Name: Atif Ansari Roll no: 04 Class: D15B

MPL Experiment - 1

A1

* Aim: To install and configure the Flutter development environment for building cross-platform mobile applications.

* Theory: Flutter is an open-source UI toolkit by Google for creating natively compiled applications for mobile, web, and desktop from a single codebase. It uses the Dart programming language and provides a rich set of pre-designed widgets. The setup process involves:

1. Installing Flutter SDK: Downloading and Setting up Flutter from the official website.

2. Setting up an IDE: Configuring Visual Studio Code or Android Studio (in our experiment) with Flutter and Dart pluggins.

3. Setting up an Emulator: Installing Android Emulator or using a real device for testing.

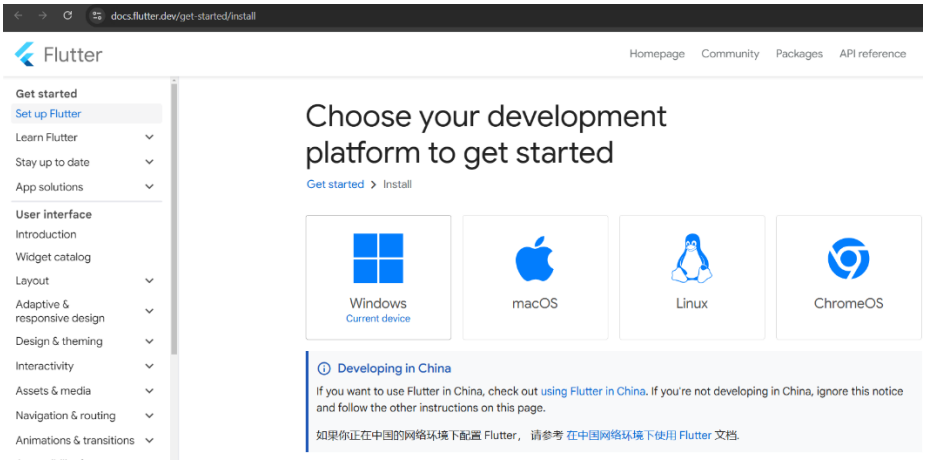
4. Configuring Environment Variables: Adding Flutter to the system PATH.

5. Verifying Installation: Running flutter doctor to check dependencies.

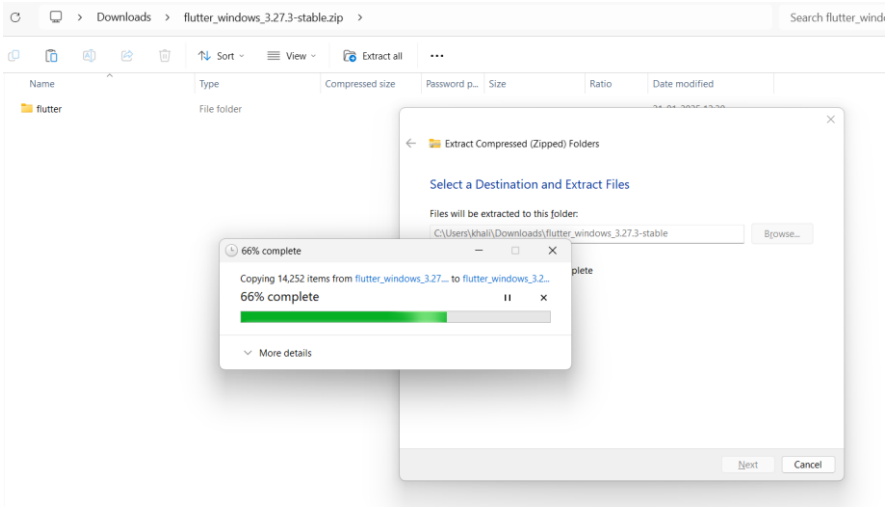
* Conclusion: In this experiment, we successfully installed and configured the Flutter development

environment. The Flutter SDK was downloaded and added to the system PATH, allowing global access. A suitable IDE, such as Android studio, was set up with the required Flutter and Dart plugins. Additionally an emulator or a real device was configured for testing Flutter applications. Finally, flutter doctor was executed to verify that all dependencies were properly installed, ensuring a fully functional Flutter development setup. With this environment ready, we can ~~now~~ now build, test, and deploy cross-platform applications efficiently.

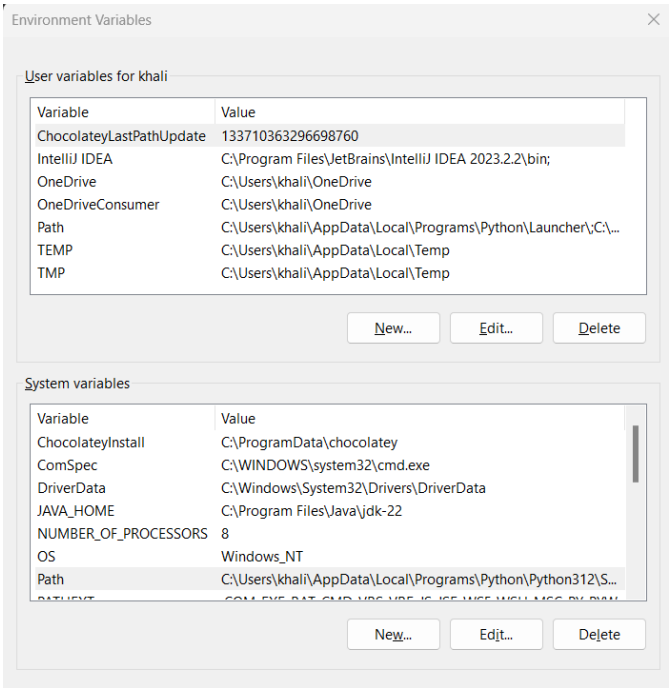
Step 1: Download the installation bundle of the Flutter Software Development Kit for windows. To download Flutter SDK, Go to its official website https://docs.flutter.dev/get-started/install , you will get the following screen.

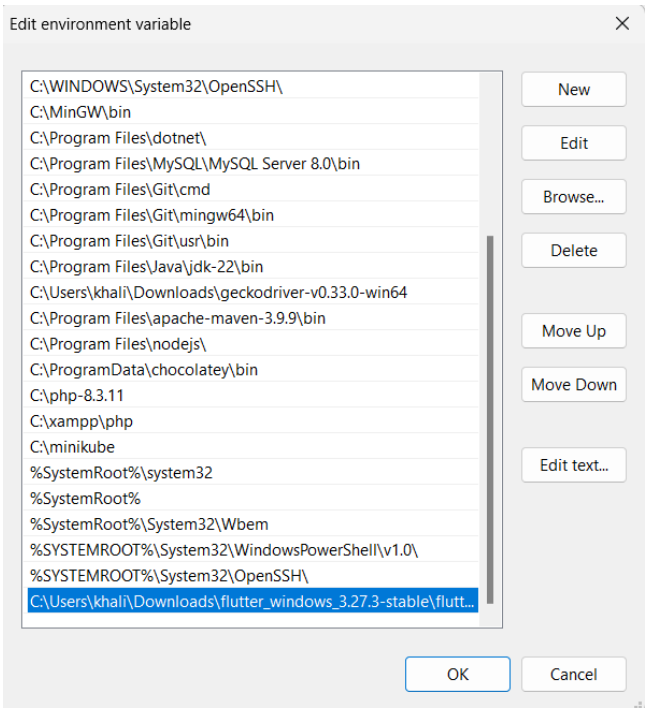


Step 2: When your download is complete, extract the zip file and place it in the desired installation folder or location.



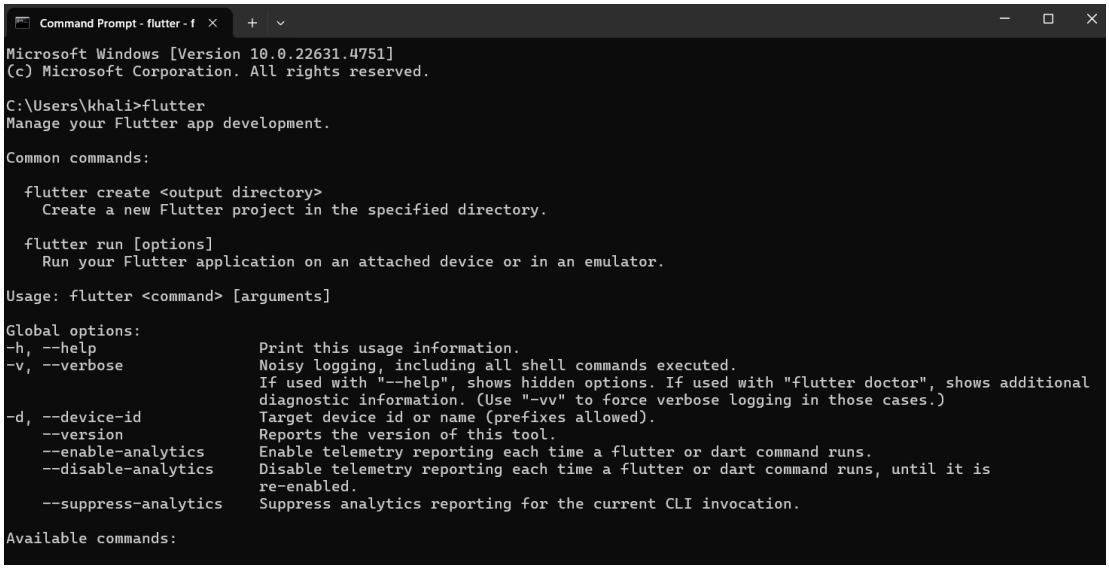
Step 3: To run the Flutter command in regular windows console, you need to update the system path to include the flutter bin directory.



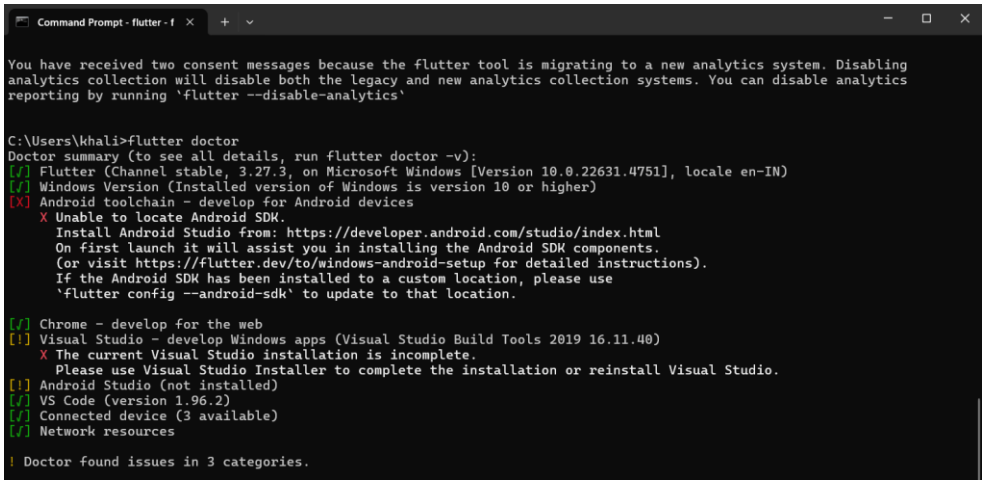


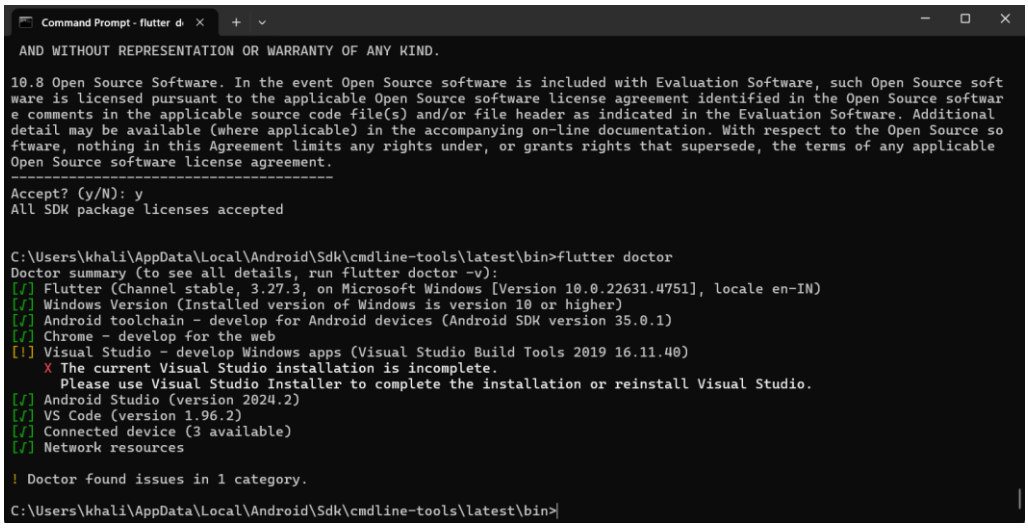
Step 4: Now, run the $ flutter command in command prompt.

Now, 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.

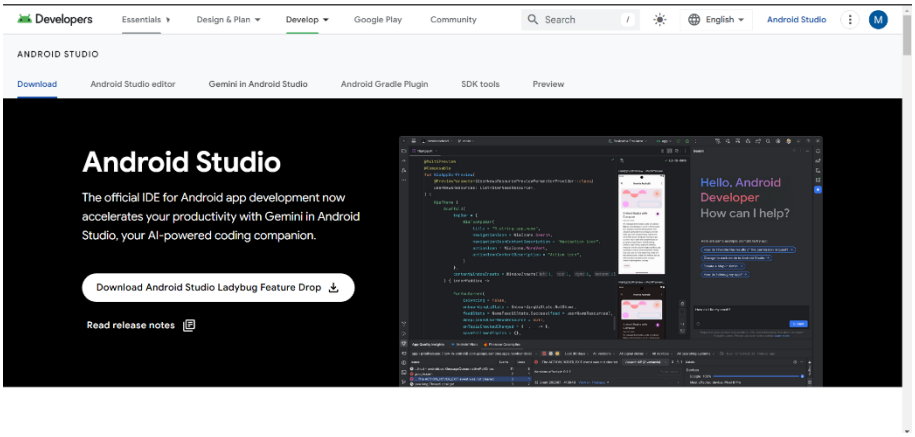


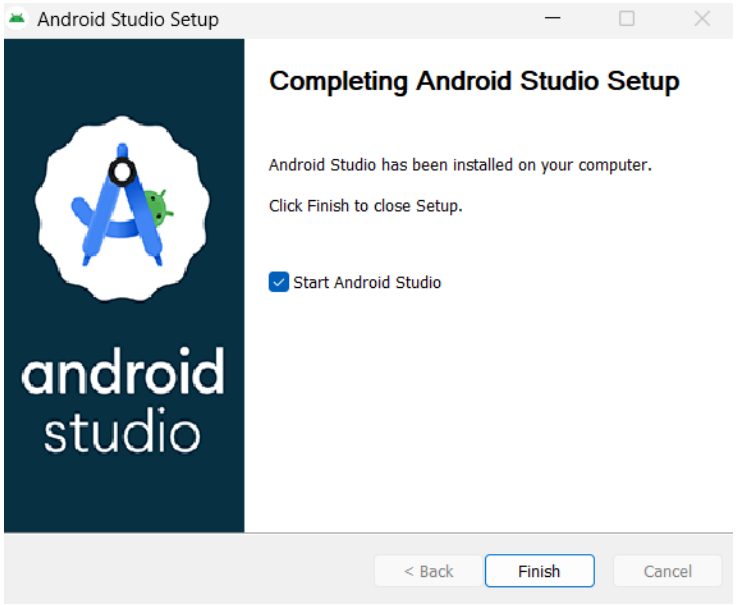
Step 5: Run the $ flutter doctor command and Run flutter doctor --android-licenses command.



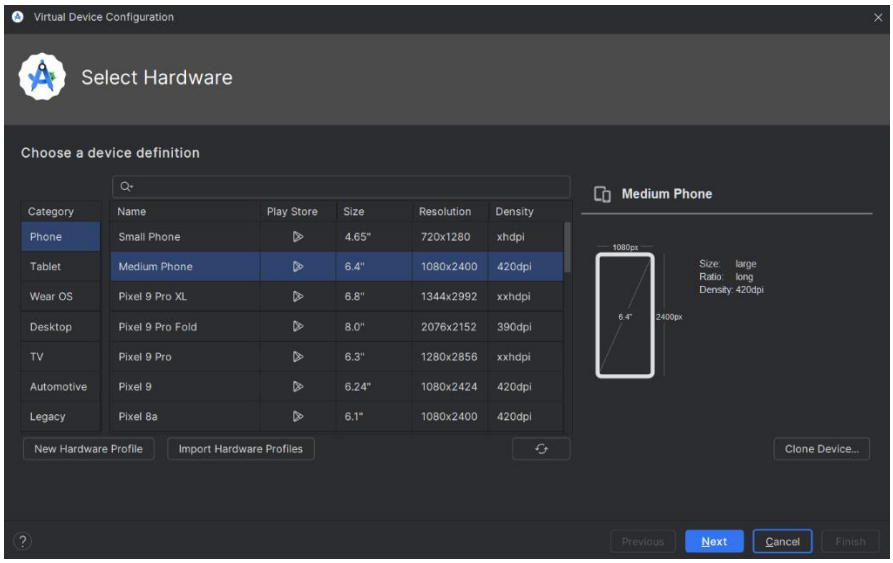


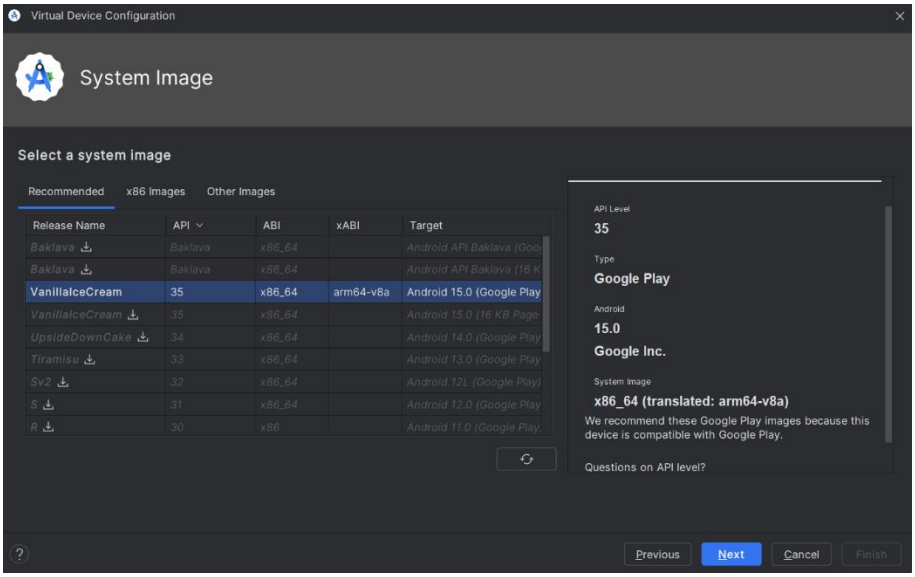
Step 6: Download the latest Android Studio executable or zip file from the official site.



