OULUN AMMATTIKORKEAKOULU

Khoa Dinh & Tram Vu

GOVIET:

CAR SHARING APPLICATION FOR SOLO TRAVELER IN VIETNAM

GOVIET:

CAR SHARING APPLICATION FOR SOLO TRAVELER IN VIETNAM

Khoa Dinh & Tram Vu
Final project
Spring 2024
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Thesis

Author(s): Khoa Dinh & Tram Vu
Title of the thesis: GoViet: Car Sharing Application for Solo Traveller in Vietnam
Thesis examiner(s): Jukka Nevalainen
Term and year of thesis completion: Spring 2024 Pages: 55 + 11 appendices

In the context that all countries are simplifying immigration procedures to attract more tourism resources, Vietnam is also among the countries that want to promote tourism promotion programs. to be known by more and more international tourists.

However, due to certain limitations such as language and complicated public transportation system, it has partly affected tourism development. Realizing those limitations, GoViet was born to help foreign tourists in Vietnam visit more tourist destinations easily, quickly, and at a reasonable price.

Furthermore, to suit the majority of modern users, GoViet was developed as a mobile phone application using the React Native software framework, based on the JavaScript language platform with the React software framework mainly used for Web development. With React Native, this is a software framework that is constantly updated, has a large user community, and mobile phone applications can easily run on two platforms: Android and IOS. Combined with the use of Firebase services as a place to design, store and query data sources along with Google Maps Platform in the map part, GoViet hopes that end users will have a good experience in locating, booking a ride efficiently and quickly.

Based on the needs of booking a ride to travel from one to another location whilst travelling, GoViet was developed in four main features: Sign in or Sign up thanks to Firebase Authentication method, Book a ride and Publish a ride using Google Maps Platform to navigate pick-up and drop-off point, and lastly, User profile where a passenger can also be a driver at the same time.

Keywords: Travelling Application, Mobile Application, Solo Traveller, React Native, JavaScript, Android, IOS, Google Maps Platform.

CONTENTS

ABSTRACT .... … 3
ABBREVIATIONS AND TERMS 5
1 INTRODUCTION 7
2 WORKING DIAGRAMS AND GRAPHICAL PROTOTYPE 8
2.1 Working Diagrams 8
2.2 Graphical Prototype 10
3 CODING FRAMEWORK: REACT NATIVE 12
3.1 Framework Comparison 12
3.2 Basic Operations 13
3.2.1 Render Method 13
3.2.2 Props and State 13
3.2.3 Component Lifecycle Methods 14
3.2.4 Style 14
3.3 Common Components 14
3.3.1 View 15
3.3.2 Text and TextInput 15
3.3.3 FlatList 15
3.3.4 Modal 15
3.3.5 Button and TouchableOpacity 16
3.4 Stack Navigator 16
3.5 Redux 17
4 DATABASE: FIREBASE 18
4.1 Firebase Authentication 18
4.2 Firebase Cloud Firestore 20
5 GOOGLE MAPS APPLICATION PROGRAMMING INTERFACE 23
6 RESULTS 25
6.1 Onboarding Screen 25
6.2 Authentication Screen 26
6.3 User Profile Screen 28
6.4 Home or Book a Ride Screen 29
6.5 My Rides Screens 33

6.6 Publish a Ride Screen 34
7 DISCUSSION AND SUMMARY 39
REFERENCES 40
APPENDICES 42

ABBREVIATIONS AND TERMS

API	Application Programming Interface
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
SDK	Software Development Kit
SQL	Structured Query Language
OS	Operational System
RNF	React Native Firebase
UI	User Interface

1 INTRODUCTION

In the current era of technology 4.0, it is safe to say that mobile phone has become an indispensable item for everyone when phone applications exist in every aspect of life. For example, users can play games, control home devices such as lights, camera systems, remote security locking systems and many more. An application fad that probably contributes as a game player is the banking application, where the user's personal information is protected safely through various account authentication steps to prevent information leakage.

To contribute to increasing the diversity of the mobile application ecosystem, GoViet was born with the purpose of making traveling in Vietnam quick, convenient, and budget friendly for foreigners. And with the purpose of easily reaching as many users as possible, the application has been built for both Android and IOS operating systems.

Besides the benefits that the application brings, users can have another interesting practical experience when using GoViet, which is privacy, safety during each trip, flexible booking time as well as care-free time. No more concerns about the language barrier which is completely a different story from using public transportation that costs quite the same.

In the future, GoViet will upgrade with many different types of services besides the current limit of just booking trips. These can be short sightseeing tours where the driver can accompany as a local guide, updating more diverse payment methods (currently only paying in cash).

The thesis has been divided into six chapters. Chapter one introduces the purpose, and the general components of the application Chapter two mainly shows the flow of the application as well as each screen and its function. Followed by chapter three is about the usage of coding framework in creating the mobile app. Deeply diving into the back end of the app by getting to know the Node.js server and the Firebase services which were employed, that have been clearly described in chapter four and chapter five, respectively. The last piece in creating GoViet is Google Maps API which is put into words in chapter 6.

2 WORKING DIAGRAMS AND GRAPHICAL PROTOTYPE

2.1 Working Diagrams

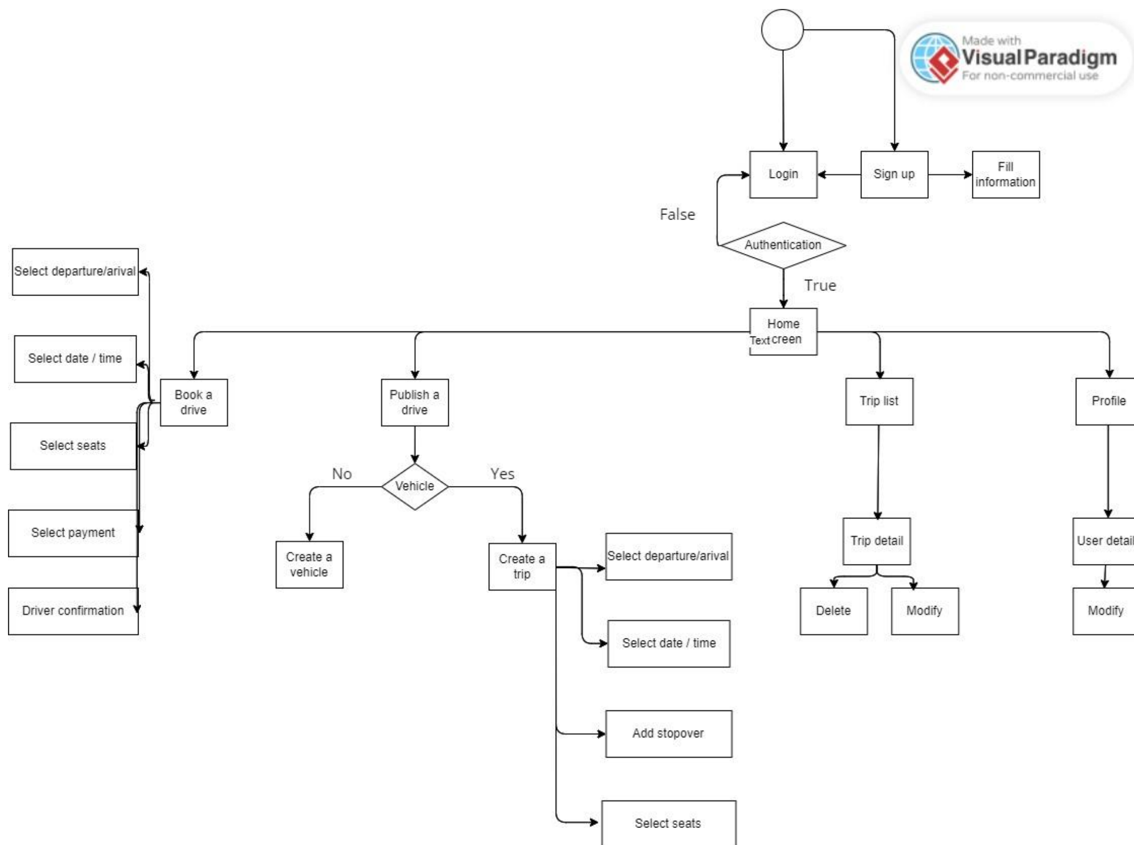

FIGURE 1. System Flow Diagram

Figure 1 illustrates the five main groups of activities of GoViet which are: Authentication, User Profile, My Rides List, Book a Ride and Publish a Ride. Technically speaking in general, GoViet requires users to sign in or log in to benefit its services, therefore, Authentication step is always the very first one to have everything started. There will be two flows of the system after step one is under processing: if the account is totally new, user will be navigated to User Profile for modifying their personal information; whereas, once the system has verified the already in-use account, user will be directed to Home screen or so-called Book a Ride, in which they can start booking rides based on their needs. When the booking process is undertaken, user as a Passenger will be guided to My Rides screen containing all the ride or trip options from the Drivers who published those as a list and easily making up their mind.

The last group is somewhat separate in the flow of the system; however, it also plays a crucial role in the whole. As GoViet is built based on simple but adequate criteria, so one user can be a Passenger or a Driver at the same time. Hence, user after registering themselves as a Driver, they can navigate to this group and post a ride with all necessary information.

From a user wise in Figure 2 below, a closer look can be profoundly accounted for the above system flow.

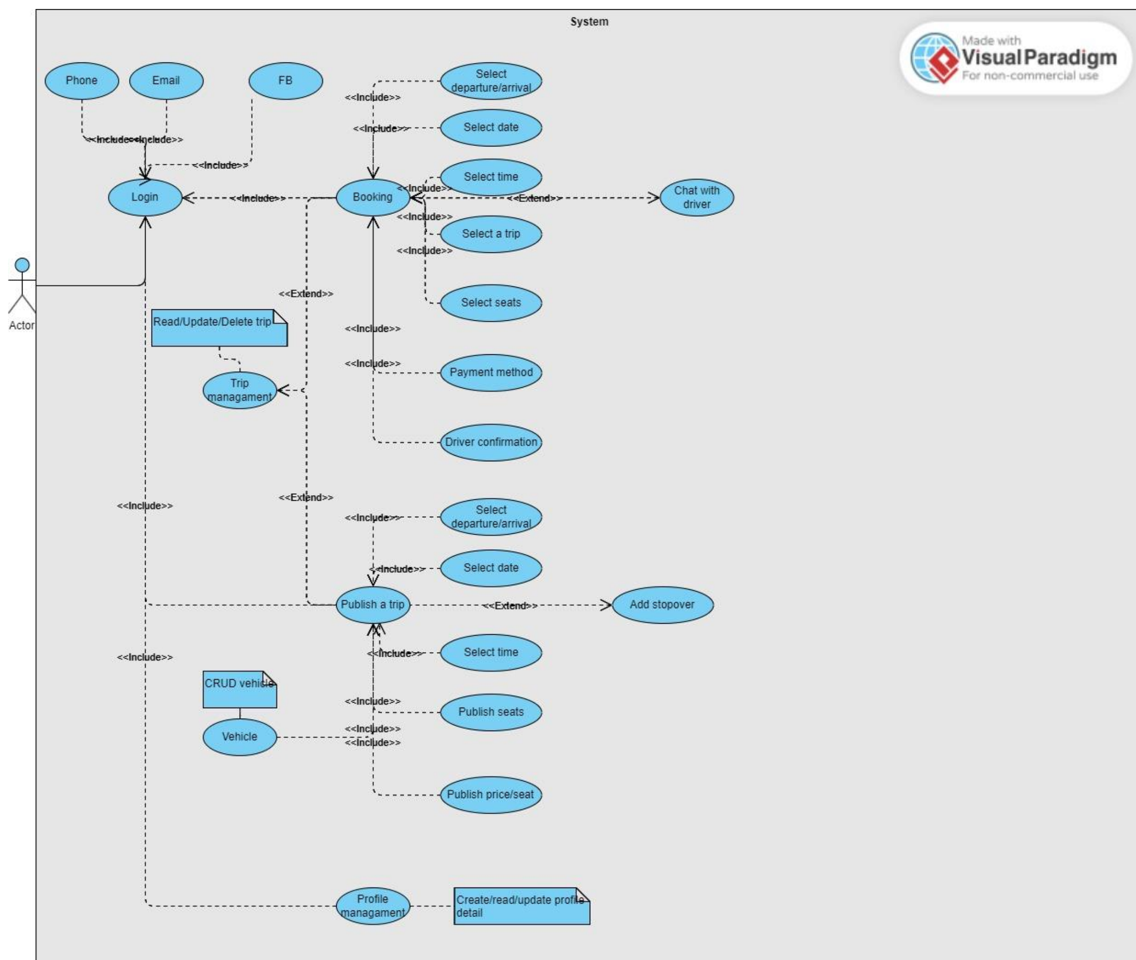

FIGURE 2. Use Case Diagram

In the Authentication step at first, the user can choose either contact number, Email or Facebook as an option for registration. However, the contact number is always a mandatory step as the system will send the verification code through the phone number for account confirmation. This applies for a new user only (who creates new account), while for those who already have an account, these steps can be shortly skipped.

