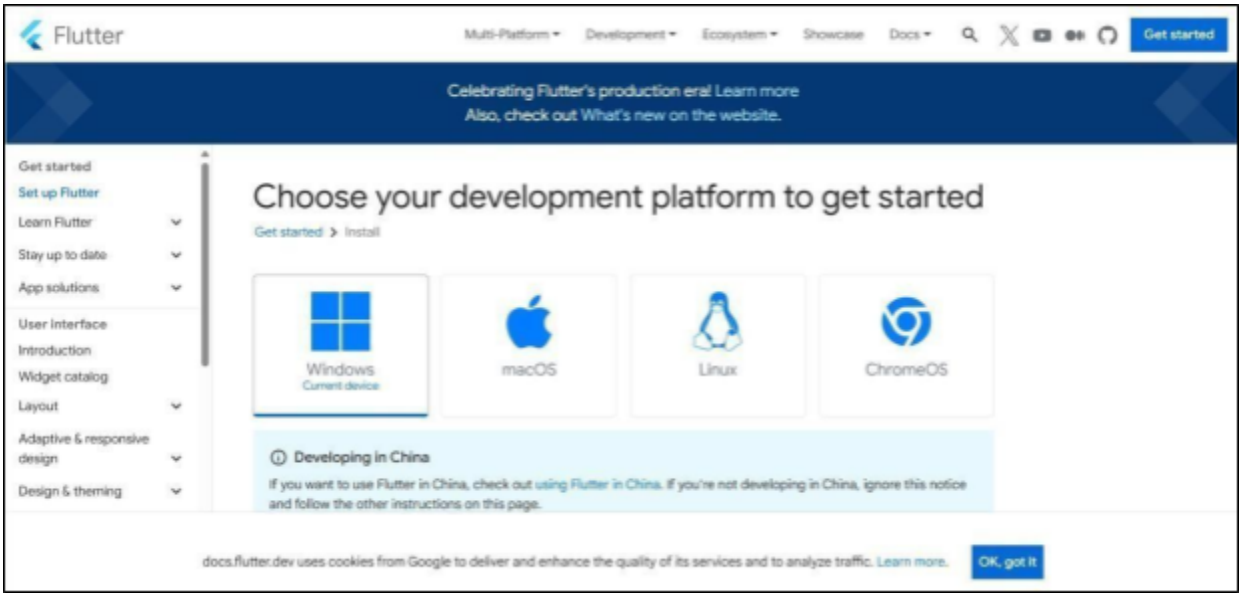
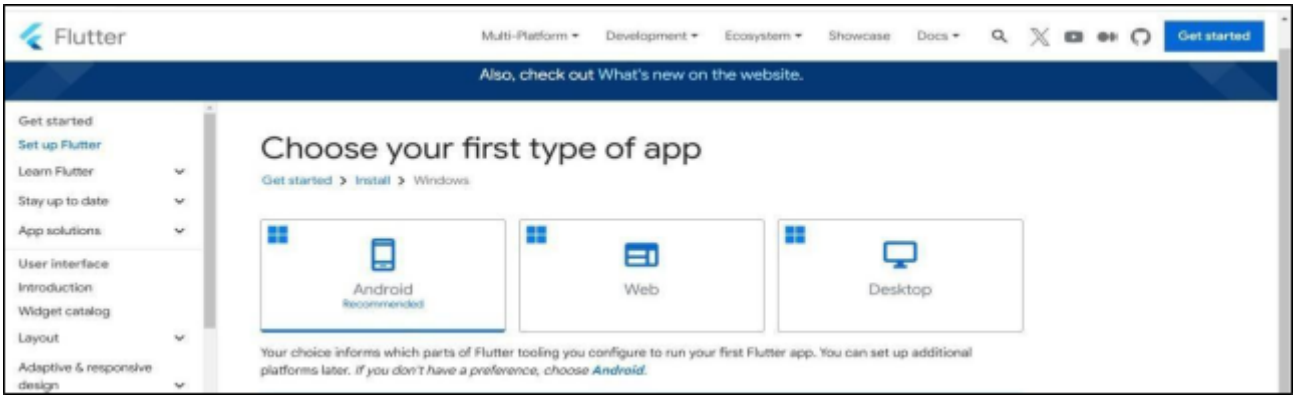
AIM: - Installation and Configuration of Flutter Environment.

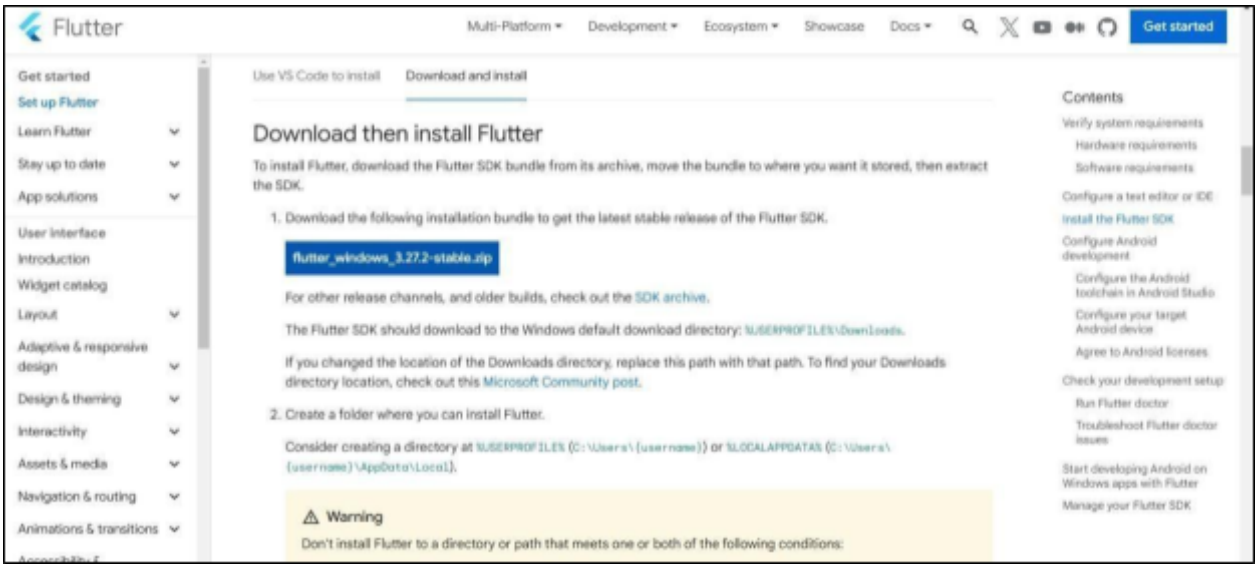
Step 1: Go to the official Flutter website: https://docs.flutter.dev/get-started/install



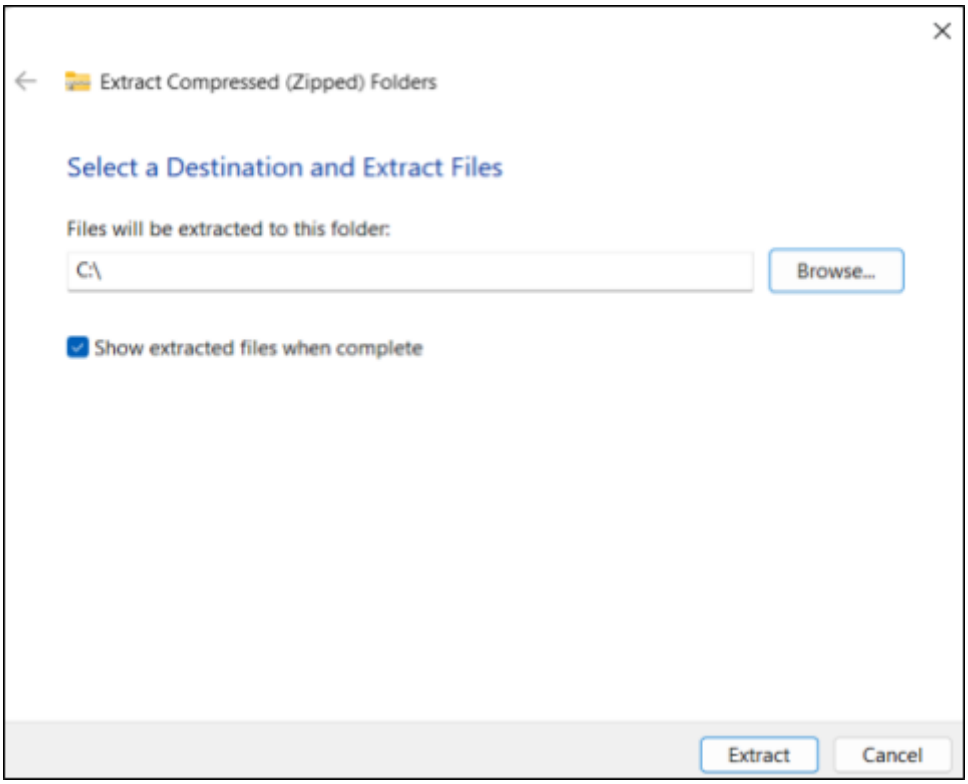
Step 2: To download the latest Flutter SDK, click on the Windows icon > Android



Step 3: For Windows, download the stable release (a .zip file).



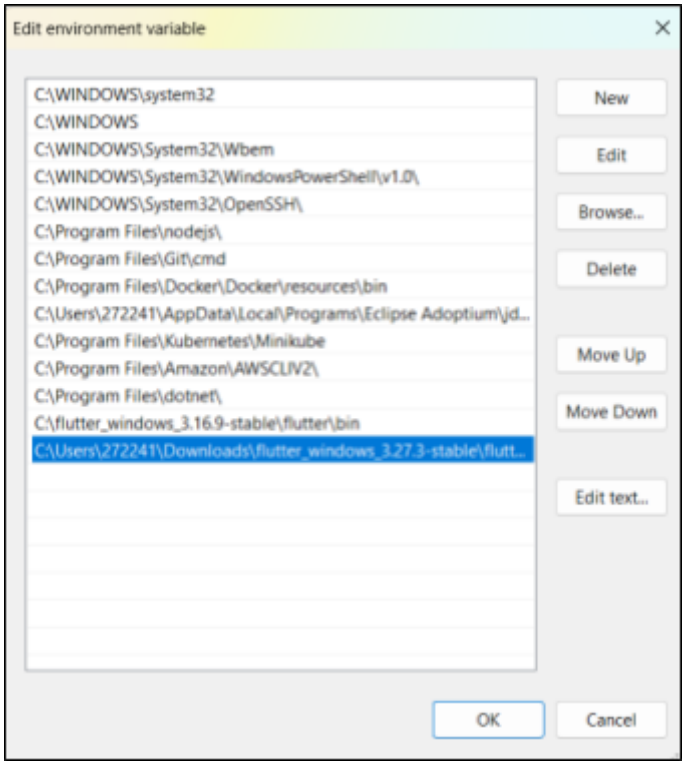
Step 4: Extract the ZIP file to a folder (e.g., C:\flutter).



Step 5 :- Add Flutter to System PATH

Right-click on the Start Menu > System > Advanced system settings > Environment Variables. Under System Variables, find Path and click Edit.

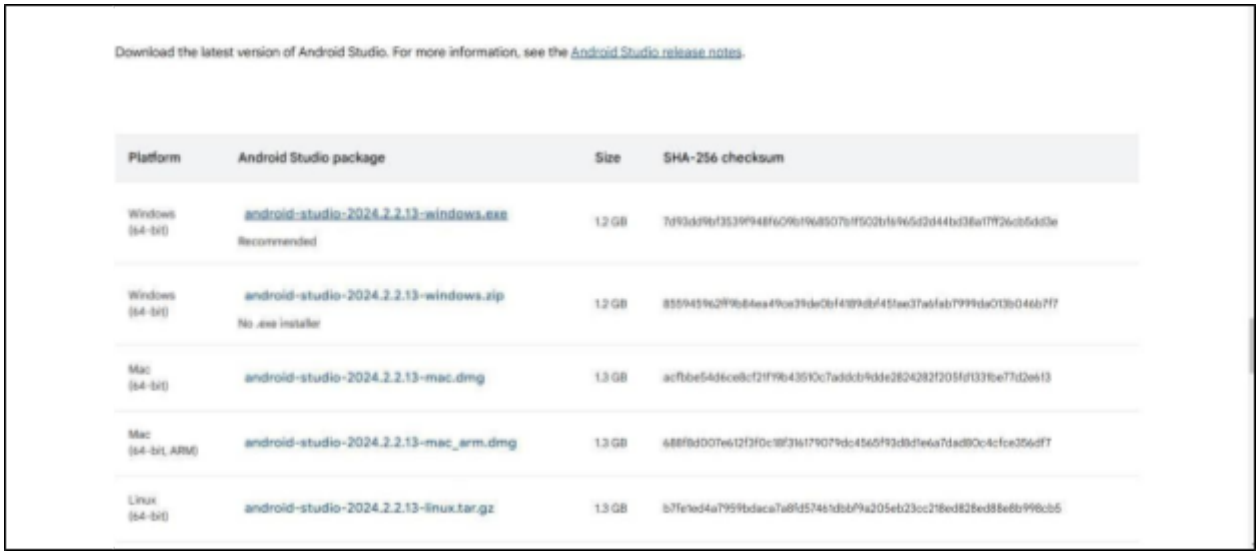
Add the full path to the flutter/bin directory (e.g., C:\flutter\bin).



