MIDDLE EAST TECHNICAL UNIVERSITY
NORTHERN CYPRUS CAMPUS

Computer Engineering Program

CNG 495 FALL 2022-2023

Term Project Final Report

Team Members : Hamzeh Abu Ali 2419471, (Ahmed Jaber 2470490)

Project Name : Sharek

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Brief

Our application aims to connect people around our village by providing an online taxi service platform. This application will allow users to order a taxi through a navigation system that shows available taxi drivers around the users. Users can register as taxi drivers or as normal customer users. Taxi drivers can be drivers with taxi cars, or students who own cars and are willing to provide a taxi service. The application will also provide the users with the functionality of sharing a ride with other users. They can open a "share a ride" form, and other interested users can see that form and participate in the ride. Taxi drivers have the option of taking those requests or not. After ordering a ride, users can call the taxi driver or contact them on WhatsApp.

We aim to help our fellow students and village residents have a better transportation experience by connecting them to available taxi drivers and students willing to provide a service. As transportation is a critical issue in students' lives, we hope that this application can ease our lives by providing a platform to connect us all. The novelty of this project is in providing drivers and students with a unified platform to communicate about transportation issues rather than posting their requests on general Facebook groups or asking friends, which may be risky due to security reasons, or timing constraints.

Our project's idea in essence is not new, Uber is a popular example of a transportation company. It is a company that provides a platform for individuals to hire cars, taxis, and other vehicles to travel from one place to another. Users can request a ride and track the location of the vehicle in real time. Payment is made through the app, and the fare is calculated based on the distance and time of the ride. Lyft is also another ride-hailing company that is similar to Uber. It was founded in 2012 and is also headquartered in San Francisco. It operates in the United States, Canada, and several other countries. It also provides bike and scooter rentals in select cities. Another car transportation company is BlaBlaCar. It is a long-distance ride-sharing company, founded in 2006 and headquartered in Paris. It operates in more than 22 countries across Europe, North Africa, and Latin America, and connects car owners traveling to the same destination with people looking for a ride.

The GitHub URL link for the project will be addressed in [1] in the list of references.

Chapter 2

Project Description

2.1 Project Components

Figure 2.1 below demonstrates the flow chart of the project between all the screens for normal user operations, and the actions taken based on certain conditions. Admin operations will be explained in the figures below, and the screens shown in the Figure will be further explained below. For further detailed code, the GitHub repository in [1] will include all the necessary code.

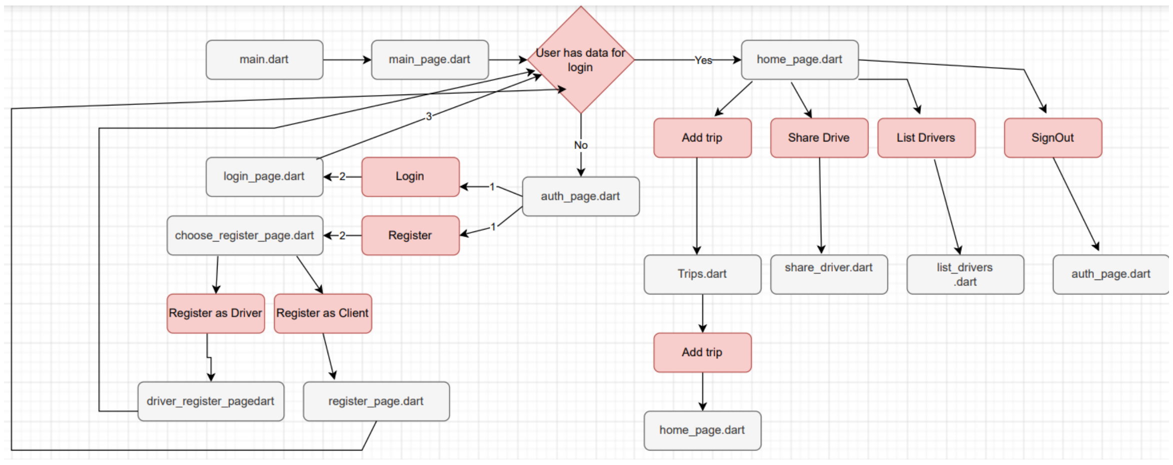
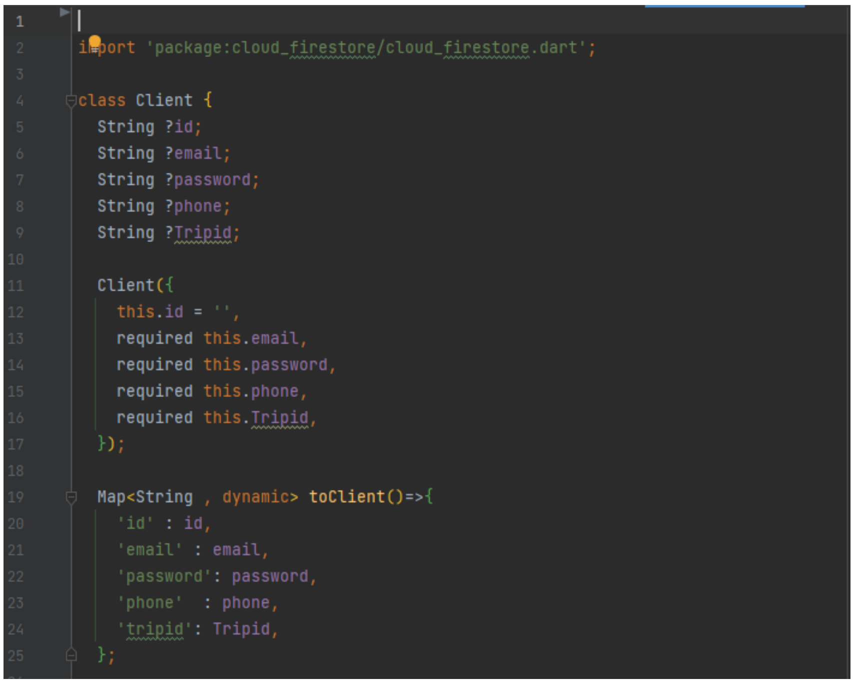


Figure 2.1: Application Flow Chart

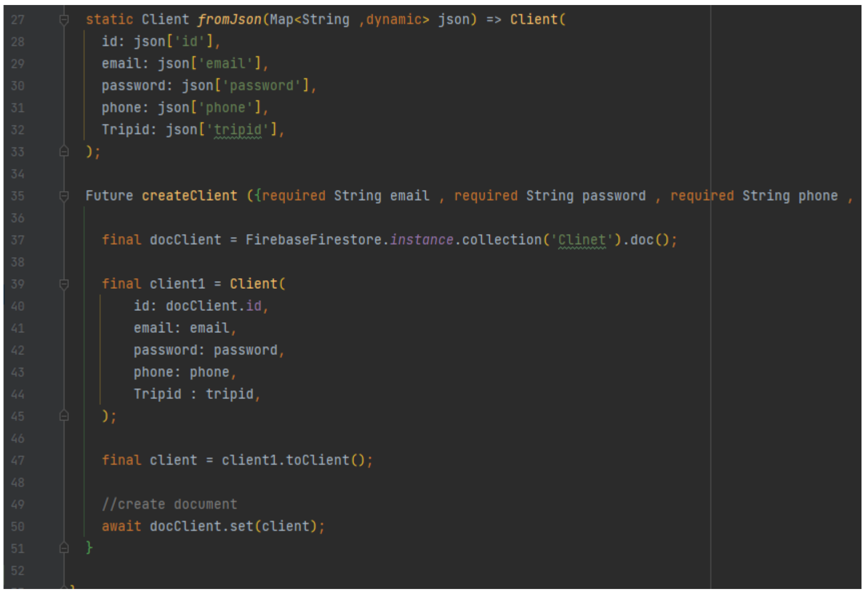
2.1.1 Client Class

Client class is shown below in Figures 2.2 and 2.3. Required constructors and attributes for the client is shown in the code.



```dart

import 'package:cloud_firestore/cloud_firestore.dart';

class Client {
  String ?id;
  String ?email;
  String ?password;
  String ?phone;
  String ?Tripid;

  Client({
    this.id = '',
    required this.email,
    required this.password,
    required this.phone,
    required this.Tripid,
  });

  Map<String , dynamic> toClient()=>{
    'id' : id,
    'email' : email,
    'password': password,
    'phone'  : phone,
    'tripid': Tripid,
  };
```

Figure 2.2: Client Class Code Snippet 1

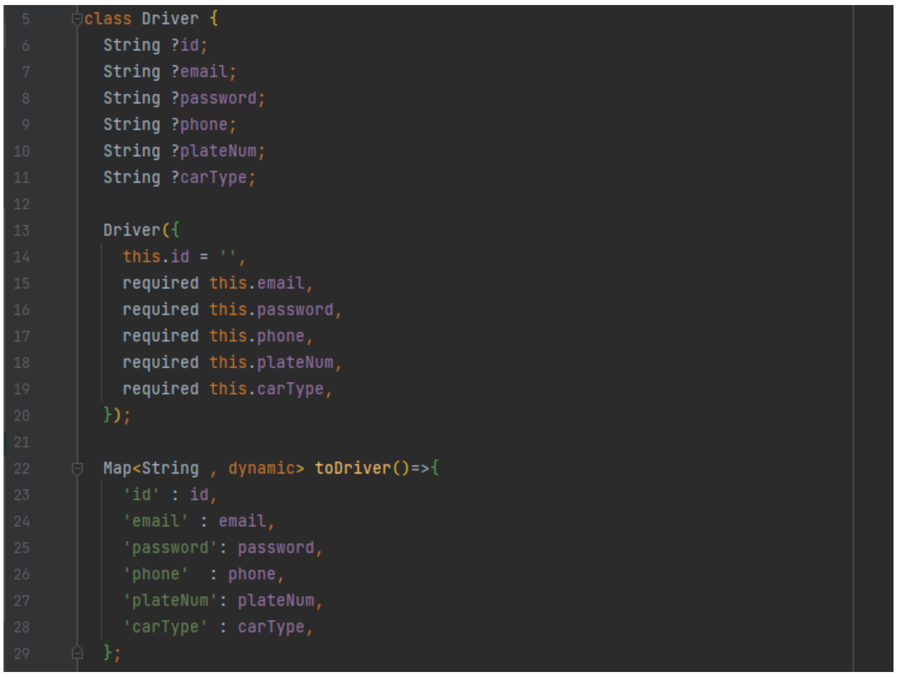


```dart
static Client fromJson(Map<String ,dynamic> json) => Client(
  id: json['id'],
  email: json['email'],
  password: json['password'],
  phone: json['phone'],
  Tripid: json['tripid'],
);

Future createClient ({required String email , required String password , required String phone ,

  final docClient = FirebaseFirestore.instance.collection('Clinet').doc();

  final client1 = Client(
      id: docClient.id,
      email: email,
      password: password,
      phone: phone,
      Tripid : tripid,
  );

  final client = client1.toClient();

  //create document
  await docClient.set(client);
}

