

P1

## MPL Experiment - 1

\* 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.

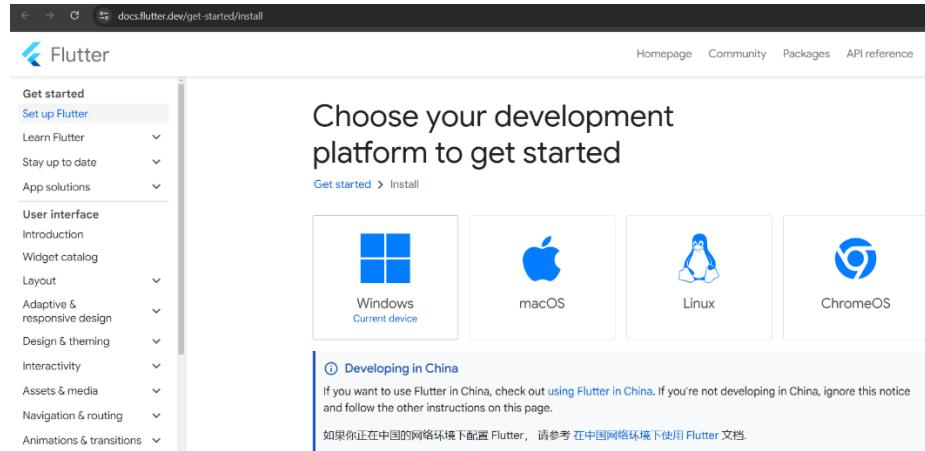
4. Configuring Environment Variables: Adding Flutter to the system PATH.

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

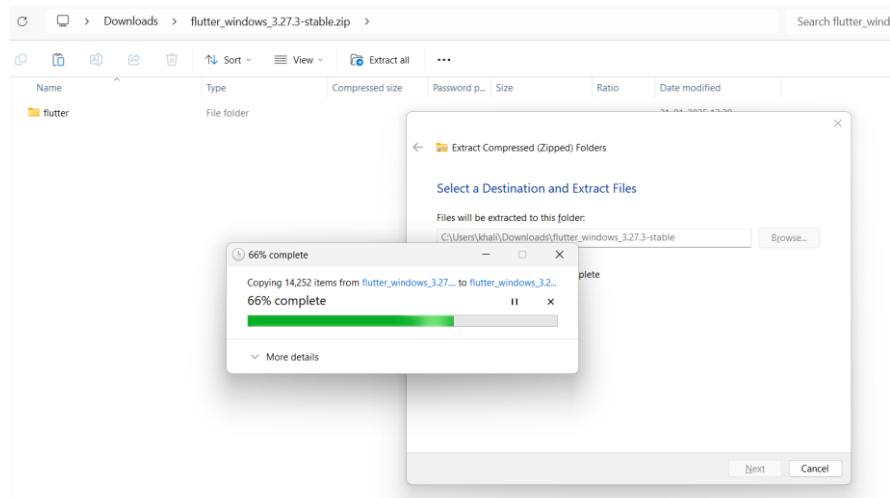
\* Conclusion: In this experiment, we successfully installed and configured the flutter development

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 build, test, and deploy cross-platform applications efficiently.

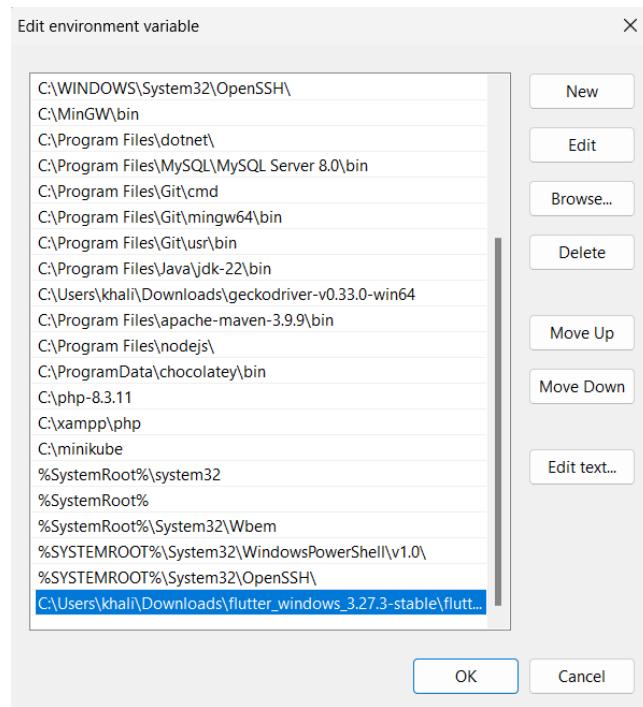
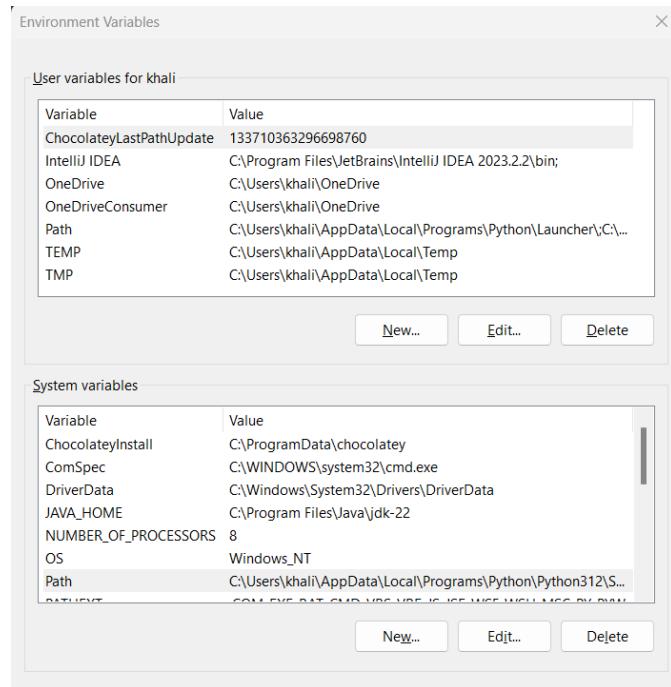
**Step 1: Download the installation bundle of the Flutter Software Development Kit for windows.**  
To download Flutter SDK, Go to its official website <https://docs.flutter.dev/get-started/install> , you will get the following screen.



**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.**

```

Command Prompt - flutter - f × + ▾
Microsoft Windows [Version 10.0.22631.4751]
(c) Microsoft Corporation. All rights reserved.

C:\Users\khali>flutter
Manage your Flutter app development.

Common commands:

  flutter create <output directory>
    Create a new Flutter project in the specified directory.

  flutter run [options]
    Run your Flutter application on an attached device or in an emulator.

Usage: flutter <command> [<arguments>]

Global options:
-h, --help           Print this usage information.
-v, --verbose        Noisy logging, including all shell commands executed.
If used with "--help", shows hidden options. If used with "flutter doctor", shows additional
diagnostic information. (Use "-vv" to force verbose logging in those cases.)
-d, --device-id     Target device id or name (prefixes allowed).
--version           Reports the version of this tool.
--enable-analytics Enable telemetry reporting each time a flutter or dart command runs.
--disable-analytics Disable telemetry reporting each time a flutter or dart command runs, until it is
re-enabled.
--suppress-analytics Suppress analytics reporting for the current CLI invocation.

Available commands:

```

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

```

Command Prompt - flutter d × + ▾
You have received two consent messages because the flutter tool is migrating to a new analytics system. Disabling
analytics collection will disable both the legacy and new analytics collection systems. You can disable analytics
reporting by running 'flutter --disable-analytics'

C:\Users\khali>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.27.3, on Microsoft Windows [Version 10.0.22631.4751], locale en-IN)
[✓] Windows Version (Installed version of Windows is version 10 or higher)
[✗] Android toolchain - develop for Android devices
  X Unable to locate Android SDK.
    Install Android Studio from: https://developer.android.com/studio/index.html
    On first launch it will assist you in installing the Android SDK components.
    (or visit https://flutter.dev/to/windows-android-setup for detailed instructions).
    If the Android SDK has been installed to a custom location, please use
    'flutter config --android-sdk' to update to that location.

[✓] Chrome - develop for the web
[!] Visual Studio - develop Windows apps (Visual Studio Build Tools 2019 16.11.40)
  X The current Visual Studio installation is incomplete.
    Please use Visual Studio Installer to complete the installation or reinstall Visual Studio.
[!] Android Studio (not installed)
[✓] VS Code (version 1.96.2)
[✓] Connected device (3 available)
[✓] Network resources

! Doctor found issues in 3 categories.

Command Prompt - flutter d × + ▾
AND WITHOUT REPRESENTATION OR WARRANTY OF ANY KIND.

10.8 Open Source Software. In the event Open Source software is included with Evaluation Software, such Open Source software
is licensed pursuant to the applicable Open Source software license agreement identified in the Open Source software
comments in the applicable source code file(s) and/or file header as indicated in the Evaluation Software. Additional
detail may be available (where applicable) in the accompanying on-line documentation. With respect to the Open Source
software, nothing in this Agreement limits any rights under, or grants rights that supersede, the terms of any applicable
Open Source software license agreement.

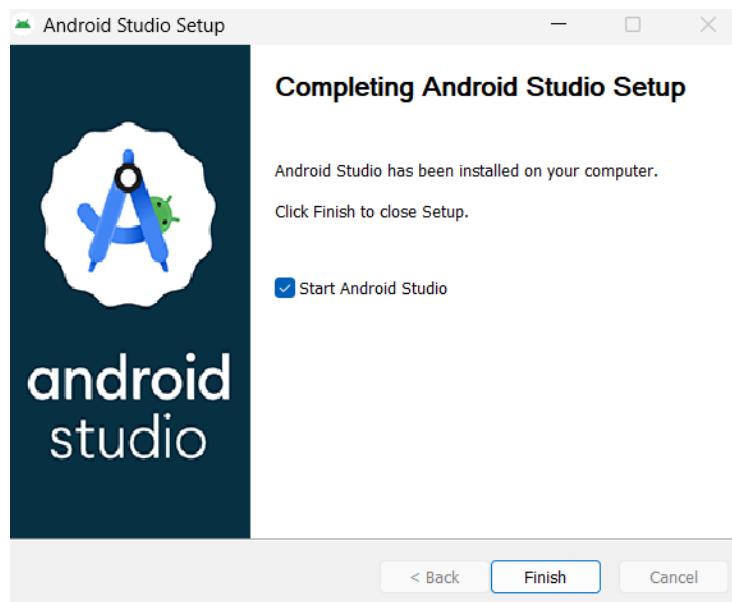
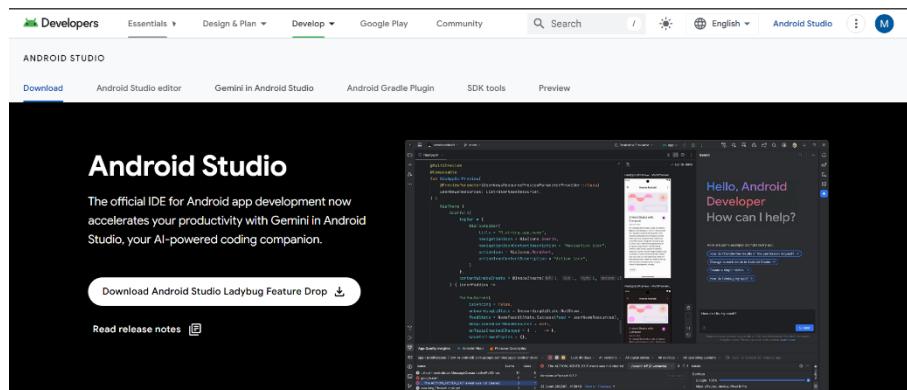
Accept? (y/N): y
All SDK package licenses accepted

C:\Users\khali\AppData\Local\Android\Sdk\ cmdline-tools\latest\bin>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.27.3, on Microsoft Windows [Version 10.0.22631.4751], locale en-IN)
[✓] Windows Version (Installed version of Windows is version 10 or higher)
[✓] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
[✓] Chrome - develop for the web
[!] Visual Studio - develop Windows apps (Visual Studio Build Tools 2019 16.11.40)
  X The current Visual Studio installation is incomplete.
    Please use Visual Studio Installer to complete the installation or reinstall Visual Studio.
[✓] Android Studio (version 2024.2)
[✓] VS Code (version 1.96.2)
[✓] Connected device (3 available)
[✓] Network resources

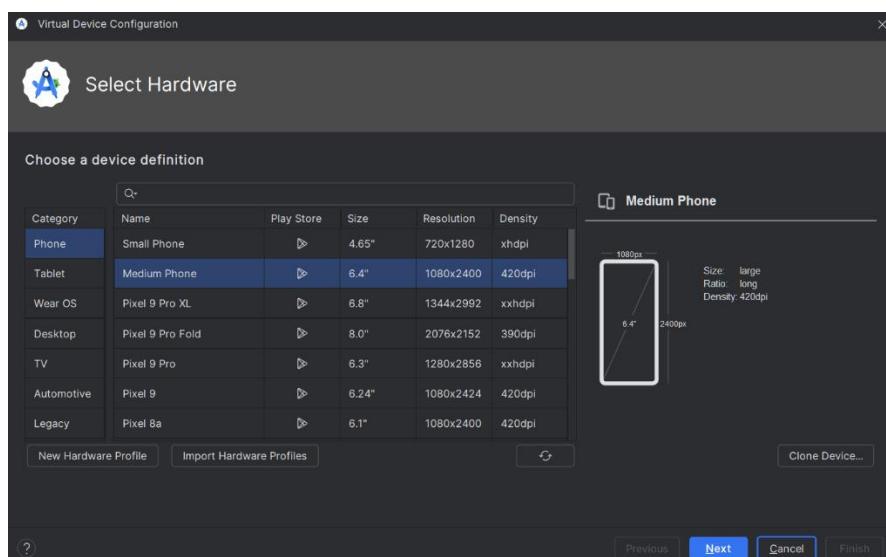
! Doctor found issues in 1 category.

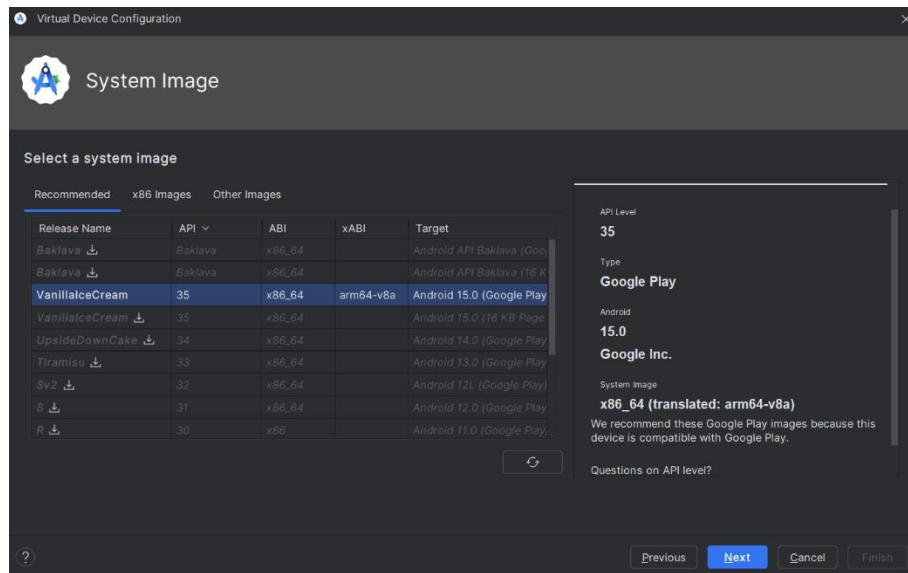
C:\Users\khali\AppData\Local\Android\Sdk\ cmdline-tools\latest\bin>
```

