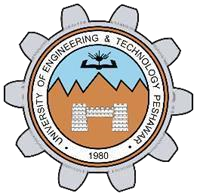
# Object Oriented Programming

# 

# Project Lab Report



**FALL 2024**

Submitted by: **Maryam Fareed**

Registration No: **23PWCSE2228**

Section: **A**

Submitted by: **Youmna Saifullah**

Registration No: **23PWCSE2246**

Section: **A**

**Submitted To:**

Mam Sumayyea Salahuddin

**Department Of Computer Systems Engineering**

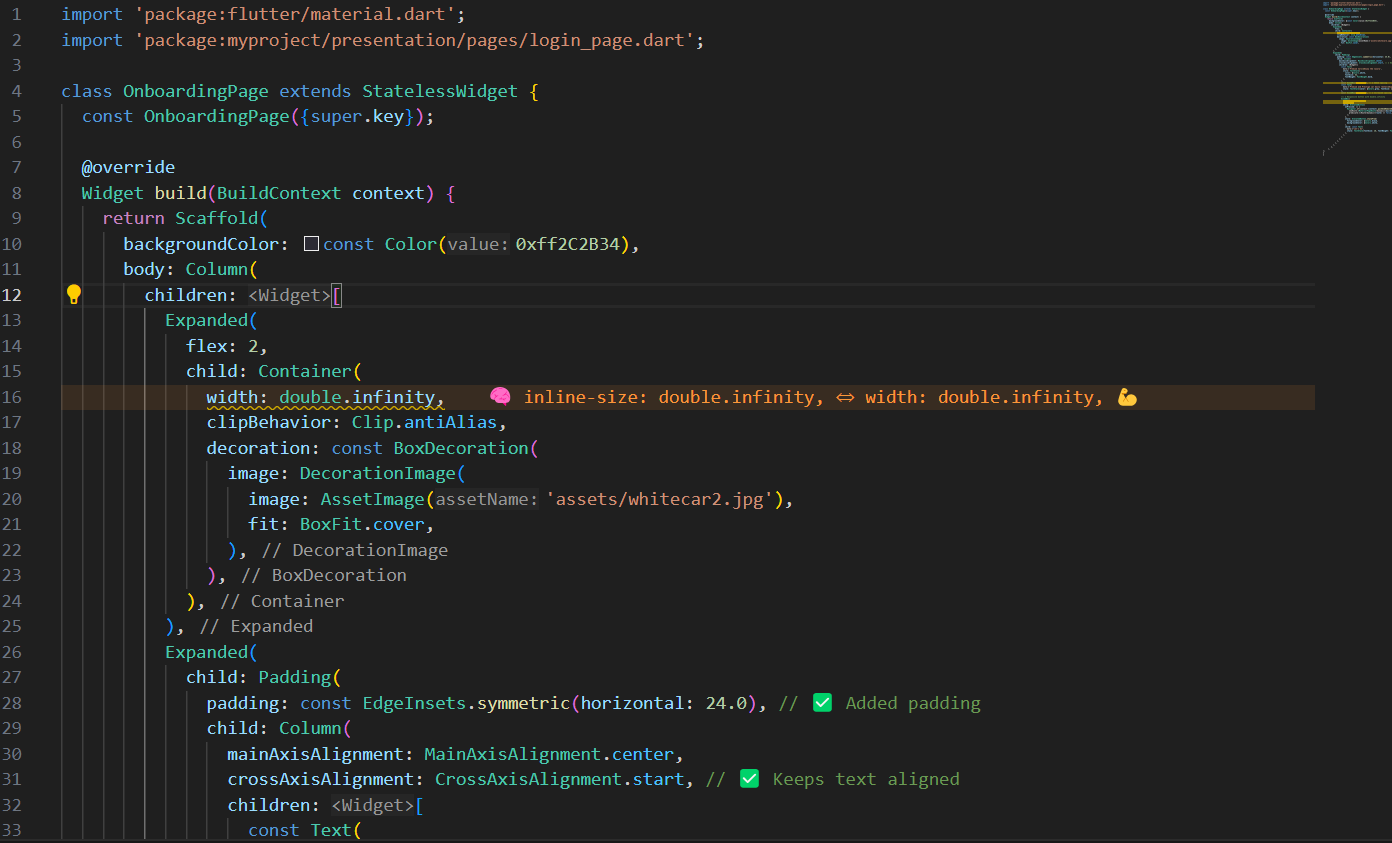
**University of Engineering And Technology, Peshawar**

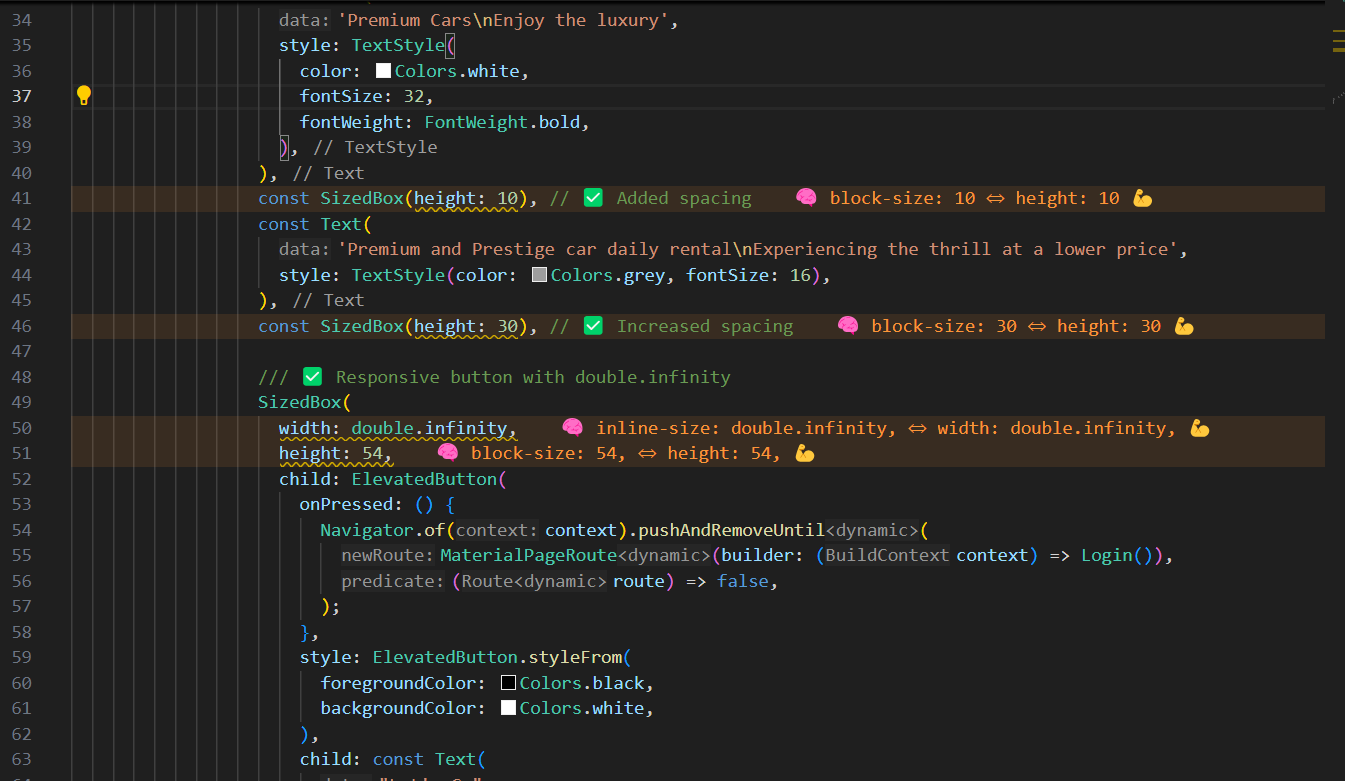
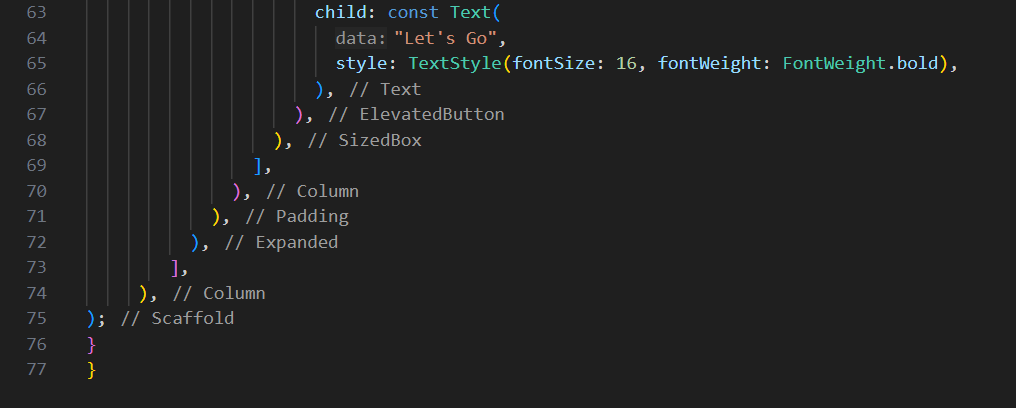
# Rental Car App in Flutter using OOPS

**Explanation of code:**

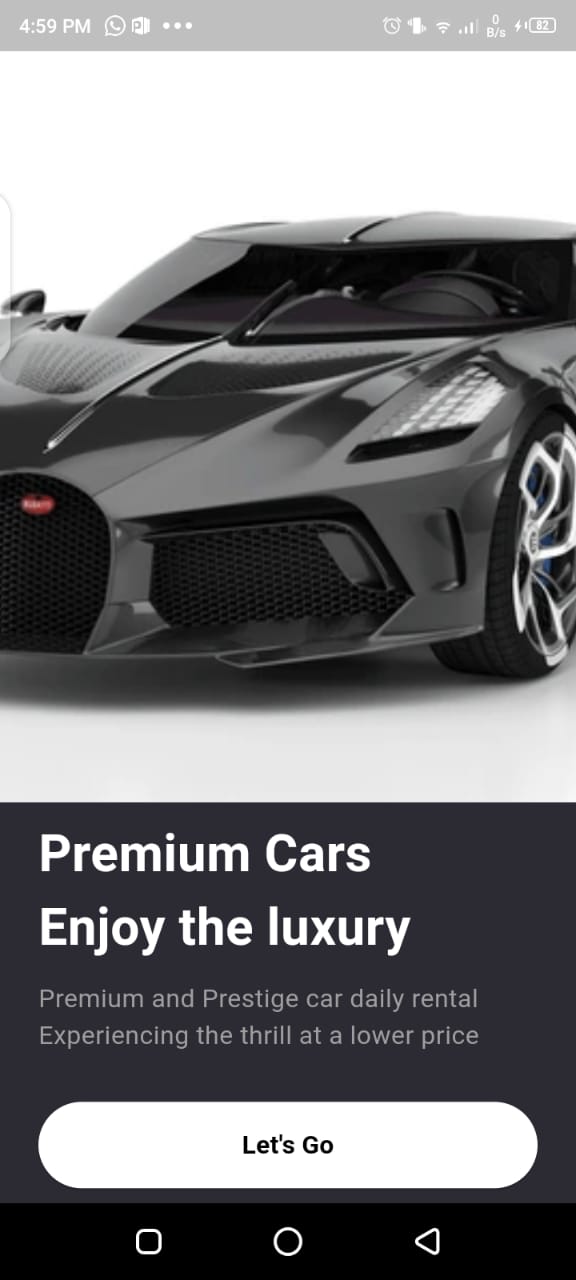
Step by step:

**On boarding page:**





**Output:**

****

**Explanation of code:**

**Importing Required Packages**

* The flutter/material.dart package is imported to use Flutter UI components.
* The login\_page.dart file is imported to enable navigation to the login screen.

**Defining the OnboardingPage Class**

* OnboardingPage is a **stateless widget**, meaning it does not maintain any state.
* The constructor includes {super.key} to improve widget rebuilding efficiency.

**Building the UI with Scaffold**

* The Scaffold widget provides the basic structure of the screen.
* The background color is set to a dark shade using a hexadecimal color code.

