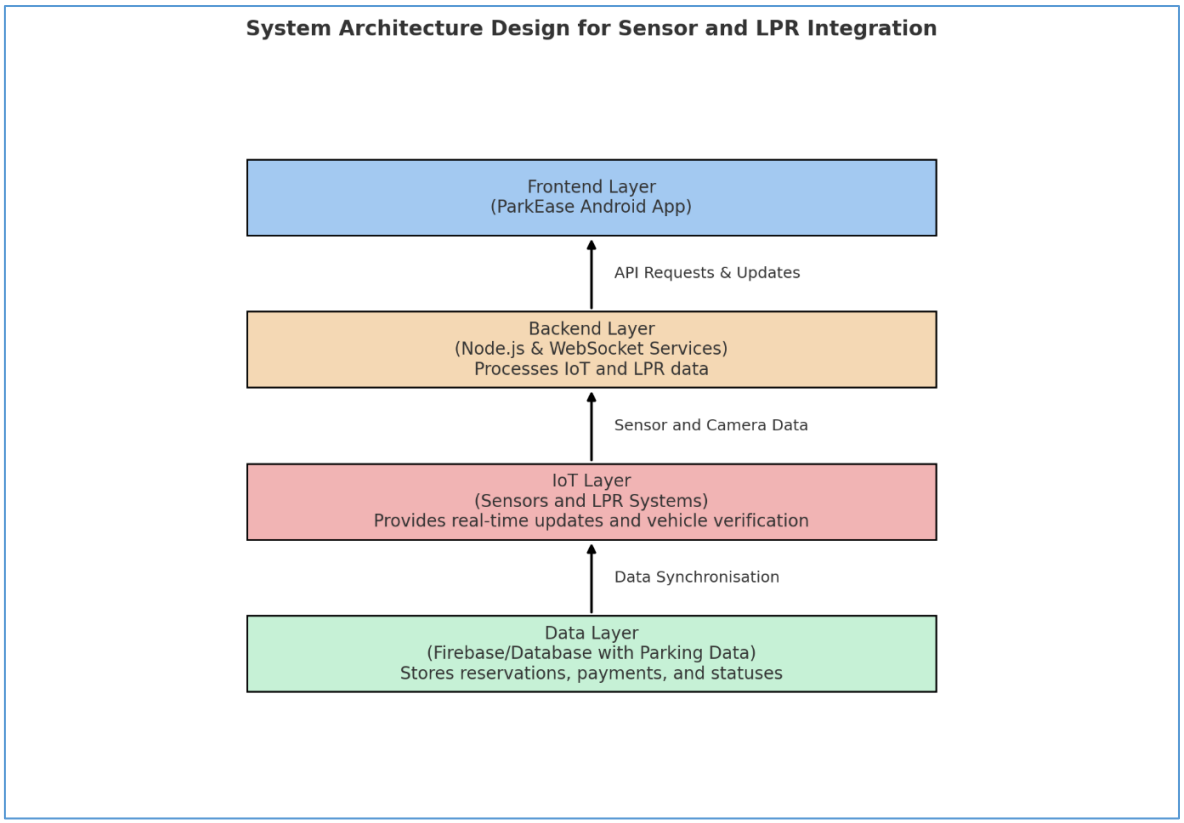


Figure 1: System Architecture Diagram

The system architecture for CarStop integrates IoT sensors and License Plate Recognition (LPR) technology across four layers: the Frontend, Backend, IoT, and Data layers. The Android app serves as the user interface for drivers and operators, enabling reservations, payments, and real-time updates. The Backend Layer processes data from IoT sensors and LPR cameras, managing workflows like vehicle verification, space availability updates, and notifications. Sensors monitor parking spots, while LPR cameras identify vehicles at entry and exit points, transmitting data through a gateway to the backend. The Data Layer stores user profiles, reservations, and payment records in a structured database, ensuring seamless synchronisation and scalability. This architecture allows automated parking operations, real-time updates, and secure, efficient management for both users and operators.

3.4 Hardware Requirements

Hardware	Manufacturer	Quantity	Function	Reason For Choosing	Connection	Cost
Laptop	HP	1	Main computer for coding	N/A	N/A	Already owned
Camera		1	For license plate recognition	N/A	N/A	N/A
HC-SR04 Sensors		5+	For detecting parking spots	Most affordable option		€5
Raspberry Pi	Sony	1	For sensors			€50
Breadboard kit		1	For connecting everything together	Only way to connect sensors		€20

Table 1: Hardware Requirements

3.5 Software Requirements

Software	Manufacturer	Function	Reason For Choosing	Cost
Android Studio	Google	IDE that will be used for coding	It's what use in mobile app development	Free
Kotlin	JetBrains	Language that will be used	We are also using this in app development	Free
PowerShell	Windows	To SSH to Raspberry Pi and write pythons scripts	Already installed with Windows	Free
Firebase	Google	For storing data	Used in mobile app development	Free

Table 2: Software Requirements

3.6 User Interface

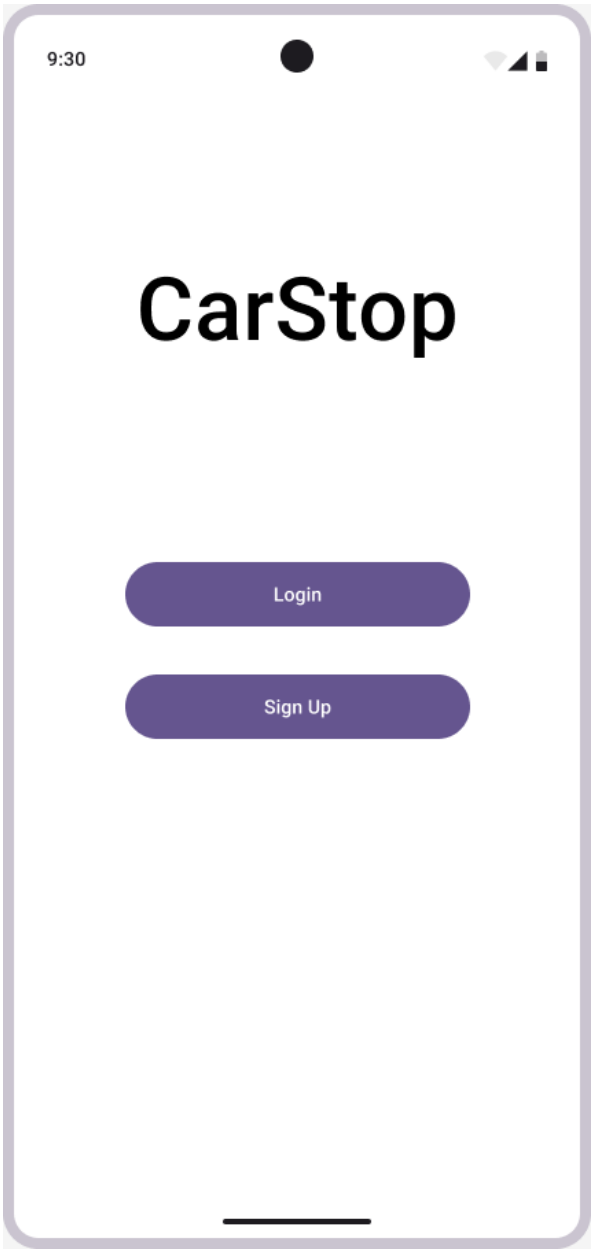


Figure 2: Login Screen

CarStop's login screen that is used to login for both admins and drivers.

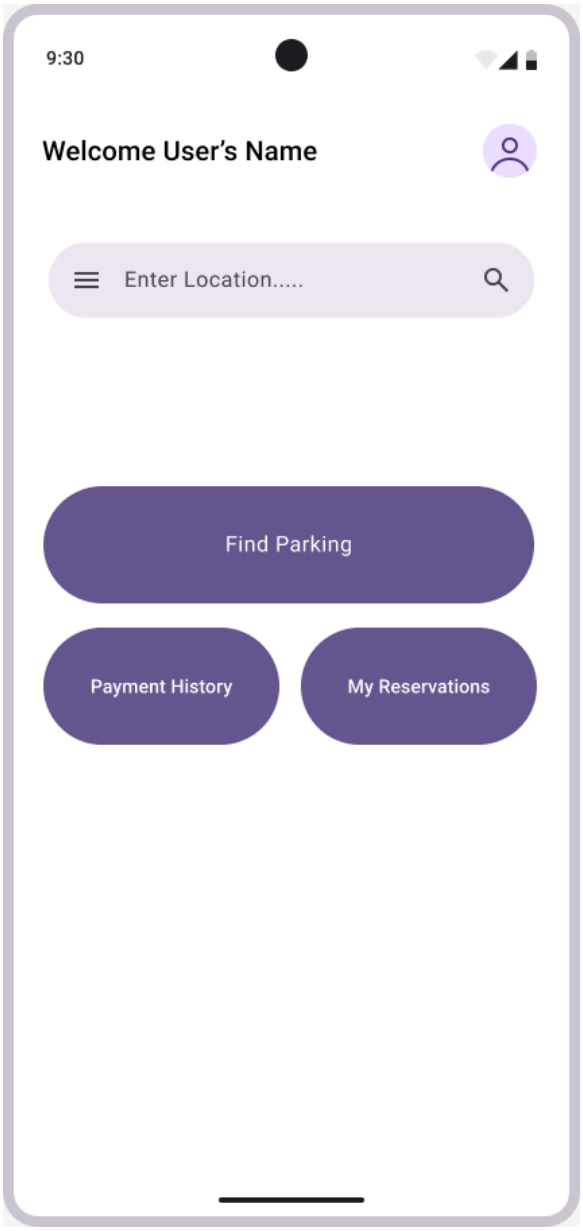


Figure 3: Home Screen

This is the home screen of the "CarStop" app, greeting the user by name. It includes a search bar to enter a location, along with three main buttons: "Find Parking," "Payment History," and "My Reservations." The design is clean and user-friendly, with a consistent purple colour scheme. A profile icon is located at the top right for easy access to user settings.