The main function as well as the Home Screen of this application is booking a ride. The user as a Passenger looks for a ride or a trip based on their needs by selecting departure and arrival locations, date, and time for the ride as well as number of passengers. After this step, they will be directed to My Rides screens consisting of all related rides, choose the best suit one and heading toward to the payment method. When everything is decided, then the system will send the booking request to a corresponding Driver who published that ride for approval.

Profile management screens allow the user to create, update and register to become a Driver by uploading all the related vehicle information as required by the system.

User who is a Driver, can publish a ride in Publish screens also by typing all the necessary information for that ride such as: departure and arrival locations, date and time, price of the ride as well as the availability of seats. After posting about the rides, Driver can go to My Rides screen as of right now it will show all the publicly rides and wating for the booking from a Passenger.

2.2 Graphical Prototype

The development of the first design is tremendously fundamental for developers to have a comprehensive view of the direction of the system in general and each feature in particular. Furthermore, the first look helps to resemble the final product after a lot of development and adjustments, identity potential issues to get the full-scale production. In this phase, to construct a high transparent prototype, a powerful design online tool named Figma was applied to materialize our notion, by building graphical visualization and functional features based on many materials and elements.

The preliminary model in Figure 3 displays a general view of screens in which containing all functional features and navigation, testing the system workflow as mentioned above which is a helpful criterion for user experience.

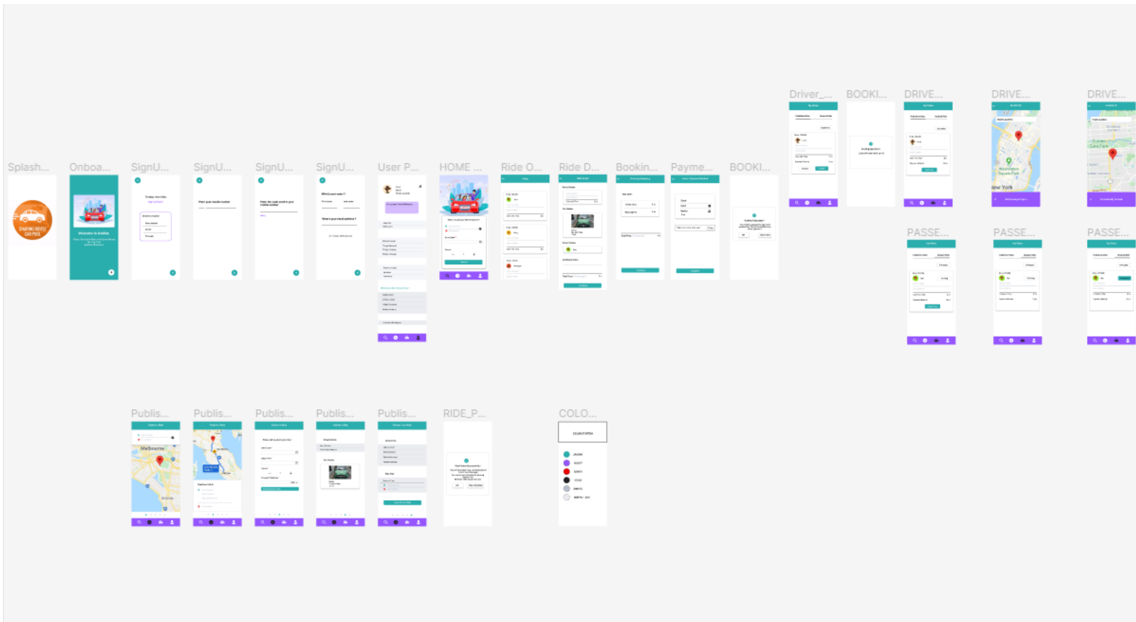

FIGURE 3. The Preliminary Model of GoViet

3 CODING FRAMEWORK: REACT NATIVE

The main purpose of this project is to support the end users the most convenient booking application as much as possible in the most two common mobile platforms: Android and iOS, therefore, the developer team needs to decide which is the best suit of hybrid language between React Native and Flutter, that allows to create both OS at the same time.

With the web development skills which have been accumulated throughout years, and the familiarity in using JavaScript and its framework - React as a coding language, React Native is the perfect chosen one because of its clarity and simplicity when applying it to writing code. Besides, there are more reasons why this is a great pick that were explained in the upcoming part.

3.1 Framework Comparison

Flutter is built using native C/C++ and Dart, contrary to React Native which is JavaScript- based programming language. UI development is one of the contributing reasons for that Flutter is discarded in the application. With a unique set of widgets of its own, Material Design widgets and Cupertino widgets, making the UI components behave naturally just as the native ones. Nevertheless, since everything needs to be built from scratch using this unique technique, it might lead to a result of inconsistency among different platforms. On the other hand, React Native supports a declarative Style attribute and APIs acting like bridges to connect and integrate the iOS and Android codes as JavaScript modules. Moreover, the language offers platform components, for instance, the button in React Native will be identically and technically translated and transformed into iOS system or Android system button with just a block of code, helping the UI rendering easier and more timesaving but the result is like the native ones. (1.)

Another plus point of React Native is that it was invented by Facebook (Meta) in 2013 and released in 2015, the long developing time making it has a large community of developers, a state-of-the-art coding language that permit to go deeper to write native language like Swift, Java if needed (2).

3.2 Basic Operations

React Native employs only JavaScript to it mobile applications and uses the same declarative methods, attributes and UI components as React does to design and display things on the user screens. The apps created in React Native are not originally mobile web applications based, since it takes advantage of the identical UI building blocks as normal iOS and Android apps. Meaning that instead of using native language such as Swift or Java, a developer can seamlessly utilize JavaScript and React to put those UI blocks together. (1.)

3.2.1 Render Method

After coding a block of code of components, the system will process those lines and result in the display of what needs to be shown in the screen. The process is called render, consequently, every component must have its own render method, from the simplest one such as View component to the very complex one. Render needs to be put at the end of each coding block inside the *render()* function part so that it could wrap around that component to show what is visible to the end users, contrarily, there will be a “fatal error screen”. (3.)

3.2.2 Props and State

In each component, there are always two parameters: a property (or so-called props) and state. Props is a way that letting a developer to access, customize or pass data from a parent component to a child component. It is set wherever the component is created to give more information or detail of that component, and is accessed by the code: *this.props* or by utilizing some props named Style, Source, etcetera. As props are static, as a result, they cannot be modified after a component is rendered. And because of its inheritance feature, if there are changes in parent prop’s value, the child’s one will also be re-rendered to get updated. (4.)

Besides, state is a parameter that keeps track on any changes within a data storage and is initially declared by the syntax *setState* within a component constructor scope. It usually is a result of a trigger activity by the user. For example, when the user wishes to update their profile personal information, after inputting all the needed information, they hit the Save button, that action has

triggered the initial state which has old information, then setting new updates on it via *setState*. This leads to the mentioned changes in component, so everything will be re-rendered. (4.)

3.2.3 Component Lifecycle Methods

In general, each component, particularly class component has phases of its own and several lifecycle methods. One does not need to pass all the phases altogether, but based on the purpose of usage, and in specific times. According to the mean of use, a certain action or a specific implementation will be taken into account with some familiar methods like: *componentDidMount()*, *componentDidupdate()*, *componentWillUnmount()*, *componentDidCatch()*, etcetera. (5.)

Besides, specifically the functional components which have been used in GoViet, Hooks have played vital roles, however, not directly replaced the lifecycle methods. Some familiar Hook functions are *useState* which is mentioned in part Props and State to manage local states, useEffect which has identical functions as *componentDidMount()*, *componentDidupdate()*, *componentWillUnmount()* in class components, *useSelector*, *useDispatch*, etcetera.

3.2.4 Style

As the name says its function, Style has played a vital role in designing, displaying and making the app attractive to users. Style is usually considered as CSS for mobile app designs and each component can possess its own prop Style value to define the exclusiveness. When it comes to a complex design that requires more than one style on it, applying the component StyteSheet with syntax *StyleSheet.create()* to help the block of code have reusability and ease of maintenance. (6.)

3.3 Common Components

React Native in general has a long list of cores or common components which have been applied effectively by developers based on the purpose of use. Within the scope of this thesis, five of the most used components to build GoViet will be introduced. They are View, Text and TextInput, FlatList, Modal, Button and TouchableOpacity.

3.3.1 View

The most fundamental component acted as building blocks in creating UI. Technically speaking, this component has the same function as <div> tag in HTML in web-based development, which is like a container for other components such as Text, Image, Button and many more. It has a default layout with flexbox which is identical to CSS in web development, style property, touch handling and accessibility controls which are more helpful in creating applications for the disabilities. Since a container can contain other containers as well, as a result, a View component is able to carry more than one View thank to its nested design. (7.)

3.3.2 Text and TextInput

Text is a component mainly used for showing text in user's screens. Similarly to View, Text also supports nesting, styling and touch handling that give it more useful, functional in UI creation. Another updated level of Text is TextInput component which allows users to directly input the needed information under various types of formats. TextInput provides some events which trigger the action for text such as onChangeText() which update the state of the current text (mainly employed in GoViet), onSubmitEditing(), onFocus(), onPressIn(), onPressOut(), etcetera. (7.)

3.3.3 FlatList

This component is effectively used to show smoothly scrollable many types of lists in GoViet like the Ride List whose syntax will be shown in the Appendix 1 and 2. The component will receive an array of data (which has been declared and filled with information before) from its inner prop called *data*, in which each item has its own id number that letting FlatList know exactly which item should be extracted by calling *keyExtractor()* function. The item list that needs to be rendered can be customized through a prop called *renderItem()*. This function triggers a render method for an item that has been declared with adjustment to be shown in user's end. (7.)

3.3.4 Modal

This component is the simplest way to show a temporary content or action to the end user after one or many actions were triggered, usually under forms as a pop-up, push notification or small,

alarming bar. Due to its characteristic, the component overlays the current screen during a short period of time with a customized appearance to remind user about the input or notify the complement of actions. Modal visibility is controlled by State, particularly, by *setModalVisible* state and it also has some props creating effects on it like *animationType*, *transparent*, *visible*, etcetera, which are shown in Appendix 3. (7.)

3.3.5 Button and TouchableOpacity

Button is the most basic component in React Native and a platform-based component, hence, it runs smoothly and adaptively in multiplatform giving the outcome in each OS as similar to the native ones. The mandatory prop for this component is *onPress* which will trigger simple actions such as submitting or navigating after pressing it. For more complex actions that require more interactions from touch gestures such as choosing number of seats, pressing to trigger a state, TouchableOpacity button with the compulsory *onPress* prop is brought into play in the application. (7.)