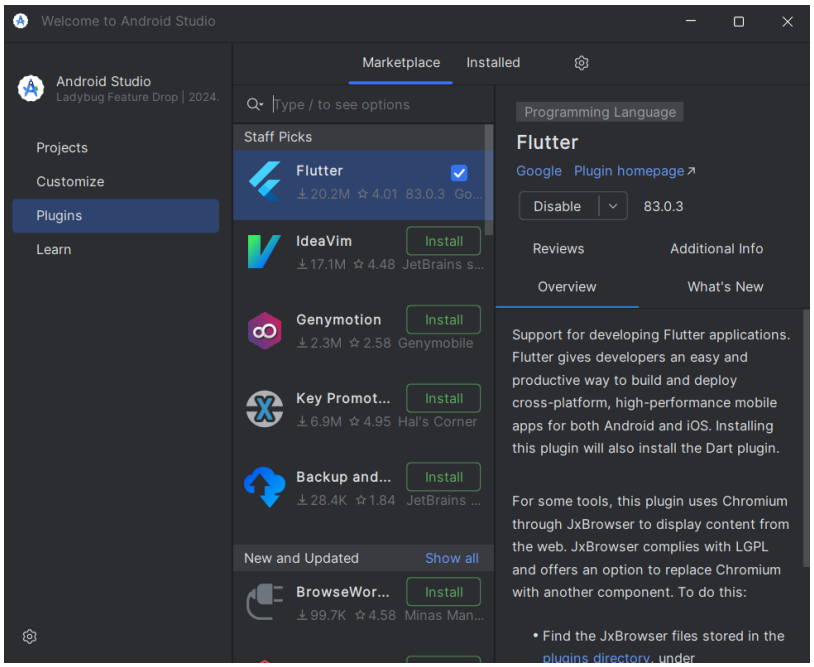


Step 7: To set an Android emulator, go to Android Studio > Tools > Android > AVD Manager and select Create Virtual Device. Or, go to Help->Find Action->Type Emulator in the search box, verify the all AVD configuration. If it is correct, click on Finish.

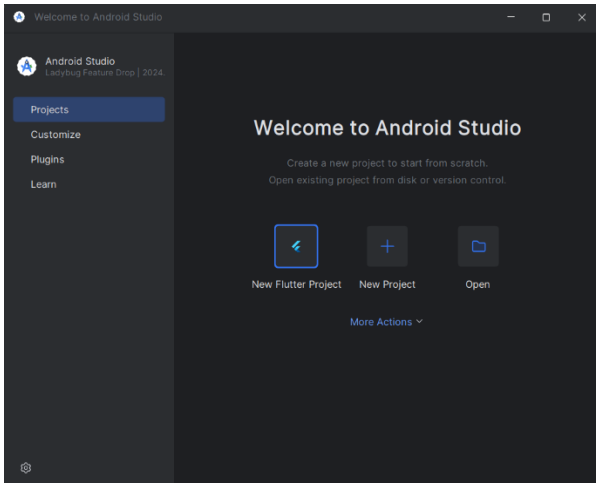




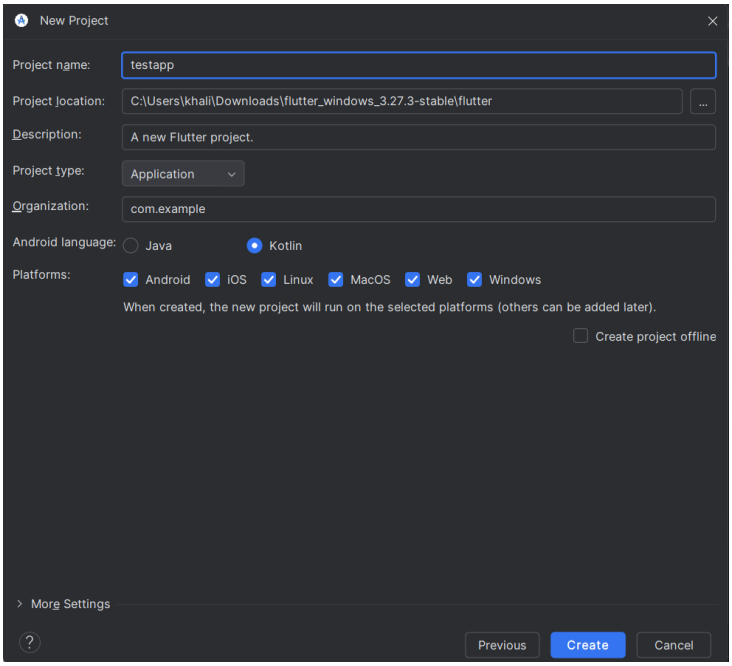
Step 8: Now, install Flutter and Dart plugin for building Flutter application in Android Studio. These plugins provide a template to create a Flutter application, give an option to run and debug Flutter application in the Android Studio itself.



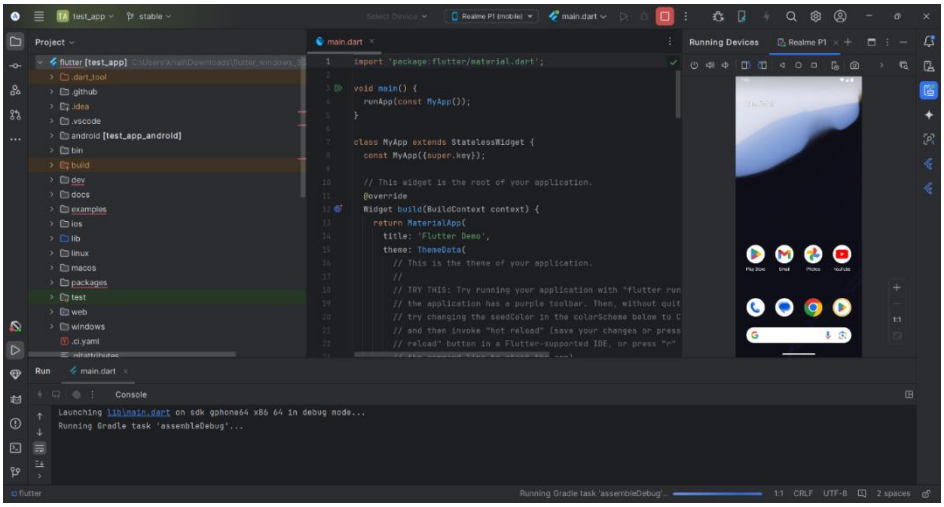
Step 9: Now we can a new create Flutter Project.



Step 10: Set name, location, description and other project details as per your choice.



Step 11: Now start the emulator and we are ready to execute the code and it will be displayed on the emulator.



Name: Atif Ansari Roll no: 04 Class: D15B

MPL Experiment - 2

* **Aim:** To Design Flutter UI by including common widgets.

* **Theory:** Flutter provides a variety of pre-build UI widgets that help in creating rich and interactive user interfaces for mobile applications. These widgets are categorized into Structural Widgets, Interactive Widgets, Layout Widgets, and Styling Widgets.

Types of Widgets used in Flutter:

1. **Structural Widgets:**
   - **Scaffold:** Provides a basic layout structure with an app bar, body, and floating buttons.
   - **AppBar:** Displays a toolbar with a title and actions.
   - **SafeArea:** Ensures content does not overlap with system UI elements.
   - **SingleChildScrollView:** Allows scrolling when content exceeds screen size.

2. **Interactive Widgets:**
   - **ElevatedButton:** A material-styled button that elevates when pressed.
   - **IconButton:** A clickable icon.

3. **Layout Widgets:**
   - **Column:** Arranges child widgets in a vertical direction.

- Padding: Adds spacing around widgets.
- Align: Positions a widget within its parent.

4. Styling Widgets:
- Text: Displays styled text.
- Icon: Displays material icons.
- Container: Used for decoration and styling.

* Conclusion: This experiment focused on building a Flutter UI using common widgets. By using Scaffold, AppBar, Text, Column, ElevatedButton, Icon, and Padding, we successfully created a structured Settings UI with different sections. The Scaffold provided the screen layout, AppBar created a navigation bar, and Column was used to align elements vertically. Each section header was styled using Text, and interactive buttons allowed user actions.

We also made use of SafeArea and ~~Single~~ SingleChildScrollView to ensure the UI was responsive and ~~adaptable~~ adaptable across devices. The use of ElevatedButton.icon() showcased how we could combine icons with text in buttons to improve user experience.

Overall, this experiment demonstrated the power of flutter's widget-based approach to UI development, allowing for flexibility, reusability, and responsive designs.

Code:

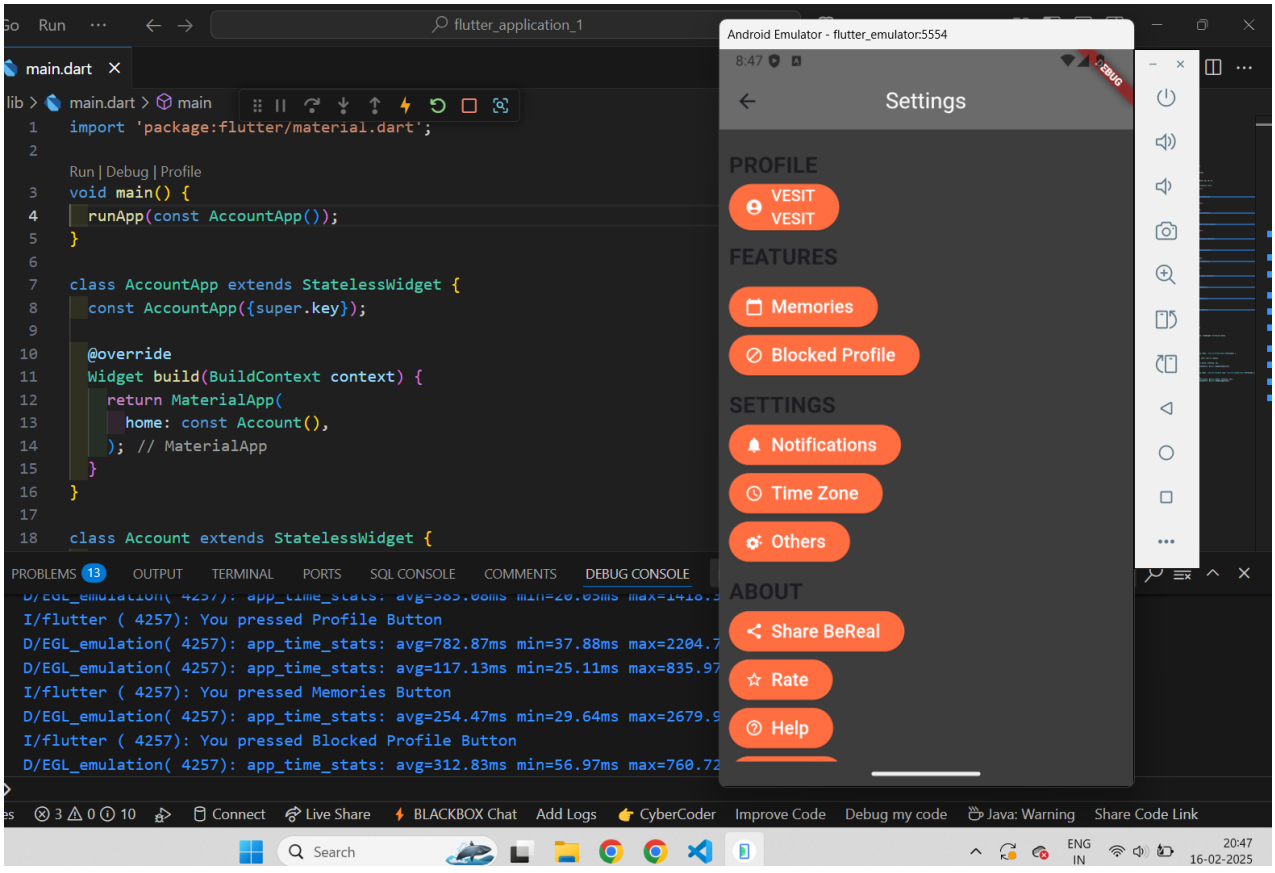
```
import 'package:flutter/material.dart';
void main() {
 runApp(const AccountApp());
}
class AccountApp extends StatelessWidget {
 const AccountApp({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   home: const Account(),
  );
 }
}
class Account extends StatelessWidget {
 const Account({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
  return Scaffold(
   backgroundColor: Colors.white24,
   appBar: AppBar(
    centerTitle: true,
    backgroundColor: Colors.white24,
    leading: IconButton(
     onPressed: () {
      Navigator.pop(context);
     },
     icon: const Icon(Icons.arrow_back),
    ),
    title: const Text(
     "Settings",
     style: TextStyle(color: Colors.white),
    ),
   ),
   body: SafeArea(
    child: Padding(
     padding: const EdgeInsets.fromLTRB(10, 20, 10, 0),
     child: SingleChildScrollView(
      child: Column(
       crossAxisAlignment: CrossAxisAlignment.start,
       children: [
        _buildSectionHeader("PROFILE"),
        const SizedBox(height: 2),
        _buildProfileButton(
         label: "VESIT\nVESIT",
         onPressed: () {
          print("You pressed Profile Button");
         },
```

```
),
const SizedBox(height: 10),
_buildSectionHeader("FEATURES"),
const SizedBox(height: 10),
_buildFeatureButton(
  label: "Memories",
  icon: Icons.calendar_today,
  onPressed: () {
    print("You pressed Memories Button");
  },
),
_buildFeatureButton(
  label: "Blocked Profile",
  icon: Icons.block,
  onPressed: () {
    print("You pressed Blocked Profile Button");
  },
),
const SizedBox(height: 10),
_buildSectionHeader("SETTINGS"),
_buildFeatureButton(
  label: "Notifications",
  icon: Icons.notifications,
  onPressed: () {
    print("You pressed Notifications Button");
  },
),
_buildFeatureButton(
  label: "Time Zone",
  icon: Icons.access_time,
  onPressed: () {
    print("You pressed Time Zone Button");
  },
),
_buildFeatureButton(
  label: "Others",
  icon: Icons.settings_suggest,
  onPressed: () {
    print("You pressed Others Button");
  },
),
const SizedBox(height: 10),
_buildSectionHeader("ABOUT"),
_buildFeatureButton(
  label: "Share BeReal",
  icon: Icons.share,
  onPressed: () {
    print("You pressed Share BeReal Button");
  },
```

```
            ),
            _buildFeatureButton(
              label: "Rate",
              icon: Icons.star_outline,
              onPressed: () {
                print("You pressed Rate Button");
              },
            ),
            _buildFeatureButton(
              label: "Help",
              icon: Icons.help_outline,
              onPressed: () {
                print("You pressed Help Button");
              },
            ),
            _buildFeatureButton(
              label: "About",
              icon: Icons.info,
              onPressed: () {
                print("You pressed About Button");
              },
            ),
          ],
        ),
      ),
    ),
  );
}
Widget _buildSectionHeader(String title) {
  return Row(
    children: [
      Text(
        title,
        style: const TextStyle(fontSize: 22, fontWeight: FontWeight.bold),
      ),
      const VerticalDivider(
        color: Colors.transparent,
        thickness: 1,
        width: 10,
      ),
    ],
  );
}
Widget _buildProfileButton({required String label, required VoidCallback onPressed}) {
  return ElevatedButton.icon(
    onPressed: onPressed,
    icon: const Icon(Icons.account_circle, color: Colors.white),
    label: Text(
```

```
      label,
      style: const TextStyle(color: Colors.white, fontSize: 16),
    ),
    style: ElevatedButton.styleFrom(backgroundColor: Colors.deepOrangeAccent),
  );
 }
 Widget _buildFeatureButton({required String label, required IconData icon, required VoidCallback onPressed})
{
  return ElevatedButton.icon(
    onPressed: onPressed,
    icon: Icon(icon, color: Colors.white),
    label: Text(label, style: const TextStyle(color: Colors.white, fontSize: 18)),
    style: ElevatedButton.styleFrom(backgroundColor: Colors.deepOrangeAccent),
  );
 }
}
```

Output:



Name: Atif Ansari Roll no: 04 Class: D15B

MPL Experiment - 3

* Aim: To explore and implement different Flutter widgets, including Text, Button, and Image widgets, to understand their properties and behaviors in a Flutter application.

* Theory: Flutter widgets are the fundamental building blocks of the UI. They are categorized into:

1. Visible Widgets: ~~Displays text with customizable properties like alignment~~ alignment, ~~font style, and color~~
   - Text widget: Displays text with customizable properties like alignment, font style, and color.
   - Button widget: Performs an action when clicked. Common types include FlatButton, ElevatedButton, and TextButton.
   - Image widget: Displays images from assets, network, memory, or files.

These widgets provide user interface elements such as text, buttons, and images that are visible.

2. Invisible widgets: These include layout and control widgets like Container, Column, and Row, which structure the UI but are not directly visible.

* Conclusion: In this experiment, we successfully explored and implemented key Flutter widgets:

Text, Button, and Image. We understood how to customize the Text widget using alignment and style properties. We implemented different types of buttons to handle user interactions. Additionally, we displayed an image from the assets folder using the Image.asset widget.

We also created a custom-styled button using a Container with specific padding, border styles, and background color. This demonstrated how invisible widgets can be used to control layout and styling.

Through this experiment, we gained practical experience in designing UI elements in Flutter, laying a strong foundation for building interactive applications.