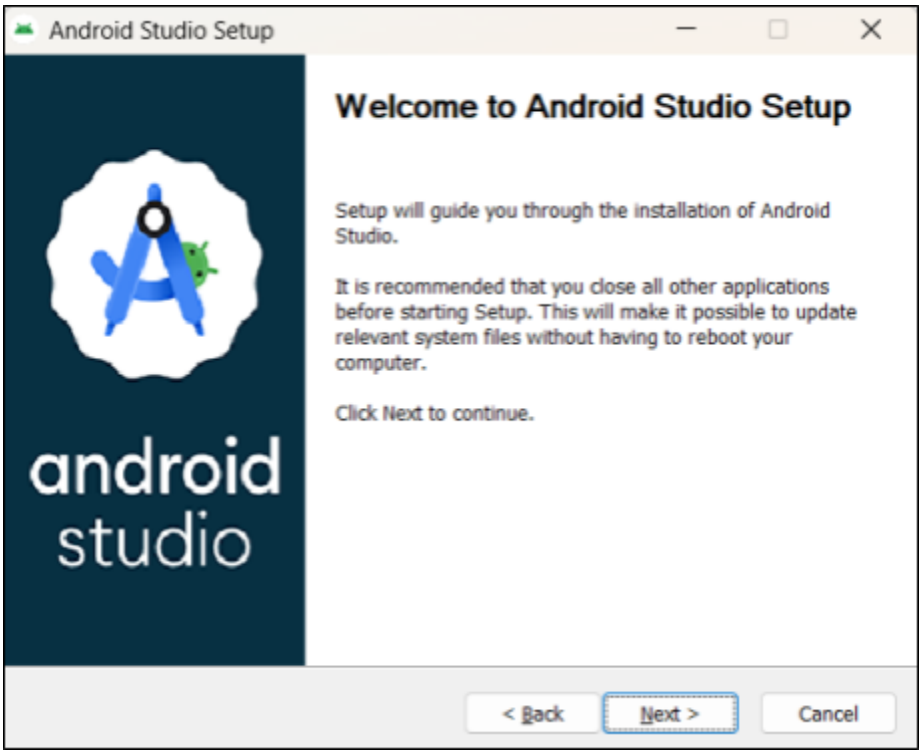
Step 7:- Run the $ flutter doctor command. This command checks for all the requirements of Flutter app development and displays a report of the status of your Flutter installation

```
C:\Windows\System32>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 3.27.4, on Microsoft Windows [Version 10.0.26100.3624], locale en-IN)
[√] Windows Version (Installed version of Windows is version 10 or higher)
[√] Android toolchain - develop for Android devices (Android SDK version 35.0.0)
[√] Chrome - develop for the web
[√] Visual Studio - develop Windows apps (Visual Studio Community 2022 17.9.0)
[√] Android Studio (version 2024.2)
[√] VS Code (version 1.99.0)
[√] VS Code, 64-bit edition (version 1.91.0)
[√] Connected device (3 available)
[√] Network resources

• No issues found!
```

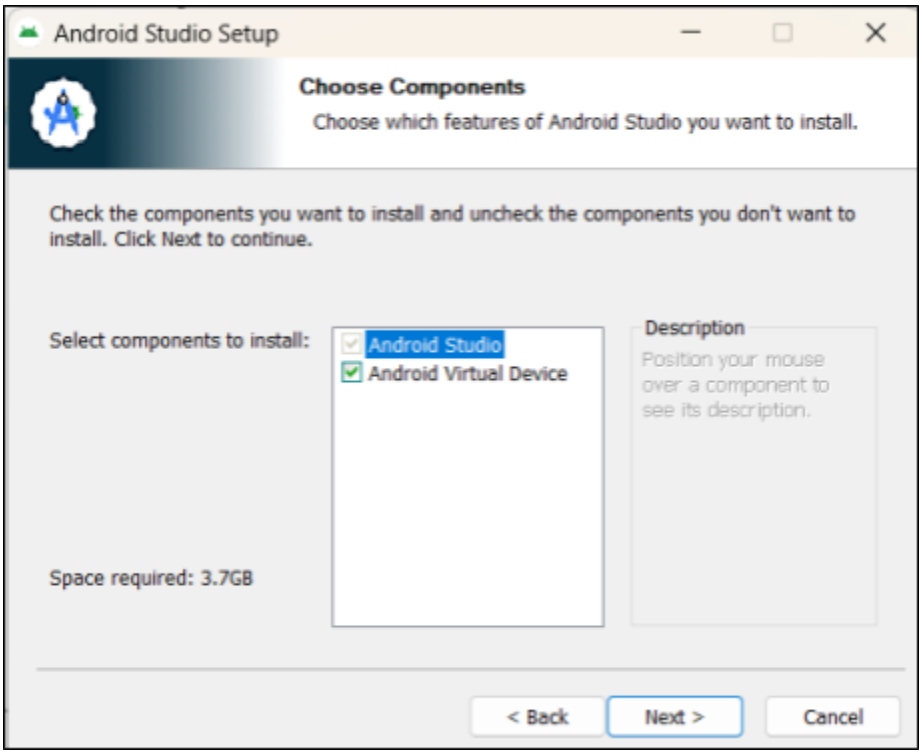
Step 8 : - Go to Android Studio and download the installer.



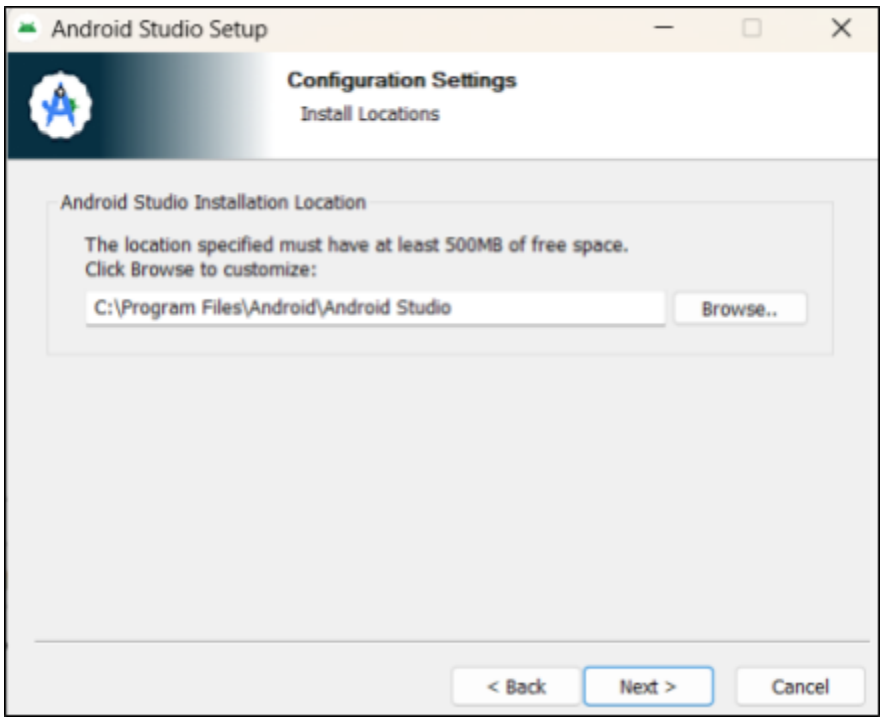
Step 8.1: - When the download is complete, open the .exe file and run it. You will get the following dialog box



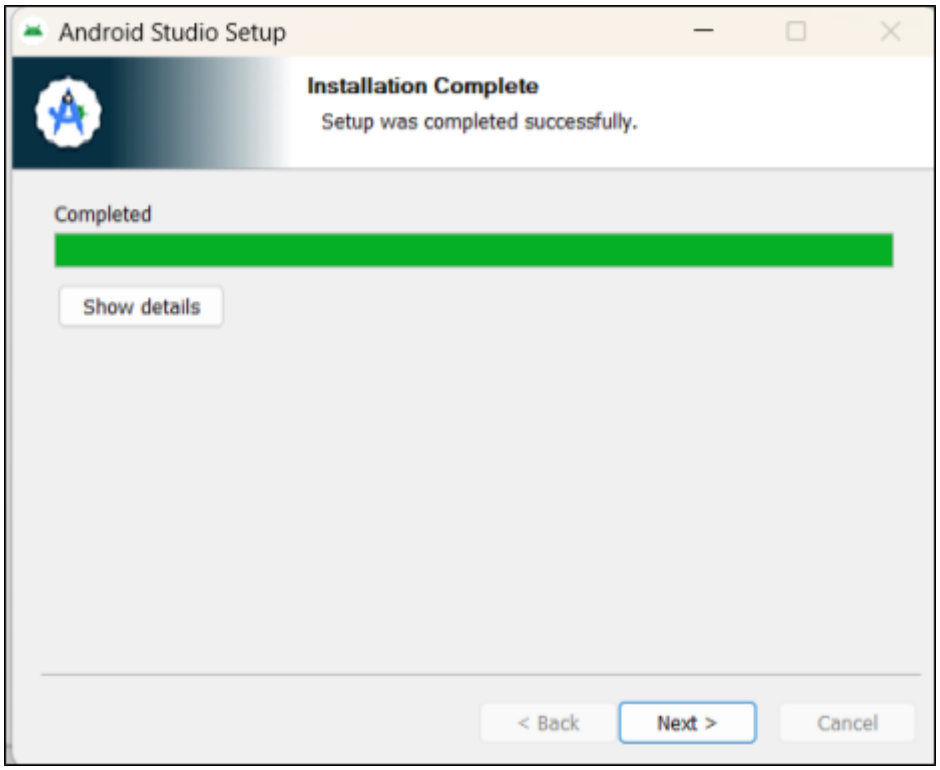
Step 8.2: - Select all the Checkboxes and Click on 'Next' Button.

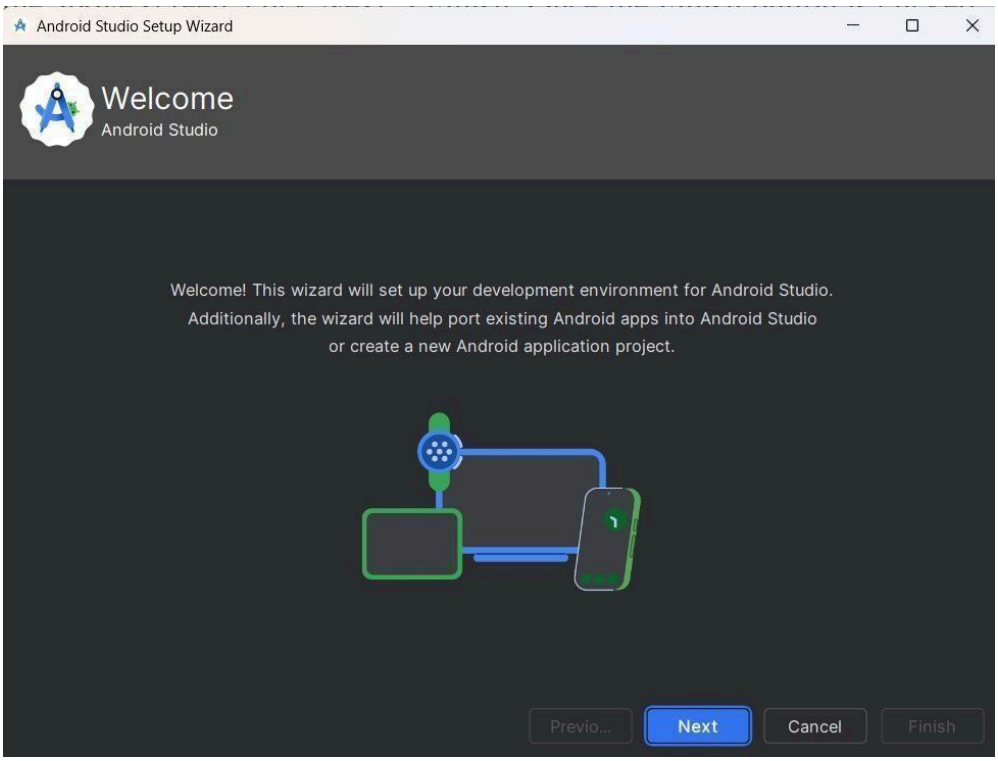


Step 8.3: - Change the destination as per your convenience and click on ‘Next’ Button.