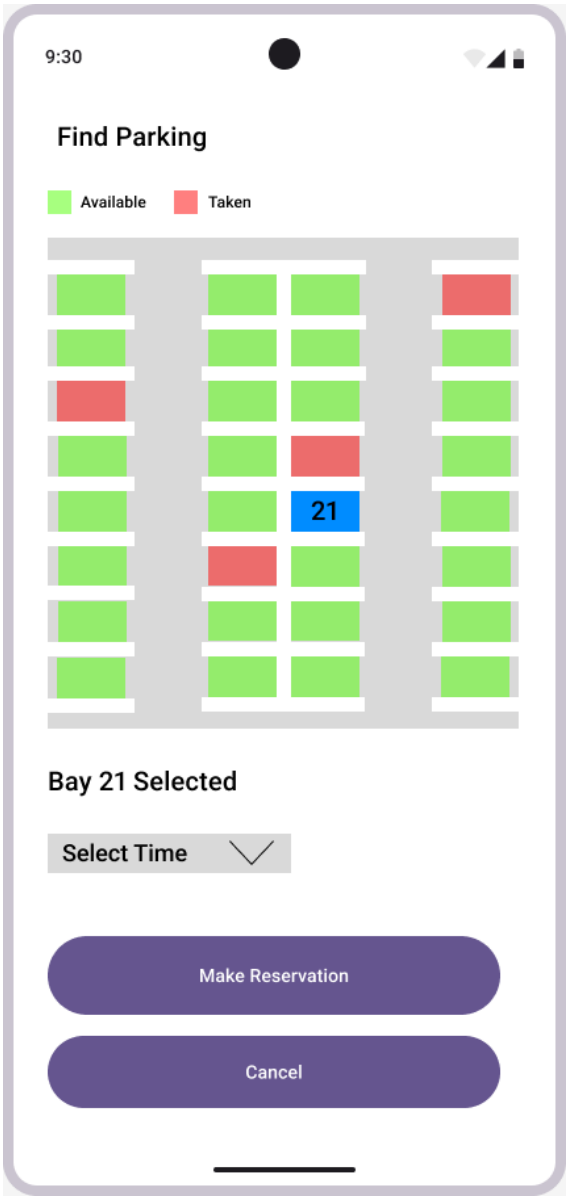


Figure 4: Find Parking Screen

This is the "Find Parking" screen, where users can view available and taken parking spots. The parking grid shows green for available spots, red for taken ones, and blue highlights the selected spot (Bay 21). Below the grid, users can select a time for their reservation. Two buttons are available: "Make Reservation" and "Cancel." The screen offers a simple and intuitive interface for users to find and reserve parking spots.

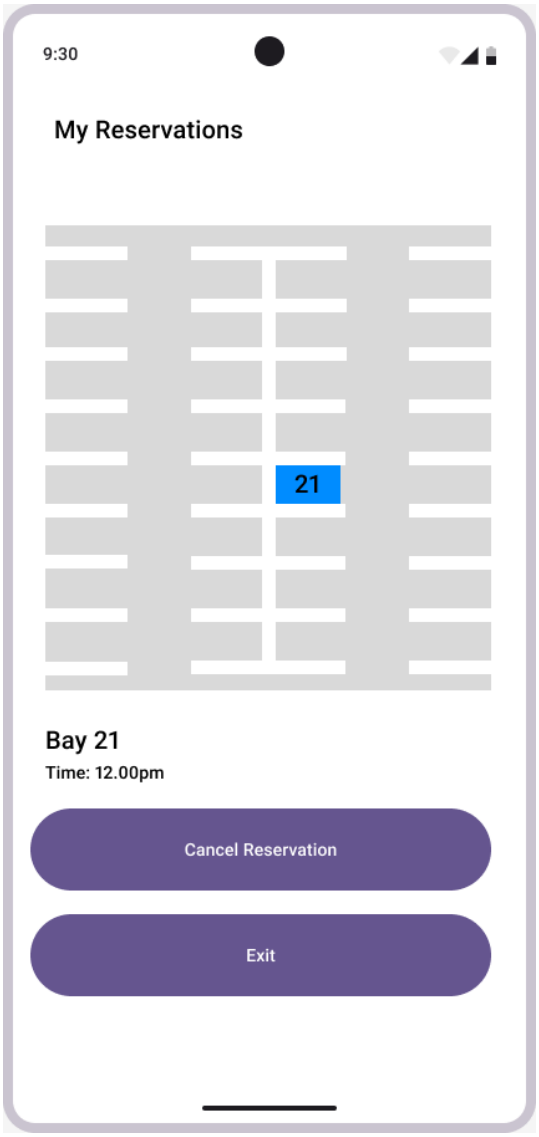


Figure 5: My Reservations Screen

This is the "My Reservations" screen, where users can view their current reservation. The selected parking spot (Bay 21) is highlighted in blue, and the reservation time (12:00 pm) is displayed below. Users can either cancel their reservation using the "Cancel Reservation" button or exit the screen with the "Exit" button. The layout is simple, offering an easy way for users to manage their parking reservations.

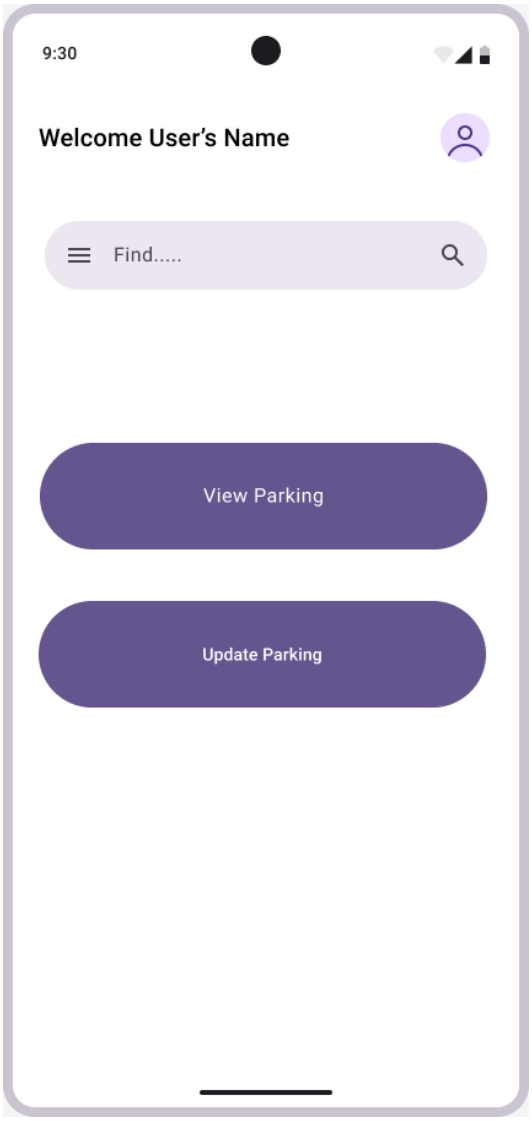


Figure 6: Admin Screen

This is the admin home screen for the "CarStop" app. It displays a personalized welcome message at the top, followed by a search bar labelled "Find" for quick access to parking-related information. Below, there are two primary action buttons: "View Parking" and "Update Parking." These options allow the admin to manage and update parking details. The layout is simple and efficient, with easy navigation to key functions for the admin user.

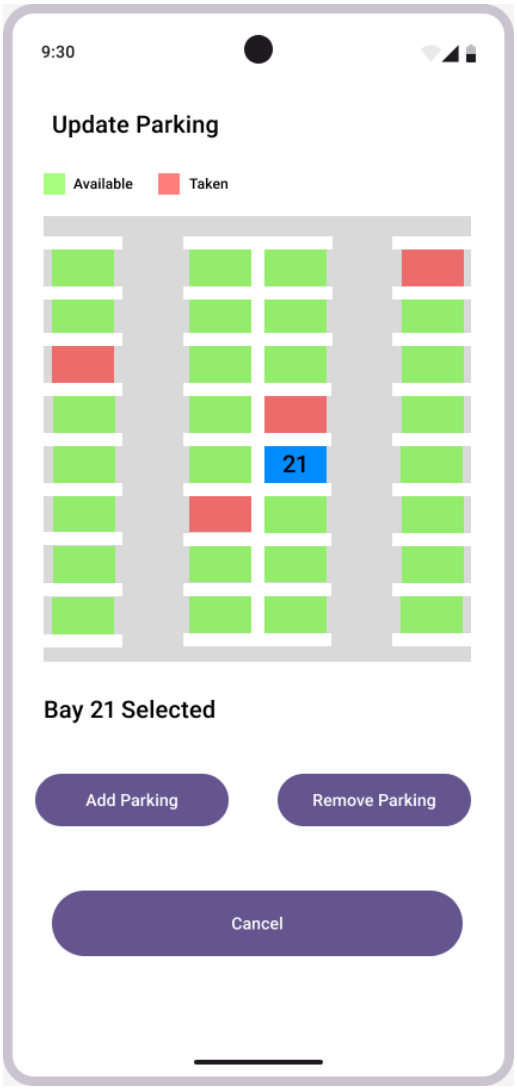


Figure 7: Update Parking Screen

This is the "Update Parking" screen for the admin. It displays a parking grid where available spots are shown in green and taken spots in red. Bay 21 is highlighted in blue as the selected spot. Below the grid, the admin can either "Add Parking" or "Remove Parking" for the selected spot. A "Cancel" button is also available to exit the screen without making changes. The interface allows the admin to efficiently manage the parking layout.

