Carpool Application

Abdel Rahman Mohamed Salah | Mobile Programming | 22/12/2023

Table of Contents

Test Credentials

Use these account to login directly:

For driver: Email=”test@gmail.com” Password=”test@1”

For user: Email=”19p0000@eng.asu.edu.eg” Password=”test@2”

Introduction

This documentation outlines the steps taken to develop a rideshare app that meets the specified requirements. The app aims to provide a user-friendly interface for managing rides, payments, and order tracking. The development process involves implementing features such as route listing, cart management, order history, payment processing, and order tracking.

The application, meticulously crafted for the unique needs of the academic community, requires users to sign in with their active @eng.asu.edu.eg accounts, creating a trusted and exclusive closed community. The application is designing for ride share and for it to give the best user experience it should always be used in portrait mode and that is why I locked the orientation in the manifest file.

In addition to its focus on user authentication, the CarPool application introduces a revolutionary approach to rideshare services within the academic setting. Specifically tailored for the Faculty of Engineering Community at Ainshams University, the app concentrates on rides to and from Ain Sahm University, with designated pickup points at Gate 3 and 4. To streamline the service, the application is operated entirely by students, ensuring a unique, student-centric experience. The pilot project imposes specific regulations, requiring users to reserve seats in advance for morning and afternoon rides. This proactive reservation system ensures optimal planning, with morning ride reservations due by 10:00 pm the previous day and afternoon ride reservations by 4:30 pm on the same day.

Furthermore, the app caters to the practical preferences of its users, as it is designed to be used exclusively in portrait mode for an enhanced and intuitive user experience. This deliberate choice in orientation, underscored by the manifest file configuration, reflects a commitment to providing a seamless and user-friendly interface, aligning with the app's overarching objective of simplifying rideshare logistics for the academic community.

Features

In the development of Carpool, key features have been meticulously crafted to address the unique needs of the academic community, fostering a sense of trust, exclusivity, and ease of use. The application requires users to sign in with their active @eng.asu.edu.eg accounts, ensuring a secure and trusted closed community. Here are some key features of the carpool application:

- Secure Authentication:
  - Users must sign in using their official @eng.asu.edu.eg accounts, ensuring a trusted and closed community environment.
- Intuitive Interface:
  - The application boasts a user-friendly interface, featuring a login page with a sign-up option for seamless onboarding.
- Route Information:
  - A comprehensive list of available routes to and from Ainshams campus, elegantly presented using a recycler view.
- Effortless Payments:
  - A cart page allows users to review and finalize their orders
  - Smooth and efficient payment process
- Order Tracking:
  - Users can keep track of their ride history and monitor the status of their reservations through a dedicated tracking/status page.
- Real-time Database Integration:
  - The application utilizes Firebase real-time database for route information and order status, ensuring up-to-date and synchronized data.
- Local Profile Storage:
  - User profiles, including essential information, are stored locally using Room database for efficient retrieval and management.
  - Previously logged in user on that device are allowed to login while offline to check profile information.

UI Design

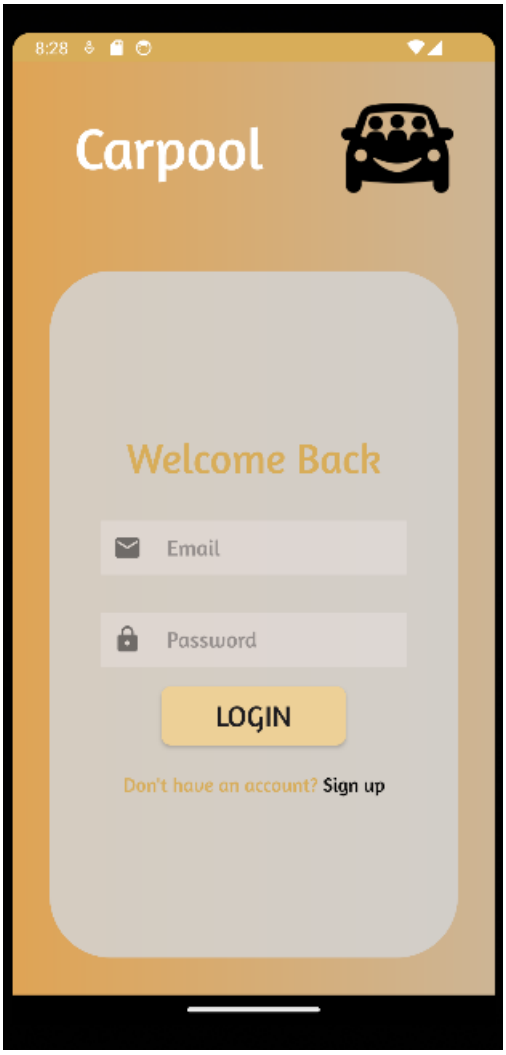
I tried to used different layouts in my design to my best and choose the most suitable for each activity/fragment, The fragments with list or simpler fragments I chose linear layout as I did not want to complicate the design and the linear layout would give the required output design, while more complex designs like the tracking page I went with the constraint layout to give me more flexibility in the designing process.

OUTPUT PAGES

For the main theme I chose the golden colors as I thought it was a relaxing color and gives a good contrast with white and black which are also used in my design.

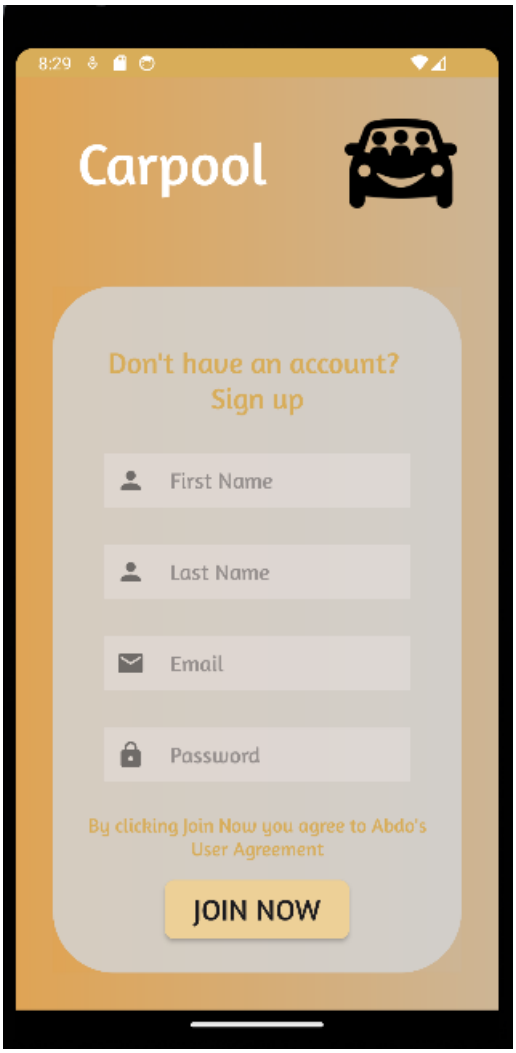
Login Page

Implemented a login page to allow user access to the application and the first page the users sees when the application starts. Included a signup option for new users (the word “sign up” is actually a hyperlink to the sign up page), after logging in the users will be directed to the home page.