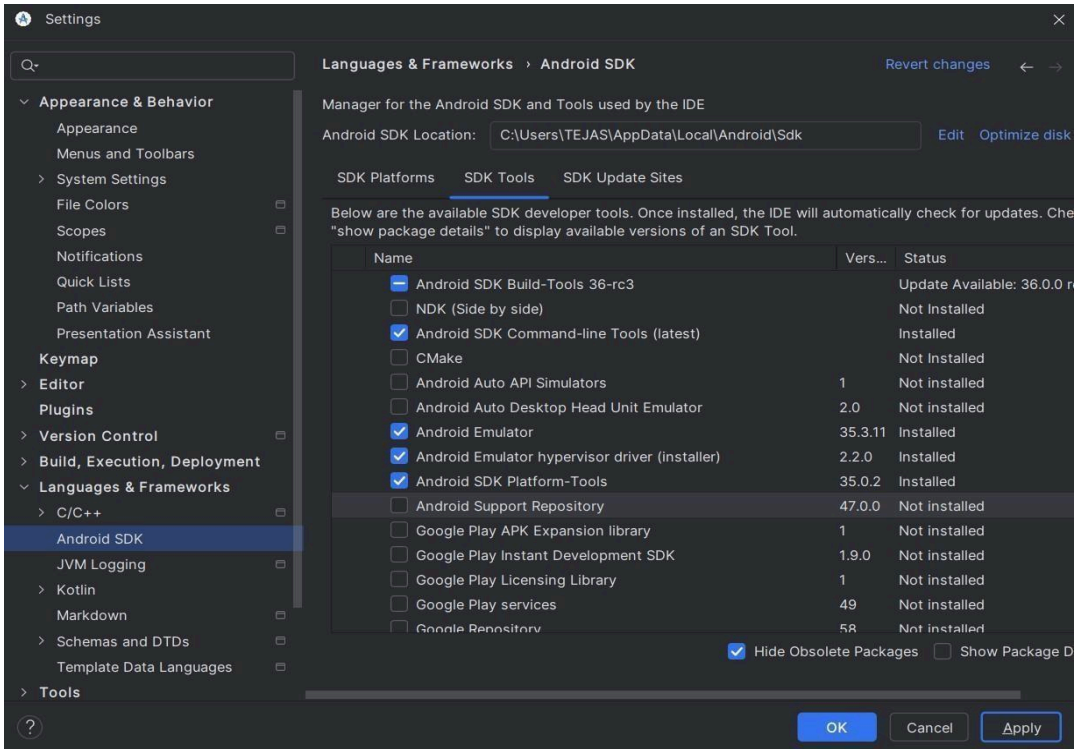


Step 8.4: - Follow the steps of the installation wizard. Once the installation wizard completes, you will get the following screen.





Step 8.5: - Go to Preferences > Appearance & Behavior > System Settings > Android SDK. Select the SDK Tools tab and check Android SDK Command-line Tools and Install it.

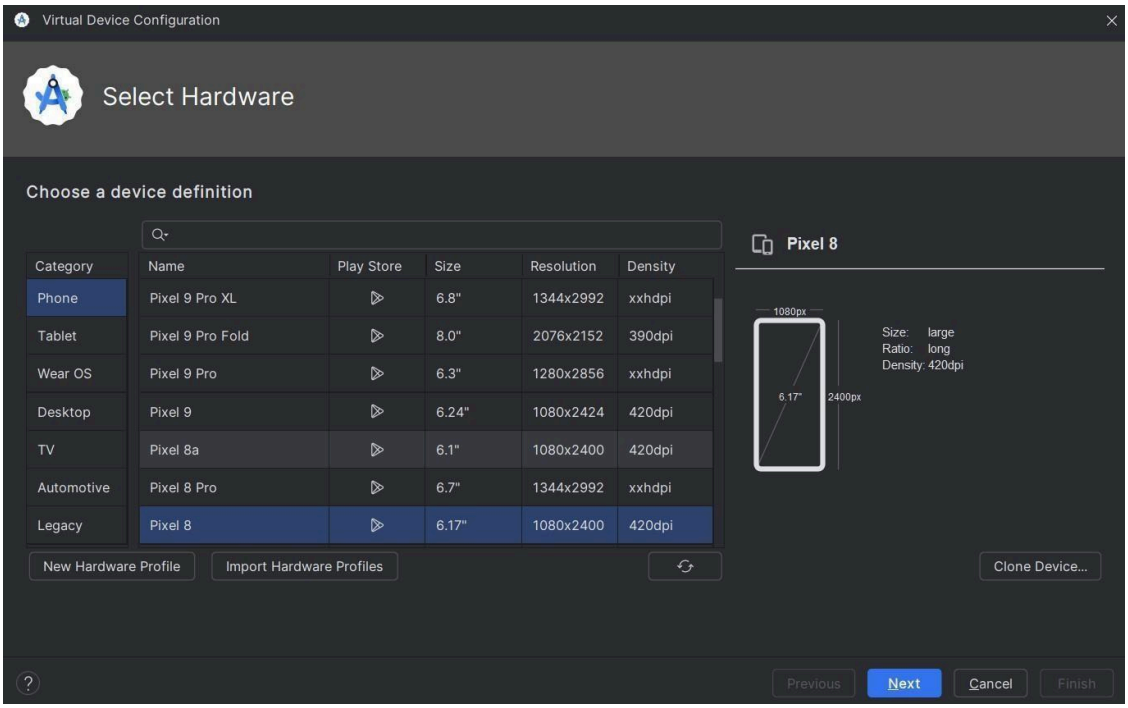


Step 9: - Open a terminal and run the following command

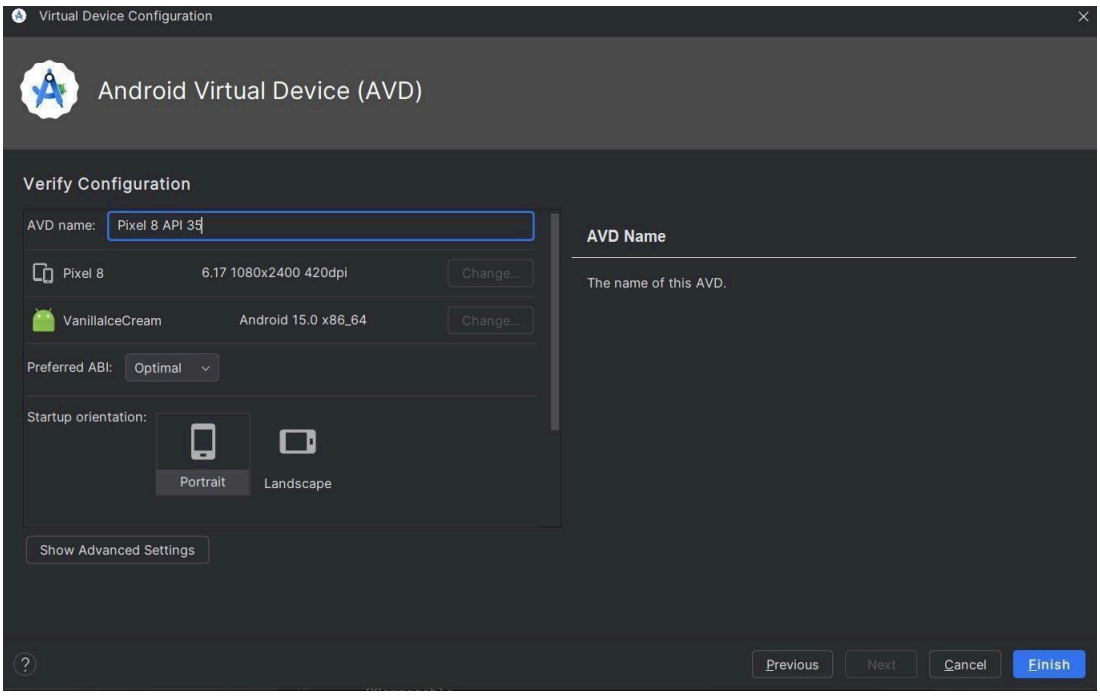
```
:\Windows\System32>flutter doctor --android-licenses
                                                                                Warning: Errors during XML parse:
                                                                                Warning: Additionally, the fallback load
r failed to parse the XML.
======================================] 100% Computing updates...
ll SDK package licenses accepted.
```

```
C:\Windows\System32>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 3.27.4, on Microsoft Windows [Version 10.0.26100.3624], locale en-IN)
[√] Windows Version (Installed version of Windows is version 10 or higher)
[√] Android toolchain - develop for Android devices (Android SDK version 35.0.0)
[√] Chrome - develop for the web
[√] Visual Studio - develop Windows apps (Visual Studio Community 2022 17.9.0)
[√] Android Studio (version 2024.2)
[√] VS Code (version 1.99.0)
[√] VS Code, 64-bit edition (version 1.91.0)
[√] Connected device (3 available)
[√] Network resources

• No issues found!
```

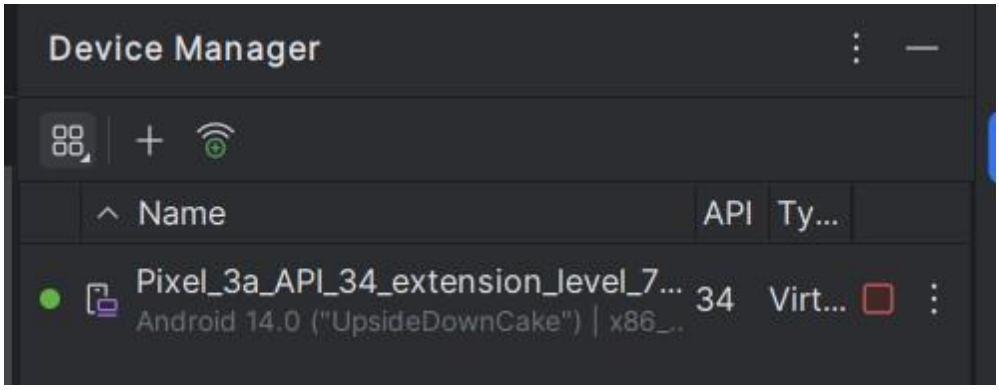
Step 10: - Next, you need to set up an Android emulator. It is responsible for running and testing the Flutter application



Step 10.1: - Open Android Studio and go to Tools > AVD Manager. Create a new virtual device.