3.4 Stack Navigator

It is the norm that in most mobile applications, there are always more than one screen to show the whole apps content and its benefits. User can freely move around to discover each screen by pressing buttons, slide left or right or even scroll up or down the screens. As React Native itself does not have this built-in feature, for that reason, an external library called React Navigation will be installed.

At first, the whole app needs to be wrapped inside *NavigationContainer* component, which should be placed ideally in the entry file. Then creating the stack with *createStackNavigator()* in which all the screens were added for the movement. To access to this container, the *Navigator* component was applied, followed by *Screen* component to define the list of screens in the current stack. From this step onward, each screen will become one independent component and have a prop called *navigation* including *navigate()* method that creates system flow from this screen to the screen whose the name was put in brackets.

Additionally, a navigational bottom tab bar was also constructed by utilizing *createbottomTabNavi-agator()* component. The code in Appendix 4 will show a deeper look for this explanation.

3.5 Redux

To optimize the application's state management, the very well-known Redux has been brough into play. Redux was mainly applied to the application's authentication methods, asynchronous operation, error handling as well as user profile update.

Firstly, all the actions that want to be managed needed to be declared, controlled by Redux through its *reducers* predictable state functions. Taking userAction.js file, whose code will be displayed in Appendix 5, as an example. The input for the initial state is the user data from which whenever a user inputs personal information to update their profile, the next actions will be dispatched as assigned via *then()* function which are updating user data and navigating to Main Screen (this happens to new user).

Reducers itself are simple functions returning the same output for the given input without affecting other components or unassigned actions. The function basically consists of two arguments: the initial state and the action need to be dispatched. Continuing to take the userAction.js as an example, for utilizing this function, another file called useReducer.js was created to handle all state changes for profile update, which will be shown in Appendix 6. Reducers takes the switch statement to perform many types of actions in each case while controlling its manners. Accessing the action type using *action.type* prop which is "UPDATE_USER" in the code, as a result, the case performs updating the profile and return a new state based on the given payload which is user data. (8.)

4 DATABASE: FIREBASE

Since the launch in April 2012 by Tamplin and Andrew Lee, Firebase caught the eyes of Google by its progressively well-known technologies such as real-time database, as a result, two years later it was obtained by Google and has been developed rapidly since then, making it one of the most famous Backend-as-a-service (BaaS) tools in continuously mastering real-time database, cloud storage, cloud hosting, authentication in web and mobile applications (9).

For this application, Firebase is a top choice for all of it support, particularly, its authentication and database management services.

4.1 Firebase Authentication

Most of the applications would require user identification in order to securely save personal data in the cloud and effectively enhance the sign-in methods. The feature offers a wide range of ways for an end-to-end authentication such as email and password; telephone number verification, Facebook, Gmail or GitHub login, which have been integrated very well in GoViet giving the end users an outboarding experience (9). The coding lines in Appendix 7 represents the phone verification method in the application. This has been achieved by utilizing the FirebaseUI – UI libraries supporting the UI flows in sign-in process as well as the Firebase Authentication SDK integrating one or more sign-in methods into the application (10).

Once one account is created, all the related user detail will be also generated and appeared in the Firebase servers, furthermore, a unique identifier (uid) for each user will relatively be produced, which will be displayed in Figure 4. This is technically very helpful in making the logical flow when user signs in or logs in to the application, especially for those who already had the account in GoViet. Thanks to this technique, the function called *checkUserExistence()*can be set*,* in which "uid" was used as an essential criteria distinguishing between the initiated account and the brand new one. From the second use of the ready-made account, user will be directly taken to Main Screen thanks to *navigate()* function of React Navigate, the other one will be navigated to User Profile Screen for personal information update or adjustment.

Additionally, there are three more fundamental methods for the authentication actions:

- loginSuccess()
- loginFailure()
- logout()

These methods were associated with RNF to perform their own functions increasing user experience and were wrapped and managed by reducers functions of Redux to guarantee that all actions must be dispatched correctly as it assigned. *authAction.js* file in Appendix 8 will cover the code of this part.

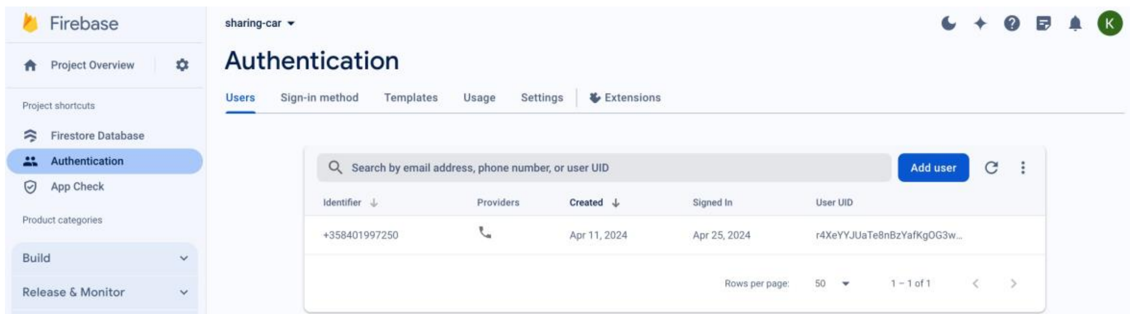

FIGURE 4. Firebase Authentication Server

For handling the sign out activity, the *signOut()* method was imported that was managed and dispatched by Redux whenever the user wants to log out of the app. This action will trigger the navigation accordingly to the system flow and take the user back to the main screen. Figure 5 will cover the code of signing out activity.

```
};

const user = useSelector(userData);
const {name, email, phone} = user

const handleSignOut = async () => {
    try {
        await auth().signOut();
        navigation.navigate("WelcomeScreen")
        // Optionally, navigate to a different screen after sign-out
    } catch (error) {
        console.error('Error signing out:', error);
        // Handle sign-out error
    }
};
```

Figure 5: Sign Out Handling Method

4.2 Firebase Cloud Firestore

Instead of using Real-time Database, the next generation of it was employed in GoViet, which is Cloud Firestore. Besides the real-time updates like its ancestor, this version automatically scales and manipulates intensely large and tough queries.

For server-side, browser and mobile development, Firebase and Google Cloud provide developers with Cloud Firestore, an adaptable and expandable database management tool. Similarly to Real-time Database, it facilitates offline support for mobile and web, enabling users to construct responsive apps that function regardless of network delay or Internet connectivity. It also uses real-time listeners to keep user data synchronized across client apps. (11.)

Though both Cloud Firestore and Realtime Database are NoSQL type of database and managed and handled by Google Firebase, Cloud Firestore still the most suitable tool to apply in GoViet because of its way of storing JSON-like documents data structures. Firestore is organized based on the Collections-Documents-data model, in which Collections can be considered as directories of a computer and Documents are files in those directories and they must have their unique name, otherwise the newly created one will overwrite the old one having the identical name. Meaning that, one Collections can contain from none to many Documents, plus a Document can have Collections inside of it and so on. This helps in creating a hierarchical data structure, which makes the data query and retrieve much more powerful and flexible. (11.)

Since Collections are like tables in relational databases, within GoViet's scope of data structure, there are three Collections relates to the main purpose of the application: User, Trip and Vehicle, which will be displayed in Figure 6. Taking User Collections as an example, there are seven Documents with a pair of key-value set relatively building in the characteristics of this Collections, in which a key is always a string type while the value can be any type of data based on mean of use.

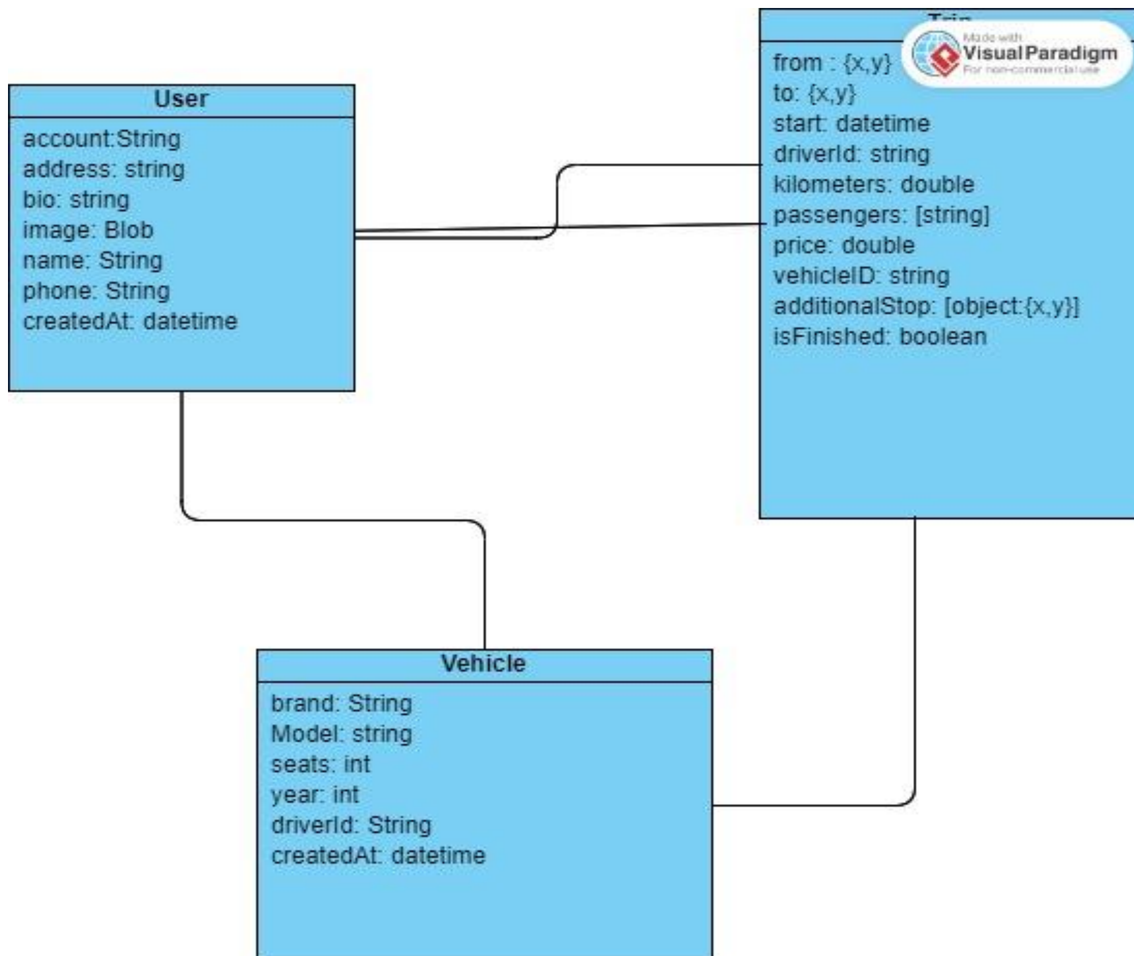

FIGURE 6. The Database Structure of GoViet

As referring this to the application's perspective, User Collections can be considered again. After a user is successfully logged into to the app, *dispatch* method in redux will be used in calling the function *getTrips()* in Figure 7, to retrieve data from the database in Firestore, yet this only is doable for account with fully input information, thanks to checkUserExistence() via the mentioned "uid" (Appendix 8 covers the code already). For that reason, newly created account will be taken to User Profile screen to update and or fill all the related missing information. The differences between and after the action had been made in Firestore server will be shown in Figure 8, and 9. As a default setting, three basic information will be required through the authentication steps which are email address, name and contact number; the other information will be updated later on.

```
export const getTrips = () => {
    return async (dispatch) => {
        try {
            const tripsSnapshot = await firestore().collection('trips').get();
            const trips = [];

            tripsSnapshot.forEach(async documentSnapshot => {
                //Trip data
                const tripData = documentSnapshot.data();
                //Driver data according trip
                const driverSnapshot = await firestore().collection('users').doc(tripData.driverID).get();
                  //Driver data according trip
                const vehicleSnapshot = await firestore().collection('vehicles').where('driverID', '==', tripData.driverID).limit(1).get();

                let vehicleData = null;

                if (!vehicleSnapshot.empty) {
                    vehicleData = vehicleSnapshot.docs[0].data();
                }
                const driverData = driverSnapshot.data();
                const trip = {
                    id: documentSnapshot.id,
                    ...tripData,
                    driver: driverData,
                    vehicle:vehicleData
                };
                trips.push(trip); // Pushing into a temporary array is not a state mutation
            });

            dispatch(setTrips(trips));
        } catch (error) {
            console.error('Error fetching trips: ', error);
            // Handle error, such as showing an error message to the user
            dispatch(/* Action to handle error */);
        }
    };
};
```