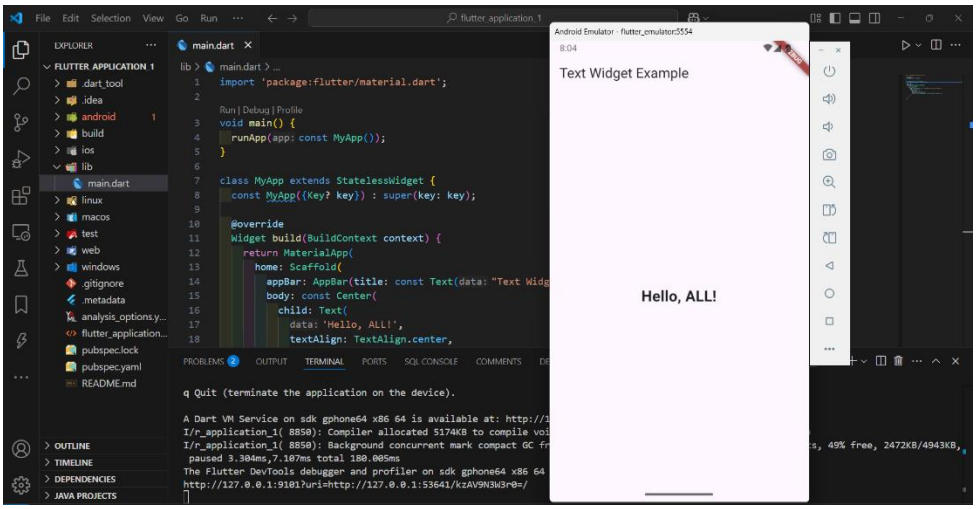
---

1. Text Widget Example

Code:

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

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text("Text Widget Example")),
        body: const Center(
          child: Text(
            'Hello, ALL!',
            textAlign: TextAlign.center,
            style: TextStyle(fontWeight: FontWeight.bold, fontSize: 24),
          ),
        ),
      ),
    );
  }
}
```

Output:

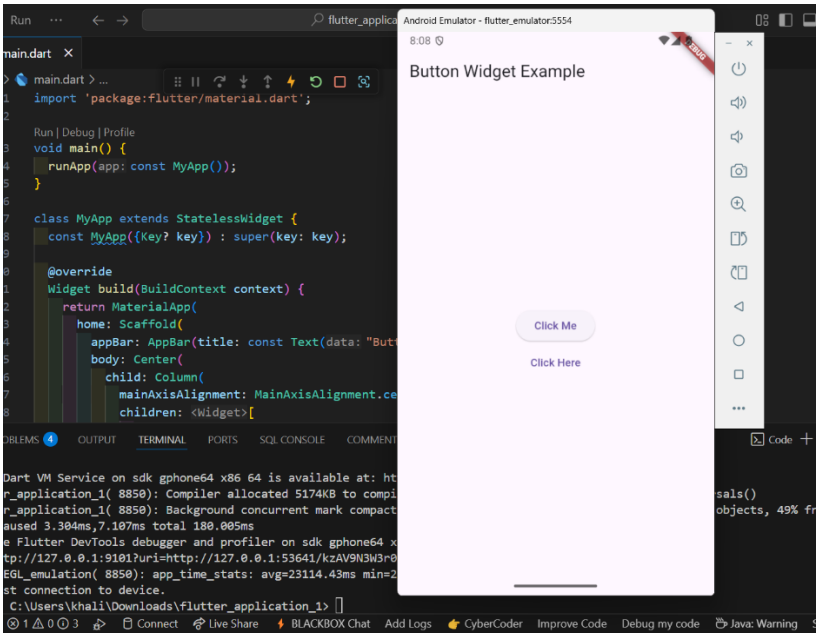


2. Button Widgets Example

Code:

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

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text("Button Widget Example")),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              ElevatedButton(
                onPressed: () {
                  print("Raised Button Clicked");
                },
                child: const Text("Click Me"),
              ),
              TextButton(
                onPressed: () {
                  print("Flat Button Clicked");
                },
                child: const Text("Click Here"),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

Output:

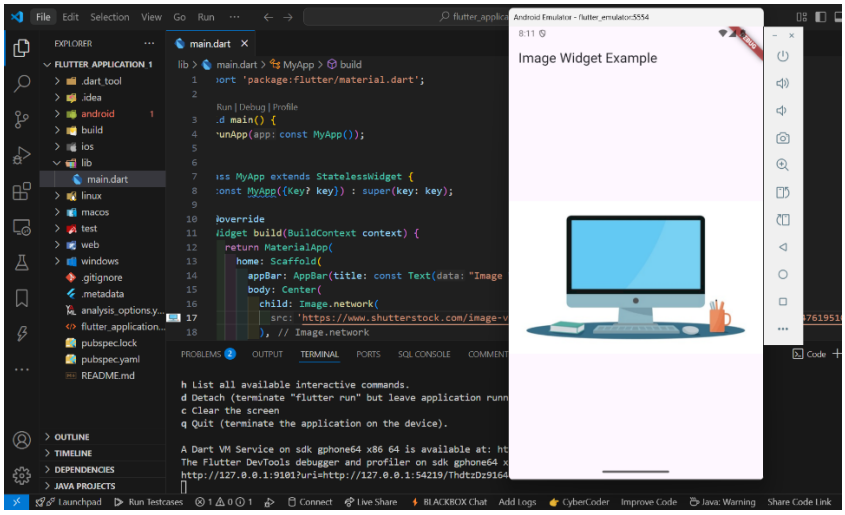


3. Image Widget Example

Code:

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

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text("Image Widget Example")),
        body: Center(
          child: Image.network(
            'https://via.placeholder.com/150', // Replace with any image URL
          ),
        ),
      ),
    );
  }
}
```

Output:



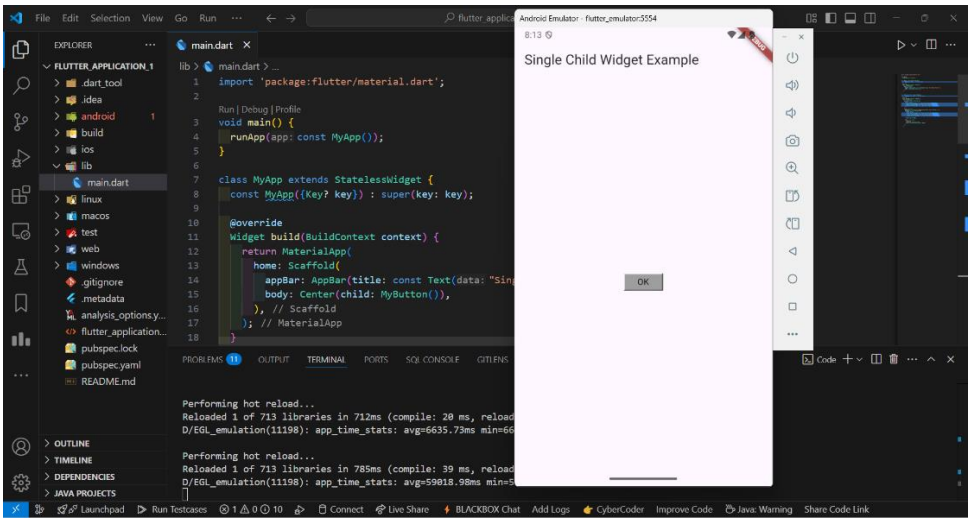
4. Single Child Widget Example

Code:

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

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text("Single Child Widget Example")),
        body: Center(child: MyButton()),
      ),
    );
  }
}

class MyButton extends StatelessWidget {
  const MyButton({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: const BoxDecoration(
        border: Border(
          top: BorderSide(width: 1.0, color: Color(0xFFFFFFFFFF)),
```

```
        left: BorderSide(width: 1.0, color: Color(0xFFFFFFFFFF)),
        right: BorderSide(width: 1.0, color: Color(0xFFFF000000)),
        bottom: BorderSide(width: 1.0, color: Color(0xFFFF000000)),
      ),
    ),
    child: Container(
      padding: const EdgeInsets.symmetric(horizontal: 20.0, vertical: 2.0),
      decoration: const BoxDecoration(
        border: Border(
          top: BorderSide(width: 1.0, color: Color(0xFFFFDFDFDF)),
          left: BorderSide(width: 1.0, color: Color(0xFFFFDFDFDF)),
          right: BorderSide(width: 1.0, color: Color(0xFFFF7F7F7F)),
          bottom: BorderSide(width: 1.0, color: Color(0xFFFF7F7F7F)),
        ),
        color: Colors.grey,
      ),
      child: const Text(
        'OK',
        textAlign: TextAlign.center,
        style: TextStyle(color: Colors.black),
      ),
    ),
  );
 }
}
```

Output:



Name: Atif Ansari Roll no: 04 Class: D15B

MPL Experiment - 4

* Aim: To create an Interactive form using Form Widget.

* Theory: Forms are essential for user interactions in mobile applications, allowing users to enter and submit information. In Flutter, the Form widget provides a structured way to handle user input with validation. The key concepts include:

1. GlobalKey <formstate> : A global key to manage form validation and submission.
2. TextFormField : A field where users input text, with built-in validation.
3. DropdownButtonFormField : A dropdown menu for selecting options.
4. Checkbox : A selection box for Boolean values.
5. ElevatedButton : A button to validate and submit the form.

The form validates user input, displays error messages for invalid input, and processes data upon successful submission.

* Conclusion: This experiment successfully demonstrates how to create and validate a form in Flutter. The form accepts user input, ensures correct data entry through validation, and processes the data when submitted.

1. Interactive Form

Code:

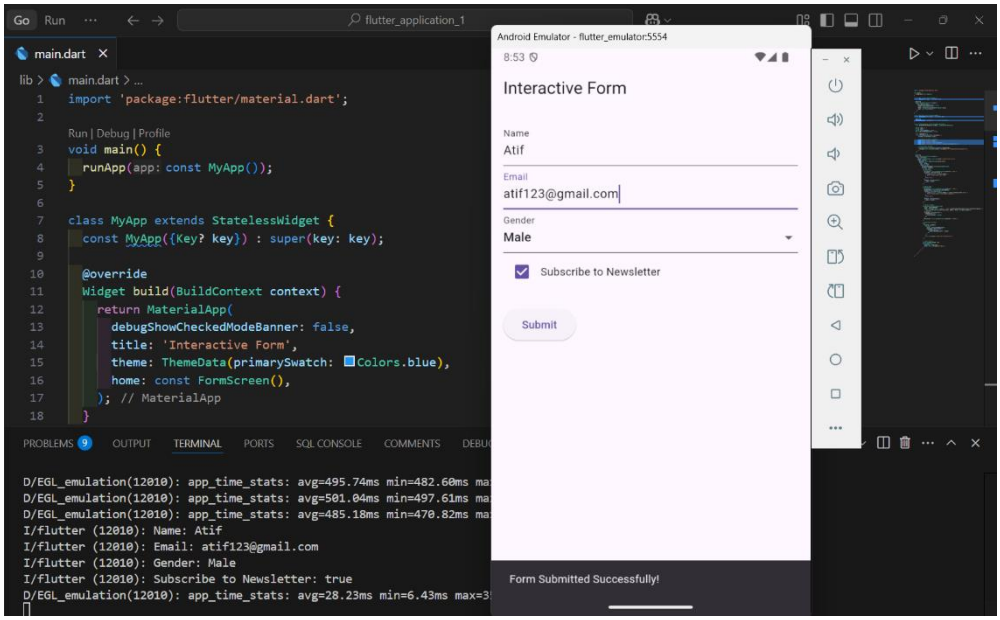
```
import 'package:flutter/material.dart';

void main() {
 runApp(const MyApp());
}

class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);

 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   debugShowCheckedModeBanner: false,
   title: 'Interactive Form',
   theme: ThemeData(primarySwatch: Colors.blue),
   home: const FormScreen(),
  );
 }
}

class FormScreen extends StatefulWidget {
 const FormScreen({Key? key}) : super(key: key);

 @override
 _FormScreenState createState() => _FormScreenState();
}

class _FormScreenState extends State<FormScreen> {
 final _formKey = GlobalKey<FormState>();

 String _name = '';
 String _email = '';
 String _selectedGender = 'Male';
 bool _subscribeToNewsletter = false;

 void _submitForm() {
  if (_formKey.currentState!.validate()) {
   _formKey.currentState!.save();

   // Display user input in console
   print('Name: $_name');
   print('Email: $_email');
   print('Gender: $_selectedGender');
```

```
    print('Subscribe to Newsletter: $_subscribeToNewsletter');

    // Show success message
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('Form Submitted Successfully!')),
    );
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Interactive Form')),
    body: Padding(
      padding: const EdgeInsets.all(16.0),
      child: Form(
        key: _formKey,
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            // Name Input
            TextFormField(
              decoration: const InputDecoration(labelText: 'Name'),
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Please enter your name';
                }
                return null;
              },
              onSaved: (value) {
                _name = value!;
              },
            ),

            // Email Input
            TextFormField(
              decoration: const InputDecoration(labelText: 'Email'),
              keyboardType: TextInputType.emailAddress,
              validator: (value) {
                if (value == null || value.isEmpty || !value.contains('@')) {
                  return 'Please enter a valid email address';
                }
                return null;
              },
              onSaved: (value) {
```

```
            _email = value!;
          },
        ),

        // Gender Dropdown
        DropdownButtonFormField<String>(
          value: _selectedGender,
          items: ['Male', 'Female', 'Other'].map((gender) {
            return DropdownMenuItem(value: gender, child: Text(gender));
          }).toList(),
          onChanged: (value) {
            setState(() {
              _selectedGender = value!;
            });
          },
          decoration: const InputDecoration(labelText: 'Gender'),
        ),

        // Newsletter Checkbox
        Row(
          children: [
            Checkbox(
              value: _subscribeToNewsletter,
              onChanged: (value) {
                setState(() {
                  _subscribeToNewsletter = value!;
                });
              },
            ),
            const Text('Subscribe to Newsletter'),
          ],
        ),

        // Submit Button
        const SizedBox(height: 20),
        ElevatedButton(
          onPressed: _submitForm,
          child: const Text('Submit'),
        ),
      ],
    ),
  ),
 );
}
```

}

Output:



2. Login Form with Email and Password Fields

Code:

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

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  static const String _title = 'Sample App';

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: _title,
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          elevation: 0,
          backgroundColor: Colors.blue[300],
          centerTitle: true,
          title: const Text(
            'Login',
            style: TextStyle(
              color: Colors.white,
              fontFamily: 'Roboto',
              fontSize: 24,
```

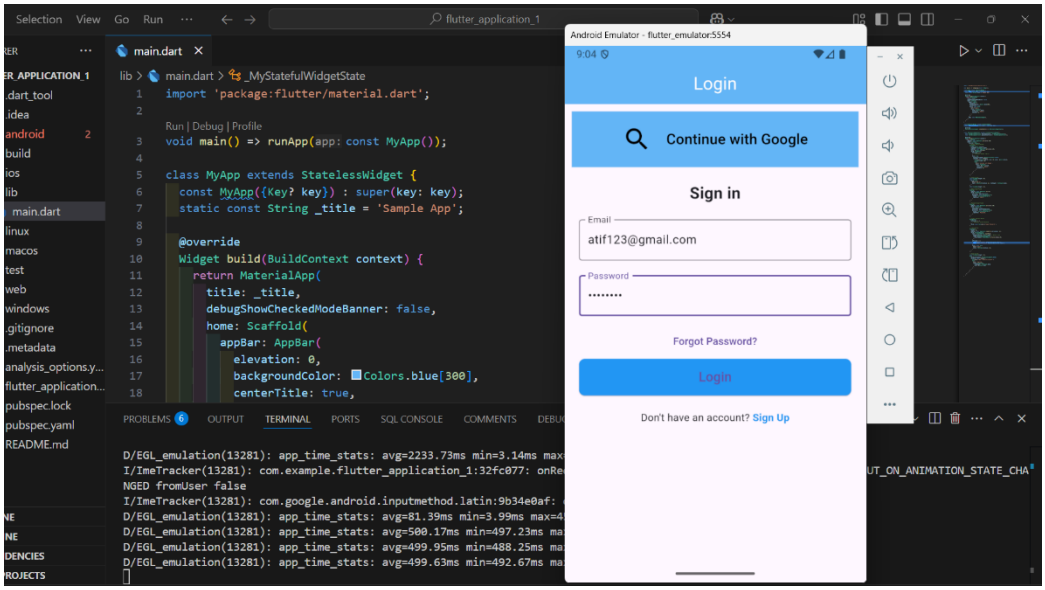
```
          ),
        ),
      ),
      body: const MyStatefulWidget(),
    ),
  );
 }
}

class MyStatefulWidget extends StatefulWidget {
 const MyStatefulWidget({Key? key}) : super(key: key);

 @override
 State<MyStatefulWidget> createState() => _MyStatefulWidgetState();
}

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
 final TextEditingController emailController = TextEditingController();
 final TextEditingController passwordController = TextEditingController();

 @override
 Widget build(BuildContext context) {
  return Padding(
   padding: const EdgeInsets.all(10),
   child: ListView(
    children: <Widget>[
     // Google Sign-In Button
     Container(
      width: double.infinity,
      alignment: Alignment.center,
      padding: const EdgeInsets.all(10),
      color: Colors.blue[300],
      child: TextButton(
       onPressed: () {
        // Implement Google Sign-in
       },
       child: Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: const [
         Icon(Icons.search, size: 40, color: Colors.black),
         SizedBox(width: 20),
         Text(
          'Continue with Google',
          style: TextStyle(
           color: Colors.black,
```

```
          fontFamily: 'Roboto',
          fontSize: 20,
        ),
      ),
    ],
   ),
  ),
),

// Sign-in Text
const SizedBox(height: 20),
const Center(
  child: Text(
    'Sign in',
    style: TextStyle(fontSize: 22, fontWeight: FontWeight.bold),
  ),
),
const SizedBox(height: 10),

// Email Input Field
Padding(
  padding: const EdgeInsets.all(10),
  child: TextField(
    controller: emailController,
    decoration: const InputDecoration(
      border: OutlineInputBorder(),
      labelText: 'Email',
    ),
  ),
),

// Password Input Field
Padding(
  padding: const EdgeInsets.all(10),
  child: TextField(
    obscureText: true,
    controller: passwordController,
    decoration: const InputDecoration(
      border: OutlineInputBorder(),
      labelText: 'Password',
    ),
  ),
),

// Forgot Password Button
```

```
TextButton(
  onPressed: () {
    // Implement Forgot Password
  },
  child: const Text('Forgot Password?'),
),

// Login Button
Container(
  height: 50,
  padding: const EdgeInsets.symmetric(horizontal: 10),
  child: ElevatedButton(
    style: ElevatedButton.styleFrom(
      backgroundColor: Colors.blue, // Button color
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(8),
      ),
    ),
    onPressed: () {
      print('Email: ${emailController.text}');
      print('Password: ${passwordController.text}');
    },
    child: const Text(
      'Login',
      style: TextStyle(fontSize: 18),
    ),
  ),
),

// Sign Up Text
const SizedBox(height: 20),
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: const [
    Text("Don't have an account? "),
    Text(
      'Sign Up',
      style: TextStyle(
        fontWeight: FontWeight.bold,
        color: Colors.blue,
      ),
    ),
  ],
),
],
```

```
    ),
   );
  }
}
```

Output:



3. Form Validation

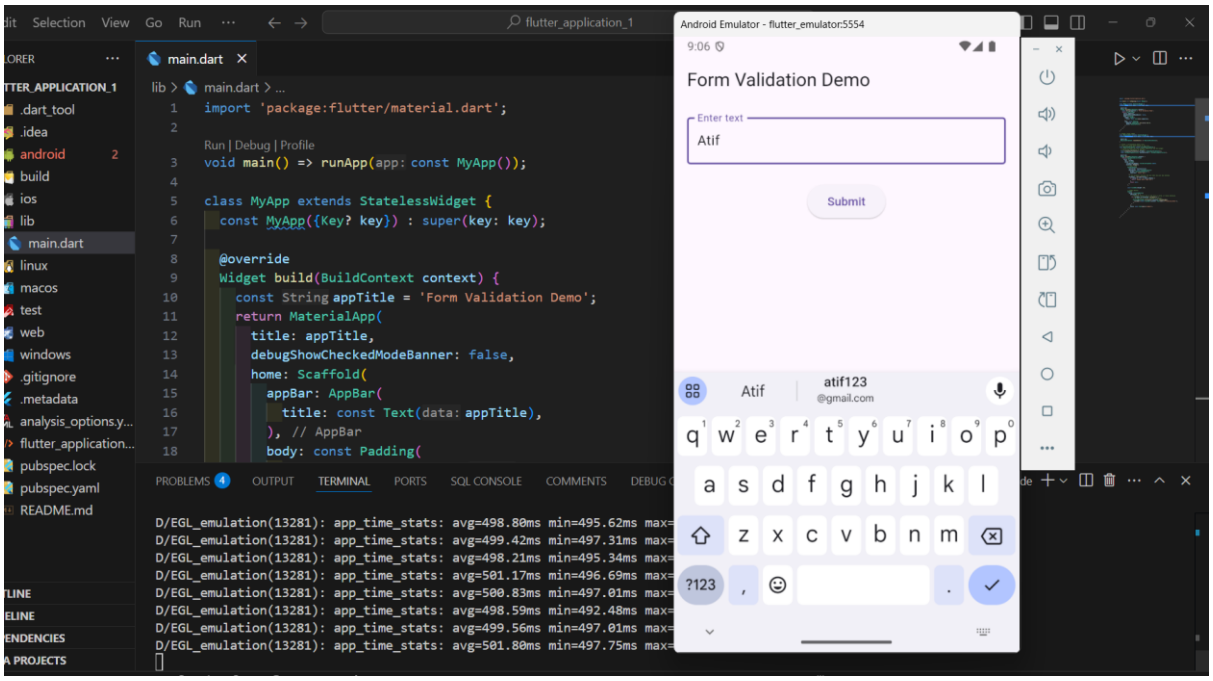
Code:

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

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);

 @override
 Widget build(BuildContext context) {
  const appTitle = 'Form Validation Demo';
  return MaterialApp(
   title: appTitle,
   debugShowCheckedModeBanner: false,
   home: Scaffold(
    appBar: AppBar(
     title: const Text(appTitle),
    ),
    body: const Padding(
     padding: EdgeInsets.all(16.0),
     child: MyCustomForm(),
    ),
```

```
    ),
   );
  }
}

// Create a Form widget.
class MyCustomForm extends StatefulWidget {
  const MyCustomForm({Key? key}) : super(key: key);

  @override
  MyCustomFormState createState() => MyCustomFormState();
}

// Create a corresponding State class.
// This class holds data related to the form.
class MyCustomFormState extends State<MyCustomForm> {
  // Create a global key that uniquely identifies the Form widget
  // and allows validation of the form.
  final _formKey = GlobalKey<FormState>();
  final TextEditingController textController = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return SingleChildScrollView(
      child: Form(
        key: _formKey,
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            // Input Field
            TextFormField(
              controller: textController,
              decoration: const InputDecoration(
                border: OutlineInputBorder(),
                labelText: 'Enter text',
              ),
              // The validator receives the text that the user has entered.
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Please enter some text';
                }
                return null;
              },
            ),
```

```
        const SizedBox(height: 20),

        // Submit Button
        Center(
          child: ElevatedButton(
            onPressed: () {
              // Validate returns true if the form is valid, or false otherwise.
              if (_formKey.currentState!.validate()) {
                // If the form is valid, display a snackbar.
                ScaffoldMessenger.of(context).showSnackBar(
                  const SnackBar(content: Text('Processing Data')),
                );
              }
            },
            child: const Text('Submit'),
          ),
        ),
      ],
    ),
  ),
 );
 }
}
```

Output:



Name: Atif Ansari Roll no: 04 Class: D15B

MPL Experiment - 5

* Aim: To implement Navigation, Routing, and Gestures in Flutter.

* Theory:

Navigation and Routing in Flutter.
In Flutter, screens are referred to as routes. Navigation between these routes is managed using the Navigator class. Flutter provides multiple ways to navigate between screens:

1. Using Navigator.push() and Navigator.pop()
- Navigator.push(context, MaterialPageRoute(builder: (context) => SecondScreen())) is used to navigate from one screen to another.
- Navigator.pop(context) is used to return to the previous screen.

2. Named Routes.
- Named routes allow better management of navigation in larger apps by defining routes in a central location.
- Navigator.pushNamed(context, '/second') is used to navigate to a named route.
- Navigator.pop(context) is used to return to the previous route.

Gestures in Flutter: Gestures allow users to interact with UI elements using touch-based

actions like tap, drag, swipe and pinch. The GestureDetector widget in Flutter listens for user gestures and responds accordingly.

1. Single Gesture Handling (GestureDetector): Used to detect single gestures like tap, double tap, long press, drag, etc.

2. Multiple Gesture Handling (RawGestureDetector): Used when handling multiple gestures simultaneously. The AllowMultipleGestureRecognizer class enables recognition of multiple gestures within the same widget tree.

* Conclusion: This experiment demonstrated how to implement navigation, routing, and gestures in a Flutter application. By using Navigator.push() and Navigator.pop(), we successfully transitioned between screens. Additionally, named routes provided a ~~structed~~ structured approach to navigation management.

For Gestures, the GestureDetector widget enabled user interaction through touch-based actions. The RawGestureDetector facilitated handling multiple gestures simultaneously. Understanding navigation and gestures in fundamental for building interactive and user-friendly Flutter applications.

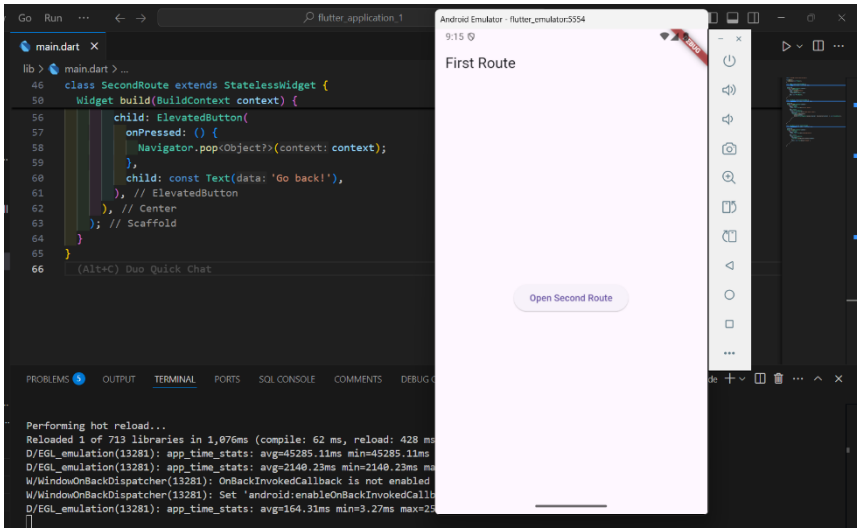
1. Basic Navigation Using Navigator.push() and Navigator.pop()

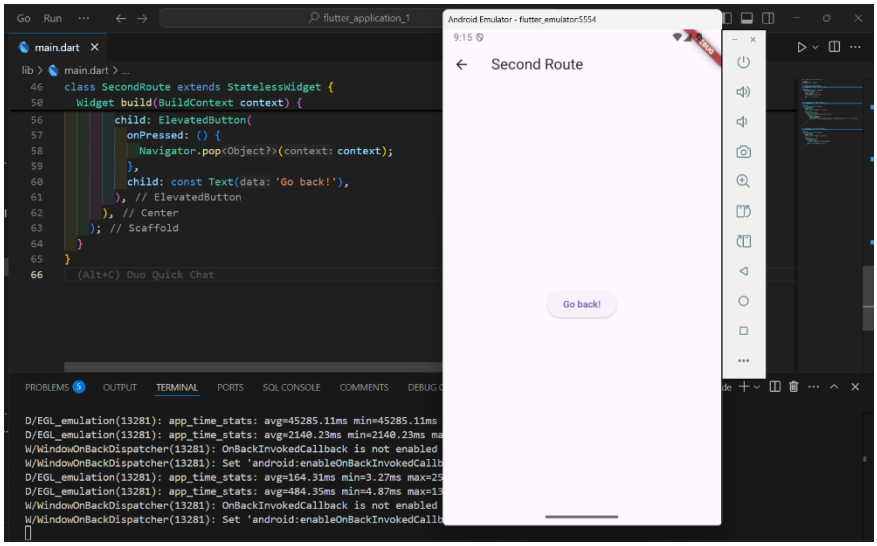
Code:

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

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Navigation Basics',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const FirstRoute(),
    );
  }
}

class FirstRoute extends StatelessWidget {
  const FirstRoute({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Route'),
      ),
      body: Center(
        child: ElevatedButton(
          child: const Text('Open Second Route'),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => const SecondRoute()),
            );
          },
        ),
      ),
```

```
    );
  }
}

class SecondRoute extends StatelessWidget {
  const SecondRoute({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Second Route'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: const Text('Go back!'),
        ),
      ),
    );
  }
}
```

Ouput:





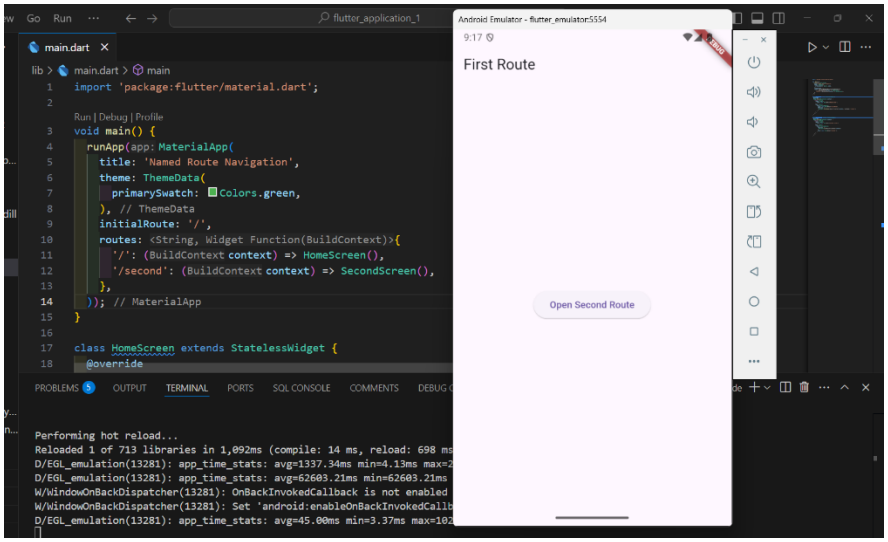
2. Navigation Using Named Routes

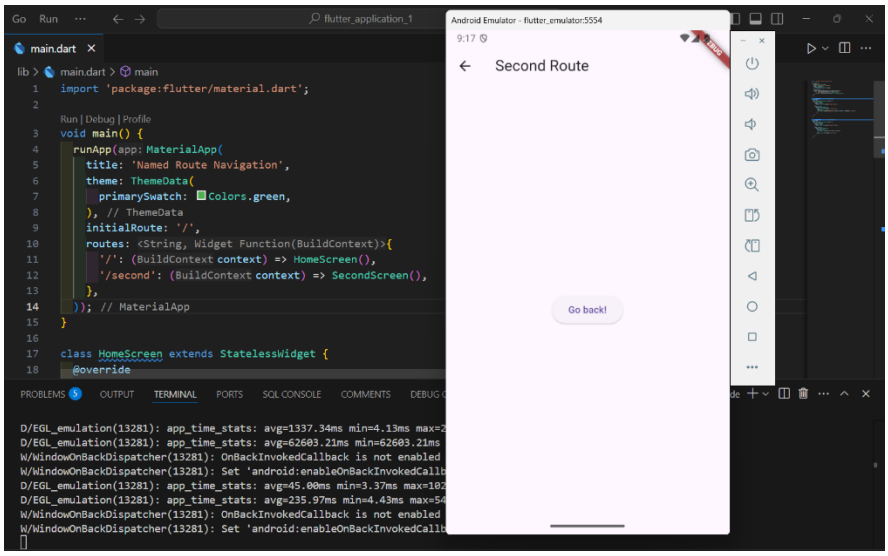
Code:

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

void main() {
 runApp(const MyApp());
}

class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);

 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   title: 'Named Route Navigation',
   theme: ThemeData(
    primarySwatch: Colors.green,
   ),
   initialRoute: '/',
   routes: {
    '/': (context) => const HomeScreen(),
    '/second': (context) => const SecondScreen(),
   },
  );
 }
}

class HomeScreen extends StatelessWidget {
```

```
 const HomeScreen({Key? key}) : super(key: key);

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: const Text('Home Screen'),
   ),
   body: Center(
    child: ElevatedButton(
     child: const Text('Click Here'),
     onPressed: () {
      Navigator.pushNamed(context, '/second');
     },
    ),
   ),
  );
 }
}

class SecondScreen extends StatelessWidget {
 const SecondScreen({Key? key}) : super(key: key);

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: const Text("Second Screen"),
   ),
   body: Center(
    child: ElevatedButton(
     onPressed: () {
      Navigator.pop(context);
     },
     child: const Text('Go back!'),
    ),
   ),
  );
 }
}
```

Ouput:





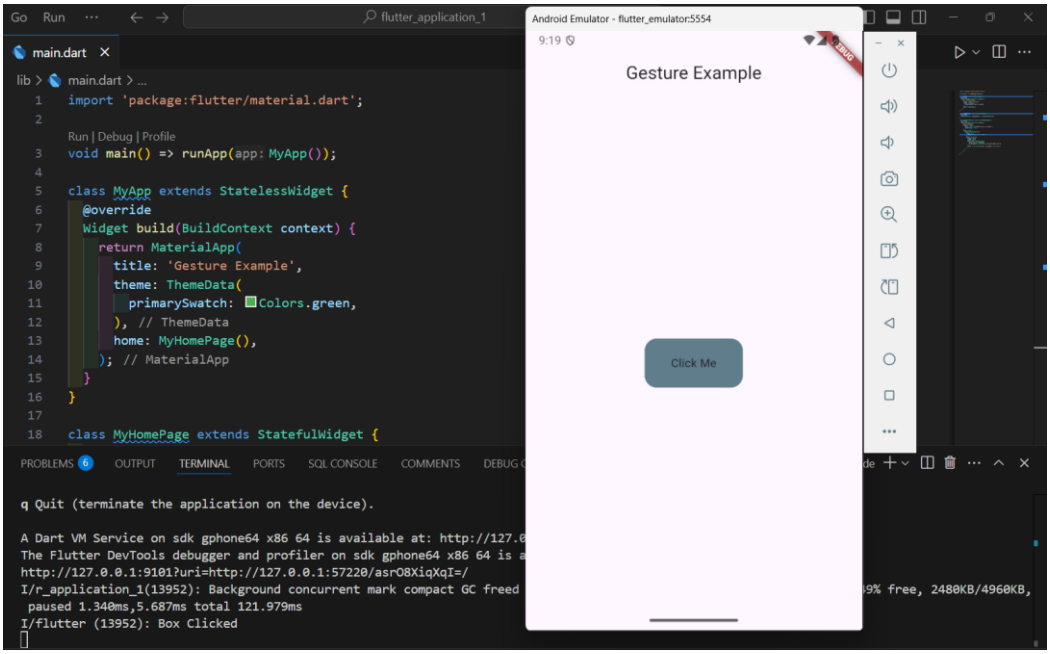
3. Gesture Example Using GestureDetector

Code:

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

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Gesture Example',
      theme: ThemeData(
        primarySwatch: Colors.green,
      ),
      home: MyHomePage(),
```

```
    );
  }
}

class MyHomePage extends StatefulWidget {
  @override
  MyHomePageState createState() => MyHomePageState();
}

class MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Gesture Example'),
        centerTitle: true,
      ),
      body: Center(
        child: GestureDetector(
          onTap: () {
            print('Box Clicked');
          },
          child: Container(
            height: 60.0,
            width: 120.0,
            decoration: BoxDecoration(
              color: Colors.blueGrey,
              borderRadius: BorderRadius.circular(15.0),
            ),
            child: const Center(child: Text('Click Me')),
          ),
        ),
      ),
    );
  }
}
```

Output:



4. Multiple Gesture Handling Using RawGestureDetector

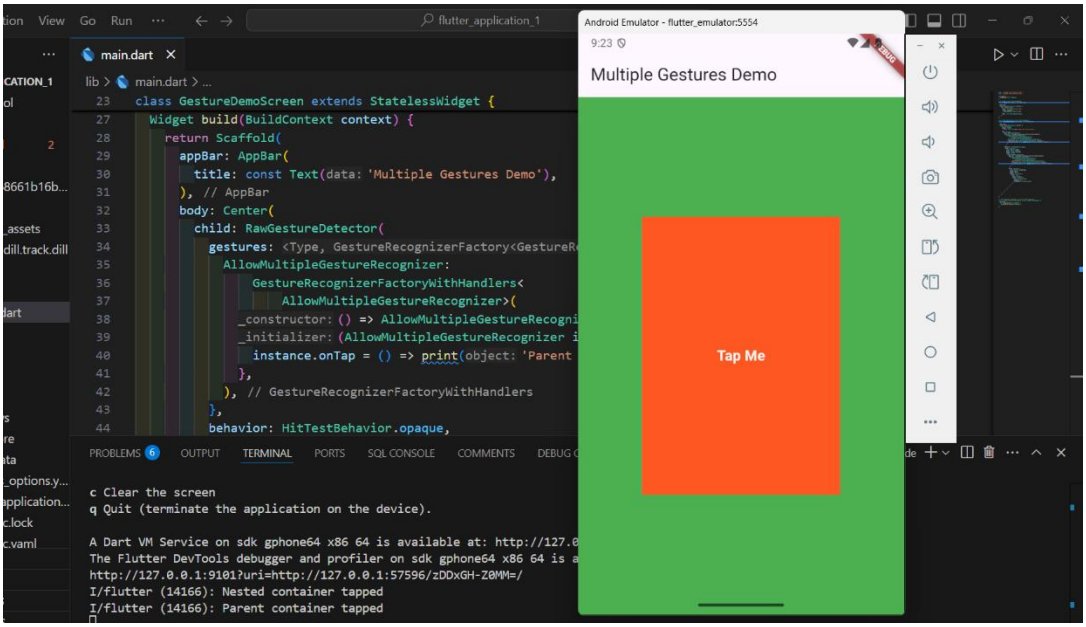
Code:

```
import 'package:flutter/gestures.dart';
import 'package:flutter/material.dart';

void main() {
 runApp(const MyApp());
}

class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);

 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   title: 'Multiple Gestures Demo',
   theme: ThemeData(
    primarySwatch: Colors.blue,
   ),
   home: const GestureDemoScreen(),
  );
 }
}

class GestureDemoScreen extends StatelessWidget {
 const GestureDemoScreen({Key? key}) : super(key: key);
```

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Multiple Gestures Demo'),
    ),
    body: Center(
      child: RawGestureDetector(
        gestures: {
          AllowMultipleGestureRecognizer:
              GestureRecognizerFactoryWithHandlers<
                  AllowMultipleGestureRecognizer>(
            () => AllowMultipleGestureRecognizer(),
            (AllowMultipleGestureRecognizer instance) {
              instance.onTap = () => print('Parent container tapped');
            },
          ),
        },
        behavior: HitTestBehavior.opaque,
        child: Container(
          color: Colors.green,
          width: double.infinity,
          height: double.infinity,
          child: Center(
            child: RawGestureDetector(
              gestures: {
                AllowMultipleGestureRecognizer:
                    GestureRecognizerFactoryWithHandlers<
                        AllowMultipleGestureRecognizer>(
                  () => AllowMultipleGestureRecognizer(),
                  (AllowMultipleGestureRecognizer instance) {
                    instance.onTap = () => print('Nested container tapped');
                  },
                ),
              },
              child: Container(
                color: Colors.deepOrange,
                width: 250.0,
                height: 350.0,
                child: const Center(
                  child: Text(
                    "Tap Me",
                    style: TextStyle(
                      color: Colors.white,
```

```
                fontSize: 18,
                fontWeight: FontWeight.bold,
              ),
            ),
          ),
        ),
      ),
    ),
  ),
 );
 }
}