```

Figure 2.3: Client Class Code Snippet 2

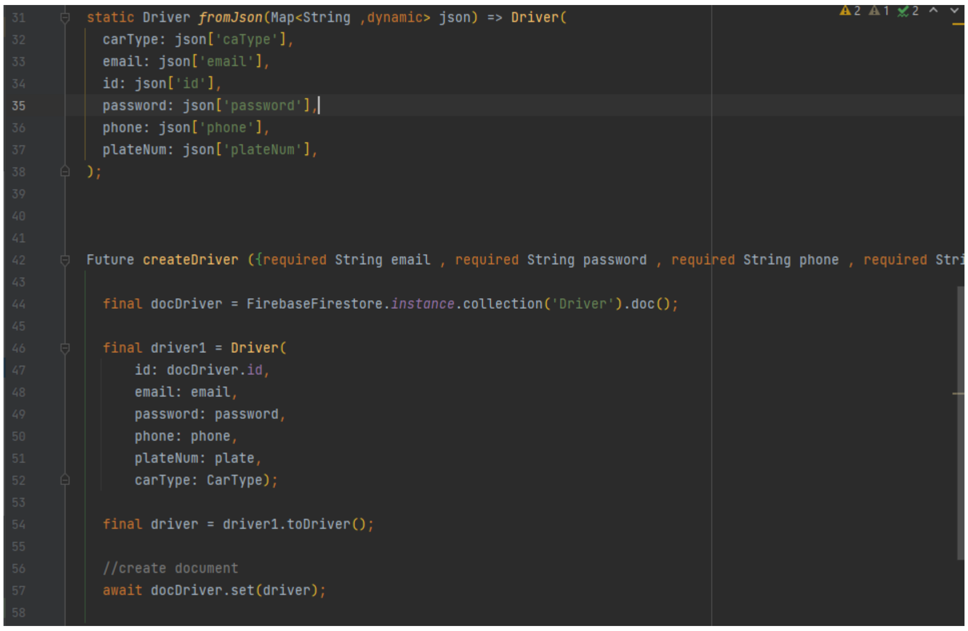
2.1.2 Driver Class

Driver class is shown below in Figure 2.4 and 2.5. Required constructors and attributes for the driver is shown in the code.



```
class Driver {
  String ?id;
  String ?email;
  String ?password;
  String ?phone;
  String ?plateNum;
  String ?carType;

  Driver({
    this.id = '',
    required this.email,
    required this.password,
    required this.phone,
    required this.plateNum,
    required this.carType,
  });

  Map<String , dynamic> toDriver()=>{
    'id' : id,
    'email' : email,
    'password': password,
    'phone'  : phone,
    'plateNum': plateNum,
    'carType' : carType,
  };
```

Figure 2.4: Driver Class Code Snippet 1

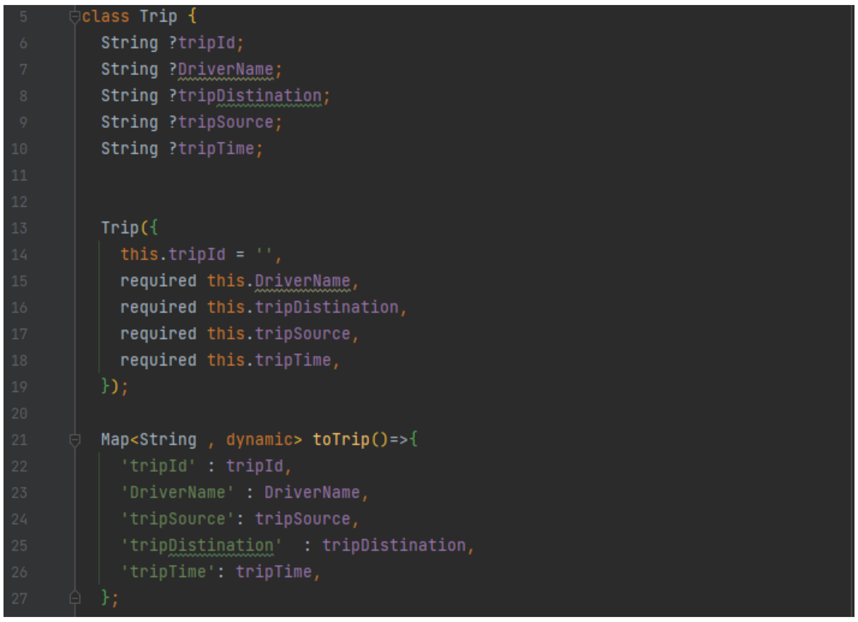


```
static Driver fromJson(Map<String ,dynamic> json) => Driver(
  carType: json['caType'],
  email: json['email'],
  id: json['id'],
  password: json['password'],
  phone: json['phone'],
  plateNum: json['plateNum'],
);



Future createDriver ({required String email , required String password , required String phone , required Stri

  final docDriver = FirebaseFirestore.instance.collection('Driver').doc();

  final driver1 = Driver(
      id: docDriver.id,
      email: email,
      password: password,
      phone: phone,
      plateNum: plate,
      carType: CarType);

  final driver = driver1.toDriver();

  //create document
  await docDriver.set(driver);

```

Figure 2.5: Driver Class Code Snippet 2

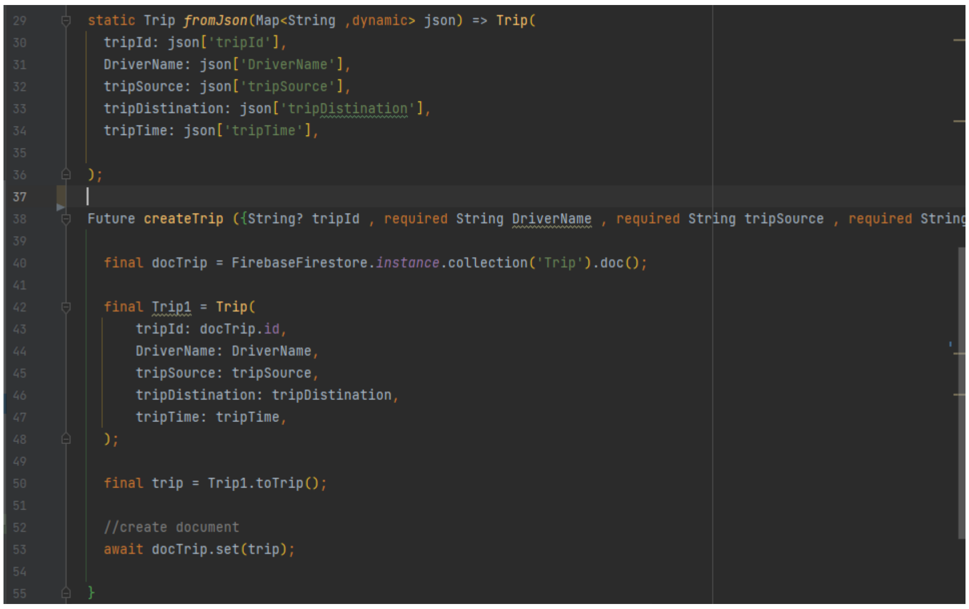
2.1.3 Trip Class

Trip class is shown below in Figure xx. Required constructors and attributes for the driver is shown in the code.



```
class Trip {
  String ?tripId;
  String ?DriverName;
  String ?tripDistination;
  String ?tripSource;
  String ?tripTime;


  Trip({
    this.tripId = '',
    required this.DriverName,
    required this.tripDistination,
    required this.tripSource,
    required this.tripTime,
  });

  Map<String , dynamic> toTrip()=>{
    'tripId' : tripId,
    'DriverName' : DriverName,
    'tripSource': tripSource,
    'tripDistination'  : tripDistination,
    'tripTime': tripTime,
  };
```

Figure 2.6: Trip Class Code Snippet 1



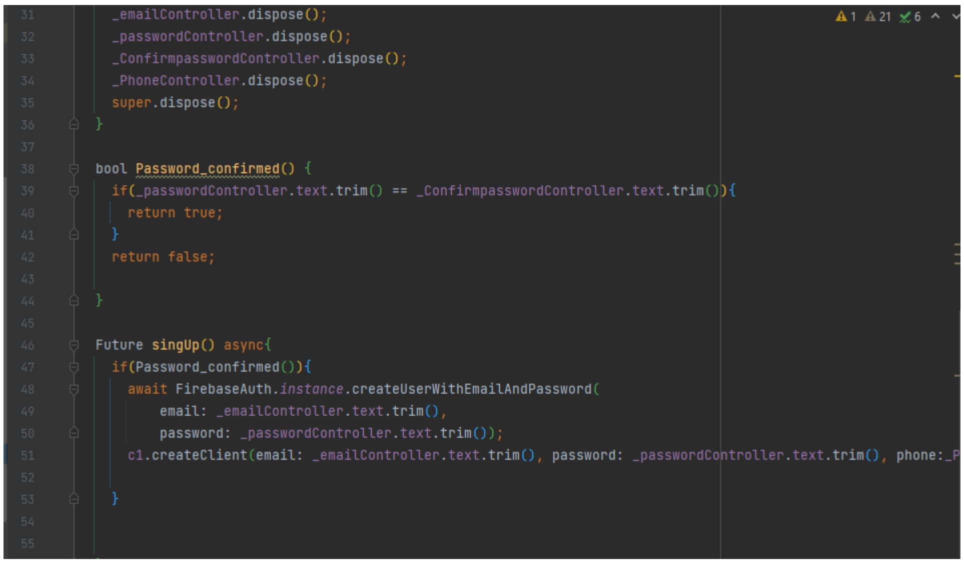
```
static Trip fromJson(Map<String ,dynamic> json) => Trip(
  tripId: json['tripId'],
  DriverName: json['DriverName'],
  tripSource: json['tripSource'],
  tripDistination: json['tripDistination'],
  tripTime: json['tripTime'],

);

Future createTrip ({String? tripId , required String DriverName , required String tripSource , required Strin

  final docTrip = FirebaseFirestore.instance.collection('Trip').doc();

  final Trip1 = Trip(
      tripId: docTrip.id,
      DriverName: DriverName,
      tripSource: tripSource,
      tripDistination: tripDistination,
      tripTime: tripTime,
  );

  final trip = Trip1.toTrip();

  //create document
  await docTrip.set(trip);

}
```

Figure 2.7: Trip Class Code Snippet 2

2.2 User Functionalities

2.2.1 Register and Login Pages Code and User Interface

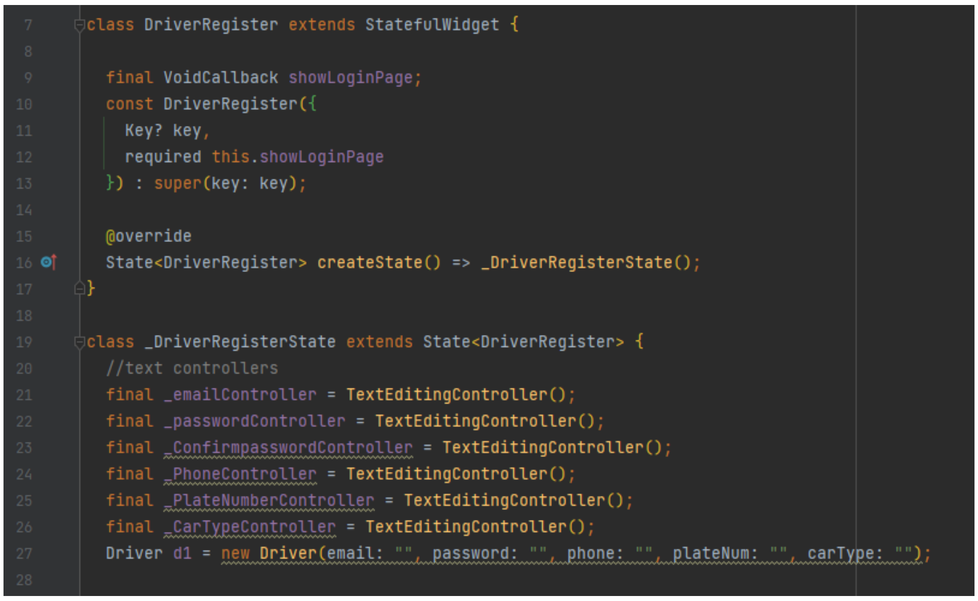
Figures 2.2 below shows a snippet of the inputs taken from the user for registration such as e-mail, password, phone number,..., and shows the authentication on the e-mail and password with the database in lines 46-53.



```
31      _emailController.dispose();
32      _passwordController.dispose();
33      _ConfirmpasswordController.dispose();
34      _PhoneController.dispose();
35      super.dispose();
36    }
37
38    bool Password_confirmed() {
39      if(_passwordController.text.trim() == _ConfirmpasswordController.text.trim()){
40        return true;
41      }
42      return false;
43
44    }
45
46    Future singUp() async{
47      if(Password_confirmed()){
48        await FirebaseAuth.instance.createUserWithEmailAndPassword(
49            email: _emailController.text.trim(),
50            password: _passwordController.text.trim());
51        c1.createClient(email: _emailController.text.trim(), password: _passwordController.text.trim(), phone:_P
52
53      }
54
55
```

