Driver's Logbook: A Cross-Platform React Native App

Practical Course: Computer Science 2

Hynek Zemanec*

Faculty of Computer Science, University of Vienna, Währinger Str. 29, 1090 Vienna, Austria

2022S

Contents

Abstract

An app called The Driver's Logbook was developed using the JavaScript framework React Native to evaluate the feasibility of cross-platform development. The app supports both Android and iOS and aims to assist users in tracking and managing their car logbook information, such as distance driven, trip times, and locations. The app incorporates Azure Computer Vision for optical character recognition and Azure blob storage for storing odometer images. The app's code is structured using a layered architecture approach. The app is designed to help user easily track and manage their car logbook information in an efficient way.

1 Introduction

To expand their reach and appeal to more users, many leading companies in the smartphone app industry are now targeting both of the the two most widely used mobile operating systems, Android and iOS. To create native apps for iOS, developers use the programming language Swift or Objective C. Additionally, to build and compile iOS apps, Apple's IDE Xcode is needed. However, it's important to note that Xcode is only available for macOS.

Figure 1: Demo App QR code for use with Expo Go

On the other hand, native apps for Android are primarily developed using the programming language Java, and increasingly, the more modern Java-compatible language Kotlin. Development and compilation is carried out in Google's IDE Android Studio. It is worth noting that both Android's operating system and required development environment are generally considered to be less restrictive than iOS.

The differences mentioned suggest that supporting both platforms can be costly, as at minimum, two specialized teams, one for each technology, are required. Additionally, there may be challenges in synchronizing the look and feel and features of the app across both platforms. A solution to the problem of 2 code bases is offered by the open-source JavaScript framework React Native. It allows developers to create mobile apps for both iOS and Android using a single codebase. This

*e-mail:hynekz20@univie.ac.at

is achieved by leveraging JavaScript UI library React, and applying it to mobile app development. React Native uses a bridge to communicate with the native components of the target operating system. This enables developers to access the native features of the device, such as the camera and GPS, while still writing code in JavaScript. The framework is maintained by a large open-source community with Meta Platforms (formerly Facebook) as one of its largest contributors.

2 Project Definition

The project aims to develop a native application for operating systems iOS and Android. The main objective is to create a digital drivers logbook, which simplifies the recording of a driver's trips. The purpose of the application is to investigate the capabilities and constraints of cross-platform development using the open-source UI framework React Native and the JavaScript ecosystem. The department's initial explanation of the project states:

React Native advertises with the slogan "Learn once, write anywhere." Is it really that simple? Prove it by developing a React Native application that implements a driver's logbook (Fahrtenbuch). Entries may be added by a) manual entry of kilometer reading or b) image recognition by taking a photo, or c) fully automatic mode (experimental). Add meaningful extended functionality (e.g., start and destination of each entry, costs, driver, ...). This topic aims to achieve very high usability of the prototype. The mostly dull task of filling out a driver's logbook should be supported using state-of-the-art technology to allow a driver to fill out the logbook with minimal effort, letting the program do all possible work for the driver in the logging process.

2.1 Technologies

The app was built using a variety of technologies to ensure compatibility, optimal performance and pleasant user experience. Leveraging robust, statically typed system, TypeScript, a superset of JavaScript, was used to provide robust and maintainable codebase. Expo SDK, a set of tools and services built around React Native, was utilized to speed up the development process and handle native modules.

The app's user interface was built using React Native Paper (v4) and Styled Components. At the time of writing, version 4 of React Native Paper is stable implementation of second generation of Material Design, a system for design consistency. Styled Components is a library for styling of React components that enables advanced dynamic style declaration that can be combined with styling rules used on the web. React Navigation, a popular library for routing in React Native that integrates well with React Paper was also used to provide a seamless navigation experience.

The Yarn package manager was used to manage dependencies and improve development workflow. Moreover, following a decision to prefer local storage over remote alternatives, Async Storage, a local storage management technology was used in the development of the app to persist and retrieve collected data. Finally, in order to ensure the quality of the app, Jest, a widely used JavaScript testing framework, was utilized for unit testing.

3 Project Management

During development the development strategy was to follow agile software philosophy. This means putting emphasis on short development cycles characterized by rapid prototyping adjusted by received feedback. To better facilitate planning, the project was divided into three major milestones: CONCEPT, IMPL and REPORT as as depicted in the Gantt chart in the figure 2 and described in the table 1. All milestones were further subdivided into smaller milestones to better track the progress of the project. All milestones were tracked using the GitLab platform, which allowed for easy monitoring of the team's progress and identified any issues that needed to be addressed. All milestones were fulfilled on time and the project was completed successfully within the given timeframe. The use of milestones and GitLab allowed for clear and effective project management which ultimately led to the successful completion of the project.

3.1 Version Control

The open-source GIT platform GitLab operated by the University of Vienna, has been used to version-control and remotely archive the source code. It is available online at `https://git01lab.cs.univie.ac.at/p2/2022ws/12010957-hynek-zemanec`. Each directory contains its own `README.md` file with descriptions and directions.

Initially, I attempted to use a branching strategy that included a `main` branch for production-ready code, a `dev` branch for actively developed code, and a `feature` branch for individual features as depicted on the figure 5.

However, I found that the feature branch turned out to be unnecessary overhead in my workflow. Since I was the only developer working on the project I found it more efficient to simply use the `dev` branch and merge changes directly into `main` when they were ready. This streamlined approach allowed me to focus on development without the added complexity of managing multiple branches.