/// Custom Gesture Recognizer to allow multiple gestures
class AllowMultipleGestureRecognizer extends TapGestureRecognizer {
 @override
 void rejectGesture(int pointer) {
  acceptGesture(pointer);
 }
}
```

Output:



Name: Atif Ansari Roll no: 04 Class: D15B

MPL Experiment - 6

Aim: To set up Firebase with Flutter for iOS and Android Apps.

Theory: Firebase is a Backend-as-a-Service (BaaS) platform by Google that provides serverless backend functionalities such as database management, authentication, cloud storage, and real-time analytics. It allows developers to build scalable and real-time applications without managing dedicated backend servers.

Key Features of Firebase for Flutter:

- Firebase Authentication: Supports login via Google, Email/Password, and Social media.
- Cloud Firestore: A NoSQL database that provides real-time data synchronization.
- Firebase Realtime Database: Syncs data across multiple users instantly.
- Firebase Cloud Messaging: Enables push notifications for mobile applications.
- Firebase Storage: Stores and retrieves user-generated media like images and videos.
- Firebase Analytics: Tracks user interactions

within the app for insights and improvements.

Other advantages of using firebase are:

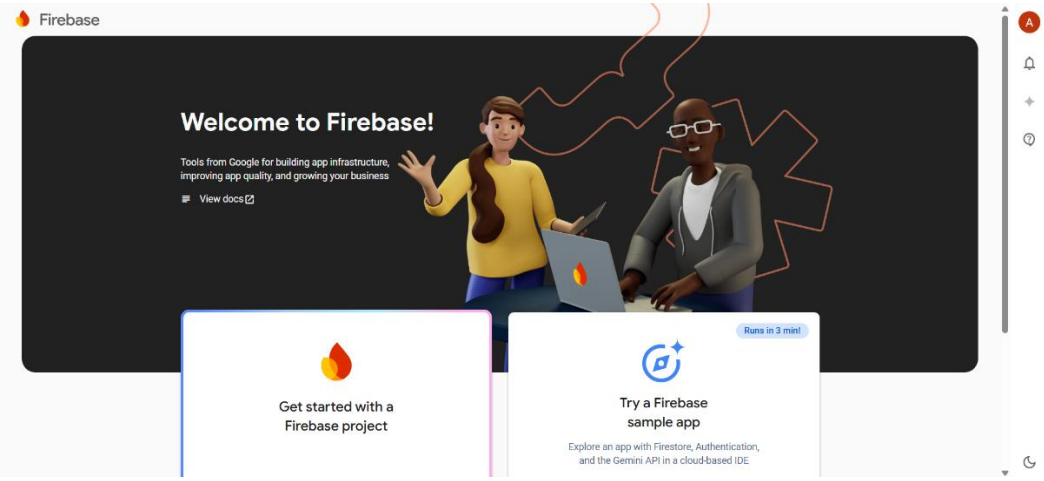
1. Cross-Platform Support.
2. Easy Integration.
3. Serverless Architecture.
4. Real-time Capabilities.

Conclusion: In this experiment, we successfully configured and integrated Firebase with a flutter application for both Android and iOS. This setup established a real-time backend connection, enabling services such as authentication, database management, and cloud storage.

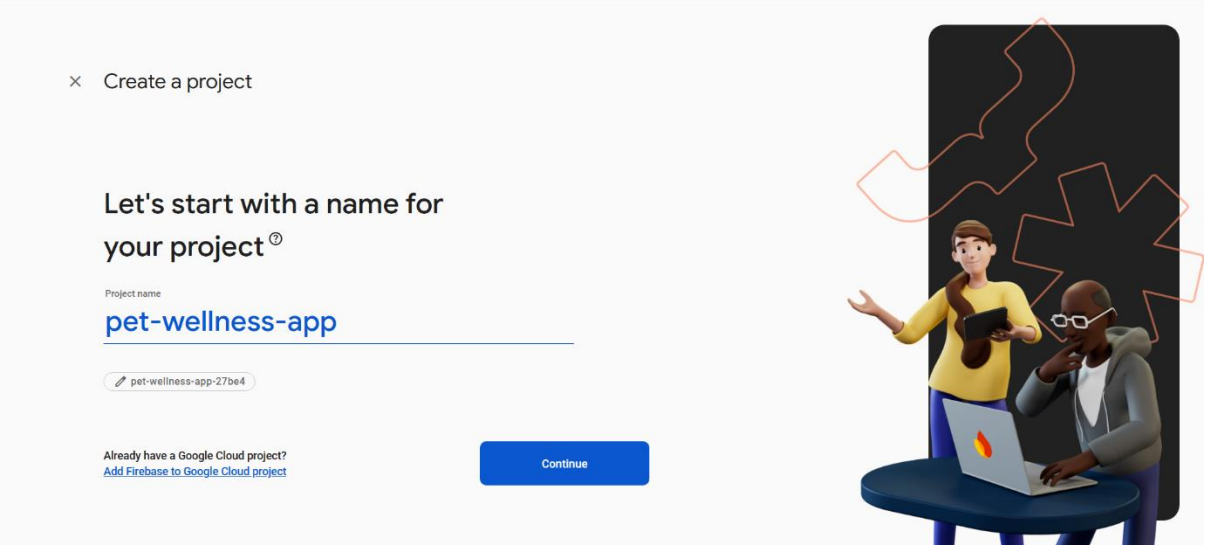
With firebase integrated, the app can now:

- Support user authentication.
- Store and manage data using firestore or Realtime Database.
- Send push notifications using firebase Cloud Messaging.
- Upload and retrieve media files using Firebase Storage.

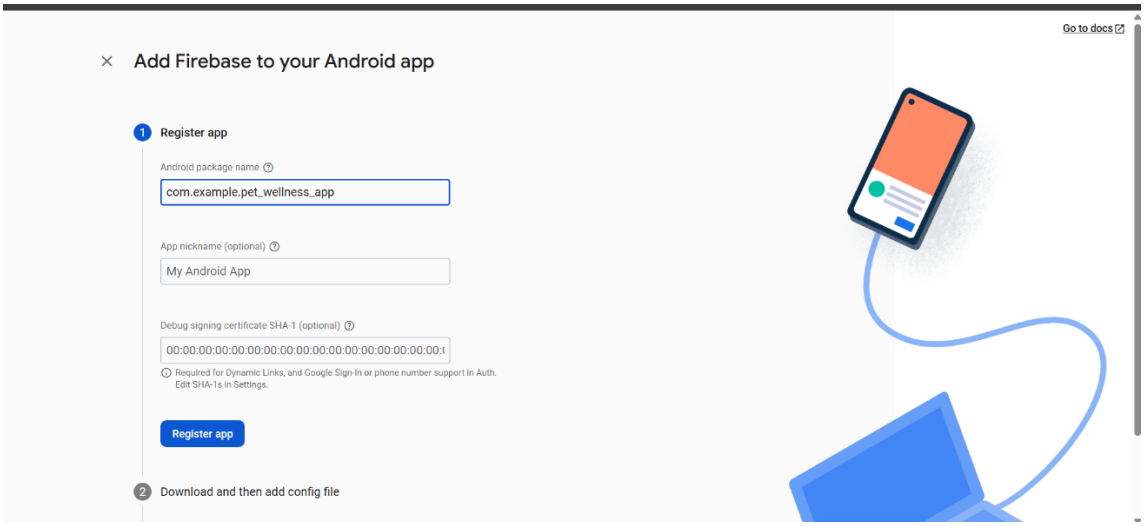
Step 1: First, log in with your Google account to manage your Firebase projects.



Step 2: From within the Firebase dashboard, select the Create new project button and give it a name:

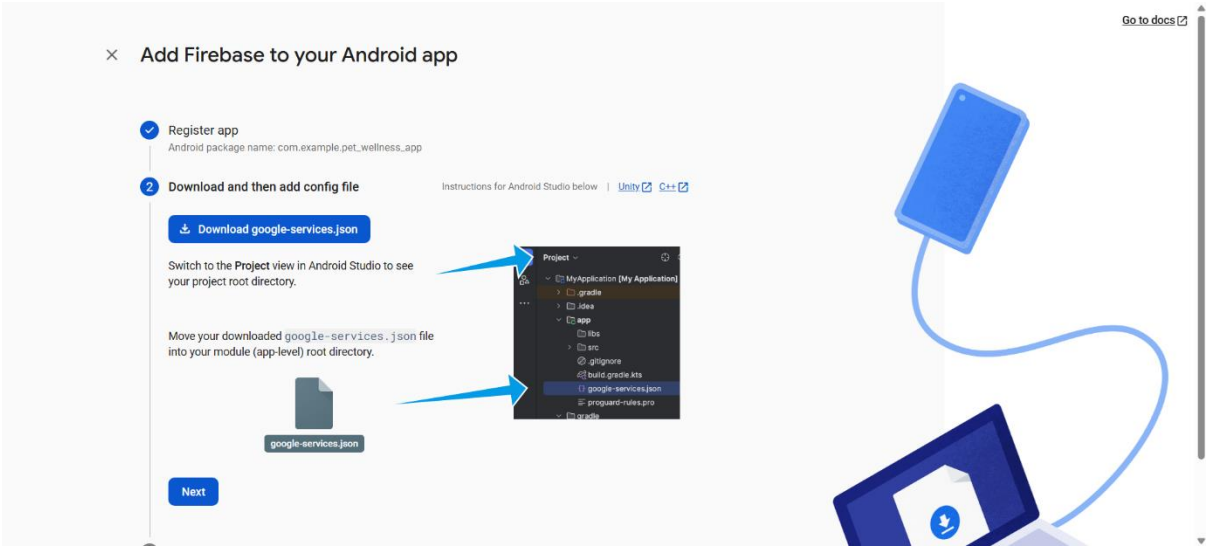


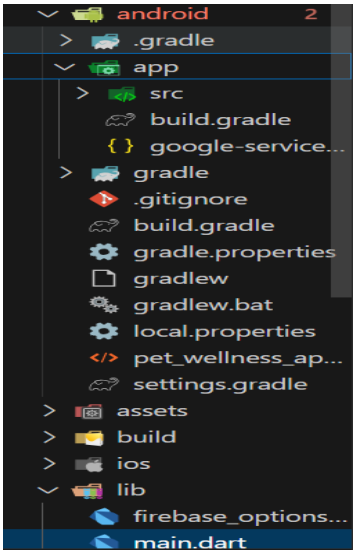
Step 3: In order to add Android support to our Flutter application, select the Android logo from the dashboard. This brings us to the following screen:



Step 4: The next step is to add the Firebase configuration file into our Flutter project. This is important as it contains the API keys and other critical information for Firebase to use.

Select Download google-services.json from this page:





Step 5: Open android/build.gradle in your code editor and modify it to include the following:

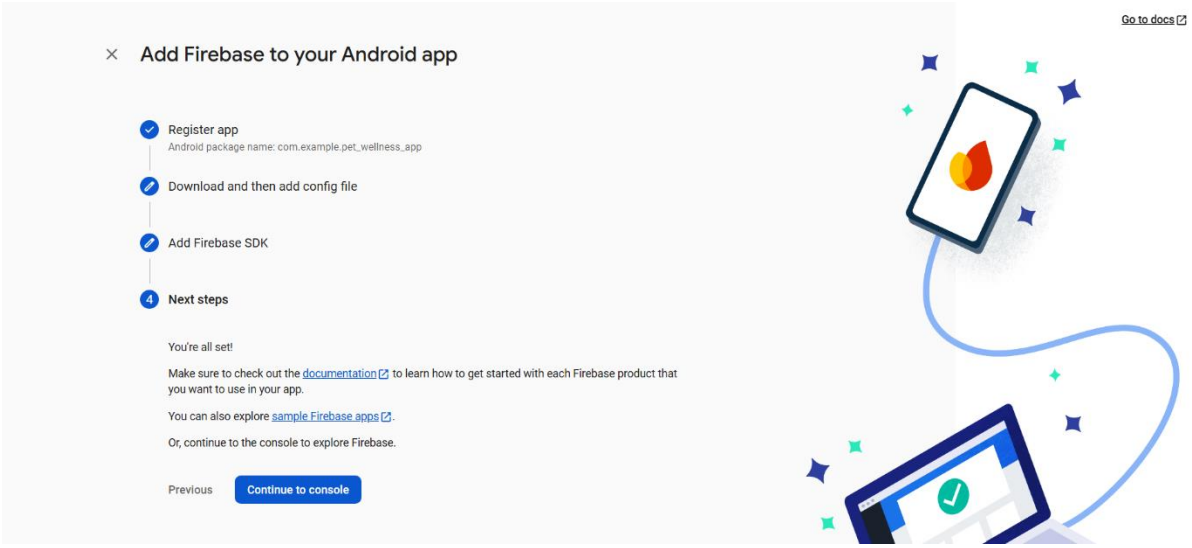
```
buildscript {
    repositories {
        google()
        mavenCentral()
    }
    dependencies {
        // START: FlutterFire Configuration
        classpath 'com.google.gms:google-services:4.3.15'
        // END: FlutterFire Configuration
        // Use a version of the Android Gradle Plugin that supports your setup.
        // Check https://developer.android.com/studio/releases/gradle-plugin for the latest stable version.
        classpath 'com.android.tools.build:gradle:8.3.0'
        id 'com.google.gms.google-services' version '4.4.2' apply false
    }
}
```

Step 6: Finally, update the app level file at android/app/build.gradle to include the following:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'com.google.gms.google-services'

android {
    namespace "com.example.pet_wellness_app"
    compileSdkVersion flutter.compileSdkVersion

    defaultConfig {
        applicationId "com.example.pet_wellness_app"
        minSdkVersion 23
        targetSdkVersion flutter.targetSdkVersion
        versionCode flutter.versionCode
        versionName flutter.versionName
    }
```

Step 7: With this update, we're essentially applying the Google Services plugin as well as looking at how other Flutter Firebase plugins can be activated such as Analytics.

From here, run your application on an Android device or simulator. If everything has worked correctly, you should get the following message in the dashboard:



MPL Experiment - 7

* **Aim:** To implement the "Add to Home Screen" feature in a Progressive Web App (PWA) using a service worker, manifest file, and caching techniques, allowing users to install the web application on their device like a native app.

* **Theory:** A PWA is a modern web application that behaves like a native mobile or desktop app. It allows users to install the app on their home screen, work offline, and load quickly, providing a seamless user experience.

Key features of a PWA:

- **Responsive & Adaptive** - Works on all screen sizes.
- **Offline Support** - Uses a service worker for caching and offline access.
- **App-like Experience** - Runs in standalone mode, without a browser UI.
- **Push Notifications** - Supports background updates.
- **Secure & Fast** - Uses HTTPS and caching techniques.

How "Add to Home Screen" Works in a PWA?

1. **Manifest File (manifest.json)**
   - Defines app metadata (name, icons, theme, start URL, etc.).
   - Enables installation on the home screen.

2. **Service Worker (serviceworker.js)**

- Handles caching and offline access.
- Ensures app loads quickly even with poor network conditions.

3. Registration in index.html
- Registers the service worker to enable background operations.

When a PWA is correctly configured, the browser detects it and prompts the user with an "Install this site as an app" option, allowing them to add to their home screen.

* Conclusion: In this experiment, we successfully implemented the "Add to Home Screen" feature in a Progressive Web App (PWA). This was achieved by:
1. Manifest file (manifest.json) - Defined app metadata, making the app installable.
2. Service Worker (serviceworker.js) - Handled caching and enabled offline access.
3. Service Worker Registration (index.html) - Allowed the browser to detect and prompt app installation.

After implementation, we verified the installation process and confirmed that the PWA icon appeared on the home screen. This experiment demonstrated how PWAs provide a native-like experience, improve user engagement, and enhance performance by allowing fast, offline-ready access.

Code Implementation of Experiment 7:

Index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>SneakCart - Sneaker Store</title>
  <link rel="manifest" href="manifest.json" />
  <link rel="stylesheet" href="style.css" />
  <meta name="theme-color" content="#000000" />
</head>
<body>
  <h1>Welcome to SneakCart</h1>
  <p>Shop the coolest sneakers in town!</p>
</body>
</html>
```

Manifest.json:

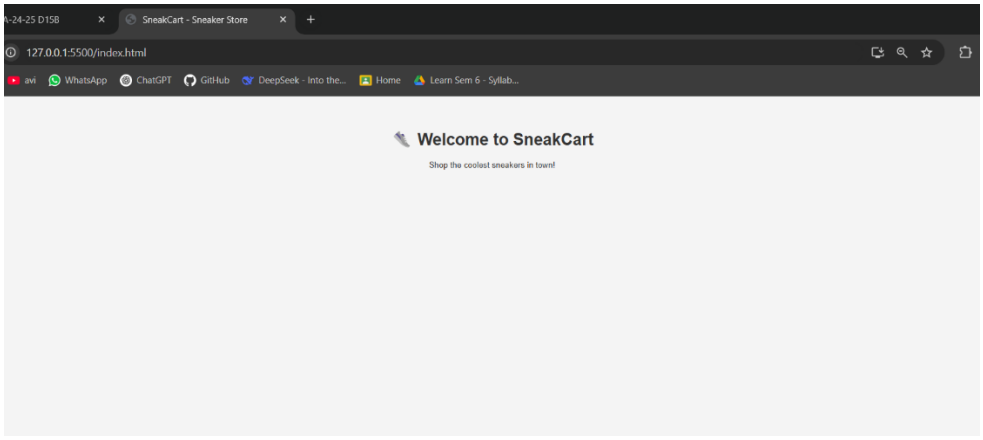
```
{
    "name": "SneakCart - Sneaker Shop",
    "short_name": "SneakCart",
    "start_url": "index.html",
    "display": "standalone",
    "background_color": "#ffffff",
    "theme_color": "#000000",
    "description": "A simple PWA sneaker e-commerce app",
    "icons": [
      {
        "src": "icons/icon-192.png",
        "sizes": "192x192",
        "type": "image/png"
      },
      {
        "src": "icons/icon-512.png",
        "sizes": "512x512",
        "type": "image/png"
      }
    ]
  }
```

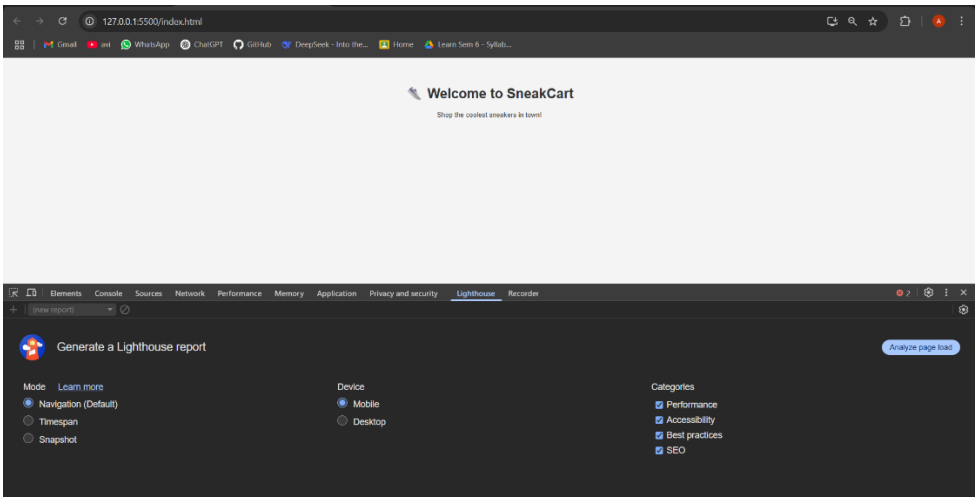
Style.css:

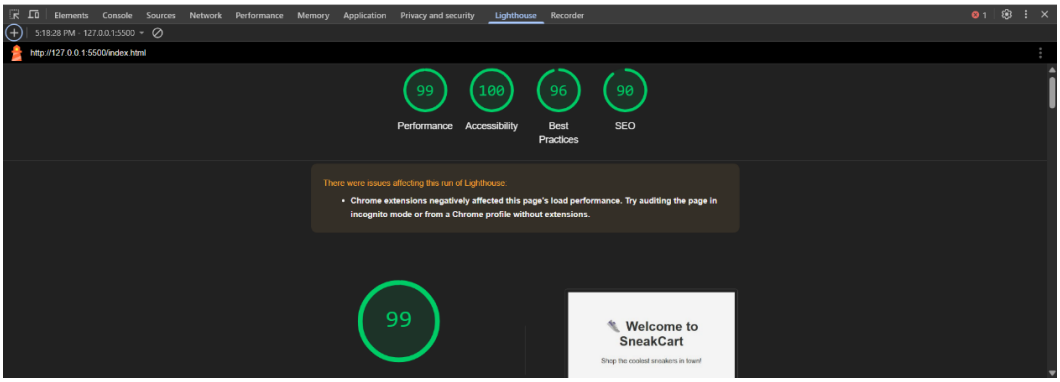
```
body {
```