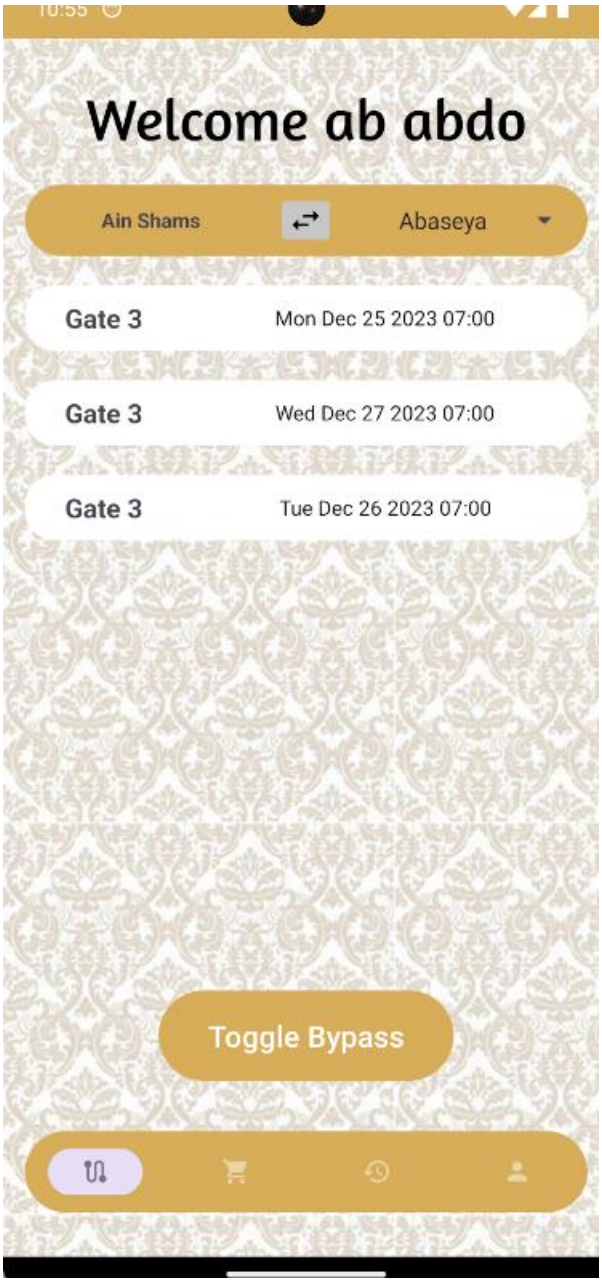
Sign Up Page

For new users to enter their credentials and have and account so they can use the application.



Route listing

Designed a route listing page using RecyclerView to display available routes to and from Ain Shams Campus. Each route is presented in a card-like format for better visualization. Ability to filter results.

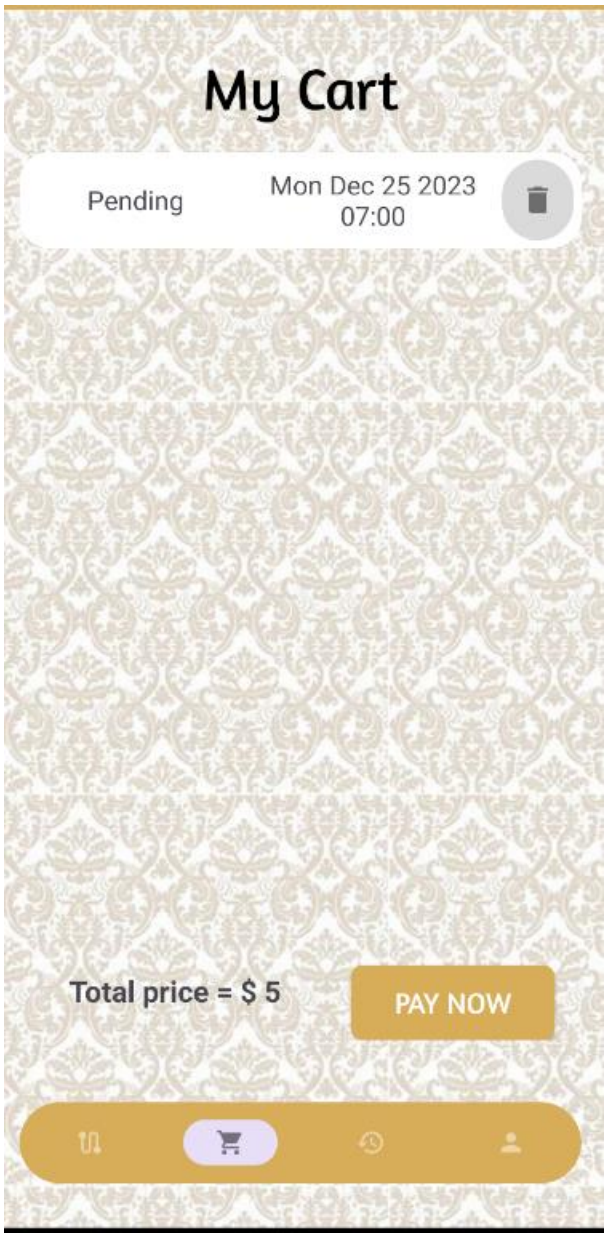


The navigation bar

Very important part of the application as it simplifies the transitioning between pages for the users. Has three items to go to three pages easily and efficiently, the routes page, cart page, history page and profile page each having an icon to show which page to go to.

Cart Page

Developed a cart page that allows users to review their orders. The cart interface includes details such as the selected routes, quantities, and cost. Also, there is a button to remove any undesired item.

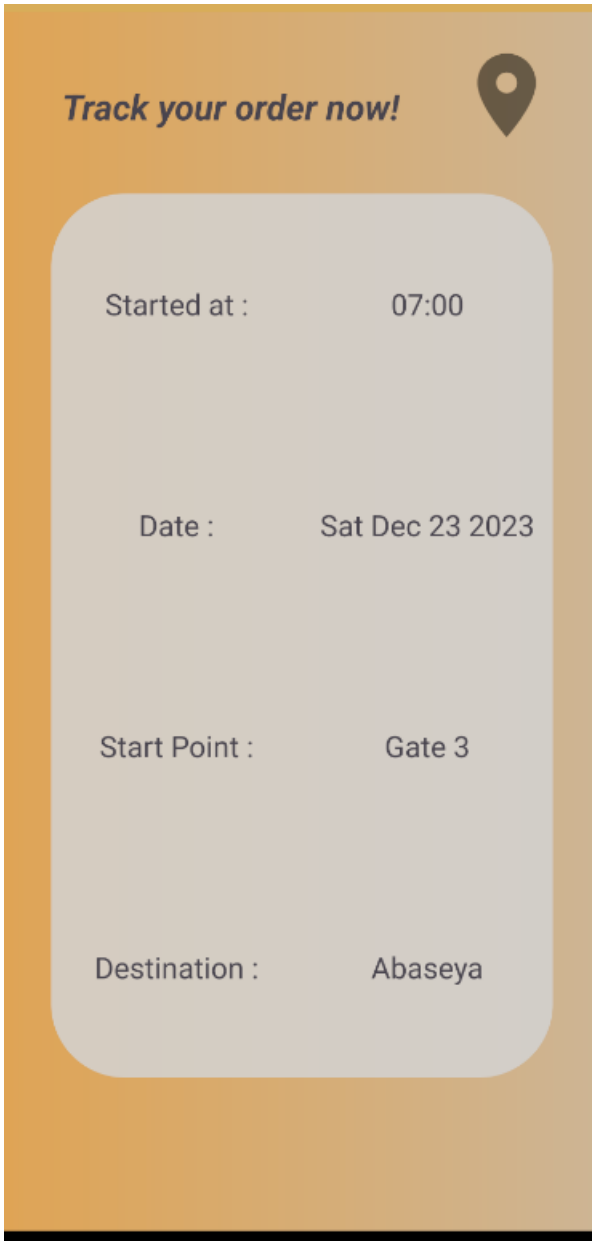


Order History

Implemented an order history page with a tracking/status section. Users can view the details of their past orders.