**Using a Column for Layout**

* The Column widget is used to arrange UI elements vertically.
* Two Expanded widgets divide the screen into two sections: one for the image and one for text and a button.

**Displaying the Image**

* An Expanded widget with flex: 2 ensures the image takes two-thirds of the screen height.
* The image is wrapped inside a Container and displayed using AssetImage().
* BoxFit.cover ensures the image covers the entire container.

**Adding Text Content**

* The second Expanded widget holds the text and button.
* Padding is applied to maintain spacing from screen edges.
* The **title text** is styled with bold white color and a font size of 32.
* A **subtitle** is added in grey color with a smaller font size for additional details.

**Adding a Button for Navigation**

* A SizedBox is used to make the button **full width** and **54 pixels high**.
* An ElevatedButton is created with a white background and black text.
* The onPressed function uses Navigator.of(context).pushAndRemoveUntil() to navigate to the Login page, removing the onboarding screen from memory.

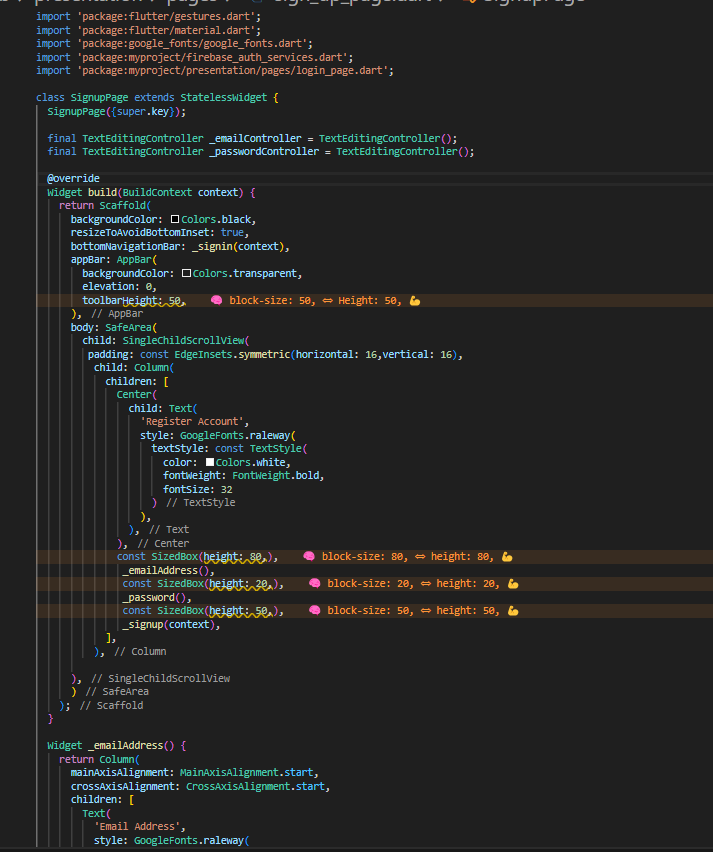
**Key OOP Concepts Used**

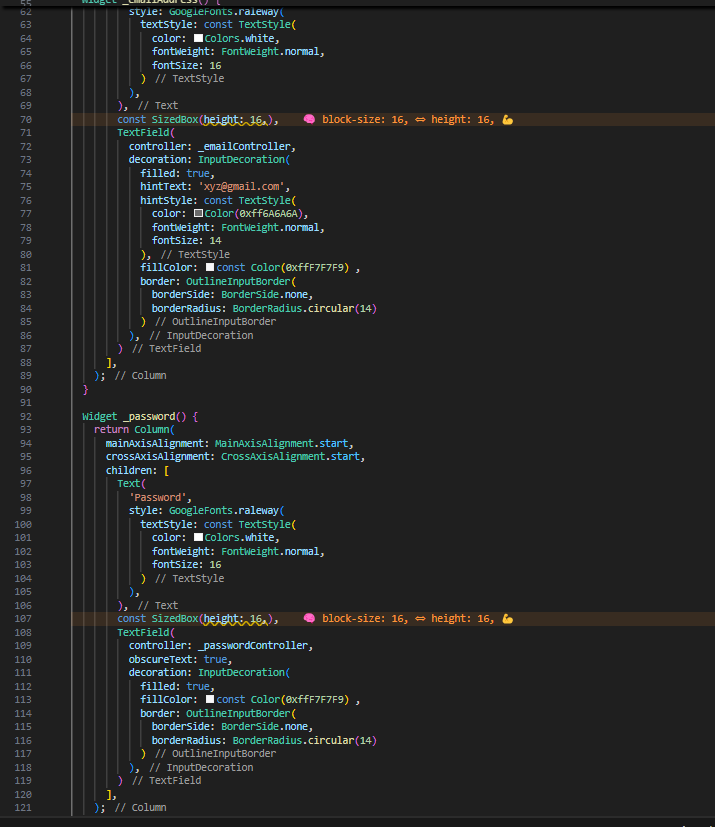
* **Encapsulation:** The OnboardingPage class encapsulates all UI elements inside a single widget.
* **Abstraction:** Implementation details are hidden within widgets like Container, Text, and ElevatedButton.
* **Reusability:** Using a StatelessWidget makes this page easily reusable.
* **Navigation:** Uses the Navigator class to handle screen transitions efficiently.

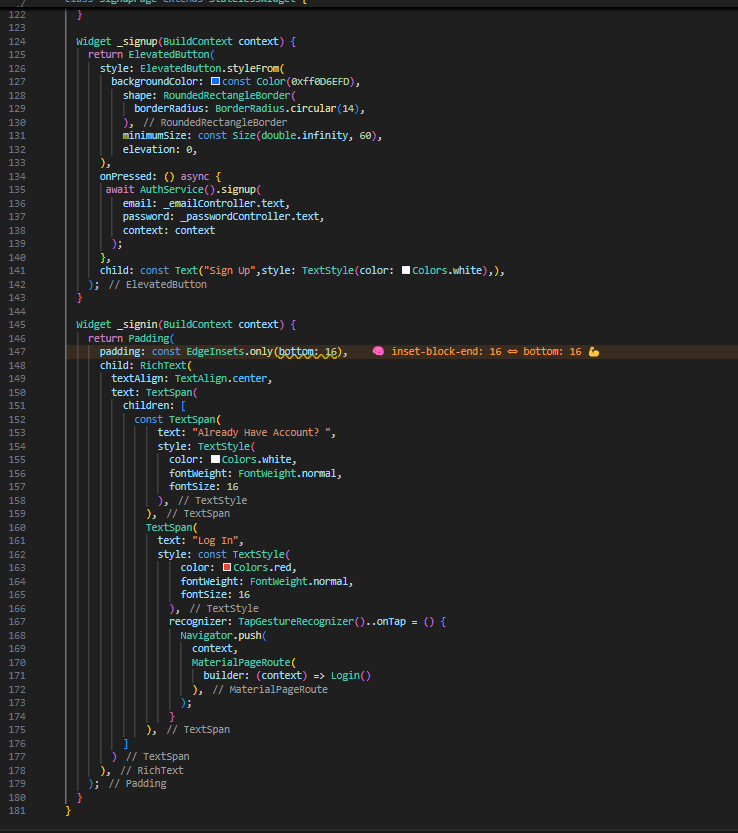
**Conclusion**

* This onboarding screen provides a visually appealing introduction to the app.
* It effectively combines **text, images, and navigation** to create a smooth user experience.