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

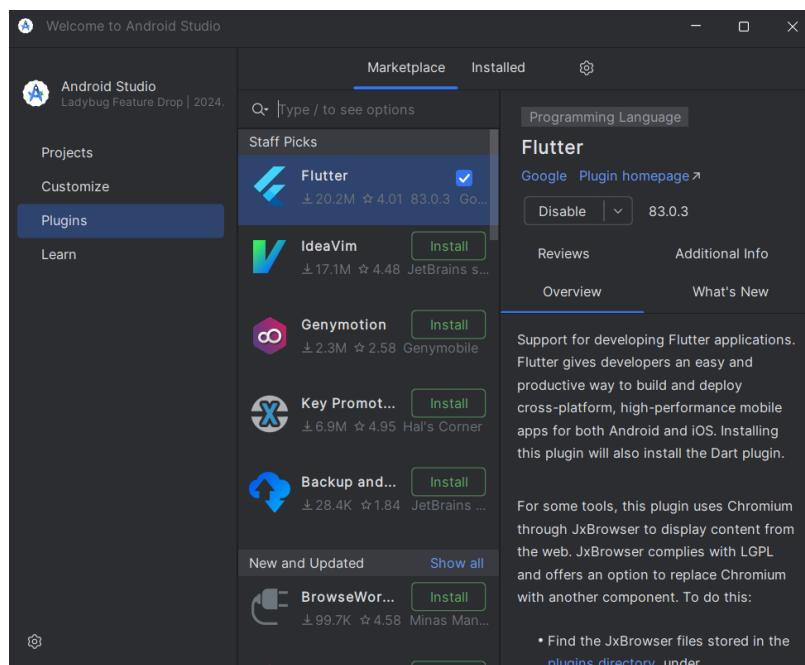


**Step 7: To set an Android emulator, go to Android Studio > Tools > Android > AVD Manager and select Create Virtual Device. Or, go to Help->Find Action->Type Emulator in the search box, verify the all AVD configuration. If it is correct, click on Finish.**

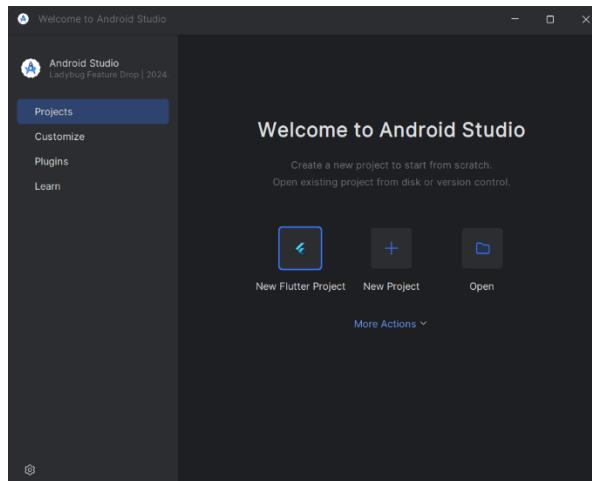




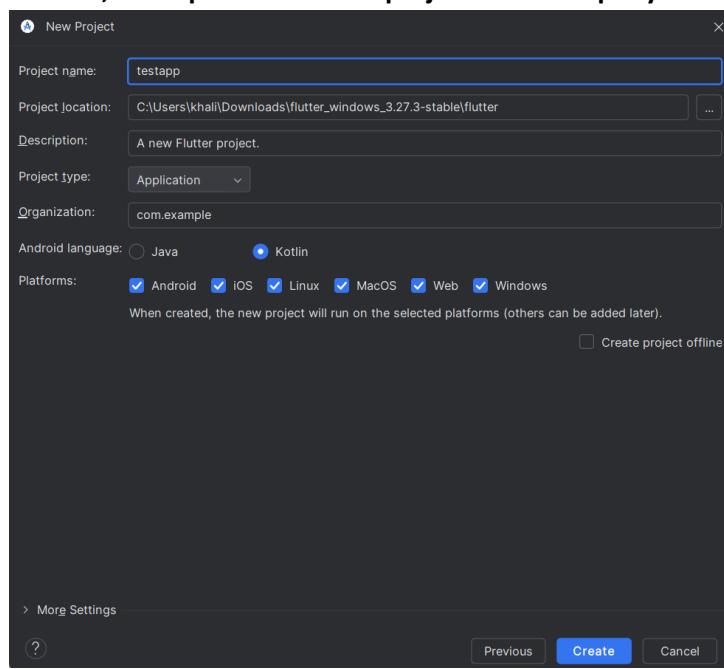
**Step 8: Now, install Flutter and Dart plugin for building Flutter application in Android Studio.**  
**These plugins provide a template to create a Flutter application, give an option to run and debug Flutter application in the Android Studio itself.**



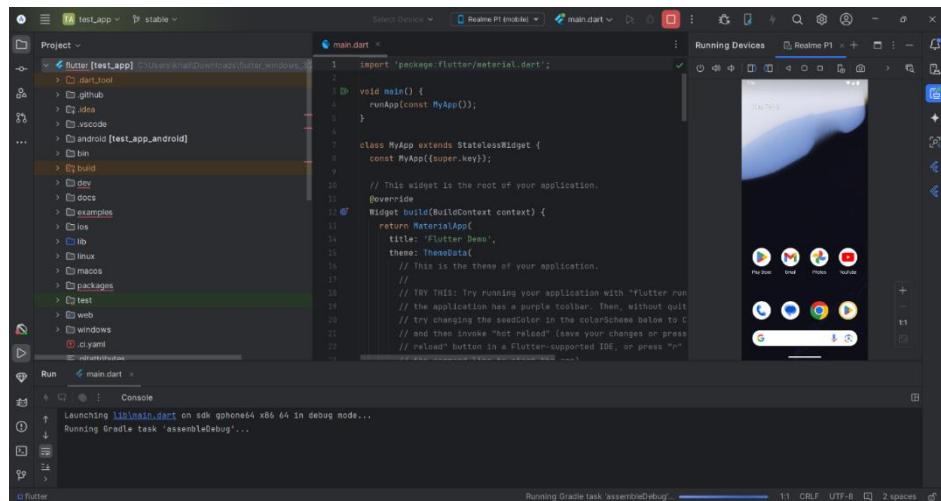
**Step 9: Now we can a new create Flutter Project.**



### 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.



## 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 SingleChildScrollView to ensure the UI was responsive and 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\nnVESIT",
                  onPressed: () {
                    print("You pressed Profile Button");
                  },
                ),
              ],
            ),
          ),
        ),
      ),
    );
  }
}

Widget _buildSectionHeader(String title) {
  return Container(
    padding: const EdgeInsets.all(10),
    child: Text(
      title,
      style: TextStyle(
        color: Colors.black,
        fontWeight: FontWeight.bold,
      ),
    ),
  );
}

Widget _buildProfileButton({
  required String label,
  required void Function() onPressed,
}) {
  return Container(
    padding: const EdgeInsets.all(10),
    child: Text(
      label,
      style: TextStyle(
        color: Colors.black,
        fontWeight: FontWeight.bold,
      ),
    ),
  );
}
```