FIGURE 7. getTrips() Function

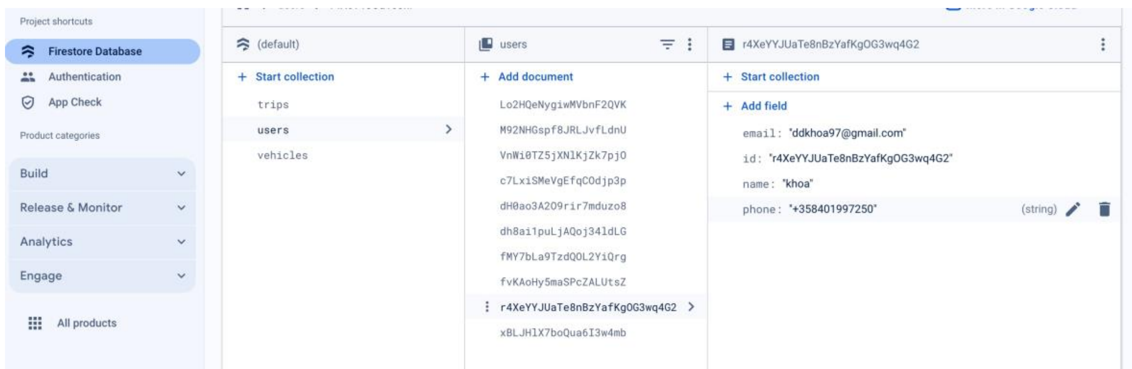

FIGURE 8. Default Setting for User Information

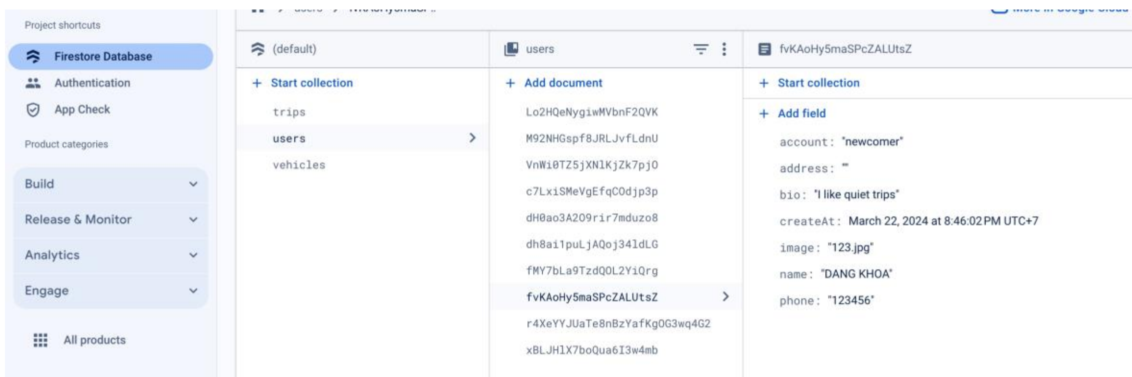

FIGURE 9. User Information Updated

5 GOOGLE MAPS APPLICATION PROGRAMMING INTERFACE

The last but not least part of every transport-based mobile application is probably the map integration. Google Maps API was applied into the application thanks to its various types of APIs which support the navigation, geolocation, checking current location.

To take benefit of this service from Google, firstly, the API key for the project needed to be generated, called “GOOGLE_PLACES_API_KEY”, then installing the dependencies to integrate the Google Maps into the application via the command “npm install react-native-maps” (12).

For showing the map in GoViet’s screens, the codes designing the map appearance were wrapped in <MapView> tag, from that the *GooglePlacesAutocomplete* method was used, which helps the users with a time-saving auto-filling function when searching for location by typing only short words, there will be a drop-down list with related suggestions. Technically, the drop-down list is the FlatLsit component of React Native making the UI more aesthetics. Figure 10 will cover the code of the said part.

```
<View style={{ flex: 1 }}>
    <MapView
        style={styles.map}
        initialRegion={position}
        showsUserLocation={true}
        showsMyLocationButton={true}
        followsUserLocation={true}
        showsCompass={true}
        ref={mapRef}
    >
        {from && <Marker coordinate={from} />}
        {to && <Marker coordinate={to} />}
    </MapView>
    <View style={styles.searchContainer}>
        <FlatList
            data={[{ key: 'GooglePlacesAutocomplete' }]}
            keyboardShouldPersistTaps="handled"
            renderItem={() => (
                <GooglePlacesAutocomplete
                    placeholder='From'
                    onPress={(data, details = null) => handlePlaceSelect(data, details, method = "From")}
                    query={{
                        key: GOOGLE_PLACES_API_KEY,
                        language: 'en',
                    }}
                    ref={googlePlacesRef}
                    fetchDetails={true}
                    styles={{
                        textInputContainer: {
                            backgroundColor: 'rgba(0,0,0,0)',
                            borderTopWidth: 0,
                            borderBottomWidth: 0,
                        },
                        textInput: {
                            marginLeft: 0,
                            marginRight: 0,
                            height: 38,
```

FIGURE 10. Displaying Map with MapView and FlatList

Figure 11 shows how the current location of user and driver are taken by utilizing the method *getCurrentPosition* of “Geolocation” library. Within each process, a pair of latitude and longitude is set to locate the position, and then useSelector() function in Redux which is illustrated in Figure 12, plays its role in taking the premade dataset (“trip” in this case) to return the specific location based on the latitude and longitude delta.

```
useEffect(() => {
    Geolocation.getCurrentPosition((pos) => {
        const crd = pos.coords;
        setPosition({
            latitude: crd.latitude,
            longitude: crd.longitude,
            latitudeDelta: 0.0421,
            longitudeDelta: 0.0421,
        });

        setLoading(false); // Set loading to false once position is obtained
    });
}, []);
useEffect(() => {
    if (from && to && position) {
        const coordinates = [from, to];
        mapRef.current.fitToCoordinates(coordinates, {
            edgePadding: { top: 100, right: 100, bottom: 100, left: 100 },
            animated: true,
        });
        // const coordinatesWithin5km = coordinatesWithinRadius(from, coordinates_2);
        // console.log('coordinatesWithin5km: ',coordinatesWithin5km);

    }
```

FIGURE 11. getCurrentPosition() Function

```
export const trip = state => state.trip;

const RideList = ({ navigation
  const tripList = useSelector(trip);
  const dispatch = useDispatch()
  return (
    <FlatList
      data={tripList.trips}
      renderItem={({ item }) => <Ride item={item} navigation={navigation} />}
      keyExtractor={item => item.id}
      contentContainerStyle={styles.container}
    />
  );
};

export default RideList;
```

FIGURE 12. useSelector() Fucntion Retrieves Data

6 RESULTS

This chapter will cover an overview of the final prototype of GoViet and describe all the functions, screens, and navigation. There are six mainly function-based parts which were built in GoViet and divided into screens correspondingly as described more profoundly below.

6.1 Onboarding Screen

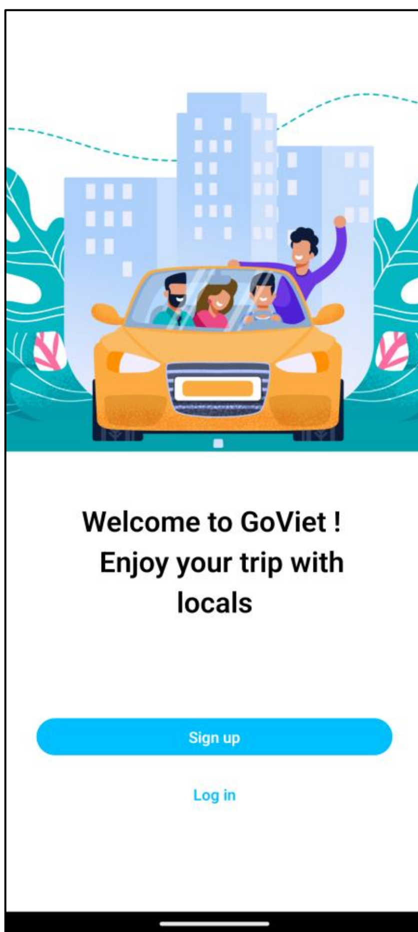

FIGURE 13. Onboarding Screen

Figure 13 has illustrated the welcoming screen for every user when they open the application. There will be a three-second loading time before this screen showing up to the end user, which contains two options that are Sign Up for a brand-new user or Log In for users who already created accounts in GoViet.

New user will be navigated to the Authentication Screens which will be described in the following chapter, whereas the other one will directly be taken to the Home Screen in chapter 6.4.

6.2 Authentication Screen

As mentioned above, only new user will be directed to these screens, hence, there are three selectable options as shown in Figure 14. Either of the options are mandatory to input the phone number later on, for a reason that the *signInWithPhoneNumber()* method was ultilized in Firebase Authentication for the account verification.

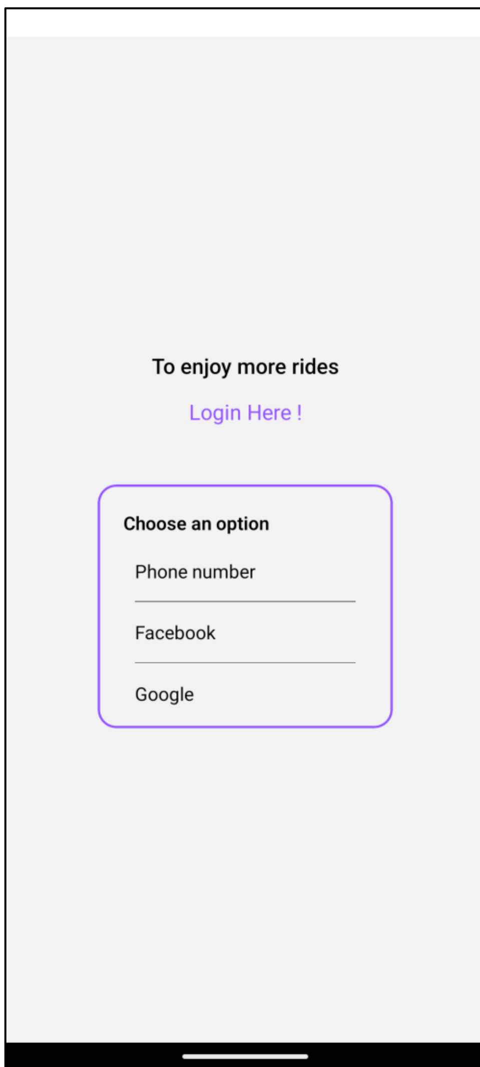

FIGURE 14. Sign-in Options

Moving on to the verification steps after user has chosen the option for signing in. Since the main purpose for GoViet creation is to support foreigners easily, safely, and efficiently to travel around Vietnam, as a result, one of the features was built providing this idea is the international selectable phone number code. Once finishing to input the phone number, a one-time-password (OTP) will be sent to the provided number. By inserting this code into the last step of authentication, user is successfully logged into the application with their own account. Figure 15 will describe steps in this session.