Payment and Order Tracking Page

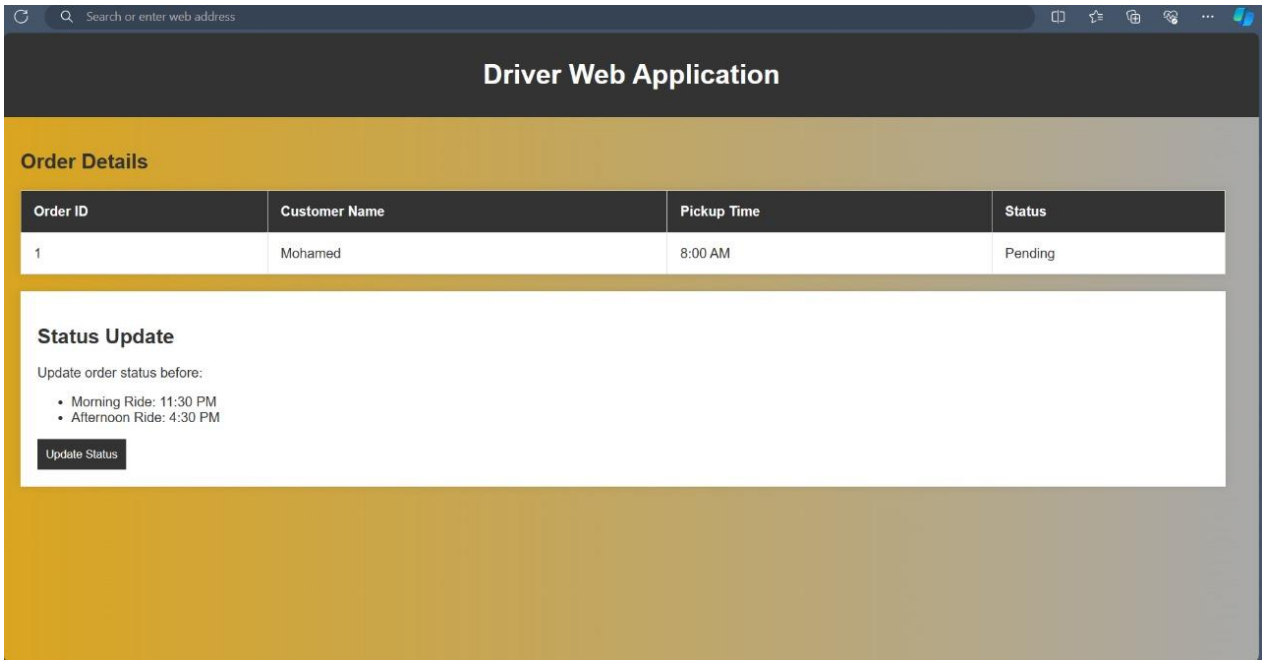
Designed an order tracking page to view details of history items.



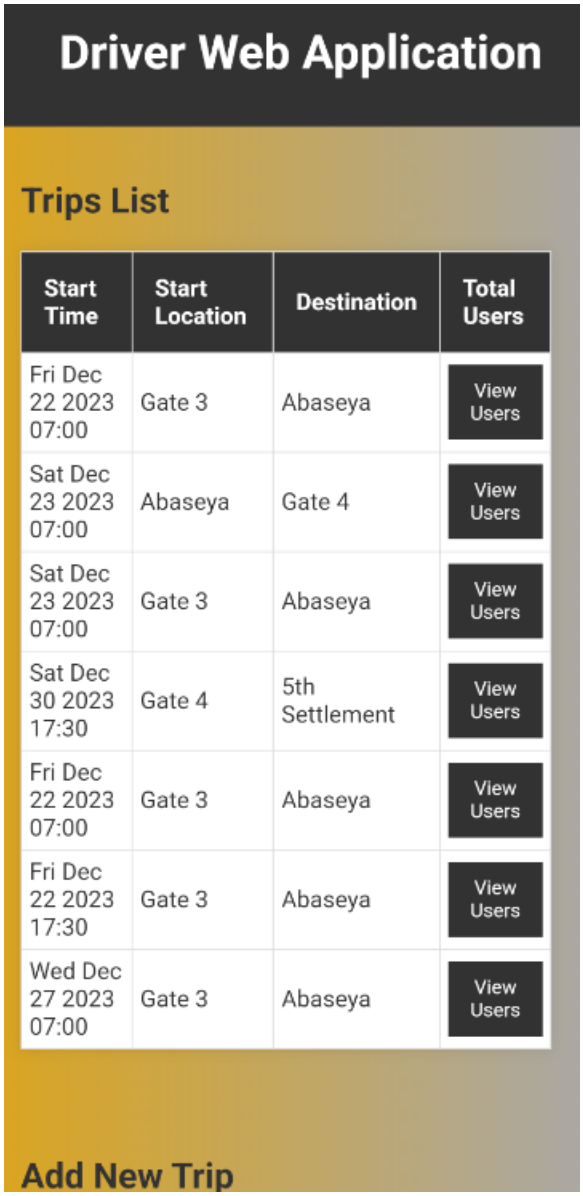
Web Application for Drivers

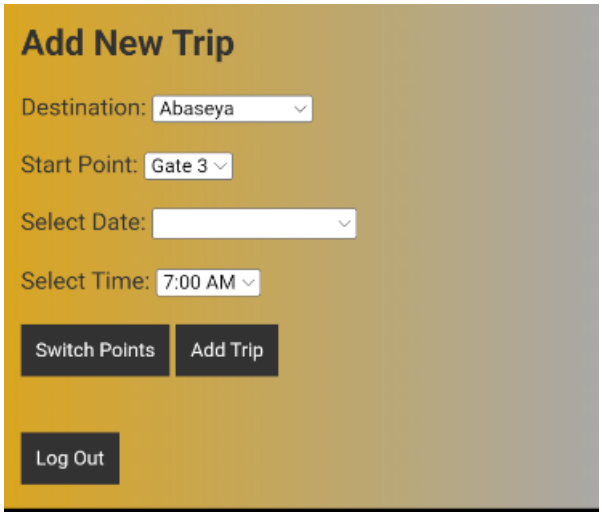
Developed a web application for drivers to confirm orders and update user request status data. Implemented time restrictions to ensure morning and afternoon ride orders are confirmed on time.

Webpage Design



Webpage displayed in the app (webview)



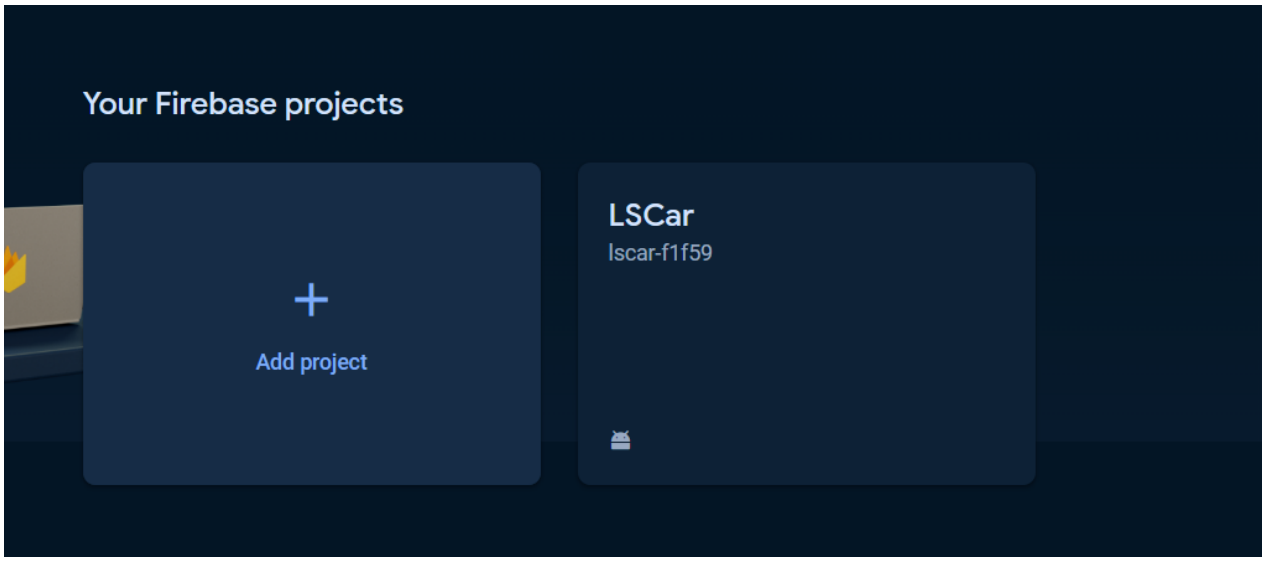


Database Design

Firebase is utilized for user authentication in both the “Login” and “Sign_up” fragments. Firebase Authentication is a robust service that integrates with Android applications, offering secure and scalable user sign-up and sign-in functionalities.

The FirebaseAuth instance is instantiated in both fragments, allowing for the creation of user accounts with email and password credentials (createUserWithEmailAndPassword), as well as user sign-in (signInWithEmailAndPassword). Additionally, Firebase's real-time database or Firestore could be employed to store and retrieve user-related data, enhancing the potential for personalized user experiences within the application. Firebase provides a comprehensive and easy-to-use solution for handling user authentication and data management, reducing the development effort required to implement these critical features while ensuring robust security practices.