```
),
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(

```

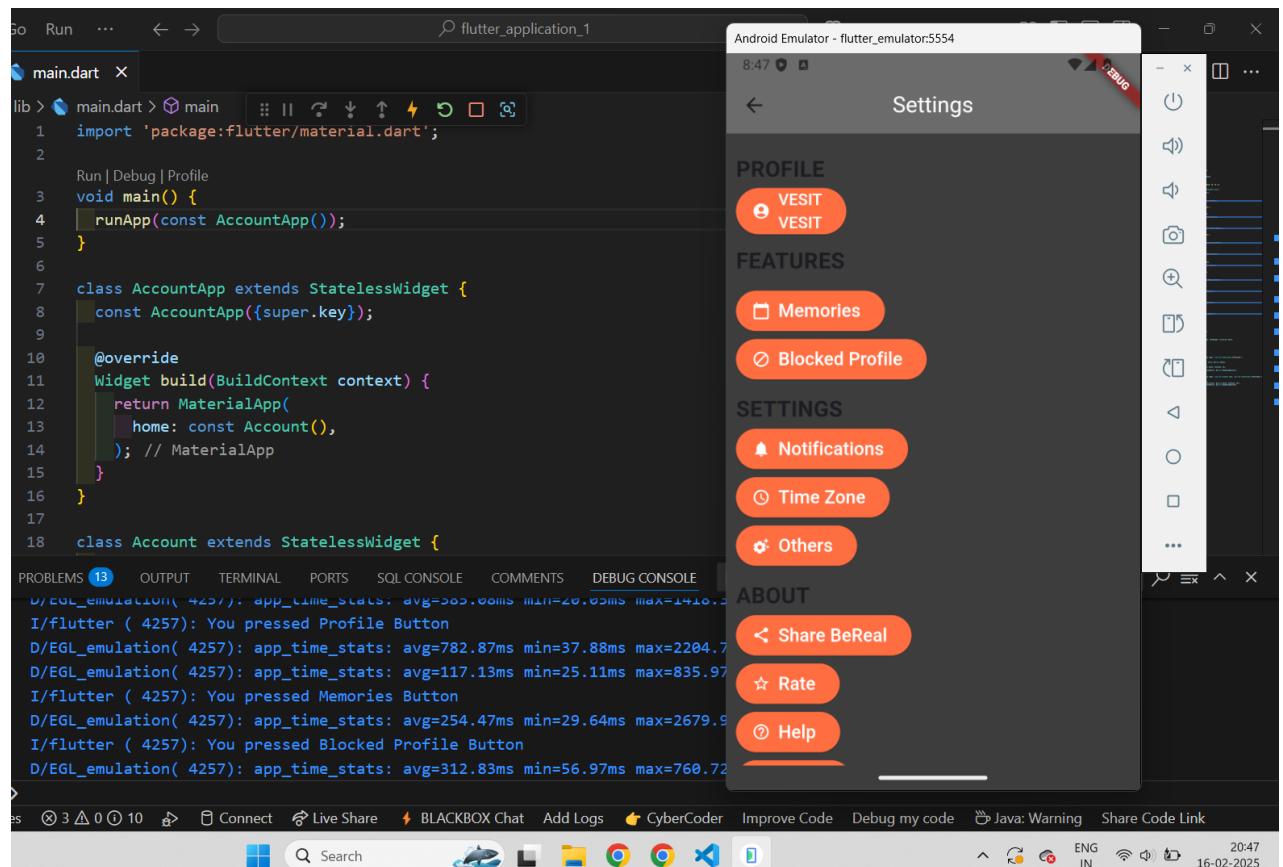
```

        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:



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, 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),
```

```
                ),
```

```
            ),
```

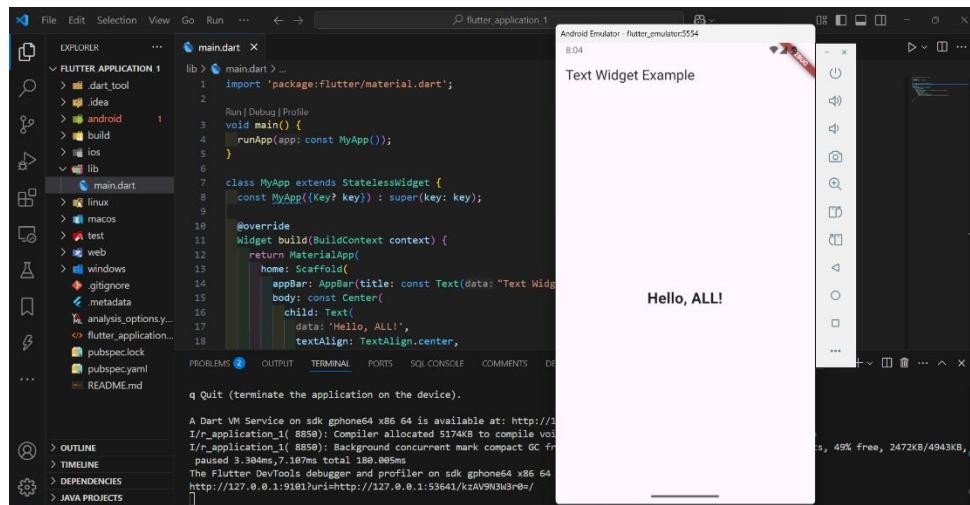
```
        ),
```

```
    );
```

```
}
```

```
}
```

Output:



## 2. Button Widgets Example

Code:

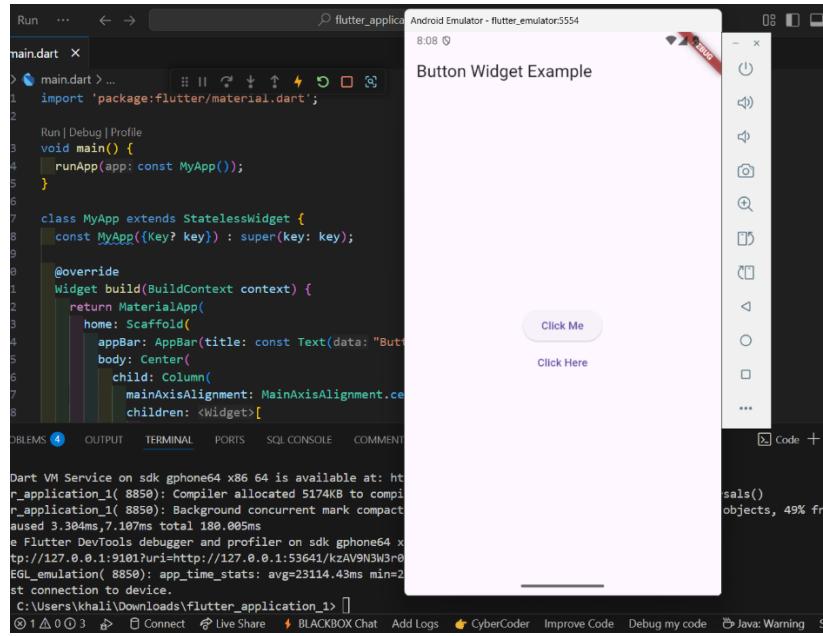
```
import 'package:flutter/material.dart';

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
```

```
            ),
```

```
            ),
```

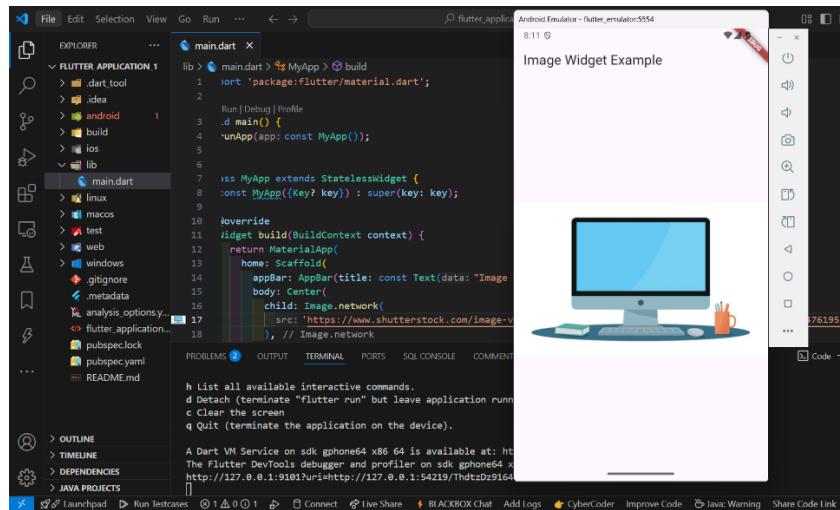
```
            ),
```