Additionally, apart from the first registration, users will skip these steps thanks to Firebase Authentication, which will define, record, and retrieve user log information if there is, or not, the user is logged in the system in order to save that activity for next time.

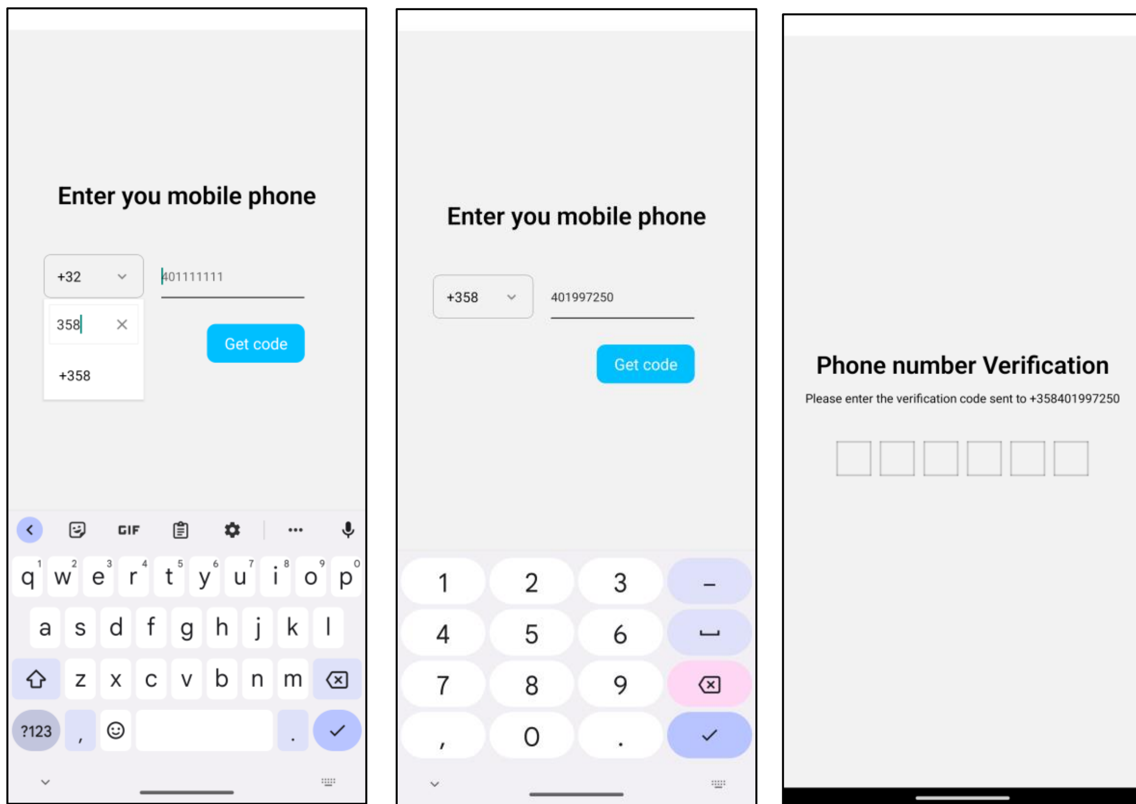

FIGURE 15. Phone Number Verification Steps

6.3 User Profile Screen

Figure 16 illustrates the interface for this screen. As of currently, it has been under development, however, it would have some fundamental functionalities for updating user profile, they are: personal information modifying, Travel preference during a trip, account and payment management, especially, if a Passenger wishes to become a Driver, they can upload all the related documents and information in this screen, and easily turn into the Driver who start publishing rides. All in one application and this feature is switchable.

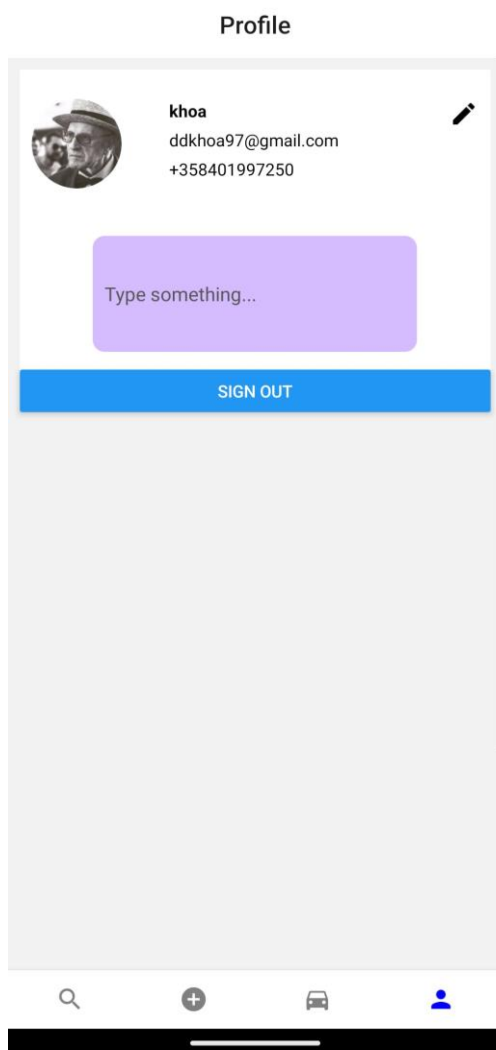

FIGURE 16. User Profile Screen

6.4 Home or Book a Ride Screen

As mentioned above, user who already has an account will directly be in this screen as Firebase Auth remembers the log in activity from the first time, similarly happens for new user with a newly created account.

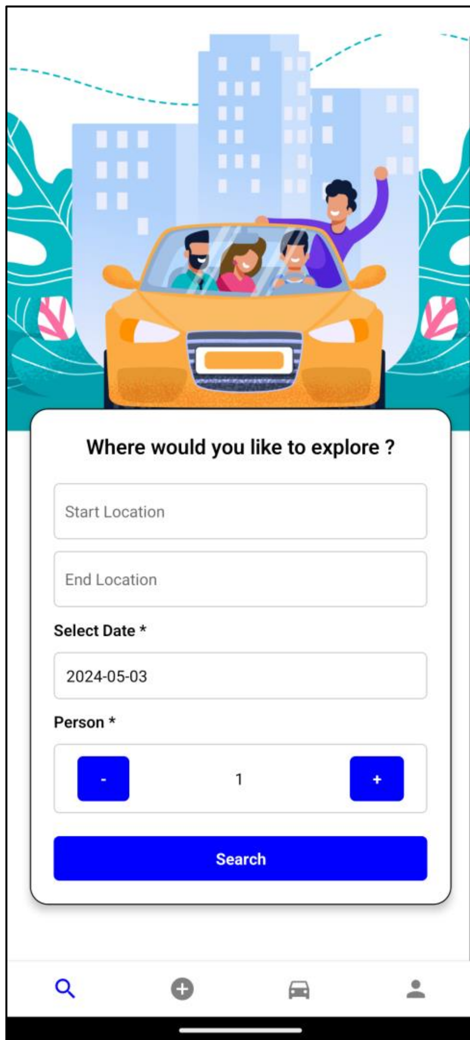

FIGURE 17. Home Screen

This Main Screen will provide users with finding and booking a ride base on their choices, such as pick-up and drop-off locations, number of seats as well as date and time for the ride. When everything is set from the Main Screen, the Search button will trigger a scrollable list of rides (Figure 18) because of the system received all the requirements and will start looking for the corresponding rides. Each option or so-called ride has the same information as route details, car details, driver details and the total price of the ride (Figure 18).

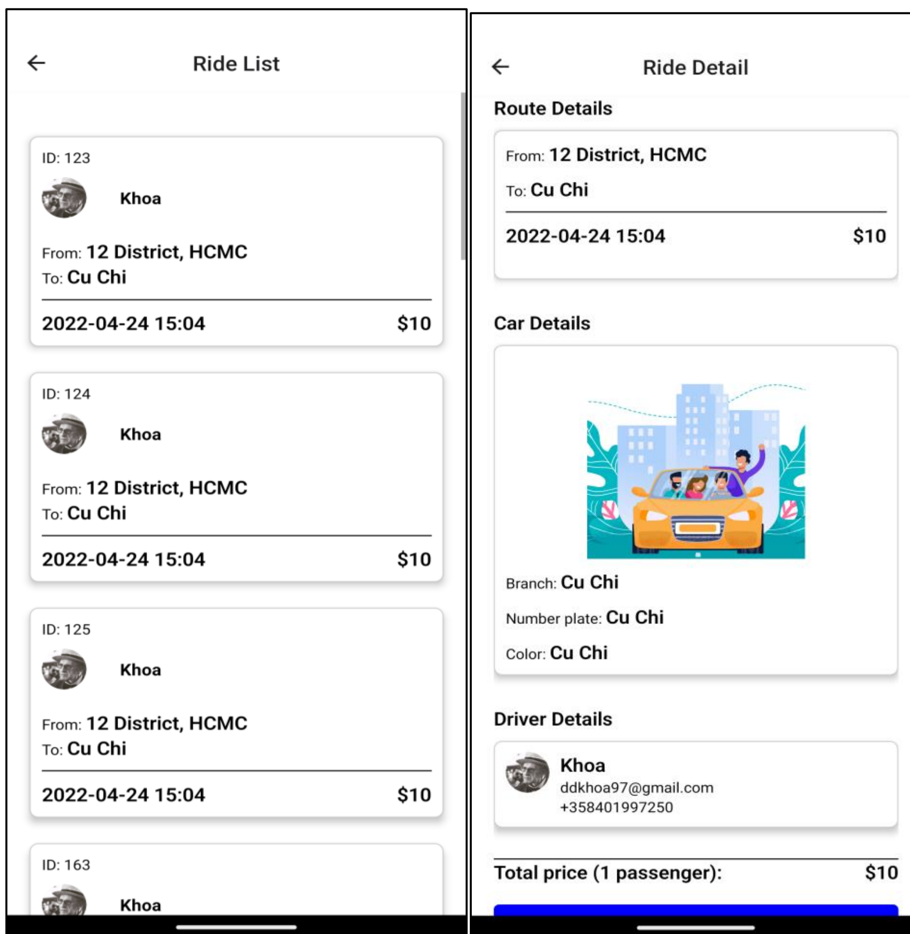

FIGURE 18. List of Rides (left) and Ride Details (right)

After choosing the best suitable ride, users will be led to payment screen in which he or she can decide the payment method for the ride. Currently, GoViet only supports paying by cash, however, this minor disadvantage will be taken as a good advantage to get more development on the side of the payment itself and the application in general. This is the last step in booking a ride, a Complete button will trigger a push notification pops up when users finish all the steps thus far announcing that their booking has been successfully completed. Figure 19 will show these two screens.