4 Features

The usability and functionality of the app were guided and improved through the use-case analysis and subsequent feature implementation. During analysis of the project, 16 use-cases were identified to guide the development of the app. However, during the implementation process, changes were made to the original plan to accommodate clarified end-user description. Several

Table 1: Prospective Schedule

WEEK(s)	Duration in weeks	Milestone	Description
1	1	CONCEPT: Model	Model Gathering and Modelling of Requirements, Domain discovery, Technology Research
2	1	CONCEPT: Environment	Dev Environement set up, Low Fidelity Design
3-4	2	IMPL: Prototype	Initial React Native Prototype,
5-10	6	IMPL: Core	Implementation, Design Iterations & Continuous Testing
11	1	IMPL: Final	Final feature addition/modification & Deployment
12	1	REPORT	Finalize the Implementation, Begin Report
13-14	2	REPORT: FINAL	Finish report, Present Results, Lessons Learned

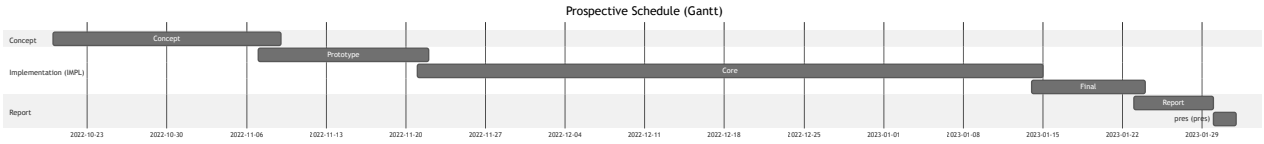

Figure 2: Prospective schedule: Gantt Chart

modifications were made in comparison to the original use-cases, notably the statistics calculated based on the trip were removed. Additionally, new functionalities were added to the app, such as the ability to export trips and database backups, restore from previous backups, and the provision of image proof of the odometer reading. As a result of these changes, the number of use-cases increased to 21 and the number of implemented features increased to 26. The definitive list of implemented features is shown in table 2 and the use-cases are depicted on figure 6.

4.1 Trip creation

The core feature of the app is the ability for users to easily create trips. The sequence of steps for starting and ending a trip, which are subsequently stored in the trip history, is depicted in figure 7 in the appendix. When the app is launched, the user's trip history data is requested from the async storage. To begin a trip, the user taps the `start trip` button on the Overview screen. This takes the user to a camera prompt where they are prompted to take a picture of the car's odometer. The picture is then cropped, compressed, and uploaded to Azure Blob storage. It is also sent for processing to Azure Computer Vision for OCR. Once the recognized values are sent back to the app, the user selects the correct value and the trip is considered in progress until the `end trip` button is pressed.

Ending a trip is a similar process, with the addition of a trip summary screen being displayed. After confirming the values on the summary screen, the collected data is persisted to async storage and the user is redirected to the Overview page. Here, the user can see the updated trip history. The app's ability to easily and accurately track and store trip data is a key feature that makes it a useful tool for managing car logbook information.

5 Architecture

For the purposes of the app, a student account for Azure Cloud services has been set up. The account provides a cost-effective solution for the app's needs, since it enables usage based on the pay-as-you-go pricing model. This account provides access to a range of cloud-based services, including Azure Computer Vision and Azure Blob Storage. Azure Computer Vision is used for optical character recognition (OCR) from sent images while Azure Blob storage is used for storing images sent for recognition. The app communicates with cloud services via HTTP as depicted in the component diagram on the figure 3.

As depicted on the deployment diagram on the figure 4, the prototype of the app runs in Expo Go app, a container through which native APIs of the physical device is accessed . Data persistence within the app is managed using Async Storage which abstracts communication with the device-specific database implementa-

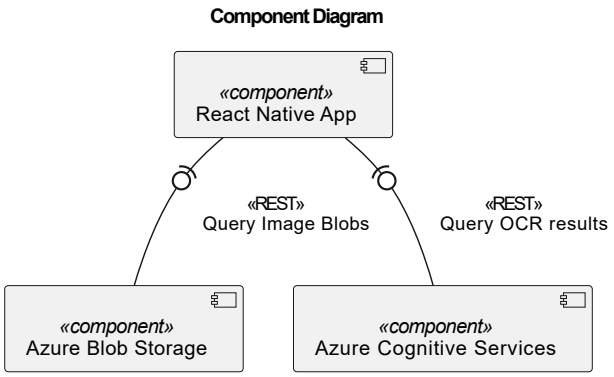

Figure 3: Component Diagram

tion. This physical infrastructure allows for efficient and secure data storage and processing, resulting in a seamless app performance.

5.1 Layered Architecture

The logic of the app has been structured using a layered architecture, as illustrated in the component diagram on the figure 8. There are four layers:

1. Presentation Layer
2. Application Layer
3. Domain Layer
4. Persistence Layer

The Presentation Layer consists of individual screens that encapsulate components, responsible for displaying the user interface. The components in this layer are designed to be reusable and are used to create the visual structure of the app. The Application Layer consists of components that encapsulate logic using logic defined in Contexts. The components in this layer are responsible for handling user interactions and updating the state of the app. It is the bridge between the Presentation layer and Domain layer. The Domain Layer consists of Context objects representing global state that invoke methods from Persistence layer. The Contexts in this layer manage the state of the app and provide the necessary data to the Presentation layer. The Persistence Layer is responsible for communicating with the storage. It implements methods for storing and retrieving data from the storage. The Persistence layer is the bridge between Domain layer and storage.