```
        );
```

```
}
```

```
}
```

#### 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(0xFFFFFFFF)),

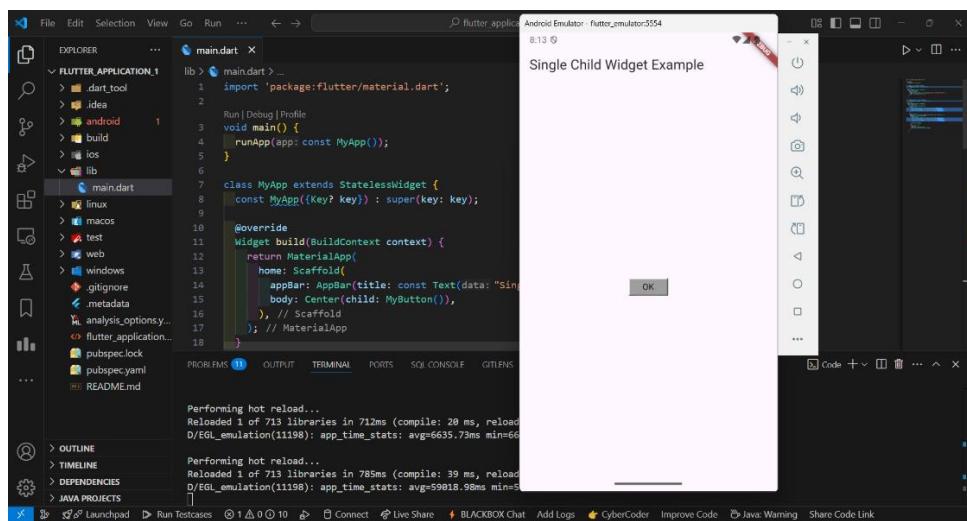
```

```

        left: BorderSide(width: 1.0, color: Color(0xFFFFFFFF)),
        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:



## 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. DropdownButton FormField: 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 GlobalKey<FormState> _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!')),
);
}
}

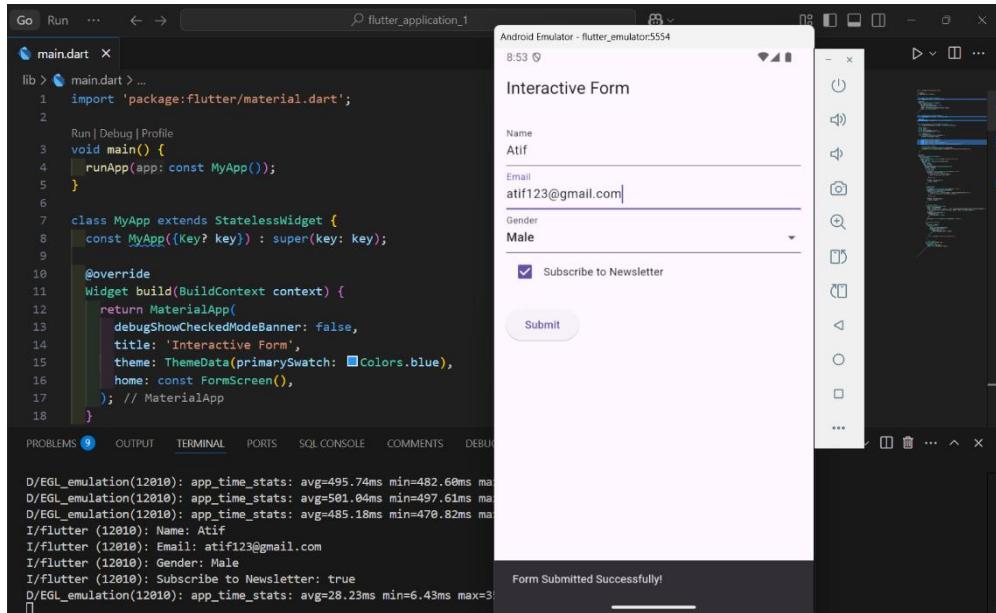
@Override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Interactive Form')),
    body: Padding(
      padding: const EdgeInsets.all(16.0),
      child: Form(
        key: _formKey,
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          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,
      ),
    ),
  ],
),
],
),
],

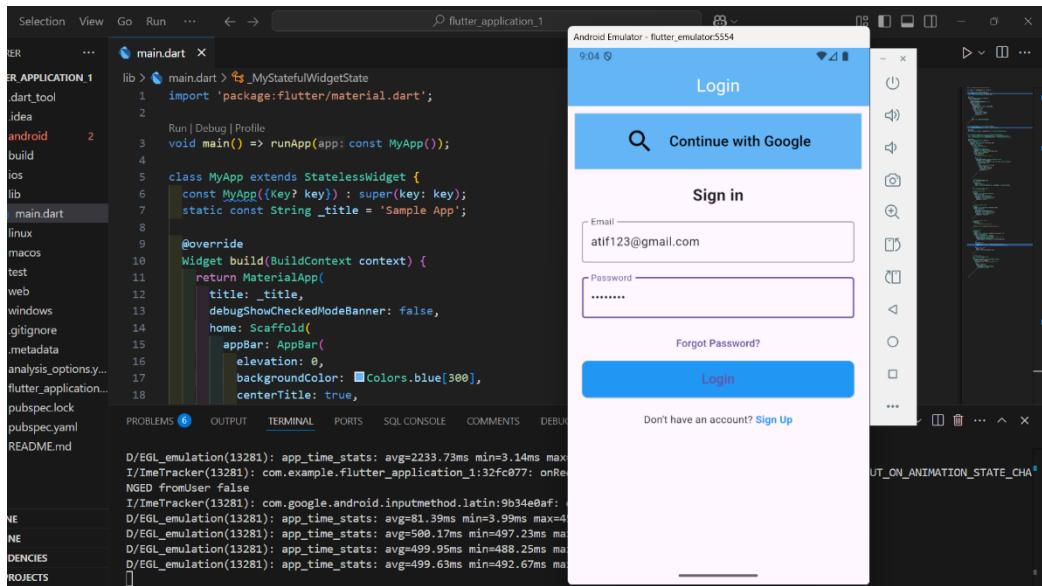
```

```

        ),
    );
}
}