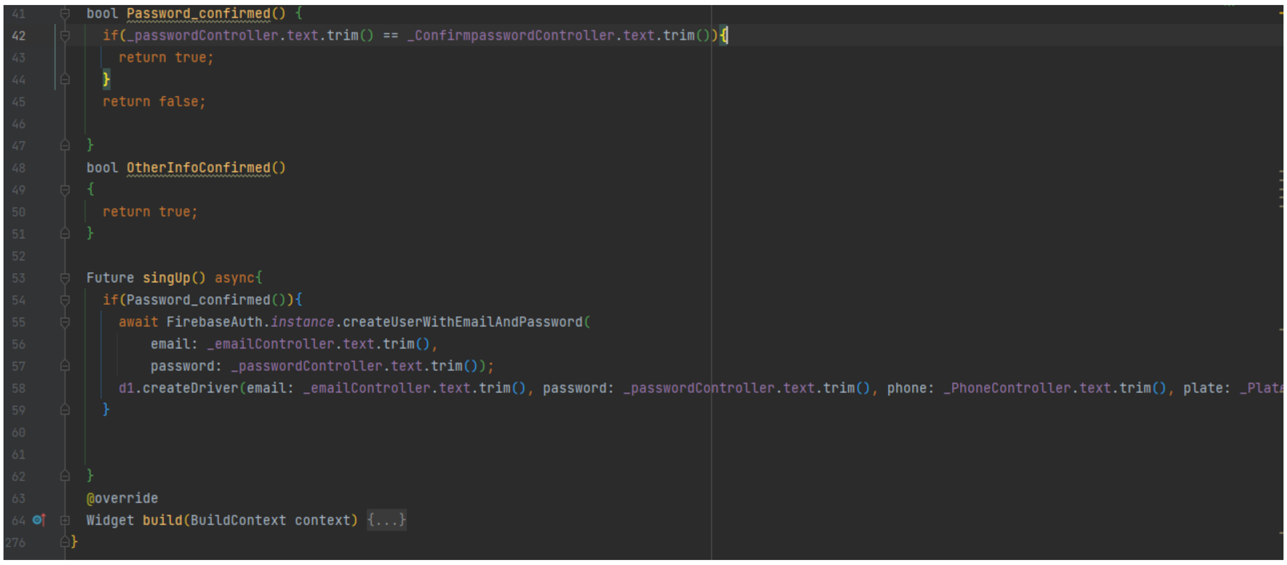
Figure 2.8: Register as a Client Code Snippet

In Figures 2.3 and 2.4, we show the driver registration inputs such as e-mail, password, phone number, and in addition, plate number, and car type for verification and security. The following lines in Figure 2.4 show the authentication on the e-mail and password, in addition to the building of the widget-screen- on the UI which can be referred to in more detail in [1].



```
class DriverRegister extends StatefulWidget {

  final VoidCallback showLoginPage;
  const DriverRegister({
    Key? key,
    required this.showLoginPage
  }) : super(key: key);

  @override
  State<DriverRegister> createState() => _DriverRegisterState();
}

class _DriverRegisterState extends State<DriverRegister> {
  //text controllers
  final _emailController = TextEditingController();
  final _passwordController = TextEditingController();
  final _ConfirmpasswordController = TextEditingController();
  final _PhoneController = TextEditingController();
  final _PlateNumberController = TextEditingController();
  final _CarTypeController = TextEditingController();
  Driver d1 = new Driver(email: "", password: "", phone: "", plateNum: "", carType: "");

```

Figure 2.9: Register as a Driver Code Snippet 1



```
bool Password_confirmed() {
  if(_passwordController.text.trim() == _ConfirmpasswordController.text.trim()){
    return true;
  }
  return false;

}
bool OtherInfoConfirmed()
{
  return true;
}

Future singUp() async{
  if(Password_confirmed()){
    await FirebaseAuth.instance.createUserWithEmailAndPassword(
        email: _emailController.text.trim(),
        password: _passwordController.text.trim());
    d1.createDriver(email: _emailController.text.trim(), password: _passwordController.text.trim(), phone: _PhoneController.text.trim(), plate: _Plat
  }


}
@override
Widget build(BuildContext context) {...}
}
```

Figure 2.10: Register as a Driver Code Snippet 1

The User Interface of Registration of User and Driver is shown below in Figure 2.5.

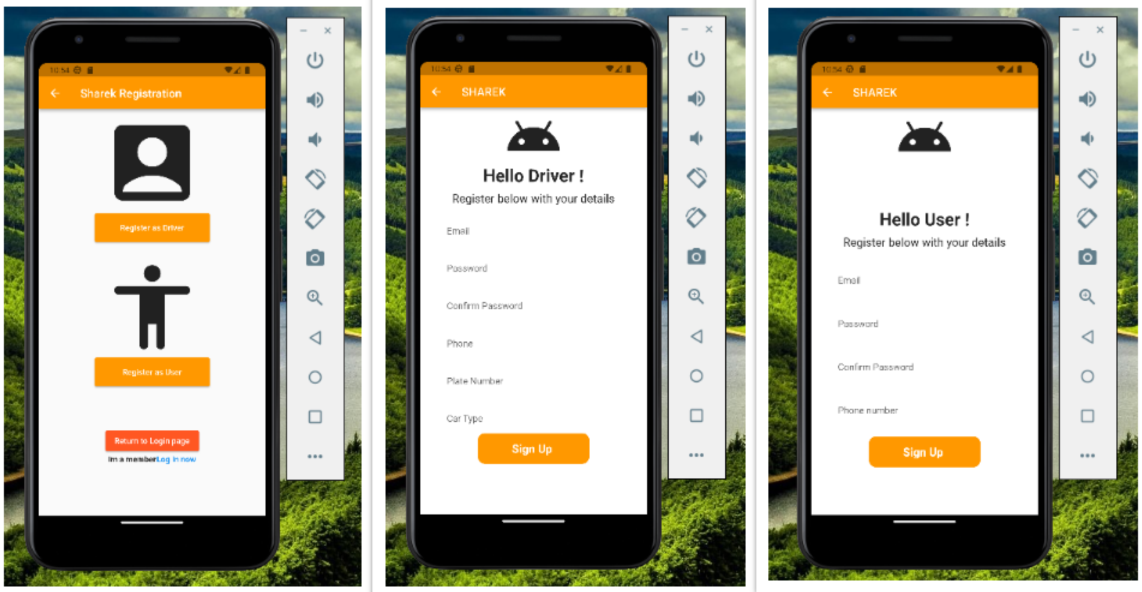


Figure 2.11: Registration User Interface

Login Page for user is shown in Figure 2.6. We try to login here with a normal user e-mail to show the functionalities present for users.

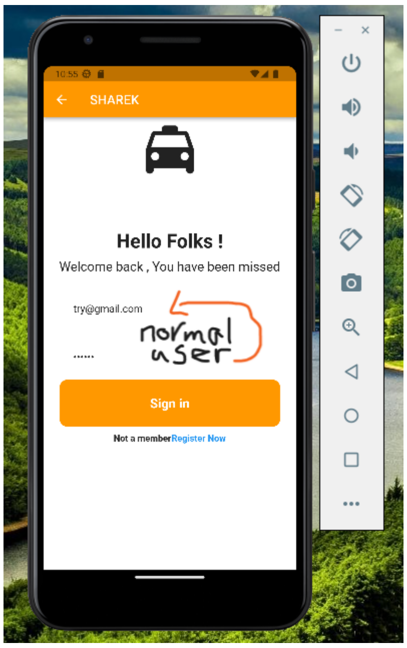

SHAREK
Hello Folks !
Welcome back , You have been missed
try@gmail.com
normal user
Sign in
Not a member Register Now

Figure 2.12: Login User Interface

2.2.2 Main Page User Interface

Figure 2.13 below show the options for users for the application. They can either list all available drivers, or list all the available trips so they can share in. There is also the option of adding a trip where other users can join in, and drivers can accept the offers.

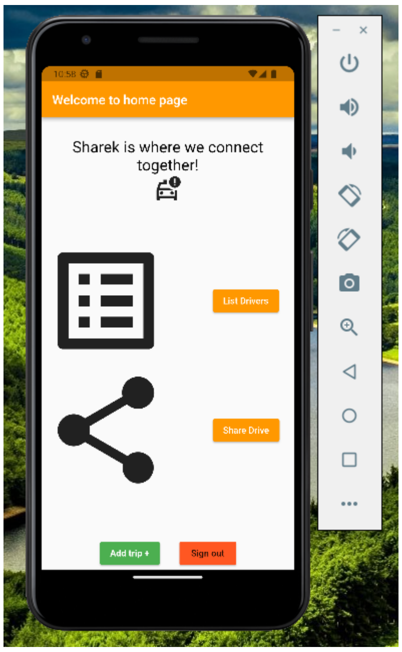
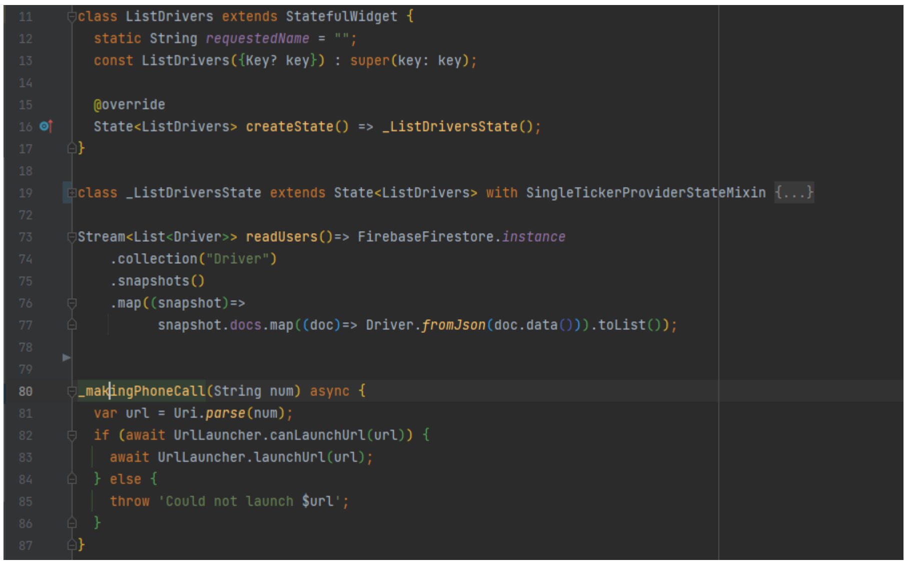


Figure 2.13: Main Page User Interface

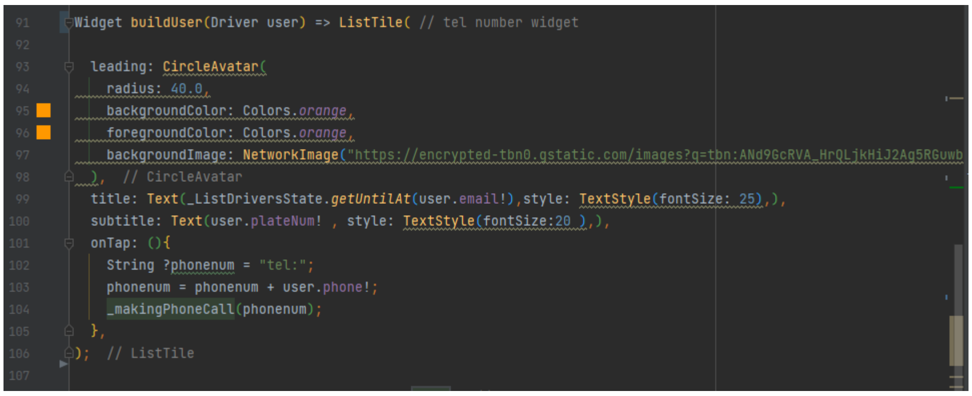
2.2.3 List Drivers Code and User Interface