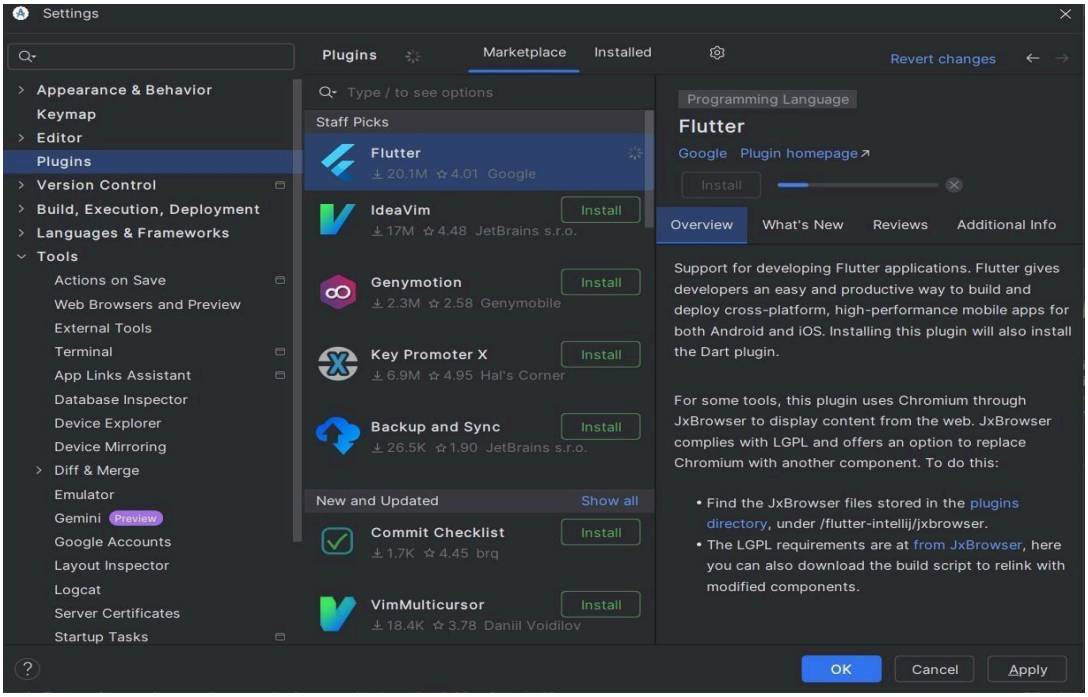


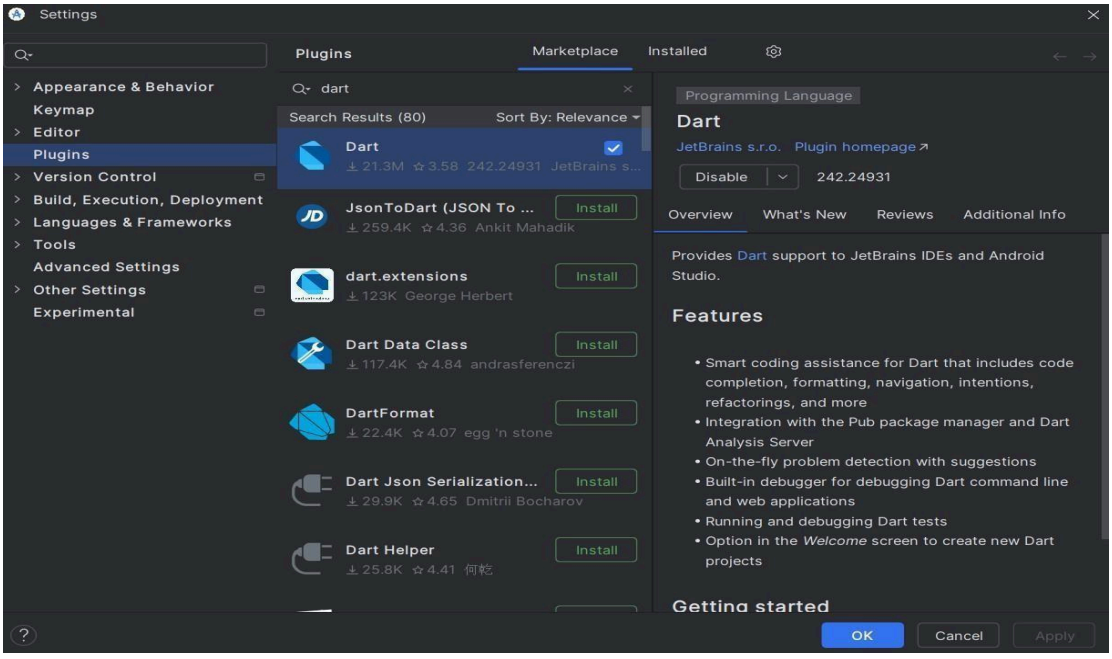
Step 10.2: - Click on the icon pointed into the red color rectangle. The Android emulator displayed as below screen



Step 11: - Now, install Flutter and Dart plugin for building Flutter application in Android Studio. These plugins provide a template to create a Flutter application, give an option to run and debug Flutter application in the Android Studio itself

Step 11.1: - Open the Android Studio and then go to File->Settings->Plugins. Now, search the Flutter plugin. If found, select Flutter plugin and click install





Step 11.2: - Restart the Android Studio

Step 12: - Go to File > New Project > Create Flutter Project, then select the project name and location, and click Next to proceed.

Aim: To design Flutter UI by including common widgets.

Theory:

Flutter follows a widget-based approach where everything in the UI is a widget. Widgets can be classified into two main types:

- Stateless Widgets: Do not change their state once built (e.g., Text, Container).
- Stateful Widgets: Can update dynamically based on user interaction (e.g., TextField, Checkbox).

Commonly Used Widgets in Flutter-

(a) Scaffold Widget

The Scaffold widget provides the basic structure for a Flutter app, including an AppBar, Drawer, FloatingActionButton, and BottomNavigationBar. It is a fundamental widget used to create a standard screen layout in Flutter.

(b) Container Widget

A Container is a box model widget that can hold other widgets. It is commonly used for adding padding, margins, borders, and background decorations.

(c) Row and Column Widgets

- Row: Arranges widgets horizontally.
- Column: Arranges widgets vertically.
  These two widgets are fundamental for designing layouts in Flutter.

(d) ListView Widget

The ListView widget is used for displaying a scrollable list of items. It is useful for showing large amounts of data dynamically.

(e) Stack Widget

The Stack widget is used to place widgets on top of each other. This is useful for creating overlapping UI elements such as banners, profile images, or layered designs.

(f) ElevatedButton Widget

The ElevatedButton widget is used for clickable buttons with a raised effect. It is a commonly used button in Flutter applications.

(g) TextField Widget

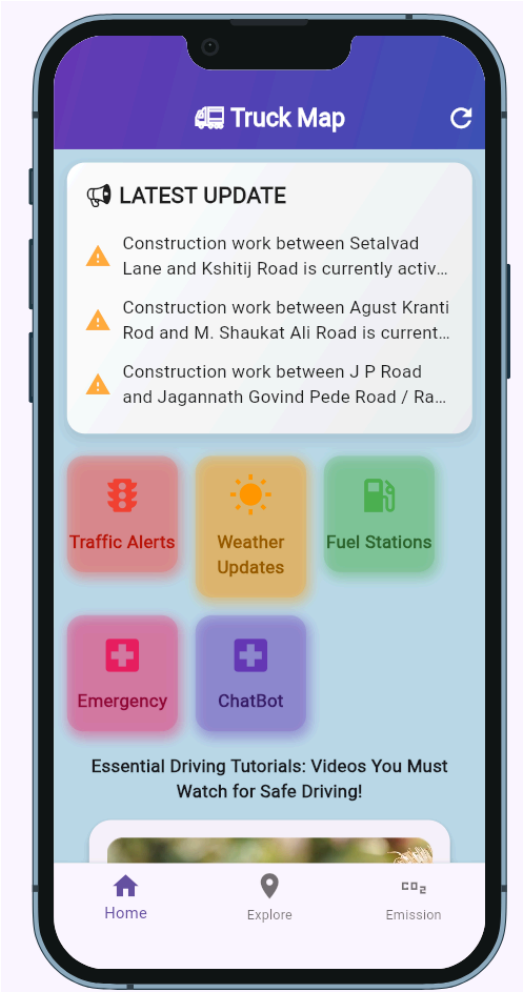
The TextField widget is used to take user input, such as entering a name, email, or password. It is commonly used in forms and authentication screens.

Code :

```dart
class HomePage extends StatefulWidget {
  const HomePage({super.key});

  @override
  State<HomePage> createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  int currentPage = 0;
  List<Widget> pages = [
    HomeSectionPage(),
    MapPageVersion(),
    CarbonEmissionPage(),
  ];
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: IndexedStack(
        index: currentPage,
        children: pages,
      ), // IndexedStack
      bottomNavigationBar: BottomNavigationBar(
        type: BottomNavigationBarType.fixed,
        iconSize: 28,
        currentIndex: currentPage,
        onTap: (value) {
          setState(() {
            currentPage = value;
          });
        },
```

```dart
  // 📌 Search Results List
  if (searchResults.isNotEmpty)
    Container(
      color: ■Colors.white,
      child: ListView.builder(
        shrinkWrap: true,
        itemCount: searchResults.length,
        itemBuilder: (context, index) {
          var result = searchResults[index];
          return ListTile(
            title: Text(result['address']['freeformAddress']),
            onTap: () {
              double lat = result['position']['lat'];
              double lon = result['position']['lon'];
              _moveToLocation(lat, lon);
            },
          ); // ListTile
        },
      ), // ListView.builder
    ), // Container
],
```

Output : -



Conclusion:

Flutter's widget-based architecture allows for flexible and efficient UI design. By using common widgets like `Scaffold`, `Container`, `Row`, `Column`, `ListView`, `Stack`, `ElevatedButton`, and `TextField`, developers can build responsive and interactive user interfaces with ease.

Aim: To include icons, images, fonts in Flutter app

Theory:

Using Icons in Flutter

Icons in Flutter can be added using the built-in Material Icons or custom icon packs.

(a) Material Icons

Flutter provides a collection of built-in Material Icons, which can be used with the Icon widget.

Eg: Icon(Icons.home, size: 30, color: Colors.blue)

(b) Custom Icons

If you need icons that are not available in the Material Icons set, you can use external icon packs like:

- Font Awesome (font_awesome_flutter package)
- Custom SVG Icons (flutter_svg

  package) Eg in pubspec.yaml file -

  dependencies:
  font_awesome_flutter: ^10.5.0

In code -

```
import 'package:font_awesome_flutter/font_awesome_flutter.dart';

IconButton(
  icon: FaIcon(FontAwesomeIcons.heart, color: Colors.red),
  onPressed: () {},
)
```

Adding Images in Flutter

Images can be loaded in Flutter from different sources like assets, network, or memory.

(a) Using Network Images

Network images are loaded from an online URL. Example:

Eg: Image.network("https://example.com/sample.jpg", width: 200, height: 150)

(b) Using Asset Images

To use images from the local project folder (assets/), follow these steps:

1. Place the image inside the assets/images/ folder.
2. Declare the image in pubspec.yaml:

```
flutter:
  assets:
    - assets/images/sample.png
```

In code: Image.asset("assets/images/sample.png", width: 200, height: 150)

Adding Custom Fonts in Flutter

Custom fonts improve the visual identity of an app.

Steps to Add a Custom Font:

1. Download the font and place it inside the assets/fonts/ folder.
2. Declare the font in pubspec.yaml:

```
flutter:
  fonts:
    - family: CustomFont
      fonts:
        - asset: assets/fonts/CustomFont-Regular.ttf
        - asset: assets/fonts/CustomFont-Bold.ttf
          weight: 700
```

In code

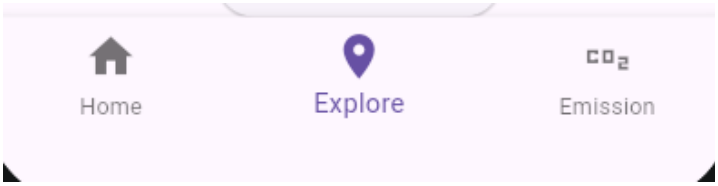
```
- Text(
  "Hello, Flutter!",
  style: TextStyle(fontFamily: "CustomFont", fontSize: 20, fontWeight: FontWeight.bold),
)
```

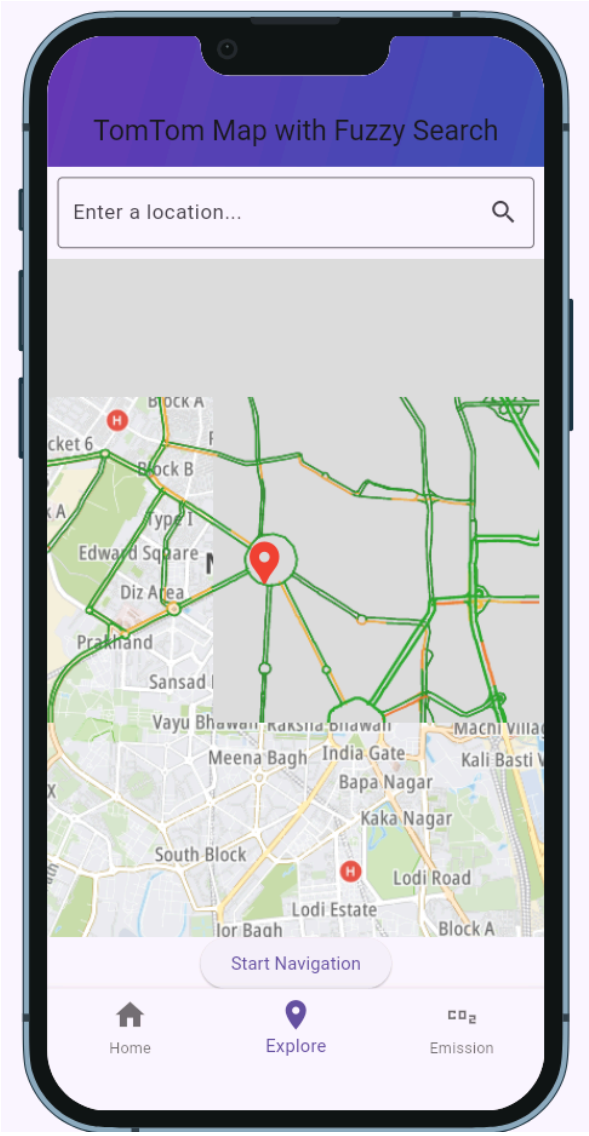
main.dart file

```
final markers = <Marker>[
  // Current Location Marker
  Marker(
    width: 50.0,
    height: 50.0,
    point: currentLocation!,
    child: const Icon(Icons.location_pin, color:
Colors.blue, size: 40),
  ),

  // Searched Location Marker
  if (searchedLocation != null)
    Marker(
      width: 50.0,
      height: 50.0,
      point: searchedLocation!,
      child: const Icon(Icons.location_pin, color:
Colors.red, size: 40),
    ),

  // Turn Markers
  ...turnMarkers.map(
    (turn) => Marker(
      width: 30.0,
      height: 30.0,
      point: turn["location"],
      child: _getTurnIcon(turn["turnType"]),
    ),
  ),
];
```

```
Icon _getTurnIcon(String turnType) {
  switch (turnType.toLowerCase()) {
    case "turn-right":
      return Icon(Icons.arrow_right_alt, color:
Colors.green, size: 30);

    case "turn-left":
      return Icon(Icons.arrow_left, color:
Colors.blue, size: 30);

    case "u-turn":
      return Icon(Icons.u_turn_left, color:
Colors.orange, size: 30);

    case "straight":
      return Icon(Icons.arrow_upward, color:
Colors.grey, size: 30);

    case "roundabout":
      return Icon(Icons.sync, color:
Colors.purple, size: 30);

    default:
      return Icon(Icons.circle, color:
Colors.black, size: 20);
  }
}
```