This layered architecture allows for clear separation of concerns, making the codebase more maintainable and scalable. The separation of the layers allows for a more modular codebase which can be tested more easily, and it is easier to make changes or updates in the future.

5.2 Directory Structure

To accommodate described architecture, appropriate directory structure that respects unwritten React conventions, avoids ambiguity and separates different components and responsibilities was used. The top level directory tree representation is shown below:

```
/
└── src
    └── app-driver-logbook
        ├── assets
        ├── components
        ├── contexts
        ├── model
        ├── routes
        ├── screens
        └── theme
```

The `api` directory is responsible for subscriptions and calls to the back-end infrastructure. It holds subscription logic to the remote services. The `assets` directory contains all available static resources, such as images and fonts. The `components` directory contains single-purpose reusable pieces of UI that can be used across multiple screens. The `contexts` directory contains controlling layer using methods from the model, represents global state management separated by various contexts. The `model` directory is responsible for abstracting direct communication with the Data Store. It provides an interface for other layers to access and manipulate data, while keeping the details of the data store hidden. The `routes` directory contains the screen routing logic, separating primary and secondary screens. It is responsible for controlling the flow of the application and the navigation between screens. This allows for easy navigation between the different screens of the application. The `screens` directory is where all rendered screens are located. These screens encapsulate containers for components, making it easy to understand and maintain the structure of the application. The `theme` directory contains the global styling settings for the design system. This allows for a consistent look and feel throughout the application, making it easy to change the overall design without having to make changes to each individual component.

6 Code Quality

In order to ensure the quality of the codebase, a number of best practices were employed throughout the development process. Descriptive names were used for variables, functions, and components to make the code easy to understand. TypeScript was used as the primary programming language, which allowed for better understanding of the code by the integrated development environment (IDE). Additionally, emphasis was placed on ensuring that every function contains JSDoc comments. While TypeScript provided automated documentation of parameters and return types, JSDoc was used to provide clear documentation of the function's purpose.

Defensive programming was also employed to anticipate and handle unexpected input and situations, making the code more robust. To maintain readability and clarity, a strict rule was implemented that no more

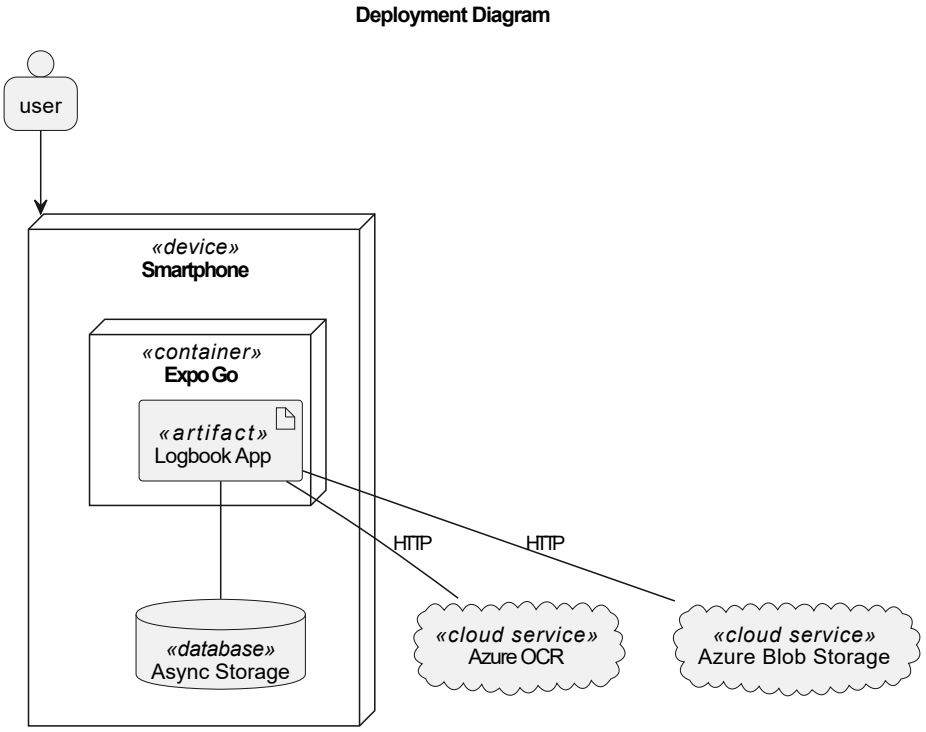

Figure 4: Deployment Diagram

than 2 levels of nesting of blocks of code should be used. In the same respect, for functions that fit on one line `const` followed an anonymous function was used instead of `function` when possible.

Unit testing was conducted using JEST to ensure that critical functions and components were functioning correctly. Furthermore, manual end-to-end testing was also performed with testing scenarios described in `E2E_testing_scenarios.md`. To ensure that the app behaves as expected in real-world scenarios. These efforts were aimed at ensuring that the codebase is of high quality and easy to understand, which will make it easier to maintain and improve in the future.

7 Challenges

Throughout the development process, a number of challenges were encountered. One of the main challenges was setting up the development environment, which was long and error-prone, requiring many dependencies including Android Studio and others. Setting up the Android Emulator, in particular, was challenging to run on Linux. Debugging was also challenging, particularly when setting up the emulator and even more difficult when using a physical device. While Expo SDK provides a vast library of well supported native APIs, integrating outside libraries that do not directly support projects built with Expo SDK tends to be difficult. In terms of styling, React Native uses flexbox for layout, but the styling rules are different than on the web, causing some confusion during styling. Another challenge was the JavaScript engine used for bridging React Native to the platform, which is different for Android and iOS, resulting in minor differences that needed to be tested. Furthermore, in some instances, Expo Go tends to be unstable for one platform while stable for the other, which necessitated additional testing and debugging. Lastly, Expo proves to be dynamic service with frequent releases. This adds to the pressure of constantly updating required tooling while testings for potentially breaking changes, since backwards compatibility is guaranteed for only 3 latest versions.