3.7 Implementation

The implementation phase involves transforming the design and analysis into a functional prototype of the CarStop application. This phase focuses on building the app's core features, integrating IoT simulation, and ensuring that all components—frontend, backend, and database—function seamlessly to deliver the desired user experience. The implementation will be carried out iteratively, with each phase tested thoroughly to ensure reliability and performance.

3.7.1 Development Environment Setup

The first step in the implementation is setting up the development environment. Android Studio will be used as the primary integrated development environment (IDE) for building the CarStop Android app, with Kotlin as the programming language. For backend services, Node.js with Express.js will be employed to manage API endpoints and real-time communication. Firebase will serve as the backend database for real-time updates and user authentication.

A testing environment will be established using Android emulators and a physical Android device. To simulate parking spaces, basic hardware like Raspberry Pi or Arduino will be connected with sensors, or software-based simulations will emulate parking lot conditions.

3.7.2 Core Feature Development

The app's development will focus on the following core features:

1. User Authentication and Role-Based Access

The first module to be implemented will handle user registration, login, and role-based access. Firebase Authentication will manage user credentials, ensuring security and scalability.

2. Real-Time Parking Availability

The app will display parking availability in real-time, using simulated sensor data stored in Firebase. A WebSocket or Firebase real-time database listener will push updates to the frontend, ensuring live status changes are reflected.

3. Parking Reservation and Payment Integration

Users will be able to reserve parking spots and pay directly within the app. Stripe's payment gateway will be integrated to handle secure transactions, with dummy payment data used for testing during the implementation phase.

4. Admin Dashboard for Parking Management

An admin interface will allow parking operators to add or remove spaces, view reservations, and monitor parking lot occupancy. This module will also include basic analytics for tracking revenue and usage patterns.

5. License Plate Recognition Integration

A simple LPR system will be integrated using image processing libraries like OpenCV. The system will process static or live images (simulated) to match license plates with reservation data, enabling automated entry and exit.

3.7.3 IoT Simulation

For IoT integration, a local simulation of parking sensors will be developed. This involves:

- Setting up basic sensors (e.g., ultrasonic or infrared) connected to a microcontroller (like Raspberry Pi or Arduino).
- Simulating real-time changes in parking spot status (occupied or available) based on these sensors.
- Transmitting data from the sensors to the backend through HTTP or MQTT protocols, ensuring the app reflects changes accurately.

If hardware is unavailable, a software-based simulation using Python scripts or mock data updates in Firebase will emulate the behaviour of physical sensors.

3.7.4 Testing and Debugging

As each feature is implemented, unit and integration testing will ensure all components work together seamlessly. For example:

- Authentication workflows will be tested for edge cases, such as invalid logins or missing fields.

- Parking availability will be tested with multiple simulated users to verify real-time updates.
- Payment functionality will be validated with dummy transactions to ensure secure and error-free processing.
- LPR functionality will be tested using a sample dataset of license plate images.

3.7.5 Deployment

The final step involves deploying the app for demonstration. The backend services will be hosted on a free-tier cloud platform like Firebase or AWS, while the app will be compiled and installed on physical Android devices for testing and demonstration. A working prototype will be submitted to the Google Play Store for review, ensuring it is accessible to users for feedback and further development.

This phased implementation approach ensures that CarStop is developed systematically, focusing on quality and functionality while addressing potential challenges through continuous testing and iteration.

Chapter 4 Implementation

This chapter details the practical development of the CarStop smart parking detection system. It describes the end-to-end process of integrating hardware (a Raspberry Pi and ultrasonic sensor) with cloud-based infrastructure (Firebase Realtime Database) using Python. The aim of this implementation phase was to simulate a working parking bay that can detect the presence or absence of a vehicle and reflect that status in real time through a cloud-connected system. Each stage of the build was tested thoroughly to ensure that both the sensor and data transmission systems worked reliably. The implementation phase marks a significant milestone in bridging the gap between theoretical planning and functional real-world application.

4.1 Development Environment

The system was developed across both hardware and software platforms. Android Studio served as the core IDE for mobile app development. Kotlin, alongside Jetpack Compose, was used for building reactive user interfaces. Firebase was configured to handle authentication, database storage, and data syncing. The Raspberry Pi 4 was used to represent a car park controller, simulating three parking bays using ultrasonic sensors. Python was used to interface the Raspberry Pi's GPIO pins with the sensors and to send updates to Firebase.

To allow efficient development on the Pi without a dedicated monitor, SSH access was enabled. Code for the Pi was developed and tested using VS Code and terminal access. Firebase configurations were managed through the Firebase Console, while testing on the mobile side was done using both Android Emulator and physical devices.

4.2 Technologies and Tools Used

The system uses a combination of technologies across layers. Kotlin powered the mobile interface, while Python ran the core scripts for sensor input on the Pi. Firebase was central to data communication — its real-time syncing capability allowed live updates between the sensor readings and app state. Google's authentication system enabled secure user logins. Data visualisation was handled through the MPAndroidChart library, which provided pie charts and bar graphs in the admin dashboard. For mapping functionality, the Google Maps SDK was used to display a map centred on the TUS Moylish campus.

On the hardware side, the system included three ultrasonic HC-SR04 sensors, connected to a Raspberry Pi 4. These sensors measured distance and helped determine whether a car was present in a simulated bay. A breadboard and jumper wires allowed non-permanent connections, and a logic level converter ensured safe communication between 5V sensors and the Pi's 3.3V GPIO pins.

4.3 Hardware Setup & Wiring

The CarStop prototype combines hardware and software to simulate a smart parking solution. At its core, the system uses a Raspberry Pi 4 to read data from an ultrasonic sensor (HC-SR04) to determine whether a parking space is occupied. This information is then transmitted to a Firebase Realtime Database, which serves as the cloud-based backend for the system. The sensor readings are processed in Python, and based on a defined distance threshold, the parking bay is marked as either "Available" or "Occupied." The data in Firebase can later be accessed by a mobile app to display real-time parking availability to users.

The goal of this system is to demonstrate the feasibility of using inexpensive, widely available components to replicate core functionality found in commercial smart parking solutions. While only one sensor has been implemented at this stage, the system has been designed to allow for easy scalability — multiple sensors can be added to monitor several parking spaces at once.

4.4 Python Scripting & Testing

A Python script was written to control the HC-SR04 sensor using the RPi.GPIO library. The script sends a 10-microsecond pulse to the TRIG pin, causing the sensor to emit an ultrasonic signal. It then measures the time taken for the signal to bounce off a nearby object and return to the ECHO pin. This time is used to calculate the distance to the object in centimeters using the formula:

distance = (time * speed_of_sound) / 2

If the distance measured is below a certain threshold (e.g., 10 cm), the system assumes that a car is parked over the sensor and labels the spot as "Occupied." If the distance is greater, it is marked as "Available."

The script was initially tested by placing a hand over the sensor and watching the printed distance values update in real time. This helped confirm that the sensor was working as expected. The script was also designed to loop continuously, checking for updates every few seconds.

4.5 Firebase Integration and Cloud Connectivity

Once local sensor readings were verified, the script was expanded to include cloud connectivity using Firebase. A Firebase project was created, and the Realtime Database service was enabled. A service account key was generated and securely transferred to the Raspberry Pi.

Using the Firebase Admin SDK for Python, the script was modified to push data to the cloud each time a distance reading was taken. The database structure was kept simple at this stage, with a single node representing one parking space:

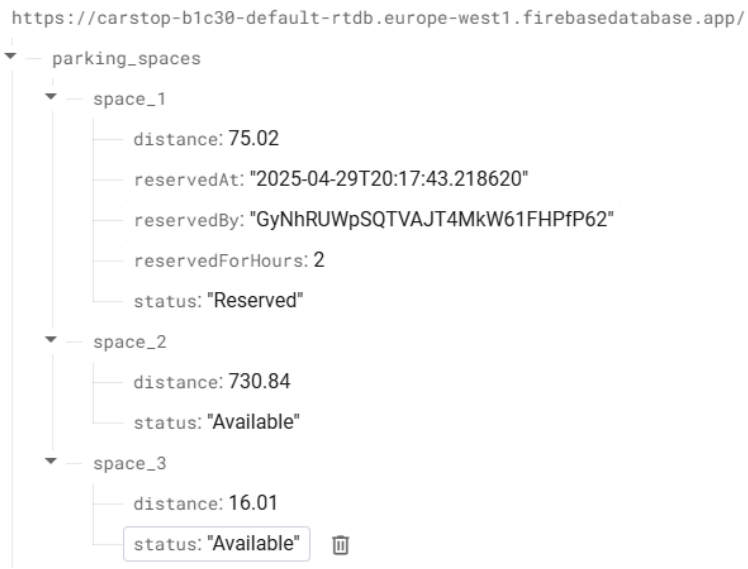


Figure 8: Database Structure

4.6 Testing Hardware & Output Verification

The implementation was verified through multiple stages of manual testing. The ultrasonic sensor was initially tested in isolation, with distance readings printed to the terminal. Once cloud connectivity was added, further testing was done to ensure that sensor readings were being correctly uploaded to Firebase.

Common issues encountered included SSH failures due to miswiring of the logic level converter, power instability from improper voltage rail sharing, and minor Python errors during initial Firebase integration. Each of these issues was resolved through troubleshooting and small hardware or code adjustments.