Output :-



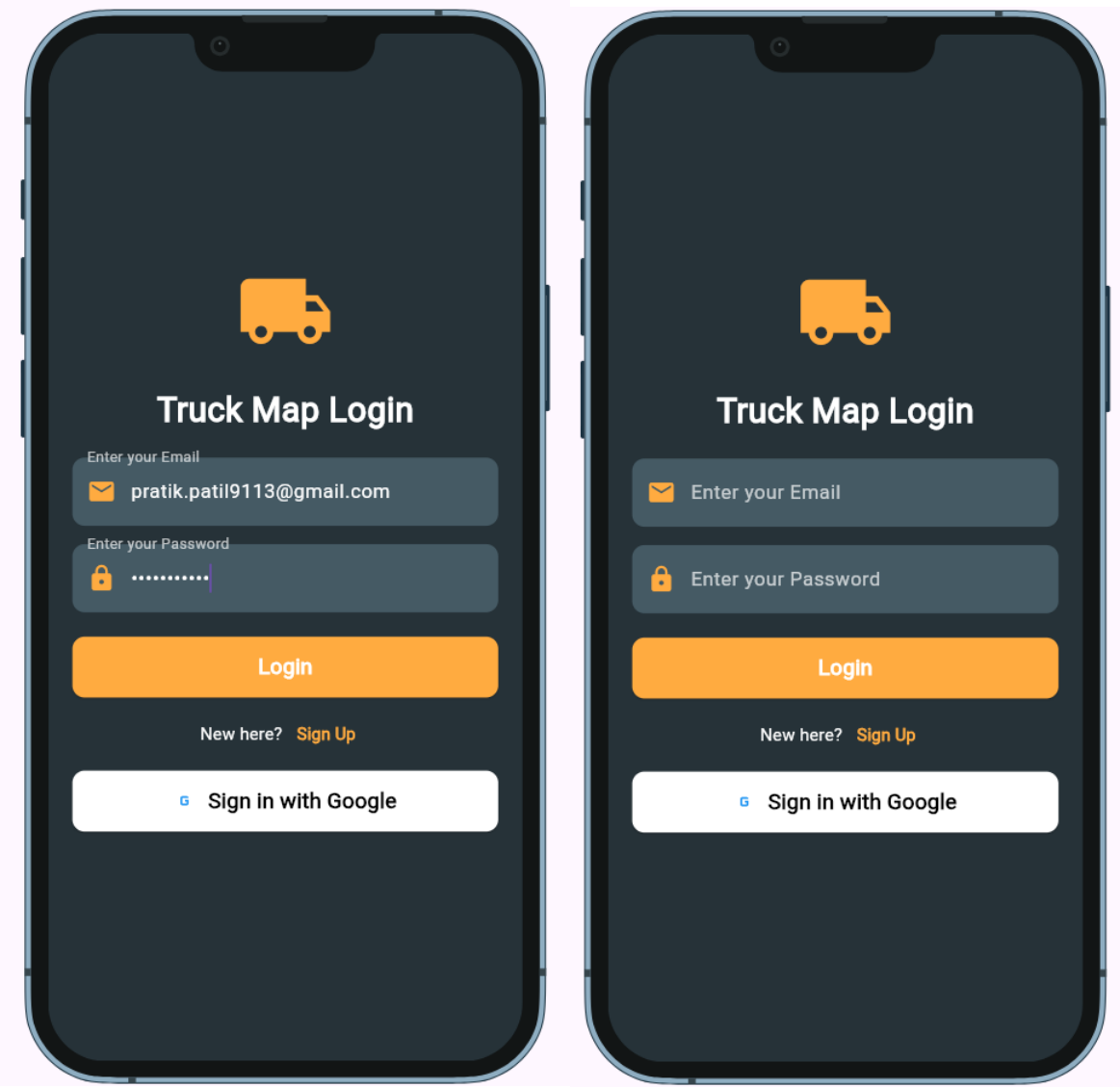


Conclusion : -

Flutter offers powerful and flexible support for incorporating visual elements like icons and images to enhance UI design. Developers can easily use **built-in Material Icons** for common UI elements, or integrate **custom icons** using packages like `font_awesome_flutter` or `flutter_svg` for more design flexibility. Additionally, images in Flutter can be loaded from various sources such as assets, network, or memory, allowing dynamic and visually appealing user experiences. Mastery of these tools enables developers to create rich, intuitive, and engaging mobile interfaces with ease.

Aim: create an interactive form using form widgets in flutter.

Theory:



Flutter provides a comprehensive set of widgets for building forms, allowing developers to create interactive user interfaces for collecting user input. These form widgets streamline the process of handling user input validation, submission, and data processing.

Form Widget:

The Form widget is the foundation for creating forms in Flutter. It acts as a container for form fields and provides methods for form validation and submission. The Form widget

manages the form state internally and provides access to the FormState object, which can be used to interact with the form fields.

TextFormField Widget:

The TextFormField widget is used to create text input fields within a form. It provides various properties for customizing the appearance and behavior of the input field, such as decoration, validation, and input formatting. Developers can specify validators to enforce input constraints and error messages to provide feedback to users when input validation fails.

```
const SizedBox(height: 20),
Text(
  isLogin ? 'Truck Map Login' : 'Truck Map Sign Up',
  style: const TextStyle(
    fontSize: 28,
    fontWeight: FontWeight.bold,
    color: Colors.white,
  ), // TextStyle
), // Text
const SizedBox(height: 20),
_buildTextField(emailController, 'Enter your Email', Icons.email),
const SizedBox(height: 15),
_buildTextField(
    passwordController, 'Enter your Password', Icons.lock,
    isPassword: true),
const SizedBox(height: 20),
_buildButton(isLogin ? 'Login' : 'Sign Up',
    createUserWithEmailAndPassword),
const SizedBox(height: 10),
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: [
    Text(
      isLogin ? 'New here?' : 'Already registered?',
      style: const TextStyle(color: Colors.white),
    ), // Text
    TextButton(
      onPressed: () {
        setState(() {
          isLogin = !isLogin;
        });
      },
```

```dart
Widget _buildTextField(
    TextEditingController controller, String hint, IconData icon,
    {bool isPassword = false}) {
  return TextField(
    controller: controller,
    obscureText: isPassword,
    decoration: InputDecoration(
      labelText: hint,
      prefixIcon: Icon(icon, color: Colors.orangeAccent),
      filled: true,
      fillColor: Colors.blueGrey[700],
      border: OutlineInputBorder(
        borderRadius: BorderRadius.circular(10),
        borderSide: BorderSide.none,
      ), // OutlineInputBorder
      labelStyle: const TextStyle(color: Colors.white70),
    ), // InputDecoration
    style: const TextStyle(color: Colors.white),
  ); // TextField
}
```

Form Validation:

Flutter provides built-in support for form validation using the validator property of form fields. Developers can define validator functions that evaluate the input value and return an error message if the input does not meet the specified criteria. The Form widget automatically triggers validation when the form is submitted, and displays error messages for invalid fields.

```dart
bool isLogin = true;
final TextEditingController emailController = TextEditingController();
final TextEditingController passwordController = TextEditingController();

@override
void dispose() {
  emailController.dispose();
  passwordController.dispose();
  super.dispose();
}
```

Form Submission:

Form submission in Flutter involves handling user input after the form is validated. Developers typically use the onPressed callback of a submit button to trigger form submission. Within the submit callback, developers can access the current state of the form using the FormState object and retrieve the values of individual form fields for further processing, such as data submission to a server or local storage.

Feedback and Error Handling:

Providing feedback to users during form interaction is crucial for a positive user

experience. Flutter allows developers to display error messages and visual indicators to guide users when input validation fails. Error messages can be displayed inline with form fields or in a separate section of the form, depending on the design requirements.

Additionally, developers can use dialogs or snack bars to provide feedback upon successful form submission or error handling.

By leveraging these form widgets and techniques, developers can create intuitive and responsive forms in Flutter applications, enabling seamless interaction with users and efficient data collection and processing.

Conclusion:

Creating interactive forms in Flutter using form widgets is essential for building user-friendly applications that collect and process user input effectively. By utilizing form widgets such as Form and TextFormField, along with form validation and submission techniques, developers can design robust and responsive forms that enhance the overall user experience. With Flutter's flexibility and rich set of features, developers have the tools they need to create dynamic and interactive forms tailored to their application's requirements.

Aim: To apply navigation, routing and gestures in Flutter App

Theory:

Flutter provides tools to handle navigation, routing, and gestures, allowing users to move between screens and interact with the app smoothly. These features help create a user-friendly experience in mobile applications.

---

1. Navigation in Flutter

Navigation is the process of moving between different screens (or pages) in a Flutter app. Flutter uses a stack-based approach for navigation, where new screens are pushed onto the stack and removed when the user navigates back.

Types of Navigation:

- Push Navigation: Moves to a new screen and adds it to the stack.
- Pop Navigation: Removes the current screen and returns to the previous one.
- Named Routes: Uses pre-defined route names to navigate.
- Navigation with Data: Allows passing data between screens when navigating.

---

2. Routing in Flutter

Routing helps in managing different screens efficiently. Instead of manually handling each screen transition, Flutter allows defining routes in a structured way.

Types of Routing:

- Direct Routing: Navigates to a specific screen using explicit methods.
- Named Routing: Uses a predefined route name to navigate, making the app more organized.

Routing improves app maintainability, especially in apps with multiple screens.

---

3. Gestures in Flutter

Gestures enable user interaction in Flutter applications. Flutter provides built-in gesture detection capabilities for touch-based interactions.

Common Gestures:

- Tap: A single touch interaction.
- Double Tap: Two quick consecutive taps.

- Long Press: Holding a touch for a longer duration.
- Swipe: Moving a finger across the screen.
- Drag: Moving an object by pressing and holding it.

Gestures are essential for making apps interactive and responsive.

---

4. Combining Navigation and Gestures

Navigation and gestures can be combined to enhance user experience. For example:

- Tapping on a button can navigate to another screen.
- Swiping a card can delete an item or move to another page.
- Dragging an element can reposition items within the app.

Navigation, routing, and gestures are fundamental to creating an interactive Flutter application. Navigation allows movement between screens, routing helps manage screens efficiently, and gestures enable touch interactions. Mastering these concepts helps in developing dynamic and user-friendly Flutter applications.