The code in Figures 2.14 and 2.15 shows the steps implemented to fetch all the drivers from the cloud database. In Figure 2.14 we return a stream of Drivers in a list that contains all the information needed to be displayed. In addition, a user has the option to call the driver to agree on trip details and check their availability. This function is called in Figure 2.15 in widget creation.



```
class ListDrivers extends StatefulWidget {
  static String requestedName = "";
  const ListDrivers({Key? key}) : super(key: key);

  @override
  State<ListDrivers> createState() => _ListDriversState();
}

class _ListDriversState extends State<ListDrivers> with SingleTickerProviderStateMixin {...}

Stream<List<Driver>> readUsers()=> FirebaseFirestore.instance
    .collection("Driver")
    .snapshots()
    .map((snapshot)=>
          snapshot.docs.map((doc)=> Driver.fromJson(doc.data())).toList());


_makingPhoneCall(String num) async {
  var url = Uri.parse(num);
  if (await UrlLauncher.canLaunchUrl(url)) {
    await UrlLauncher.launchUrl(url);
  } else {
    throw 'Could not launch $url';
  }
}
```

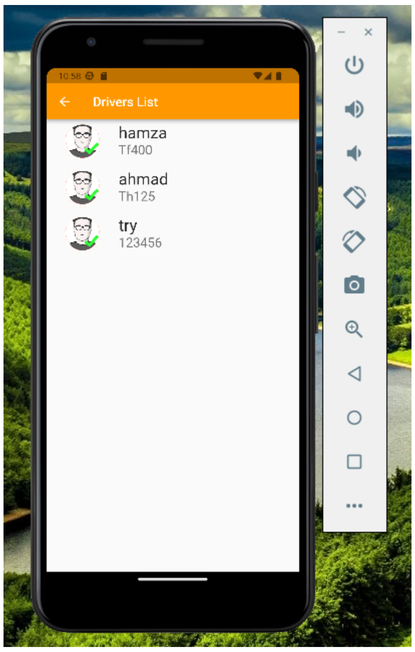
Figure 2.14: List Drivers Code Snippet 1



```
Widget buildUser(Driver user) => ListTile( // tel number widget

  leading: CircleAvatar(
    radius: 40.0,
    backgroundColor: Colors.orange,
    foregroundColor: Colors.orange,
    backgroundImage: NetworkImage("https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcRVA_HrQLjkHiJ2Ag5RGuwb
  ), // CircleAvatar
  title: Text(_ListDriversState.getUntilAt(user.email!),style: TextStyle(fontSize: 25),),
  subtitle: Text(user.plateNum! , style: TextStyle(fontSize:20 ),),
  onTap: (){
    String ?phonenum = "tel:";
    phonenum = phonenum + user.phone!;
    _makingPhoneCall(phonenum);
  },
); // ListTile
```

Figure 2.15: List Drivers Code Snippet 2

List Drivers User Interface is shown below in Figure 2.16.

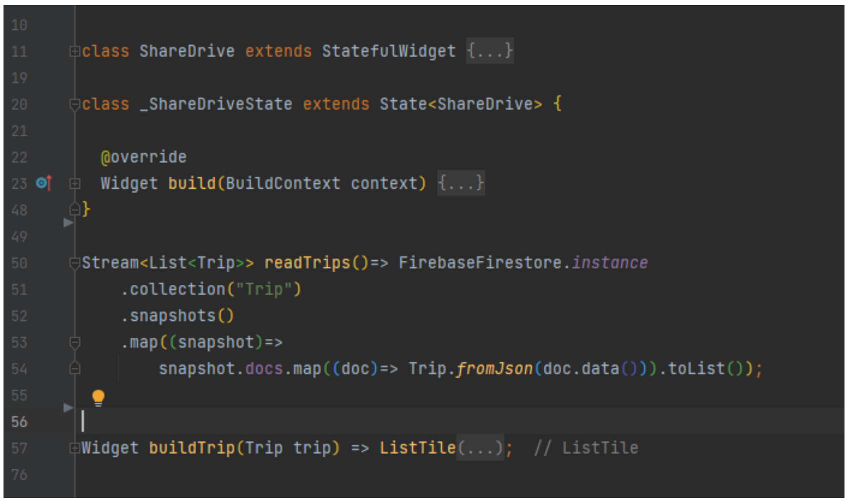


hamza
Tf400
ahmad
Th125
try
123456

Figure 2.16: List Drivers User Interface

2.2.4 List Trips Code and User Interface

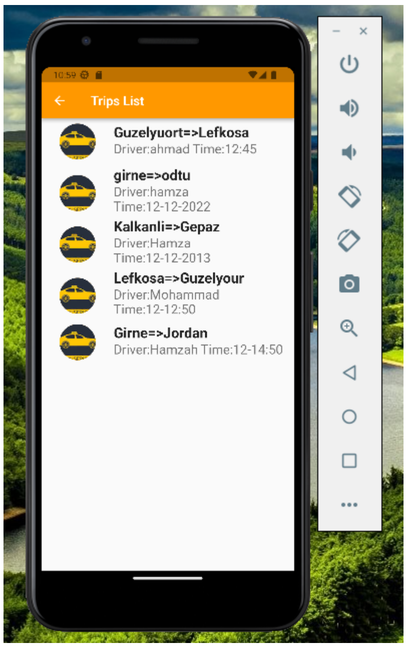
The code in Figure 2.17 shows the steps implemented to fetch all the available trips to be shared from the cloud database. In Figure 2.17 we return a stream of Trips in a list that contains all the information needed to be displayed. The stream is then used in the widget in line 57.



```
10
11  class ShareDrive extends StatefulWidget {...}
19
20  class _ShareDriveState extends State<ShareDrive> {
21
22    @override
23    Widget build(BuildContext context) {...}
48  }
49
50  Stream<List<Trip>> readTrips()=> FirebaseFirestore.instance
51      .collection("Trip")
52      .snapshots()
53      .map((snapshot)=>
54          snapshot.docs.map((doc)=> Trip.fromJson(doc.data())).toList());
55
56
57  Widget buildTrip(Trip trip) => ListTile(...);  // ListTile
76
```

Figure 2.17: Share Trip Code Snippet

Share Trips User Interface is shown below in Figure 2.18.



10:59

Trips List

Guzelyuort=>Lefkosa
Driver:ahmad Time:12:45

girne=>odtu
Driver:hamza
Time:12-12-2022

Kalkanli=>Gepaz
Driver:Hamza
Time:12-12-2013

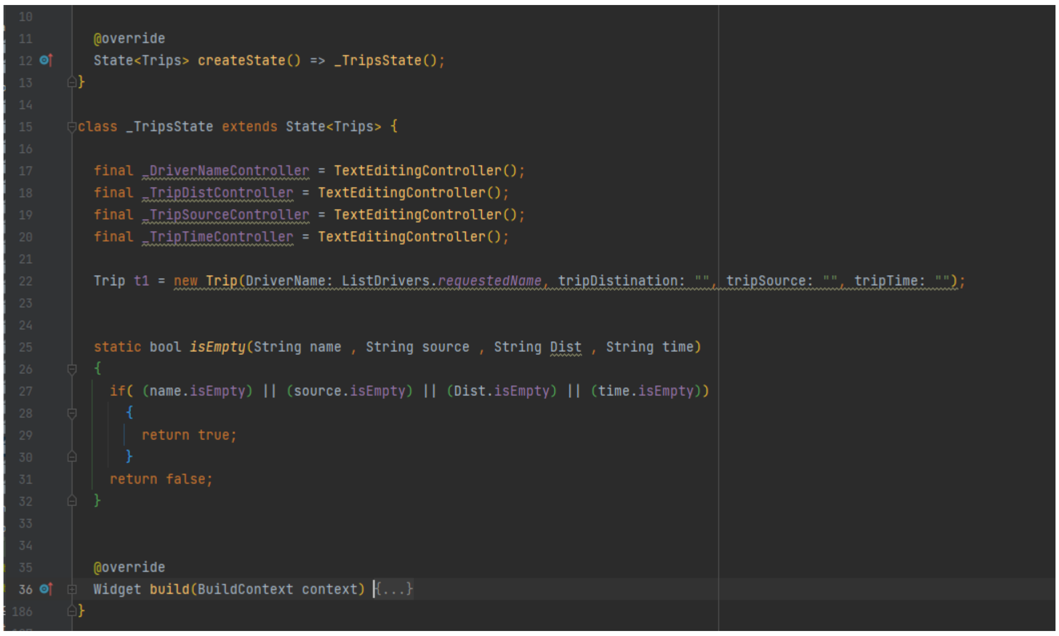
Lefkosa=>Guzelyour
Driver:Mohammad
Time:12-12:50

Girne=>Jordan
Driver:Hamzah Time:12-14:50

Figure 2.18: Share Trips User Interface

2.2.5 Add a Trip Code and User Interface

In Figure 2.19, we show the code snippet of Add a Trip functionality. A user has the option to add a trip, and to allow others users to share in, and drivers to accept them. Information needed are Driver Name, Trip Destination, Pick Up Location, and Trip Timing as shown below.



```

  @override
  State<Trips> createState() => _TripsState();
}

class _TripsState extends State<Trips> {

  final _DriverNameController = TextEditingController();
  final _TripDistController = TextEditingController();
  final _TripSourceController = TextEditingController();
  final _TripTimeController = TextEditingController();

  Trip t1 = new Trip(DriverName: ListDrivers.requestedName, tripDistination: "", tripSource: "", tripTime: "");


  static bool isEmpty(String name , String source , String Dist , String time)
  {
    if( (name.isEmpty) || (source.isEmpty) || (Dist.isEmpty) || (time.isEmpty))
      {
        return true;
      }
    return false;
  }


  @override
  Widget build(BuildContext context) {...}
}
```

Figure 2.19: Add a Trip Code Snippet

The user interface for this functionality is shown below in Figure 2.20.

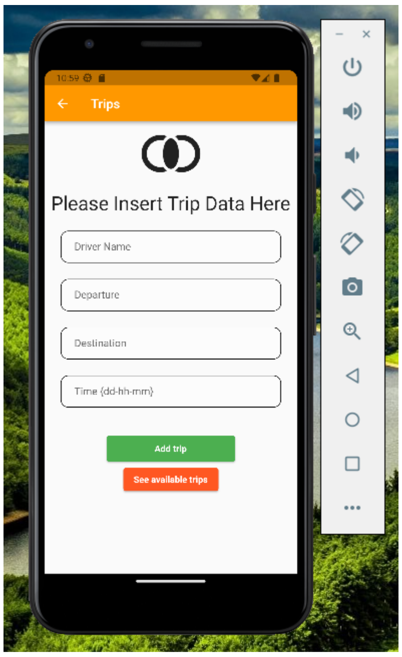
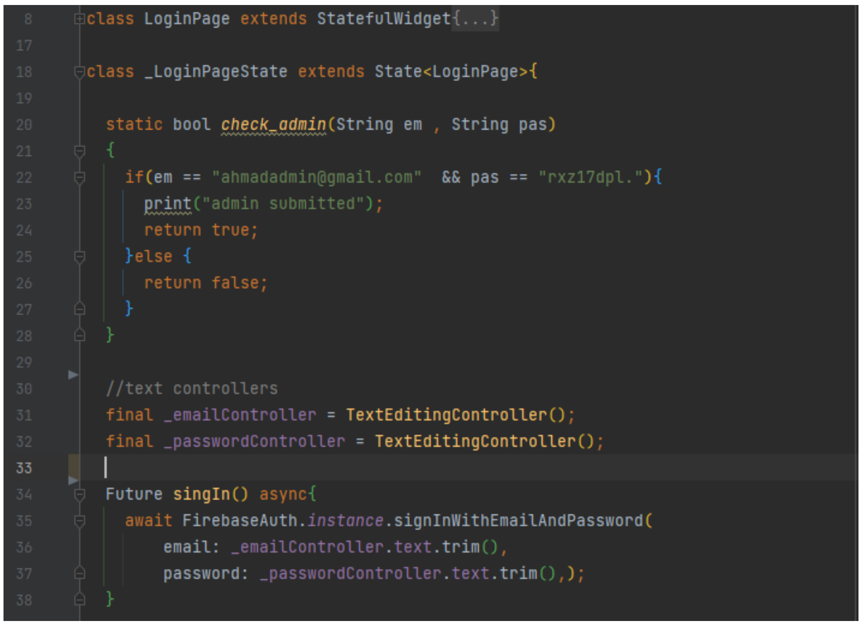


Figure 2.20: Add a Trip User Interface

2.3 Admin Functionalities

2.3.1 Admin Login Code and User Interface

In Figure 2.21, we see the code of admin verification for admin. We assume that we have one admin with the e-mail "ahmadadmin@gmail.com".



```
class LoginPage extends StatefulWidget{...}

class _LoginPageState extends State<LoginPage>{

  static bool check_admin(String em , String pas)
  {
    if(em == "ahmadadmin@gmail.com"  && pas == "rxz17dpl."){
      print("admin submitted");
      return true;
    }else {
      return false;
    }
  }

  //text controllers
  final _emailController = TextEditingController();
  final _passwordController = TextEditingController();

  Future singIn() async{
    await FirebaseAuth.instance.signInWithEmailAndPassword(
        email: _emailController.text.trim(),
        password: _passwordController.text.trim(),);
  }
```

Figure 2.21: Admin Login User Interface

In Figure 2.22, we try and login with the admin e-mail ”ahmadadmin@gmail.com” to show the admin functionalities.