8 Future work

Next steps for for the application is build and distribution of the production version. Publishing of the application are possible via native application stores operated by each respective platform. While Android charges flat one time fee, Apple charges for annual subscription with its Apple Developer program. Since the app uses remote services with limited free tier, potential paths of app monetization should be explored to cover the expenses. To make the app commercially viable, there are several options to choose from, including advertising, paid subscription or one-time purchase.

From the technical perspective, the app should be maintained and supported by releasing periodic bug fixes and feature updates. Some ideas for attractive features are automatized notifications, integration with car interfaces such as car play and android play, integration of the app with popular navigation apps such as Google Maps, Apple Maps, and Waze, refueling price

statistics, cloud backup and other user requested features. A critical area of improvement is fault tolerance. Specifically, crash recovery for occasions when the app is closed or unexpectedly terminated during trip in progress. This could be solved by periodically persisting the trip data into the database.

Lastly, it would be interesting to investigate and compare the developer experience as well as resulting apps of React Native with Googles competitor technology Flutter.

Conclusion

React Native is a viable framework for cross-platform app development. It delivers on its promise while being supplemented by a wide range of optional technologies. For an experienced React developer, the transition from Web environment to app development is seamless. The power of being able to deliver app for both platforms and potentially even for web from single code base outweighs the downsides of having to individually test each platform for minor inconsistencies and relatively lower performance. This is proved by the delivered app that leveraged vast ecosystem of JavaScript libraries, originally intended for web.

Appendix

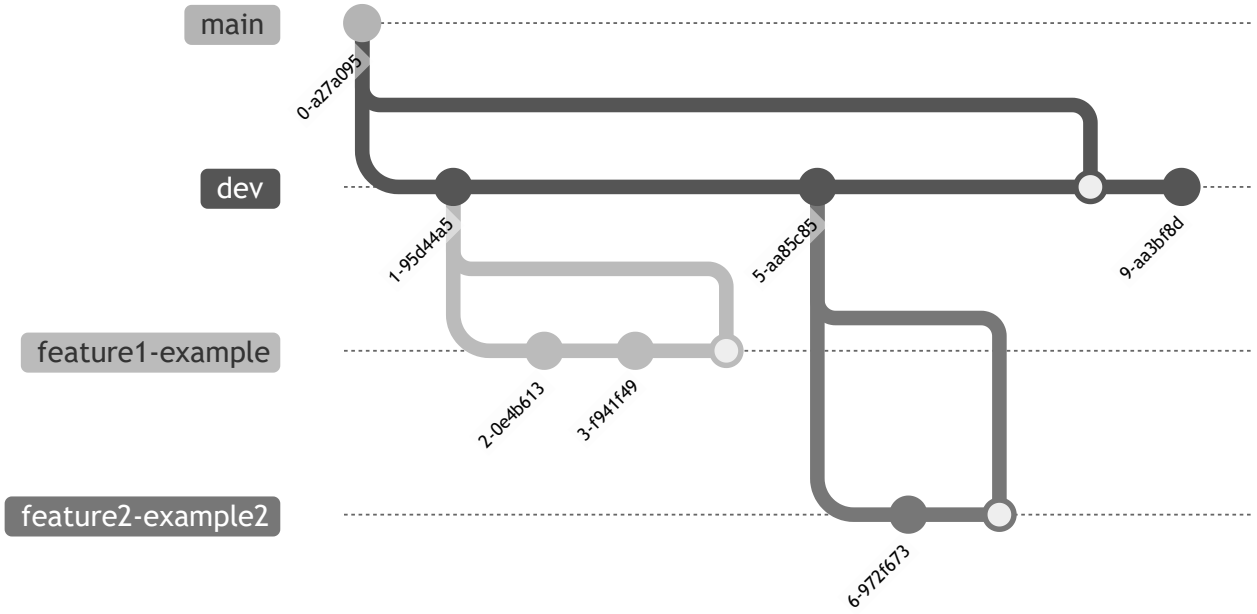

Figure 5: GIT branching strategy

Table 2: Definitive list of implemented features

Feature	Description
Guided image capture	Each trip starts and ends by taking an image of odometer. User is guided to take an image such that the value of the odometer is visible in masked rectangular area.
Image Manipulation	Captured image is cropped to fit the intended area and compressed by factor of 0.75
Azure Computer Vision OCR	Recognizes textual content from sent image and returns results to the app
Azure Blob Storage	Remotely saves captured & processed images, that can be later retrieved
Odometer Image Proof	In trip history (home tab), each trip has a button to download and display the images and optionally to store these images on device
Retrieving location	The app utilizes the device's location services to determine the current location When the odometer picture is taken.
Elapsed time of trip in progress	When Trip is in progress the user see clock updated in real time, informing about how long it has been since the start of the trip
Bottom Navigation	Navigates between tabs a.k.a. secondary pages (Overview, Saved Places, Profile, Settings)
Persisting Trip	when a new trip is in progress it is kept in memory until it is ended, after ending the trip each trip is persisted in local storage
Pagination	trips are organized using sequence numbers, enabling storing trips as separate entities in the database resulting in better performance
Chronological sort of trips	sorted from the latest trip by default, this can be changed to sort from the oldest loaded trip
Load More Trips	By default only 5 trips are loaded in Overview, if more trips are available 5 more can be loaded, until there are no more trips left