The system was ultimately able to consistently detect the presence of an object and update Firebase accordingly, simulating how a real-world parking bay system might behave in a deployed environment.

4.7 Firebase Communication

Firebase serves as the cloud backend for both the app and the Raspberry Pi. Firebase Authentication manages user sessions, with a condition to check whether the signed-in email belongs to the admin. Realtime Database is used to hold live parking space data including status, distance, reservedAt, and reservedBy.

The Firebase structure also includes user balance data and a reservation history log. When a user reserves a space, Firebase is updated with the user's ID, reserved time, and number of hours. When the time runs out or the user cancels, these values are cleared, and a log entry is added to their reservation history node. Firebase is also used to detect when a user runs out of funds, and to track payments and top-ups for their virtual balance.

4.8 Android App Functionality

The Android app is split into user and admin flows. When a user logs in, their email is checked — if it matches the designated admin email, the app navigates to the admin dashboard; otherwise, it proceeds to the user dashboard. The user app includes the Home screen, Find Parking screen, My Reservations screen, and Payment History screen. In contrast, the admin interface includes access to charts, analytics, and tools to remove user reservations.

Real-time Firebase listeners are used throughout the app to reflect changes without refreshing. When the Raspberry Pi updates a parking bay status, the colour of the corresponding bay in the app updates instantly. When a user reserves a space, the app deducts balance and logs the action to Firebase. Cancelled or expired reservations are displayed in the history screen. The admin app includes charts to show the ratio of occupied, reserved, and available bays, with pie and bar charts generated dynamically.

4.9 Code

The Android application is divided into standard user and admin experiences. Key components include:

HomeScreen (User): Shows balance, buttons for Find Parking, My Reservations, and Payment History.

FindParkingScreen: Displays 3 spaces in real time. Allows reservations, cancels, and payment confirmation.

MyReservationsScreen: Tracks reservation status and includes countdown and extension logic.

PaymentHistoryScreen: Shows balance, top-up options, and transaction history.

AdminHomeScreen: Includes dashboard, real-time charts, and system-wide data.

ManageParkingScreen: Lets admin view live sensor data, time remaining, and manually remove reservations.

The following code snippets are some of the more important that help these components to function properly:

```python
# --- Main Loop ---
try:
    print("Starting 3-sensor parking monitor... Press Ctrl+C to stop.")
    while True:
        for space in spaces:
            print(f"Checking {space['id']}...")
            dist = measure_distance(space["trig"], space["echo"])
            check_and_update_status(space["id"], dist)
        time.sleep(2)

except KeyboardInterrupt:
    print("Stopping monitor. Cleaning up GPIO.")
    GPIO.cleanup()
```

Figure 9: Python Sensor Loop

What it does: This code continuously triggers each HC-SR04 ultrasonic sensor to emit a sound pulse and calculates the time it takes for the echo to return. Using this time, the script calculates the distance between the sensor and any object above it (e.g., a car). If the distance is less than a set threshold (e.g., 10 cm), the space is considered “Occupied”.

Why it’s important: This is the core functionality that determines whether a parking space is in use.

How it fits: The result of this calculation is immediately pushed to Firebase to update the app’s live parking space statuses.

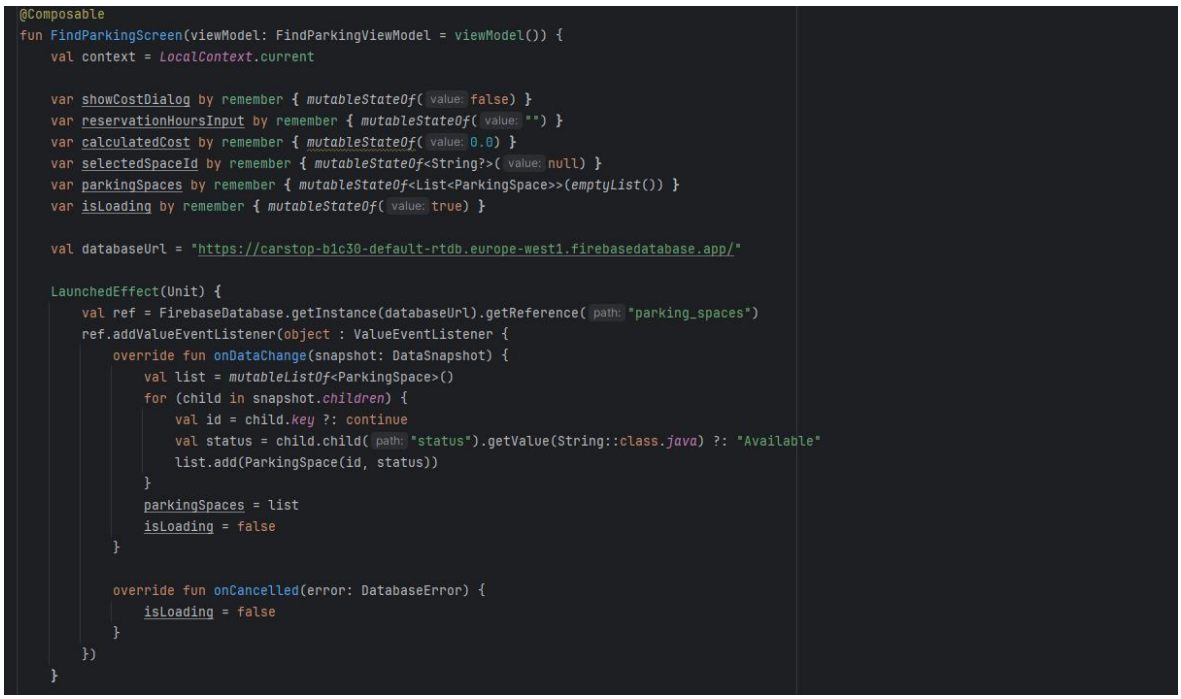
```
val uid = FirebaseAuth.getInstance().currentUser?.uid ?: return
val databaseUrl = "https://carstop-b1c30-default-rtdb.europe-west1.firebasedatabase.app/"
val userRef = FirebaseDatabase.getInstance(databaseUrl).getReference( path: "users/$uid")
val spaceRef = FirebaseDatabase.getInstance(databaseUrl).getReference( path: "parking_spaces/space_$spaceId")
val transactionsRef = FirebaseDatabase.getInstance(databaseUrl).getReference( path: "transactions/$uid")

userRef.child( pathString: "balance").get().addOnSuccessListener { snapshot ->
    val balance = snapshot.getValue(Double::class.java) ?: 0.0
    if (balance >= totalCost) {
        // Deduct balance
        userRef.child( pathString: "balance").setValue(balance - totalCost)

        // Reserve the space
```

Figure 10: Firebase Listener in ViewModel

What it does: This code listens to changes in the Firebase Realtime Database under the parking_spaces node. Whenever a space's status or distance changes, the app automatically updates the UI without needing a refresh.

Why it's important: It enables real-time interactivity for the user, ensuring that the displayed parking availability is always up to date.

How it fits: This is the client-side mechanism that reflects sensor status live in the mobile app.



```
@Composable
fun FindParkingScreen(viewModel: FindParkingViewModel = viewModel()) {
    val context = LocalContext.current

    var showCostDialog by remember { mutableStateOf( value: false) }
    var reservationHoursInput by remember { mutableStateOf( value: "") }
    var calculatedCost by remember { mutableStateOf( value: 0.0) }
    var selectedSpaceId by remember { mutableStateOf<String?>( value: null) }
    var parkingSpaces by remember { mutableStateOf<List<ParkingSpace>>(emptyList()) }
    var isLoading by remember { mutableStateOf( value: true) }

    val databaseUrl = "https://carstop-b1c30-default-rtdb.europe-west1.firebasedatabase.app/"

    LaunchedEffect(Unit) {
        val ref = FirebaseDatabase.getInstance(databaseUrl).getReference( path: "parking_spaces")
        ref.addValueEventListener(object : ValueEventListener {
            override fun onDataChange(snapshot: DataSnapshot) {
                val list = mutableListOf<ParkingSpace>()
                for (child in snapshot.children) {
                    val id = child.key ?: continue
                    val status = child.child( path: "status").getValue(String::class.java) ?: "Available"
                    list.add(ParkingSpace(id, status))
                }
                parkingSpaces = list
                isLoading = false
            }

            override fun onCancelled(error: DatabaseError) {
                isLoading = false
            }
        })
    }
```

Figure 11: Reservation Logic

What it does: This block handles the reservation process when a user taps a parking space. It presents a dialog where they enter the number of hours (up to 5). If they have sufficient funds, the system calculates the cost (€1.50/hour), deducts it from their balance, and reserves the space.

Why it's important: This manages the core interaction between users and the parking system.

How it fits: It's tied into Firebase Authentication and Database, updating both the user's balance and the parking space's status.

```
@Composable
fun PaymentHistoryScreen() {
    val auth = FirebaseAuth.getInstance()
    val uid = auth.currentUser?.uid
    val databaseUrl = "https://carstop-b1c30-default-rtdb.europe-west1.firebasedatabase.app/"
    val userRef = FirebaseDatabase.getInstance(databaseUrl).getReference( path: "users/$uid")
    val transactionsRef = FirebaseDatabase.getInstance(databaseUrl).getReference( path: "transactions/$uid")

    val context = LocalContext.current

    var balance by remember { mutableStateOf( value: 0.0) }
    var transactions by remember { mutableStateOf(listOf<Transaction>()) }
    var showTopUpDialog by remember { mutableStateOf( value: false) }
    var customAmount by remember { mutableStateOf( value: "") }
    var topUpAmount by remember { mutableStateOf( value: 0.0) }

    LaunchedEffect(Unit) {
        userRef.child( pathString: "balance").addValueEventListener(object : ValueEventListener {
            override fun onDataChange(snapshot: DataSnapshot) {
                balance = snapshot.getValue(Double::class.java) ?: 0.0
            }

            override fun onCancelled(error: DatabaseError) {}
        })
```

Figure 12: Balance Top-Up System

What it does: This code enables users to add credit to their balance. They can choose a fixed amount (e.g., €5, €10) or enter a custom value. Upon confirmation, their balance in Firebase is updated and a transaction record is created.