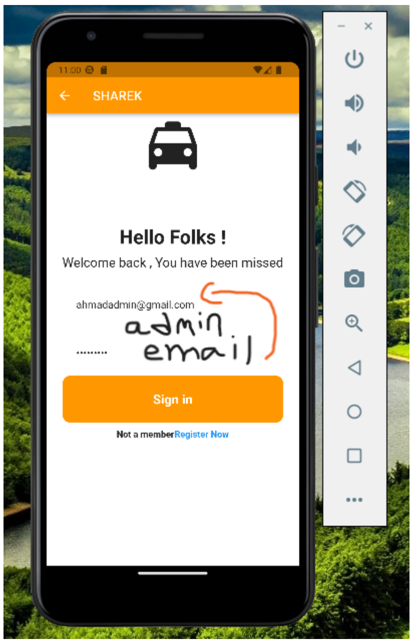

SHAREK
Hello Folks !
Welcome back , You have been missed
ahmadadmin@gmail.com
admin email
Sign in
Not a member Register Now

Figure 2.22: Admin Login User Interface

2.3.2 Admin Menu User Interface

In Figure 2.23, we display the list of functionalities available for the admin. Initializing the database adds a 100 client records to start the program. Reset Database deletes all the records in the database. View Database allows the admin to list all the Clients, Drivers, and Trips. Modify Database gives the admin the option to modify some information about Clients, Drivers, and Trips -will be demonstrated below-. Backup Database saves all the information in Client, Driver, and Trip tables locally into .txt files -will be demonstrated below-.

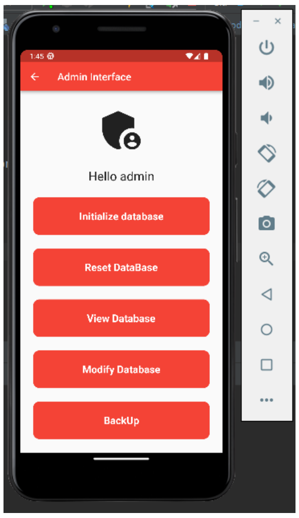
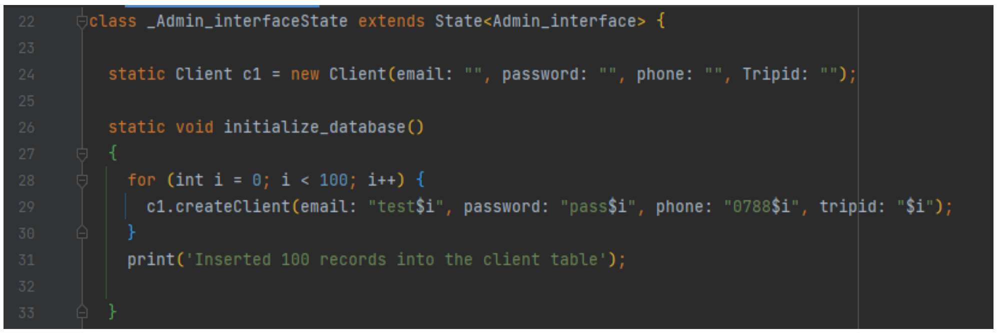


Figure 2.23: Admin Menu User Interface

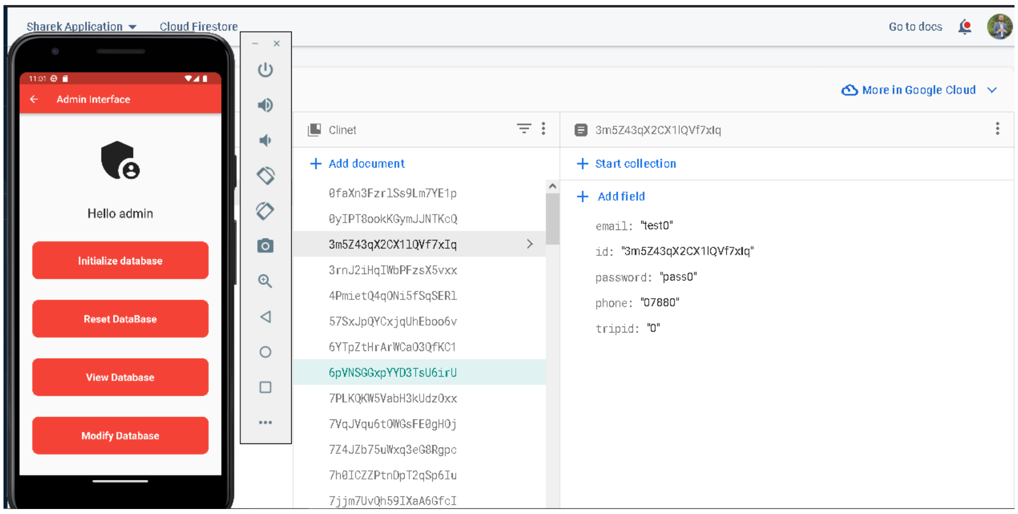
2.3.3 Initialize Database Code and Database Effect

In Figure 2.24, random clients are created with random values sent with the constructor to initialize the database with some values. Figure 2.25 shows the effect of addition on the cloud database.



```
class _Admin_interfaceState extends State<Admin_interface> {

  static Client c1 = new Client(email: "", password: "", phone: "", Tripid: "");

  static void initialize_database()
  {
    for (int i = 0; i < 100; i++) {
      c1.createClient(email: "test$i", password: "pass$i", phone: "0788$i", tripid: "$i");
    }
    print('Inserted 100 records into the client table');

  }
```

Figure 2.24: Initialize Database Code Snippet



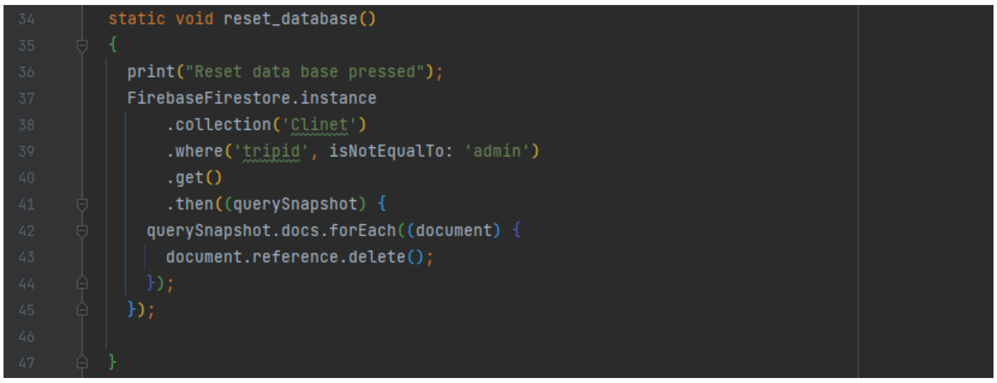
```
0faXn3Fzr1Ss9Lm7YE1p
0yIPT8ookKGymJJNTKcQ
3m5Z43qX2CX1lQVf7xIq
3rnJ2iHqIWbPFzsX5vxx
4PmietQ4qONi5fSqSERl
57SxJpQYCxjqUhEboo6v
6YTpZtHrArWCaO3QfKC1
6pVNSGGxpYYD3TsU6irU
7PLKQKW5VabH3kUdzOxx
7VqJVqu6tOWGsFE0gH0j
7Z4JZb75uWxq3eG8Rgpc
7h0ICZZPtnDpT2qSp6Iu
7jjm7UvQh59IXaA6GfcI
```

```
3m5Z43qX2CX1lQVf7xlq
email: "test0"
id: "3m5Z43qX2CX1lQVf7xlq"
password: "pass0"
phone: "07880"
tripid: "0"
```

Figure 2.25: Initialize Database User Interface

2.3.4 Reset Database Code and Database Effect

In Figure 2.26, we show the code for resetting the database, here we show the client records being deleted. Note that we do not delete the admin record. In Figure 2.27, we see the effect of this function being called. Note that the only record left is the admin record as seen.



```
static void reset_database()
{
  print("Reset data base pressed");
  FirebaseFirestore.instance
      .collection('Clinet')
      .where('tripid', isNotEqualTo: 'admin')
      .get()
      .then((querySnapshot) {
    querySnapshot.docs.forEach((document) {
      document.reference.delete();
    });
  });

}
```

Figure 2.26: Reset Database Code Snippet

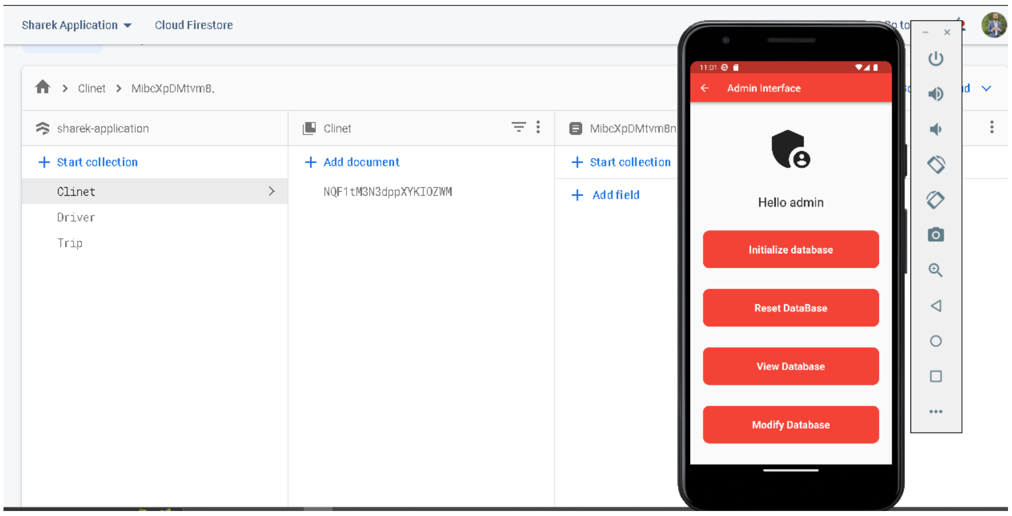
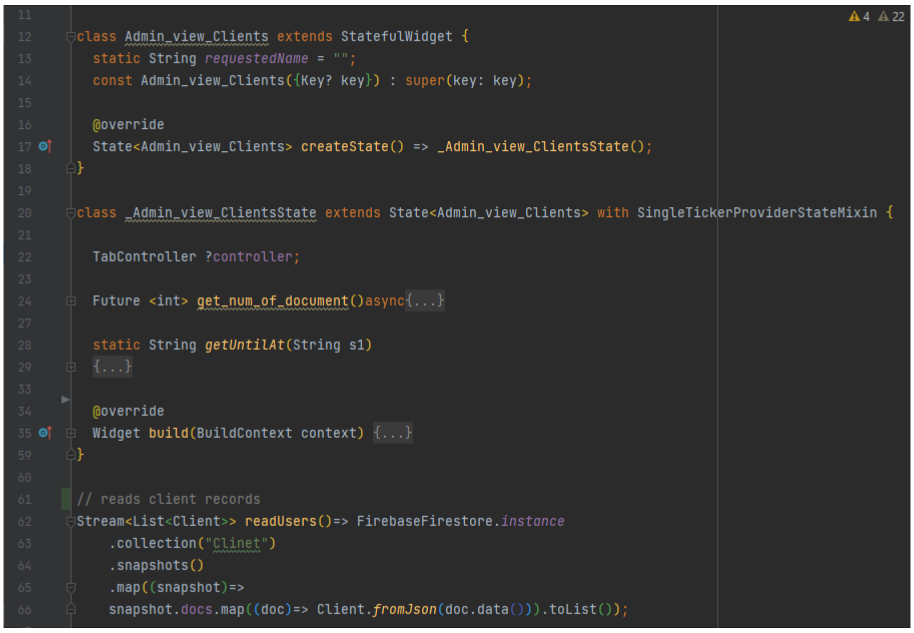