Continued on next page

Table 2 – continued from previous page

Feature	Description
Edit Existing Trip	In admin mode, each field of the stored trip can be edited (start location, end location, start odometer, end odometer, start time, end time) such that updated value is not in conflict with potential previous or next value. Each trip has a 3-dot button to access the updating menu.
Editing trip summary	The final step in creating a new trip is displaying a summary of the trip's data. Each field, including the Start Location, End Location, Start Odometer, End Odometer, Start Time, and End Time, can be revised before confirming, ensuring that the updated value does not conflict with any previous trips.
Delete Existing Trip	Each trip can be deleted from the local storage
Simple Trip Share	Information about each trip can be shared individually in a textual form using native share dialog
Export Trips in CSV	transforms trip information to a CSV file that can then be shared using native share dialog
Export Trips in JSON	transforms trip information to a JSON file that can then be shared using native share dialog
Export Backup	exports local storage in its entirety including trips, saved places and user information as JSON file.
Restore from backup	Exported backup file can be used to recover to a snapshot of database
Saved Places	User can create and edit list of frequently visited locations. These can be then used to adjust location of the trip.
Vehicle Information	On Profile screen user can store and edit information about his vehicle (Vehicle name, license plate, km until service, km until oil change, vignette expiration date and type of tires)
Local Storage	For performance benefits the app uses native database of the device through the JavaScript abstraction Local Storage, storing only textual JSON data.
Data Protection	All data are owned by the user. Except for the remotely stored images, all data are stored locally on the device.
Offline mode	App can be used without connection to the internet for viewing and editing functionalities, for creating trips the internet connection is required (guarding prompt is displayed)
Experimental Mode	In this mode, the app selects the recognized value automatically, or if recognition fails, it attempts to select the value of the last trip, saving the user several steps when creating a trip.
Demo Mode	Demo mode allows the user to test features of the app with mocked data (trips, locations and odometer values).
Debug Mode	For a developer the debug mode allows for logging capabilities (log latest trip, dump contents of database, purging database and restarting the app).