The following image shows the new project I created and linked it to my android application to use firebase services.



DATABASE STRUCTURE

Room Database Structure:

In the development of the CarPool application, a robust and well-structured database architecture has been implemented to seamlessly manage user profiles, driver information, and route details. The database comprises both a local Room database and a cloud-based Firestore database, each serving specific purposes to ensure data integrity, reliability, and efficient retrieval.

Note that in the password table the password is hashed for security reasons. And the stored hashed password allows offline logging in. Room includes a separate table for securely storing user passwords.

User Database (Room):

Table Name: "user"

Columns:

- userId: Primary key, uniquely identifying each user.
- name: User's name.
- email: User's email address.
- isStudent: Boolean indicating whether the user is a student.

Password Database (Room):

Table Name: "password"

Columns:

- userId: Foreign Primary key referencing the user, ensuring a one-to-one relationship.
- password: Securely stored user password.

Firebase Firestore Database Structure

Users Collection (Firestore):

Each document corresponds to a user and is identified by the user's ID.

Fields include userId, name, email, and isStudent.

Drivers Collection (Firestore):

Each document corresponds to a driver, identified by the driver's ID.

Fields include driver-specific information such as userId, name, email, and any additional driver-related data.

Routes Collection (Firestore):

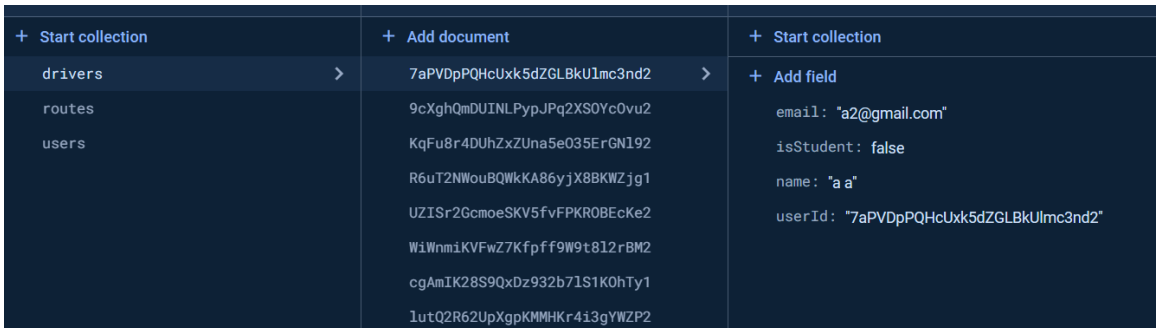
Each document represents a route, identified by a unique ID.

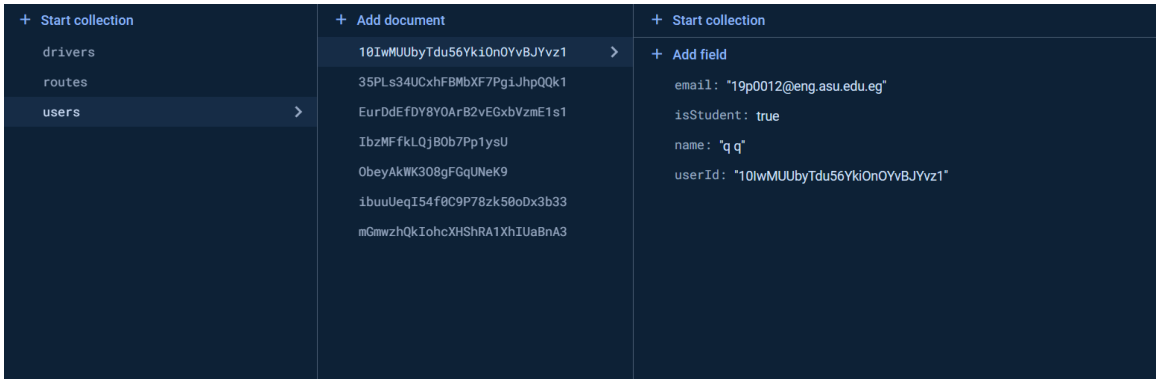
Fields include driverEmail, startTime, startPoint, destination, and a map (userRequestStatusMap) indicating users in the trip and their payment status.

For Firestore, the driverRoutes subcollection allows nesting driver-specific route details within the routes document, providing a structured way to organize data.

This database structure ensures a well-organized and scalable design for the application, supporting efficient data retrieval and updates for user profiles, route details, and driver-specific information. The use of both local (Room) and cloud (Firestore) databases allows for optimal performance and data synchronization between the app and the server.

The **UserViewModel** plays a pivotal role in the MVVM architecture, acting as an intermediary between the UI (User Interface) and the underlying data. UserViewModel orchestrates the flow of user-related data, ensuring that the UI remains responsive and up-to-date. By utilizing LiveData, the UserViewModel efficiently communicates changes in user data to the UI, enabling dynamic updates without unnecessary manual intervention.





Rules

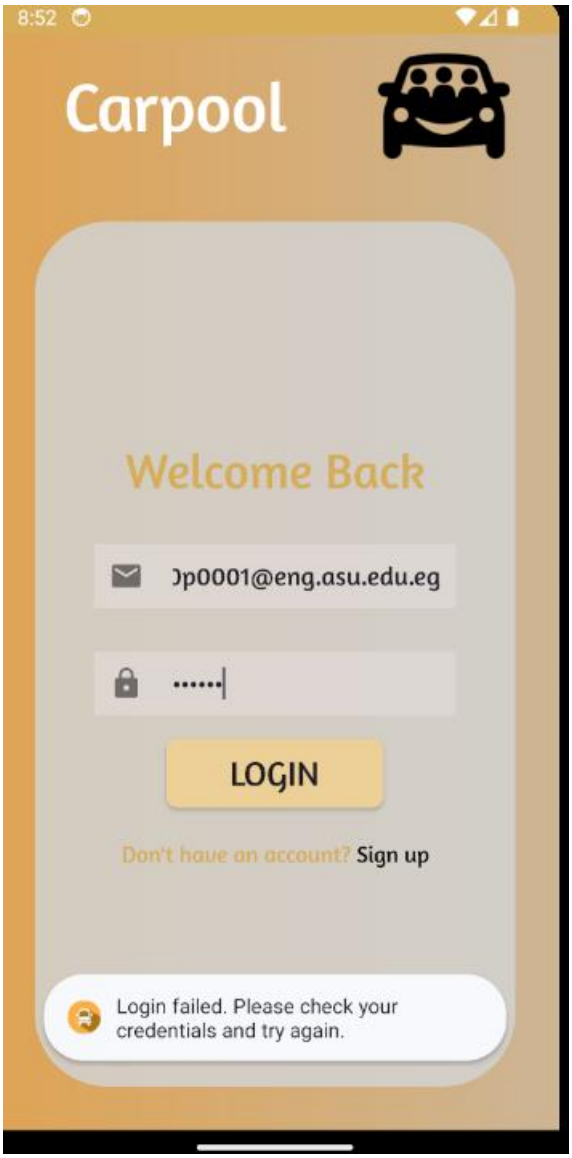
In the CarPool application, the Firebase Security Rules are meticulously crafted to ensure a robust and secure environment for user and driver data. These rules implement a fine-grained access control strategy, allowing users and drivers to read and write their respective documents within the "users" and "drivers" collections based on their unique identifiers (UIDs).