```

## 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'),  

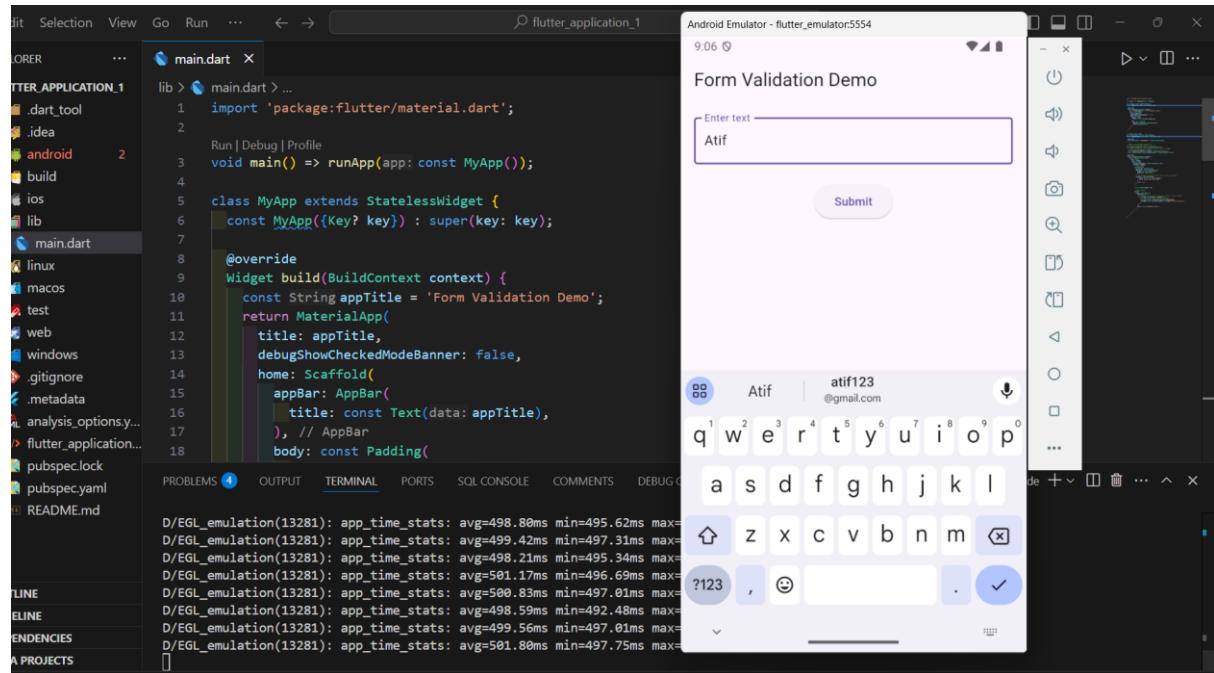
  ),  

),  

),
),
);
}
}

```

## Output:



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 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 is 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()),
            );
          },
        ),
      ),
    );
  }
}
```

```

    );
}

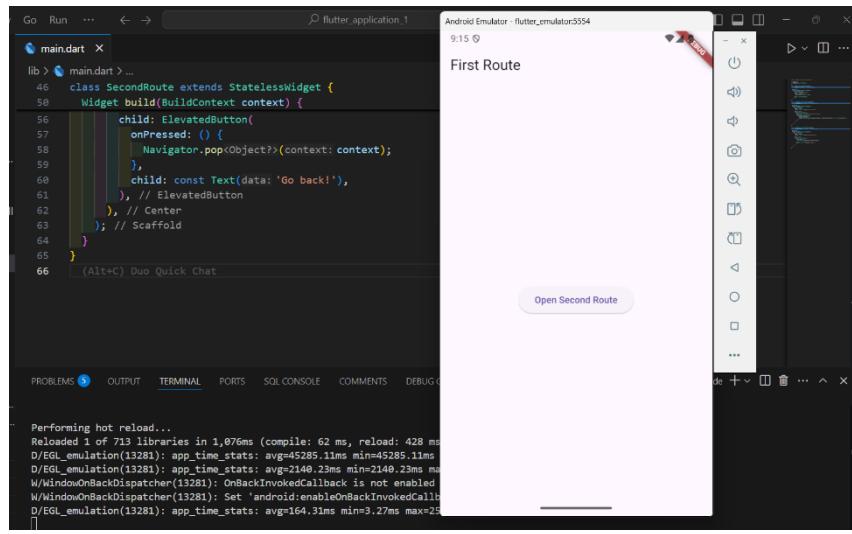
}

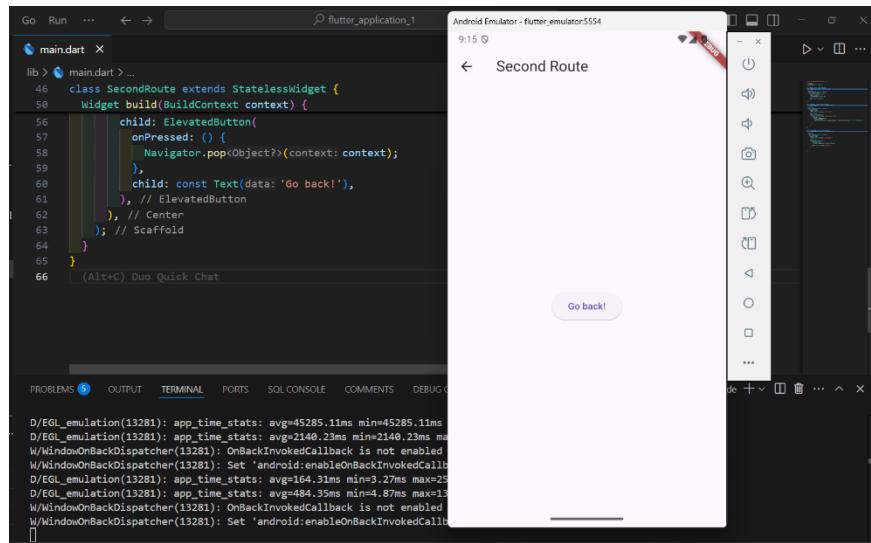
class SecondRoute extends StatelessWidget {
  const SecondRoute({Key? key}) : super(key: key);

  @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);

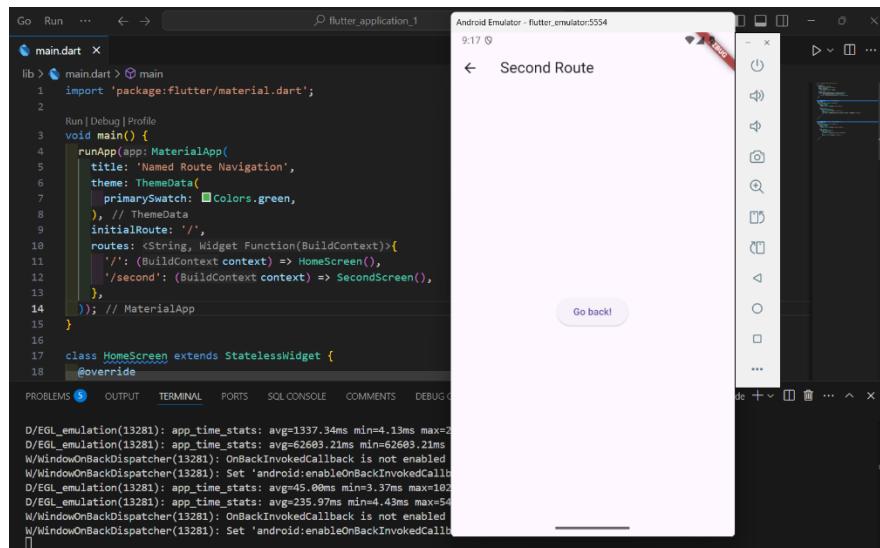
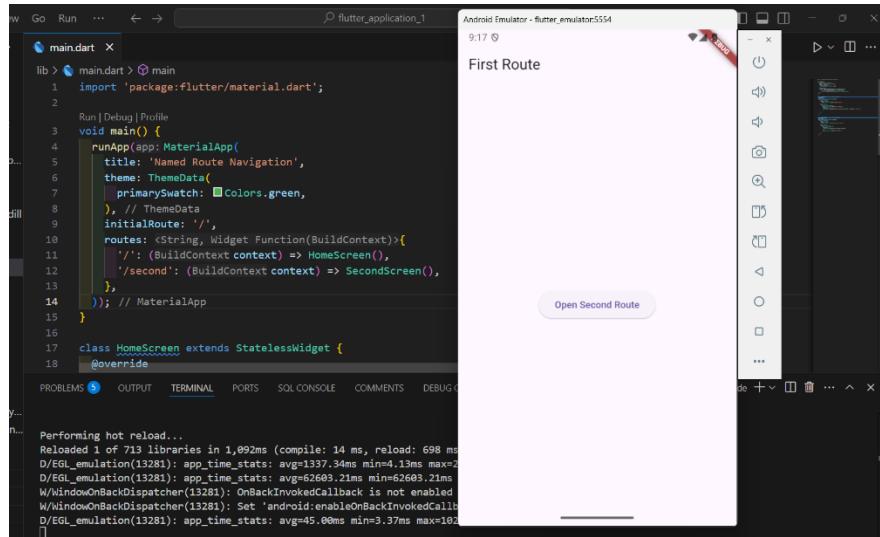
@Override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Home Screen'),
    ),
    body: Center(
      child: ElevatedButton(
        child: const Text('Click Here'),
        onPressed: () {
          Navigator.pushNamed(context, '/second');
    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

}
```