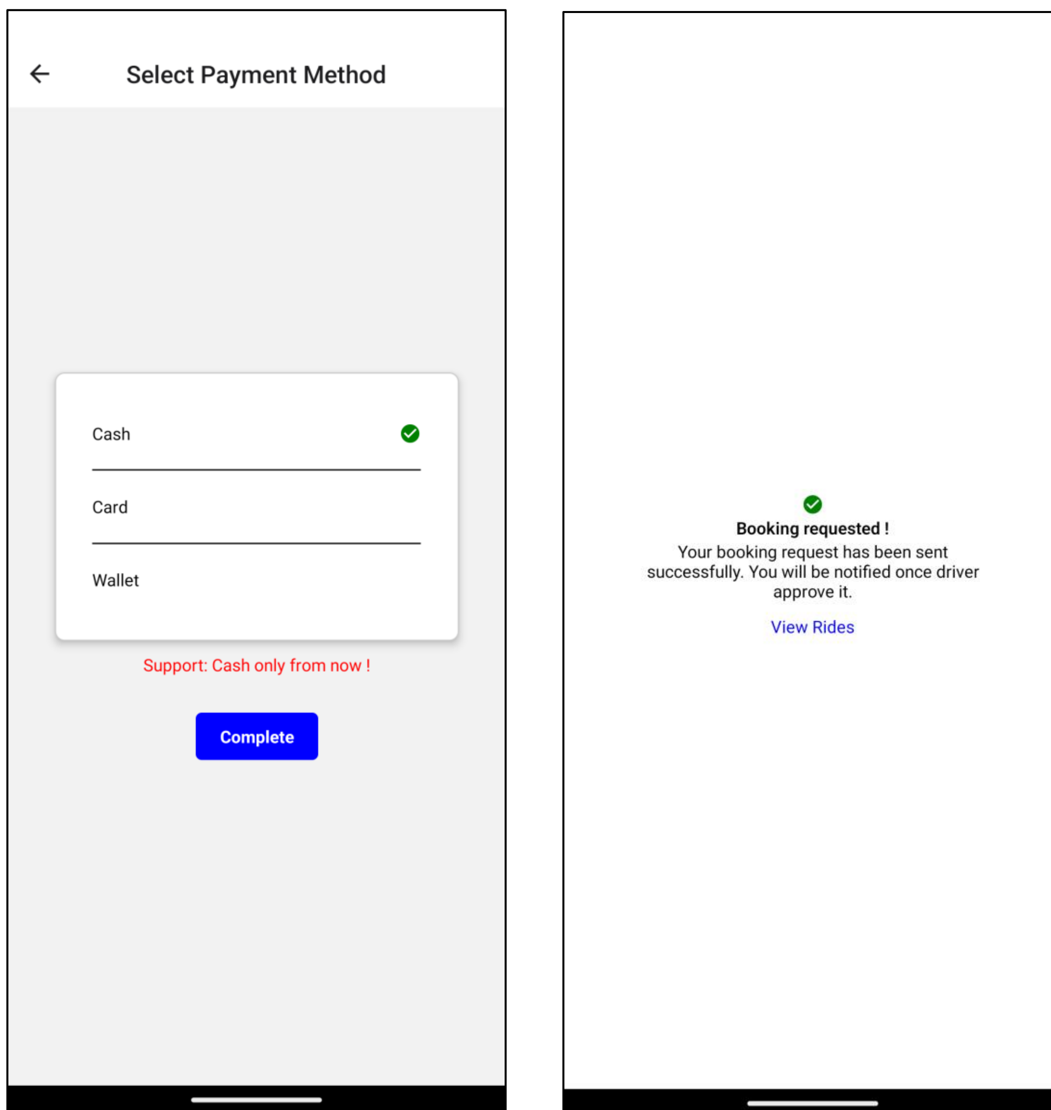

FIGURE 19. Payment Methods (left) and Push Notification (right)

There will be a small thread running in the background when the Complete button is pressed. The system will update the user ID into trips Collection – passengers Document in Firestore server. By doing so, the history of booking for that user can be easily traced back. Figures 20 and 21 will

describe the function illustrating the Payment process with user ID showed in the log as written "'id': 'r4XeYY…wq4G2' " as well as the updated information in the server. The code covers the function for server updating process in Appendix 9.

```
export default function Payment({navigation,route}) {
  const [selectedOption, setSelectedOption] = useState('Cash');
  const { data,item } = route.params;
  console.log(data)
  console.log(item.id)
  const handleOptionSelect = (option) => {
    setSelectedOption(option);
    // Perform any other actions based on the selected option
  };
  const closeModal = () => {
    setModalVisible(false);
  };
  const [modalVisible, setModalVisible] = useState(false);

  const BookTrip = () =>{
    updateTrip(data.id,item.id)
    setModalVisible(true)
  }
  return (
    <View style={styles.container}>
      <Modal
        animationType="slide"
        transparent={true}
        visible={modalVisible}
        onRequestClose={closeModal}
      >
        <View style={styles.modalContainer}>
          <View style={styles.modalContent}>
            <MaterialIcons name="check-circle" size={18} color="green" />
```

```
LEMS    OUTPUT    TERMINAL    AZURE    JUPYTER

ERMINAL

LOG  yqroVqydiXhaRH9UR2EA
LOG  {"email": "ddkhoa97@gmail.com", "id": "r4XeYYJUaTe8nBzYafKg0G3wq4G2", "name": "k
oa", "phone": "+358401997250"}
LOG  yqroVqydiXhaRH9UR2EA
LOG  {"email": "ddkhoa97@gmail.com", "id": "r4XeYYJUaTe8nBzYafKg0G3wq4G2", "name": "k
oa", "phone": "+358401997250"}
LOG  yqroVqydiXhaRH9UR2EA
LOG  {"email": "ddkhoa97@gmail.com", "id": "r4XeYYJUaTe8nBzYafKg0G3wq4G2", "name": "k
oa", "phone": "+358401997250"}
LOG  yqroVqydiXhaRH9UR2EA
LOG  {"email": "ddkhoa97@gmail.com", "id": "r4XeYYJUaTe8nBzYafKg0G3wq4G2", "name": "k
oa", "phone": "+358401997250"}
LOG  yqroVqydiXhaRH9UR2EA
```

FIGURE 20. Payment Method with User ID.

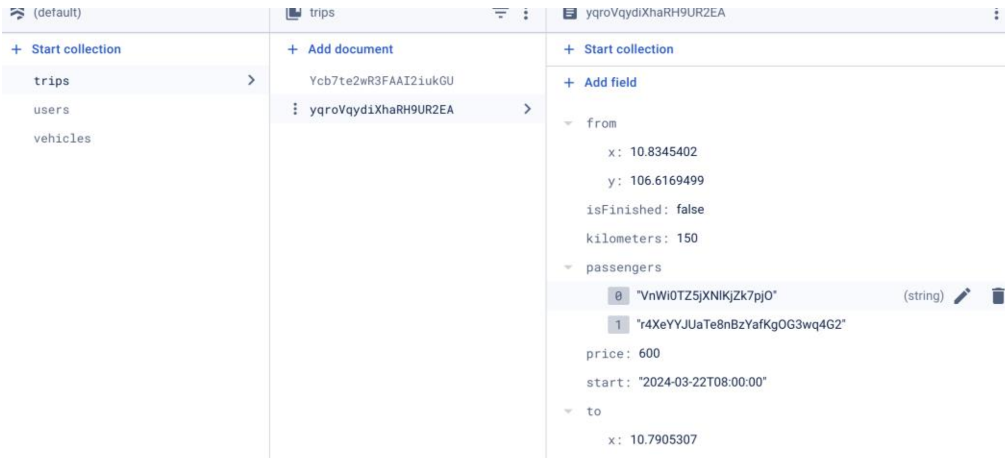

FIGURE 21. Firestore Server Is Updated with New User ID

6.5 My Rides Screens

As being a Driver in GoViet, users have two mainly technical activities which are: publishing rides and declining or accepting a booking request from Passenger. Figure 22 will display this screen in which published rides shown as a list. Sharing the same screen with Passenger but there is a small slidable bar distinguishing the Published Rides for Driver on the left side, with the Booked Rides for the Passenger which is the right picture.

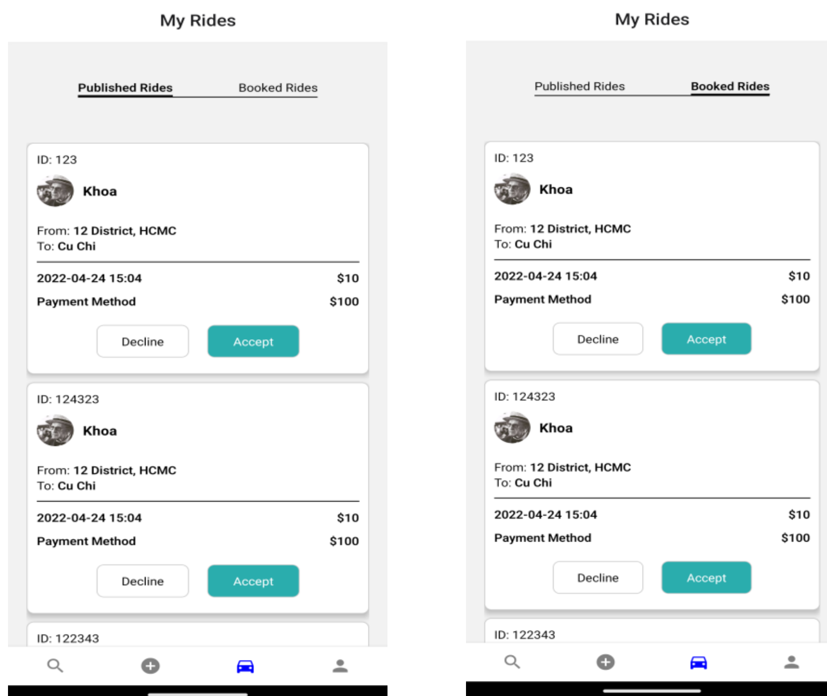

FIGURE 22. Driver's Published Rides (left) and Passenger's Booked Rides (right)

6.6 Publish a Ride Screen

As the name suggests, this screen is for a Driver to publish rides based on their availability. There are also options for Passenger to choose when they want to book a ride such as date and time, number of seats, price per person and the suggested price if necessary. Figure 23 will briefly show the output, in which the left side picture is GoViet's default screen initially for publishing a ride by choosing the start and stop points; the right-side picture is an overview of the location between points with a computed distance. Haversine formular is applied in the calculation, however, the output is not as expected and the code in Appendix 10 will cover it.

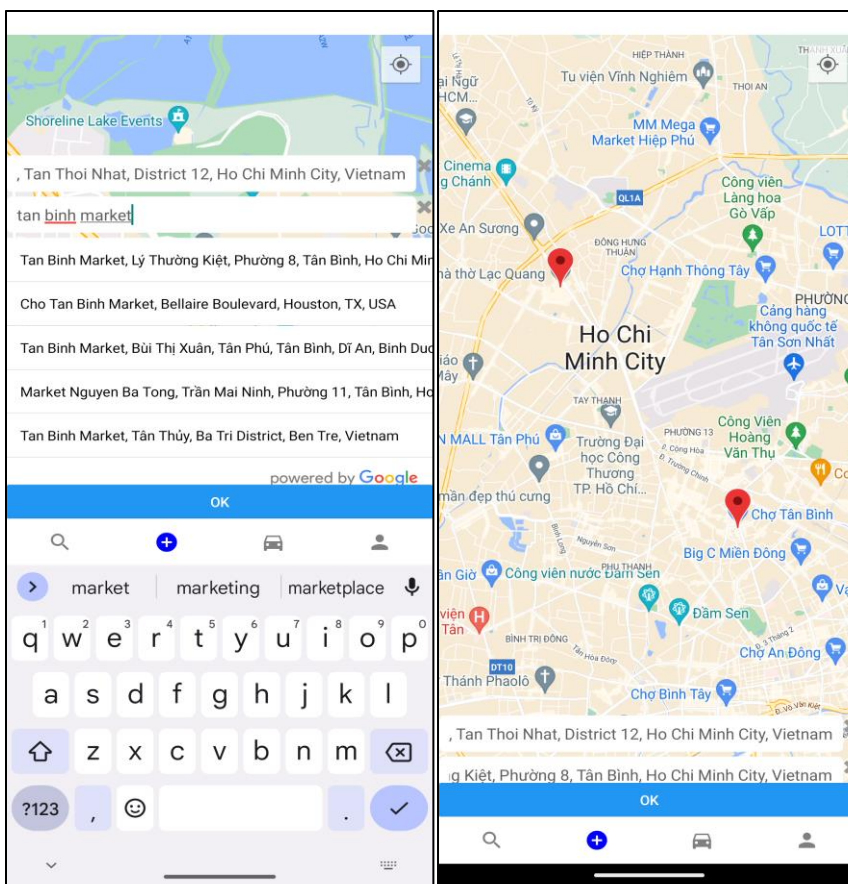

FIGURE 23. Default Screen (left) and An Overview of Two Location Points (right)

After finishing choosing the locations, Driver will be guided to other screens with a few more steps to finalize the process. Figure 24 will illustrate the screens as below.