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {

    // Allow read and write access to the "users" collection only for authenticated users
    match /users/{userId} {
      allow read, write: if request.auth != null && request.auth.uid == userId;
    }

    // Allow read and write access to the "drivers" collection only for authenticated drivers
    match /drivers/{driverId} {
      allow read, write: if request.auth != null && request.auth.uid == driverId;
    }

    // Allow read and write access to the "routes" collection only for authenticated users
    match /routes/{routeId} {
      allow read, write: if request.auth != null;
    }

    // Allow read and write access to the "driverRoutes" subcollection only for authenticated drivers
    match /routes/{routeId}/driverRoutes/{driverId} {
      allow read, write: if request.auth != null && request.auth.uid == driverId;
    }
  }
}
```

Test Cases

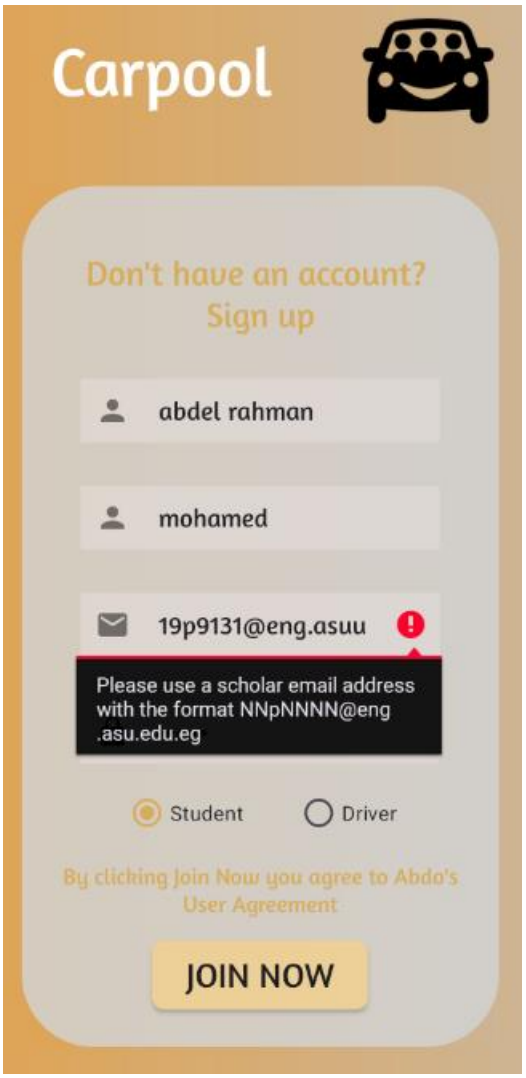
CASE 1

The user entered wrong sign in credentials:



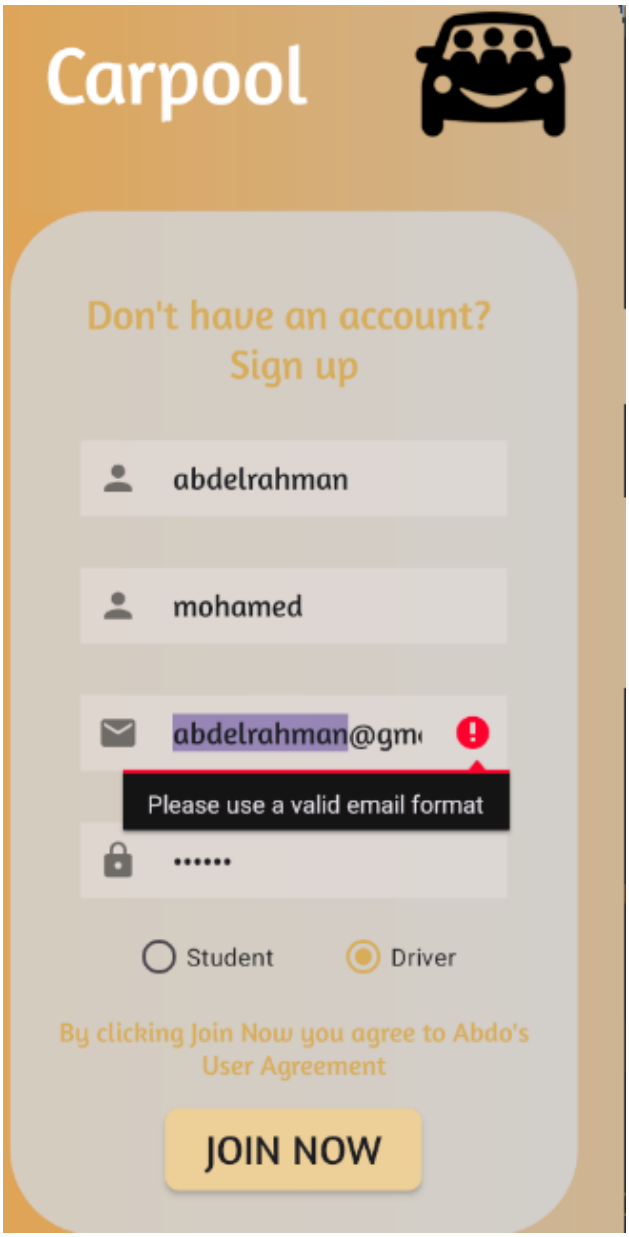
CASE 2

The user entered wrong email format in sign up:



CASE 3

The driver entered wrong email format in sign up:



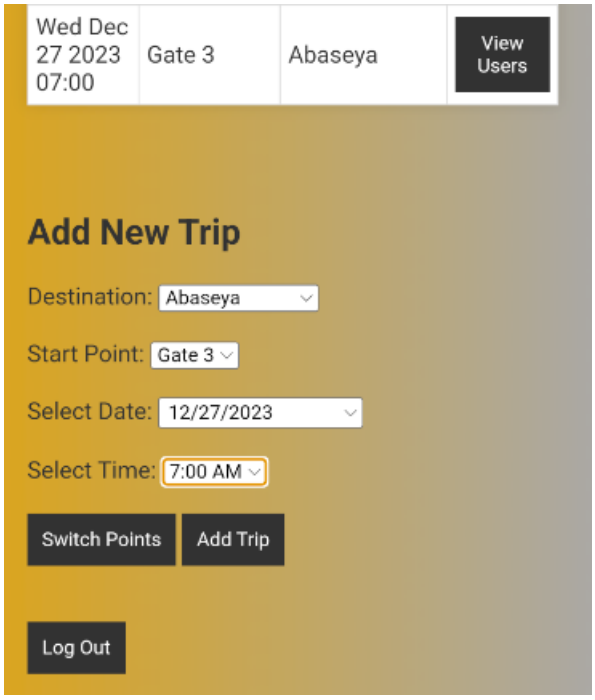
CASE 4

Successful sign up added user in firebase:

```
email: "19p9131@eng.asuu.eg"
isStudent: false
name: "abdel rahman mohamed"
userId: "WiWnmiKVFwZ7Kfpff9W9t8l2rBM2"
```

CASE 5

Driver adds a trip




