WIA2007 MOBILE APPLICATION DEVELOPMENT

SEMESTER 1 2022/2023

ASSIGNMENT PART II

PROJECT REPORT

LECTURER : DR. ONG SIM YING

TUTORIAL GROUP : OCC 3

GROUP NAME : MADisOne

No.	Name	Matric No.
1.	Surin A/L Pubalan	U2001368
2.	Awangku Aniq Hamiz Bin Awangku Badaruddin	U2001358
3.	Samiha Tasnim Dristy	S2112287
4.	Muhammad Muqri Qawiem Bin Hanizam	U2000726

Table Of Contents

1.0 Introduction

Our mobile app intends to solve an issue that the Malaysian electric scooter rental market is currently experiencing. Electric scooter servicing and repair are not currently offered by any businesses, despite the fact that several organizations provide electric scooter rental services. To close this gap, we created an app that makes it simple for users of electric scooters to contact mechanics for scooter repairs.

The app provides a range of services, including brake maintenance, tyre care, and battery servicing. The creation of a user-friendly experience was the main goal of the development process. In order to accomplish this, the app has a pick-up and drop-off feature that enables users to choose the place where their scooters should be picked up and returned after repair.

With this mobile application, we have three goals in mind. We first want to make users' lives easier by saving them time and effort by letting them schedule scooter maintenance through the app. By offering a forum for workshops and technicians to market their services for maintaining electric scooters, we also intend to contribute to the local economy. By offering the user an anticipated repair cost, we hope to provide high-quality repairs at a fair price.

Attached below is a video link of our mobile application demonstration.
Video link :
https://drive.google.com/file/d/16LqXX3RLSWrDkDCNEVYRCVolW7dhj04s/view?usp=sharing

2.0 Application functionalities

When we first proposed the modules and functions of our application, we did not know about the issues we were about to face as development continues. Because of this, some of our modules and functions were scrapped and replaced with better and more logical functions and modules.

You can clearly see this in our table and see the difference between our Assignment Part 1 table and Assignment Part 2 table.

Assignment Part I		
Proposed Module	**Functions**	**Status**
General	☑ Login	Complete
	☑ Logout	
	☑ Registration	
Managing user profile	☑ User profile	Complete
	☑ Change password	Complete
	☑ Select type of scooter	Complete
	☐ Add location	Incomplete -The function does not seem to add usefulness to the app and would just waste time
Managing mechanic profile	☑ Mechanic profile	Completed
	☑ Change password	Completed
	☐ Shop location	Incomplete -Feature did not seem helpful and would encourage users to not use our app and just visit the mechanic directly
	☑ Mechanic's rate	Completed
Cost repairing services	☐ Calculate cost estimation	Incomplete -Hard to implement this feature as we do not know what to base the fees on

	☑ Managing transaction	Completed
Delivery	☑ Pickup & dropoff	Completed
SOS	☐ Choose emergency service	Incomplete -Useless feature as why would someone open the SCOOT app when they need emergency services instead of opening and dialing the number themselves
Details : ☑ Accomplished ☐ Unaccomplished		

Assignment Part II	
New Module	**Functions**
Booking	☑ Add information
	☑ Add location
	☑ Place booking
View booking activities	☑ View status
	☑ View price
	☑ Pay
E-wallet	☑ Top-up
	☑ Transaction history
Mechanic view request	☑ View requests
	☑ Accept requests
	☑ Reject requests
Mechanic view orders	☑ View orders

	☑ Update status
	☑ Give price
	☑ Auto-update
Mechanic e-wallet	☑ View transactions
	☑ Bank out

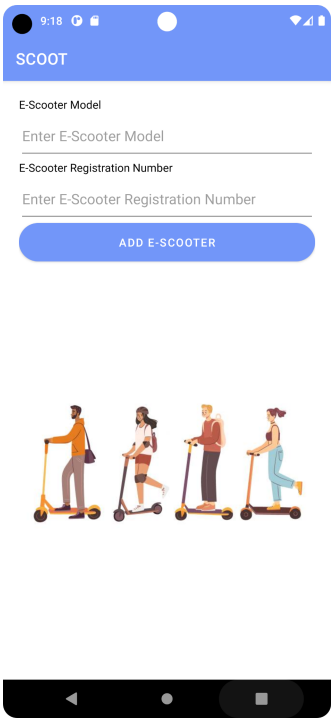
2.1 New modules detailed descriptions

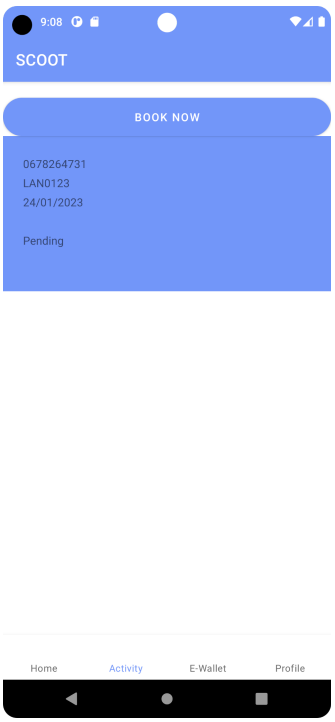
No.	Module	Function	Function brief description
1.	Booking	Add information	To provide users with crucial information that is needed for delivery and repairing their e-scooter.
		Add location	To enable users to enter their preferred drop-off and pick-up address.
		Place booking	To enable user to place a booking with all the added details towards mechanics to fix their broken e-scooter
2.	View booking activities	View status	To let users see the status of their past bookings whether it is still pending., mechanic fixing or done.
		View price	To let users see the quoted by the mechanic after their e-scooter is fixed
		Pay	To pay the mechanic after they fixed the user's e-scooter through our app's e-wallet
3.	E-wallet	Top-up	To allow users add money into their e-wallet which will be used to pay for the mechanic's work

			after they have fixed the e-scooter
		Transaction history	To view previous transactions is a list for users to see who they paid to and the amount paid.
4.	Mechanic view requests	View requests	To allow mechanics to see bookings made by users and the information regarding their name, phone number and model of their e-scooter which has to be fixed.
		Accept requests	To enable mechanics to accept the booking made by users and add them to a mechanic;s unique order list.
		Reject requests	To enable mechanics to reject the booking made by users and remove them from their request list view.
5.	Mechanic view orders	View orders	To allow mechanics see their list of orders to be done and the information given by the users regarding their details and e-scooter.
		Update status	To change the status of order from ‘fixing’ to ‘done’ and updating the status for users to see about their e-scooter which are being fixed.
		Give price	To let mechanics quote the price needed to fix e-scooter and remind the user about the price quoted.
		Auto-update	To update the list of done orders by detecting if the customer has paid for the order or not.
6.	Mechanic e-wallet	View transactions	To view previous transactions in a list for mechanics to see how

			much they earned and who they received it from
		Bank out	To allow mechanics to bank out money from their e-wallet and receive it in their real bank.

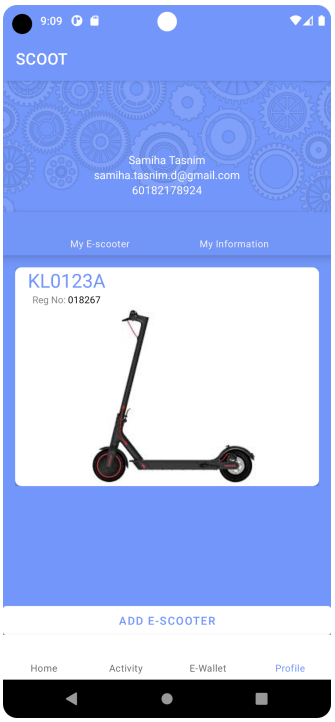
3.0 Screenshot of all the activity / pages with descriptions

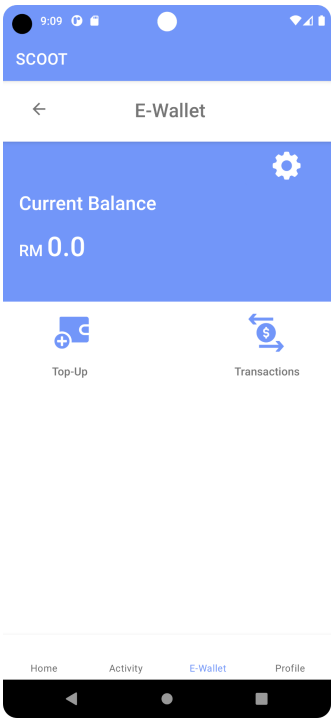
Screenshot	XML layout file	Java file
Add E-Scooter 	The layout is defined using a LinearLayout container with a vertical orientation. The layout contains 2 TextViews to display the text “E-Scooter Model” and “E-Scooter Registration Number." It also contains two editable texts to get user input on the registration number and model of the scooter. Has one button that is used to submit the scooter's information and an ImageView with the ID escooter that shows an image of a scooter.	This activity is responsible for adding a new electric scooter to our Firebase database and it is using the Firebase Auth for user authentication. In the onCreate() method, the activity sets its layout to R.layout.activity_add_escooter and gets a reference to the Firebase Database and the Firebase Auth instances. When the button is clicked, it creates an instance of EScooter with the model and registration number entered by the user. It generates and uses unique ID for the scooter and sets the EScooter instance to the corresponding child node in the firebase database using setValue(). Finally, it displays a Toast message to confirm that the record has been inserted.
Booking Detail List	booking_detail_list.xml is an XML layout file that defines the layout of each item in the ListView for the BookingDetailList activity. The layout is defined using a LinearLayout container with a horizontal orientation. The layout contains two main UI elements: - A LinearLayout with a vertical orientation which contains multiple TextView elements that will display the booking details, such as phone number, e-	BookingDetailList.java is an custom adapter class that extends the ArrayAdapter<BookingDetails> class used to display a list of BookingDetails objects in a ListView. It displays the phone number, escooter model, booking date, mechanic name, status, price and a button to pay the order. The list view is populated with the data from the BookingDetails object and also provides a

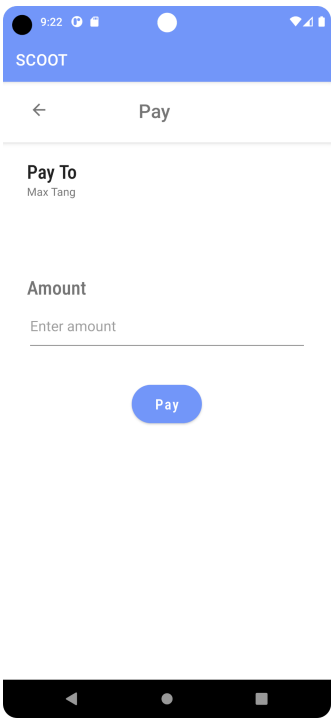
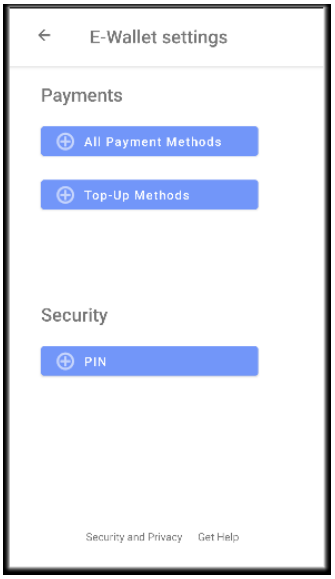
	scooter model, start date, mechanic name, status, price. - An ImageButton with the ID BtnPayOrder that is used to pay the order and the default visibility is gone.	way for the user to pay for their bookings by clicking on the pay order button which will redirect the user to EWalletPay activity.
Booking Details		BookingDetails.java is an activity that defines a BookingDetails class that implements the Serializable interface. The class has a number of attributes such as date, time, pick-up and drop-off addresses and phone numbers, e-scooter model and name, keys, and boolean values for if the booking has a mechanic, if the mechanic is done, and if payment has been made. The class also has methods to get and set the values of these attributes. The class also has a static method called toOrder which convert Booking details to Order.
Create E-Wallet	The XML file "activity_create_ewallet.xml" is a layout file for the "CreateEWallet" activity in the android app. It	CreateEWallet.java is an activity for creating an e-wallet for the user. When the user clicks on the