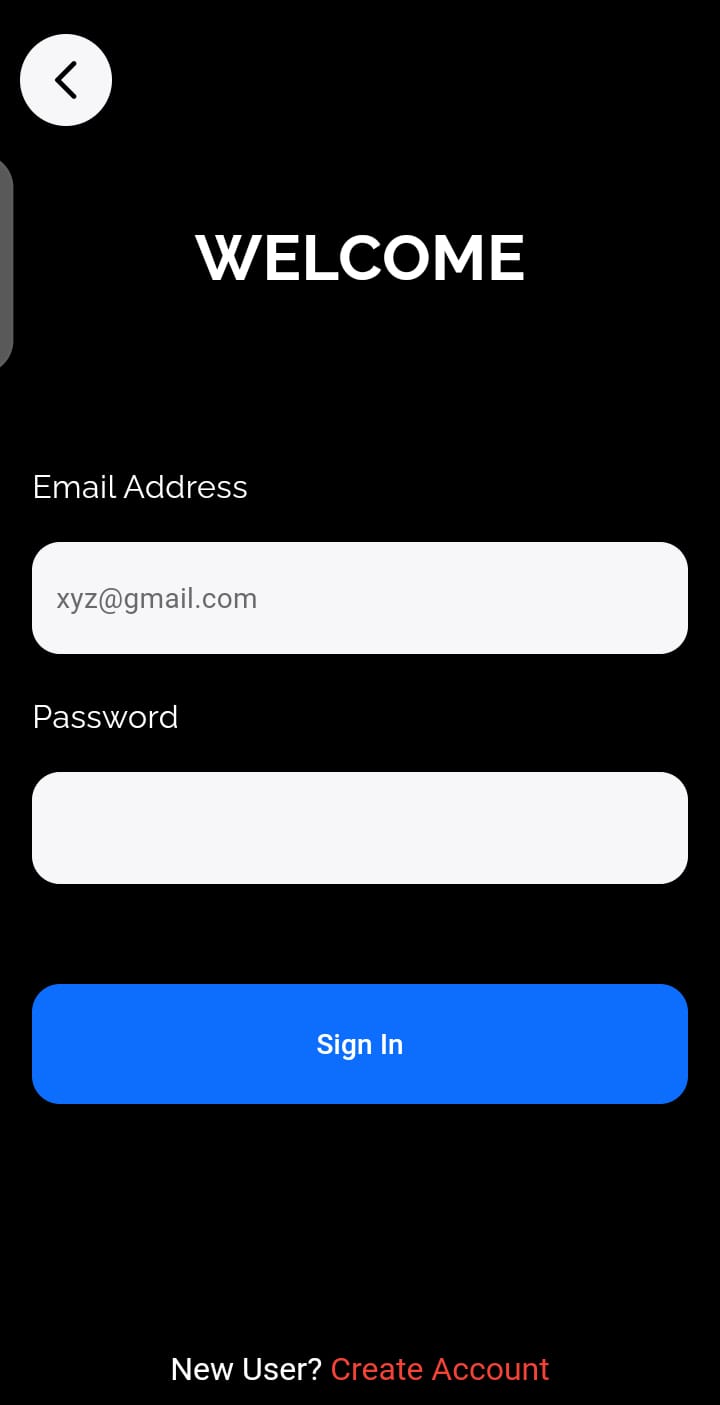
**Sign-up Page:**







**Output:**



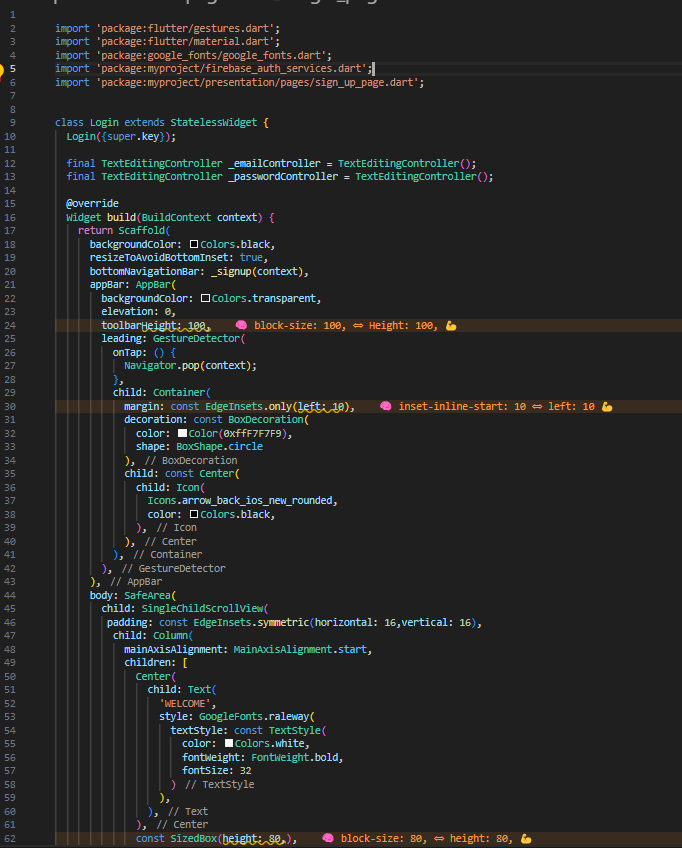
**Explanation of code:**

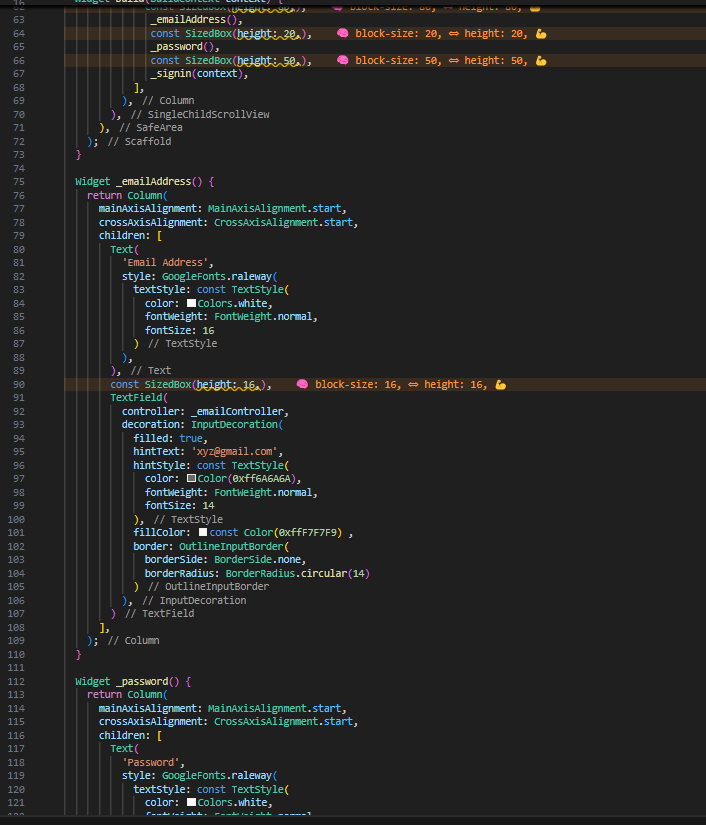
1. **Importing Required Packages**
   * flutter/material.dart is used for UI components.
   * flutter/gestures.dart is imported to handle gesture recognition for clickable text.
   * google\_fonts.dart is included to use custom fonts.
   * firebase\_auth\_services.dart is used for Firebase authentication.
   * login\_page.dart is imported for navigation to the login screen.
2. **Defining the SignupPage Class**
   * SignupPage is a **stateless widget**, meaning it does not maintain any state.
   * Two TextEditingController objects are created to handle user input for email and password.
3. **Building the UI with Scaffold**
   * The Scaffold provides the overall page structure.
   * The background color is set to black.
   * resizeToAvoidBottomInset: true ensures the UI adjusts when the keyboard appears.
   * \_signin(context) is used as the bottomNavigationBar to display a login option at the bottom.
   * The AppBar is transparent with no elevation and a height of 50.
4. **Using SafeArea and SingleChildScrollView**
   * SafeArea ensures content does not overlap with the system UI.
   * SingleChildScrollView allows vertical scrolling to prevent UI overflow.
   * padding: EdgeInsets.symmetric(horizontal: 16, vertical: 16) provides uniform spacing.
5. **Adding a Title**
   * The title "Register Account" is centered and styled using GoogleFonts.raleway.
   * The text is bold, white, and has a font size of 32.
6. **Creating the Email Input Field**
   * \_emailAddress() method returns a Column containing:
     + A label **"Email Address"** with white color and normal weight.
     + A TextField where users enter their email.
     + The field has a hint text "xyz@gmail.com" with a grey color.
     + The text field has rounded corners (borderRadius: 14) and a light fill color (#F7F7F9).
7. **Creating the Password Input Field**
   * \_password() method returns a Column containing:
     + A label **"Password"** styled similarly to the email label.
     + A TextField where users enter their password.
     + obscureText: true hides the entered characters for security.
     + The input field styling is the same as the email field.
8. **Creating the Signup Button**
   * \_signup(context) method returns an ElevatedButton:
     + The button has a blue background (#0D6EFD) and rounded corners (borderRadius: 14).
     + It has a height of **60 pixels** and expands to full width.
     + When clicked, it calls the signup method from AuthService to register the user with Firebase.
9. **Creating the Sign-in Option at the Bottom**
   * \_signin(context) returns a RichText widget that displays:
     + **"Already Have an Account?"** in white.
     + **"Log In"** in red, which is clickable.
   * TapGestureRecognizer is used to detect taps and navigate to the Login page.
10. **Key OOP Concepts Used**

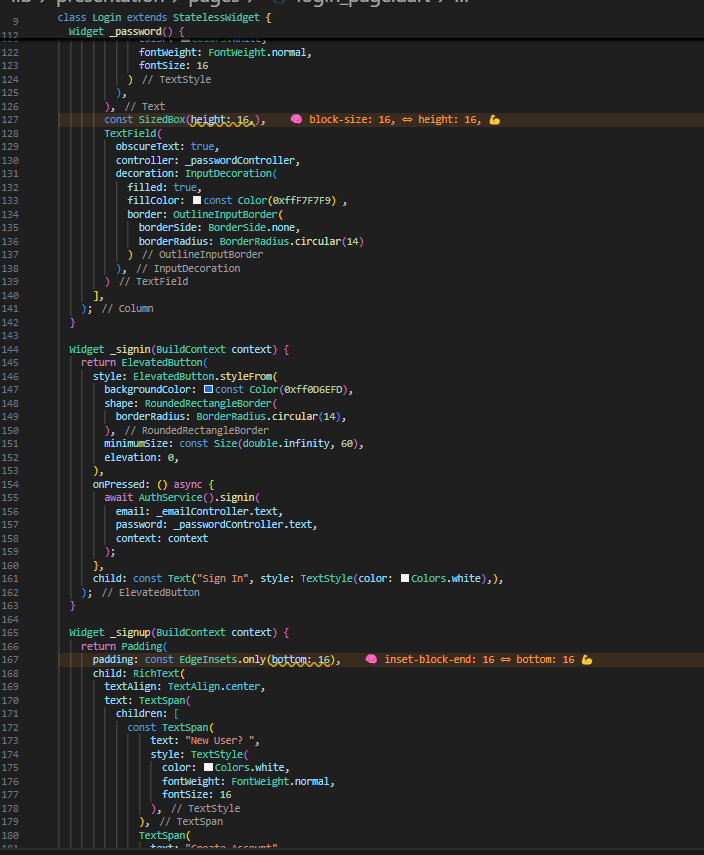
* **Encapsulation:** The UI components are wrapped inside separate functions for better modularity.
* **Abstraction:** \_emailAddress(), \_password(), \_signup(), and \_signin() hide the implementation details.
* **Reusability:** The UI is structured into reusable methods to avoid redundancy.
* **Navigation:** Uses Navigator.push() for smooth screen transitions.

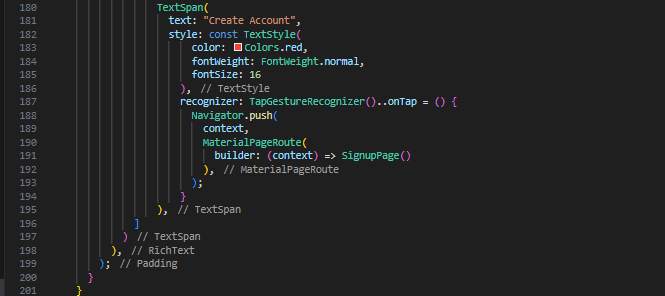
1. **Conclusion**

* This signup page provides a **modern UI** with Firebase authentication.
* The **scrollable layout** ensures usability on small screens.
* **It follows best practices for structuring UI code in Flutter.**

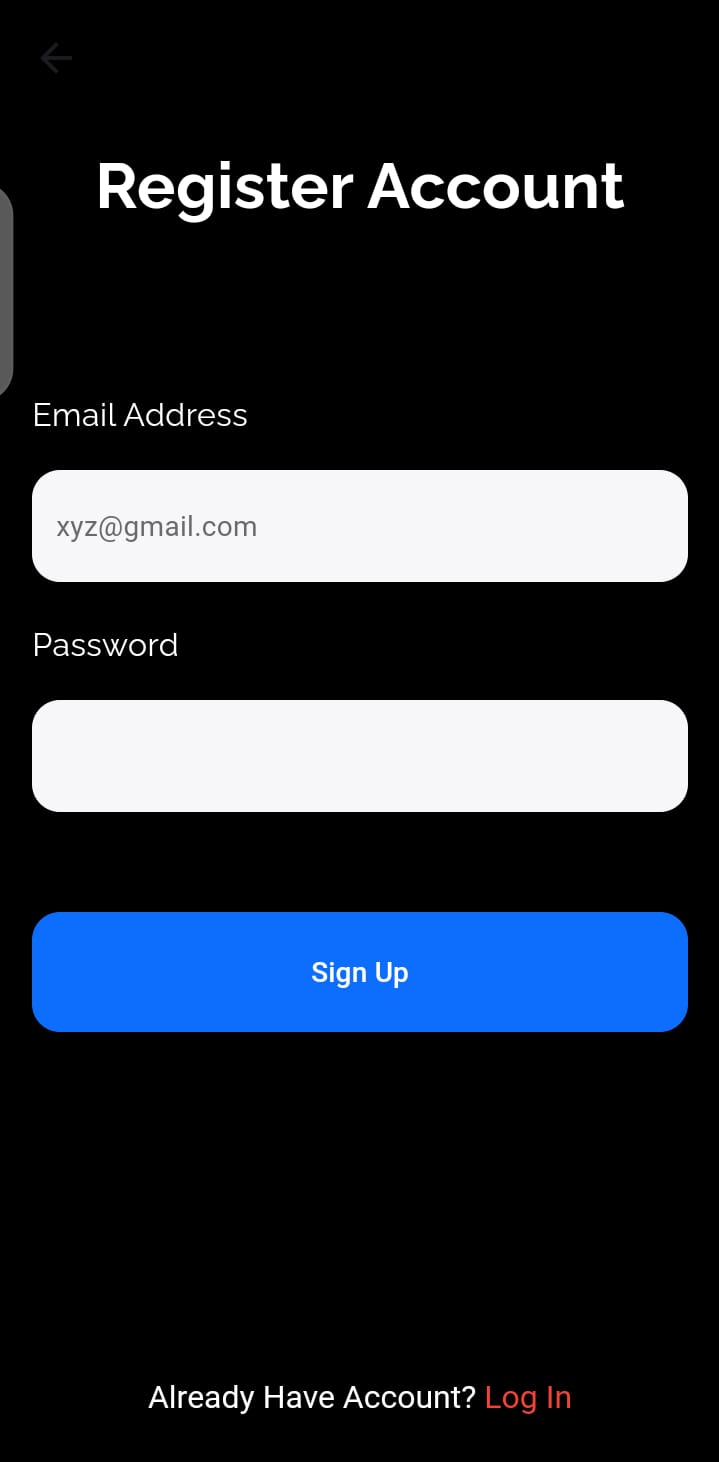
**Login Page: **

****

****



**Output:**



**Explanation of code:**

1. **Importing Required Packages**
   * flutter/material.dart for UI components.
   * flutter/gestures.dart for handling clickable text.
   * google\_fonts/google\_fonts.dart for using custom fonts.
   * firebase\_auth\_services.dart for authentication.
   * sign\_up\_page.dart for navigation to the sign-up page.
2. **Defining the Login Class**
   * Login is a **stateless widget**, meaning it does not maintain a state.
   * Two TextEditingController objects are created for email and password input.
3. **Building the UI with Scaffold**
   * The Scaffold is used to structure the page.
   * backgroundColor: Colors.black sets a dark theme.
   * resizeToAvoidBottomInset: true ensures the UI adjusts when the keyboard appears.
   * \_signup(context) is used as bottomNavigationBar to display a sign-up option.
4. **AppBar with a Custom Back Button**
   * The AppBar is transparent with no elevation and a height of 100.
   * A **custom back button** (a circular button with an arrow icon) is placed in the leading section.
   * Navigator.pop(context) is called to go back to the previous screen when tapped.
5. **Using SafeArea and SingleChildScrollView**
   * SafeArea ensures content does not overlap with the system UI.
   * SingleChildScrollView prevents overflow issues when the keyboard appears.
   * padding: EdgeInsets.symmetric(horizontal: 16, vertical: 16) provides spacing.
6. **Adding a Welcome Title**
   * A centered **"WELCOME"** title with bold, white text and a font size of 32.
7. **Creating the Email Input Field**
   * \_emailAddress() method returns a Column:
     + Label **"Email Address"** styled in white.
     + TextField for user input with a **light fill color (#F7F7F9) and rounded corners**.
     + Placeholder text "xyz@gmail.com" in grey.
8. **Creating the Password Input Field**
   * \_password() method returns a Column:
     + Label **"Password"** styled in white.
     + TextField for password input with obscureText: true for security.
     + The styling matches the email input field.
9. **Creating the Sign-in Button**
   * \_signin(context) method returns an ElevatedButton:
     + Background color is **blue (#0D6EFD)**.
     + Button has rounded corners (borderRadius: 14) and expands full width.
     + Calls AuthService().signin() to authenticate the user using Firebase.
10. **Creating the Sign-up Option at the Bottom**

* \_signup(context) returns a RichText widget that displays:
  + **"New User?"** in white.
  + **"Create Account"** in red, which is clickable.
* TapGestureRecognizer is used to detect taps and navigate to the SignupPage().