```
destination: "Abaseya"
driverEmail: "test1@gmail.com"
id: "8df7bc6b-b999-49bd-bb31-20ece82b1cf9"
startPoint: "Gate 3"
startTime: "Wed Dec 27 2023 07:00"
```

CASE 6

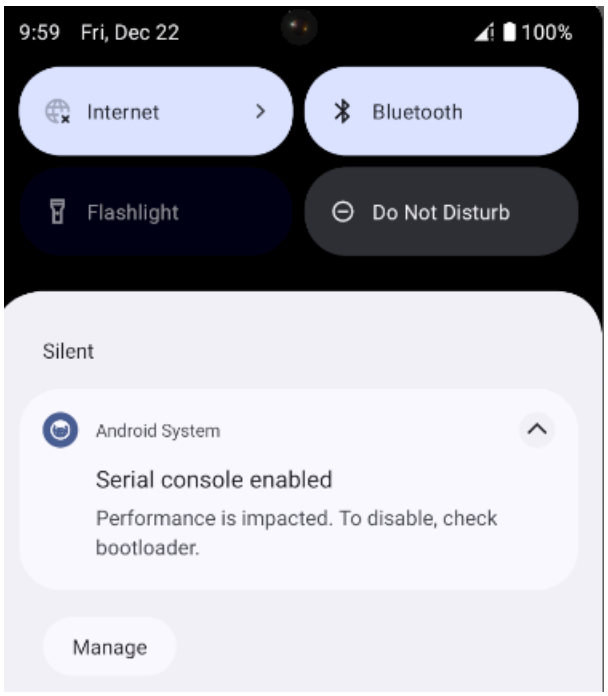
The user entered want to see routes so previous routes are not seen as they passed the time constraints:

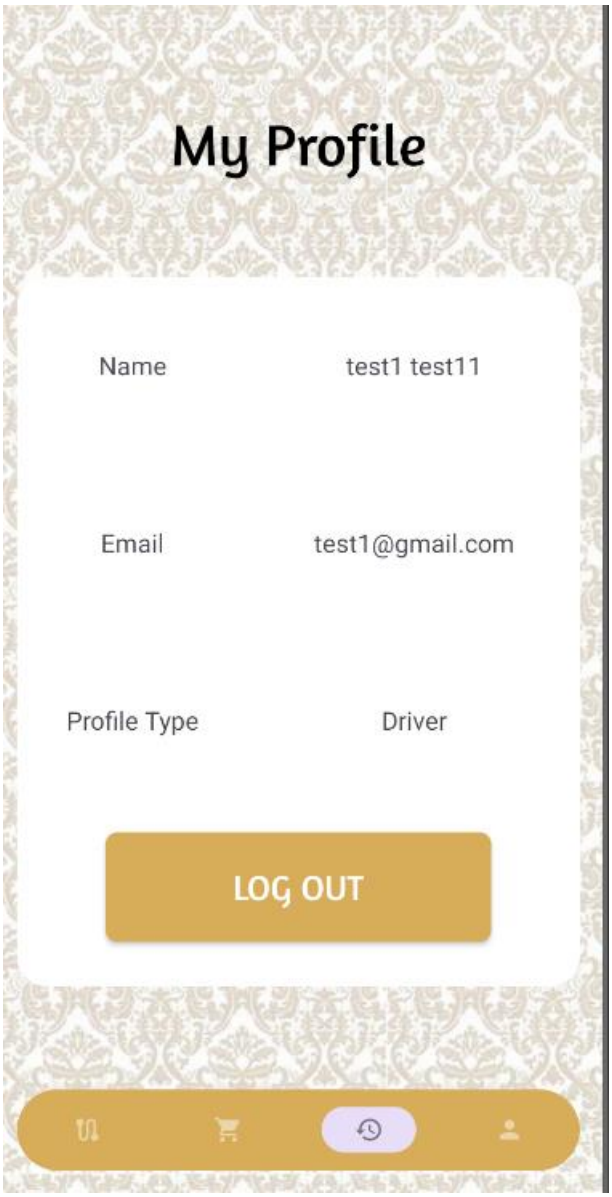
Current time: 9:14 PM 12/22/2023

Bypass time constraints

CASE 7

The user closed internet connection (and logs in using an account that was logged in before while internet was connected)





Navigation disabled only sees profile (from Room)

Database Codes

USER

```
package com.example.mile.User;


import androidx.annotation.NonNull;
import androidx.room.ColumnInfo;
import androidx.room.Entity;
import androidx.room.PrimaryKey;

import com.google.firebase.firestore.PropertyName;

import java.io.Serializable;
import java.util.HashMap;
import java.util.Map;

@Entity(tableName = "user")
public class User implements Serializable {
    @PrimaryKey
    @NonNull
    private String userId;
    @ColumnInfo(name="name")
    private String name;

    @ColumnInfo(name="email")
    private String email;
    @ColumnInfo(name="isStudent")
    private boolean isStudent;


    public User() {
    }

    public User(String userId, String username, String email,boolean
isStudent) {
        this.userId = userId;
        this.name = username;
        this.email = email;
        this.isStudent = isStudent;
    }


    public Map<String, Object> toMap() {
        Map<String, Object> userMap = new HashMap<>();
        userMap.put("userId", userId);
        userMap.put("name", name);
```

```
        userMap.put("email", email);
        userMap.put("isStudent", isStudent);

        return userMap;
    }

    public String getUserId() {
        return userId;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }

    @PropertyName(value="isStudent")
    public boolean isStudent() {
        return this.isStudent;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @PropertyName(value="isStudent")
    public void setStudent(boolean value) {
        this.isStudent = value;
    }

}
```

USERDAO

```
package com.example.mile.User;

import androidx.room.Dao;
import androidx.room.Insert;
import androidx.room.OnConflictStrategy;
import androidx.room.Query;

import com.example.mile.User.User;

@Dao
public interface UserDao {
    @Insert
    void insert(User user);

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insertPassword(Password password);


    @Query("SELECT * FROM user WHERE userId = :userId")
    User getUserById(String userId);

    @Query("SELECT * FROM user")
    User getUsers();

    @Query("SELECT * FROM password WHERE userId = :userId")
    Password getPasswordById(String userId);

    @Query("SELECT * FROM user WHERE email = :email")
    User getUserByEmail(String email);



}
```

USERDATABASE

```
package com.example.mile.User;

import android.content.Context;

import androidx.room.Database;
import androidx.room.Room;
import androidx.room.RoomDatabase;

import com.example.mile.User.User;

@Database(entities = {User.class, Password.class}, version = 2,
exportSchema = false)
public abstract class UserDatabase extends RoomDatabase {
    private static UserDatabase instance;

    public abstract UserDao userDao();

    public static synchronized UserDatabase getInstance(Context
context) {
        if (instance == null) {
            instance = Room.databaseBuilder(
                    context.getApplicationContext(),
                    UserDatabase.class,
                    "user_database"
            ).fallbackToDestructiveMigration().build();
        }
        return instance;
    }

}
```

USERREPO

```
package com.example.mile.User;

import android.content.Context;
import android.util.Log;

import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class UserRepo {
    private UserDao userDao;
    private static final Executor databaseReadExecutor =
Executors.newSingleThreadExecutor();

    private UserRepo(UserDao userDao) {
        this.userDao = userDao;
    }
    private static final ExecutorService databaseWriteExecutor =
            Executors.newFixedThreadPool(1);
    public UserRepo(Context context) {
        UserDatabase database = UserDatabase.getInstance(context);
        userDao = database.userDao();
    }

    public static UserRepo getInstance(UserDao userDao) {
        return new UserRepo(userDao);
    }

    public User getUserById(String userId) {
        return userDao.getUserById(userId);
    }

    public void insert(User user) {
        databaseWriteExecutor.execute(() -> userDao.insert(user));
    }

    public void insertPassword(Password password) {
        databaseWriteExecutor.execute(() ->
userDao.insertPassword(password));
    }
    public interface GetUserCallback {
        void onUserLoaded(User user, Password existingPass);
    }

    public interface GetPassCallback {
        void onReturn(boolean value,User user);
    }
    public void login( User newUser, String password,GetUserCallback
callback) {
        databaseReadExecutor.execute(() -> {
```

```
            User existingUser =
userDao.getUserById(newUser.getUserId());
            Password existingPass =
userDao.getPasswordById(newUser.getUserId());


        if (existingUser == null) {
            insert(newUser);
            existingPass=new Password(newUser.getUserId(),password);
            insertPassword(existingPass);
        }
        else {
        }
        callback.onUserLoaded(newUser,existingPass);


        });
    }
public boolean v;
    public void verifyUserCredentials(String email, String
password,GetPassCallback callback) {
        databaseReadExecutor.execute(() -> {
        User user = userDao.getUserByEmail(email);

        if (user != null) {
            Password storedPassword =
userDao.getPasswordById(user.getUserId());
            if (storedPassword != null) {
                v=
Password.verifyPassword(storedPassword.getPassword(), password);
            }
        }
            callback.onReturn(v,user);
        });
    }

}
```

USERVIEWMODEL

```
package com.example.mile.User;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

import com.example.mile.User.User;
import com.google.firebase.firestore.FirebaseFirestore;
import com.google.firebase.firestore.ListenerRegistration;

public class UserViewModel extends ViewModel {

    private final FirebaseFirestore db =
FirebaseFirestore.getInstance();
    private final MutableLiveData<User> userLiveData = new
MutableLiveData<>();
    private ListenerRegistration listenerRegistration;

    public LiveData<User> getUserLiveData(String userId) {
        // Remove previous listener if any
        if (listenerRegistration != null) {
            listenerRegistration.remove();
        }

        // Set up a new listener for the specified user ID
        listenerRegistration = db.collection("users").document(userId)
                .addSnapshotListener((snapshot, e) -> {
                    if (e != null) {
                        // Handle error
                        return;
                    }

                    if (snapshot != null && snapshot.exists()) {
                        // Convert the DocumentSnapshot to a User
object
                        User user = snapshot.toObject(User.class);
                        userLiveData.setValue(user);
                    } else {
                        // Document does not exist
                        userLiveData.setValue(null);}
});

        return userLiveData;
    }

    @Override
    protected void onCleared() {
        // Remove the listener when the ViewModel is cleared
        if (listenerRegistration != null) {
```