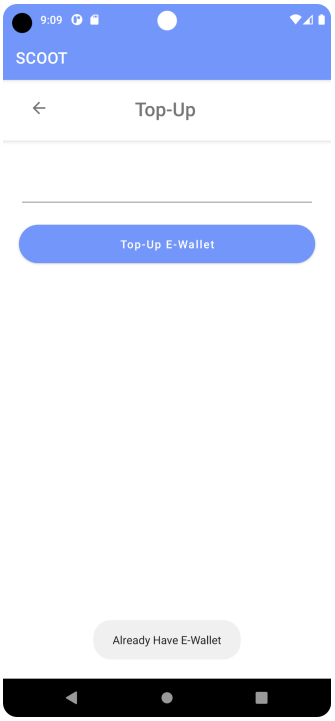
	contains a button with the id "BtnCreateEWallet" which when clicked, creates an E-Wallet account, and an ImageView which displays an image. It also uses ConstraintLayout to align the button and image view to the bottom, top and center of the parent layout.	"Create E-Wallet" button, it creates a new instance of the "Money" class and sets it as a value in the Firebase realtime database under the E-Wallet node with the user's ID as its child. Once the e-wallet is created, the user is taken to the EWallet activity and if the user already has an e-wallet, they are taken straight to the EWallet activity and a toast message is displayed. In the onStart() method, it checks whether the user already has an e-wallet or not.
Create E-Wallet Mechanic 	activity_create_ewallet_mechanic.xml is an android layout file which contains a button with the id of BtnCreateEWalletMechanic and an ImageView. The button is used to create an e-wallet and the ImageView is used to display an image. The layout is arranged using Constraint Layout and the button is positioned at the center of the screen. The ImageView is positioned below the button and takes up the whole screen. The image used in the ImageView is the one that is specified in the srcCompat attribute.	"CreateEWalletMechanic.java" is an activity class in Android. It creates an E-wallet for the user, using Firebase's authentication and database services. When the activity is launched, it sets the layout to "activity_create_ewallet_mechanic.xml". The class contains a button "BtnCreateEWalletMechanic" which creates an E-wallet account for the mechanic user. When the button is clicked, the activity creates a new instance of 'Money' class, and sets it's value to the E-wallet of the mechanic user. The activity then starts the "MechanicEWallet" activity and finishes itself. Also, the activity checks if the user already has an E-wallet, in such case, it takes the user to the "MechanicEWallet" activity

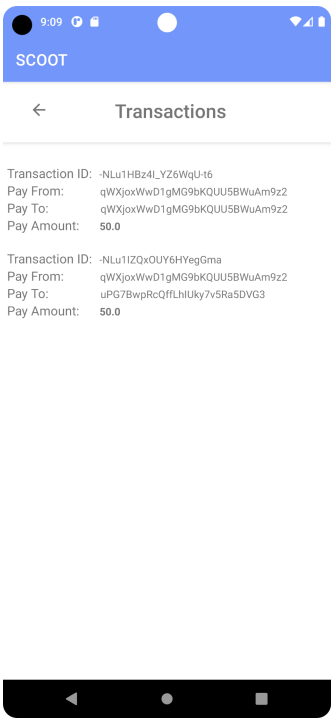
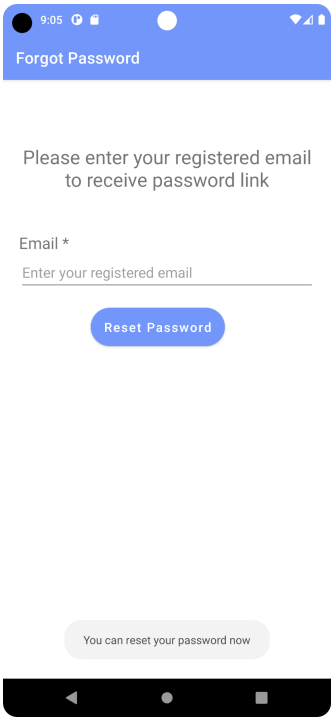
		and finishes itself.
DAO Booking Details		"DAOBookingDetails.java" is a class that interacts with Firebase to store and retrieve data related to bookings. It creates a reference to the "BookingDetails" node in the Firebase database and also another reference to the "Registered Users" node in the Firebase Database. The class contains methods to add and remove booking details to the database. The add method takes an instance of the "BookingDetails" class, generates a unique key for the booking, and stores the booking details under the unique key. The remove method takes the user's unique ID and the key of the booking as parameters and removes the corresponding booking from the database.
DAO Order		"DAOOrder.java" is a class that is used to add Order data to a Firebase Database. The class creates an instance of Firebase Authentication and Database Reference. The class has a method called 'add' which takes an 'Order' object as a parameter. Inside the 'add' method, it gets the current user's UID, generates a new key for the order, sets the order's status to false, and sets the order's key to the generated key. Finally, it pushes the order

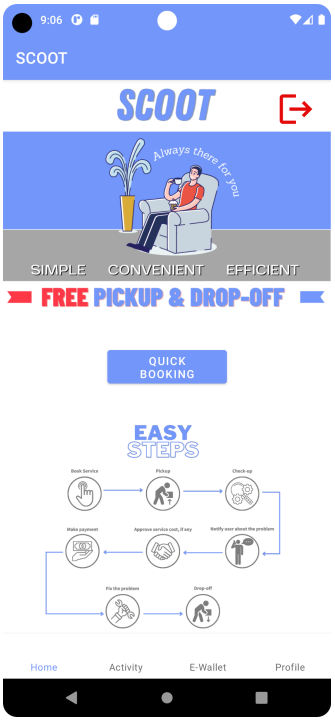
		to the database under the child of the current user's UID and the generated key.
E-Scooter		The EScooter class is a simple Java class that defines an e-scooter object. It has three properties: ScooterKey, model, and RegNo. The class also has getters and setters for each of these properties. The class has a constructor that takes two parameters, model and regNo, which are used to initialize the object's properties. The class also has a default constructor, which creates an empty object. This class will be used to store information about e-scooters.
E-Scooter List  	The file "e_scooter_list.xml" is an Android XML layout file that defines the layout and appearance of a list item for displaying e-scooter information. The layout is a CardView, which is a type of container that provides a box-like structure with rounded corners and an elevation, giving it a 3D look. Inside the CardView, there is a LinearLayout with a vertical orientation that holds the views which include, a text view for displaying scooter model, another text view for displaying scooter registration number, and an ImageView for displaying an image of a scooter. The layout also uses some other attributes such as layout_width, layout_height, layout_margin, and textSize to specify the	"EScooterList.java" is an ArrayAdapter class that is used to display a list of e-scooters on the user interface. It takes in an Activity context and a List of EScooter objects as input. It overrides the getView method, which is used to create a view for each item in the list. The getView method inflates a layout, "e_scooter_list.xml", populates the layout with the values from the EScooter object at the current position in the list and returns the view. The class also has two TextViews, txt_ProfileScooterModel and txt_ProfileScooterRegistration, that are used to display the model and registration

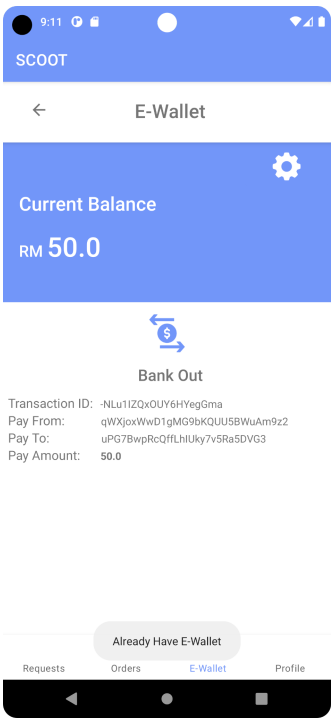
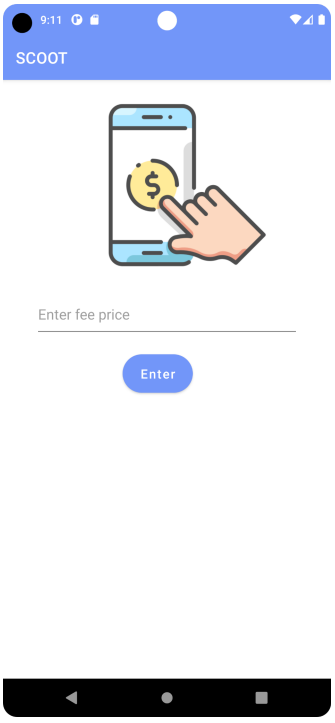
	dimensions and appearance of the views.	number of the e-scooter, respectively.
E-Wallet 	activity_ewallet.xml is an android layout file for the EWallet activity. It defines the layout for the E-Wallet feature in the mobile application. It contains elements such as a LinearLayout, TextView, ImageButton and others. It is used to design the user interface for the E-Wallet feature, which includes displaying the user's balance, top-up button, pay button, transaction button, and e-wallet setting button. The layout is designed to be responsive and user-friendly.	EWallet.java is an activity file that defines the behavior of an E-wallet feature in our Android app. The file contains methods to display the current balance of an E-wallet, top up the E-wallet, pay using the E-wallet, view transactions and settings of the E-wallet. The file also contains a navigation bar at the bottom of the activity to navigate to other activities such as profile, home, activities and E-wallet. The current balance of the E-wallet is obtained from a Firebase database and is displayed on the activity. The file also contains onClickListeners for the various buttons on the activity such as topup, pay and transaction buttons.
E-Wallet Pay	activity_ewallet_pay.xml is an android layout file that defines the structure and appearance of the UI elements in the EWalletPay activity. It includes several views such as AppBarLayout, ImageButton, TextView, EditText, LinearLayout, ConstraintLayout etc. All of these UI elements will be rendered on the EWalletPay activity, and the user can interact with them. The layout includes a back button, text fields for displaying payment information and a button for confirming the payment. The layout uses a number of layout constraints to specify how the	"EWalletPay.java" is a Java class for an activity in our Android app that handles the functionality for paying for a scooter rental through an e-wallet feature. The class includes code for displaying the name of the mechanic to be paid, an EditText field for the user to input the amount they want to pay, and a button to submit the payment. The class also includes code for updating the user's and mechanic's e-wallet balance in a Firebase database and creating a transaction record

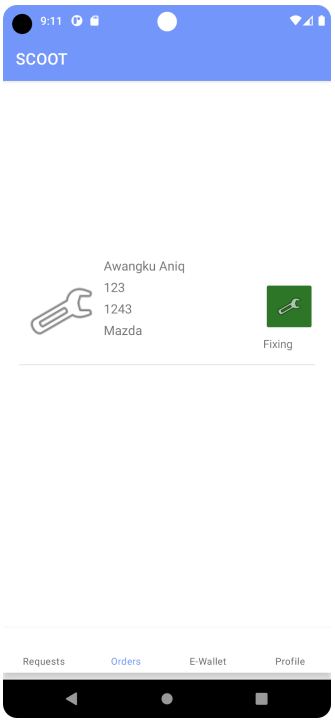
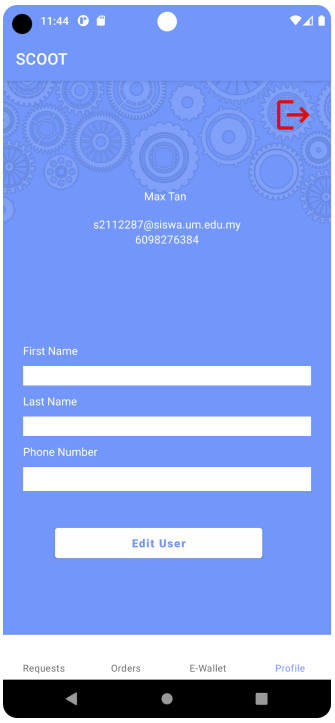
	elements should be positioned and sized on the screen.	in the same database after a successful payment.
E-Wallet Setting 	"activity_ewallet_setting.xml" is an XML layout file for an activity in our Android app which defines the layout for an E-Wallet settings page. The layout includes an AppBarLayout at the top, which contains an ImageButton for navigating back to the E-Wallet page and a TextView displaying the title "E-Wallet settings". The layout also includes a NestedScrollView which contains a ConstraintLayout, and within that, a LinearLayout that holds a TextView and a Button. The TextView displays "Payments" and the button displays "All Payment Method". The layout also contains a few other buttons and textviews for different settings and options. The purpose of this layout is to create the UI for the E-Wallet settings page in the app where users can	"EWalletSetting.java" is an Android activity class that sets up the user interface for the E-Wallet settings page in an app. It sets up the buttons for Payment Methods, Top-up Methods, Pin E-Wallet and Back To E-Wallet. When the Back to E-Wallet button is clicked, it creates a new intent to go back to the E-Wallet menu and starts the activity. It also uses the layout file "activity_ewallet_setting.xml " to inflate the UI for this activity.