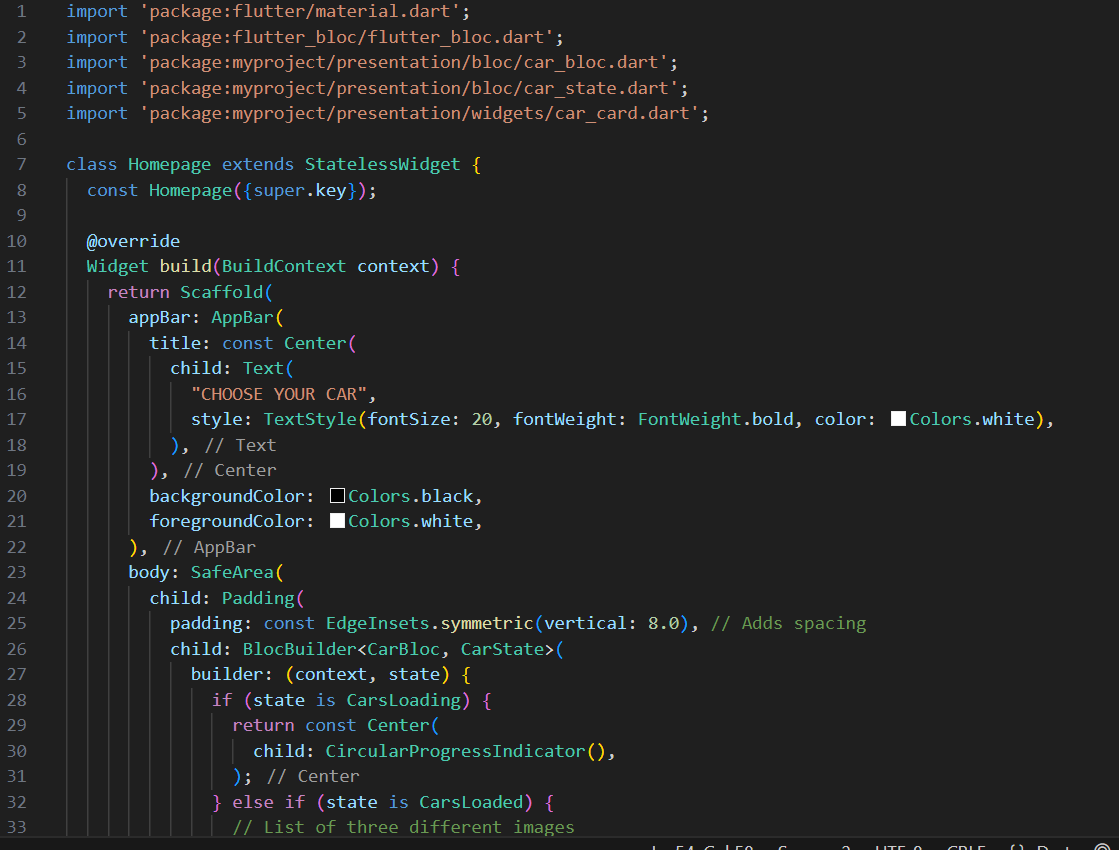
1. **Key OOP Concepts Used**

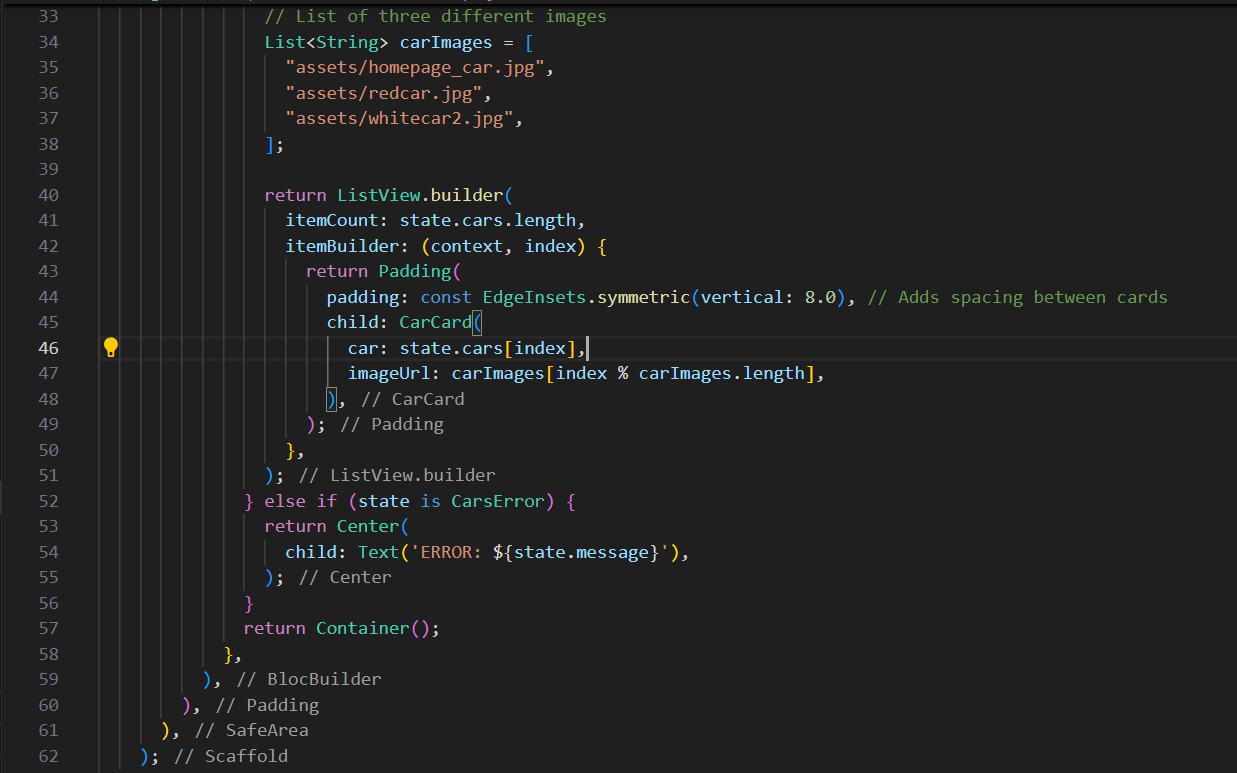
* **Encapsulation:** UI components are modularized into separate methods.
* **Abstraction:** \_emailAddress(), \_password(), \_signin(), and \_signup() hide implementation details.
* **Reusability:** The UI components follow a modular structure.
* **Navigation:** Uses Navigator.push() for smooth screen transitions.

1. **Conclusion**

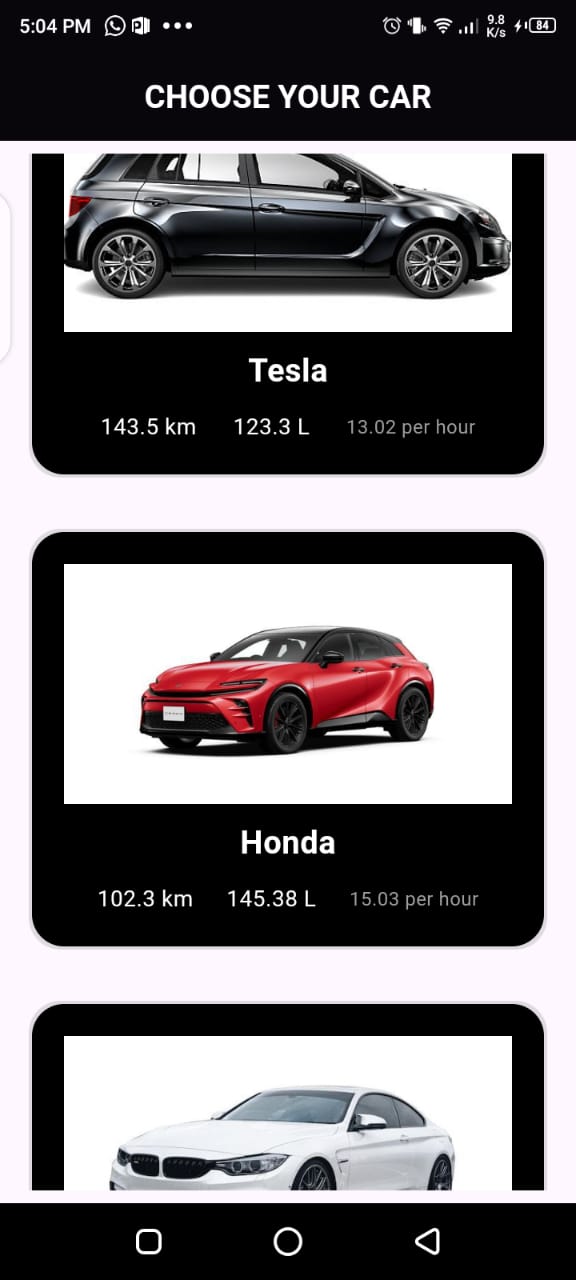
* This login page provides a **modern UI** with Firebase authentication.
* The **scrollable layout** ensures usability on small screens.
* It follows **best practices** for structuring Flutter UI code.

**Homepage:**

****

****

**Output:**

****

**Explanation of code:**

### **Homepage Widget Explanation**

This Homepage widget in Flutter displays a list of cars using the **BLoC (Business Logic Component) pattern**, ensuring a clean separation between UI and business logic. It listens to state changes and updates the UI accordingly.

### **1. AppBar Configuration**

* The AppBar has a **title ("CHOOSE YOUR CAR")**, centered in **bold white text**.
* The **background color is black**, while the foreground (icons) is white.

### **2. Using BLoC for State Management**

* BlocBuilder<CarBloc, CarState> listens for state changes and updates the UI.
* The three possible states are:
  1. **CarsLoading** → Displays a loading indicator.
  2. **CarsLoaded** → Displays the list of cars.
  3. **CarsError** → Shows an error message.

### **3. Handling the Loading State**

* If the state is CarsLoading, a CircularProgressIndicator() is displayed in the center.

### **4. Displaying the List of Cars**

* When CarsLoaded state is received, a **list of cars** is displayed using ListView.builder().
* A **list of three car images** is defined and cycled through to assign an image to each car.
* CarCard is used as a separate widget to display individual car details.