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

```

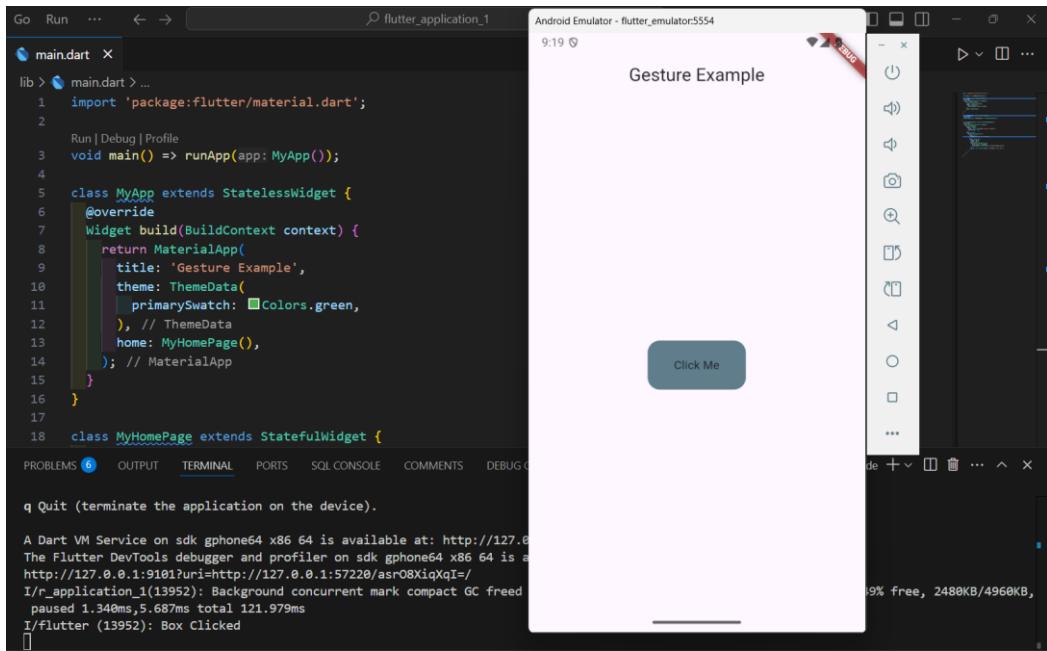
    );
}
}

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);
```