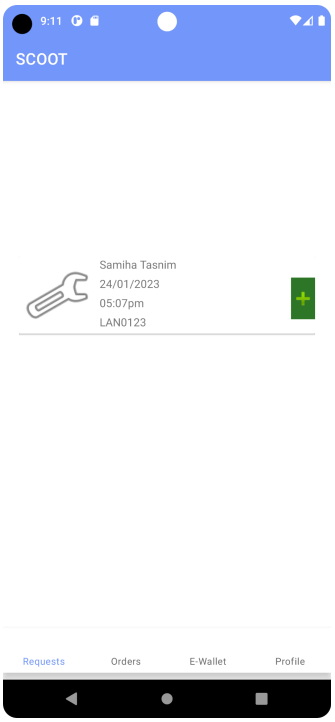
	manage their E-Wallet settings.	
E-Wallet Top Up 	activity_ewallet_topup.xml is an XML layout file for the EWalletTopup activity in the android project. It defines the layout of the user interface for the E-Wallet top-up feature. It contains a back button, a title, an edit text for the user to enter the top-up amount, and a button to confirm the top-up.	"EWalletTopup.java" is a Java file for our Android app that handles the functionality for a user to top-up their e-wallet. It includes buttons and EditText fields to input top-up amount, and uses Firebase to update the user's balance in the database. The file includes a listener that listens for changes in the user's e-wallet balance and updates it with the top-up amount. It also includes a listener to go back to the e-wallet menu when the back button is clicked. Additionally it updates the transaction history with the transaction details for the topup.
E-Wallet Transactions	"activity_ewallet_transactions.xml" is an XML layout file for our Android app that displays the user's e-wallet transactions. It uses a CoordinatorLayout as the top-level layout and includes an AppBarLayout for the app bar and a ListView for displaying the transactions. The layout includes an ImageButton for navigating back to the e-wallet menu, a TextView for displaying the title of the page, and a ListView for displaying the transactions.	"EWalletTransactions.java" is a Java file that represents the transactions activity of an e-wallet app. It uses Firebase Authentication to get the current user's unique ID and uses it to reference the "Transaction" node in the Firebase Database. The activity has a ListView that displays a list of transactions stored in the database. It uses a ValueEventListener to listen for changes in the database and updates the list accordingly. It also has a back button that takes the user back to the e-wallet menu when clicked.

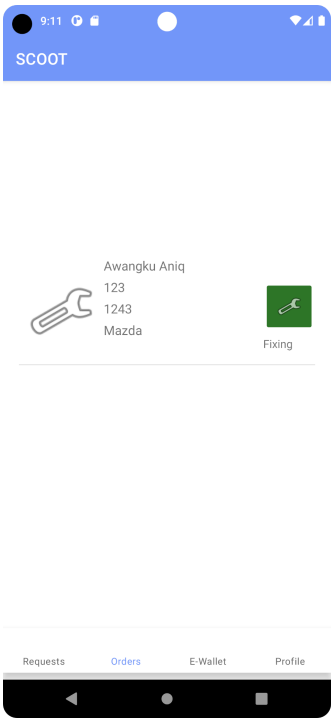
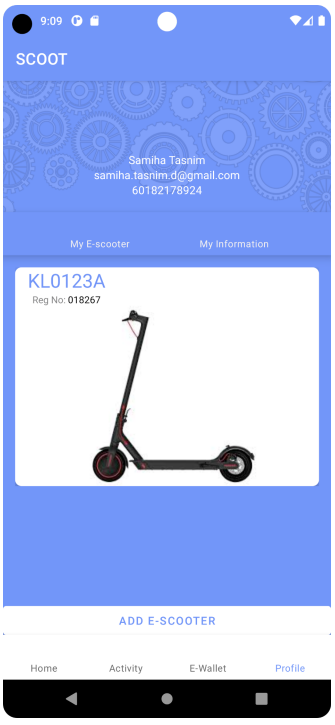
		
Forgot Password 	"activity_forgot_password.xml" is an xml file that sets the layout for the "ForgotPassword.java" activity. It contains several layout elements such as TextView, EditText and a Button, which are used to display text, take user input and trigger an action respectively. It sets the layout to be a vertical LinearLayout and sets the width and height to match the parent. The layout includes a TextView that explains the purpose of the activity, an EditText for the user to enter their email, a TextView labeled 'Email' and a button labeled "Reset Password" which triggers the password reset action when clicked.	The ForgotPassword.java file is an activity file for our Android app that handles the process of resetting a user's password. It includes a layout file, activity_forgot_password, which contains a button for resetting the password and an edit text field for the user to enter their email address. When the reset password button is clicked, the code checks if the email field is empty or not a valid email address. If it is, it displays an error message. If it is not, it calls the resetPassword() method, which sends a password reset email to the provided email address using FirebaseAuth. If the password reset email is sent successfully, the user is directed to the MainActivity

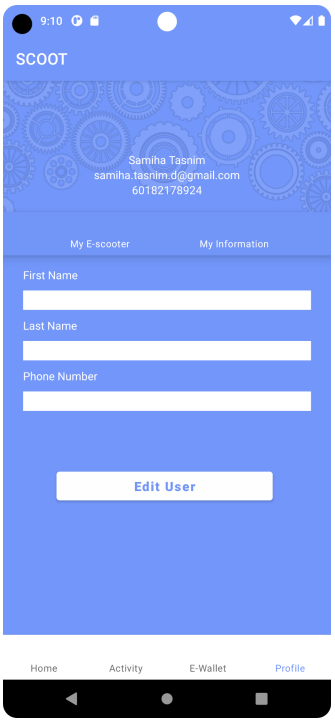
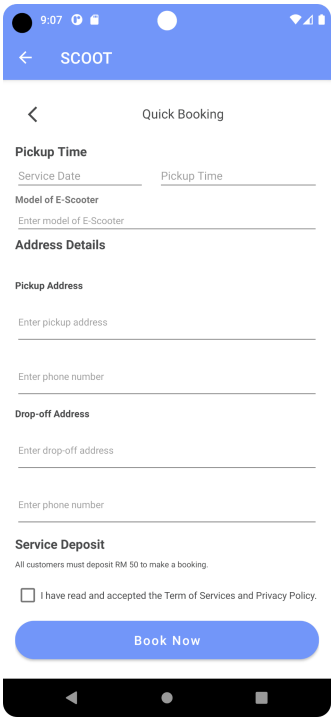
		and if not, an error message is displayed. The app also handles the case when the provided email address is not associated with any user.
Main Activity 	activity_main.xml is an XML layout file that defines the visual structure of the MainActivity in the Scoot app. It is made up of a number of views, including a BottomNavigationView, multiple ImageViews, and a Button. The BottomNavigationView is used to navigate between different parts of the app, the ImageViews are used to display images, and the Button is used for quick booking feature. The layout uses ConstraintLayout, which allows for precise positioning of elements on the screen. The layout also includes a logout button.	"MainActivity.java" is a Java file that contains the code for the MainActivity of our Android app named "Scoot". This activity serves as the main home screen of the app, where users can access different features of the app through a bottom navigation bar. The activity contains code for a button that starts the QuickBooking activity, a log out button that signs out the current user, and a bottom navigation bar that allows the user to navigate to different activities like ViewBookingActivity, CreateEWallet, and Profile. It also sets the home button on the bottom navigation bar as selected when the activity is created.
Mechanic E-Wallet	"activity_mechanic_ewallet.xml" is an XML layout file for an activity in our Android app that displays the user interface for a mechanic's e-wallet. It uses a CoordinatorLayout as the root layout, which contains an AppBarLayout, a NestedScrollView, and a ConstraintLayout. The AppBarLayout has a LinearLayout that contains a ImageButton and a TextView which is used to navigate back to	"MechanicEWallet.java" is an activity class for the E-wallet feature for the mechanics in the Scoot app. The class sets up the layout for the E-wallet feature and displays the current balance on the wallet. It also has a bottom navigation bar that allows the mechanics to navigate to other features such as requests, orders, and profile. Additionally, it retrieves the transaction

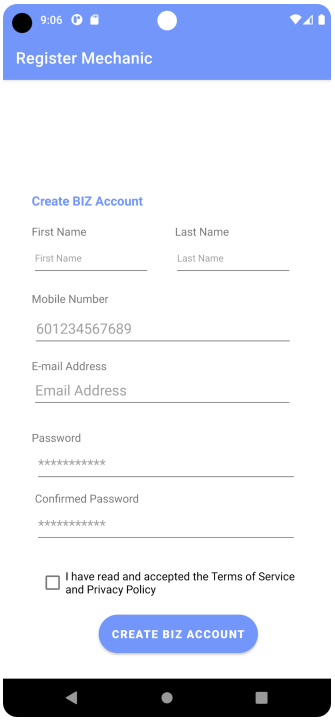
	the main menu and display the title "E-Wallet". NestedScrollView contains a ConstraintLayout which has several LinearLayouts, ImageButtons and TextViews that is used to display the balance and transaction history of the E-wallet and also to navigate to the E-wallet Setting and Transaction history. It also uses a BottomNavigationView to navigate between different activities.	history from Firebase and displays it in a ListView for the mechanics to view. This class also includes logic for signing out of the app and going back to the login/register page.
Mechanic Order Price 	"activity_mechanic_order_price.xml" is an XML layout file for an activity in our Android app. It defines the layout of the UI elements that will be displayed on the screen when the activity is launched. The layout includes an ImageView, an EditText and a Button. The ImageView will be used to display an image and the EditText will be used to take input from the user. The Button will be used to submit the order price. The layout also includes margins, padding and hint for the EditText, and a background for the button. The design is a simple layout for a screen where the mechanic can enter the price for the order.	"MechanicOrderPrice.java" is an activity file in our Android app that allows a mechanic user to enter a price for a specific order. The activity layout includes a form with an EditText field for the price and a submit button. When the button is pressed, the entered price is saved to the order object, and then is updated in the Firebase real-time database under the specific order and also updates the price in the BookingDetails node. The activity also implements Serializable, this is used to pass the order object from one activity to another.