Code:

login.dart file

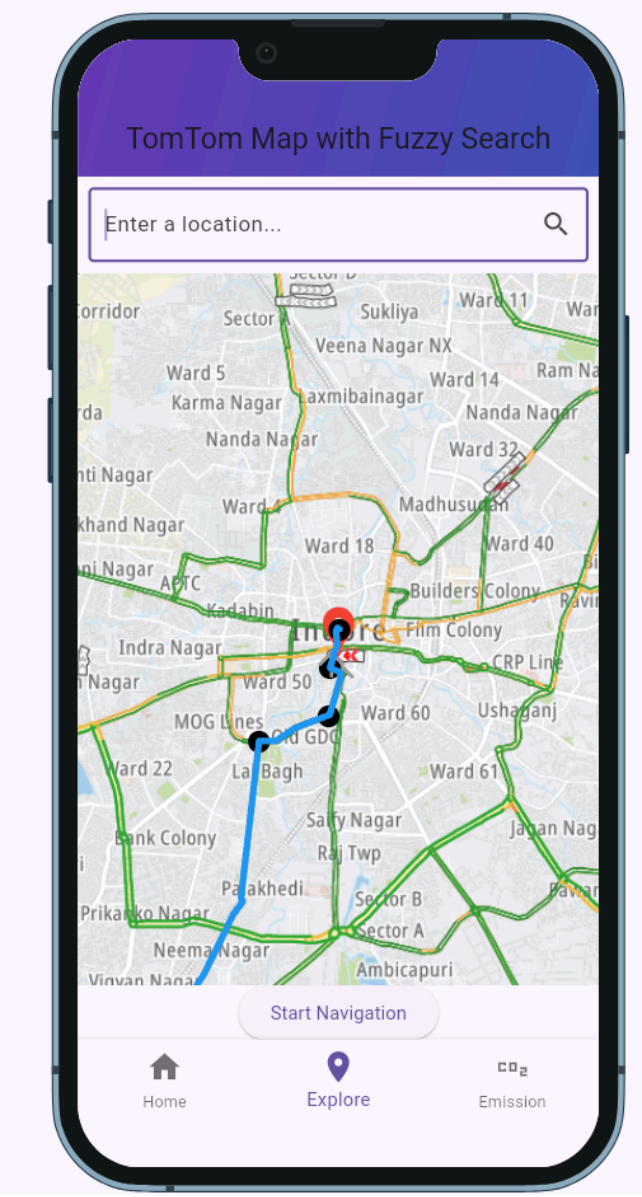
```
class _HomePageState extends
State<HomePage> {
  int currentPage = 0;
  List<Widget> pages = [
    HomeSectionPage(),
    MapPageVersion(),
    CarbonEmissionPage(),
  ];
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: IndexedStack(
        index: currentPage,
        children: pages,
      ),
      bottomNavigationBar:
BottomNavigationBar(
        type: BottomNavigationBarType.fixed,
        iconSize: 28,
        currentIndex: currentPage,
        onTap: (value) {
          setState(() {
            currentPage = value;
          });
        },
        items: [
          BottomNavigationBarItem(
            icon: Icon(Icons.home),
            label: 'Home',
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.location_on),
            label: 'Explore',
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.co2_sharp),
            label: 'Emission',
          ),
        ],
      ),
    );
  }
}
```

```
  Widget build(BuildContext context) {
    return Scaffold(
      body:
        Stack(
        children:
        [
          // Background Image
          Align(
            alignment:
            Alignment.topCenter, child:
            Container(
              height: 580, // Adjust the height as
needed
              decoration: BoxDecoration(
                image: DecorationImage(
                image:
AssetImage("assets/background.png
                  "), fit: BoxFit.scaleDown,
                ),
              ),
            ),
          )

Widget build(BuildContext context) {
  final List<Map<String, dynamic>> categories =
[
    {
      'title': 'Traffic Alerts',
      'icon': Icons.traffic,
      'color': Colors.red,
      'route': TrafficAlertScreen(),
    },
    {
      'title': 'Weather Updates',
      'icon': Icons.wb_sunny,
      'color': Colors.orange,
      'route': WeatherScreen(),
    },
    // {'title': 'Navigation','icon': Icons.map,'color':
Colors.blue},
    {
      'title': 'Fuel Stations',
      'icon': Icons.local_gas_station,
      'color': Colors.green,
      'route': FuelStationList(),
    },

    {
      'title': 'Emergency',
      'icon': Icons.local_hospital,
      'color': Colors.pink,
      'route': HospitalPage(),
    },
    {
      'title': 'ChatBot',
      'icon': Icons.local_hospital,
      'color': Colors.deepPurple,
      'route': ChatScreen(),
    },

  ];

  final List<Map<String, String>> videos = [
    {
      'title': 'Traffic Updates',
      'description': 'Stay updated with real-time
traffic alerts.',
      'videoUrl':

'https://flutter.github.io/assets-for-api-docs/assets
/videos/butterfly.mp4',
    },
    {
      'title': 'Weather Forecast',
      'description': 'Get accurate weather
predictions for your region.',
      'videoUrl':

'https://flutter.github.io/assets-for-api-docs/assets
/videos/bee.mp4',
    },
    {
      'title': 'Fuel Stations Nearby',
      'description': 'Find the nearest fuel stations
with ease.',
      'videoUrl':

'https://flutter.github.io/assets-for-api-docs/assets
/videos/butterfly.mp4',
    },
  ];

  bool isDarkMode =
Theme.of(context).brightness ==
Brightness.dark;

  return Scaffold(
    backgroundColor: const
Color.fromARGB(255, 187, 218, 234),
```

```
      appBar: AppBar(
        title: const Text(
          '🚛 Truck Map',
          style: TextStyle(
              fontSize: 22, fontWeight:
FontWeight.bold, color: Colors.white),
        ),
        flexibleSpace: Container(
          decoration: const BoxDecoration(
            gradient: LinearGradient(
              colors: [Colors.deepPurple,
Colors.indigo],
              begin: Alignment.topLeft,
              end: Alignment.bottomRight,
            ),
          ),
        ),
        elevation: 4,
        actions: [
          IconButton(
            onPressed: fetchTrafficAlerts,
            icon: const Icon(Icons.refresh, size: 28,
color: Colors.white),
          )
        ],
      ),
      body: ListView(
        padding: const EdgeInsets.all(12.0),
        children: [
          _buildLatestUpdateCard(isDarkMode),
          const SizedBox(height: 20),

          // Categories Section
          Wrap(
            spacing: 16.0,
            runSpacing: 16.0,
            children: categories
                .map(
                  (category) => GestureDetector(
                    onTap: () {
                      if (category.containsKey('route')) {
                        Navigator.push(
                          context,
                          MaterialPageRoute(
                            builder: (context) =>
category['route'],
                          ),
                        );
                      }
                    },
                    child: CategoryPage(
                      title: category['title'],
                      icon: category['icon'],
                      color: category['color'],
                    ),
                  ),
                )
                .toList(),
          ),
          // Suggested Videos
          const SizedBox(height: 20),
          Text(
            "Essential Driving Tutorials: Videos You
Must Watch for Safe Driving!",
            style: TextStyle(
              fontSize: 16,
              fontWeight: FontWeight.bold,
              color: Colors.black87,
            ),
            textAlign: TextAlign.center,
          ),
          Wrap(
            spacing: 16.0,
            runSpacing: 16.0,
            children: videos
                .map((video) => VideoCard(
                      title: video['title']!,
                      description: video['description']!,
                      videoUrl: video['videoUrl']!,
                    ))
                .toList(),
          ),

          SuggestedVideos(
            title: 'Rules and Regulations',
            description:
                'Drive safely with real-time traffic alerts
and navigation.',
            icon: Icons.local_police,
          ),
          const SizedBox(height: 20),
        ],
      ),
    );
  }

  Widget _buildLatestUpdateCard(bool
isDarkMode) {
```

```
    return Container(
      decoration: BoxDecoration(
        gradient: LinearGradient(
          colors: isDarkMode
              ? [Colors.deepPurple[700]!,
Colors.indigo[800]!]
              : [Colors.white, Colors.blueGrey[50]!],
          begin: Alignment.topLeft,
          end: Alignment.bottomRight,
        ),
        borderRadius: BorderRadius.circular(16),
        boxShadow: [
          BoxShadow(
            color: Colors.black.withOpacity(0.1),
            blurRadius: 10.0,
            spreadRadius: 2.0,
            offset: const Offset(0, 4),
          ),
        ],
      ),
      padding: const EdgeInsets.all(16.0),
      child: Column(
        crossAxisAlignment:
CrossAxisAlignment.start,
        children: [
          const Text(
            '📢 LATEST UPDATE',
            style: TextStyle(
                fontSize: 20,
                fontWeight: FontWeight.bold,
                color: Colors.black87),
          ),
          const SizedBox(height: 10),
          if (isLoading)
            _buildShimmerEffect()
          else if (latestUpdates.isEmpty)
            const Text(
              '🚦 No traffic alerts available',
              style: TextStyle(fontSize: 16, color:
Colors.redAccent),
            )
          else
            Column(
              children:
latestUpdates.take(3).map((update) {
                return Padding(
                  padding: const
EdgeInsets.symmetric(vertical: 6.0),
                  child: Row(
                    children: [
                      const Icon(Icons.warning,
                          color: Colors.orangeAccent, size:
24),
                      const SizedBox(width: 10),
                      Expanded(
                        child: Text(
                          update,
                          style: const TextStyle(
                              fontSize: 16, color:
Colors.black87),
                          overflow: TextOverflow.ellipsis,
                          maxLines: 2,
                        ),
                      ),
                    ],
                  ),
                );
              }).toList(),
            ),
        ],
      ),
    );
  }

  Widget _buildShimmerEffect() {
    return GestureDetector(
      onTap: () {},
      child: Column(
        children: List.generate(
          3,
          (index) => Padding(
            padding: const
EdgeInsets.symmetric(vertical: 6.0),
            child: Shimmer.fromColors(
              baseColor: Colors.grey[300]!,
              highlightColor: Colors.grey[100]!,
              child: Container(
                width: double.infinity,
                height: 20,
                decoration: BoxDecoration(
                  color: Colors.white,
                  borderRadius:
BorderRadius.circular(4),
                ),
              ),
            ),
          ),
        ),
      ),
```

Output :-


TomTom Map with Fuzzy Search
Enter a location...
Sukliya
Veena Nagar NX
Ward 5
Karma Nagar
Ward 14
Nanda Nagar
Ward 32
Madhusudan
Ward 18
Ward 40
APTC
Builders Colony
Kadabin
Film Colony
Indra Nagar
CRP Line
Ward 50
Ward 60
Ushaganj
MOG Lines
Ward 22
Ward 61
Sanjay Nagar
Raj Twp
Sector B
Sector A
Ambicapuri
Start Navigation
Home
Explore
Emission

Conclusion :-

Flutter simplifies app interaction through effective navigation, routing, and gesture handling. By using navigation and routing, developers can manage screen transitions and data flow between pages efficiently. Gesture detection enhances user experience by enabling interactive touch responses like taps and swipes. Together, these features help create smooth, intuitive, and user-friendly mobile applications.

EXPERIMENT 6

Aim: To connect flutter UI with firebase database

Theory:
Connecting a Flutter application to a Firebase database allows for seamless real-time data storage and retrieval, making apps dynamic and responsive. Firebase offers two main database services: Cloud Firestore (a flexible, scalable NoSQL cloud database) and Realtime Database (a tree-structured JSON database for real-time syncing).

In Flutter, Firebase integration is achieved using the firebase_core and cloud_firestore (or firebase_database) packages. After initializing Firebase in the app, data can be added, read, updated, and deleted directly through Firebase methods. Flutter widgets can be connected to database streams, ensuring that any changes in the database are instantly reflected in the UI.

This integration enhances app functionality by enabling persistent data storage, user-specific content, and real-time updates, making it essential for building modern mobile applications.