Why it's important: It simulates a working payment system, which is critical for managing parking transactions.

How it fits: It supports the reservation feature by ensuring users can afford bookings and maintain a positive balance.

4.10 Summary

The implementation of CarStop successfully combines sensor-driven physical interaction with cloud-based services and an intuitive mobile app. Real-time updates, Firebase authentication, simulated balance deduction, and role-based navigation all contribute to a responsive and realistic user experience. With the project now complete, the system demonstrates a working prototype of a smart parking solution that could be scaled and deployed in real-world environments.

Chapter 5 Testing

5.1 Introduction

The testing phase of this project has been one of the most important and rigorous aspects of its development. After months of iterative building and debugging, the system needed to be stress-tested in a way that mimicked real-world scenarios as much as possible. Since the CarStop application integrates multiple technologies including Android development, Firebase services, and simulated IoT hardware, each component had to be tested not only in isolation but also in conjunction with other modules to ensure a smooth and unified experience.

Testing allowed me to verify that the features I worked so hard to implement — like user authentication, parking space reservations, real-time updates, and administrative tools — actually worked as intended. It also helped me uncover flaws and edge cases I wouldn't have anticipated during development, especially when it came to real-time data synchronisation and account balance calculations.

Beyond verifying that everything functioned correctly, testing was also about validating the experience from a user's perspective. I wanted the app to not only work, but to feel right — it had to be fast, responsive, and intuitive. The lessons learned during testing also led to several design and logic changes that significantly improved the app.

5.2 Testing Environment and Setup

To begin testing the CarStop system, I set up a reliable environment that allowed me to simulate various conditions and user behaviours. For the Android application, I used both the built-in Android Emulator in Android Studio and my personal Android device for real-world testing. This helped to ensure that the application was functional across different screen sizes and hardware capabilities.

The Firebase backend was connected using real-time listeners for both the database and authentication modules. This allowed me to continuously monitor data as changes occurred during reservations, payments, and user login events. Firebase’s integration with Android also made it relatively straightforward to capture logs, errors, and data snapshots.

For the hardware side of the simulation — which mimics IoT sensors detecting whether a parking bay is occupied or not — I used Python scripts running on a Raspberry Pi 4. The sensor would detect object proximity and push updates to Firebase depending on whether a space was available, reserved, or occupied. To fully test the integration of this sensor data into the mobile app, I allowed these values to fluctuate and observed how the app reacted in real time.

Each Firebase update was validated by checking both the frontend display (to ensure the UI changed) and the backend database structure. This approach allowed for full end-to-end validation.

5.3 Functional Requirements

Each of the core features was tested against its functional requirements to ensure they behaved as intended. The login and registration features had to correctly differentiate between regular users and the admin account. I verified that if I signed in as the admin, I would land on the Admin Home Screen, and if I signed in as a regular user, I would instead be directed to the standard Home Screen. This behaviour was tested under various conditions, including re-opening the app after signing out.

Another major functional requirement was parking space reservation. When a user selected a parking space, they had to be able to choose the number of hours (between 1 and 5), and the cost had to be calculated correctly and deducted from their balance. This cost calculation was tested thoroughly to ensure that the user couldn’t reserve if they had insufficient funds.

To further test the accuracy of parking reservations, I implemented real-time tracking of countdown timers for active bookings. I observed whether the reservation time updated correctly on the My Reservations screen and whether the slot status changed appropriately once the time expired. Adding the functionality to extend a reservation added another layer of complexity, especially with the rule that no reservation could exceed five hours in total. This was tested by booking a space for two hours, extending by one hour, and checking that it blocked further extensions beyond the five-hour mark.

The admin features were tested to ensure that they had elevated privileges — for example, the ability to remove a user's reservation. When the admin removed a reservation, I verified that the user's "My Reservations" screen reflected the update by displaying the message "Removed by admin." This required careful backend logic to ensure it didn't look like a standard cancellation initiated by the user.

The payment functionality was another critical area. I tested top-up features to ensure that balances updated in real-time when a user selected €5, €10, or a custom amount. These updates also had to reflect in the payment history with accurate timestamps. Transaction records had to sort from most recent to oldest, and each entry needed to include the space ID, amount, and purpose.

5.4 Non-Functional Requirements

In terms of performance, the application had to be responsive and provide instant feedback to the user. To test this, I deliberately made rapid changes in the Firebase backend — such as switching a space from available to occupied — and timed how long it took for the UI to reflect those changes. I also tested how the app responded when switching between screens, adding loading indicators where appropriate.

Scalability, while harder to fully simulate, was tested by adding dummy data representing many parking spaces and verifying that the app could still handle rendering them without slowing down. I also simulated multiple users updating data at the same time to test Firebase's concurrency handling.

The UI design was also part of non-functional testing. I ensured consistency in font sizes, colours, and layouts across all screens. For example, all buttons needed to use the same padding and colour scheme to maintain brand consistency. Screens like "My Reservations" and "Manage Parking" had to display a lot of data while still looking tidy and easy to read. I refined padding and layout multiple times during testing.

5.5 Error Handling and Troubleshooting

Several issues came up during testing that required extensive troubleshooting. One of the earliest problems I encountered was that reservation countdowns would continue even after the user cancelled their reservation, leading to UI confusion. I resolved this by improving the backend status update to cancel all countdowns once a reservation was removed or expired.

Another issue was that real-time Firebase listeners occasionally failed silently. To address this, I added manual data refresh triggers and added more robust logging on the frontend. Toast messages were also added across the app to provide user feedback when something failed — like a failed payment due to insufficient balance or an error when contacting the database.

One particularly tricky issue arose when Firebase attempted to convert strings into timestamps and vice versa. This caused the app to crash if a transaction was incorrectly logged. I fixed this by writing more defensive Kotlin code that gracefully handled different data types and fallback conditions.

At times, the Android emulator behaved differently from physical devices, especially with keyboard overlays and timing-based animations. For example, in the emulator, dialogs appeared perfectly centred, but on a real device, they were slightly misaligned. I had to fine-tune layout parameters and test across both environments to get it right.

I also introduced unit testing for specific components like input validation, and I carried out manual testing of every user flow, both as admin and standard user. This helped ensure nothing was missed, especially as the number of features grew toward the end of development.

Chapter 6 Results

This chapter presents the results of the CarStop project following its design, development, and testing stages. The aim is to evaluate the effectiveness of the final system in meeting the objectives outlined at the beginning of the project. While the system was developed to be as comprehensive and feature rich as possible, some features originally planned were not fully implemented due to technical constraints or time limitations. However, the core functionality of the app — live parking space monitoring, reservations, user payments, admin controls, and Raspberry Pi sensor integration — was successfully built and tested. This section also acknowledges the parts of the system that were either simplified or left incomplete and provides context for those decisions.

6.1 Achievement of Functional Goals

The primary functional objective of the system was to allow users to view the real-time status of parking spaces and make reservations through a mobile application. This core goal was fully achieved. The app reliably displayed three parking bays connected to HC-SR04 sensors, with live status updates fed directly from the Raspberry Pi via Firebase. Each change in a parking bay's state — from Available to Occupied or Reserved — was reflected almost instantly in the app. This demonstrated successful real-time communication between hardware, cloud infrastructure, and the Android frontend.

User login and registration were also successfully integrated using Firebase Authentication. This system was used not just to authenticate users but also to differentiate between regular users and the admin, who had access to additional features such as removing reservations and viewing system statistics.

Reservations worked as intended. Users could click on an available space, choose a duration (up to five hours), and the system would calculate the cost (€1.50 per hour). The balance was automatically deducted, and the reservation details were saved both in Firebase and locally displayed in the “My Reservations” section. The countdown timer for each active booking was implemented and updated in real time. Users could also extend their reservations — provided they didn’t exceed the five-hour cap — and were charged appropriately for the extension.

A major success was the admin interface. Admin users could view a dashboard displaying total space usage, occupancy status, and live graphs showing space data in visual form. Admins could also remove active reservations, with the system updating all users and displaying the message “Removed by admin” in the affected user’s reservation history.

However, not all functional goals were fully realised. A significant gap was the planned use of a camera for licence plate recognition. Due to issues sourcing the camera module in time and challenges with software integration and image processing, this feature had to be left out. It was a technically ambitious task that required training image recognition models or integrating open-source libraries, which proved too time-intensive given the project timeline. In the end, omitting it allowed focus to remain on perfecting the features that were already in development.

Additionally, the payment system was simplified. Originally, there were plans to allow users to add funds using a debit or credit card through an integrated payment API. Due to the complexity of integrating a payment gateway and the sensitivity of handling actual financial transactions, this was replaced with a simulation approach. Users could enter an amount and click a “Top Up” button to simulate adding funds. While it does not reflect real-world banking processes, it allowed the balance and reservation-cost logic to function as expected and served the project's educational purpose.