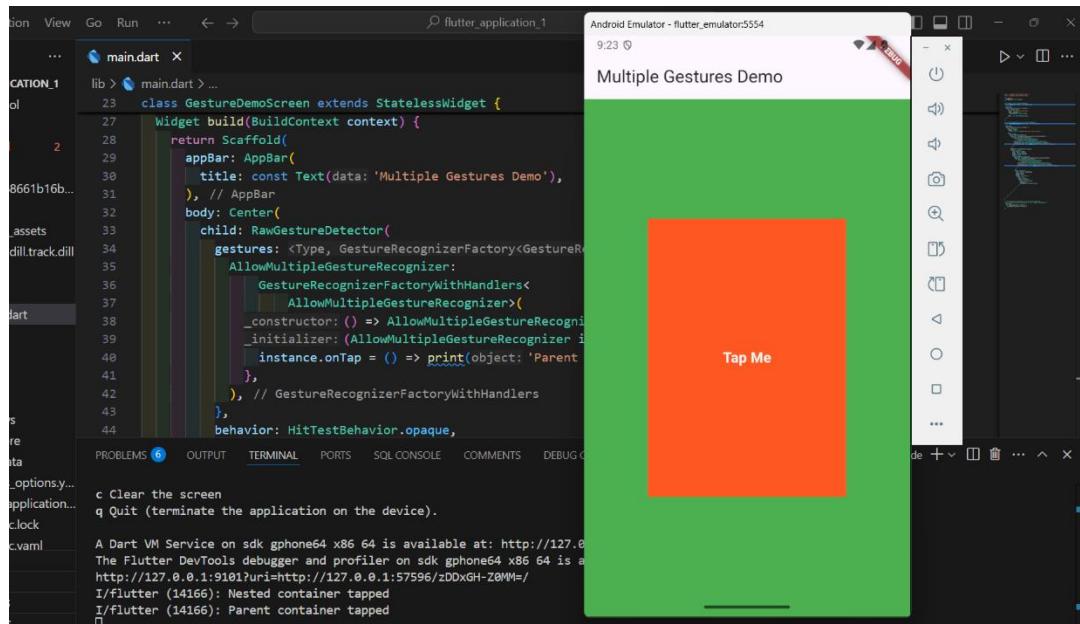
```

@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:



## 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 at 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

### 3. Implementing Firebase

within the app for insights and improvements.

Other advantages of using Firebase are :

1. Cross-Platform Support.

2. Easy Integration.

3. Serverless Architecture.

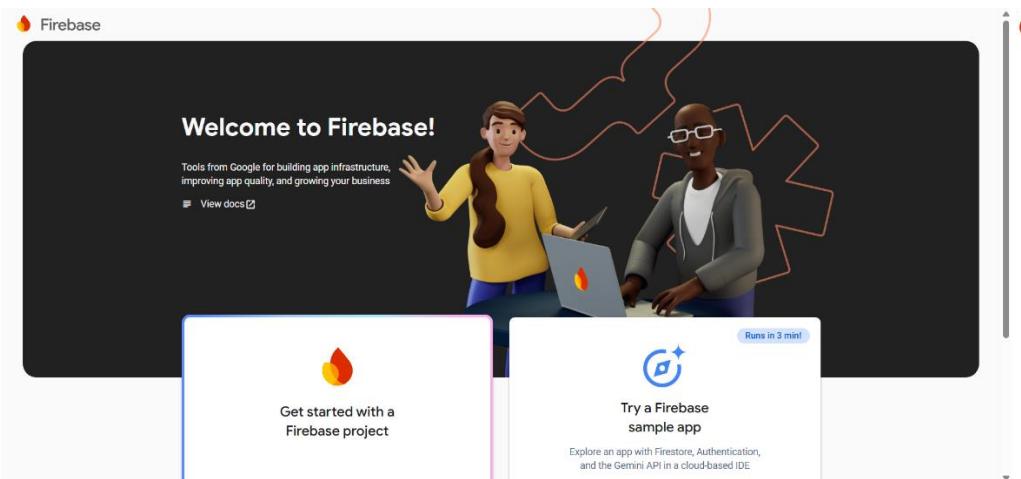
4. Real-time Capabilities.

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.

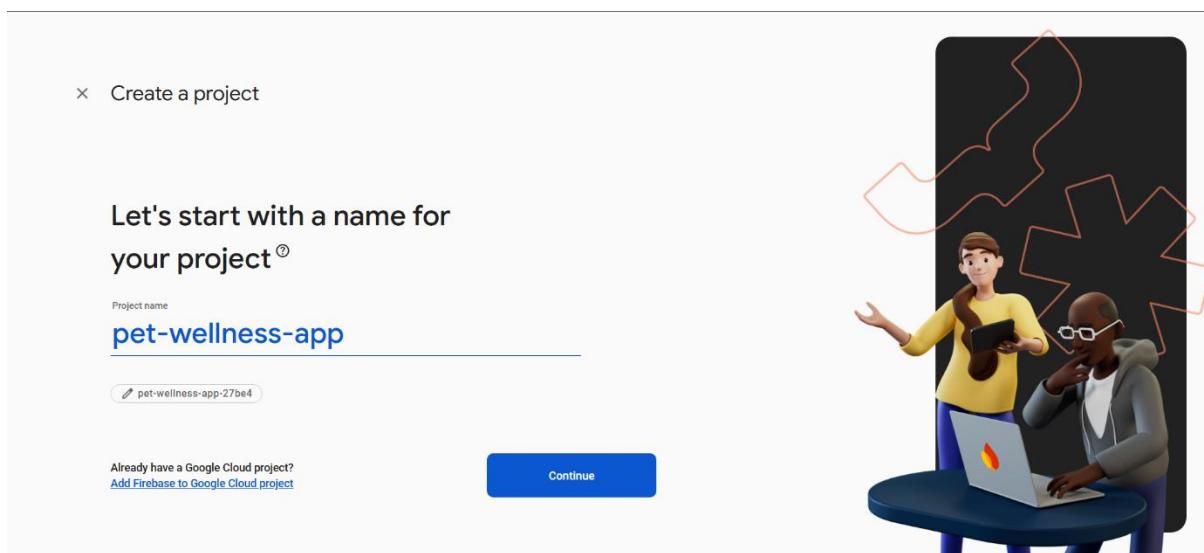
With Firebase integrated, the app can now:

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

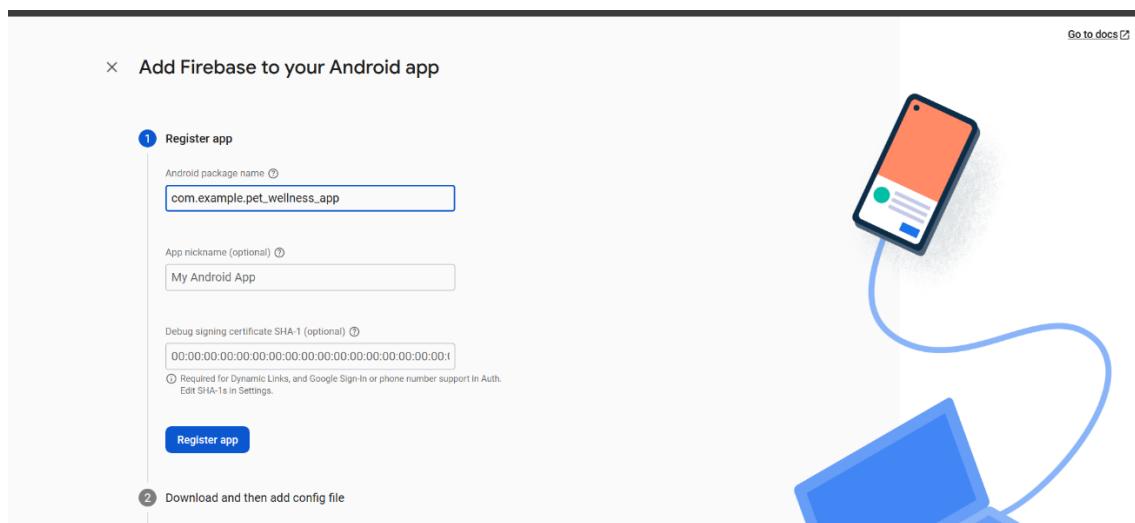
**Step 1:** First, log in with your Google account to manage your Firebase projects.



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

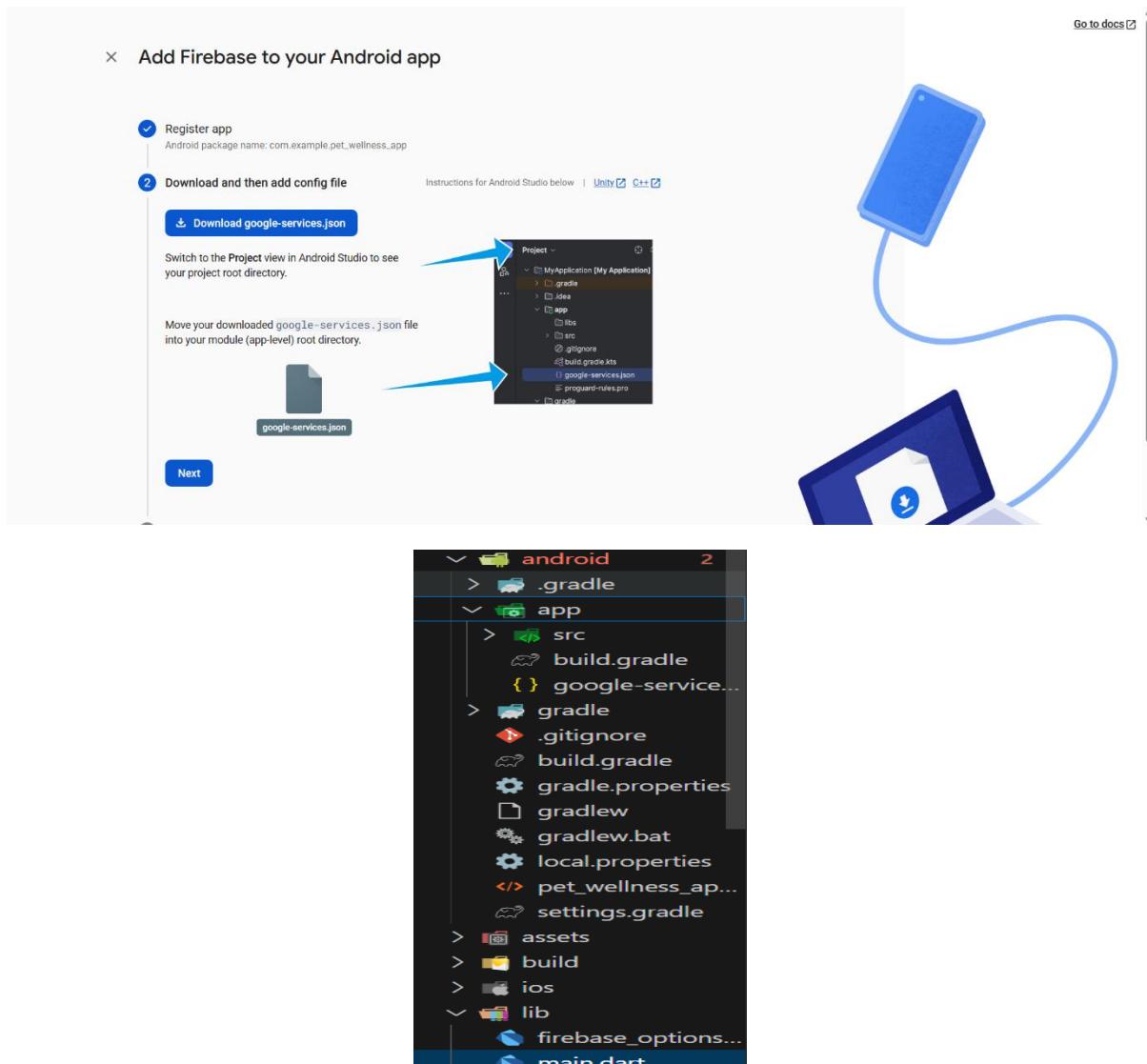


**Step 3:** In order to add Android support to our Flutter application, select the Android logo from the dashboard. This brings us to the following screen:



**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:

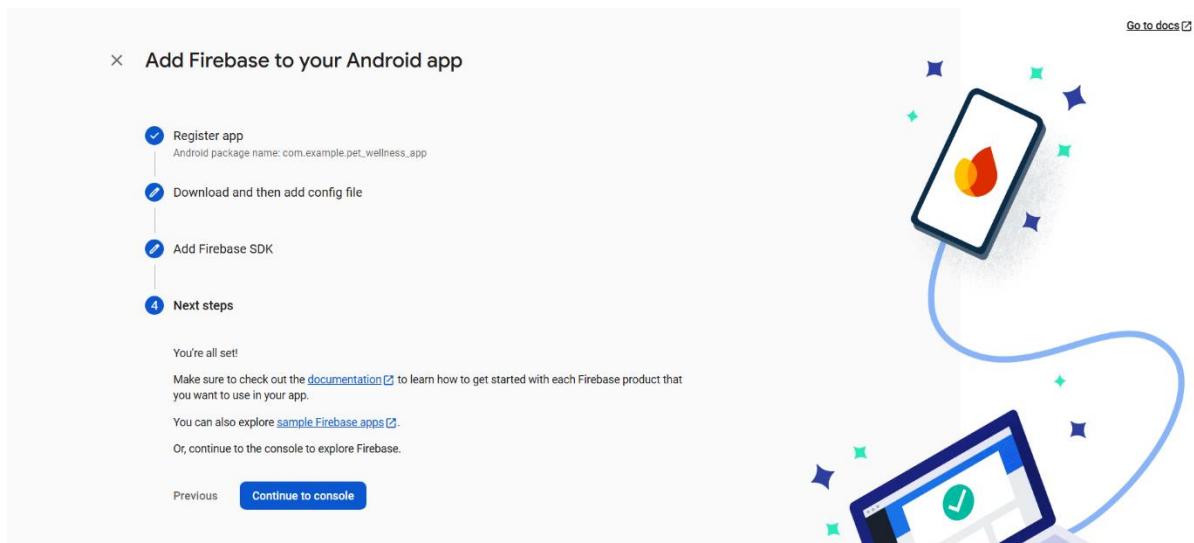
```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'com.google.gms.google-services'

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.

## 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~~ 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 separately 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

- Registration: The browser is informed about the service worker, and it gets registered.
  - Installation: Required assets are cached.
  - 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.

### 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 manages tasks when offline, and Push enables real-time notifications. These features make the app faster, more interactive, and usable even without internet access.

### MPL Experiment - 10

Aim: To study and implement the deployment of an E-commerce 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 E-commerce 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.

## 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 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 application's 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.