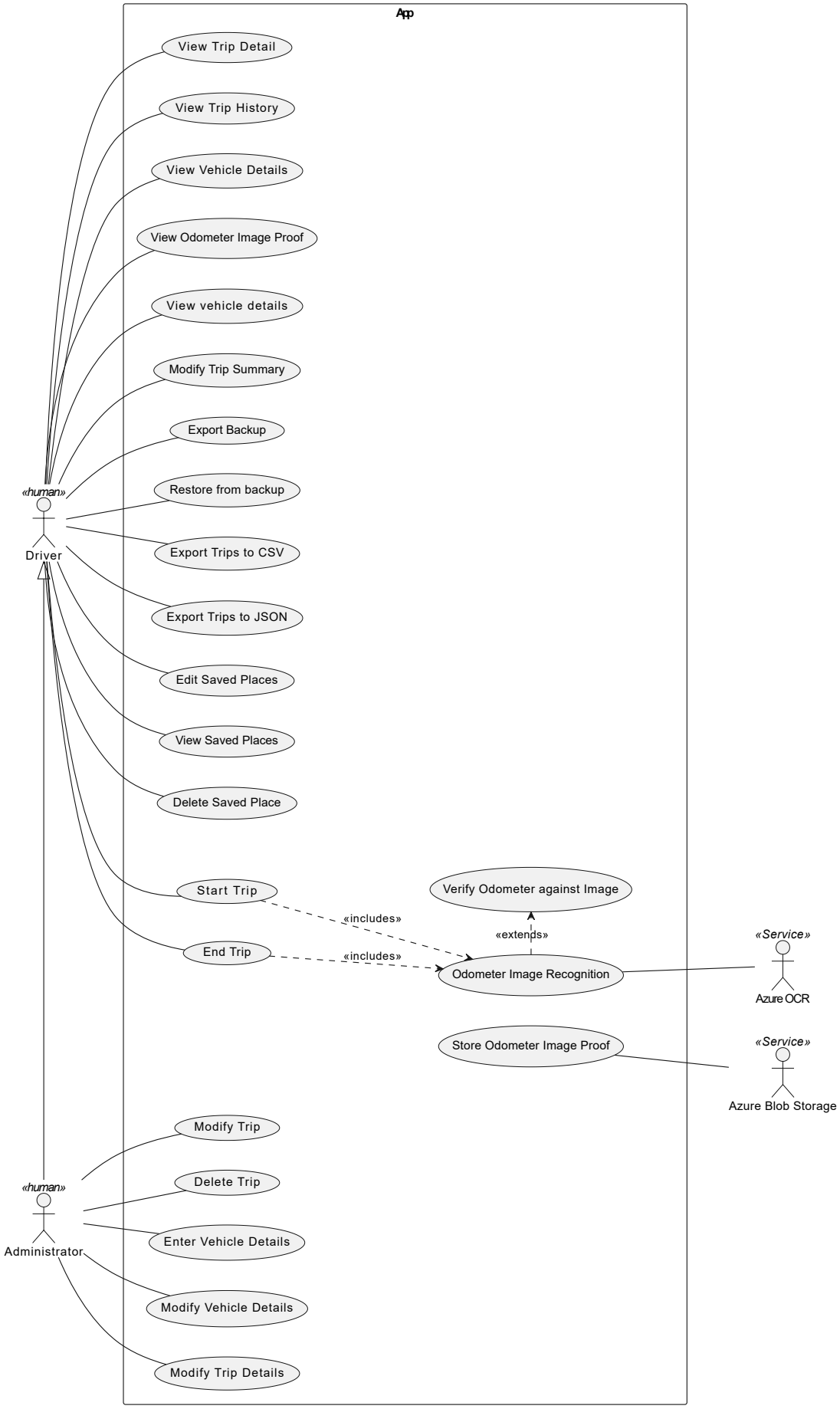

Figure 6: Use-case diagram

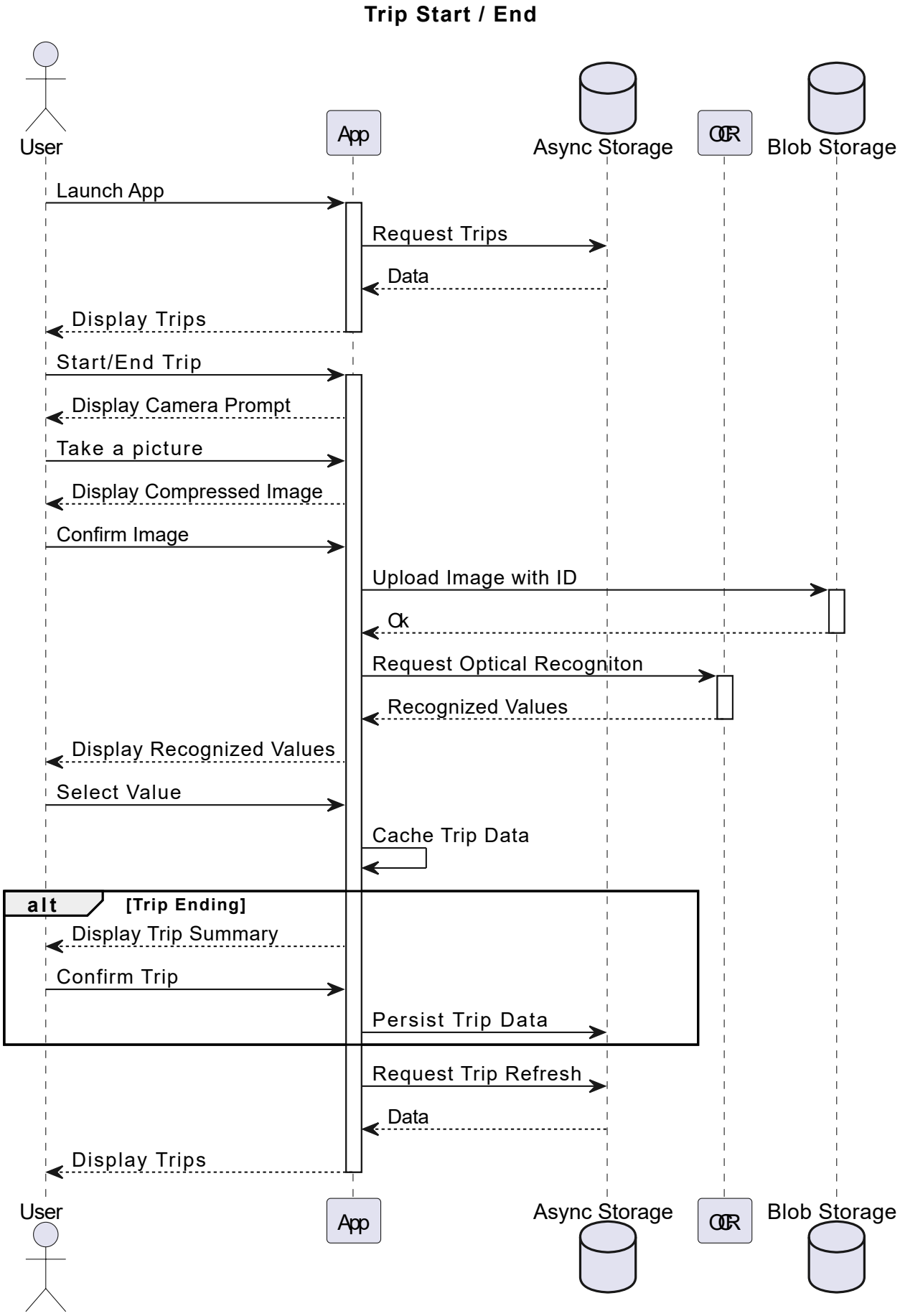

Figure 7: Sequence