Figure 2.27: Reset Database User Interface

2.3.5 View Database Code and User Interface

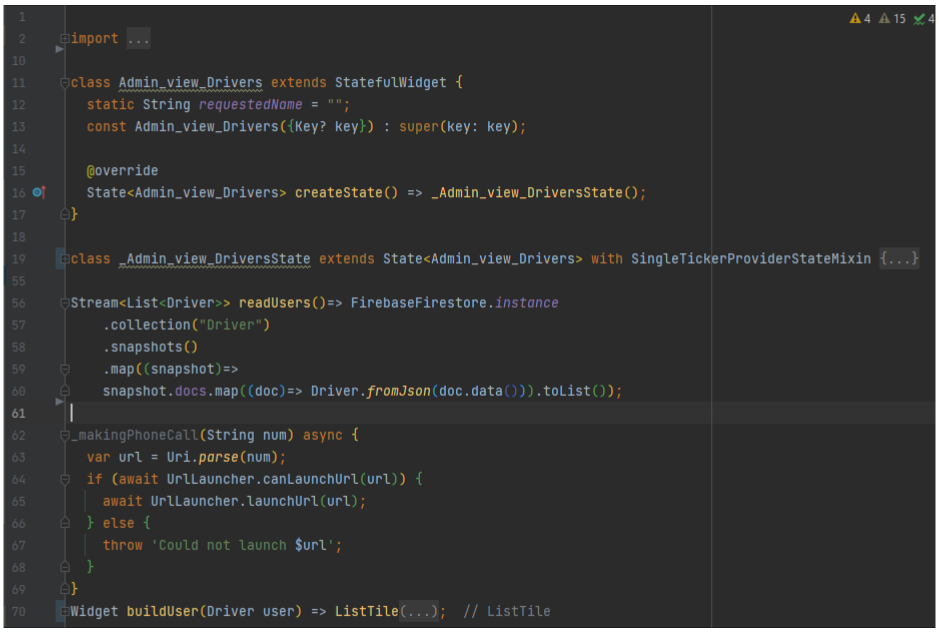
Figure 2.28 shows the fetching of all Client records in lines 62-66 and returning them as a stream of Clients to be displayed in a widget.



```
11
12  class Admin_view_Clients extends StatefulWidget {
13    static String requestedName = "";
14    const Admin_view_Clients({Key? key}) : super(key: key);
15
16    @override
17    State<Admin_view_Clients> createState() => _Admin_view_ClientsState();
18  }
19
20  class _Admin_view_ClientsState extends State<Admin_view_Clients> with SingleTickerProviderStateMixin {
21
22    TabController ?controller;
23
24    Future <int> get_num_of_document()async{...}
27
28    static String getUntilAt(String s1)
29    {...}
33
34    @override
35    Widget build(BuildContext context) {...}
59  }
60
61  // reads client records
62  Stream<List<Client>> readUsers()=> FirebaseFirestore.instance
63      .collection("Clinet")
64      .snapshots()
65      .map((snapshot)=>
66      snapshot.docs.map((doc)=> Client.fromJson(doc.data())).toList());
```

Figure 2.28: View Clients Code Snippet

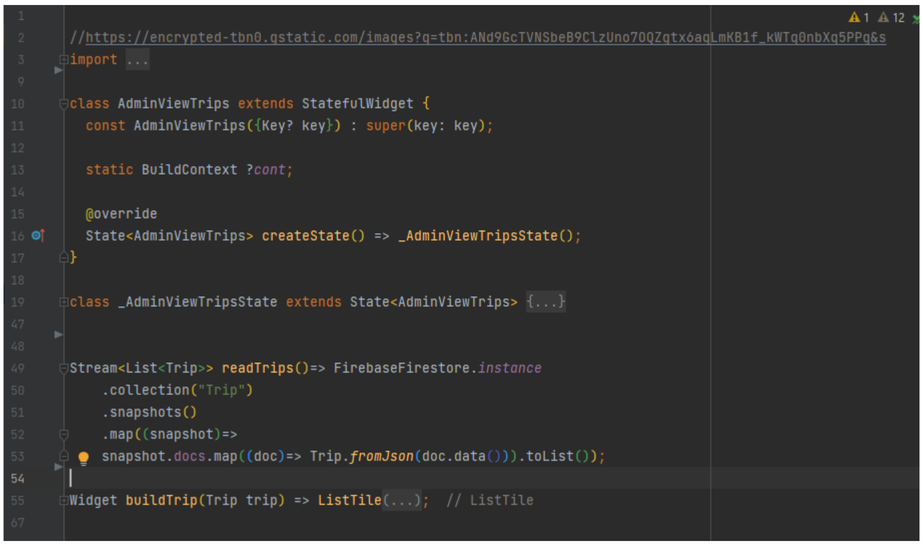
Figure 2.29 shows the fetching of all Driver records in lines 56-60 and returning them as a stream of Drivers to be displayed in a widget.



```
1
2   import ...
10
11  class Admin_view_Drivers extends StatefulWidget {
12    static String requestedName = "";
13    const Admin_view_Drivers({Key? key}) : super(key: key);
14
15    @override
16    State<Admin_view_Drivers> createState() => _Admin_view_DriversState();
17  }
18
19  class _Admin_view_DriversState extends State<Admin_view_Drivers> with SingleTickerProviderStateMixin {...}
55
56  Stream<List<Driver>> readUsers()=> FirebaseFirestore.instance
57      .collection("Driver")
58      .snapshots()
59      .map((snapshot)=>
60      snapshot.docs.map((doc)=> Driver.fromJson(doc.data())).toList());
61
62  _makingPhoneCall(String num) async {
63    var url = Uri.parse(num);
64    if (await UrlLauncher.canLaunchUrl(url)) {
65      await UrlLauncher.launchUrl(url);
66    } else {
67      throw 'Could not launch $url';
68    }
69  }
70  Widget buildUser(Driver user) => ListTile(...);  // ListTile
```

Figure 2.29: View Drivers Code Snippet

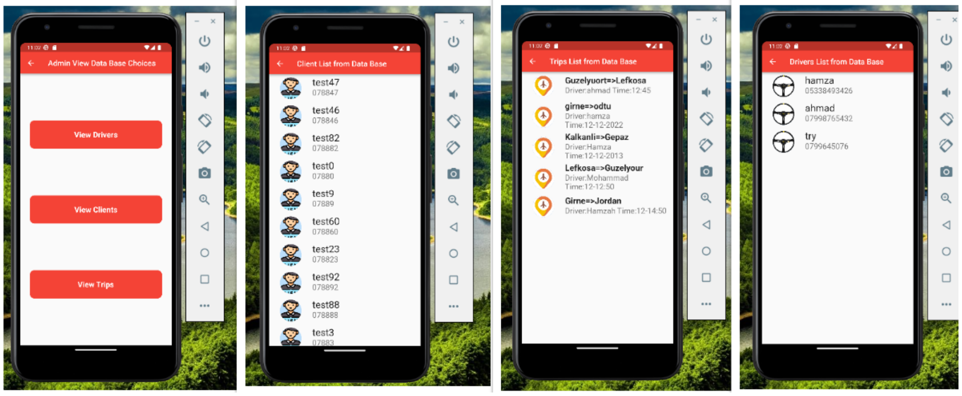
Figure 2.30 shows the fetching of all Trip records in lines 49-53 and returning them as a stream of Trips to be displayed in a widget.



```
1
2   //https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTVNSbeB9ClzUno70QZgtx6aqLmKB1f_kWTq0nbXq5PPg&s
3   import ...
9
10  class AdminViewTrips extends StatefulWidget {
11    const AdminViewTrips({Key? key}) : super(key: key);
12
13    static BuildContext ?cont;
14
15    @override
16    State<AdminViewTrips> createState() => _AdminViewTripsState();
17  }
18
19  class _AdminViewTripsState extends State<AdminViewTrips> {...}
47
48
49  Stream<List<Trip>> readTrips()=> FirebaseFirestore.instance
50      .collection("Trip")
51      .snapshots()
52      .map((snapshot)=>
53      snapshot.docs.map((doc)=> Trip.fromJson(doc.data())).toList());
54
55  Widget buildTrip(Trip trip) => ListTile(...);  // ListTile
67
```

Figure 2.30: View Trips Code Snippet

In Figure 2.31, admin has the option to view a list of all clients, drivers, or trips.


Admin View Data Base Choices
View Drivers
View Clients
View Trips
Client List from Data Base
Trips List from Data Base
Drivers List from Data Base

test47
078847
test46
078846
test82
078882
test0
07880
test9
07889
test60
078860
test23
078823
test92
078892
test88
078888
test3
07883

Guzelyuort=>Lefkosa
Driver:ahmad Time:12:45
girne=>odtu
Driver:hamza
Time:12-12-2022
Kalkanli=>Gepaz
Driver:Hamza
Time:12-12-2013
Lefkosa=>Guzelyour
Driver:Mohammad
Time:12-12:50
Girne=>Jordan
Driver:Hamzah Time:12-14:50

hamza
05338493426
ahmad
07998765432
try
0799645076

Figure 2.31: View Database User Interface

2.3.6 Modify Database Code and Database Effect

Figure 2.32, shows the options for admin to modify Clients, Drivers, and Trips information.