```
  font-family: sans-serif;
  text-align: center;
  padding: 2rem;
  background-color: #f4f4f4;
  color: #333;
}
```

Output:







Name: Atif Ansari Roll no: 04 Class: D15B

MPL Experiment - 8

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

Theory: A Service Worker is a script that runs in the background of a web browser, independent of the main webpage, and acts as a proxy between the network and the user's browser. It enables features such as caching, push notifications, background sync, and offline access, making Progressive Web Applications (PWAs) more efficient and reliable.

Characteristics of Service Workers:

- Service workers intercept network requests and can modify or respond to them directly from cache, enhancing offline functionality.
- They run seperately from the main JavaScript thread, meaning they don't directly interact with the Document Object Model (DOM).
- Information persistence is managed using IndexedDB instead of global variables.
- Service workers operate only on secure HTTPS connections to prevent security threats.

Key features of Service Workers:

- **Caching:** Stores files for offline use.
- **Network Request Interception:** Controls how requests are handled.

- Push Notifications: Sends updates even when the app is closed.
- Background Sync: Synchronize data when connectivity is restored.

Service Worker Lifecycle:

1. Registration: The browser is informed about the service worker.
2. Installation: Required assets are cached.
3. Activation: The service worker takes control, replacing old versions.

It enhances PWA by improving speed, reducing network dependency, and enabling offline functionality.

Conclusion:

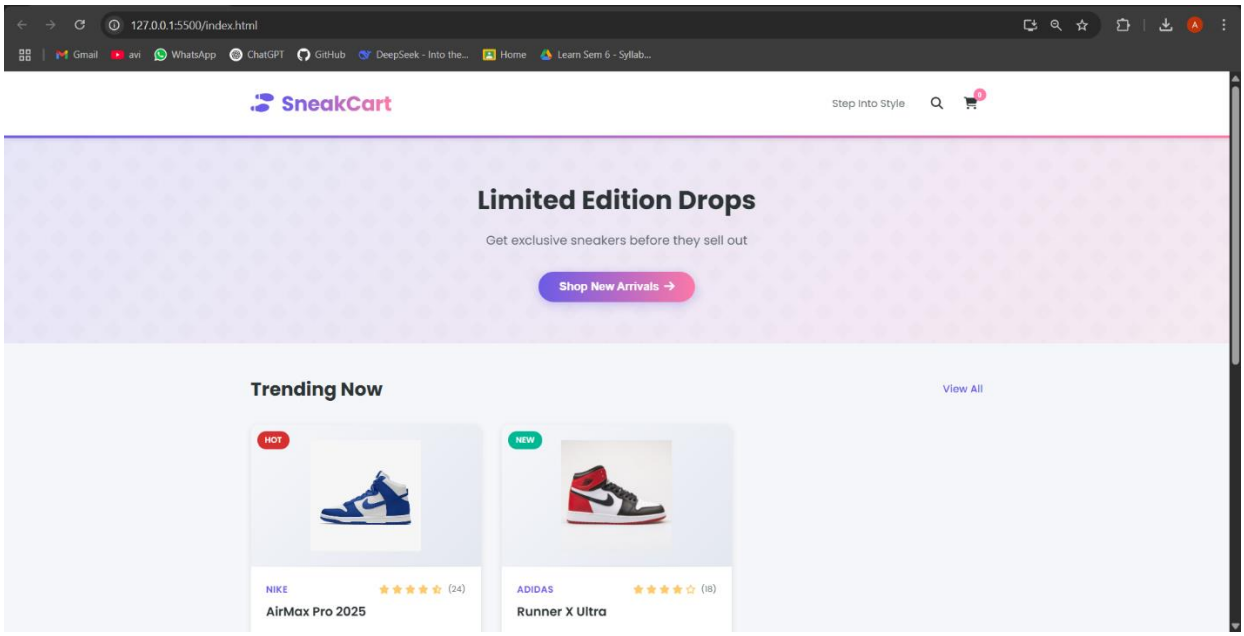
In this experiment, we successfully registered and implemented a service worker, allowing the PWA to function efficiently with caching and offline capabilities. The installation and activation steps ensured that essential assets were stored and outdated versions were replaced. This enhances performance, reliability, and user experience by enabling access even in poor network conditions.

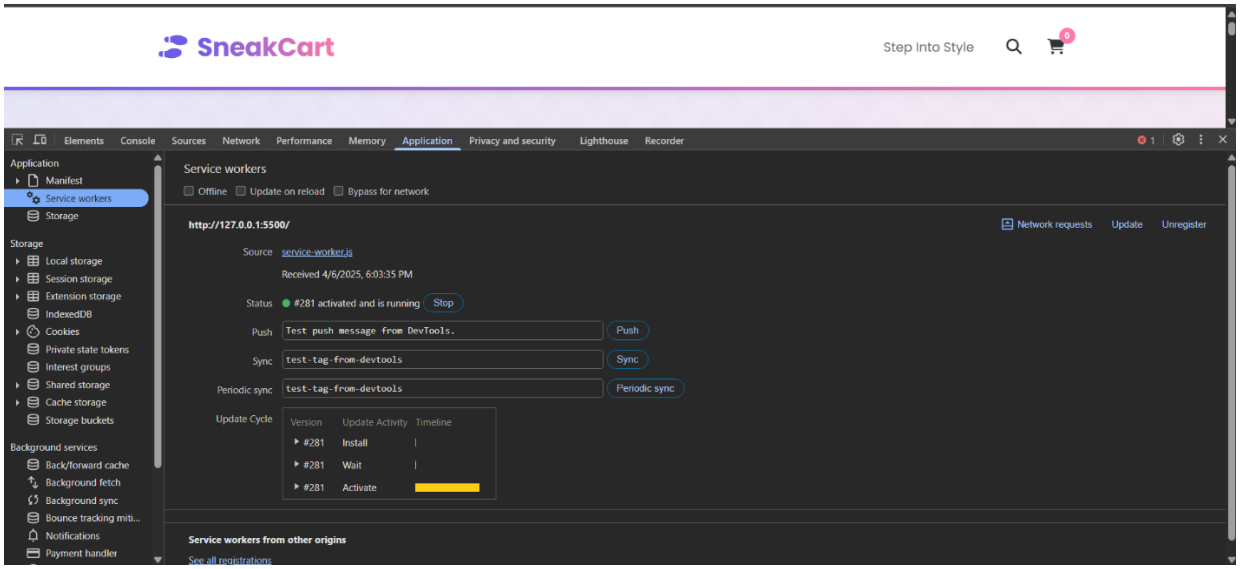
Code Implementation of Experiment 8:

Service-worker.js:

```
const CACHE_NAME = "sneakcart-cache-v1";
const urlsToCache = [
  "/",
  "/index.html",
  "/style.css",
  "/script.js",
  "/offline.html",
  "/manifest.json",
  "/products/shoe1.png",
  "/products/shoe2.jpg",
  "/icons/icon-192.png",
  "/icons/icon-512.png"
];

// Install event
self.addEventListener("install", (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME).then((cache) => {
      console.log("[Service Worker] Caching all files");
      return cache.addAll(urlsToCache);
    })
  );
  self.skipWaiting(); // Activate worker immediately
});

// Activate event
self.addEventListener("activate", (event) => {
  event.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((name) => {
          if (name !== CACHE_NAME) {
            console.log("[Service Worker] Deleting old cache:", name);
            return caches.delete(name);
          }
        })
      );
    })
  );
  self.clients.claim(); // Claim clients immediately
});

// Fetch event
self.addEventListener("fetch", (event) => {
```

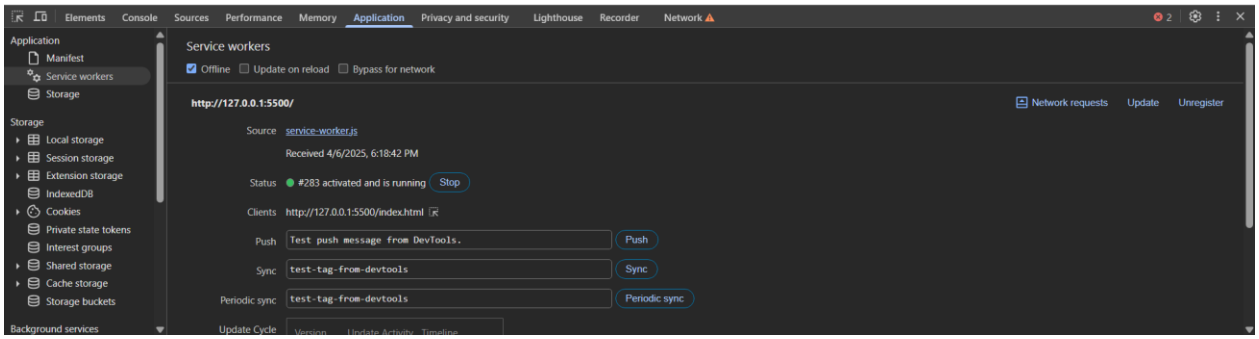
```
// Handle page navigations
if (event.request.mode === "navigate") {
  event.respondWith(
    fetch(event.request).catch(() => {
      return caches.match("/offline.html");
    })
  );
} else {
  // Handle other requests (images, CSS, JS, manifest, etc.)
  event.respondWith(
    caches.match(event.request).then((response) => {
      return (
        response ||
        fetch(event.request).catch(() => {
          // Graceful fallback if resource not cached and offline
          return new Response("", {
            status: 503,
            statusText: "Service Unavailable",
          });
        })
      );
    })
  );
 }
});
```





You're Offline

Please check your internet connection and try again.



Name: Atif Ansari Roll no: 04 Class: D15B

MPL Experiment - 9

Aim: To implement Service Worker events like Fetch, Sync, and Push for an E-commerce Progressive Web App (PWA).

Theory: A Service Worker is a background script that runs independently in the browser without direct user interaction. It acts as a network proxy, allowing developers to manage caching, track network requests, and enable offline-first web applications using the Cache API.

Key Characteristics of Service Workers:

- Operate independently in the background, enabling offline capabilities.
- Function as programmable network proxies to handle network requests efficiently.
- Require HTTPS for security, preventing man-in-the-middle attacks.
- Do not maintain a global state and rely on IndexedDB for persistent data storage.
- Use Promises extensively for asynchronous operations.

Service Worker Events:

1. Fetch Event:
   - Used to track and manage network traffic.
   - Implements caching strategies like "Cache First" and "Network first" to optimize performance and offline access.
   - Cache first: Returns cached data if available;

otherwise, fetches from the network.
- Network first: Tries fetching from the network, first; if unsuccessful, it retrieves cached data.

2. Sync Event:
- Ensures tasks complete even when the internet is temporarily unavailable.
- Data is stored in IndexedDB and processed when the connection is available.

3. Push Event:
- Handles push notifications sent from a server to the user's device.
- The Notification.requestPermission(); method is used to request permission for displaying notifications.

Conclusion:
By Implementing Service Worker events like fetch, Sync, and Push in an E-commerce PWA improves offline support, reliability, and user engagement. Fetch handles caching for better performance, Sync ~~mess~~ manages tasks when offline, and Push enables real-time notifications. These features make the app faster, more interactive and usable even without internet access.

Code Implementation of Experient 9:

index.html:

```
<section class="sync-section">
      <div class="sync-card">
        <h3><i class="fas fa-sync-alt"></i> Try Background Sync</h3>
        <p class="sync-subtitle">Test how SneakCart stays smart — even offline!</p>
        <form id="dummyForm" class="sync-form">
          <input type="text" placeholder="Enter test data..." id="dummyInput" required />
          <button type="submit" class="sync-button">
            <i class="fas fa-paper-plane"></i> Submit
          </button>
        </form>
        <div id="status" class="sync-status">Waiting for input...</div>
      </div>
    </section>
```

service-worker.js:

```
const CACHE_NAME = "sneakcart-cache-v3";
const urlsToCache = [
  "/",
  "/index.html",
  "/offline.html",
  "/style.css",
  "/script.js",
  "/products/shoe1.png",
  "/products/shoe2.jpg",
  "/icons/icon-192.png",
  "/icons/icon-512.png"
];

// Install Event
self.addEventListener('install', event => {
  console.log('[SW] Install event');
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(cache => {
        console.log('[SW] Caching all files');
        return cache.addAll(urlsToCache);
      })
      .then(() => self.skipWaiting())
  );
});
```

```
// Activate Event
self.addEventListener('activate', event => {
  console.log('[SW] Activate event');
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames.map(cache => {
          if (cache !== CACHE_NAME) {
            console.log('[SW] Deleting old cache:', cache);
            return caches.delete(cache);
          }
        })
      );
    })
    .then(() => self.clients.claim())
  );
});

// Fetch Event (Cache with Network Fallback)
self.addEventListener('fetch', event => {
  // Skip non-GET requests
  if (event.request.method !== 'GET') return;

  event.respondWith(
    caches.match(event.request)
      .then(cachedResponse => {
        // Return cached response if found
        if (cachedResponse) {
          console.log(`[SW] Serving from cache: ${event.request.url}`);
          return cachedResponse;
        }

        // Otherwise fetch from network
        return fetch(event.request)
          .then(networkResponse => {
            // Cache the new response if successful
            if (networkResponse && networkResponse.status === 200) {
              const responseToCache = networkResponse.clone();
              caches.open(CACHE_NAME)
                .then(cache => cache.put(event.request, responseToCache));
            }
            return networkResponse;
          })
          .catch(() => {
            // If both fail, show offline page for HTML requests
```

```
          if (event.request.headers.get('accept').includes('text/html')) {
            return caches.match('/offline.html');
          }
        });
    })
  );
});

// Sync Event
self.addEventListener('sync', event => {
  if (event.tag === 'sync-form') {
    console.log('[SW] Background sync triggered');
    event.waitUntil(
      (async () => {
        // Simulate sync process
        await new Promise(resolve => setTimeout(resolve, 1500));

        // Get all clients to show sync complete message
        const clients = await self.clients.matchAll();
        clients.forEach(client => {
          client.postMessage({
            type: 'sync-complete',
            data: localStorage.getItem('syncData') || 'No data'
          });
        });

        console.log('[SW] Background sync completed');
      })()
    );
  }
});

// Push Event
self.addEventListener('push', event => {
  const data = event.data ? event.data.json() : {};

  if (data.method === 'pushMessage') {
    const title = data.title || '🛍️ SneakCart Update';
    const options = {
      body: data.message || 'New deals available!',
      icon: '/icons/icon-192.png',
      badge: '/icons/icon-192.png'
    };
```

```
  event.waitUntil(
    self.registration.showNotification(title, options)
  );
 }
});

// Message Event (for communication from page)
self.addEventListener('message', event => {
 if (event.data && event.data.method === 'pushMessage') {
  self.registration.showNotification(
    event.data.title || '🛒 SneakCart',
    {
      body: event.data.message,
      icon: '/icons/icon-192.png'
    }
  );
 }
});
```

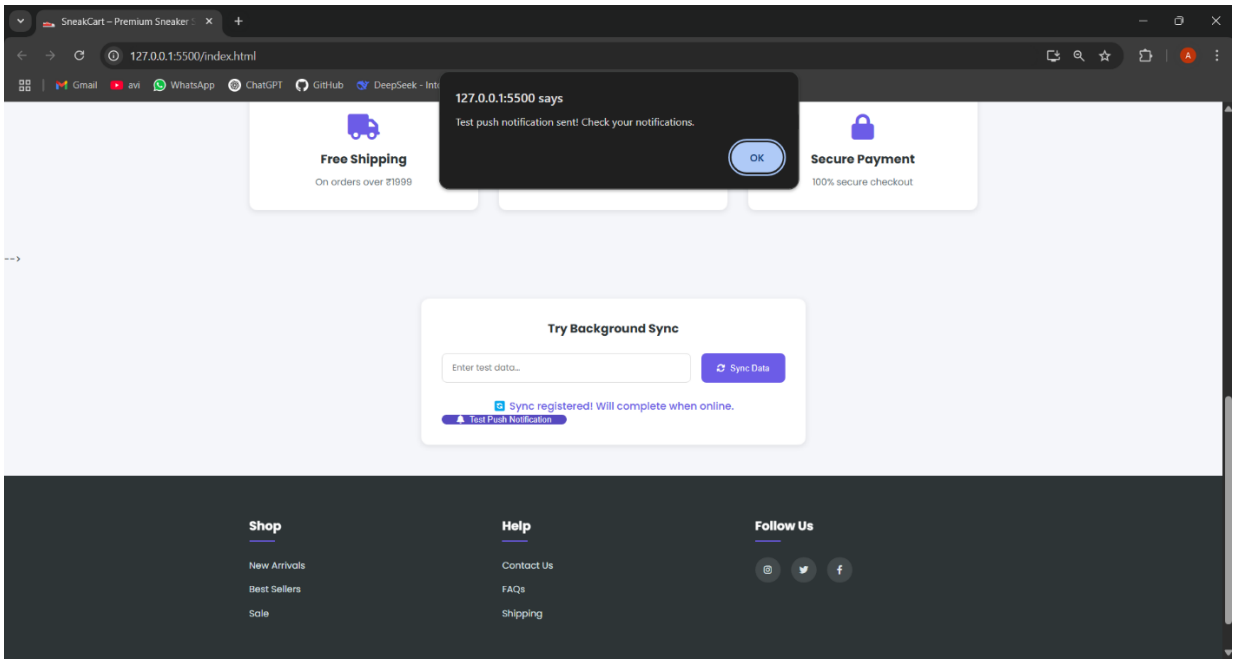
Script.js:

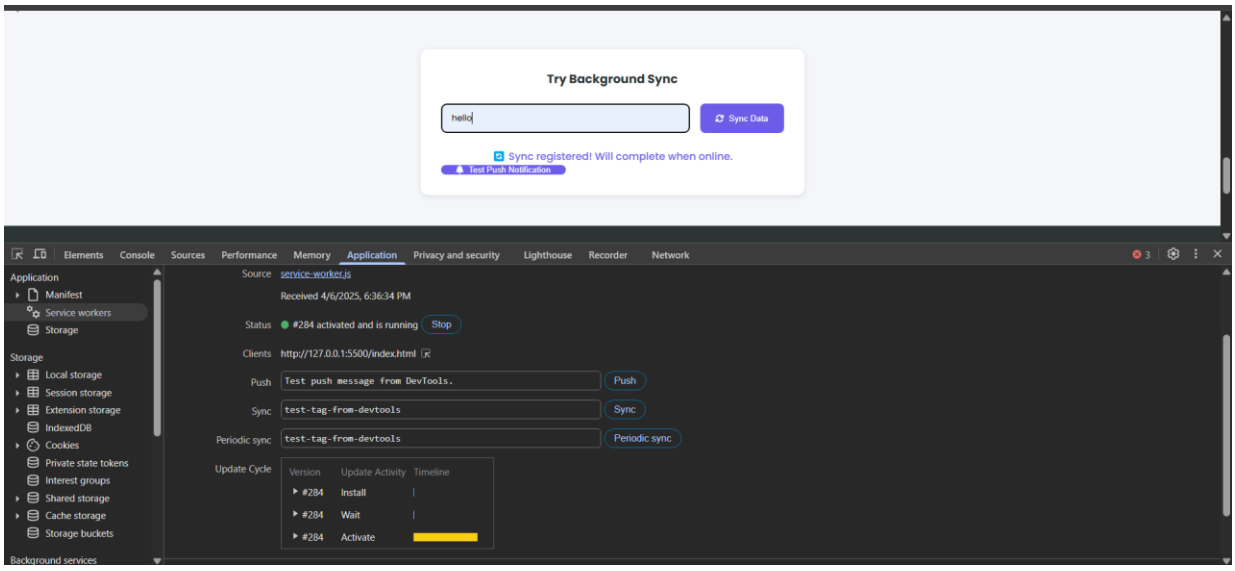
```
// Service Worker Registration
if ('serviceWorker' in navigator) {
 window.addEventListener('load', () => {
  navigator.serviceWorker.register('service-worker.js')
    .then(reg => {
      console.log('✅ Service Worker registered!', reg.scope);