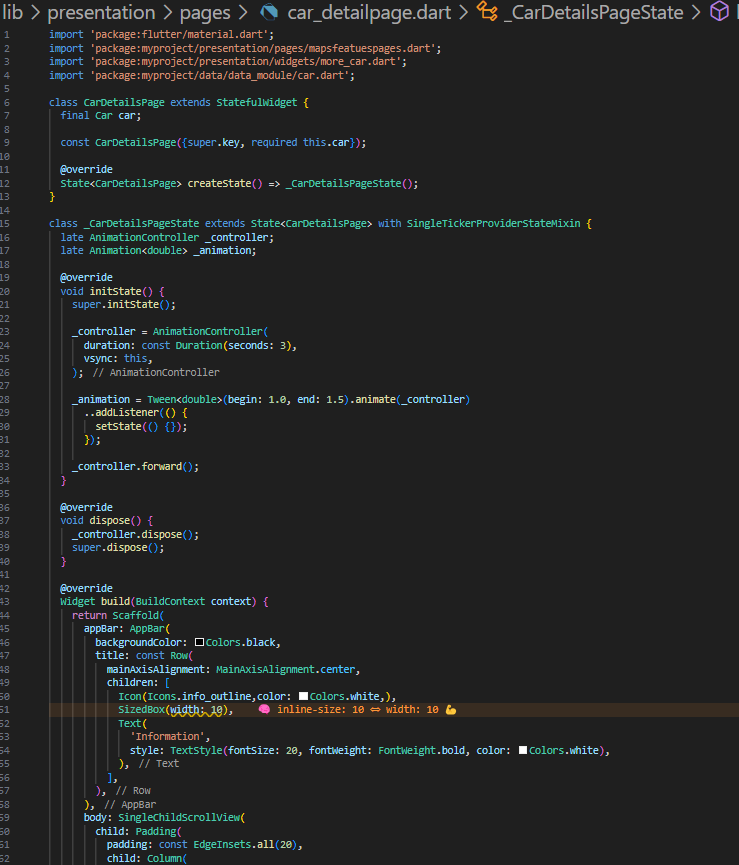
### **5. Handling Errors**

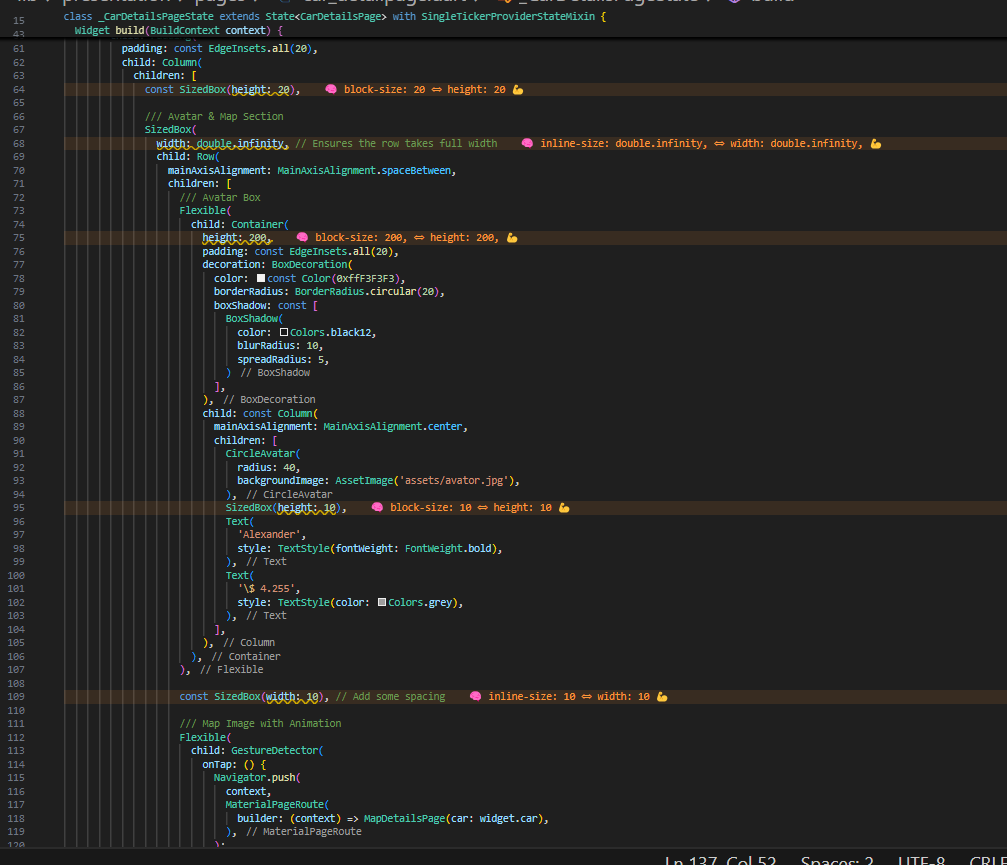
* If the state is CarsError, an error message is displayed in the center of the screen.

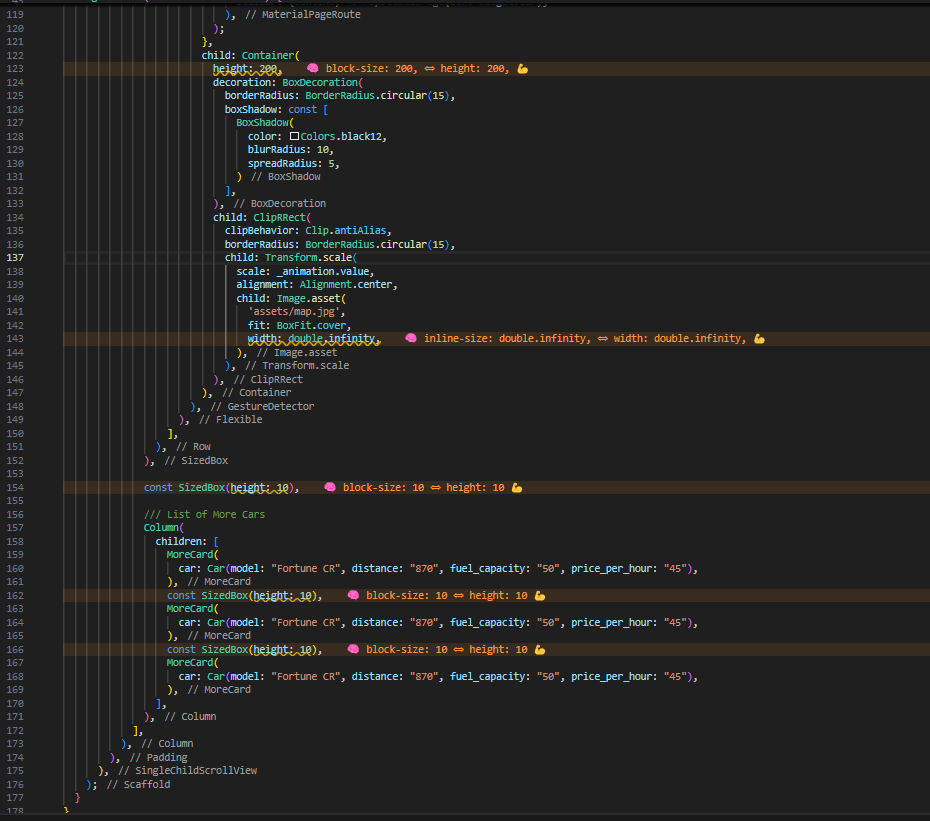
### **6. Object-Oriented Programming Concepts in Use**

* **Encapsulation** → The logic for fetching car data is handled in CarBloc, separate from the UI.
* **Abstraction** → The CarCard widget abstracts how each car is displayed.
* **Modularity** → The UI is cleanly separated from business logic, making the code more **reusable.**

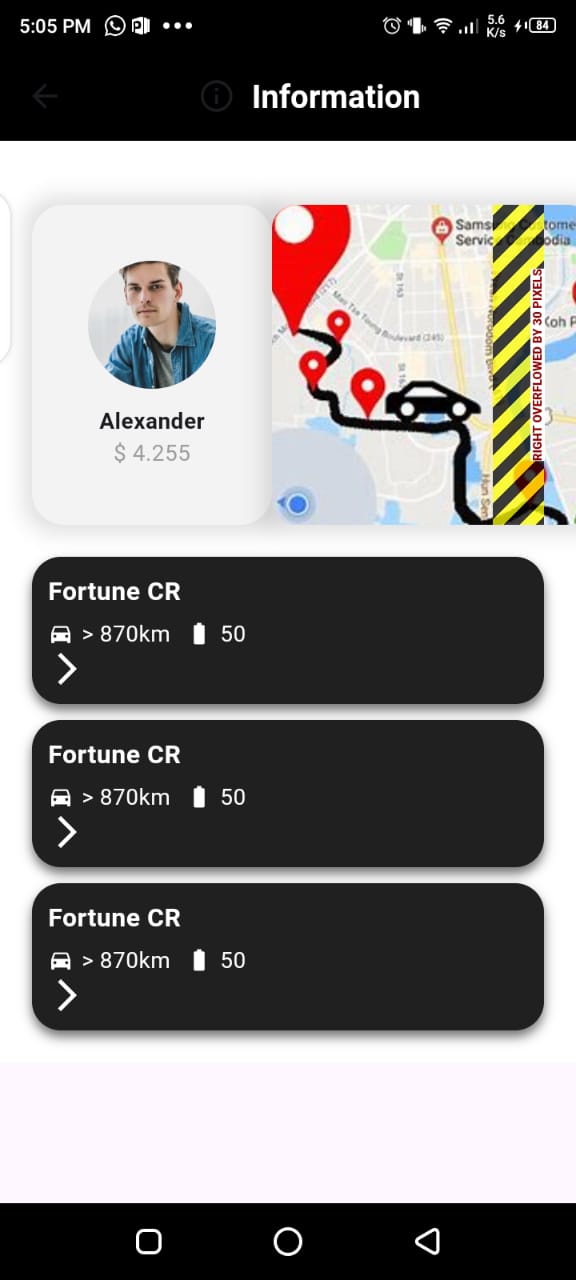
**Car Detail page:**

****

****

****

**Output:**

****

**Explanation:**

### **1. Overview**

The CarDetailsPage is a **stateful widget** that displays detailed information about a specific car, including its profile (avatar), map view (animated), and a list of similar cars.

### **2. Constructor & State Management**

* **Constructor**: The page accepts a Car object as an argument (required this.car), representing the selected car for which the details will be shown.
* **Stateful Widget**: It uses StatefulWidget because it manages an animation, making it necessary to rebuild the widget whenever the animation value changes.

### **3. Animation Controller**

* **Animation Controller**: An AnimationController named \_controller is set to run for **3 seconds**.
* **Animation**: A Tween<double> is used to scale the map image from **1.0 to 1.5**, which is applied to the Transform.scale widget for the map image.
* The animation is **triggered in initState()** and updates the UI every time the value of \_animation changes, causing the map to "zoom" in.

### **4. AppBar**

* The AppBar has a centered title "Information" with an **info icon** next to it.
* It has a black background, with white text and icons for contrast.

### **5. Body Layout**

The body is wrapped in a SingleChildScrollView to allow for scrolling when content overflows. The content is divided into three main sections:

#### **A. Avatar & Map Section**

* **Avatar Box**:
  + Contains a CircleAvatar with a profile picture (assets/avator.jpg), a name ("Alexander"), and a price (\$ 4.255).
  + It's displayed inside a Container with **rounded corners**, padding, and a subtle **box shadow** for a neat appearance.
* **Map Image (with Animation)**:
  + A **map image** (assets/map.jpg) is displayed and animated using the \_animation value (scale from 1.0 to 1.5).
  + The map image is clickable using GestureDetector, and when clicked, it navigates to the MapDetailsPage to show more details related to the map.