diagram for starting and ending trips

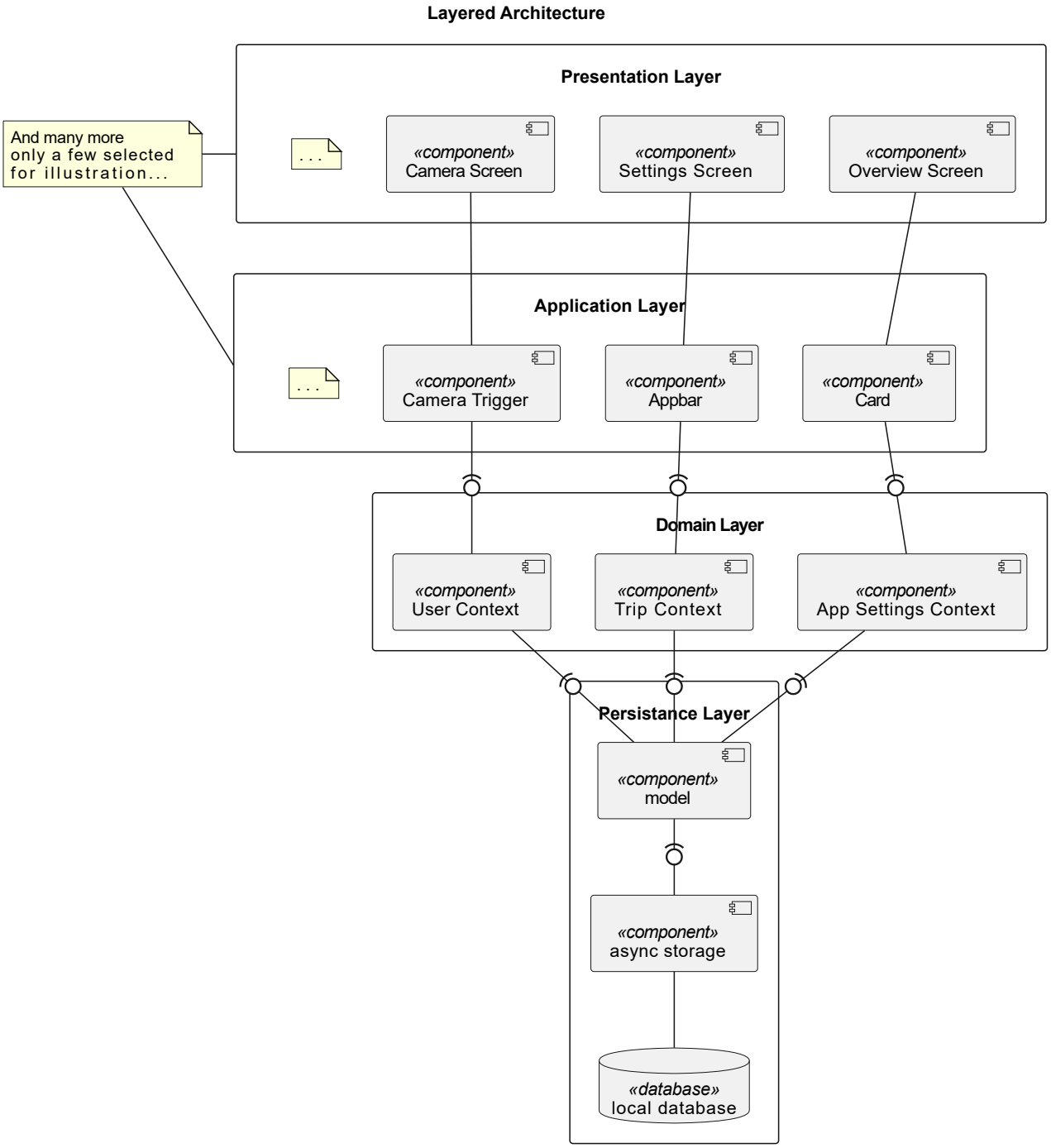

Figure 8: Layered Architecture

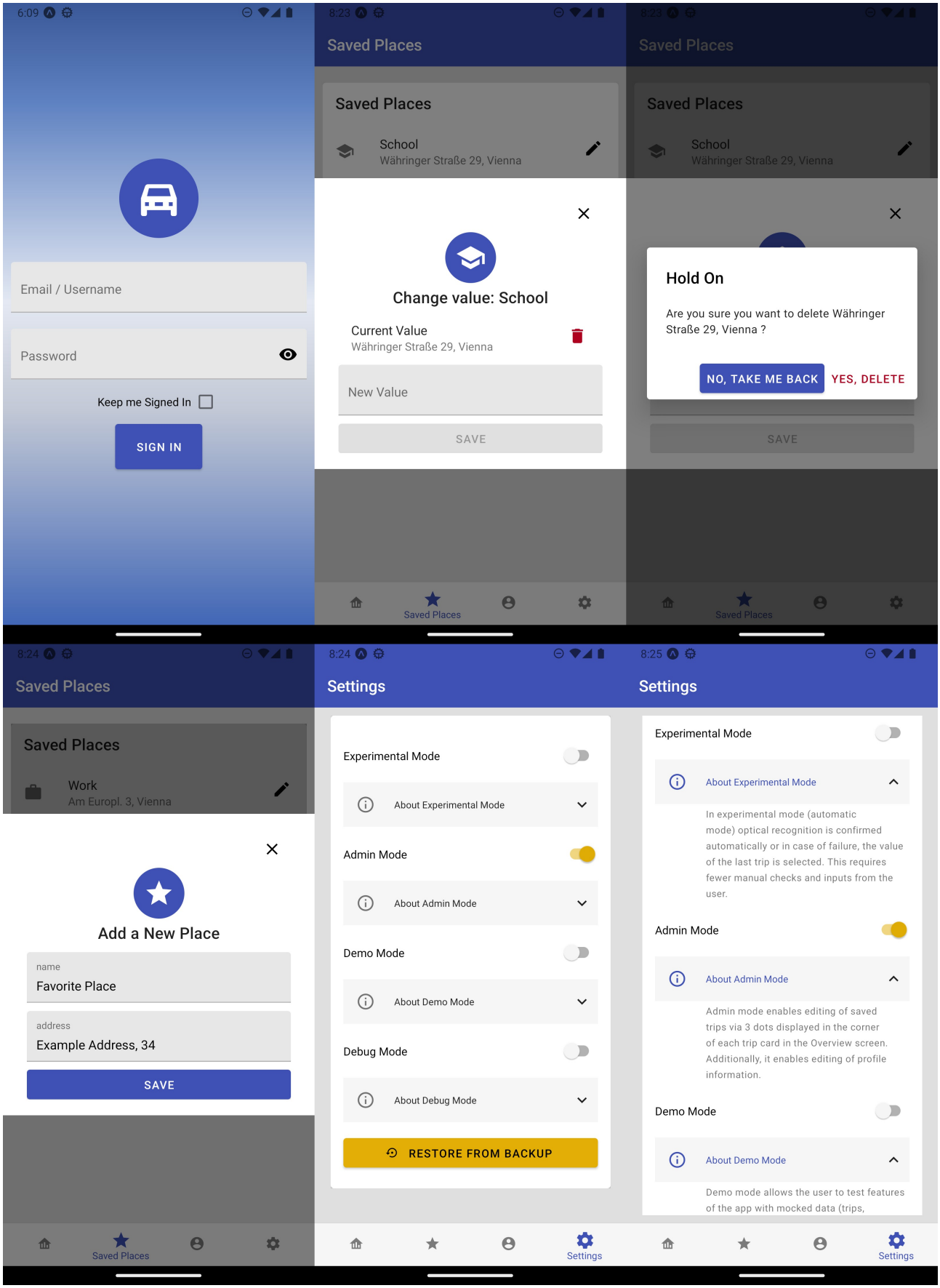