6.2 Hardware Integration Results

On the hardware side, the project achieved solid success in integrating real-world components with the digital system. The Raspberry Pi 4 was used to host Python scripts that communicated with three HC-SR04 ultrasonic sensors. Each sensor was wired via a logic level converter to safely handle the voltage difference between the sensors (5V) and the Pi's GPIO pins (3.3V). Readings were taken multiple times per second, and the distances were used to determine if a vehicle was present above each bay.

The hardware script was able to push the distance and status to Firebase in real time, which was a major milestone. All three sensors operated reliably and consistently within the constraints of the setup. The circuit design, while compact, was stable enough to run for long periods without crashing or producing inaccurate results.

Initial problems included wiring issues and level shifting errors that caused the Raspberry Pi to freeze on boot. These were resolved by carefully separating power sources and ensuring the echo pins did not feed 5V directly into the Pi. A virtual environment was used on the Pi to manage Firebase dependencies, and SSH was used for remote debugging.

Although the camera module for licence plate detection was planned as a fourth hardware input, it was ultimately left unimplemented for reasons already outlined. Nonetheless, the system's success in connecting the sensors to a remote database and seeing those updates reflected live in the app marks a significant accomplishment in terms of IoT integration.

6.3 Comparison to Initial Objectives

When comparing the final outcome of the project to the objectives set out in the beginning, it becomes clear that most goals were achieved, although with some compromises along the way. The system was envisioned to be a mobile-first, user-friendly app that allowed for the reservation and monitoring of parking spaces — and that goal was clearly fulfilled. Real-time updates, Firebase integration, and multi-role login functionality all worked seamlessly.

The project aimed to demonstrate a full end-to-end flow, from sensor detection to user booking to admin control. This flow was not only implemented but extended with extras such as a payment simulation system, a live countdown on active reservations, and visual dashboards for admins.

The camera-based licence plate recognition, which would have brought an additional layer of automation and user validation, was the most notable feature that could not be completed. This feature remains a future enhancement and would likely require dedicated image processing capabilities beyond the scope of this project.

Similarly, the plan to simulate actual payment processing through Stripe or PayPal had to be reduced to a basic credit entry system. This decision was made to reduce scope and focus on more technically achievable features that didn't require complex legal or security considerations.

Despite these omissions, the work completed still constitutes a very substantial and functional system that could easily serve as a prototype for a more advanced commercial parking solution.