```
            listenerRegistration.remove();
        }}}
```

ROUTEITEM

```
package com.example.mile.Route;

import java.io.Serializable;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.UUID;

public class RouteItem implements Serializable {
    private String driverEmail;
    private String id;
    private String startTime;
    private String destination;
    private String startPoint;
    private Map<String, String> userRequestStatusMap;

    public RouteItem() {
        // Required empty constructor for Firestore
    }

    public RouteItem(String driverEmail,String startTime, String
startPoint, String destination) {
        this.driverEmail=driverEmail;
        this.startTime = startTime;
        this.startPoint = startPoint;
        this.destination = destination;
        this.userRequestStatusMap = new HashMap<>();
        setid();
    }


    public String getDriverEmail() {
        return driverEmail;
    }
    public String getid() {
        return id;
    }
    public String getStartTime() {
        return startTime;
    }

    public String getdestination() {
        return destination;
    }

    public String getStartPoint() {
        return startPoint;
    }
    public Map<String, String> getuserRequestStatusMap(){
        return userRequestStatusMap;
    }
```

```
    public void setid() {
        this.id=UUID.randomUUID().toString();
    }



}
```

ROUTEREPO

```
package com.example.mile.User;

import android.content.Context;
import android.util.Log;

import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class UserRepo {
    private UserDao userDao;
    private static final Executor databaseReadExecutor =
Executors.newSingleThreadExecutor();

    private UserRepo(UserDao userDao) {
        this.userDao = userDao;
    }
    private static final ExecutorService databaseWriteExecutor =
            Executors.newFixedThreadPool(1);
    public UserRepo(Context context) {
        UserDatabase database = UserDatabase.getInstance(context);
        userDao = database.userDao();
    }

    public static UserRepo getInstance(UserDao userDao) {
        return new UserRepo(userDao);
    }

    public User getUserById(String userId) {
        return userDao.getUserById(userId);
    }

    public void insert(User user) {
        databaseWriteExecutor.execute(() -> userDao.insert(user));
    }

    public void insertPassword(Password password) {
        databaseWriteExecutor.execute(() ->
userDao.insertPassword(password));
    }
    public interface GetUserCallback {
        void onUserLoaded(User user, Password existingPass);
    }
```

```
    public interface GetPassCallback {
        void onReturn(boolean value,User user);
    }
    public void login( User newUser, String password,GetUserCallback
callback) {
        databaseReadExecutor.execute(() -> {
            User existingUser =
userDao.getUserById(newUser.getUserId());
            Password existingPass =
userDao.getPasswordById(newUser.getUserId());


        if (existingUser == null) {
            insert(newUser);
            existingPass=new Password(newUser.getUserId(),password);
            insertPassword(existingPass);
        }
        else {
        }
        callback.onUserLoaded(newUser,existingPass);


        });
    }
public boolean v;
    public void verifyUserCredentials(String email, String
password,GetPassCallback callback) {
        databaseReadExecutor.execute(() -> {
        User user = userDao.getUserByEmail(email);

        if (user != null) {
            Password storedPassword =
userDao.getPasswordById(user.getUserId());
            if (storedPassword != null) {
                v=
Password.verifyPassword(storedPassword.getPassword(), password);
            }
        }
            callback.onReturn(v,user);
        });
    }

}
```

ROUTEVIEWMODEL

```
package com.example.mile.User;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

import com.example.mile.User.User;
import com.google.firebase.firestore.FirebaseFirestore;
import com.google.firebase.firestore.ListenerRegistration;

public class UserViewModel extends ViewModel {

    private final FirebaseFirestore db =
FirebaseFirestore.getInstance();
    private final MutableLiveData<User> userLiveData = new
MutableLiveData<>();
    private ListenerRegistration listenerRegistration;

    public LiveData<User> getUserLiveData(String userId) {
        // Remove previous listener if any
        if (listenerRegistration != null) {
            listenerRegistration.remove();
        }

        // Set up a new listener for the specified user ID
        listenerRegistration = db.collection("users").document(userId)
                .addSnapshotListener((snapshot, e) -> {
                    if (e != null) {
                        // Handle error
                        return;
                    }

                    if (snapshot != null && snapshot.exists()) {
                        // Convert the DocumentSnapshot to a User
object
                        User user = snapshot.toObject(User.class);
                        userLiveData.setValue(user);
                    } else {
                        // Document does not exist
                        userLiveData.setValue(null);
                    }
                });

        return userLiveData;
    }

    @Override
    protected void onCleared() {
        // Remove the listener when the ViewModel is cleared
        if (listenerRegistration != null) {
            listenerRegistration.remove();
```

```
        }
    }
}
```

FIREBASEHELP

```
package com.example.mile;

import android.content.Context;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.util.Log;

import androidx.annotation.NonNull;

import com.example.mile.Route.RouteItem;
import com.example.mile.User.User;
import com.google.android.gms.tasks.OnCompleteListener;
import com.google.android.gms.tasks.Task;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.auth.FirebaseUser;
import com.google.firebase.firestore.CollectionReference;
import com.google.firebase.firestore.DocumentReference;
import com.google.firebase.firestore.DocumentSnapshot;
import com.google.firebase.firestore.FirebaseFirestore;
import com.google.firebase.firestore.QueryDocumentSnapshot;
import com.google.firebase.firestore.QuerySnapshot;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.CompletableFuture;

public class FirebaseHelp {
    FirebaseFirestore db = FirebaseFirestore.getInstance();
    public static FirebaseAuth auth = FirebaseAuth.getInstance();
    User user;
    static FirebaseUser firebaseUser;
    private static FirebaseHelp instance;

    private FirebaseHelp() {
//        this.setUser();
        auth = FirebaseAuth.getInstance();
        firebaseUser = auth.getCurrentUser();
    }
```

```
    // Public static method to get the singleton instance
    public static FirebaseHelp getInstance() {
        // Lazy initialization: create the instance if it hasn't been
created yet
        if (instance == null) {
            instance = new FirebaseHelp();
        }
        firebaseUser = auth.getCurrentUser();
        return instance;
    }
    static void set(FirebaseUser user){
        firebaseUser=user;
    }
    void addUser(User user){
        DocumentReference userRef =
db.collection("users").document(user.getUserId());
        userRef.set(user)
                .addOnSuccessListener(aVoid -> Log.d("AddingUser",
"DocumentSnapshot added with ID: " + userRef.getId()))
                .addOnFailureListener(e -> Log.w("AddingUser", "Error
adding document", e));
    }

    public String getCurrentUserId(){
        return firebaseUser.getUid();
    }


    void addDriver(User user){
        DocumentReference userRef =
db.collection("drivers").document(user.getUserId());

        userRef.set(user.toMap())
                .addOnSuccessListener(aVoid -> Log.d("AddUserSuccess",
"DocumentSnapshot added with ID: " + userRef.getId()))
                .addOnFailureListener(e -> Log.w("AddUserError", "Error
adding document", e));
    }

    public void addRoute(RouteItem routeItem){
        db.collection("routes")
                .document(routeItem.getid())
                .set(routeItem)
                .addOnSuccessListener(aVoid -> {
                })
                .addOnFailureListener(e -> {
                });
    }

    public void updateUserStatus(String routeId, String userEmail,
String userStatus) {
        db.runTransaction(transaction -> {
            DocumentReference routeRef =
db.collection("routes").document(routeId);
```

```
            // Retrieve the current userRequestStatusMap
            DocumentSnapshot snapshot = transaction.get(routeRef);
            Map<String, String> userRequestStatusMap =
snapshot.contains("userRequestStatusMap") ?
                    (Map<String, String>)
snapshot.getData().get("userRequestStatusMap") : new HashMap<>();

            // Add or update the status for the user
            userRequestStatusMap.put(userEmail, userStatus);

            // Update the userRequestStatusMap in the document
            transaction.update(routeRef, "userRequestStatusMap",
userRequestStatusMap);

            return null;
        }).addOnSuccessListener(result -> {
            Log.d("TAG", "User status updated successfully for user " +
userEmail);
        }).addOnFailureListener(e -> {
            Log.e("TAG", "Error updating user status", e);
        });
    }