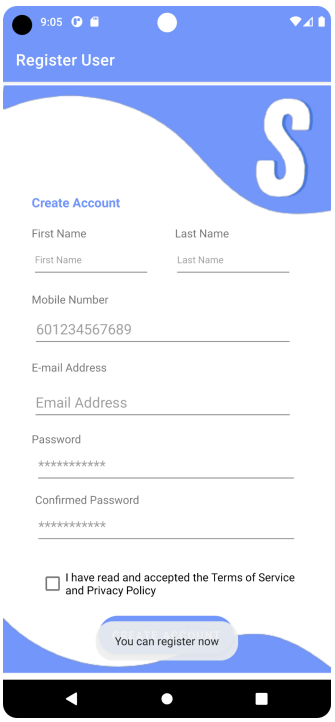
Mechanic orders 	Uses a LinearLayout to organize the layout of the screen. The layout includes a ListView that displays the list of orders, and a BottomNavigationView that allows users to navigate to different sections of the app.	Contains the logic for the activity, including retrieving data from a Firebase database and populating the ListView with the data. The bottom navigation view is set to order by default. It also has an item selected listener with switch case statement to navigate to different sections.
Mechanic profile 	The layout uses a RelativeLayout as the root layout to organize the layout of the screen. It includes a LinearLayout that displays the mechanic;s name, email and phone number. There is a BottomNavigationView for navigation and another LinearLayout containing TextVIews, EditTexts for the user to edit their first name, last name and phone number.	The java file contains the logic for the MechanicProfile activity, including retrieving data from a Firebase database, populating the UI elements with the data, and handling user interactions to update the profile information.

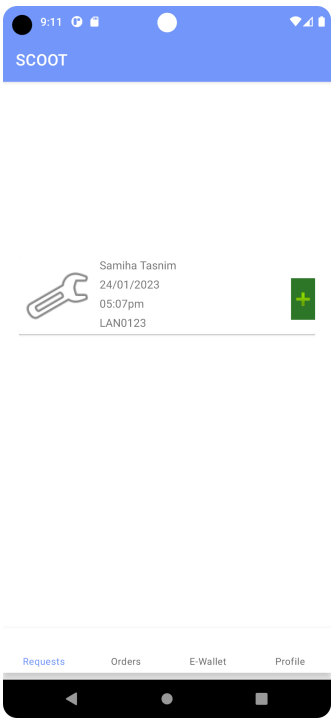
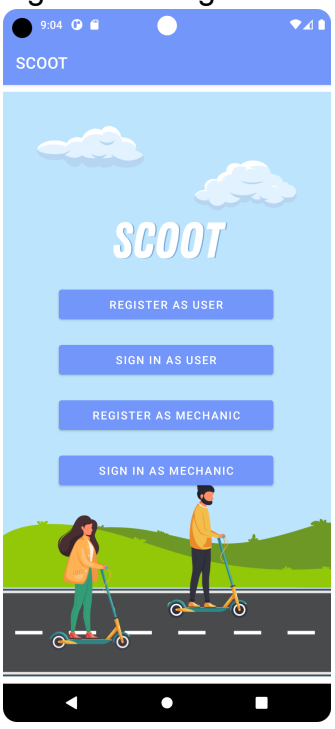
Mechanic Requests 	It uses ListVIew to display a list of requests for a mechanic, BottomNavigationVIew for navigation, LinearLayout to create space at the top of the screen and app:menu attribute to specify the menu resource file that contains the items to be displayed in the BottomNavigationView.	It displays a list of requests for a mechanic, using a ListView and Firebase database. It also uses a BottomNavigationVIew for navigation and has an onStart() method that populated the list with data from the Firebase Database. It also eliminates the default animation when switching activities.
Order		This is a java file that implements Serializable interface. It represents an order made by the user and contains various fields such as name, date, time, model, userID, status, price, bookingID, and hasPaid. It also includes getters and setters for each of these field, as well as constructors and an overridden toString() method. It uses Exclude annotation from the Firebase package to mark the key field which is used to store data in a Firebase database.

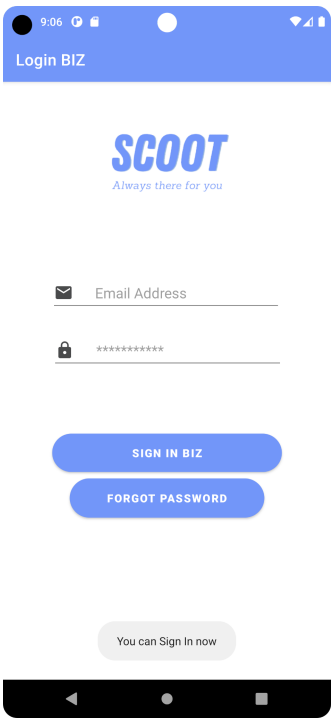
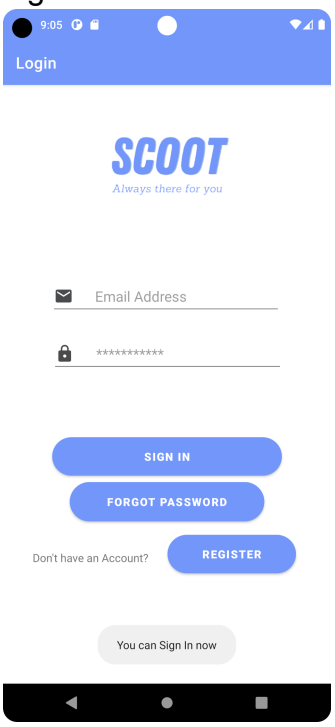
Order list 	Designed for the mechanic to view and update the status of the order. Uses a LinearLayout as the root layout to organize the layout of the screen. It contains an ImageView, LinearLayouts and TextVIews, ImageButton and TextView that displays the order name, date, time, model and price and the status of the order. The ImageButton is used for the mechanic to indicate that the job is done. The layout uses an ImageView, TextViews and an ImageButton to display an order information and its status.	It is an ArrayAdapter that displays a list of orders in a ListView. The ArrayAdapter's constructor takes in two arguments, an activity context, and a list of order objects. It populates the TextViews with the values from the order object, it also uses ImageButton for the mechanic to mark the order as done. The code also has an updateOrderStatus method that updates the status of an order in the Firebase Database.
Profile 	To display user's profile information. The layout uses a RelativeLayout as the root layout to organize the layout of the screen. It includes a LinearLayout that uses TextVIews to display the user's name, email and phone number. There is also a BottomNavigationVIew that allows users to navigate to different sections of the app and another LinearLayout which contains a ListView for displaying a list of e-scooters. Additionally, there is a button for adding a new e-scooter and another BottomNavigationVIew for profile navigation.	It uses Firebase for user authentication and database storage. The class displays information about the current logged-in user such as name, email and phone number. It also includes a ListVIew that displays the user's e-scooters and a button to add more e-scooters. The class also sets up a bottom navigation bar for navigating to different parts of the app.

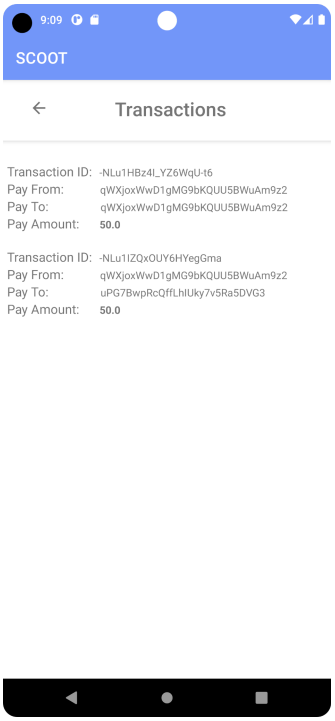
Profile information 	It has a RelativeLayout as the parent layout. Inside, there are several child layouts including a LinearLayout, a BottomNavigationView and another LinearLayout. The LinearLayout at the top contains three TextVIews to display the user's name, email and phone number. The BottomNavigationView is used for navigation within the app and the second LinearLayout contains EditText fields for the user to input or edit their first name, last name, and phone number. There are also TextVIews to label the EditTexts fields.	It retrieves user's information from a FIrebase database and displays it in TextVIews. The user can also edit their first name, last name and phone number by entering new values into EditTexts and clicking a "Edit User" button. The changes are then saved to the Firebase database. The activity also includes a bottom navigation bar that allows the user to navigate to other parts of the app.
Quick booking 	The layout is used to create the user interface for a quick booking feature which allows users to input data such as service date and time and model of e-scooter. It uses a LinearLayout as the root layout, with various elements nested inside. The layout includes RelativeLayout, ImageVIew, TextVIew and EditText elements. It also uses various layout parameters such as width, height, margin and padding to position and style the elements.	The activity displays a layout with a button "Book Now" and an ImageView "back button", EditText fields for service date, service time, pickup address, pickup phone, dropoff address, dropoff phone and model of e-scooter. The back button allows the user to navigate back to the previous activity. When a user presses the "Book Now" button, the data entered in the EditText fields are saved in the BookingDetails object and added to the database using DAOBookingDetails. The user is then taken to the "ViewBookingActivity" class.

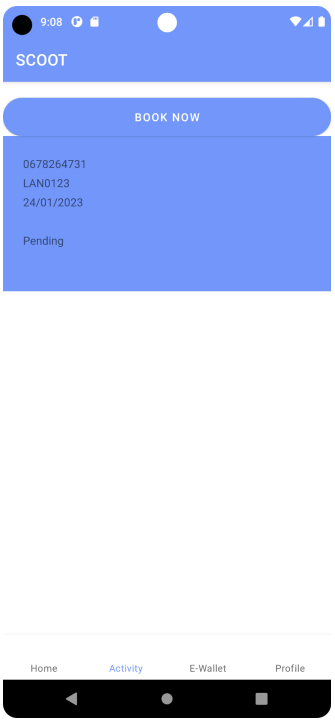
Read write user details		This is a java file that represents the user's personal details. It contains various fields and includes getters and setters for each of these fields as well as constructor. This class is used to store and retrieve user's personal details and provides a way to set and get user's personal details which can be used in other parts of the application.
Register mechanic 	It allows users to register as a "mechanic" presumably someone who works on an e-scooter. It includes several UI elements such as a checkbox, textviews and edittexts for the user to enter their information like mobile number, last name and password, There is also a "Create BIZ Account" button that when clicked will likely send this information to the database server for verification and account creation.	It uses FIrebase Auth and Firebase database to handle user registration. It has fields for the user to input their first name, last name, mobile number, email address, passwords and confirm password. It also has a checkbox for the user to confirm they have read and accepted the terms of service and privacy policy. The code includes validation for the mobile number using regular expressions and checks for empty fields and matching passwords. It also uses toast messages to display errors or success messages. If the registration is successful, the user is directed to a new page.
Register user	This is for the user registration page, with views for displaying text such as "Mobile Number"	It uses Firebase Authentication and database services to create a new

	and “password” and EditText views for the user to enter their information, such as their mobile number and last name.	user account. The user enters their first name, last name, mobile number, email address, password and confirms their password. It then performs various checks to ensure that the input data is valid, such as checking if the fields are empty, if the mobile number is in the correct format, and if the password and confirm password fields match. If all the checks pass, it creates a new user account using the Firebase Authentication service and stores the user’s information in the Firebase database. It also displays toast messages to provide feedback to the user on the success or failure of the registration process.
Request list	It creates a card view that contains a linear layout with horizontal orientation. The layout contains an image view, a linear layout with four text views and another linear layout with an image button.	This class is passed a context (the current activity) and a list of BookingDetails objects when it is created. The getView method is overridden to display the details of each BookingDetails object in a list item layout. The AcceptRequest method is called when the accept button is clicked on a list item, it sets the ‘hasMechanic’ to true and assigns the current mechanic to the booking by

		updating the data on the Firebase database.
Sign-in and register 	This layout contains a background image and four buttons. The buttons are inside a linear layout that is vertically oriented and the layout uses constraint layout to set the position of each element.	It sets onClickListeners for the four buttons in the layout. First button 'btnSignIn" opens the "SignInUser" activity when clicked. The second button "btnRegister" opens the "RegisterUser" activity when clicked. The third button "btnSignInMechanic" opens the "SignInMechanic" activity when clicked and the fourth button "btnRegisterMechanic" opens the "RegisterMechanic" activity when clicked.
Sign-in mechanic	This layout contains an image, three EditText fields for email, password and a button for login,	It uses Firebase Authentication to authenticate the user's

	and another button for forgot password. It uses constraint layout to position the elements.	email and password and allow them to sign in. If the authentication is successful, it will take the user to the next activity. If not, it will show an error message.
Sign-in user 	This layout contains several UI elements like edit text fields for email and password, buttons for sign-in, register, forgot password and an image view. It uses constraint layout to place the elements on the screen.	User can enter their email and password and then press the login button to sign-in. If the email or password is empty or the email is not in a valid format, the app will display an error message. If the login is successful, the app will redirect the user to the next page.
Transaction		This java file represents a financial transaction. It contains various fields,