Code:

signup.dart
The signup.dart file connects the Flutter UI with Firebase Authentication and Cloud Firestore. When a user signs up, their email and password are authenticated using FirebaseAuth. After successful registration, the user's additional details like name and email are stored in Firestore under the users collection using their unique UID. This demonstrates how user registration data is securely stored in the Firebase database through Flutter.

```
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flutter/material.dart';
import 'package:google_sign_in/google_sign_in.dart';

class Login extends StatefulWidget {
  const Login({super.key});

  @override
  State<Login> createState() => _LoginState();
}

class _LoginState extends State<Login> {
  bool isLogin = true;
```

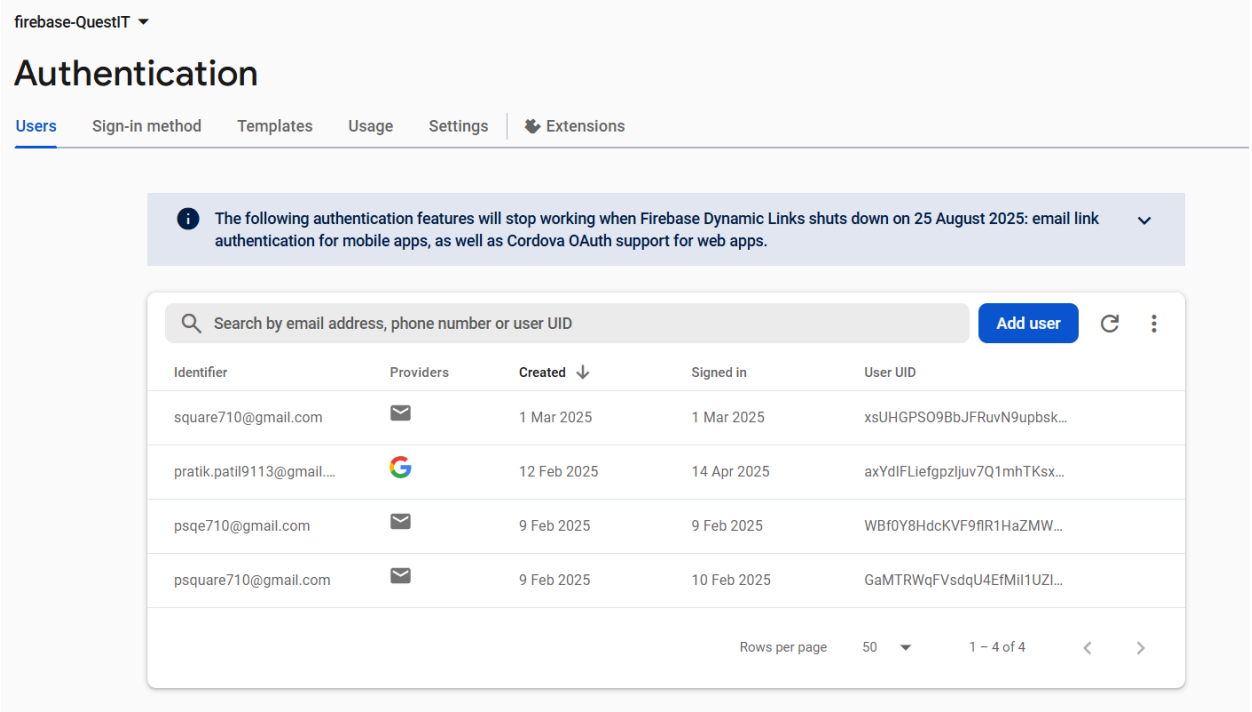
```
final TextEditingController emailController = TextEditingController();
final TextEditingController passwordController = TextEditingController();

@override
void dispose() {
  emailController.dispose();
  passwordController.dispose();
  super.dispose();
}

Future<UserCredential?> signInWithGoogle() async {
  try {
    final GoogleSignInAccount? googleUser = await GoogleSignIn().signIn();

    if (googleUser == null) {
      print("User canceled the Google sign-in.");
      return null;
    }
    final GoogleSignInAuthentication googleAuth =
        await googleUser.authentication;
    final AuthCredential credential = GoogleAuthProvider.credential(
      accessToken: googleAuth.accessToken,
      idToken: googleAuth.idToken,
    );
    return await FirebaseAuth.instance.signInWithCredential(credential);
  } catch (e) {
    print("Google sign-in failed: $e");
    return null;
  }
}

Future<void> createUserWithEmailAndPassword() async {
  try {
    if (isLogin) {
      await FirebaseAuth.instance.signInWithEmailAndPassword(
        email: emailController.text.trim(),
        password: passwordController.text.trim(),
      );
    } else {
```

```
      await FirebaseAuth.instance.createUserWithEmailAndPassword(
        email: emailController.text.trim(),
        password: passwordController.text.trim(),
      );
    }
  } on FirebaseAuthException catch (e) {
    print(e.message);
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Colors.blueGrey[900],
    body: Center(
      child: Padding(
        padding: const EdgeInsets.all(20.0),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Icon(
              Icons.local_shipping,
              size: 80,
              color: Colors.orangeAccent,
            ),
            const SizedBox(height: 20),
            Text(
              isLogin ? 'Truck Map Login' : 'Truck Map Sign Up',
              style: const TextStyle(
                fontSize: 28,
                fontWeight: FontWeight.bold,
                color: Colors.white,
              ),
            ),
            const SizedBox(height: 20),
            _buildTextField(emailController, 'Enter your Email', Icons.email),
            const SizedBox(height: 15),
            _buildTextField(
                passwordController, 'Enter your Password', Icons.lock,
```

```
            isPassword: true),
        const SizedBox(height: 20),
        _buildButton(isLogin ? 'Login' : 'Sign Up',
            createUserWithEmailAndPassword),
        const SizedBox(height: 10),
        Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(
              isLogin ? 'New here?' : 'Already registered?',
              style: const TextStyle(color: Colors.white),
            ),
            TextButton(
              onPressed: () {
                setState(() {
                  isLogin = !isLogin;
                });
              },
              child: Text(
                isLogin ? 'Sign Up' : 'Login',
                style: const TextStyle(
                    color: Colors.orangeAccent,
                    fontWeight: FontWeight.bold),
              ),
            ),
          ],
        ),
        const SizedBox(height: 10),
        _buildButton('Sign in with Google', signInWithGoogle,
            isGoogle: true),
      ],
    ),
  ),
 ),
);
}

Widget _buildTextField(
  TextEditingController controller, String hint, IconData icon,
```

```
    {bool isPassword = false}) {
  return TextField(
    controller: controller,
    obscureText: isPassword,
    decoration: InputDecoration(
      labelText: hint,
      prefixIcon: Icon(icon, color: Colors.orangeAccent),
      filled: true,
      fillColor: Colors.blueGrey[700],
      border: OutlineInputBorder(
        borderRadius: BorderRadius.circular(10),
        borderSide: BorderSide.none,
      ),
      labelStyle: const TextStyle(color: Colors.white70),
    ),
    style: const TextStyle(color: Colors.white),
  );
}

Widget _buildButton(String text, Function() onPressed,
    {bool isGoogle = false}) {
  return SizedBox(
    width: double.infinity,
    height: 50,
    child: ElevatedButton(
      onPressed: onPressed,
      style: ElevatedButton.styleFrom(
        backgroundColor: isGoogle ? Colors.white : Colors.orangeAccent,
        foregroundColor: isGoogle ? Colors.black : Colors.white,
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(10),
        ),
      ),
      child: Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          if (isGoogle)
            Icon(
              Icons.g_mobiledata,
```

```
    color: Colors.blue,
  ),
if (isGoogle) const SizedBox(width: 10),
Text(
  text,
  style: const TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
```



Conclusion:

This experiment successfully demonstrates how to integrate Firebase services with a Flutter application. By implementing user signup, login, and data submission features, we explored the use of Firebase Authentication for secure user management and Cloud Firestore for real-time database storage. The flow between screens and backend interaction highlights the power and simplicity of using Firebase in Flutter apps, making it an ideal choice for building scalable and responsive mobile applications.

Experiment 7	
Name	Pratik Manish Patil
Roll No	40
DOP	
DOS	
Sign	
Grade	

Aim: To write meta data of your Dynamic PWA in a Web app manifest file to enable “add to homescreen feature”.

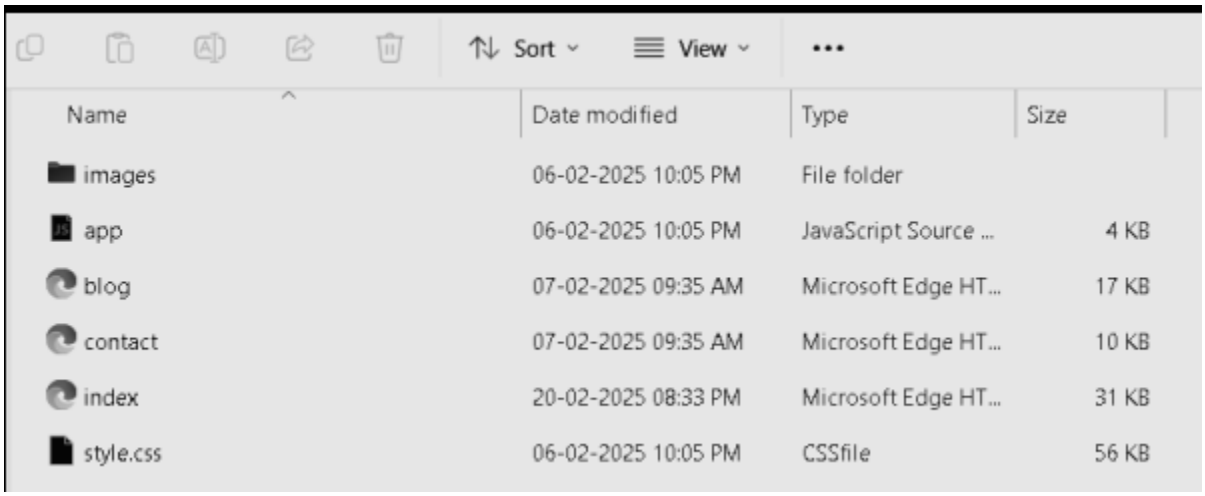
Theory:

Progressive Web Apps (PWAs) are web applications enhanced with modern web capabilities to deliver an app-like experience. One essential part of a PWA is the Web App Manifest — a JSON file that contains metadata about the app. This metadata includes:

- App name and short name
- Start URL and scope
- Icons for different screen sizes
- Theme and background colors
- Display mode (e.g., standalone, fullscreen)

By linking this manifest.json file in the HTML, the app becomes installable on user devices and can appear on the home screen like a native app. This enhances user engagement, accessibility, and branding for the Ecommerce platform.
In this experiment, metadata was defined in a manifest file and connected to the main HTML file, enabling the “Add to Home Screen” functionality for the Ecommerce PWA.

Output:




```html
<html>
<head>
  <base href="/">

  <meta charset="UTF-8">
  <meta content="IE=Edge" http-equiv="X-UA-Compatible">
  <meta name="description" content="A new Flutter project.">
  <meta name="google-signin-client_id" content="175543833376-iep7akagt2r6dm31jtjnq4c

  <!-- iOS meta tags & icons -->
  <meta name="mobile-web-app-capable" content="yes">
  <meta name="apple-mobile-web-app-status-bar-style" content="black">
  <meta name="apple-mobile-web-app-title" content="questit">
  <link rel="apple-touch-icon" href="icons/Icon-192.png">

  <!-- Favicon -->
  <link rel="icon" type="image/png" href="favicon.png"/>

  <title>questit</title>
  <link rel="manifest" href="manifest.json">
</head>
<body>
  <script src="flutter_bootstrap.js" async></script>
</body>
</html>
```

Conclusion:

In this experiment, we successfully create and integrated a Web App Manifest file for our Progressive Web App. By defining essential metadata such as the app's name, icons, theme color, and display mode, we enabled the "Add to Home Screen" functionality.

Experiment 8	
Name	Pratik Manish Patil
Roll No	40
DOP	
DOS	
Sign	
Grade	

Aim: To code and register a service worker, and complete the install and activation process for a new service worker for the PWA.