      // Set up sync after SW registration
      setupBackgroundSync(reg);
      setupPushDemo();
    })
    .catch(err => {
      console.error('❌ Service Worker registration failed:', err);
    });
 });
}

// Background Sync Setup
function setupBackgroundSync(swReg) {
 const form = document.getElementById('dummyForm');
 if (!form) return;

 form.addEventListener('submit', function(e) {
```

```
    e.preventDefault();
    const input = document.getElementById('dummyInput');
    const status = document.getElementById('status');

    // Store data in localStorage for demo
    localStorage.setItem('syncData', input.value);

    // Register sync
    swReg.sync.register('sync-form')
      .then(() => {
        status.textContent = '🔄 Sync registered! Will complete when online.';
        status.style.color = 'var(--primary)';
        input.value = '';
      })
      .catch(err => {
        status.textContent = '❌ Sync registration failed';
        status.style.color = 'var(--danger)';
        console.error('Sync registration failed:', err);
      });
  });
}

// Push Notification Demo
function setupPushDemo() {
  const pushButton = document.createElement('button');
  pushButton.className = 'sync-button';
  pushButton.innerHTML = '<i class="fas fa-bell"></i> Test Push Notification';
  pushButton.onclick = triggerTestPush;

  const syncSection = document.querySelector('.sync-section');
  if (syncSection) {
    syncSection.appendChild(pushButton);
  }
}

function triggerTestPush() {
  navigator.serviceWorker.ready
    .then(reg => {
      reg.active.postMessage({
        method: "pushMessage",
        message: "🎉 New deal on Sneakers! 50% OFF today only!"
      });
      alert('Test push notification sent! Check your notifications.');
    })
```

```
    .catch(err => {
      console.error('Push test failed:', err);
    });
}
```

Output:





Name: Atif Ansari Roll no: 04 Class: D15B

MPL Experiment - 10

Aim: To study and implement the deployment of an Ecommerce Progressive Web App (PWA) using GitHub Pages.

Theory: GitHub Pages is a free hosting service provided by GitHub for deploying static websites directly from a repository. It simplifies website deployment by enabling users to publish web pages by pushing updates to a specific branch. It supports Jekyll, custom domains, and automatic page generation.

Advantages of GitHub Pages:

1. Free and easy to set up.
2. Works seamlessly with GitHub repositories.
3. Supports Jekyll for blogging.
4. Allows custom domain integration.

Limitation of GitHub Pages:

1. The repository must be public unless using a paid private plan.
2. Limited plugin support for Jekyll.
3. Custom domains lack built-in HTTPS support.

Alternative - Firebase Hosting

Firebase, a cloud service by Google, offers a real-time backend solution for web applications. It supports authentication, database storage, and hosting services. Firebase provides an HTTPS-secured environment, making it ideal

for hosting dynamic applications.

Advantages of Firebase Hosting:

1. Built-in HTTPS and security features.
2. Real-time database for dynamic applications.
3. Scalable cloud-based infrastructure.

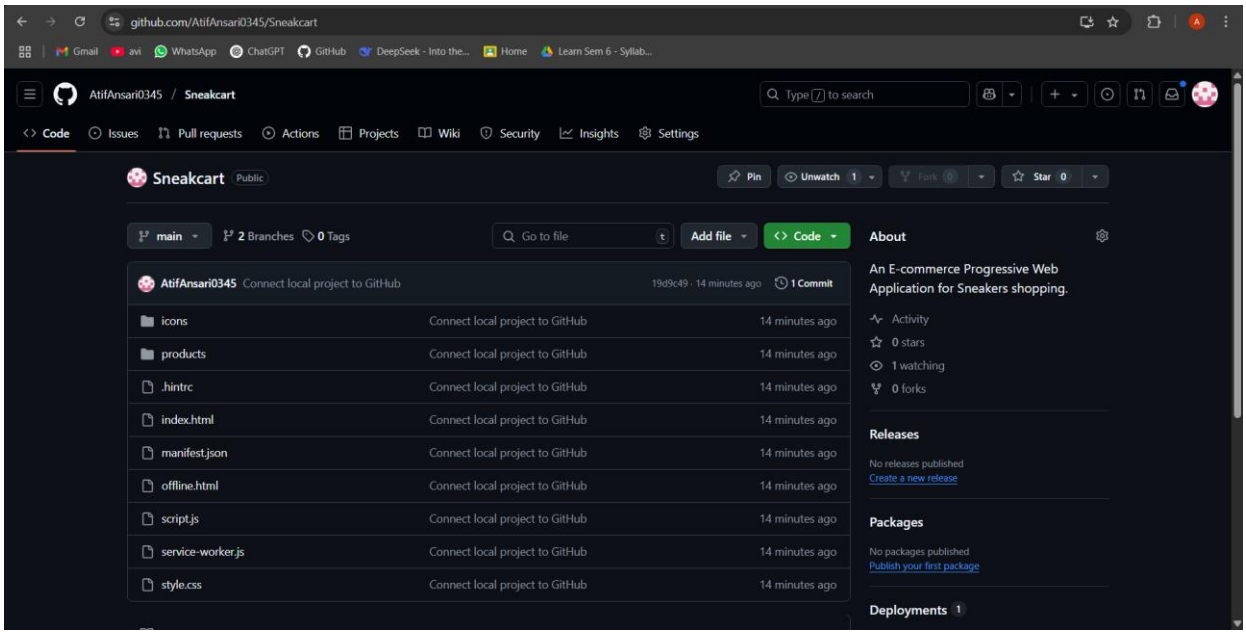
Limitations of Firebase Hosting:

1. Limited free-tier data transfer (10 GB per month)
2. Requires a command-line interface (CLI) for deployment.
3. No built-in static site generator support.

Conclusion:

In this experiment, we successfully deployed an Ecommerce PWA using GitHub Pages. We explored the advantages and limitations of GitHub Pages and Firebase Hosting. GitHub Pages provides a simple, free, and Git-integrated method for hosting static web applications, making it ideal for projects that do not require a backend. However, for more advanced applications requiring real-time data handling and authentication, Firebase Hosting is a more suitable alternative.

Implementation of Experiment 10:



Run command:

npm init -y

Package.json:

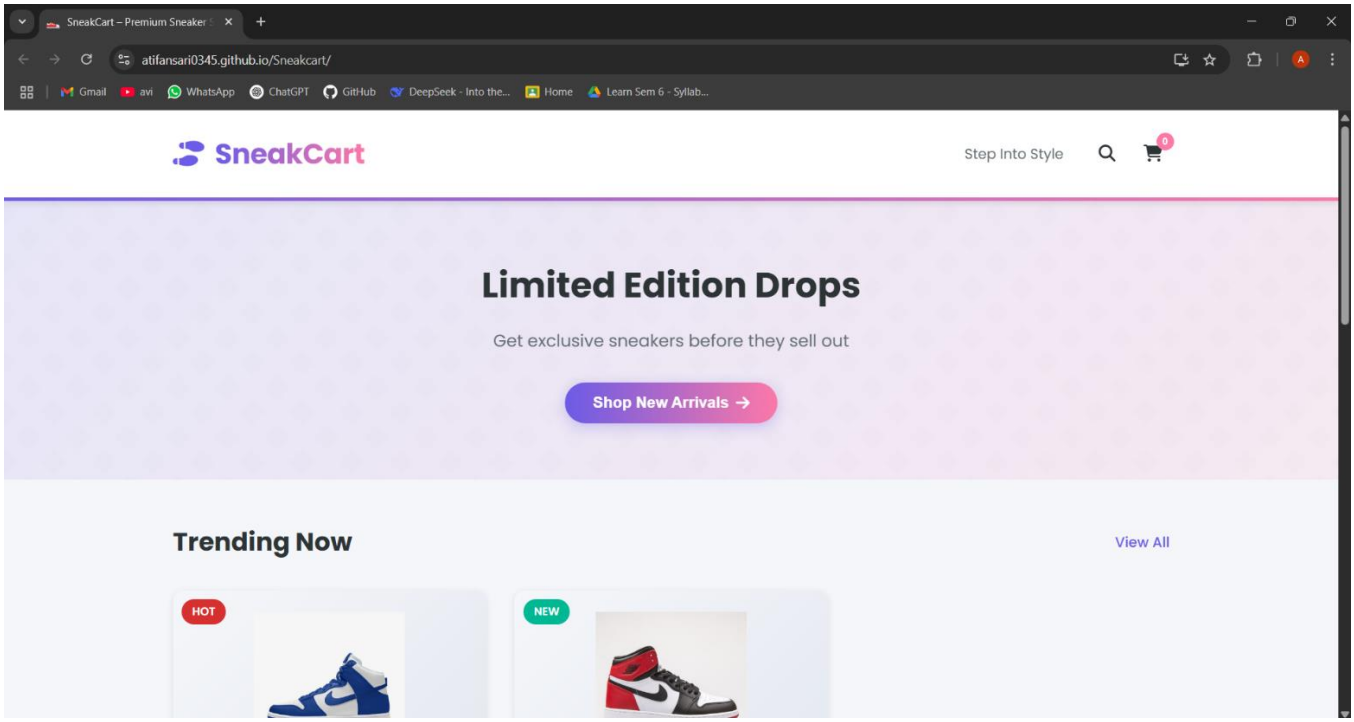
```
{
  "name": "sneakcart",
  "version": "1.0.0",
  "description": "",
  "main": "script.js",
  "scripts": {
    "predeploy": "echo 'Preparing deployment...'",
    "deploy": "gh-pages -d .",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/AtifAnsari0345/Sneakcart.git"
```

```
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/AtifAnsari0345/Sneakcart/issues"
  },
  "homepage": "https://AtifAnsari0345.github.io/Sneakcart",
  "devDependencies": {
    "gh-pages": "^6.3.0"
  }
}
```

Run commands:

npm install gh-pages --save-dev

npm run deploy



Name: Atif Ansari Roll no: 04 Class: D15B

MPL Experiment - 11

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 analyze and audit web applications. It provides detailed reports on various performance aspects of a website, including speed, accessibility, and adherence to Progressive Web App (PWA) standards.

Key Features and Audit Metrics:

Google Lighthouse evaluates web applications based on multiple factors:

1. Performance: Measures the loading speed, rendering times, and how quickly a website becomes interactive. A high performance score indicates an optimized and fast-loading website.
2. PWA Score: Assesses whether the web application follows Google's PWA guidelines, including Service Worker implementation, offline capabilities, and mobile responsiveness.
3. Accessibility: Ensures that the website is user-friendly for all users, including those with disabilities, by analyzing ARIA attributes, semantic HTML ~~pages~~ usage, and screen-reader compatibility.
4. Best Practices: Checks for security implementations like HTTPS usage, modern JavaScript practices, and user-friendly experiences such as clear

cookie alerts and password protection measures.

To enhance PWA functionality, modifications were made to the manifest.json file and index.html including:

- Adding a theme color using <meta name = "theme-color" content = "#4285f4">
- Implementing a maskable icon in the manifest file.
- Adding an Apple Touch icon for better compatibility.

Conclusion:

In this experiment, we successfully utilized Google Lighthouse to analyze the performance and PWA compatibility of our web application. The tool provided insights into various aspects such as loading speed, accessibility compliance, and adherence to best practices. By making necessary adjustments to the manifest file and implementing features like theme color and maskable icons, we improved the applications overall PWA score.

The experiment demonstrated the importance of auditing a web app's performance and user experience using Google Lighthouse. By addressing identified issues, developers can create web applications that are faster, more accessible, and better optimized for mobile and offline use. This ensures a higher level of user satisfaction and compliance with modern web standards.

Implementation of Experiment 11:

Service-worker.js

```
const CACHE_NAME = "sneakcart-v2";
const ASSETS_TO_CACHE = [
 "/Sneakcart/",
 "/Sneakcart/index.html",
 "/Sneakcart/style.css",
 "/Sneakcart/script.js",
 "/Sneakcart/manifest.json",
 "/Sneakcart/icons/icon-192.png",
 "/Sneakcart/icons/icon-512.png",
 "/Sneakcart/products/shoe1.png",
 "/Sneakcart/products/shoe2.jpg",
 "/Sneakcart/products/shoe3.jpg", // ➕ Add any additional assets here
 "https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.0/css/all.min.css",
 "https://fonts.googleapis.com/css2?family=Poppins:wght@300;400;500;600;700&display=s
wap"
];

// ✅ Install
self.addEventListener("install", (event) => {
 console.log("📥 Service Worker installing...");
 event.waitUntil(
  caches.open(CACHE_NAME).then((cache) => {
   console.log("✅ Caching app shell and assets");
   return cache.addAll(ASSETS_TO_CACHE);
  })
 );
 self.skipWaiting(); // Activate worker immediately
});

// ✅ Activate
self.addEventListener("activate", (event) => {
 console.log("🔄 Service Worker activating...");
 event.waitUntil(
  caches.keys().then((keyList) => {
   return Promise.all(
    keyList.map((key) => {
     if (key !== CACHE_NAME) {
```

```
          console.log("🗑 Removing old cache:", key);
          return caches.delete(key);
        }
      })
    );
  })
 );
 self.clients.claim(); // Take control immediately
});

// ✅ Fetch (Offline-first)
self.addEventListener("fetch", (event) => {
 event.respondWith(
  caches.match(event.request).then((cachedResponse) => {
   return (
    cachedResponse ||
    fetch(event.request).catch(() => {
     // Offline fallback for navigation requests
     if (event.request.mode === "navigate") {
      return caches.match("/Sneakcart/index.html");
     }
    })
   );
  })
 );
});

// ✅ Background Sync
self.addEventListener("sync", (event) => {
 if (event.tag === "sync-dummy-form") {
  event.waitUntil(
   // Simulate form re-submission or API retry
   new Promise((resolve) => {
    console.log("🔁 Background Sync triggered!");
    setTimeout(resolve, 2000);
   })
  );
 }
});
```

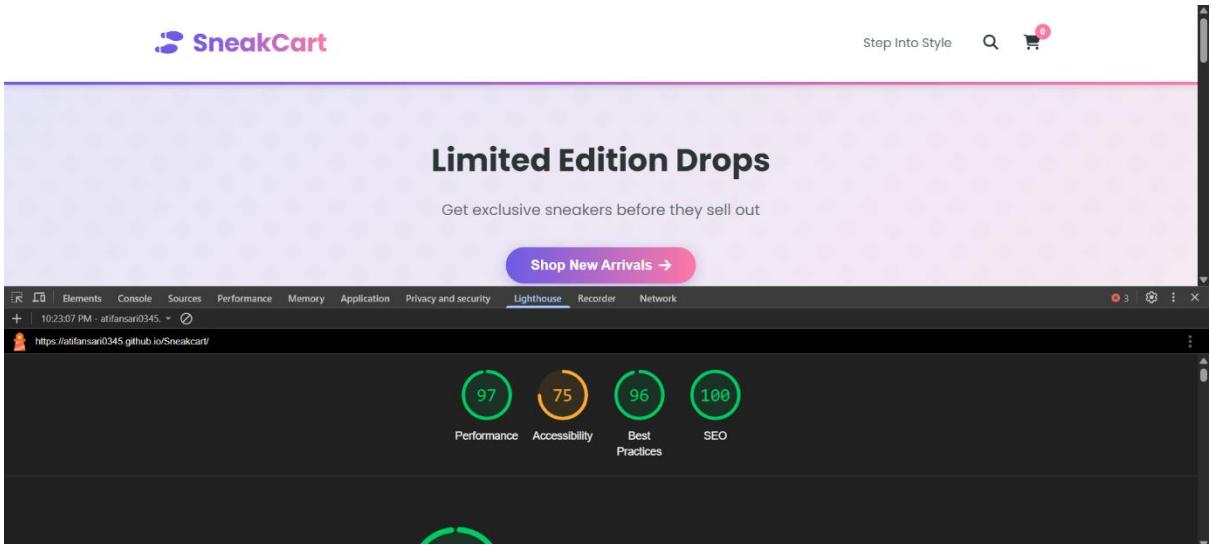
```
// ✅ Push Notifications (demo setup)
self.addEventListener("push", (event) => {
  const data = event.data ? event.data.text() : "🔥 Hot new drops just landed!";
  event.waitUntil(
    self.registration.showNotification("👟 SneakCart", {
      body: data,
      icon: "/Sneakcart/icons/icon-192.png",
      badge: "/Sneakcart/icons/icon-192.png"
    })
  );
});