FIGURE 24. Filling Missing Ride Details for Publishing

The Driver must fill in all the mandatory fields marked with the asterisk icon as those are the supportive details for a Passenger to know which ride they should go for. The last step before a ride is published is reviewing the trip detail one more time for the Driver to make sure that his ride is totally suitable for the market, the scheduling as well as checking any mistakes. After which, the Publish button will trigger a push notification announcing that the ride is successfully published and waiting to be booked by the Passengers.

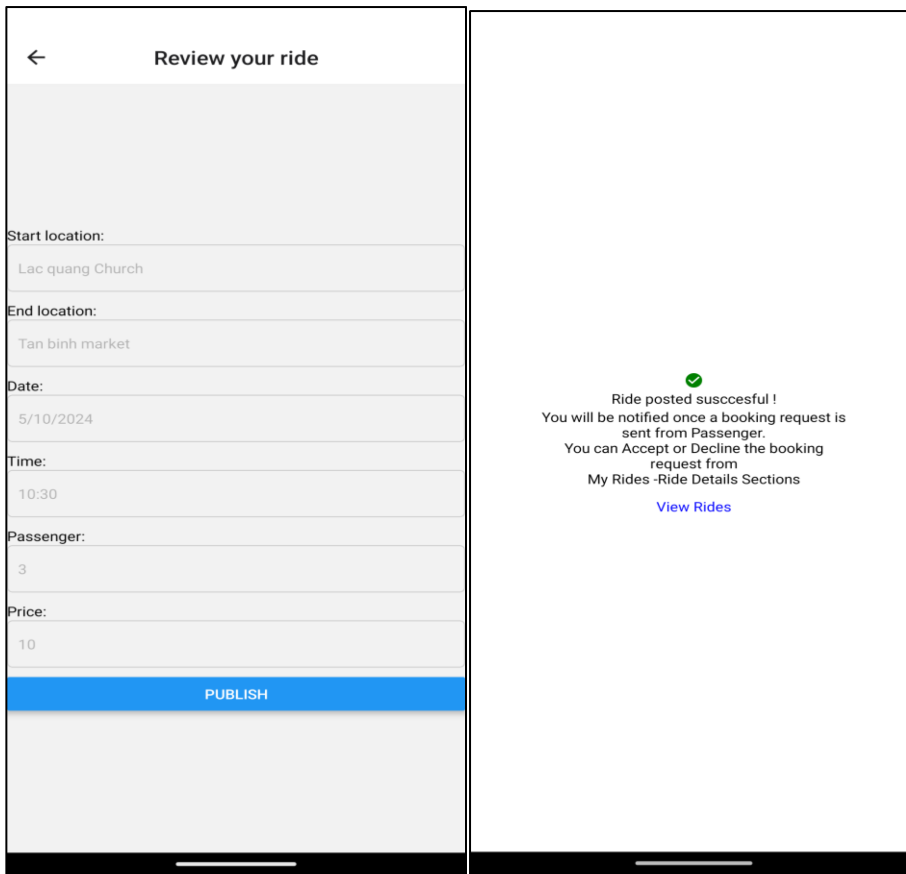

FIGURE 25. Review the Ride (left) and Push Notification (right)

In the background, some algorithms are running to update the collected data to the Firestore server when the action is completed. Similarly to authentication and user profile activities, Redux is also utilized to maintain, manipulate, and manage this action whenever a trigger is made. All the methods are contained in a *publishReducer()* function whose code will be presented in Appendix 11. By the time the ride is completely published, ride data is also recorded then reconstructed according to the preset formatting, then the *publishTrip()* function handling the push process will play its role which sends the data to database in Firestore server and has it updated. Figures 26, 27 and 28 will describe these steps.

```
26
27    const handlePublish = () => {
28        // Handle publishing the trip with the provided details
29        const {id} = userData.user
30        const {to,from,additional,vehicleID} = publishTripData
31        const dateTimeStr = additional.date + 'T' + additional.time;
32        const dateTime = new Date(dateTimeStr);
33
34        const formattedData= {
35            driverID:id,
36            passenger:[],
37            isFinished:fals  any
38            from: {x: from.latitude, y:from.longitude},
39            to: {x: to.latitude, y:to.longitude},
40            price: additional.price,
41            vehicle:vehicleID,
42            start:dateTime
43        }
44        console.log(formattedData)
45        publishTrip(formattedData)
46        navigation.navigate("MainScreen", { screen: "My Rides" });
47        // setModalVisible(true)
48    };
49    const closeModal = () => {
50        setModalVisible(false);
51    };
52
53    return (
54        <View style={styles.container}>
```

```
PROBLEMS   OUTPUT   TERMINAL   AZURE   JUPYTER
TERMINAL

Android Bundled 107ms (index.js)
 LOG  {"date": "2024-05-08", "price": "1000", "time": "00:00"}
 LOG  {"driverID": "r4XeYYJUaTe8nBzYafKgOG3wq4G2", "from": {"x": 10.8349959, "y": 10
6.6189173}, "isFinished": false, "passenger": [], "price": "1000", "to": {"x": 10.78
61046, "y": 106.653277}, "vehicle": "xsdFsaxS23SvcxWW123"}
 LOG  Document successfully updated!
› Reloading apps
Android Bundled 106ms (index.js)
 LOG  {"date": "2024-05-08", "price": "10", "time": "00:00"}
 LOG  {"driverID": "r4XeYYJUaTe8nBzYafKgOG3wq4G2", "from": {"x": 10.8349959, "y": 10
6.6189173}, "isFinished": false, "passenger": [], "price": "10", "start": 2024-05-07
T17:00:00.000Z, "to": {"x": 10.7861046, "y": 106.653277}, "vehicle": "xsdFsaxS23Svcx
WW123"}
 LOG  Document successfully updated!
```

FIGURE 26. Function for Formatting Ride Data Which is Showed in Log

```
export const publishTrip = (data) => {
    firestore()
        .collection(TRIP)
        .add(data)
        .then(() => console.log("Document successfully updated!")).
        catch((error) => console.error("Error updating document: ", error));
}
```

FIGURE 27. publishTrip() Function Handles Updating Data in Server

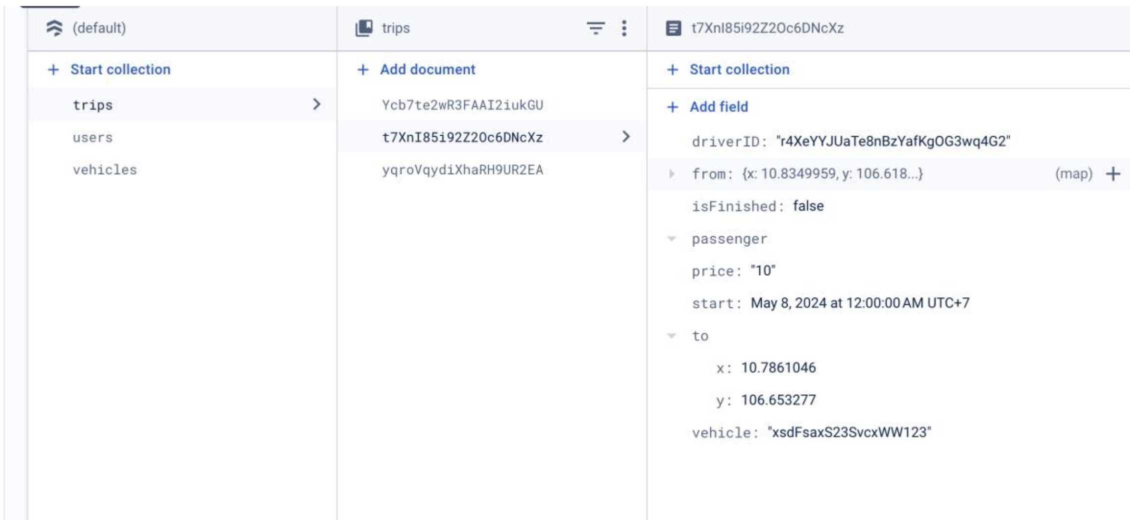

FIGURE 28. Firestore Server Updated with Ride Data

7 DISCUSSION AND SUMMARY

The aim of the final chapter is to address the objectives of the thesis in order to review the overall achievements of this project as mentioned in the first chapter. Therefore, discussions towards the outcome of the project will be inscribed from GoViet's developer's perspectives for future growth opportunity. Eventually, a summary will be made available according to the discussions and plans.

The overall outcome of the project was successful. All the functional features worked well as expected combining with the support of Firebase services, plus, the system flow has been a game-changing experience to the testers as it gave the end users smooth activities from the beginning.

The objective of the project was to build a multiplatform booking system with integrated map, selectable sign-in options, and payment methods, finding and booking a ride. The highlight of our application was the publish feature where a Passenger can be a Driver and be able to publish their own rides. Judging from the testing results, it can be considered that this goal was slightly achieved upon the completion of GoViet.

However, there are some features that have not been fully developed like the User Profile screen detail; payment methods should be expanded in the options as off right now it is limited to pay by cash only; also, we want to integrate more APIs from Google Maps Platform for tools to calculate the suggested price per ride (13). This would be a competitive advantage for our application in setting the margin price to the market without confusing the users.

In summary, the project has achieved its main purpose by providing a functional application that can be used in IOS or Android system. All the planned features have been working as well as expected to say the least, though a whole project has not been developed to the maximum potential. Moreover, the reasons behind are not only the timeline of the report, but also the difference in qualifications between developers. This whole project was a journey which was truly one of a kind for us to keep learning nonstop to gain the output as we planned. An experience worth remembering and a constant reminder to cultivate knowledge for the future.

REFERENCES

1. Skuza, Bartosz & Janiec, Jakub & Bartosińska, Inez 2024. Flutter vs React Native – Which is Better for Your Project in 2024?. Blog: Flutter Vs React Native Comparison. Search date 26.04.2024. https://www.thedroidsonroids.com/blog/flutter-vs-react-native-comparison.

2. Deshpande, Chinmayee 2024. Flutter vs. React Native: Which One to Choose in 2024?. Search date 26.04.2024. https://www.simplilearn.com/tutorials/reactjs-tutorial/flutter-vs-react-native.

3. Meta Platforms Inc 2024. Render, Commit, and Mount. Search date 26.04.2024. https://reactnative.dev/architecture/render-pipeline

4. Meta Platforms Inc 2024. React Fundamentals. Search date 26.04.2024. https://reactnative.dev/docs/intro-react?language=javascript#your-first-component

5. Kolodiy, Maks 2023. React Native Component Lifecycle. Blog: React Native Lifecycle. Search date 26.04.2024. https://www.netguru.com/blog/react-native-lifecycle

6. Meta Platforms Inc 2024. Style. Search date 26.04.2024. https://reactnative.dev/docs/style.

7. Meta Platforms Inc 2024. Core Components and APIs. Search date 27.04.2024. https://reactnative.dev/docs/components-and-apis

8. Abramov, Dan & The Redux documentation authors 2015-2024. Redux Essentials, Part 1: Redux Overview and Concepts. Search date 28.04.24 https://redux.js.org/tutorials/essentials/part-1-overview-concepts

9. S, Ashok Kumar 2018. Mastering Firebase for Android Development. Birmingham: Packt Publishing Ltd. Search date 30.04.2024. https://books.google.fi/books?id=RMNiD-wAAQBAJ&printsec=frontcover&dq=in-title%3AMastering%2BFire-base%2Bfor%2BAndroid%2BDevelopment%2B%3A%2BBuild%2BReal-Time%2C%2BScal-able%2C%2Band%2BCloud-ena-bled%2BAndroid%2BApps%2Bwith%2BFire-base&hl=en&sa=X&ved=0ahUKEwiziJbG-ZDhAhWE16YKHduYB34Q6AEIK-TAA#v=onepage&q&f=false

10. 0941176 B.C. Ltd 2024. Firebase Authentication. Search date 30.04.2024. https://firebase.google.com/docs/auth/

11. 0941176 B.C. Ltd 2024. Cloud Firestore. Search date 02.05.2024. https://firebase.google.com/docs/firestore.