Figure 9: Screenshots from the App 1

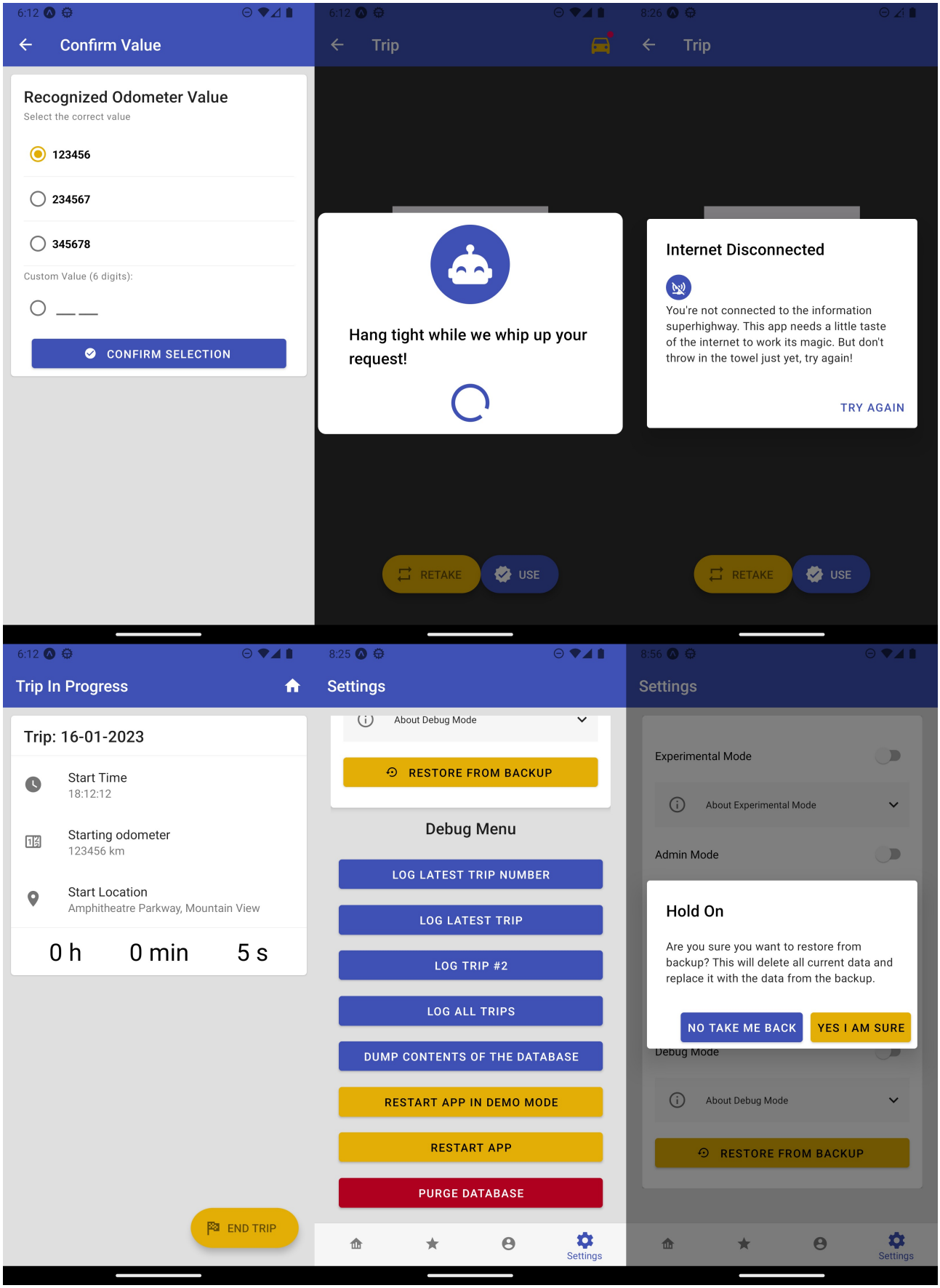

Figure 10: Screenshots from the App 2