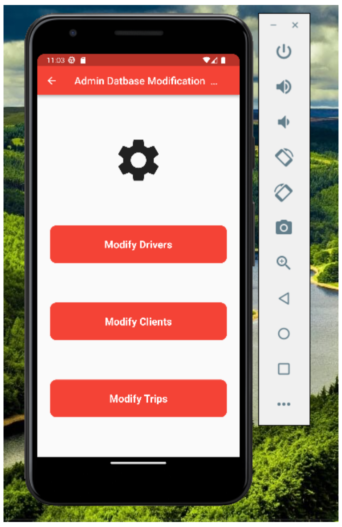
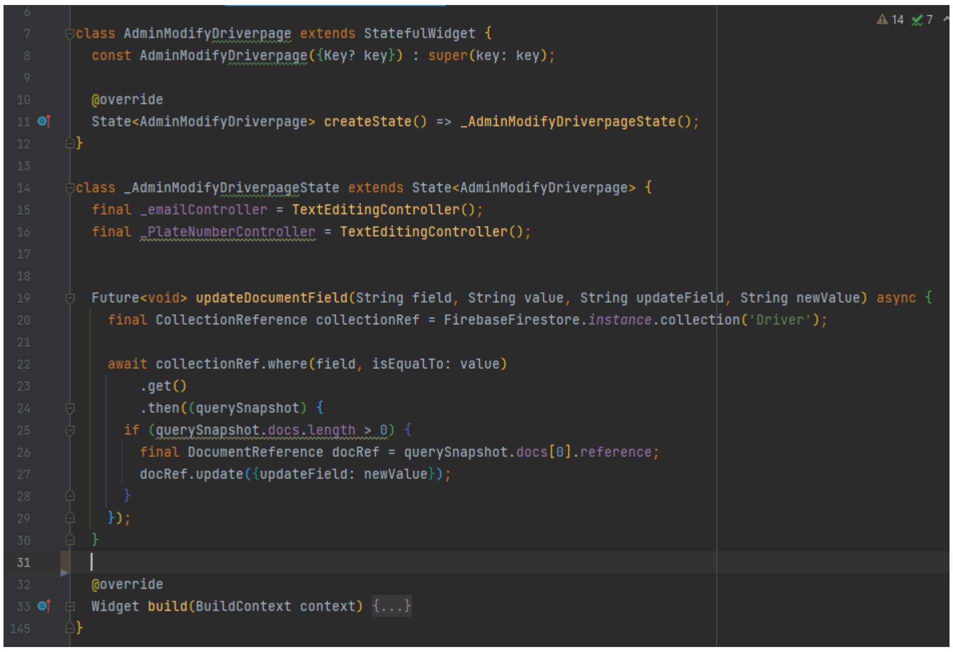


Figure 2.32: Modify Database User Interface

Modify Driver Code and User Interface

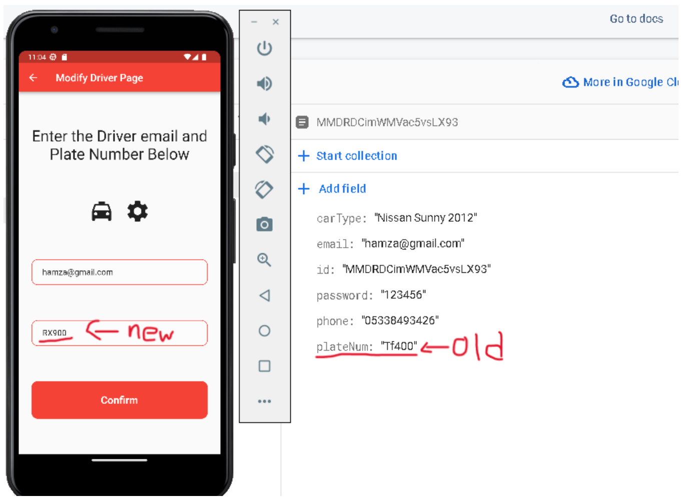
In Figure 2.33, we show in lines 19-30 the process of authenticating the entered e-mail, and updating the entered plate number.



```
6
7   class AdminModifyDriverpage extends StatefulWidget {
8     const AdminModifyDriverpage({Key? key}) : super(key: key);
9
10    @override
11    State<AdminModifyDriverpage> createState() => _AdminModifyDriverpageState();
12  }
13
14  class _AdminModifyDriverpageState extends State<AdminModifyDriverpage> {
15    final _emailController = TextEditingController();
16    final _PlateNumberController = TextEditingController();
17
18
19    Future<void> updateDocumentField(String field, String value, String updateField, String newValue) async {
20      final CollectionReference collectionRef = FirebaseFirestore.instance.collection('Driver');
21
22      await collectionRef.where(field, isEqualTo: value)
23          .get()
24          .then((querySnapshot) {
25        if (querySnapshot.docs.length > 0) {
26          final DocumentReference docRef = querySnapshot.docs[0].reference;
27          docRef.update({updateField: newValue});
28        }
29      });
30    }
31
32    @override
33    Widget build(BuildContext context) {...}
145 }
```

Figure 2.33: Modify Driver Code Snippet

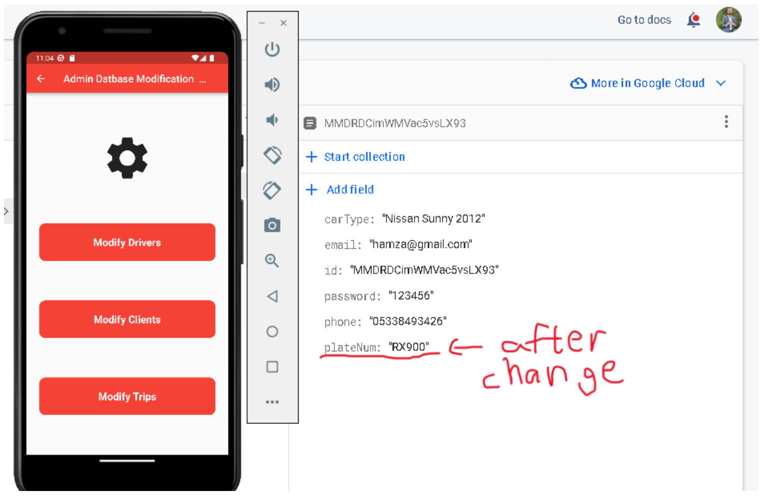
Figure 2.34 shows the old plate number for a user before being updated. Figure 2.35 shows the new updated plate number taking affect in the database.



MMDRDCimWMVac5vsLX93

```
carType: "Nissan Sunny 2012"
email: "hamza@gmail.com"
id: "MMDRDCimWMVac5vsLX93"
password: "123456"
phone: "05338493426"
plateNum: "Tf400"
```

Figure 2.34: Modify Driver User Interface Before Changes



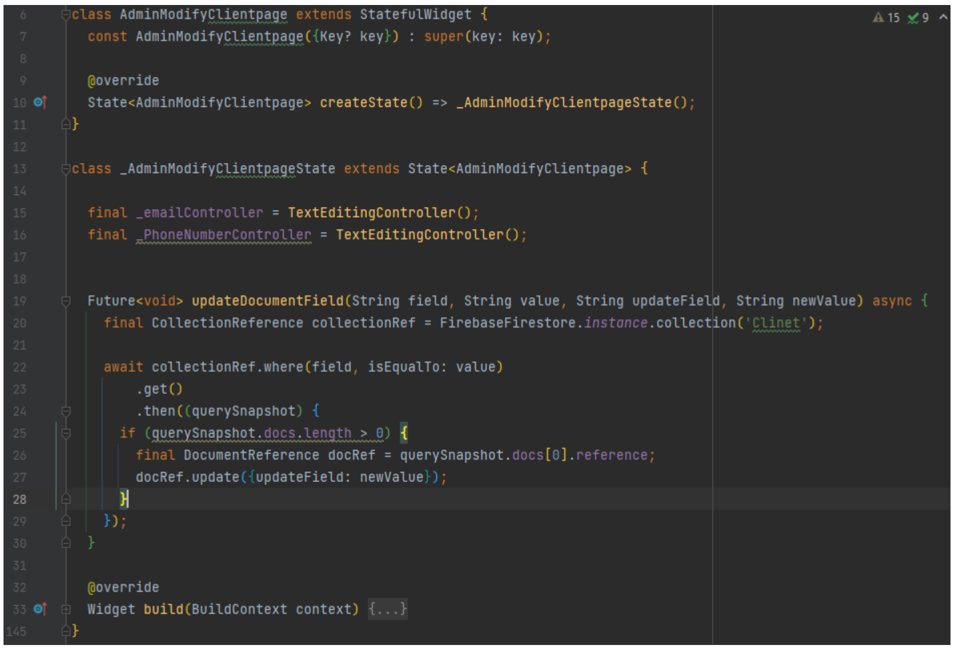
MMDRDCimWMVac5vsLX93

```
carType: "Nissan Sunny 2012"
email: "hamza@gmail.com"
id: "MMDRDCimWMVac5vsLX93"
password: "123456"
phone: "05338493426"
plateNum: "RX900"
```

Figure 2.35: Modify Driver User Interface After Changes

Modify Client Code and User Interface

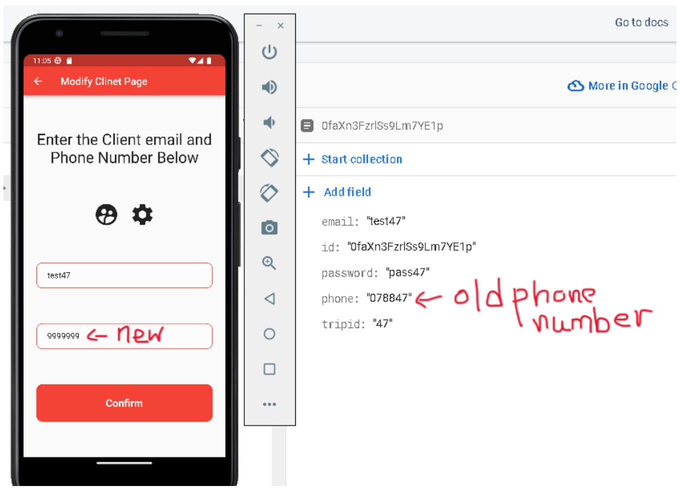
In Figure 2.36, we show in lines 19-30 the process of authenticating the entered e-mail, and updating the entered phone number.



```
6   class AdminModifyClientpage extends StatefulWidget {
7     const AdminModifyClientpage({Key? key}) : super(key: key);
8
9     @override
10    State<AdminModifyClientpage> createState() => _AdminModifyClientpageState();
11  }
12
13  class _AdminModifyClientpageState extends State<AdminModifyClientpage> {
14
15    final _emailController = TextEditingController();
16    final _PhoneNumberController = TextEditingController();
17
18
19    Future<void> updateDocumentField(String field, String value, String updateField, String newValue) async {
20      final CollectionReference collectionRef = FirebaseFirestore.instance.collection('Clinet');
21
22      await collectionRef.where(field, isEqualTo: value)
23          .get()
24          .then((querySnapshot) {
25        if (querySnapshot.docs.length > 0) {
26          final DocumentReference docRef = querySnapshot.docs[0].reference;
27          docRef.update({updateField: newValue});
28        }
29      });
30    }
31
32    @override
33    Widget build(BuildContext context) {...}
145 }
```

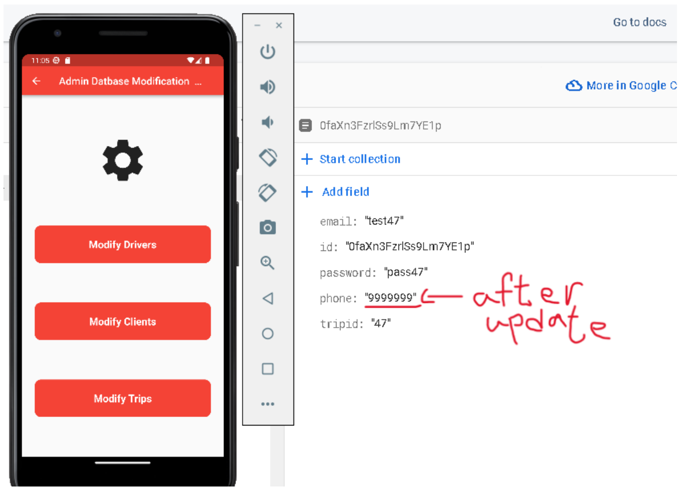
Figure 2.36: Modify Client Code Snippet

Figure 2.37 shows the old phone number for a user before being updated. Figure 2.38 shows the new updated phone number taking affect in the database.



```
test47
9999999

0faXn3FzrlSs9Lm7YE1p
email: "test47"
id: "0faXn3FzrlSs9Lm7YE1p"
password: "pass47"
phone: "078847"
tripid: "47"
```