12. 0941176 B.C. Ltd 2024. Maps JavaScript API. Search date 14.05.2024. https://developers.google.com/maps/documentation/javascript

13. 0941176 B.C. Ltd 2024. Distance Matrix Service. Search date 14.05.2024. https://developers.google.com/maps/documentation/javascript/distancematrix?hl=en&_gl=1%2A1vs1zgz%2A_ga%2AMTY1OD-kyMzAzMC4xNzE1NjkxNTU0%2A_ga_NRWSTWS78N%2AMTcxNTY5MTU1NC4xLjEuMTcxNTY5MTY0OC4wLjAuMA

APPENDICES

An Array of Data Appendix 1
Props of FlatList Appendix 2
Modal Syntax Appendix 3
Stack Navigation Appendix 4
userAction.js File Appendix 5
userReducer.js File Appendix 6
Phone Verification Method Appendix 7
authAction.js File Appendix 8
Function Updating Firestore Server Appendix 9
Haversine Formular Appendix 10
publishReducer() Function Appendix 11

An Array of Data APPENDIX 1

```
7     const data = [{
8       id: "123",
9       avatar: "",
10      from: "12 District, HCMC",
11      to: "Cu Chi",
12      date: "2022-04-24 15:04",
13      price: "10",
14      name: "Khoa"
15    },{
16        id: "124",
17        avatar: "",
18        from: "12 District, HCMC",
19        to: "Cu Chi",
20        date: "2022-04-24 15:04",
21        price: "10",
22        name: "Khoa"
23      },{
24        id: "125",
25        avatar: "",
26        from: "12 District, HCMC",
27        to: "Cu Chi",
28        date: "2022-04-24 15:04",
29        price: "10",
30        name: "Khoa"
31      },{
32        id: "163",
33        avatar: "",
34        from: "12 District, HCMC",
35        to: "Cu Chi",
36        date: "2022-04-24 15:04",
37        price: "10",
38        name: "Khoa"
```

```
187       return (
188         <FlatList
189           data={data}
190           renderItem={Ride}
191           keyExtractor={item => item.id}
192           contentContainerStyle={styles.container}
193         />
194       );
195     };
196
197     export default RideList;
```

Modal Syntax APPENDIX 3

```
9     const [modalVisible, setModalVisible] = useState(false);
10    const [count, setCount] = useState(1);
11    const [showCalendar, setShowCalendar] = useState(false);
12    const [selectedDate, setSelectedDate] = useState(new Date().toISOString().split('T')[0]);
13
14 >  const handleSubmit = () => { ...
25    };
26 >  const onDayPress = (day) => { ...
31    };
32    const [message, setMessage] = useState('');
33
34
35    const closeModal = () => {
36        setModalVisible(false);
37        setMessage('');
38      };
39 >  const increment = () => { ...
47      };
48
49 >    const decrement = () => { ...
57      };
58
59    return (
60        <ScrollView style={styles.container} automaticallyAdjustKeyboardInsets={true}>
61          <Modal
62        animationType="slide"
63        transparent={true}
64        visible={modalVisible}
65        onRequestClose={closeModal}
66      >
67        <View style={styles.modalContainer}>
68          <View style={styles.modalContent}>
69            <Text>{message}</Text>
70            <TouchableOpacity onPress={closeModal}>
71              <Text style={styles.closeButton}>Close</Text>
72            </TouchableOpacity>
73          </View>
74        </View>
75      </Modal>
```

Stack Navigation APPENDIX 4 (1/2)

```
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import WelcomeScreen from '../screens/WelcomeScreens/WelcomeScreen';
import LoginScreen from '../screens/Login/LoginScreen';
import PhoneNumber from '../screens/Login/PhoneNumber';
import VerifyScreen from '../screens/Login/VerifyScreen';
import CustomerInfo from '../screens/MainScreens/CustomerInfo';
import Profile from '../screens/MainScreens/Profile';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import BottomTabNavigator from './bottomNavigation';
import RideList from '../screens/MainScreens/RideList';
import TripDetail from '../screens/MainScreens/TripDetail';
import Payment from '../screens/MainScreens/Payment';

const Stack = createStackNavigator();
const Tab = createBottomTabNavigator();

const NavigationWrapper = () => {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="WelcomeScreen">
        {/* <Stack.Screen name="FirstScreen" component={FirstScreen} options={{ header-
Shown: false }} /> */}
        <Stack.Screen name="WelcomeScreen" component={WelcomeScreen} options={{
headerShown: false }} />
        <Stack.Screen name="Login" component={LoginScreen} options={{ headerShown: false
}} />
        <Stack.Screen name="Phone" component={PhoneNumber} options={{ headerShown:
false }} />
        <Stack.Screen name="VerifyStep" component={VerifyScreen} options={{ headerShown:
false }} />
```

```
        <Stack.Screen name="CustomerInfo" component={CustomerInfo} options={{ header-
Shown: false }} />
        {/* <Stack.Screen name="MainScreen" component={Profile} options={{ headerShown:
false }} /> */}
        <Stack.Screen name="MainScreen" component={BottomTabNavigator} options={{
headerShown: false }} />
        <Stack.Screen name="RideList" component={RideList} options={{ headerTitle: "Ride
List", headerTitleAlign: 'center' }} />
        <Stack.Screen name="TripDetail" component={TripDetail} options={{ headerTitle: "Ride
Detail", headerTitleAlign: 'center' }} />
        <Stack.Screen name="Payment" component={Payment} options={{ headerTitle: "Se-
lect Payment Method", headerTitleAlign: 'center' }} />

      </Stack.Navigator>
    </NavigationContainer>
  );
};

export default NavigationWrapper;
```

userAction.js File APPENDIX 5

```
1     import firestore from '@react-native-firebase/firestore';
2     export const updateUser = (user) => ({
3         type: 'UPDATE_USER',
4         payload: user,
5     });
6
7
8     export const updateUserInfo = (userData, navigation) => {
9         return async (dispatch) => {
10            try {
11                // // You can perform actions with the user's input here
12                firestore()
13                    .collection('users')
14                    .doc(userData.id) // Assuming userData contains the user's ID
15                    .set(userData)
16                    .then(() => {
17                        navigation.navigate("MainScreen", { screen: "Profile" });
18                        dispatch(updateUser(userData)); // Dispatch action after user update
19                    })
20                    .catch(error => {
21                        console.error('Error updating user: ', error);
22                    });
23            } catch (error) {
24                console.error('Error updating user: ', error);
25            }
26        };
27    };
```

userReducer.js File APPENDIX 6

```
1  const initialState = {
2      user: null,
3  };
4
5  const userReducer = (state = initialState, action) => {
6      switch (action.type) {
7          case 'UPDATE_USER':
8              return {
9                  ...state,
10                 user: action.payload,
11             };
12
13         default:
14             return state;
15     }
16 };
17
18 export default userReducer;
```

Phone Verification Method APPENDIX 7

```
….
import auth from '@react-native-firebase/auth';

import { RecaptchaVerifier, signInWithPhoneNumber } from "firebase/auth"
export default function PhoneNumber({ navigation }) {
  // If null, no SMS has been sent
  const [number, setNumber] = useState('');
  const [selectedValue, setSelectedValue] = useState('+84');
  const inputRef = useRef(null);

  // verification code (OTP - One-Time-Passcode)

  const onChangeText = (inputNumber) => {
    // Remove non-numeric characters using regular expression
    const cleanedText = inputNumber.replace(/[^0-9]/g, '');
    setNumber(cleanedText);
  };

  useEffect(() => {
    if (inputRef.current) {
      // Focus the TextInput after the component has rendered
      inputRef.current.focus();
    }
  }, []); // Run this effect only once after the initial render

  const GetVerifyCode = () => {

    navigation.navigate('VerifyStep', { phoneNumber: selectedValue + number })
  }
```

authActionjs File APPENDIX 8 (1/2)

```
import firestore from '@react-native-firebase/firestore';

export const loginSuccess = (user) => ({
  type: 'LOGIN_SUCCESS',
  payload: user,
});

export const loginFailure = (error) => ({
  type: 'LOGIN_FAILURE',
  payload: error,
});

export const logout = () => ({
  type: 'LOGOUT',
});

export const checkUserExistence = (uid, navigation) => {
  return async (dispatch) => {
    try {
      firestore()
        .collection('users')
        .doc(uid)
        .onSnapshot(documentSnapshot => {
          if (documentSnapshot.exists) {
            const userData = { id: documentSnapshot.id, ...documentSnapshot.data() };
            // If user exists, dispatch login success action or set some flag indicating the user is
logged in
            dispatch(loginSuccess(userData)); // Example action, you should define your action
accordingly
            dispatch(getTrips()); //Get the trip list from Firestore
```

authAction.js File APPENDIX 8 (2/2)

```
                    navigation.navigate("MainScreen", { screen: "Searching" });
                } else {
                    dispatch(loginSuccess({ id: documentSnapshot.id }));
                    navigation.navigate('CustomerInfo');
                    // If user does not exist, dispatch logout action or set some flag indicating the user
is logged out
                    // dispatch(logout()); // Example action, you should define your action accordingly
                }
            });
        } catch (error) {
            console.error('Error checking user existence: ', error);
            // Handle error, such as showing an error message to the user
        }
    };
};
```

Function Updating Firestore Server APPENDIX 9

```
import firestore from '@react-native-firebase/firestore';

const TRIP = 'trips'
export const updateTrip = (userID,tripID) =>{
        firestore()
            .collection(TRIP)
            .doc(tripID)
            .update({ 'passengers': firestore.FieldValue.arrayUnion(userID)  })
            .then(() => console.log("Document successfully updated!")).
            catch((error) => console.error("Error updating document: ", error));

}
```

Haversine Formular APPENDIX 10

```
const radius = 5; // Radius in kilometers

// Function to calculate distance between two coordinates using Haversine formula
function haversine(lat1, lon1, lat2, lon2) {
  const R = 6371; // Radius of the Earth in kilometers
  const dLat = toRad(lat2 - lat1);
  const dLon = toRad(lon2 - lon1);
  const a =
    Math.sin(dLat / 2) * Math.sin(dLat / 2) +
    Math.cos(toRad(lat1)) * Math.cos(toRad(lat2)) *
    Math.sin(dLon / 2) * Math.sin(dLon / 2);
  const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
  const distance = R * c; // Distance in kilometers
  return distance;
}

// Function to convert degrees to radians
function toRad(degrees) {
  return degrees * Math.PI / 180;
}

// Function to filter coordinates within a given radius
export function coordinatesWithinRadius(marker, coordinates) {
  const result = [];
  coordinates.forEach(coord => {
    const distance = haversine(marker.latitude, marker.longitude, coord.latitude, coord.longitude);
    if (distance <= radius) {
      result.push(coord);
    }
  });
  return result;
}
```

publishReducer() Function APPENDIX 11

```
import { SET_TO_LOCATION, SET_FROM_LOCATION, SET_ADDITIONAL_DATA } from '../types';

const initialState = {
    from: null,
    to: null,
    additional: null,
    isFinished: false,
    passengers:[]
};

const publishReducer = (state = initialState, action) => {
    switch (action.type) {
        case SET_TO_LOCATION:
            return {
                ...state,
                to: action.payload,
            };
        case SET_FROM_LOCATION:
            return {
                ...state,
                from: action.payload,
            };
        case SET_ADDITIONAL_DATA:
            return {
                ...state,
                additional: action.payload,
            };
        default:
            return state;
    }
};
```