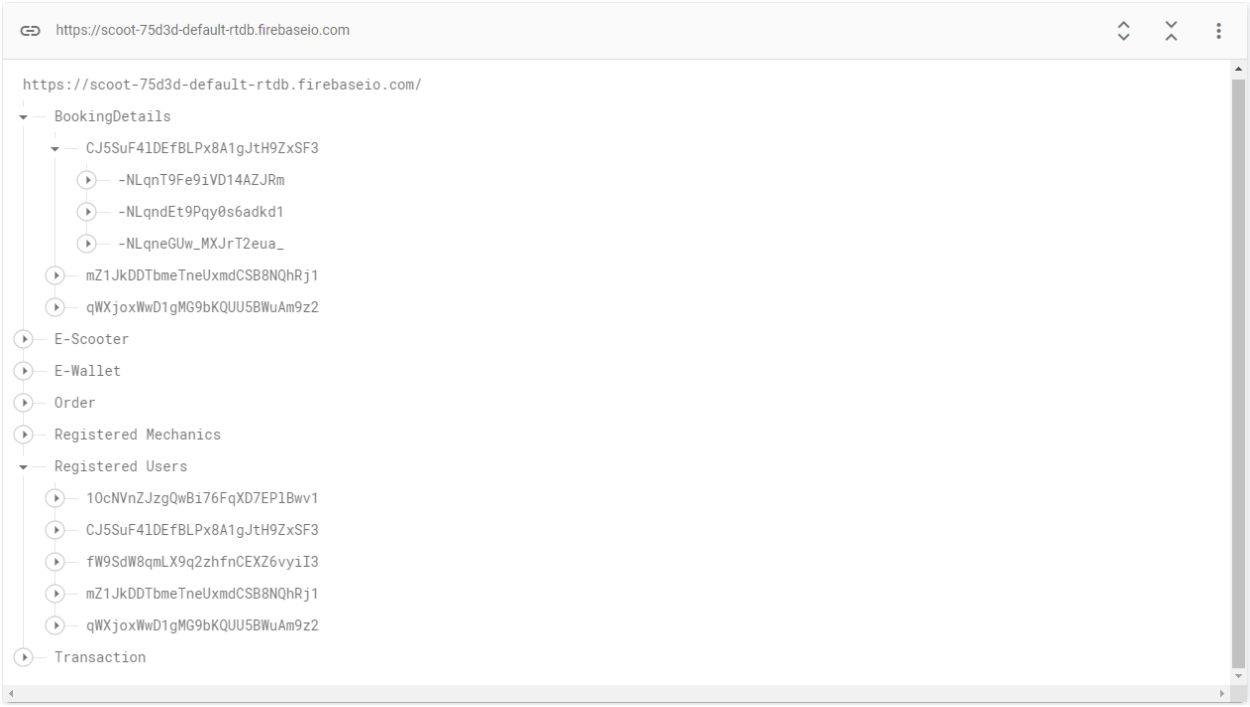
		getters and setters and constructor. It has a method named 'getPayTo' which returns the payee of the transaction and a method 'getPayAmount' which calculates the amounts of the transaction.
Transaction list adapter 	It defines a linear layout containing several nested linear layouts, each of which contains a textView. Each textVIew displays some text, and some of the textViews have an id attribute to reference the textView in the Java code.	This is a custom adapter for a listVIew. It is used to display a list of transactions, each with a transaction ID, payer, payee and amount.
View booking activity	It contains a list view, a button, a linear layout and BottomNavigationVIew.	It displays a list of bookings to the user. It initializes and assigns variables such as a bottomNavigationView and a ListVIew and sets listeners for when a user clicks on certain elements. When the activity starts, it retrieves booking data from a Firebase database and

		updates the ListView with the information. It also has a filter for not showing the already paid bookings.

4.0 Proof of successful implementation

Demo video link of our app :
https://drive.google.com/file/d/16LqXX3RLSWrDkDCNEVYRCVolW7dhj04s/view?usp=sharing

1. Database (CRUD)



The data being stored in our database is divided into their respective nodes. Each node has a child either based on their unique user key or their unique mechanic key depending on which module is used. Each of these children also has their own unique key to store data based on their nodes. For example, BookingDetails has a child with the key of the registered user who made a booking. That child also has another child with a unique key to store data of that booking made by the user. This is because a user is able to make many bookings but a booking is only made by one user.

```
public DAOBookingDetails(){

    authProfile = FirebaseAuth.getInstance();
    FirebaseDatabase db = FirebaseDatabase.getInstance();
    databaseReference = db.getReference( path: "BookingDetails");
    databaseUsers = db.getReference( path: "Registered Users");
}
 Aniq +1
public Task<Void> add(BookingDetails bd){
    String key = authProfile.getCurrentUser().getUid();
    String BDKey = databaseReference.child(key).push().getKey();
     Aniq
    databaseUsers.child(key).addValueEventListener(new ValueEventListener() {
         Aniq
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {
            String name = snapshot.child( path: "fullName").getValue(String.class);
            bd.setName(name);
            databaseReference.child(key).child(BDKey).setValue(bd);
        }
         Aniq
        @Override
        public void onCancelled(@NonNull DatabaseError error) {

        }
    });
    bd.setKey(BDKey);
    bd.setParentKey(key);
    return databaseReference.child(key).child(BDKey).setValue(bd);
}
 Aniq
public Task<Void> remove(String UID, String key){
    return databaseReference.child(UID).child(key).removeValue();
}
```

Creating

In this class of DAOBookingDetails, we connect our databaseReference to the BookingDetails node. We then get the key of the current user and push a unique BookingDetails key into the child of the user key. In this class, we are able to add a BookingDetails into the unique node for individual users by using .setValue.

```
//Displaying Profile Information
TextView txt_profileName = findViewById(R.id.txt_profileName);
TextView txt_profileMail = findViewById(R.id.txt_profileMail);
TextView txt_profileNumber = findViewById(R.id.txt_profileNumber);
databaseUser = FirebaseDatabase.getInstance().getReference( path: "Registered Users").child(key);
 Aniq
databaseUser.addValueEventListener(new ValueEventListener() {
     Aniq
    @Override
    public void onDataChange(@NonNull DataSnapshot snapshot) {
        String firstName = snapshot.child( path: "firstName").getValue(String.class);
        String lastName = snapshot.child( path: "lastName").getValue(String.class);
        String mobileNo = snapshot.child( path: "mobileNo").getValue(String.class);
        txt_profileName.setText(firstName + " " + lastName);
        txt_profileNumber.setText(mobileNo);
    }

     Aniq
    @Override
    public void onCancelled(@NonNull DatabaseError error) {

    }
});
```

Reading

In this java file for Profile, we are able to read the information in the Registered Users node in our database. By getting the reference of the individual user, we put a reference to that node. Inside those nodes, we store their firstName, lastName, mobileNo and much more. Using DataSnapshot, we read those data and set it to the textView in our activity layout to display these data of your profile.

```
//Edit User Details
EditText ETFirstNameEdit = findViewById(R.id.ETFirstNameEdit);
EditText ETLastNameEdit = findViewById(R.id.ETLastNameEdit);
EditText ETPhoneNumberEdit = findViewById(R.id.ETPhoneNumberEdit);
Button BtnEditUser = findViewById(R.id.BtnEditUser);
BtnEditUser.setOnClickListener(v -> {
    String firstName = ETFirstNameEdit.getText().toString();
    String lastName = ETLastNameEdit.getText().toString();
    String phoneNumber = ETPhoneNumberEdit.getText().toString();
    if(!firstName.isEmpty()) {
        databaseUser.child( pathString: "firstName").setValue(firstName);
        Toast.makeText( context: this,  text: "Record has been updated", Toast.LENGTH_SHORT).show();
    }
    if(!lastName.isEmpty()) {
        databaseUser.child( pathString: "lastName").setValue(lastName);
        Toast.makeText( context: this,  text: "Record has been updated", Toast.LENGTH_SHORT).show();
    }
    if(!phoneNumber.isEmpty()) {
        databaseUser.child( pathString: "mobileNo").setValue(phoneNumber);
        Toast.makeText( context: this,  text: "Record has been updated", Toast.LENGTH_SHORT).show();
    }
});
```

Update

In this java profile for ProfileInformation, this activity is used to edit the user's first name, last name and their mobile number. Like before, we set a reference to the individual user's node in the Registered Users node. We then use .setValue to update the existing information in the database.

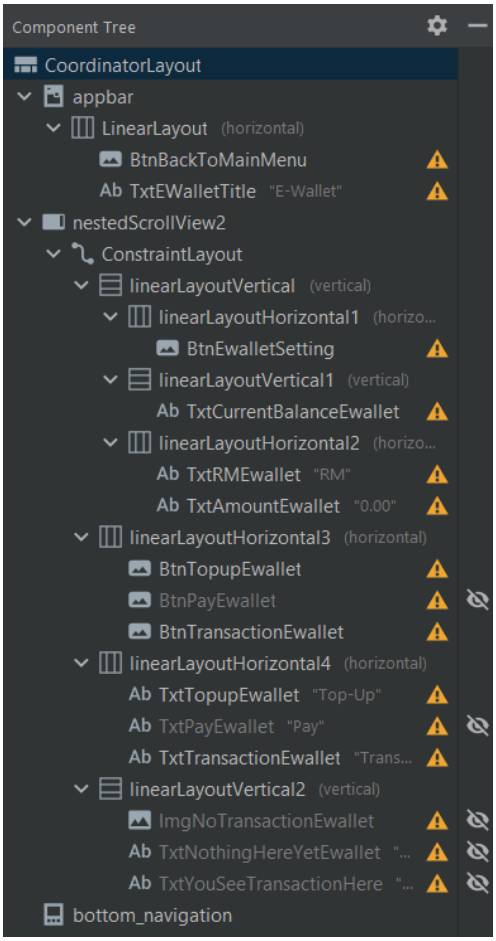
```
1 usage   Aniq
public Task<Void> remove(String UID, String key){
    return databaseReference.child(UID).child(key).removeValue();
}
```

Delete

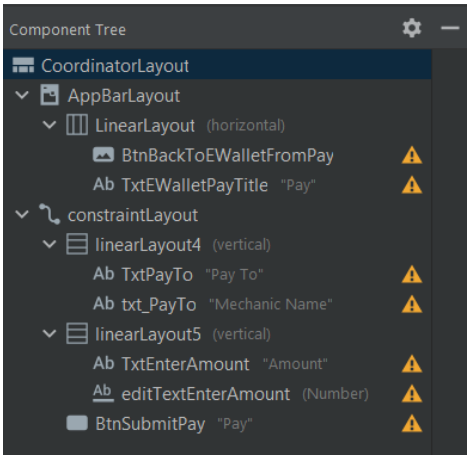
In this code in class DAOBookingDetails, we have a method to remove booking details by inputting a unique user key and a unique key of the BookingDetails to be removed.