6.4 Screenshots

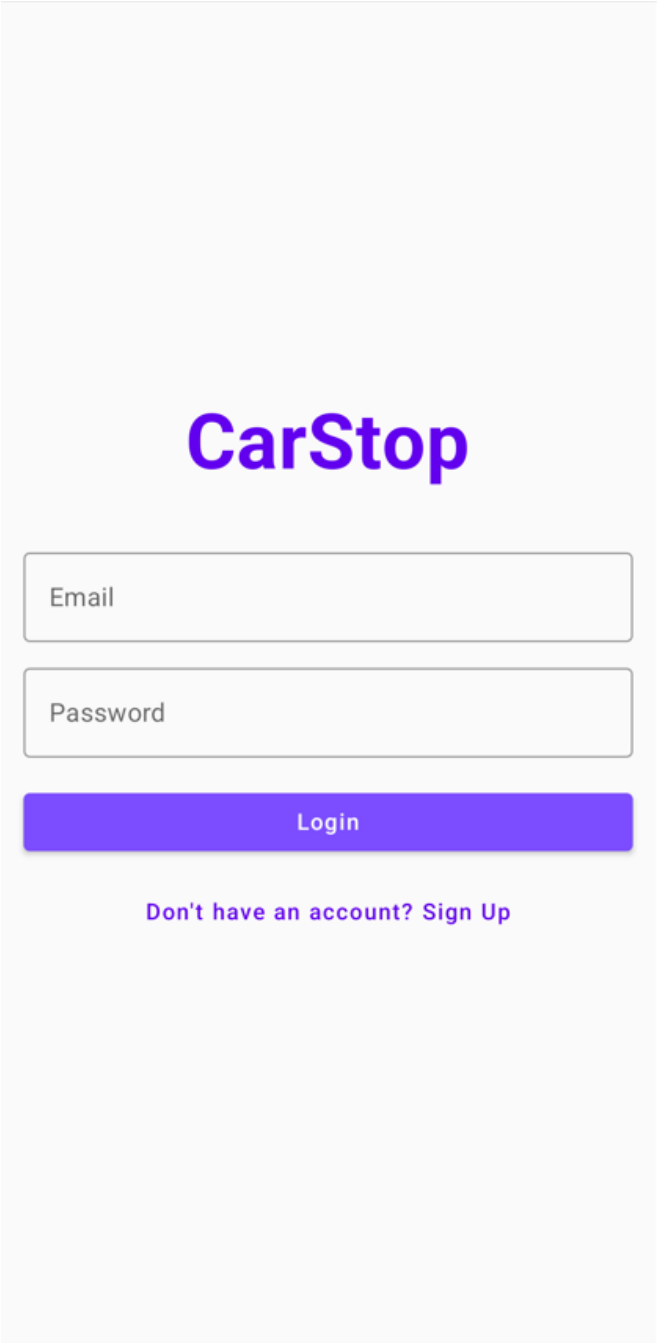


Figure 13: Login Screen

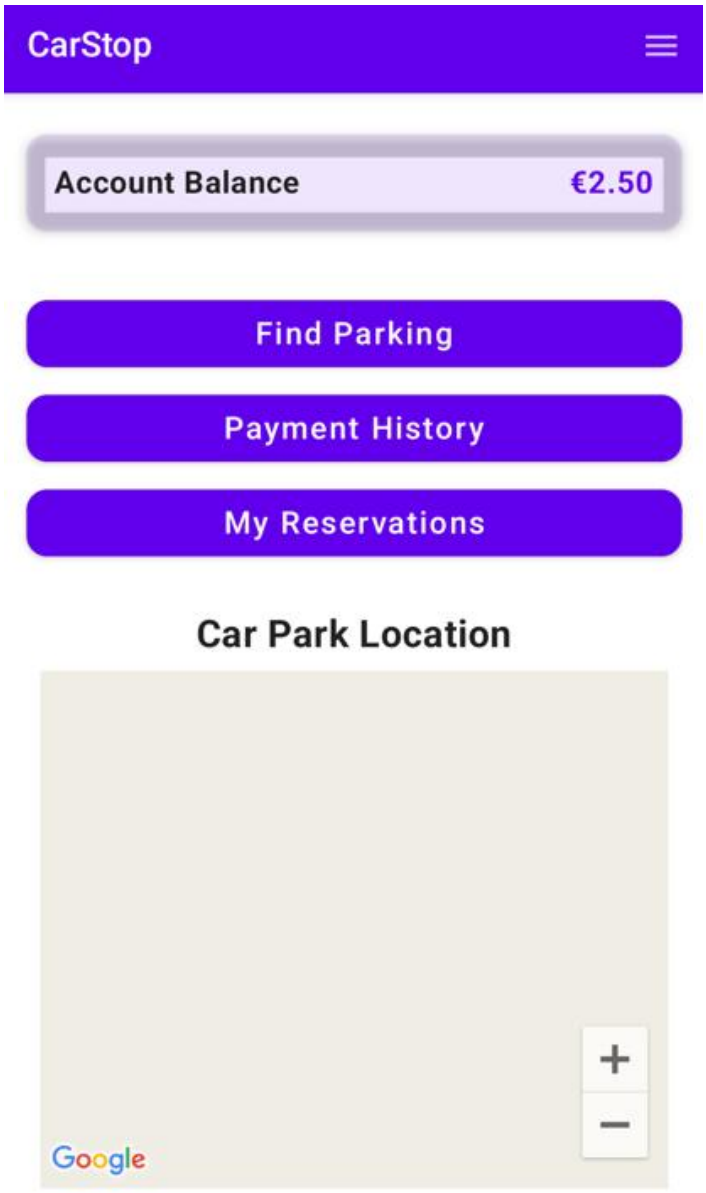


Figure 14: Home Screen

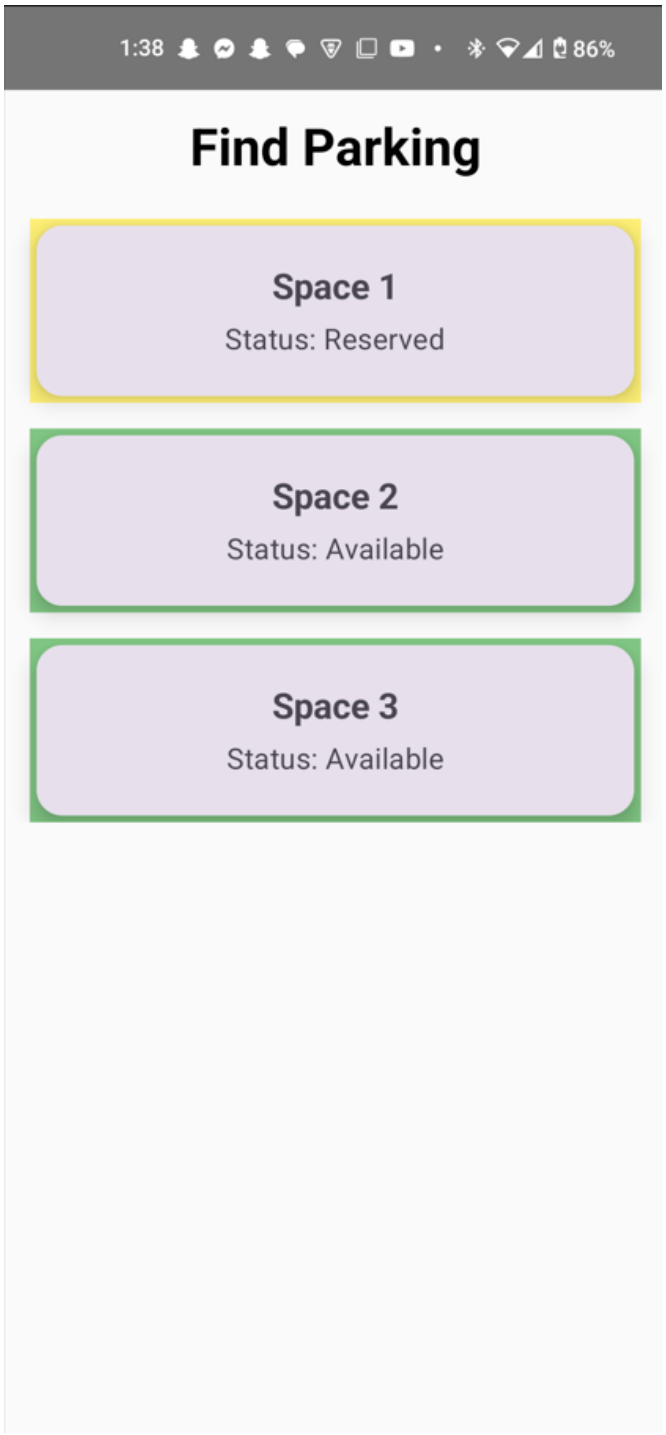


Figure 15: Find Parking Screen

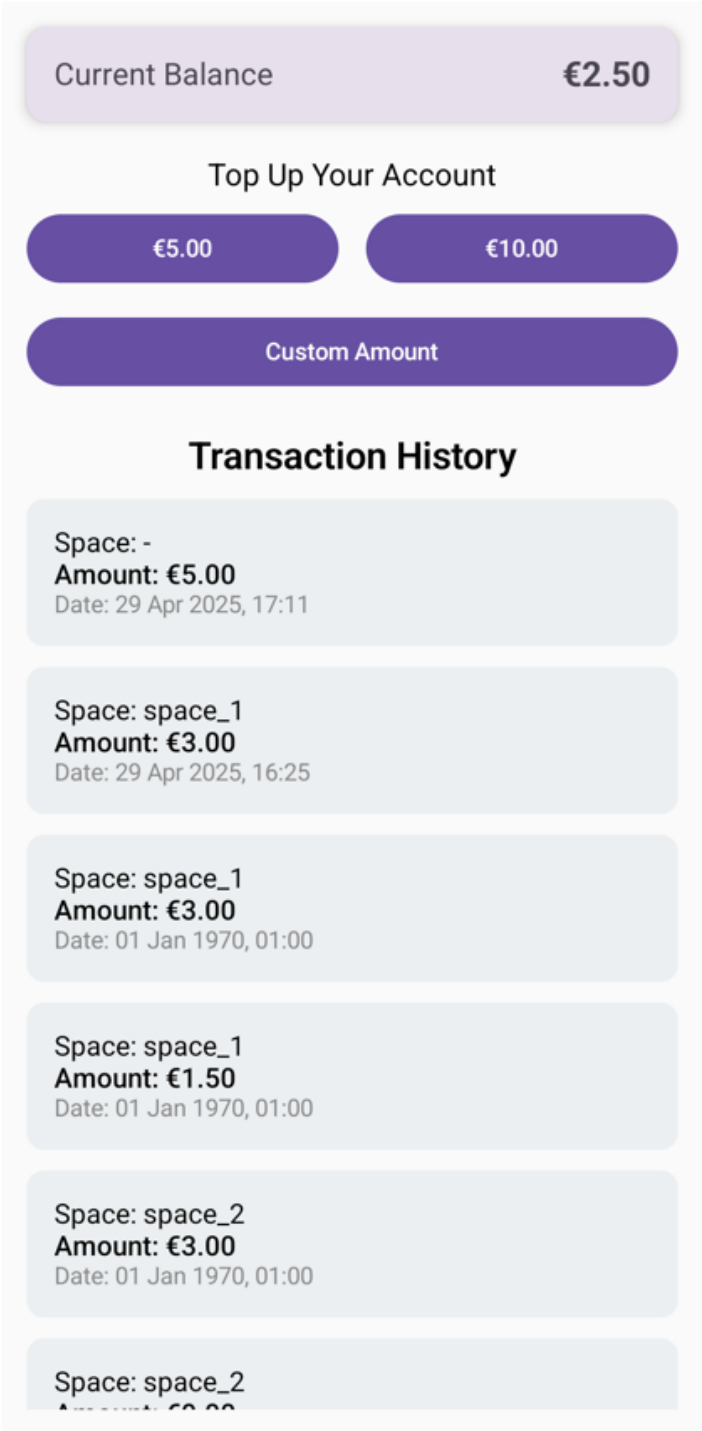


Figure 16: Payment History Screen

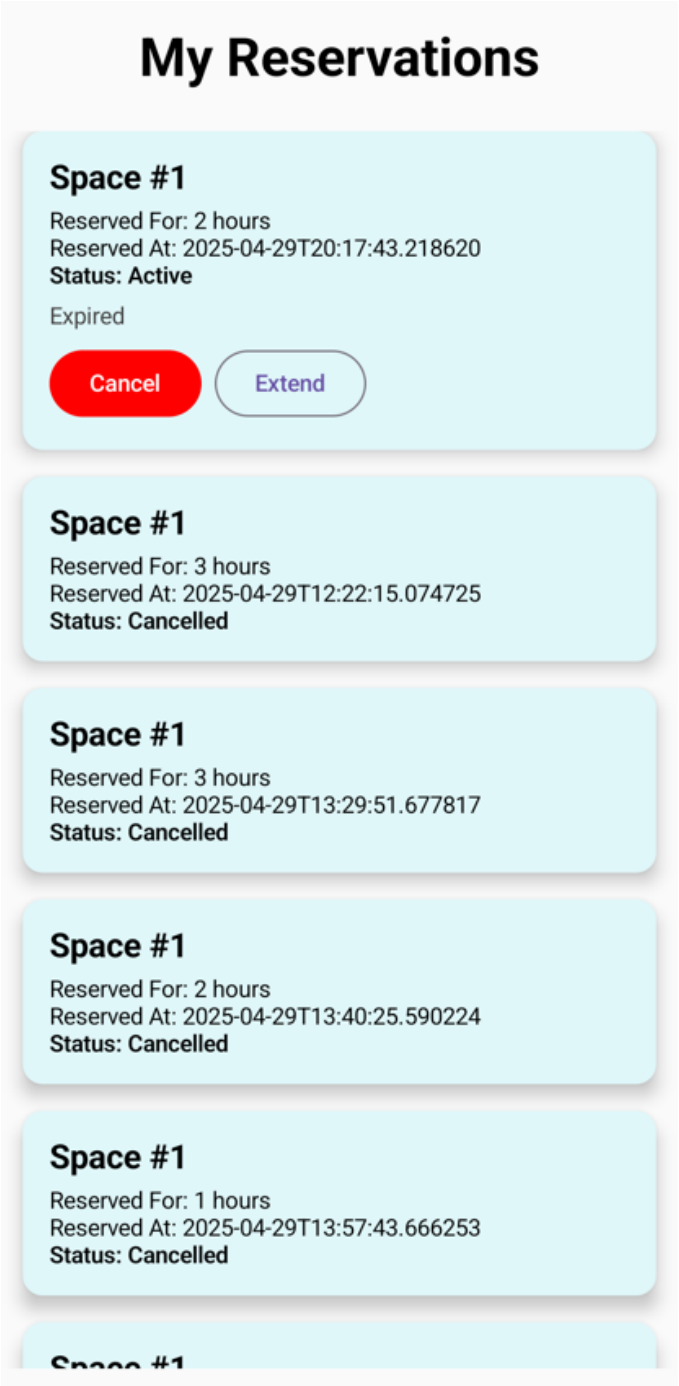


Figure 17: Reservations Screen

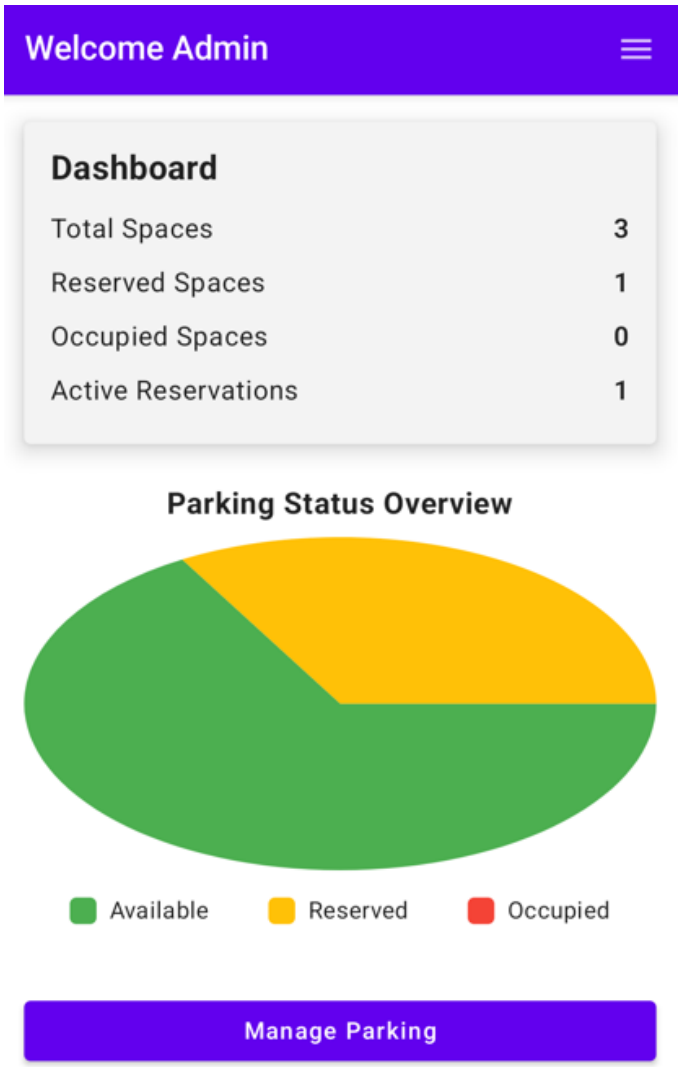


Figure 18: (Admin) Home Screen

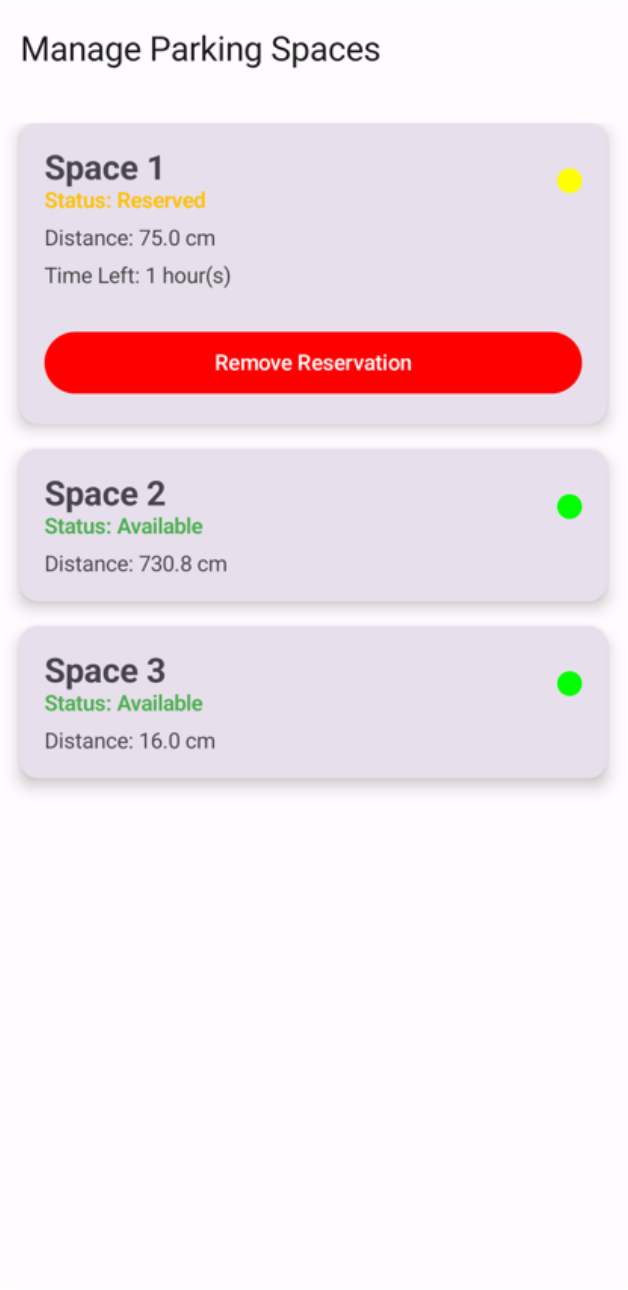


Figure 19: Manage Parking Screen

6.5 Summary

The CarStop project, as it stands, represents a significant accomplishment both technically and academically. It integrates hardware, backend infrastructure, and Android development into a single, cohesive system. Users can log in, top up their balance, reserve a space, monitor their bookings, and manage their payment history. Meanwhile, the admin interface offers advanced control, real-time oversight, and data visualisation tools.

Not every feature imagined at the start was implemented, but the project retained its core goals and adapted sensibly in the face of obstacles. Decisions like dropping camera recognition and simplifying payments allowed more time to refine the features that mattered most.

The system is well-structured, scalable, and offers a strong foundation for further development. It also reflects a practical understanding of how to manage complexity in real-world development — knowing when to push forward and when to simplify for the sake of project success.

Chapter 7 Conclusion

The CarStop project set out to build a mobile app that helps people find and reserve parking spaces using real-time data from sensors. It also aimed to give admin users a way to manage those parking spaces through an extra dashboard. While I wasn't able to include absolutely everything I had planned at the beginning, the final result still shows a working system that brings together hardware, cloud storage, and Android development in a meaningful way.

Users can sign into the app, see which spaces are available, and reserve a space for up to five hours. The app deducts credit from their balance, tracks the time left, and even allows users to extend their booking if needed. I also included a "My Reservations" screen, a top-up system, and a full admin dashboard that lets the admin monitor parking activity and remove bookings when needed. One of the biggest successes was getting real data from the sensors on the Raspberry Pi to show up live in the app using Firebase.

However, not all the features I wanted to include made it into the final project. I had hoped to use a camera for licence plate recognition, but due to time and hardware issues, I couldn't get it working. I also originally planned to integrate a real payment system, but instead I settled on a simpler version where users just enter how much money they want to top up. It isn't realistic, but it was enough to demonstrate the logic behind the system.

Overall, the system does what it set out to do — it shows available spaces in real time, handles bookings and payments (even if simplified), and gives the admin a clear overview of how things are going. It's a strong foundation for what could be developed further in the future.

7.1 Testing Environment and Setup

Working on this project was definitely challenging, but also rewarding. It was the biggest thing I've built during my degree, and it pulled together a lot of different skills, from coding and UI design to working with hardware and cloud databases. There were plenty of moments where I felt stuck, especially when trying to connect the sensors to the Firebase system or when dealing with the Android UI issues, but getting things to finally work was a great feeling.