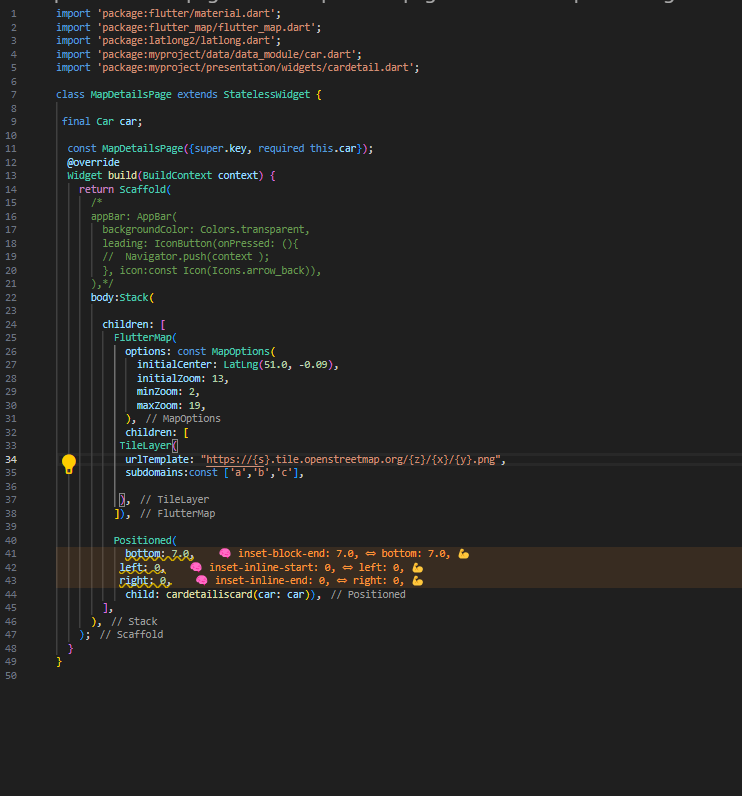
#### **B. List of More Cars**

* This section presents a list of more cars that are similar to the selected one. It uses the MoreCard widget to display each car's information (like model, distance, fuel capacity, and price per hour).

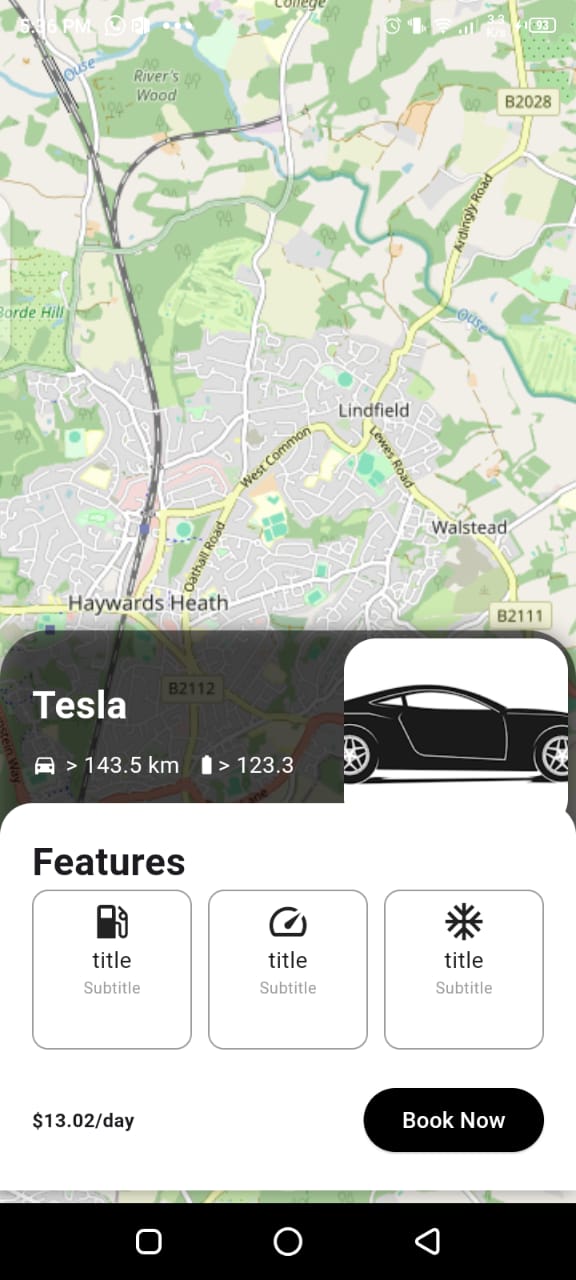
### **6. Object-Oriented Programming (OOP) Concepts**

* **Encapsulation**: The Car object encapsulates all relevant car details (model, price, etc.), making it easier to pass the data around the app without exposing unnecessary implementation details.
* **State Management**: The State class manages the widget's state, including the animation controller. This ensures the UI can be updated whenever the animation progresses.
* **Inheritance**: CarDetailsPage extends StatefulWidget, which is a part of Flutter's widget framework. This inheritance allows it to handle state and UI updates.

**Map Features page:**

****

**Ouput:**

****

**Explanation of code:**

The MapDetailsPage widget is designed to show a map using FlutterMap with OpenStreetMap tiles, and also displays the details of a specific car at the bottom of the map. Here's a breakdown of its components:

### **1. Constructor**

* The MapDetailsPage accepts a Car object (required this.car) which contains information about the car that will be displayed at the bottom of the map.

### **2. Body Layout**

* The body of the page is wrapped inside a **Stack** widget. This allows the map to be displayed in the background and the car details card (cardetailiscard) to be positioned on top of the map.

#### **A. FlutterMap**

* **FlutterMap**: This is used to render the map. It uses the latlong2 package to handle geographical coordinates and MapOptions for map settings.
  + **initialCenter**: Sets the initial center of the map (latitude 51.0, longitude -0.09), which is somewhere near London.
  + **initialZoom**: The initial zoom level is set to 13, which offers a detailed view of the area.
  + **minZoom and maxZoom**: These properties restrict the zoom range of the map.
* **TileLayer**: It defines the map tiles used for rendering the map. The URL template https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png retrieves OpenStreetMap tiles, and subdomains specifies different subdomains for better load distribution.

#### **B. Positioned Widget**

* The Positioned widget is used to position the **car details card** at the bottom of the screen.
  + **bottom: 7.0**: The car details card is positioned 7 pixels from the bottom.
  + **left: 0 and right: 0**: It stretches the card across the entire width of the screen.

#### **C. Car Details Card**

* The cardetailiscard widget is used to display the car's details, like the car model, price, or other properties. This widget is passed the car object so it can show relevant information about the selected car.

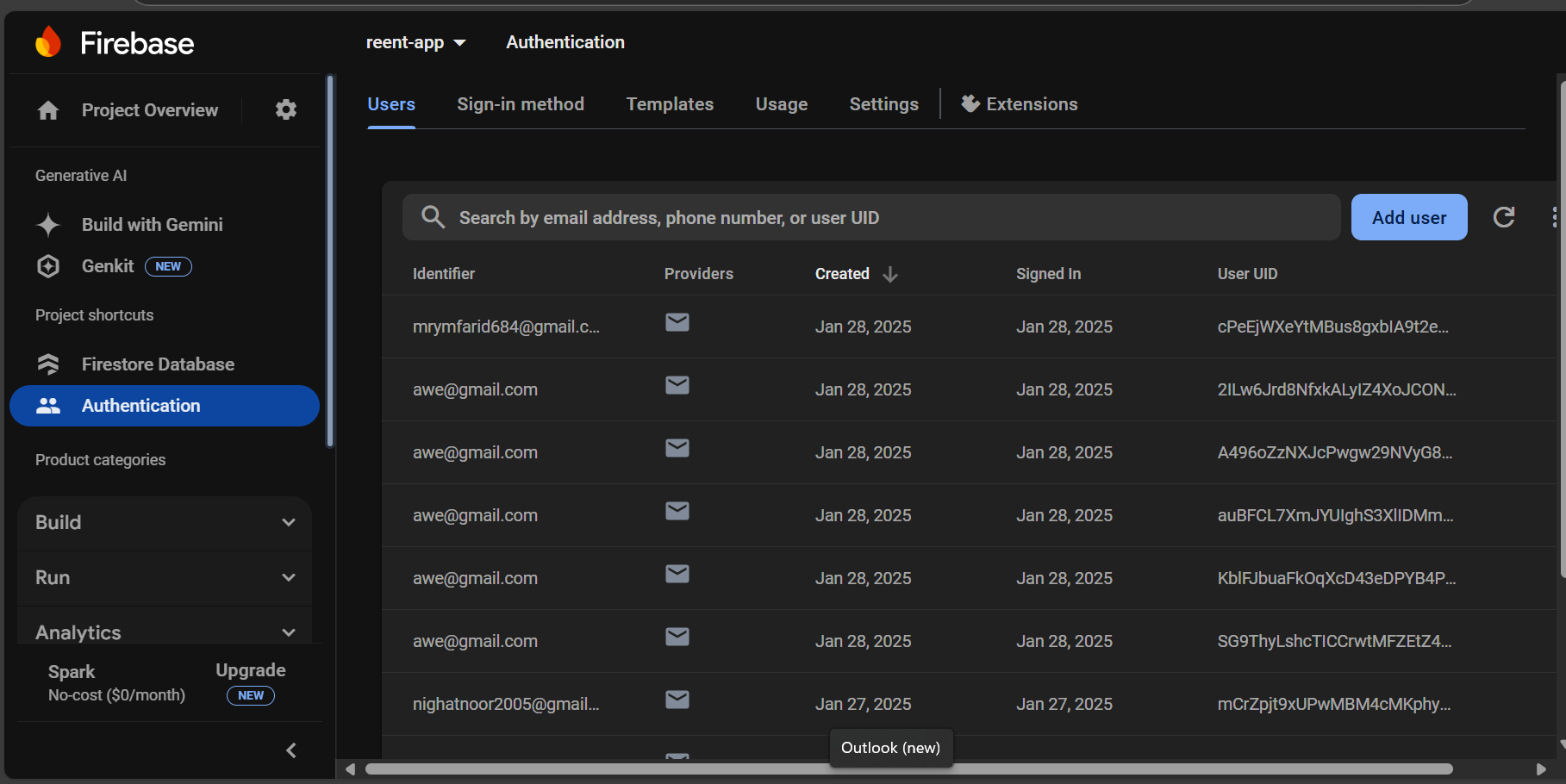
### **3. Future Improvements**

* You may want to pass in the actual location of the car by updating the initialCenter to use the car's latitude and longitude if available. You can get this from the Car object and update the map's initial position accordingly.
* You can also add **markers** on the map to represent car locations dynamically.

### **Summary**

This page displays an interactive map and the details of a car in a stacked layout. It uses FlutterMap to load OpenStreetMap tiles and places the car details card on top of the map. The page is simple and uses basic positioning with a stack to manage overlapping widgets.

**Firebase Authentication:**



### **Firebase Authentication:**

Firebase Authentication is handling user sign-in and registration in app. Based on the image:

* We have multiple users registered, with email-based authentication (@gmail.com email addresses).
* Each user has a **unique user identifier (UID)** assigned by Firebase.
* The table shows:
  + **Identifier:** The email address used for login.
  + **Providers:** Email authentication is being used.
  + **Created:** The date when the user registered.
  + **Signed In:** Last sign-in date.
  + **User UID:** A unique identifier for each user.