2. Different View and ViewGroups

For Views, we used various Views types such as button, imageButton, image, textFields and also RecyclerView. As for ViewGroups, various types of containers were used. This includes LinearLayout, RelativeLayout, and ConstraintLayout. There is also a ViewGroup that is nested inside another ViewGroups to create a more complex layout.



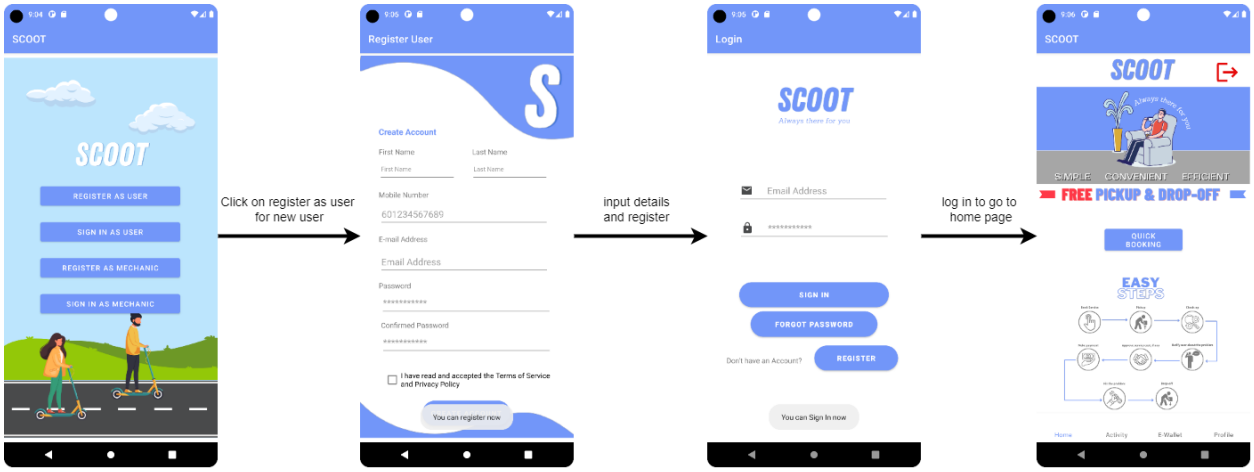
Example 1



Example 2

3. App Flow

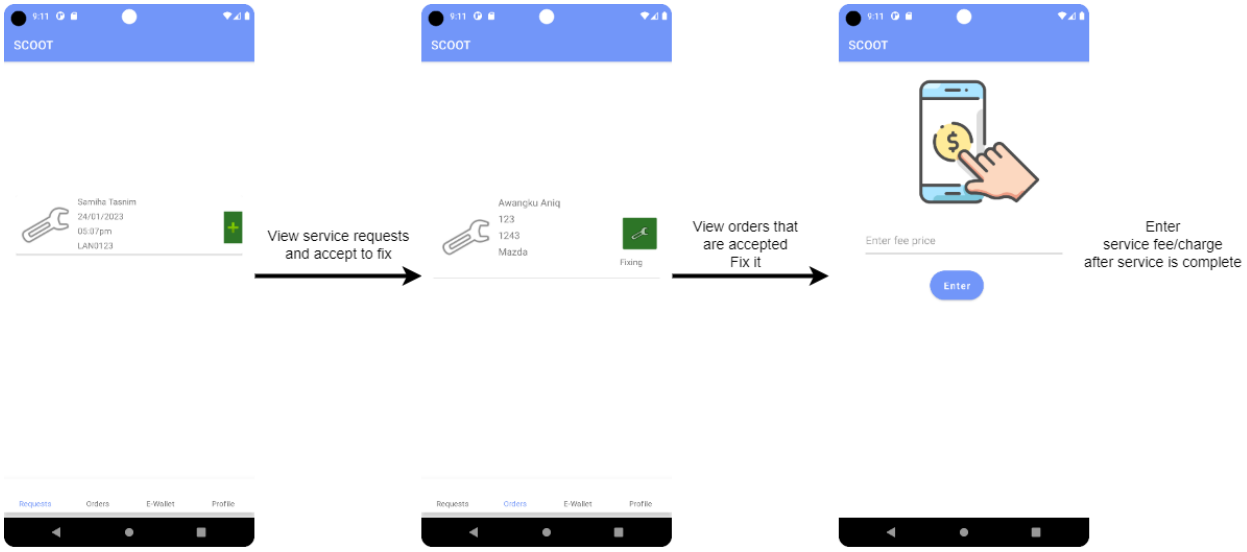
Steps to create a user account and log in.



Steps to book an e-scooter service.

Top up your e-wallet and view the user's transaction history.

View service requests, fix orders, and charge the customer a service fee as a mechanic.

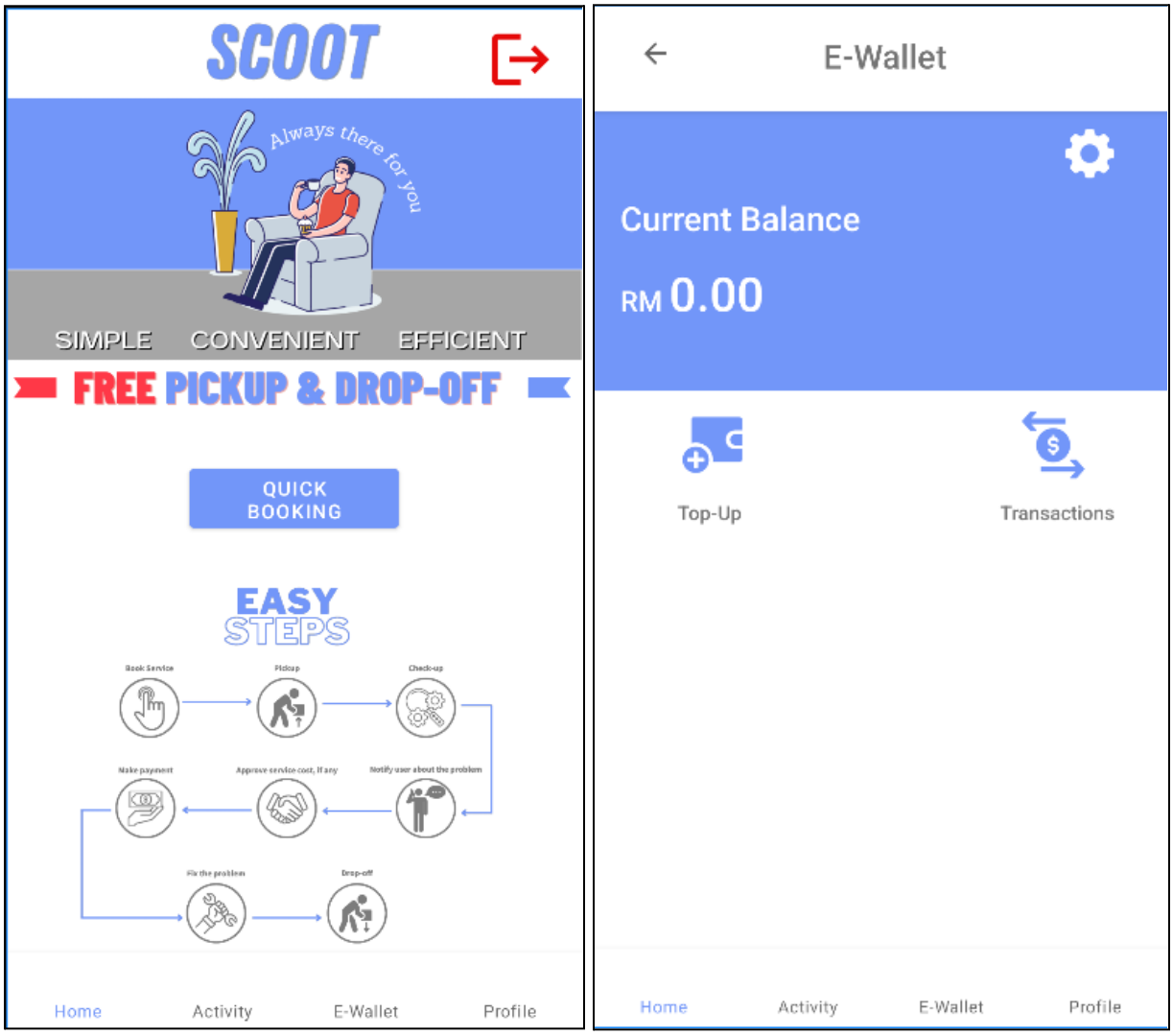


4. Accessibility

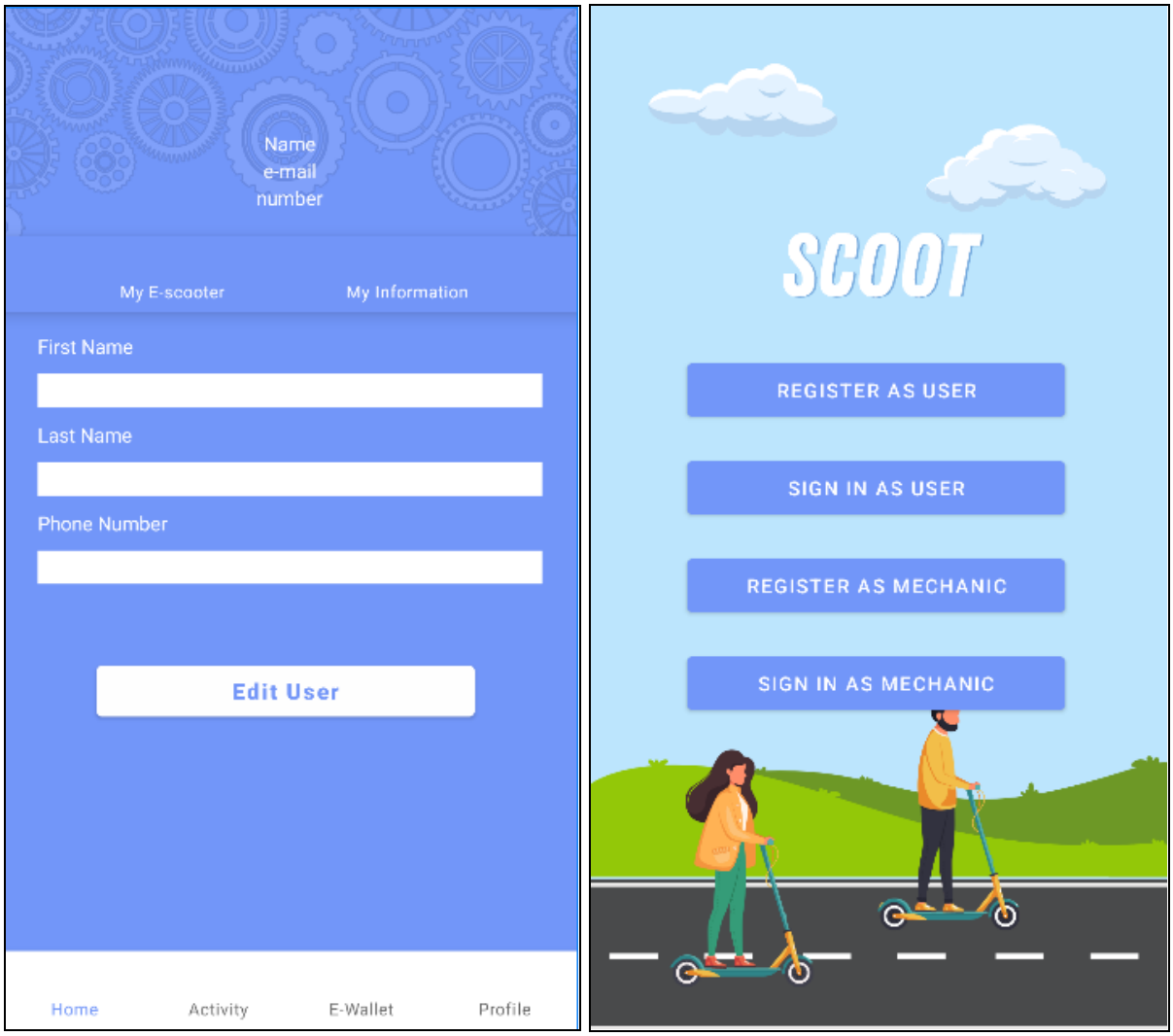
Accessibility provided in our mobile application :

1. Use clear and simple language in all interface elements such as for the button labels and error messages.
2. Provide alternative text for images and icons.
3. Use high-contrast colors and large fonts sizes to make the user with visual impairments easier to read.