There are a few things I'd definitely change if I were to start over. First of all, I should have done more research at the start. I jumped in with a lot of big ideas, like licence plate scanning and full payment systems, without fully understanding how much work or setup they would take. If I had been a bit more realistic with my expectations, I could have focused earlier on the most important features and had more time to polish them.

Time was also a big challenge. With so many other college assignments and group projects, it was hard to give this my full attention every single week. I'll be honest, I probably could have put more consistent effort in early on. There were some weeks where I made loads of progress and others where I barely touched it. If I had managed my time better, I might have been able to finish some of the extra features I had to leave out.

That said, I learned a lot from this project. It pushed me to figure things out on my own, to debug problems I'd never faced before, and to keep going even when things weren't working. I'm proud of what I built, it's a fully working system that shows live sensor data, handles bookings and payments, and gives both users and admins a smooth experience. It's not perfect, but it's real, and it works, and that's a good way to finish the year.

References

Bamboo Apps, 2024. [Online]
Available at: https://bambooapps.eu/blog/parking-app-development
Carroll, S., 2016. *Do Parking Apps Really Alleviate Parking Problems?.* [Online]
Available at: https://medium.com/move-forward-blog/do-parking-apps-really-alleviate-parking-problems-993006a88af4
Chang, S.-L., 2004. *Automatic license plate recognition.* [Online]
Available at: https://ieeexplore.ieee.org/abstract/document/1271288
Crowe, M., 2024. *How to write your Thesis By Mark Crowe.* [Online]
Available at: https://github.com/marcocrowe/thesis-templates
[Accessed 1 10 2024].
Nizio, P., 2024. *Parking revolution: how mobile apps are transforming urban mobility.* [Online]
Available at: https://parkcash.io/parking-revolution-how-mobile-apps-are-transforming-urban-mobility-113-6573
O'Donovan, A., 2020. *The future of parking reservation apps.* [Online]
Available at: https://wayleadr.com/blog/the-future-of-parking-reservation-apps/
Parker, W., 2024. *Artificial Intelligence in Car Park Management: How AI-Powered Systems are Transforming Operations in the UK.* [Online]
Available at: https://medium.com/@williamparker24091987/artificial-intelligence-in-car-park-management-how-ai-powered-systems-are-transforming-operations-25b477497bb6#:~:text=AI%2Dpowered%20car%20park%20management,pay%20seamlessly%20through%20digital%20platforms.
Stax Payments, 2022. *Understanding the nature of Payment Gateway For Your Mobile App.* [Online]
Available at: https://staxpayments.com/blog/payment-gateway-for-your-mobile-app-how-to/#Why_Would_Companies_or_Developers_Want_a_Mobile_App_Payment_Gateway
Xu, B., 2013. *Real-Time Street Parking Availability Estimation.* [Online]
Available at: https://ieeexplore.ieee.org/abstract/document/6569118
Zheng, Y., 2015. *Parking availability prediction for sensor-enabled car parks in smart cities.* [Online]
Available at: https://ieeexplore.ieee.org/abstract/document/7106902

Appendix A System Requirements

Hardware:

- Raspberry Pi 4 (4GB)
- HC-SR04 Ultrasonic Sensors × 3
- Logic Level Converter
- Jumper Wires and Breadboard
- MicroSD Card (32GB) with Raspberry Pi OS
- Power Supply (5V 3A)
- Android Smartphone (for testing)

Software:

- Android Studio Flamingo (or later)
- Firebase Realtime Database
- Firebase Authentication
- Firebase Console
- Python 3 (with firebase-admin, RPi.GPIO libraries)
- Jetpack Compose for UI Development

Appendix B Testing Account Credentials

Test Admin Account:

- Email: justinerenz98@gmail.com
- Password: password123

Test User Account:

Email: justinerenz17@gmail.com

Password: password123