#### **How it Works in App:**

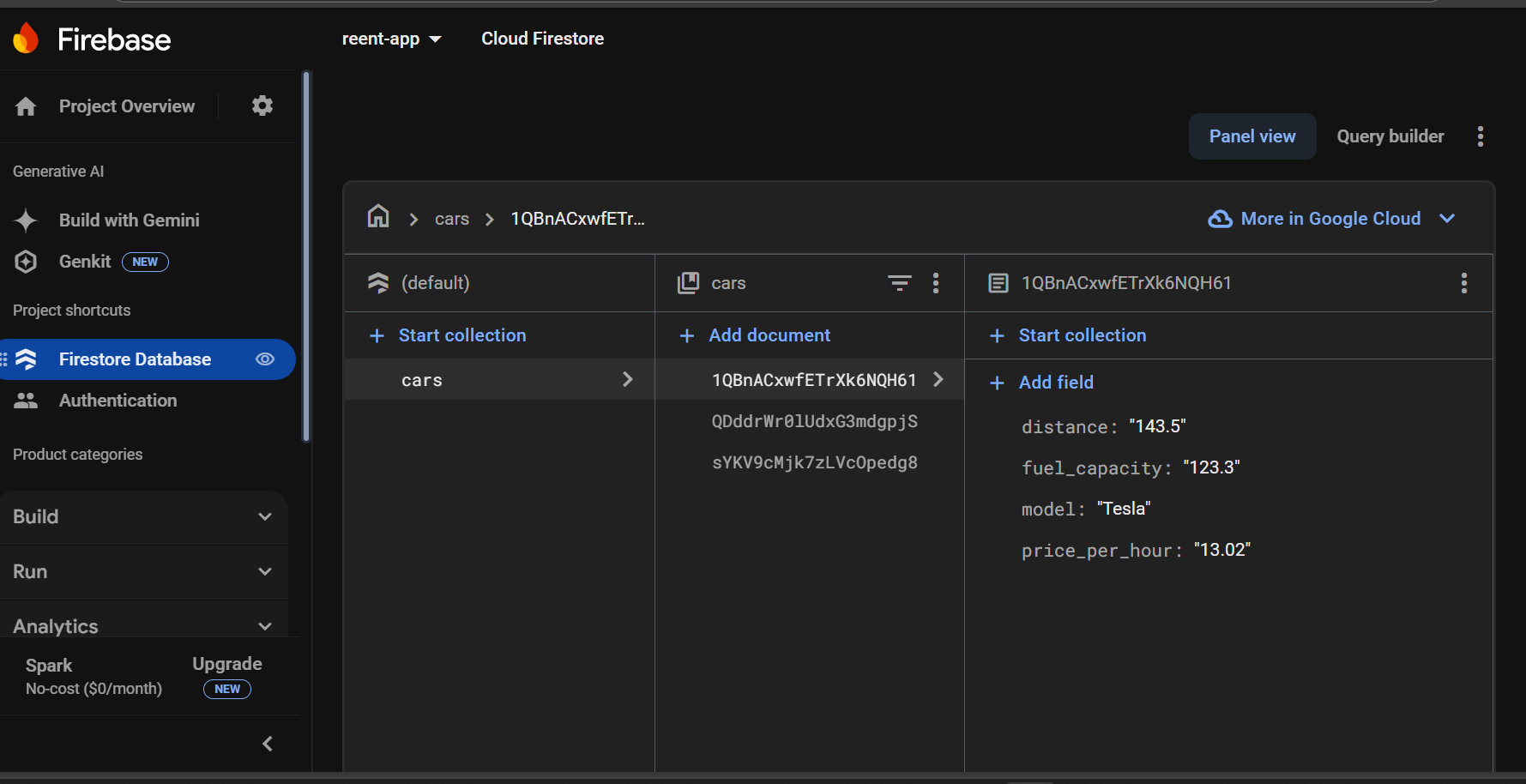
* When a user signs up, Firebase creates an account and assigns a **UID**.
* This UID can be used to associate user data in **Firestore**.
* We can retrieve user details using FirebaseAuth.instance.currentUser.

**Firebase data base:**

### **. Cloud Firestore Database :**

Our app stores car-related data in Firestore. The structure in the image shows:

* A **collection** named "cars".
* Inside this collection, there are **documents** with unique IDs (1QBnACxwFETrXk6NQH61).
* Each document represents a **car** and contains fields:
  + **model**: "Tesla" (Car model)
  + **distance**: "143.5" (Distance covered)
  + **fuel\_capacity**: "123.3" (Fuel capacity)
  + **price\_per\_hour**: "13.02" (Rental price per hour)



### **How These Connect to Your Code:**

* CarBloc is handling Firebase **data retrieval**.
* CarDetailsPage displays car info, including an animation and map feature (MapDetailsPage).
* Homepage lists cars dynamically, using Firestore images.
* The **authentication system** ensures that only signed-in users access features.

**Additional Changes:**

**Car Images Update:**

* **Previous Implementation:** The homepage previously displayed the same car image repeated multiple times for each car listed.
* **Change Implemented:** To enhance the visual appeal and provide variety, three distinct car images were added. These images are displayed dynamically, with each car in the list associated with a different image.
* **Reason for the Change:** This change was made to improve the user experience by making the car options more visually diverse and engaging. By using different images, users can better distinguish between the cars available for selection.
* **Implementation:** The ListView.builder was updated to assign a different image to each car in the list using a cyclic assignment. A list of three images was created, and each car was assigned one of the images using the modulus operator (%) to cycle through them.

**Integration of Sign-In and Sign-Up Pages with Firebase Authentication:**

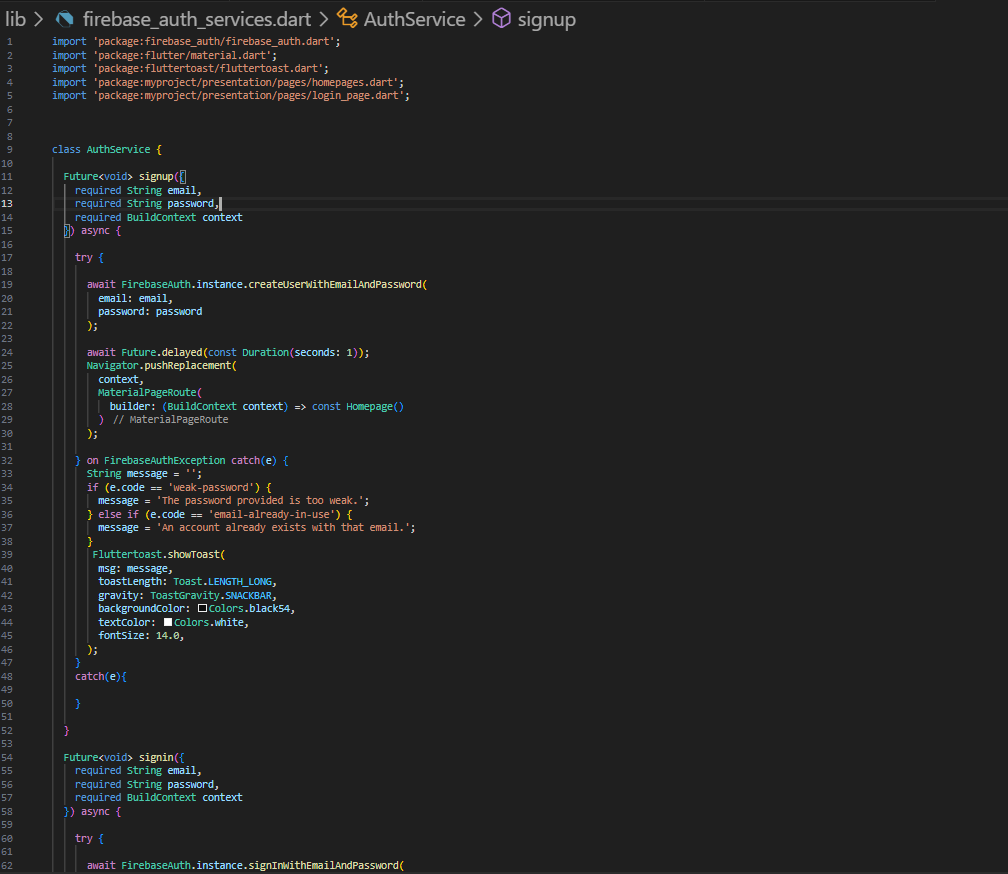
* **Previous Implementation:** The app previously featured an onboarding page followed by the homepage, with no user authentication process.
* **Change Implemented:** A new sign-in and sign-up page was introduced, integrating Firebase Authentication for user login and registration.
* **Reason for the Change:** This change was made to allow users to securely log in or register for the app using Firebase Authentication. Implementing this feature enhances the security and personalization of the app by allowing only authenticated users to access the main content.
* **Implementation:** The onboarding page was removed and replaced with a sign-in screen, where users can either sign in with their credentials or sign up for a new account. Firebase Authentication was utilized to handle user authentication, ensuring seamless login and registration experiences. The flow is now as follows: the user first encounters the sign-in page; upon successful authentication, they are redirected to the homepage.

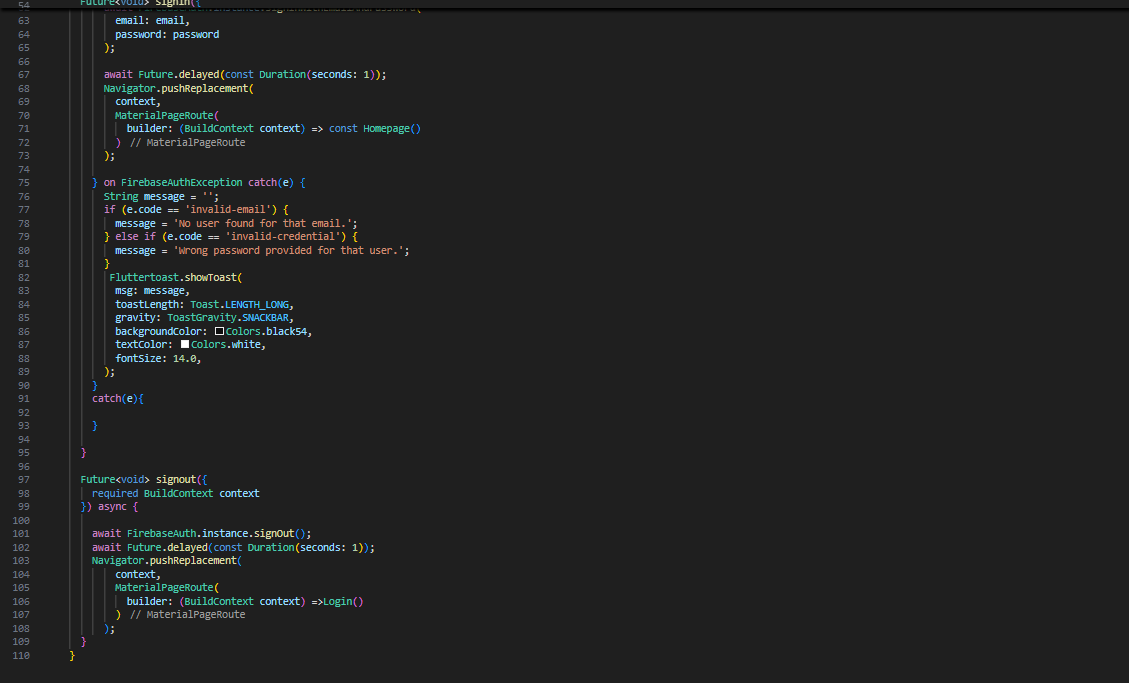
**Image Size Management Across the App:**

* **Previous Implementation:** Images in the app were not consistently managed in terms of size, which sometimes led to overflow or improper display of content.
* **Change Implemented:** Image sizes were controlled and adjusted across the entire app to ensure that they fit appropriately within the layout and prevent overflow.
* **Reason for the Change:** This change was made to maintain a consistent and responsive design, ensuring that images are displayed correctly on different screen sizes without causing layout issues or overflow.
* **Implementation:** The size of images was dynamically managed using constraints, such as BoxFit, and SizedBox, ensuring that each image adapts well to the screen size. This ensures images scale appropriately without disrupting the overall layout of the app, providing a better user experience across different devices.