Figure 2.37: Modify Client User Interface Before Changes

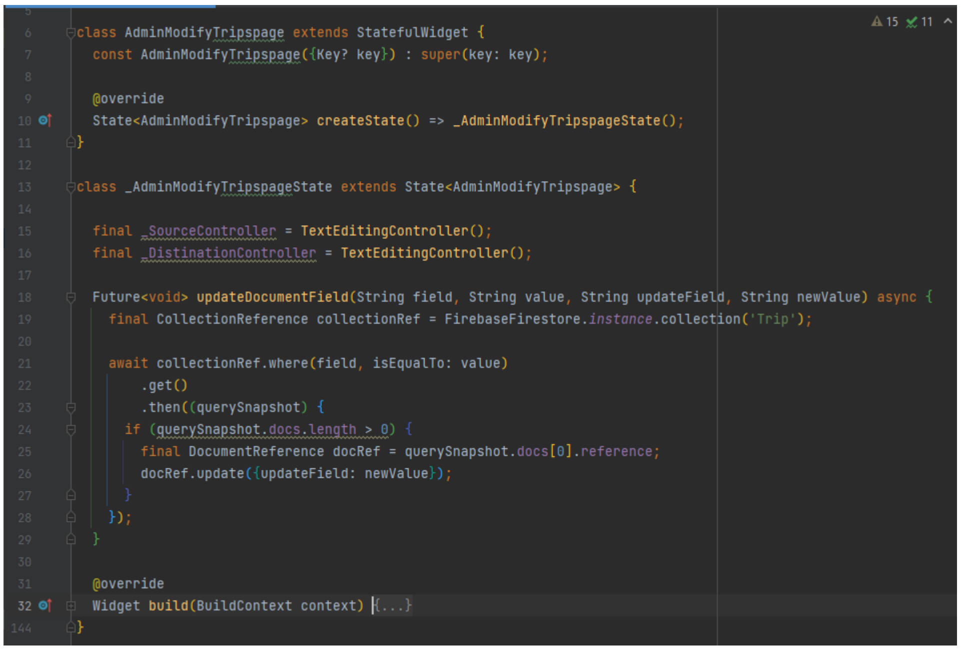


```
email: "test47"
id: "0faXn3FzrlSs9Lm7YE1p"
password: "pass47"
phone: "9999999"
tripid: "47"
```

Figure 2.38: Modify Client User Interface After Changes

Modify Trip Code and User Interface

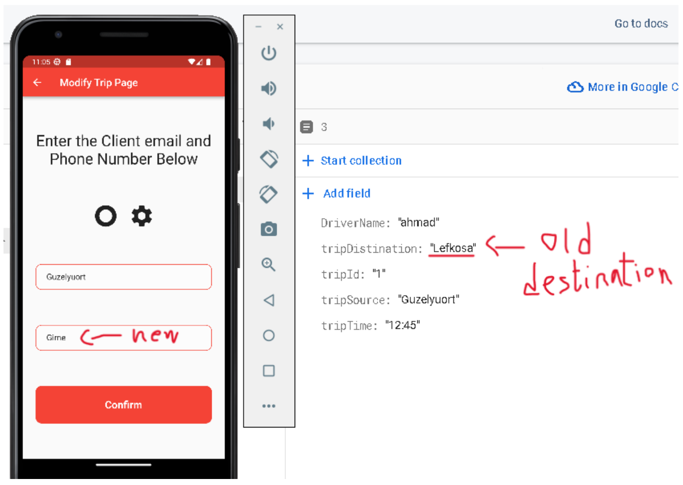
In Figure 2.39, we show in lines 18-29 the process of authenticating the entered e-mail, and updating the entered trip destination.



```
6   class AdminModifyTripspage extends StatefulWidget {
7     const AdminModifyTripspage({Key? key}) : super(key: key);
8
9     @override
10    State<AdminModifyTripspage> createState() => _AdminModifyTripspageState();
11  }
12
13  class _AdminModifyTripspageState extends State<AdminModifyTripspage> {
14
15    final _SourceController = TextEditingController();
16    final _DistinationController = TextEditingController();
17
18    Future<void> updateDocumentField(String field, String value, String updateField, String newValue) async {
19      final CollectionReference collectionRef = FirebaseFirestore.instance.collection('Trip');
20
21      await collectionRef.where(field, isEqualTo: value)
22          .get()
23          .then((querySnapshot) {
24        if (querySnapshot.docs.length > 0) {
25          final DocumentReference docRef = querySnapshot.docs[0].reference;
26          docRef.update({updateField: newValue});
27        }
28      });
29    }
30
31    @override
32    Widget build(BuildContext context) {...}
144 }
```

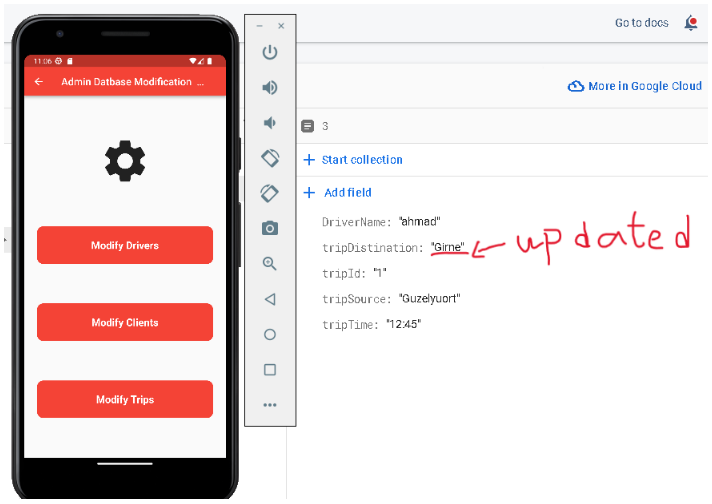
Figure 2.39: Modify Trip Code Snippet

Figure 2.40 shows the old trip destination for a user before being updated. Figure 2.41 shows the new updated trip destination taking affect in the database.



DriverName: "ahmad"
tripDistination: "Lefkosa"
tripId: "1"
tripSource: "Guzelyuort"
tripTime: "12:45"

Figure 2.40: Modify Trip User Interface Before Changes

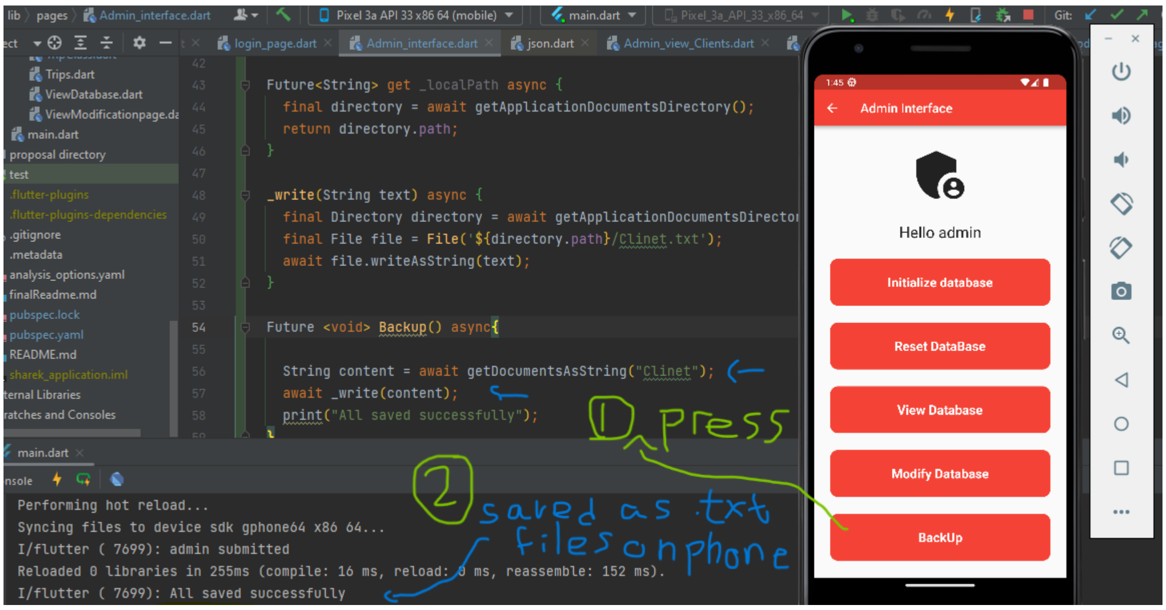


DriverName: "ahmad"
tripDistination: "Girne"
tripId: "1"
tripSource: "Guzelyuort"
tripTime: "12:45"

Figure 2.41: Modify Trip User Interface After Changes

2.3.7 Back up Database Code and Effect

In this subsection, we mention how admin can safely backup the database by pressing the Backup button in then menu. Once pressed, the database tables will be accessed and saved in separate .txt files on the admin's phone for extra reliability. Figure 2.42 demonstrates the process.



```dart
  Future<String> get _localPath async {
    final directory = await getApplicationDocumentsDirectory();
    return directory.path;
  }

  _write(String text) async {
    final Directory directory = await getApplicationDocumentsDirector
    final File file = File('${directory.path}/Clinet.txt');
    await file.writeAsString(text);
  }

  Future <void> Backup() async{

    String content = await getDocumentsAsString("Clinet");
    await _write(content);
    print("All saved successfully");
  }
```

```
Performing hot reload...
Syncing files to device sdk gphone64 x86 64...
I/flutter ( 7699): admin submitted
Reloaded 0 libraries in 255ms (compile: 16 ms, reload: 0 ms, reassemble: 152 ms).
I/flutter ( 7699): All saved successfully
```

Figure 2.42: Backup Database Effect

2.4 Project Cloud Services and Technologies

2.4.1 Firebase

Firebase is a back end platform for building Web, Android and IOS applications. It offers real-time database, different APIs, multiple authentication types and a hosting platform [2].

2.4.2 List of Services and Technologies Used

For all the project, the following services and technologies were used.

Cloud Service:	▪	Google's Fire Base Cloud Service
Database type:	▪	No SQL
Features Provided:	1)	It is a real-time database. Changes in the application code are reflected directly in the cloud database.
	2)	Authentication is available on the parameter chosen, in our case, the user password.
	3)	A dashboard is available for ease of use and configuration
Programming Languages used:	▪	Dart
API's and Frameworks:	▪	Flutter

Figure 2.43: Services and Technologies Table

Chapter 3

Project Statistics

3.1 Time Frames

3.1.1 October 31 - November 6

Parts Completed

main.dart page was completed

Lines of Code Written

21 lines in Dart.

Responsible Member

Ahmed Jaber.

3.1.2 November 7 - November 13

Parts Completed

register_page.dart page was completed.

Lines of Code Written

131 lines in Dart.

Responsible Members

Ahmed Jaber.

3.1.3 November 14 - November 20

Parts Completed

login_page.dart was completed

Lines of Code Written

204 lines in Dart.

Responsible Members

Ahmed Jaber and Hamzeh Ali.

3.1.4 November 21 - November 27

Parts Completed

This week was designated for writing the first progress report.

Lines of Code Written

No new lines were added.

Responsible Members

Ahmed Jaber and Hamzeh Ali.

3.1.5 December 19 - December 25

Parts Completed

ClientClass.dart, DriverClass.dart, TripClass.dart, home_page.dart, driver_register_page.dart, and choose_register_page.dart were completed.

Lines of Code Written

723 lines in Dart.

Responsible Members

Ahmed Jaber and Hamzeh Ali.

3.1.6 December 26 - January 1

Parts Completed

share_drive.dart, list_drivers.dart, and Trips.dart, auth_page.dart, and main_page.dart files were completed.

Lines of Code Written

465 lines in Dart.

Responsible Members

Ahmed Jaber and Hamzeh Ali.

3.1.7 January 2 - January 8

Parts Completed

Admin_Modify_Client_page.dart, Admin_Modify_Driver_page.dart, Admin_Modify_Trip_page.dart, Admin_interface.dart, Admin_view_Clients.dart, Admin_view_Drivers.dart, Admin_view_Trips.dart, ViewDatabase.dart, ViewModificationpage.dart, and Admin_backup.dart files were completed.

Lines of Code Written

1119 lines in Dart.

Responsible Members

Ahmed Jaber and Hamzeh Ali.

3.2 Memory Requirements

No specific requirements are needed.

3.3 Cloud Database Storage Restrictions

Firebase [2] Cloud Service is a Non-SQL database management system that provides up to 10 GB of storage without billing.

Chapter 4

References

1. https://github.com/Codingaway20/sharek_application
2. https://www.tutorialspoint.com/firebase/index.htm