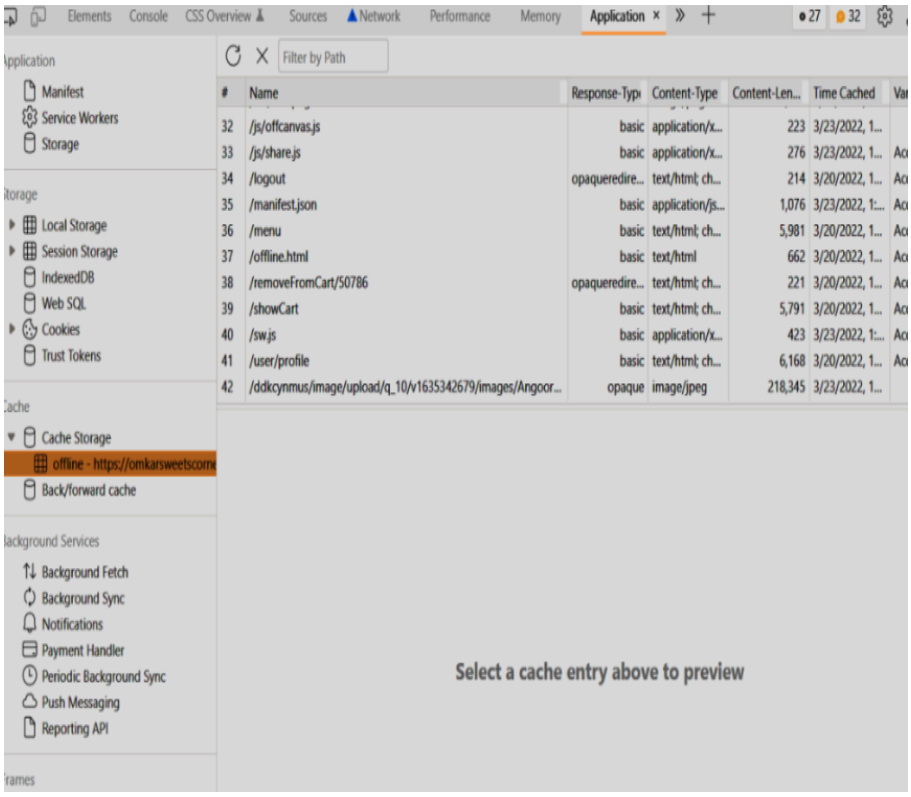
Theory:

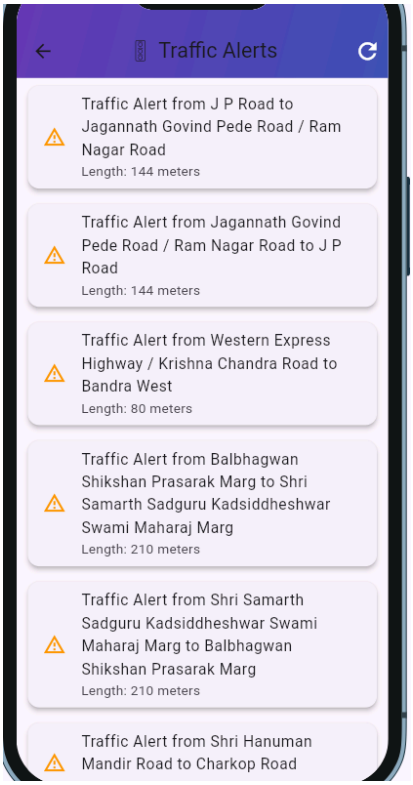
A Service Worker is a type of web worker script that runs in the background, separate from the main browser thread. It plays a key role in enabling Progressive Web App (PWA) features like offline support, background sync, and push notifications.

When a service worker is registered, it goes through three phases:
1. Install – Triggered once when the service worker is installed for the first time. Used to cache necessary assets.
2. Activate – Triggered when the service worker takes control of the page. Typically used for clearing out old caches.
3. Fetch – Intercepts network requests and serves cached responses if available, enabling offline access.

These features help improve performance, reliability, and user experience, especially in unstable or no network conditions.

Output:





Conclusion:

In this experiment, we implemented a Service Worker in our PWA to enhance performance and offline functionality. We successfully coded, registered, and completed the install and activate phases. This setup ensures that our app can cache key resources, load faster, and work in low or no network conditions, ultimately offering a more reliable and app-like experience to users..

Experiment 9	
Name	Pratik Manish Patil
Roll No	40
DOP	
DOS	
Sign	
Grade	

Aim: To implement Service worker events like fetch, sync and push for PWA

Theory:

A Service Worker is a JavaScript file that runs in the background of a Progressive Web App (PWA).

It acts as a proxy between the web app and the network, enabling features like:
- Caching content for offline use (fetch event)
- Syncing data in the background (sync event)
- Receiving and displaying push notifications (push event)

These service worker events significantly improve user experience by ensuring fast loading, real-time updates, and engagement, even in low or no internet connectivity.

Output:

1. fetch event :-

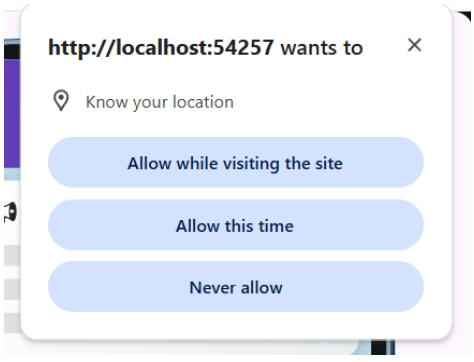
```
// Save response to cache
SharedPreferences prefs = await SharedPreferences.getInstance();
prefs.setString("hospital", json.encode(_fuelStations));
} else {
```

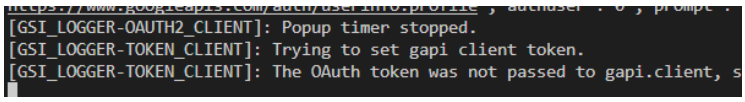
2. sync event :-

```
self.addEventListener('sync', (event) => {
  if (event.tag === 'sync-data') {
    event.waitUntil(syncDataWithServer());
  }
});
```

3. Push Event : -

```
self.addEventListener('push', (event) => {
  const data = event.data.json();
  self.registration.showNotification(data.title, {
    body: data.body,
    icon: 'icon.png'
  });
});
```





Conclusion:

In this experiment, we successfully implemented the core Service Worker events (fetch, sync, and push) in the railway PWA. This enhanced the app's ability to:

- Work offline using cache (fetch)
- Automatically sync data in the background (sync)
- Engage users with notifications (push)

These features are crucial for improving reliability, performance, and user engagement in modern web applications.

Experiment 10	
Name	Pratik Manish Patil
Roll No	40
DOP	
DOS	
Sign	
Grade	

Aim: To study and implement deployment of Ecommerce PWA to GitHub Pages.

Theory:

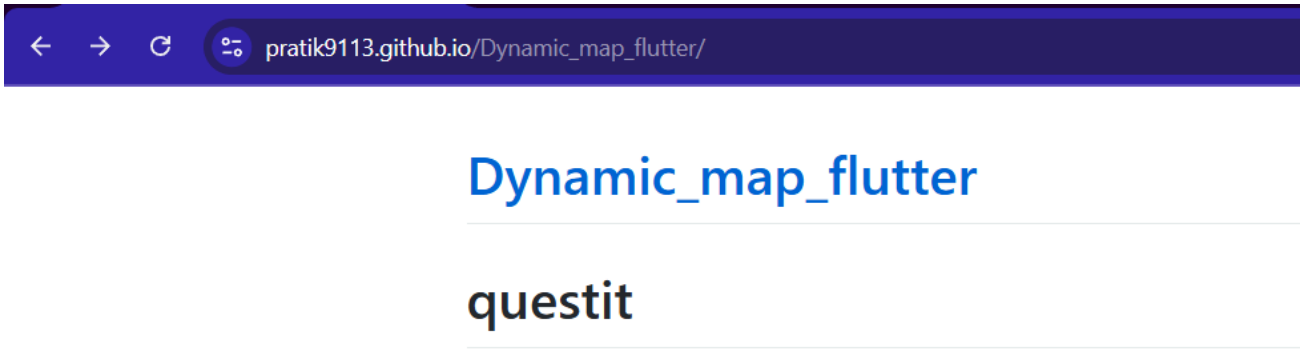
GitHub Pages is a free hosting platform by GitHub for static websites. It can serve HTML, CSS, JavaScript, and service worker files, making it a suitable option for deploying Progressive Web Apps (PWAs).

Deploying your PWA like railway app to GitHub Pages allows:

- Easy and quick sharing via a live URL.
- Hosting without any additional setup or cost.
- Support for offline features and "Add to Home Screen" via service workers and manifest file.
- Continuous updates through Git version control.

This helps you showcase and distribute your web app publicly with minimal effort

Output:



Conclusion:
By deploying Dynamic Map as a Progressive Web App on GitHub Pages, we achieved a seamless and cost-effective way to make the application publicly accessible. This deployment not only ensures easy updates and version control through GitHub, but also enables core PWA features like offline access, faster loading, and the “Add to Home Screen” experience. Hosting on GitHub Pages simplifies the process of sharing and testing while providing a reliable platform to showcase the app's capabilities in real-world scenarios.

Experiment 11	
Name	Pratik Manish Patil
Roll No	40
DOP	
DOS	
Sign	
Grade	

Aim: To use google Lighthouse PWA Analysis Tool to test the PWA functioning.

Theory:

Google Lighthouse is an open-source, automated tool developed by Google to improve the
quality of web pages. It provides audits for performance, accessibility, SEO, best practices, and
Progressive Web App (PWA) standards.

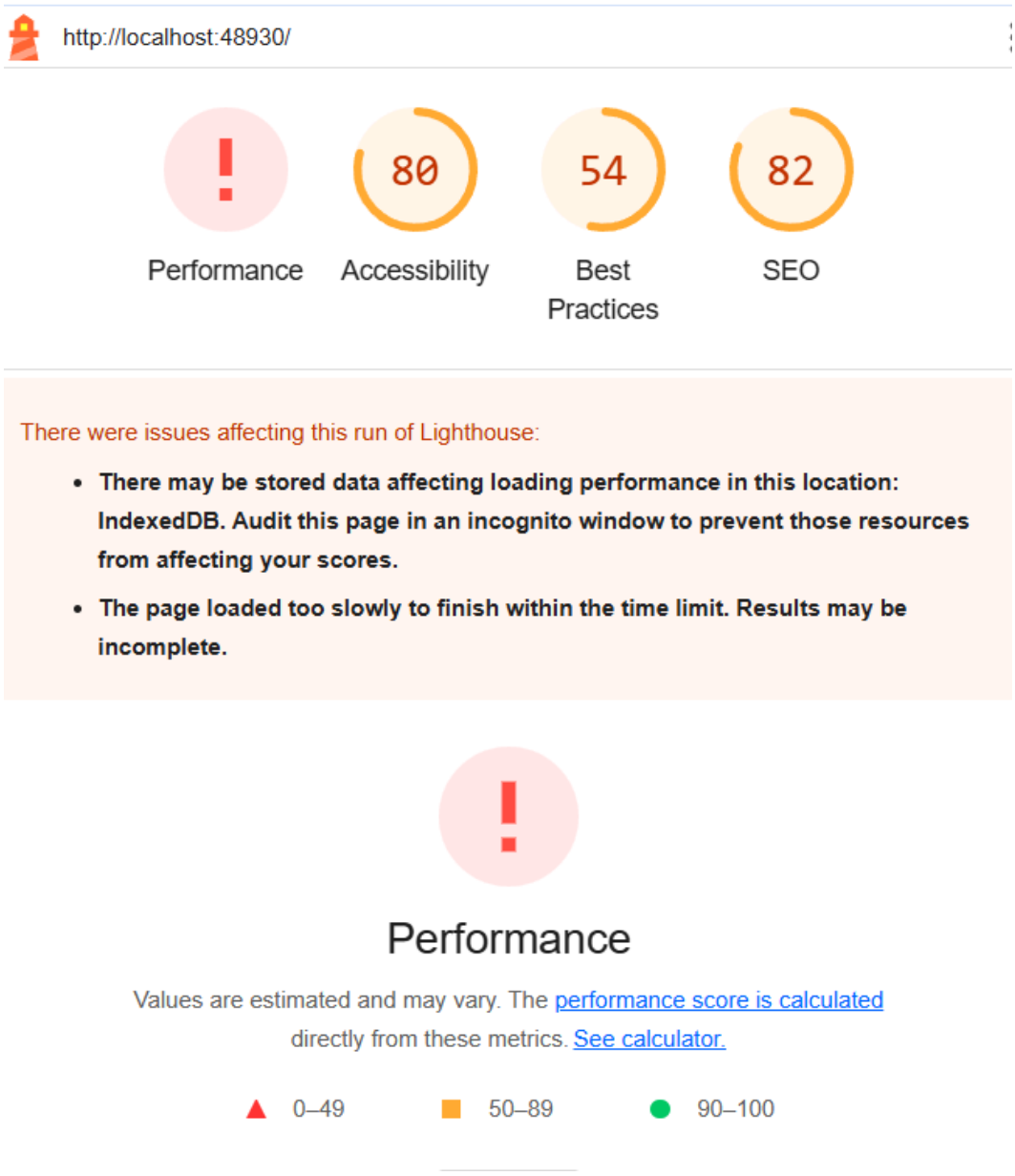
When testing a PWA, Lighthouse checks for critical requirements like

- Valid web app manifest
- Presence and correct behavior of service workers
- HTTPS usage
- Responsive design
- Offline functionality
- Add to Home Screen capability

Lighthouse gives a score out of 100 based on how well the app performs as a PWA and offers
suggestions for improvement. It is accessible directly in Chrome DevTools or as a browser
extension.

This tool is crucial for ensuring the app meets modern web standards and provides a high-
quality user experience on mobile and desktop

Output:



Conclusion:

Using Google Lighthouse, we successfully analyzed the PWA capabilities of Dynamic Truck app. The tool helped us verify important aspects like offline access, manifest configuration, and service worker functionality. It also provided valuable suggestions to optimize user experience and app performance, making it a vital step in PWA development and deployment