self.addEventListener('install', (event) => {
  console.log("📦 Service Worker installing...");
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(cache => {
        return cache.addAll(assets)
          .catch(err => {
            console.error("❌ Caching failed:", err);
          });
      })
  );
});
```

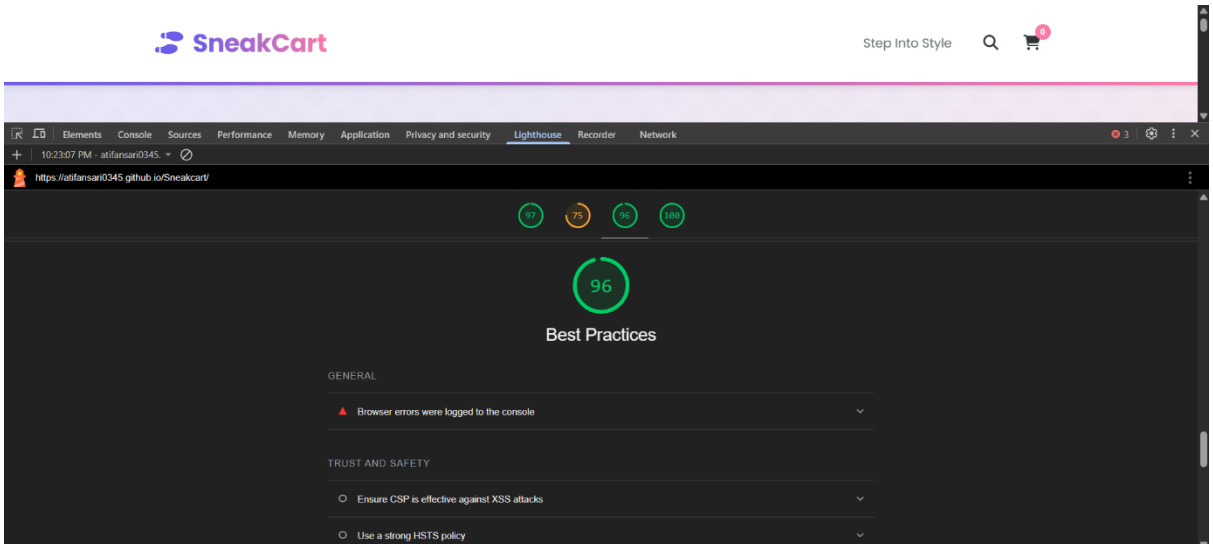
Manifest.json:

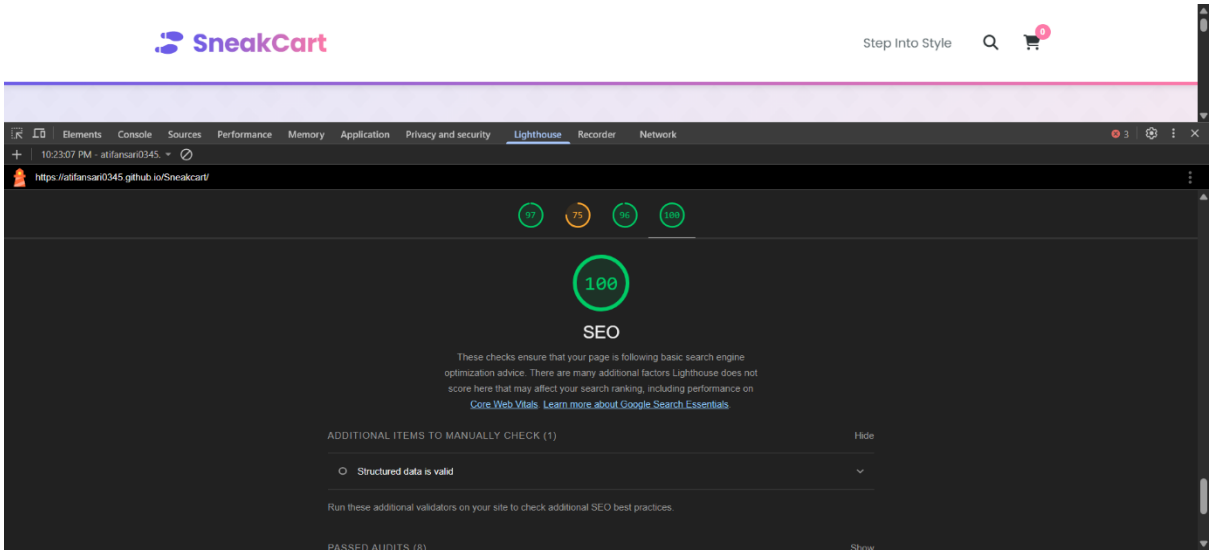
```
{
  "name": "SneakCart – Sneaker Store",
  "short_name": "SneakCart",
  "start_url": "/Sneakcart/index.html",
  "scope": "/Sneakcart/",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#000000",
  "orientation": "portrait-primary",
  "description": "Discover and shop limited edition premium sneakers. Built as a Progressive Web App (PWA) for the ultimate experience.",
  "icons": [
    {
```

```
    "src": "/Sneakcart/icons/icon-192.png",
    "sizes": "192x192",
    "type": "image/png",
    "purpose": "any maskable"
   },
   {
    "src": "/Sneakcart/icons/icon-512.png",
    "sizes": "512x512",
    "type": "image/png",
    "purpose": "any maskable"
   }
  ],
  "id": "/Sneakcart/",
  "categories": ["shopping", "fashion", "sneakers", "lifestyle"],
  "screenshots": [
   {
    "src": "/Sneakcart/screenshots/home.png",
    "sizes": "1080x1920",
    "type": "image/png",
    "label": "Homepage"
   },
   {
    "src": "/Sneakcart/screenshots/product.png",
    "sizes": "1080x1920",
    "type": "image/png",
    "label": "Product Page"
   }
  ],
  "lang": "en"
}
```

Output:


SneakCart
Step Into Style
Limited Edition Drops
Get exclusive sneakers before they sell out
Shop New Arrivals →
Elements Console Sources Performance Memory Application Privacy and security Lighthouse Recorder Network
https://atifansari0345.github.io/Sneakcart/
97 Performance
75 Accessibility
96 Best Practices
100 SEO


SneakCart
Step Into Style
Elements Console Sources Performance Memory Application Privacy and security Lighthouse Recorder Network
https://atifansari0345.github.io/Sneakcart/
96
Best Practices
GENERAL
Browser errors were logged to the console
TRUST AND SAFETY
Ensure CSP is effective against XSS attacks
Use a strong HSTS policy


SneakCart
Step Into Style
Elements Console Sources Performance Memory Application Privacy and security Lighthouse Recorder Network
https://atifansari0345.github.io/Sneakcart/
100
SEO
These checks ensure that your page is following basic search engine optimization advice. There are many additional factors Lighthouse does not score here that may affect your search ranking, including performance on Core Web Vitals. Learn more about Google Search Essentials.
ADDITIONAL ITEMS TO MANUALLY CHECK (1)
Hide
Structured data is valid
Run these additional validators on your site to check additional SEO best practices.

Name: Atif Ansari Roll no: 04 Class: D15B

MPL Assignment - 1

Q. 1] a) Explain the key features and advantages of using Flutter for mobile app development.

→ Flutter is an open-source UI software development toolkit created by Google for building natively compiled applications for mobile, web, and desktop from a single codebase.

Key features of Flutter:

1. Single Codebase: Developers can write one codebase for both Android and iOS, reducing development time.
2. Fast Performance: Uses the Dart programming language, which compiles to native ARM code for high performance.
3. Hot Reload: Instantly reflects changes in the UI without restarting the application, enhancing developer productivity.
4. Rich UI components: Provides a vast collection of customizable widgets for creating beautiful UIs.
5. Support for Web and Desktop Apps: Extends beyond mobile to support web and desktop applications.
6. Access to Device Features: Allows easy integration with device-specific APIs such as camera, GPS, and sensors.
7. Growing Community and Google Support: Actively maintained by Google and has a strong developer community.

Advantages of Using Flutter:

- **Cost and Time Efficiency**: Reduces development time and costs by allowing the same codebase for multiple platforms.
- **Faster Development**: The hot reload feature significantly speeds up the development process.
- **Open Source and Free**: Completely free to use, with extensive documentation and community support.
- **High Performance**: Since it compiles to native ARM code, it performs better than traditional cross-platform solutions like React Native.

b) Discuss how Flutter framework differs from traditional approaches and why it has gained popularity in the developer community.

→ Flutter differs significantly from traditional mobile development approaches, primarily in how it renders UI, handles performance, and supports cross-platform development.

Differences Between Flutter and Traditional Approaches:

Feature	Traditional Development	Flutter
Codebase	Separate codebases for Android and iOS.	Single codebase for both Android and iOS.

UI Rendering	Uses native UI components of the respective platform	Uses Skia rendering engine to draw UI independently
Performance	Native performance but requires separate implementations	Near-native performance due to direct compilation to machine code
Development Speed	Slower due to separate development for platforms	Faster due to Hot Reload and shared UI code
Apps size	Smaller due to native UI components.	Slightly larger due to built-in rendering engine.

Reasons for Flutter's Popularity:

1. Cross-Platform Efficiency: Enables faster ~~deve~~ development with a single codebase for multiple platforms.
2. Hot Reload for Instant Updates: Allows developers to see changes instantly, making debugging and UI design easier.
3. Modern UI Capabilities: Comes with a rich set of widgets and animations, allowing developers to create visually appealing applications.
4. Strong Google Support & Growing Ecosystem: Backend

by Google and used in popular apps like Google Ads, Alibaba, and BMW, attracting more developers.

4. Expanding Beyond Mobile: With Flutter Web and Desktop, developers can build applications beyond just mobile platforms, making it more versatile.

Q.2] (a) Describe the concept of the widget tree in Flutter. Explain how widget composition is used to build complex user interfaces.

→ In flutter, everything is a widget. A widget tree is the hierarchical structure of widgets that define the UI of a flutter application. Each widget in the tree is responsible for rendering a specific part of the UI. The tree starts from a root widget and branches into multiple child widgets.

Widgets Composition in Building Complex UIs:
Flutter follows a composition-based approach, meaning complex UIs are built by nesting smaller, reusable widgets. Instead of extending a single UI class, Flutter encourages combining multiple widgets to create sophisticated designs. for example, a login screen may consist of:

- A Column Widget for Vertical alignment.
- TextField widgets for user input.
- A Button widget for submission.
- A Padding widget for spacing.

By composing simple widgets together, developers can build scalable and maintainable UIs.

b) Provide examples of commonly used widgets and their roles in creating a widget tree.

→

Widget	Role in Widget Tree
MaterialApp	Root widget for a Material Design app.
Scaffold	Provides app structure with AppBar, body, and FloatingActionButton.
Container	A versatile widget for styling.
Row / Column	Layout widgets for horizontal and vertical alignment.
Text	Displays text content.
Image	Displays images from assets, network, or memory.
ListView	Scrollable list of widgets.
TextField	User input field.
ElevatedButton	A Material-style button with elevation

These widgets form a hierarchical structure, where parent widgets contain and manage child widgets, defining how the UI looks and behaves.

Q.3] a) Discuss the importance of state management in flutter applications.

→ State management is crucial in flutter because it determines how ~~much~~ UI components react to changes in data. Since flutter's UI is built using widgets, the UI needs to be updated dynamically whenever the application's state changes.

Why State management is Important?
- It ensures the UI remains responsive to changes.
- Helps maintain data consistency across screens.
- Reduces unnecessary widget rebuilds, improving performance.
- Makes application logic cleaner and more maintainable.

There are two types of states in flutter:
1. Ephemeral State: Affects only a single widget.
2. App-wide state: Shared across multiple widgets.

b) Compare and contrast different state management approaches in flutter.

→

Approach	Description	Best use Cases
setState	Built-in method to update local state within a StatefulWidget.	Small UI updates within a single widget.
Provider	A dependency injection framework that allows state sharing across the widget tree.	Medium-sized apps where state needs to be accessed by multiple widgets.
Riverpod	An improved version of Provider with better performance and testability.	Large-scale apps requiring better dependency management and testability

When to Use Each Approach:

- setState → When managing UI state in a single widget.
- Provider → When managing state across multiple widgets but keeping the app simple.
- Riverpod → When the app has complex business logic and requires better state handling.

Q.4] a) Explain the process of integrating Firebase with a Flutter application. Discuss the benefits of using Firebase as a backend solution.

→ Steps to Integrate Firebase with Flutter:

1. Create a Firebase Project:
   - Go to: Firebase Console.
   - Click on "Create a Project" and follow the setup.

2. Add Firebase to Flutter App:
   - Register the app for Android or iOS.
   - Add Firebase dependencies in pubspec.yaml.
   - Run flutterfire configure to link Firebase to the project.

3. Initialize Firebase in Flutter:
   - Import and initialize Firebase in main.dart file:

```
void main() async {
  WidgetFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}
```

4. Use Firebase Services:
   - Implement Firebase Authentication, Firestore database, or cloud storage as needed.

Benefits of Using Firebase as a Backend Solution:

- Real-time Database & Firestore: Syncs data across devices in real-time.
- Scalability: Supports both small and large-scale applications.
- Authentication: Provides secure user authentication with Google, Facebook, etc.
- Cloud Functions: Runs server-side logic without

managing backend infrastructure.

- **Hosting & Storage**: offers hosting for web apps and cloud storage for media files.

b) Highlight the Firebase services commonly used in Flutter development and provide a brief overview of how data synchronization is achieved.

→ **Common Firebase Services in Flutter:**

Service	Functionality
Firebase Authentication	Handles user authentication.
Cloud Firestore	NoSQL database for real-time data storage and sync.
Realtime database	JSON-based database that updates ~~inte~~ instantly.
Cloud Storage	Stores and serves user-generated content like images and videos.
Firebase Cloud Messaging	Sends push notifications to users.
Firebase Analytics	Tracks user behavior in the app.
Cloud Functions	Executes backend logic in response to events.

Data Synchronization in Firebase:

- Firestore & Realtime Database: sync data in real time across devices.
- Offline Support: Data is cached locally when offline and syncs when reconnected.
- Automatic Conflict Resolution: Ensures consistency by managing data conflicts during synchronization.

Conclusion: Flutter's widget-based structure enables efficient UI design, while state management ensures smooth interactions. Firebase enhances app functionality by providing real-time data sync, authentication, and backend services, making it a powerful backend solution for modern applications.

Name: Atif Ansari Roll no: 04 Class: D15B

MPL Assignment - 2

05/05

1] Define Progressive Web App (PWA) and explain its significance in modern web development. Discuss the key characteristics that differentiate PWAs from traditional mobile apps.

→ **Definition of Progressive Web App (PWA):**

A Progressive Web App (PWA) is a type of web application that combines the best features of web and mobile apps. PWAs offer a native app-like experience using modern web technologies while remaining accessible via a web browser.

Significance of PWAs in Modern Web Development:

- **Cross-Platform Compatibility:** Works on any device with a modern web browser.
- **Improved Performance:** Uses caching and ~~backgrouds~~ background sync for fast loading.
- **Offline Functionality:** Service Workers allow access to content even when offline.
- **Better User Engagement:** Supports push notifications and home screen installation.
- **Cost-Effective:** Reduces the need for seperate development for Android and iOS.

Key Characteristics that Differentiate PWAs from Traditional Mobile Apps:

Feature	PWA	Traditional Mo App
Installation	No need to download from an app store; can be added to home screen	Requires installati via Google Play St or App Store
Offline Support	Works offline using Service Workers	Requires explicit offline functionality
Performance	Fast loading using caching techniques	Depends on native platform optimizatio
Updates	Automatic updates through the web	Requires manual updates from stor
Device Access	Limited access to device features	Full access to native device featur
Platform Dependency	Runs on any browser across platforms	Seperate development needed for Android and iOS.

Q.2] Define responsive web design and explain its importance in the context of Progressive Web Apps. Compare and contrast responsive, fluid, and adaptive web design approache

Definition of Responsive Web Design (RWD):

Responsive Web Design (RWD) is a web development approach that ensures a website adapts to different screen sizes and devices using flexible layouts, CSS media queries, and scalable images.

Importance of RWD in PWAs:

- Ensures PWAs look good on all devices, from desktops to mobile phones.
- Enhances user experience by providing seamless interaction across screen sizes.
- Improves SEO rankings, as search engines favor mobile-friendly sites.
- Reduces development effort, as a single codebase works across all devices.

Comparison of Responsive, Fluid, and Adaptive Web Design:

Approach	Definition	Key Features	Best use case
Responsive	Uses media queries to adjust layout based on screen size	flexible grids, scalable images, CSS breakpoints	PWAs, modern websites needing cross-device support
Fluid	Uses percentage-based widths instead of fixed units	Adapts proportionally to screen size without breakpoints	Simple layouts that need smooth resizing

Adaptive	Uses predefined layouts for different screen sizes.	Detects device type and loads a specific design	Websites requiring precise control for different de

Q.3] Describe the lifecycle of Service Workers, includi registration, installation, and activation phases

→ A Service Worker is a JavaScript script that runs in the background, enabling features like offline support, caching, and push notifications in PWAs.

Service Worker Lifecycle:

1. Registration: The service worker is register in the JavaScript file, allowing the brow to recognize it.

- Example:

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js').the
    reg => console.log('Service Worker registered',
    .catch(err =>
      console.log('Service Worker registration
      failed', err));
}
```

2. Installation: The service worker is downloaded and installed in the background. Static ass can be pre-cached for offline use.

• Example:

```
self.addEventListener('install',
event => {
    event.waitUntil(
caches.open('pwa-cache-v1').then(
cache => {
        return cache.addAll([
            '/', '/index.html', '/styles.css',
            '/script.js'
        ]);
    })
    );
});
```

3. Activation: The service worker is activated and starts controlling pages. Old caches may be deleted to manage storage efficiently.

• Example:

```
self.addEventListener('activate',
event => {
    event.waitUntil(
        caches.key().then(keys => {
            return Promise.all(
                keys.filter(key => key !
== 'pwa-cache-v1').map(key =>
caches.delete(key))
    );
})
);
});
```

4. Fetch Event Handling (optional): The ser worker intercepts network requests and serve cached responses when offline.

- Example:

```
self.addEventListener('fetch',
event => {
    event.respondWith(
caches.match(event.request).then(
 response => {
      return response ||
 fetch(event.request);
  })
 );
});
```

Q.4) Explain the use of IndexedDB in the Servi Worker for Data storage.

→ IndexedDB is a low-level NoSQL databas built into modern browsers. It allows we applications, including PWAs, to store structured data efficiently for offline IndexedDB in Service Workers is used because of features like:

- Persistent Storage: Unlike localStorage, IndexedDB can storage large amounts of data.
- Asynchronous Operations: Uses Promises to handle data efficiently without blocking the main thread.

- **Indexed Storage**: Allows fast querying using keys and indexes.

Using IndexedDB in Service Workers:

1. Opening a Database:

```
let db;
const request = indexedDB.open('PWA_Database', 1);
request.onsuccess = event => {
    db = event.target.result;
};
request.onupgradeneeded = event => {
    let db = event.target.result;
    db.createObjectStore('users', { keyPath: 'id' });
};
```

2. Adding Data to IndexedDB:

```
function addUser(user) {
    let transaction = db.transaction(['user'], 'readwrite');
    let store = transaction.objectStore('users');
    store.add(user);
}
addUser({ id: 1, name: 'Atif', email: 'atif123@gmail.com' });
```

3. Retrieving Data from IndexedDB:

```
function getUser(id) {
    let transaction = db.transaction(['users'], 'readonly');
```

```
let store = transaction.objectStore('users');
let request = store.get(id);
request.onsuccess = () => {
    console.log(request.result);
};
}
getUser(1);
```

Data Synchronization in Service Workers:
When offline, data is stored in IndexedDB
When online, a background sync process sends stored data to the server.

- Example:

```
self.addEventListener('sync', event => {
  if (event.tag === 'sync-data') {
      event.waitUntil(getUser(1).then(user=
      return fetch('/sync', {
          method: 'POST',
          body: JSON.stringify(user),
          headers: {'Content-Type':
          'application/json'}
      });
    })
  );
}
});
```

IndexedDB in Service Workers enables efficient offline data storage and synchronization, ensuri seamless functionality and data persistence.