    CompletableFuture<User> getUserById(String type) {
        String userId= auth.getUid();
        return getUserById(userId,type);
    }



    CompletableFuture<User> getUserById(String userId,String type) {
        CompletableFuture<User> future = new CompletableFuture<>();

        DocumentReference userRef =
db.collection(type).document(userId);
        userRef.get().addOnCompleteListener(new
OnCompleteListener<DocumentSnapshot>() {
            @Override
            public void onComplete(@NonNull Task<DocumentSnapshot>
task) {
                if (task.isSuccessful()) {
                    DocumentSnapshot document = task.getResult();
                    if (document.exists()) {
                        Log.d("FirestoreData", "DocumentSnapshot data:
" + document.getData());
                        User user = document.toObject(User.class);
                        if (user != null) {
                            // Resolve the CompletableFuture with the
user object
                            future.complete(user);
                        } else {
                            Log.w("RetrieveUser", "Error deserializing
User");
```

```
                            // Complete exceptionally if user is null
                            future.completeExceptionally(new
RuntimeException("Error deserializing User"));
                        }
                    } else {
                        Log.d("RetrieveUser", "No such document");
                        // Complete exceptionally if document doesn't
exist
                        future.completeExceptionally(new
RuntimeException("No such document"));
                    }
                } else {
                    Log.w("RetrieveUser", "Error getting document",
task.getException());
                    // Complete exceptionally with the task exception
                    future.completeExceptionally(task.getException());
                }
            }
        });

        return future;

    }


    public CompletableFuture<List<RouteItem>> getAllDriversRoutes() {
        CompletableFuture<List<RouteItem>> future = new
CompletableFuture<>();

        db.collection("routes")
                .get()
                .addOnCompleteListener(new
OnCompleteListener<QuerySnapshot>() {
                    @Override
                    public void onComplete(@NonNull Task<QuerySnapshot>
task) {
                        if (task.isSuccessful()) {
                            List<RouteItem> allRoutes = new
ArrayList<>();
                            for (QueryDocumentSnapshot document :
task.getResult()) {
                                db.collection("routes")
                                        .document(document.getId())
                                        .collection("driverRoutes")
                                        .get()
                                        .addOnCompleteListener(new
OnCompleteListener<QuerySnapshot>() {
                                            @Override
                                            public void
onComplete(@NonNull Task<QuerySnapshot> task) {
                                                if
(task.isSuccessful()) {
                                                    for
(QueryDocumentSnapshot routeDocument : task.getResult()) {
                                                        RouteItem
```

```
routeItem = routeDocument.toObject(RouteItem.class);

allRoutes.add(routeItem);
                                                        }
                                                        // Check if this is
the last driver
                                                        if
(allRoutes.size() == task.getResult().size()) {
                                                            // Resolve the
CompletableFuture with all routes

future.complete(allRoutes);
                                                        }
                                                    } else {
                                                        Log.e("TAG", "Error
getting routes for driver: " + document.getId(), task.getException());
                                                        // Complete
exceptionally with the task exception

future.completeExceptionally(task.getException());
                                                    }
                                                }
                                            });
                                }
                            } else {
                                Log.e("TAG", "Error getting all drivers",
task.getException());
                                // Complete exceptionally with the task
exception

future.completeException(task.getException());
                            }
                        }
                    });

        return future;
    }
    void signOut(){
        auth.signOut();
    }


    public String getCurrentUseremail(){
        FirebaseUser firebaseUser = auth.getCurrentUser();
        String email=firebaseUser.getEmail();
        return email;
    }
    public void setUser(){
        String type="drivers";
        FirebaseUser firebaseUser = auth.getCurrentUser();
        String email=firebaseUser.getEmail();
        if (isValidScholarEmail(email)){
            type="users";
        }
        getUserById(type).thenAccept(user -> {
```

```
            // Handle the user object here
            if (user != null) {
                this.user=user;
            } else {
            }
        }).exceptionally(ex -> {
            // Handle exceptions here
            Log.e("Get user profile", "Error getting user", ex);
            return null; // You can return a default value or handle it
as needed
        });
    }

    private boolean isValidScholarEmail(String email) {
        String emailRegex = "^[0-9]{2}p[0-9]{4}@eng\\.asu\\.edu\\.eg$";

        int currentYear =
(Calendar.getInstance().get(Calendar.YEAR))%100;

        if (email.matches(emailRegex)) {
            int enrollmentYear = Integer.parseInt(email.substring(0,
2));
            int oldestYear = currentYear-10;
            return enrollmentYear <= currentYear && enrollmentYear >=
oldestYear;
        }

        return false;
    }



        public static boolean isNetworkConnected(Context context) {
            ConnectivityManager cm = (ConnectivityManager)
context.getSystemService(Context.CONNECTIVITY_SERVICE);
            if (cm != null) {
                NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
                return activeNetwork != null &&
activeNetwork.isConnectedOrConnecting();
            }
            return false;
        }


    public void changeStatus(String tripId, String user, String
newStatus) {
        DocumentReference routeRef =
db.collection("routes").document(tripId);

        routeRef.get()
                .addOnSuccessListener(documentSnapshot -> {
                    if (documentSnapshot.exists()) {
                        // Retrieve the RouteItem object
                        RouteItem routeItem =
documentSnapshot.toObject(RouteItem.class);
```

```
                        // Update the status in the
userRequestStatusMap
                        if (routeItem != null) {

routeItem.getuserRequestStatusMap().put(user, newStatus);

                            // Update the document in Firestore
                            routeRef.set(routeItem)
                                    .addOnSuccessListener(aVoid ->
Log.d("ChangeStatus", "Status updated successfully"))
                                    .addOnFailureListener(e ->
Log.w("ChangeStatus", "Error updating status", e));
                        }
                    } else {
                        Log.d("ChangeStatus", "Route document does not
exist");
                    }
                })
                .addOnFailureListener(e -> Log.w("ChangeStatus", "Error
retrieving route document", e));
    }


    public void removeUserStatus(String tripId, String user) {
        DocumentReference routeRef =
db.collection("routes").document(tripId);

        routeRef.get()
                .addOnSuccessListener(documentSnapshot -> {
                    if (documentSnapshot.exists()) {
                        // Retrieve the RouteItem object
                        RouteItem routeItem =
documentSnapshot.toObject(RouteItem.class);

                        // Remove the entry for the specified user from
the userRequestStatusMap
                        if (routeItem != null) {

routeItem.getuserRequestStatusMap().remove(user);

                            // Update the document in Firestore
                            routeRef.set(routeItem)
                                    .addOnSuccessListener(aVoid ->
Log.d("RemoveUserStatus", "User status removed successfully"))
                                    .addOnFailureListener(e ->
Log.w("RemoveUserStatus", "Error removing user status", e));
                        }
                    } else {
                        Log.d("RemoveUserStatus", "Route document does
not exist");
                    }
                })
                .addOnFailureListener(e -> Log.w("RemoveUserStatus",
"Error retrieving route document", e));
```

```
    }

    public void pay(String userEmail){
        FirebaseFirestore db = FirebaseFirestore.getInstance();
        CollectionReference routesCollection = db.collection("routes");

        // Fetch all documents from the "routes" collection
        routesCollection.get().addOnCompleteListener(new
OnCompleteListener<QuerySnapshot>() {
            @Override
            public void onComplete(Task<QuerySnapshot> task) {
                if (task.isSuccessful()) {
                    for (QueryDocumentSnapshot document :
task.getResult()) {
                        // Convert the document data to a RouteItem
object
                        RouteItem routeItem =
document.toObject(RouteItem.class);

                        // Check if the userRequestStatusMap contains
the specified email
                        if
(routeItem.getuserRequestStatusMap().containsKey(userEmail)) {
                            // Update the status to "Paid"

routeItem.getuserRequestStatusMap().put(userEmail, "Paid");

                            // Save the updated routeItem back to
Firestore

routesCollection.document(document.getId()).set(routeItem);
                        }
                    }
                } else {
                    // Handle the error
                    Exception exception = task.getException();
                    if (exception != null) {
                        exception.printStackTrace();
                    }
                }
            }
        });
    }
}
```