5. UI Design



Home Page and E-Wallet Activity



Profile Information and Opening Activity

In our UI Design, we use the color of white and blue very consistently to maintain a minimalistic and simplistic design for our users. We also include icons for our buttons in our E-Wallet page for users to simply understand what each button does without using the texts. The placement for each button is also easy to see and has a distinct color from the background for the users to distinguish each interactive element from the background. There is also a bottom navigation bar for users to quickly navigate to each activity to save time and ease users. There is also imageView to represent some images which help users understand the app and to make our app more nicer and cleaner.

6. Extra Functionality

For our extra functionalities, we have only one in which we implemented a method that directs our Users/Mechanics immediately to the GMail app after successfully registering for easier navigation when Users/Mechanics wish to verify their emails.

7. Error Handling

Error handling refers to the process of anticipating, detecting, and resolving errors that occur during the execution of a program. It is a mechanism to handle the runtime errors so that normal flow of the application can be maintained without stopping unexpectedly. This can include things like input validation, logging errors, and providing user-friendly error messages.

a. RegisterUser Class

```
//if First Name is not entered
if (TextUtils.isEmpty(textFirstName)) {
    Toast.makeText( context: RegisterUser.this,  text: "Please enter your first name", Toast.LENGTH_LONG).show();
    editTxtFirstName.setError("First Name is required");
    editTxtFirstName.requestFocus();
}
```

This if statement is to check if the User has input his/her First Name or not. If he/she has not entered his/her First Name, an error will appear and bring the User back to the First Name input box. It will remind the User to enter his/her First Name.

```
//if Last Name is not entered
else if (TextUtils.isEmpty(textLastName)) {
    Toast.makeText( context: RegisterUser.this,  text: "Please enter your last name", Toast.LENGTH_LONG).show();
    editTxtLastName.setError("Last Name is required");
    editTxtLastName.requestFocus();
}
```

This else if statement is to check if the User has input his/her Last Name or not. If he/she has not entered his/her Last Name, an error will appear and bring the User back to the Last Name input box. It will remind the User to enter his/her Last Name.

```
//if Mobile Number is not entered
else if (TextUtils.isEmpty(textMobileNo)) {
    Toast.makeText( context: RegisterUser.this,  text: "Please enter your mobile no", Toast.LENGTH_LONG).show();
    editTextMobileNo.setError("Mobile Number is required");
    editTextMobileNo.requestFocus();
}
```

This else if statement is to check if the User has input his/her Mobile Number or not. If he/she has not entered his/her Mobile Number, an error will appear and bring the User back to the Mobile Number input box. It will remind the User to enter his/her Mobile Number.

```
//if Mobile Number does not follow the rules which is first number must be 6
else if (!mobileMatcher.find()){
    Toast.makeText( context: RegisterUser.this, text: "Please re-enter your mobile no", Toast.LENGTH_LONG).show();
    editTextMobileNo.setError("Mobile Number is invalid");
    editTextMobileNo.requestFocus();
}
```

This else if statement is to check if the Mobile Number that the user has input follows the pattern whereby the first number must be 6. If it does not follow the format, the app will show an error saying that the Mobile Number is invalid and a notification will pop up to prompt the User to re-enter his/her Mobile Number while bringing the User back to the Mobile Number input box. It will remind the User to re-enter his/her Mobile Number.

```
//if Email Address is not entered
else if (TextUtils.isEmpty(textEmailAddress)) {
    Toast.makeText( context: RegisterUser.this, text: "Please enter your email address", Toast.LENGTH_LONG).show();
    editTxtEmailAddress.setError("Email Address is required");
    editTxtEmailAddress.requestFocus();
}
```

This else if statement is to check if the User has input his/her Email Address or not. If he/she has not entered his/her Email Address, an error will appear and bring the User back to the Email Address input box. It will remind the User to enter his/her Email Address.

```
//if Email Address does not follow the format
else if (!Patterns.EMAIL_ADDRESS.matcher(textEmailAddress).matches()) {
    Toast.makeText( context: RegisterUser.this, text: "Please re-enter your email", Toast.LENGTH_LONG).show();
    editTxtEmailAddress.setError("Valid email is required");
    editTxtEmailAddress.requestFocus();
}
```

This else if statement is to check if the User has entered a valid Email Address or not. If he/she entered an invalid Email Address, an error will appear and bring the User back to the Email Address input box. It will remind the User to enter a valid Email Address.

```
//if Password is not entered yet
else if (TextUtils.isEmpty(textPassword)) {
    Toast.makeText( context: RegisterUser.this, text: "Please enter your password", Toast.LENGTH_LONG).show();
    editTxtPassword.setError("Password is required");
    editTxtPassword.requestFocus();
}
```

This else if statement is to check if the User has input his/her Password or not. If he/she has not entered his/her Password, an error will appear and bring the User back to the Password input box. It will remind the User to enter his/her Password.

```
//if Confirm Password is not entered
else if (TextUtils.isEmpty(textConfirmPassword)) {
    Toast.makeText( context: RegisterUser.this,  text: "Please enter your confirmed password", Toast.LENGTH_LONG).show();
    editTxtConfirmPassword.setError("Password Confirmation is required");
    editTxtConfirmPassword.requestFocus();
}
```

This else if statement is to check if the User has input his/her Confirmed Password or not. If he/she has not entered his/her Confirmed Password, an error will appear and bring the User back to the Confirmed Password input box. It will remind the User to enter his/her Confirmed Password.

```
//if the Confirm Password is not the same as entered Password
else if (!textPassword.equals(textConfirmPassword)){
    Toast.makeText( context: RegisterUser.this,  text: "Please set the same password", Toast.LENGTH_LONG).show();
    editTxtConfirmPassword.setError("Password Confirmation is required");
    editTxtConfirmPassword.requestFocus();
    //Clear the entered passwords
    editTxtPassword.clearComposingText();
    editTxtConfirmPassword.clearComposingText();
}
```

This else if statement is to check if the Confirmed Password that the User has input is the same as the initial Password . If they are not the same, an error will appear and bring the User back to the Confirmed Password input box. It will remind the User to enter his/her Confirmed Password. Both the Password and Confirmed Password input boxes will be cleared.

```
//if checkbox has not been checked
else if (!checkBox.isChecked()){
    Toast.makeText( context: RegisterUser.this,  text: "Please check this box to agree to our Terms of Service and Privacy Policy", Toast.LENGTH_LONG).show();
    checkBox.setError("Your agreement to the Terms of Service and Privacy Policy is required");
    checkBox.requestFocus();
} else {
```

This else if statement is to check whether the checkbox is ticked, showing the User agrees with the Terms of Service and Privacy Policy. If the checkbox is not checked, an error will pop up mentioning that the User's agreement to the terms and conditions is required in order to create an account.

```
//if Password is too weak
catch (FirebaseAuthWeakPasswordException e) {
    editTxtPassword.setError("Your password is too weak. Kindly use a mix of alphabets, numbers and special characters");
    editTxtPassword.requestFocus();
}
```

This catch statement is to check if the Password used by the User is too weak. If it is too weak, then the User will be brought back to the Password input box.

```
//if Email Address used already has an account
catch (FirebaseAuthInvalidCredentialsException e){
    editTxtEmailAddress.setError("Your email is invalid or already in use. Kindly re-enter");
    editTxtEmailAddress.requestFocus();
}
```

This catch statement is to check if the Email Address used already has an account or not. If the Email Address has already been used, an error will state that the email has been used already and the app will redirect the User back to the Email Address input box. The User will then have to re-enter his/her Email Address.

```
//if  Email already has been registered with an account
catch (FirebaseAuthUserCollisionException e){
    editTxtEmailAddress.setError("User is already registered with this email. Use another email.");
    editTxtEmailAddress.requestFocus();
}
```

This catch statement is to check if the User's Email Address has already been registered. If the Email Address is already registered, the app will redirect the User back to the Email Address input box. The User will have to use another Email Address.

b. SignInUser Class

```
if (TextUtils.isEmpty(textEmail)){
    Toast.makeText( context: SignInUser.this,  text: "Please enter your email", Toast.LENGTH_LONG).show();
    editTxtLoginEmail.setError("Email is required");
    editTxtLoginEmail.requestFocus();
}
```

This else if statement is to check if the User has input his/her Email Address or not. If he/she has not entered his/her Email Address, an error will appear and bring the User back to the Email Address input box. It will remind the User to enter his/her Email Address.

```
//if Email Address is invalid
else if (!Patterns.EMAIL_ADDRESS.matcher(textEmail).matches()){
    Toast.makeText( context: SignInUser.this,  text: "Please re-enter your email", Toast.LENGTH_LONG).show();
    editTxtLoginEmail.setError("Valid email is required");
    editTxtLoginEmail.requestFocus();
}
```

This else if statement is to check if the User has entered a valid Email Address or not. If he/she entered an invalid Email Address, an error will appear and bring the User back to the Email Address input box. It will remind the User to enter a valid Email Address.

```
//if Password has not been entered
else if (TextUtils.isEmpty(textPassword)){
    Toast.makeText( context: SignInUser.this,  text: "Please enter your password", Toast.LENGTH_LONG).show();
    editTxtLoginPassword.setError("Password is required");
    editTxtLoginPassword.requestFocus();
}
```

This else if statement is to check if the User has input his/her Password or not. If he/she has not entered his/her Password, an error will appear and bring the User back to the Password input box. It will remind the User to enter his/her Password.

```
//Check if email is verified before user can access their profile
if(firebaseUser.isEmailVerified()){
    Toast.makeText( context: SignInUser.this,  text: "You are Signed In now", Toast.LENGTH_SHORT).show();

    //Open user Profile
    //Start Home
    startActivity(new Intent( packageContext: SignInUser.this, MainActivity.class));
    finish(); //Close the Sign In Activity

}
else{
    firebaseUser.sendEmailVerification();
    authProfile.signOut(); //Sign out User
    showAlertDialog();
}
```

This if statement is to check if the Email has been verified. If it has been verified, it will let the User know he/she is signed in and direct them to the Home Page. If it has not been verified, the User is unable to Sign In.

```
//if User no longer exist or deleted by admin
catch(FirebaseAuthInvalidUserException e){
    editTxtLoginEmail.setError("User does not exist or is no longer valid. Please register again");
    editTxtLoginEmail.requestFocus();
}
```