**The globally defined widges in our app:**

**Firebase\_auth –services.dart:**

****

****

### **Explanation of AuthService (User-Defined Widget for Authentication):**

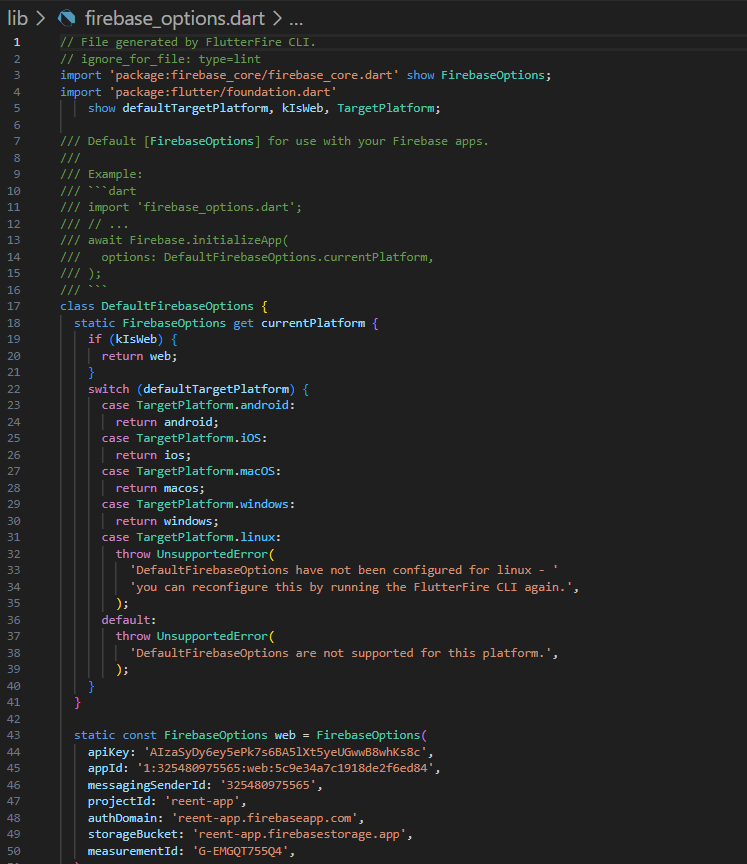
The AuthService class is a custom authentication service that manages user authentication using Firebase in a Flutter application. It includes three main functions:

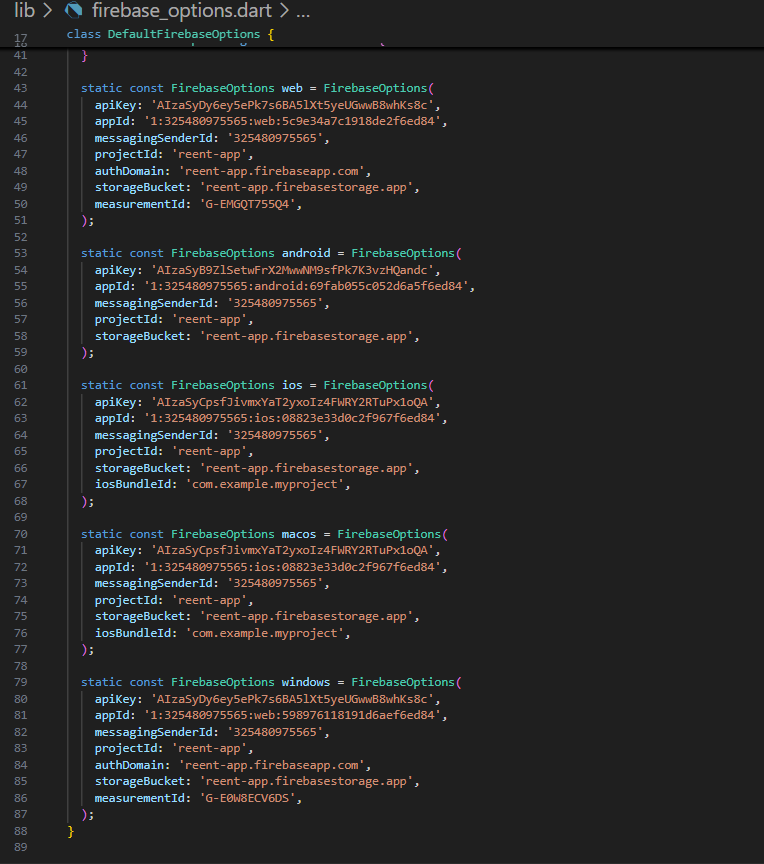
1. **Sign-Up Function (signup)**
   * This function registers a new user with an email and password using Firebase Authentication.
   * If the registration is successful, the user is redirected to the homepage.
   * Error handling is implemented to display messages if the password is too weak or if the email is already registered.
   * The function provides feedback using Fluttertoast to notify users of any authentication issues.
2. **Sign-In Function (signin)**
   * This function allows an existing user to log in using their email and password.
   * If authentication is successful, the user is navigated to the homepage.
   * Error handling ensures that appropriate messages are displayed if the email is not found or if the password is incorrect.
   * Similar to the sign-up function, Fluttertoast is used for instant feedback.
3. **Sign-Out Function (signout)**
   * This function logs out the currently signed-in user.
   * After signing out, the user is redirected to the login page.
   * A delay is added before navigation to ensure a smooth transition.

### **Features of the AuthService Class:Key**

* **Encapsulates authentication logic**, making it reusable across the app.
* **Uses Firebase Authentication** for secure login, sign-up, and logout functionality.
* **Implements error handling** to guide users in case of authentication failures.
* **Provides feedback through Fluttertoast**, enhancing user experience.
* **Utilizes navigation** to manage user flow after authentication actions.

**Firebase\_options.dart:**





### **Explanation of** firebase\_options.dart **(User-Defined Widget in Flutter):**

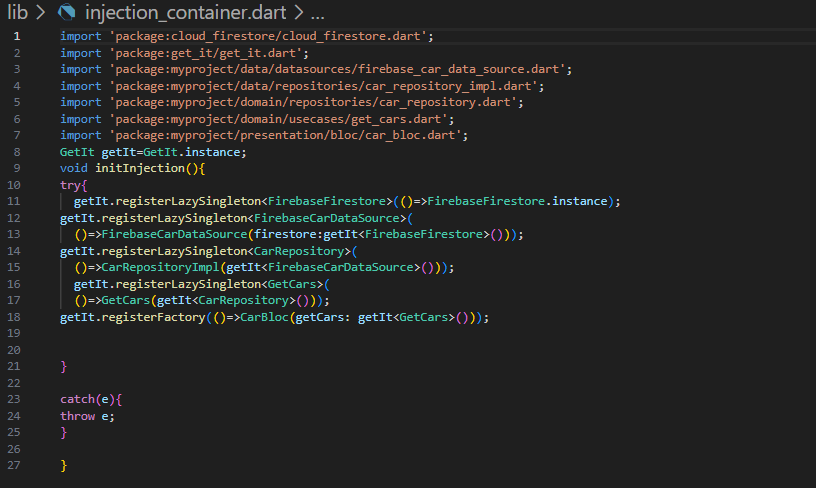
The firebase\_options.dart file is a **user-defined class** that provides Firebase configuration settings for different platforms in a Flutter app. It is generated automatically using the **FlutterFire CLI** and is essential for initializing Firebase services like authentication, Firestore, and storage.

This file contains a class called DefaultFirebaseOptions, which selects the correct Firebase settings based on the platform (Web, Android, iOS, macOS, Windows). It ensures that the app connects to Firebase using the appropriate API keys and project details without requiring manual setup for each platform.

The class determines the **target platform** using defaultTargetPlatform and kIsWeb. If an unsupported platform (like Linux) is detected, it throws an error. The Firebase configurations for each platform are stored as FirebaseOptions objects containing parameters such as apiKey, appId, projectId, and storageBucket.

To use this configuration in the Flutter app, Firebase is initialized using DefaultFirebaseOptions.currentPlatform. This ensures that Firebase works correctly on different platforms while keeping the setup **clean, modular, and scalable**.

**Injection\_container.dart:**



### **Explanation of Dependency Injection Using GetIt in Flutter:**

The given code implements **dependency injection** in a Flutter project using the GetIt package. Dependency injection helps manage dependencies centrally, reducing tight coupling and improving modularity, scalability, and testability.

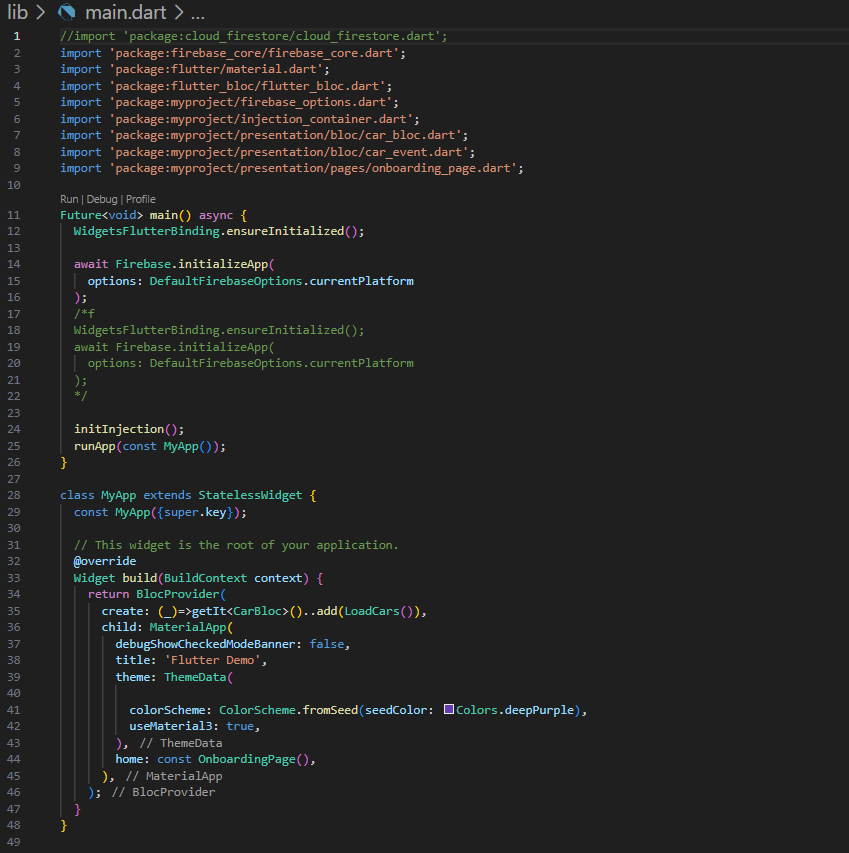
#### **1. Purpose of** initInjection()

The initInjection() function registers dependencies using GetIt, ensuring that required instances are created and managed efficiently throughout the app.

#### **2. Key Components**

* **GetIt getIt = GetIt.instance;**
  + GetIt.instance provides a global service locator to manage dependencies.
* **Dependency Registration:**
  + registerLazySingleton: Creates a **single instance** of a class that is reused whenever required.
  + registerFactory: Creates a **new instance** whenever needed.

**Main.dart:**



### **Explanation of the Main Flutter App Initialization and BLoC Setup:**

This code demonstrates the initialization of a Flutter app with Firebase, dependency injection using GetIt, and the BLoC (Business Logic Component) pattern to manage state. Below is a detailed breakdown of the code:

### **1. Firebase Initialization**

In the main() function, Firebase is initialized before the app starts:

* **Firebase.initializeApp()** is called to initialize Firebase services, using platform-specific configuration from DefaultFirebaseOptions.currentPlatform.
* **WidgetsFlutterBinding.ensureInitialized()** ensures that the Flutter framework is fully initialized before performing any asynchronous operations, such as Firebase initialization.

This setup ensures that Firebase is ready to be used across different platforms (Android, iOS, Web, etc.).

### **2. Dependency Injection Setup**

The **initInjection()** function is invoked to register all required dependencies using **GetIt**, a service locator. This includes Firebase, data sources, repositories, use cases, and BLoCs. By registering these dependencies, they can be easily accessed throughout the app.

### **3. Root Widget:** MyApp

MyApp is the root widget of the app. It is wrapped in a **BlocProvider** that provides the CarBloc to the widget tree:

* **BlocProvider**: Manages the lifecycle of the CarBloc instance and ensures it’s available to all descendants.
* **getIt<CarBloc>()** retrieves the CarBloc instance from the dependency injection container.
* **..add(LoadCars())**: Immediately dispatches the LoadCars event when the app starts, triggering the loading of car data.

The app uses **MaterialApp** for UI design, with a color scheme defined by ColorScheme.fromSeed(seedColor: Colors.deepPurple) and enables **Material 3** components.

### **4. BLoC (Business Logic Component) Integration:**

The **BLoC pattern** is used to manage state and business logic. The CarBloc listens for events like LoadCars() and manages the state related to car data.

* **CarBloc**: Handles the events and updates the state of the app, such as loading or displaying cars.
* **LoadCars()**: This event is triggered to fetch car data, likely from Firebase or another data source.

### **5. Theme and UI Configuration:**

The MaterialApp widget also configures the app's theme and appearance:

* **ColorScheme.fromSeed(seedColor: Colors.deepPurple)**: Defines the app's color scheme using a seed color (deep purple).
* **useMaterial3: true**: Activates Material 3 design components for the app’s UI.

### **6. Onboarding Page:**

The home screen is set to the **OnboardingPage**:

* **OnboardingPage** likely serves as the initial screen, providing information or guiding new users through the app.

### **Summary:**

* Firebase is initialized to ensure that Firebase services are available.
* **GetIt** is used to register and inject dependencies like Firebase, data sources, repositories, and BLoCs.
* The **BLoC pattern** is implemented to manage the state of the app by handling business logic and user actions.
* The app uses **Material Design 3** for its theme, and the **OnboardingPage** is set as the starting screen for users.