This catch statement checks if the User does not exist or is no longer valid.

```
//invalid credentials
catch(FirebaseAuthInvalidCredentialsException e){ //User input wrong credentials
    editTxtLoginEmail.setError("Invalid credentials. Kindly, check and re-enter.");
    editTxtLoginEmail.requestFocus();
}
```

This catch statement checks if the User's credentials are invalid.

c. RegisterMechanic Class

```
if (TextUtils.isEmpty(textFirstName)) {
    Toast.makeText( context: RegisterMechanic.this, text: "Please enter your first name", Toast.LENGTH_LONG).show();
    editTxtMechanicFirstName.setError("First Name is required");
    editTxtMechanicFirstName.requestFocus();
}
```

This if statement is to check if the Mechanic has input his/her First Name or not. If he/she has not entered his/her First Name, an error will appear and bring the Mechanic back to the First Name input box. It will remind the User to enter his/her First Name.

```
//if Last Name is not entered
else if (TextUtils.isEmpty(textLastName)) {
    Toast.makeText( context: RegisterMechanic.this, text: "Please enter your last name", Toast.LENGTH_LONG).show();
    editTxtMechanicLastName.setError("Last Name is required");
    editTxtMechanicLastName.requestFocus();
}
```

This else if statement is to check if the Mechanic has input his/her Last Name or not. If he/she has not entered his/her Last Name, an error will appear and bring the User back to the Last Name input box. It will remind the Mechanic to enter his/her Last Name.

```
//if Mobile Number is not entered
else if (TextUtils.isEmpty(textMobileNo)) {
    Toast.makeText( context: RegisterMechanic.this, text: "Please enter your mobile no", Toast.LENGTH_LONG).show();
    editTextMechanicMobileNo.setError("Mobile Number is required");
    editTextMechanicMobileNo.requestFocus();
}
```

This else if statement is to check if the Mechanic has input his/her Mobile Number or not. If he/she has not entered his/her Mobile Number, an error will appear and bring the Mechanic back to the Mobile Number input box. It will remind the Mechanic to enter his/her Mobile Number.

```
//if Mobile Number does not follow the rules which is first number must be 6
else if (!mobileMatcher.find()){
    Toast.makeText( context: RegisterMechanic.this,  text: "Please re-enter your mobile no", Toast.LENGTH_LONG).show();
    editTextMechanicMobileNo.setError("Mobile Number is invalid");
    editTextMechanicMobileNo.requestFocus();
}
```

This else if statement is to check if the Mobile Number that the Mechanic has input follows the pattern whereby the first number must be 6. If it does not follow the format, the app will show an error saying that the Mobile Number is invalid and a notification will pop up to prompt the Mechanic to re-enter his/her Mobile Number while bringing the Mechanic back to the Mobile Number input box. It will remind the Mechanic to re-enter his/her Mobile Number.

```
//if Email Address is not entered
else if (TextUtils.isEmpty(textEmailAddress)) {
    Toast.makeText( context: RegisterMechanic.this,  text: "Please enter your email address", Toast.LENGTH_LONG).show();
    editTxtMechanicEmailAddress.setError("Email Address is required");
    editTxtMechanicEmailAddress.requestFocus();
}
```

This else if statement is to check if the Mechanic has input his/her Email Address or not. If he/she has not entered his/her Email Address, an error will appear and bring the Mechanic back to the Email Address input box. It will remind the Mechanic to enter his/her Email Address.

```
//if Email Address does not follow the format
else if (!Patterns.EMAIL_ADDRESS.matcher(textEmailAddress).matches()) {
    Toast.makeText( context: RegisterMechanic.this,  text: "Please re-enter your email", Toast.LENGTH_LONG).show();
    editTxtMechanicEmailAddress.setError("Valid email is required");
    editTxtMechanicEmailAddress.requestFocus();
}
```

This else if statement is to check if the Mechanic has entered a valid Email Address or not. If he/she entered an invalid Email Address, an error will appear and bring the Mechanic back to the Email Address input box. It will remind the Mechanic to enter a valid Email Address.

```
//if Password is not entered yet
else if (TextUtils.isEmpty(textPassword)) {
    Toast.makeText( context: RegisterMechanic.this,  text: "Please enter your password", Toast.LENGTH_LONG).show();
    editTxtMechanicPassword.setError("Password is required");
    editTxtMechanicPassword.requestFocus();
}
```

This else if statement is to check if the Mechanic has input his/her Password or not. If he/she has not entered his/her Password, an error will appear and bring the Mechanic back to the Password input box. It will remind the Mechanic to enter his/her Password.

```
//if Confirm Password is not entered
else if (TextUtils.isEmpty(textConfirmPassword)) {
    Toast.makeText( context: RegisterMechanic.this,  text: "Please enter your confirmed password", Toast.LENGTH_LONG).show();
    editTxtMechanicConfirmPassword.setError("Password Confirmation is required");
    editTxtMechanicConfirmPassword.requestFocus();
}
```

This else if statement is to check if the Mechanic has input his/her Confirmed Password or not. If he/she has not entered his/her Confirmed Password, an error will appear and bring the Mechanic back to the Confirmed Password input box. It will remind the Mechanic to enter his/her Confirmed Password.

```
//if the Confirm Password is not the same as entered Password
else if (!textPassword.equals(textConfirmPassword)){
    Toast.makeText( context: RegisterMechanic.this,  text: "Please set the same password", Toast.LENGTH_LONG).show();
    editTxtMechanicConfirmPassword.setError("Password Confirmation is required");
    editTxtMechanicConfirmPassword.requestFocus();
    //Clear the entered passwords
    editTxtMechanicPassword.clearComposingText();
    editTxtMechanicConfirmPassword.clearComposingText();
}
```

This else if statement is to check if the Confirmed Password that the Mechanic has input is the same as the initial Password . If they are not the same, an error will appear and bring the Mechanic back to the Confirmed Password input box. It will remind the Mechanic to enter his/her Confirmed Password. Both the Password and Confirmed Password input boxes will be cleared.

```
//if checkbox has not been checked
else if (!checkBoxMechanic.isChecked()){
    Toast.makeText( context: RegisterMechanic.this,  text: "Please check this box to agree to our Terms of Service and Privacy Policy", Toast.LENGTH_LONG).show();
    checkBoxMechanic.setError("Your agreement to the Terms of Service and Privacy Policy is required");
    checkBoxMechanic.requestFocus();
}
```

This else if statement is to check whether the checkbox is ticked, showing the Mechanic agrees with the Terms of Service and Privacy Policy. If the checkbox is not checked, an error will pop up mentioning that the Mechanic's agreement to the terms and conditions is required in order to create an account.

```
//if Password is too weak
catch (FirebaseAuthWeakPasswordException e) {
    editTxtMechanicPassword.setError("Your password is too weak. Kindly use a mix of alphabets, numbers and special characters");
    editTxtMechanicPassword.requestFocus();
}
```

This catch statement is to check if the Password used by the Mechanic is too weak. If it is too weak, then the Mechanic will be brought back to the Password input box.

```
//if Email Address used already has an account
catch (FirebaseAuthInvalidCredentialsException e){
    editTxtMechanicEmailAddress.setError("Your email is invalid or already in use. Kindly re-enter");
    editTxtMechanicEmailAddress.requestFocus();
}
```

This catch statement is to check if the Email Address used already has an account or not. If the Email Address has already been used, an error will state that the email has been used already and the app will redirect the Mechanic back to the Email Address input box. The Mechanic will then have to re-enter his/her Email Address.

```
//if  Email already has been registered with an account
catch (FirebaseAuthUserCollisionException e){
    editTxtMechanicEmailAddress.setError("Mechanic is already registered with this email. Use another email.");
    editTxtMechanicEmailAddress.requestFocus();
}
```

This catch statement is to check if the Mechanics's Email Address has already been registered. If the Email Address is already registered, the app will redirect the Mechanic back to the Email Address input box. The Mechanic will have to use another Email Address.

d. SignInMechanic Class

```
//if Email Address has not been entered
if (TextUtils.isEmpty(textEmail)){
    Toast.makeText( context: SignInMechanic.this,  text: "Please enter your email", Toast.LENGTH_LONG).show();
    editTxtMechanicLoginEmail.setError("Email is required");
    editTxtMechanicLoginEmail.requestFocus();
}
```

This else if statement is to check if the Mechanic has input his/her Email Address or not. If he/she entered an invalid Email Address, an error will appear and bring the Mechanic back to the Email Address input box. It will remind the Mechanic to enter a valid Email Address.

```
//if Email Address is invalid
else if (!Patterns.EMAIL_ADDRESS.matcher(textEmail).matches()){
    Toast.makeText( context: SignInMechanic.this,  text: "Please re-enter your email", Toast.LENGTH_LONG).show();
    editTxtMechanicLoginEmail.setError("Valid email is required");
    editTxtMechanicLoginEmail.requestFocus();
}
```

This else if statement is to check if the User has entered a valid Email Address or not. If he/she entered an invalid Email Address, an error will appear and bring the User back to the Email Address input box. It will remind the User to enter a valid Email Address.

```
//if Password has not been entered
else if (TextUtils.isEmpty(textPassword)){
    Toast.makeText( context: SignInMechanic.this,  text: "Please enter your password", Toast.LENGTH_LONG).show();
    editTxtMechanicLoginPassword.setError("Password is required");
    editTxtMechanicLoginPassword.requestFocus();
```

This else if statement is to check if the User has input his/her Password or not. If he/she has not entered his/her Password, an error will appear and bring the User back to the Password input box. It will remind the User to enter his/her Password.

```
//Check if email is verified before user can access their profile
if(firebaseUser.isEmailVerified()){
    Toast.makeText( context: SignInMechanic.this,  text: "You are Signed In now", Toast.LENGTH_SHORT).show();

    //Open Mechanic Profile
    //Start Home
    startActivity(new Intent( packageContext: SignInMechanic.this, MechanicRequests.class));
    finish(); //Close the Sign In Activity

}
else{
    firebaseUser.sendEmailVerification();
    authProfile.signOut(); //Sign out User
    showAlertDialog();
}
```

This if statement is to check if the Email has been verified. If it has been verified, it will let the Mechanic know he/she is signed in and direct them to the Home Page. If it has not been verified, the Mechanic is unable to Sign In.

```
//if User no longer exist or deleted by admin
catch(FirebaseAuthInvalidUserException e){
    editTxtMechanicLoginEmail.setError("User does not exist or is no longer valid. Please register again");
    editTxtMechanicLoginEmail.requestFocus();
}
```

This catch statement checks if the User does not exist or is no longer valid.

```
//invalid credentials
catch(FirebaseAuthInvalidCredentialsException e){ //
    editTxtMechanicLoginEmail.setError("Invalid credentials. Kindly, check and re-enter.");
    editTxtMechanicLoginEmail.requestFocus();
}
```

This catch statement checks if the User's credentials are invalid.

5.0 Tasks distribution table

No.	Person-in-charge	Tasks
1.	Surin A/L Pubalan	1. Front-end Developer 2. Creating and implementing Database 3. Implementing Exception Handling (Register and Sign In) 4. UX/UI Developer 5. Implementing Register and Sign In 6. Final report
2.	Awangku Aniq Hamiz Bin Awangku Badaruddin	1. Back-end and Front-end Developer 2. Debugging Code 3. Implementing Database 4. Implementing User Profile activities 5. Implementing Mechanic Profile activities 6. Final report
3.	Samiha Tasnim Dristy	1. Back-end and front-end developer 2. Tester 3. Implementing Quick Booking activities 4. Final report
4.	Muhammad Muqri Qawiem Bin Hanizam	1. Back-end and front-end developer 2. Tester 3. Implementing E-wallet activities 4. Final report