EXPERIMENT – 1

AIM:

a) Install Flutter and Dart SDK.

b) Write a simple Dart program to understand the language basics.

c) Write a Dart console program that prints your name, checks age with Conditionals, uses a loop to count from 1 to 5, and defines a function to return the sum of two numbers.

SOLUTION:

Experiment - 1(a)

AIM: Install Flutter and Dart SDK.

Install and Configure Flutter SDK on Windows:

Flutter, a revolutionary open-source UI software development kit by Google, is a game-changer in mobile app development. Its cross-platform capabilities enable developers to create high- performance, natively compiled applications for both iOS and Android using a single codebase. This not only accelerates development but also significantly reduces time and costs. With Flutter, developers harness a robust widget library, offering a rich set of pre-designed widgets for creating visually stunning and customized user interfaces. It's hot-reload feature facilitates real-time code changes and immediate feedback, streamlining the development process. In an increasingly competitive app market, Flutter empowers businesses to reach a wider audience efficiently. Its appeal extends to both startups and established enterprises seeking a cost- effective and agile solution for mobile app development.

System requirements:

Operating Systems: Windows 10 or later (64-bit), x86–64 based.
Disk Space: 2.5 GB (does not include disk space for IDE/tools).
Tools: Flutter depends on these tools being available in your environment.

- Windows PowerShell 5.0 or newer (this is pre-installed with Windows 10)
- Git for Windows 2.x, with the **Use Git from the Windows Command Prompt** option. If Git for Windows is already installed, make sure you can run git commands from the command prompt or PowerShell.

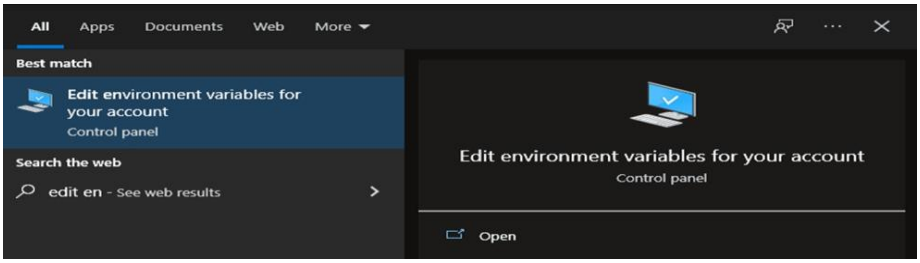
Get the Flutter SDK

Step 1: Download the following installation bundle to get the latest stable release of the Flutter SDK.
URL:
https://storage.googleapis.com/flutter_infra_release/releases/stable/windows/flutter_windows_3.13.7-stable.zip

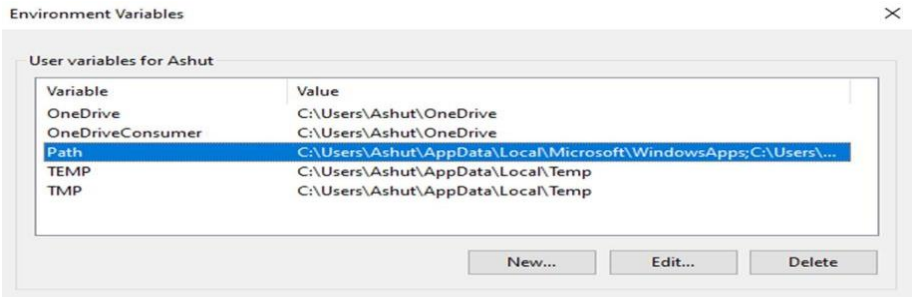
Step 2: Extract the File: Extract the downloaded zip file and move it to the desired location where you want to install Flutter SDK.
Do not install it in a folder or directory that requires elevated privileges, (such as *C:\Program Files*\) to ensure the program runs properly. For this tutorial, it will be stored in *C:\development\flutter*.
You are now ready to run Flutter commands in the Flutter Console.

Step 3: Update Path Variable for Windows PowerShell
If you wish to run Flutter commands in the regular Windows console, take these steps to add Flutter to the PATH environment variable:

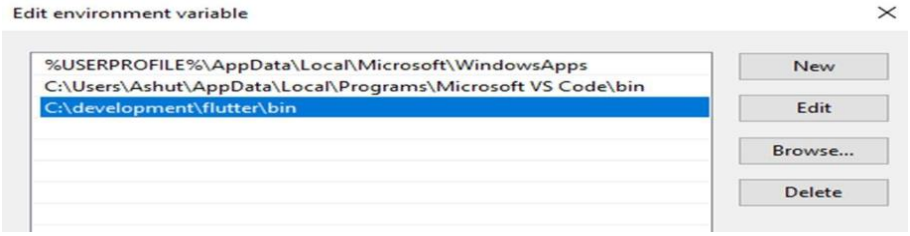
- From the Start search bar, enter 'env' and select Edit environment variables for your account.



- Under **User variables** check if there is an entry called **Path**:
- If the entry exists, append the full path to flutter\bin using; as a separator from existing values.



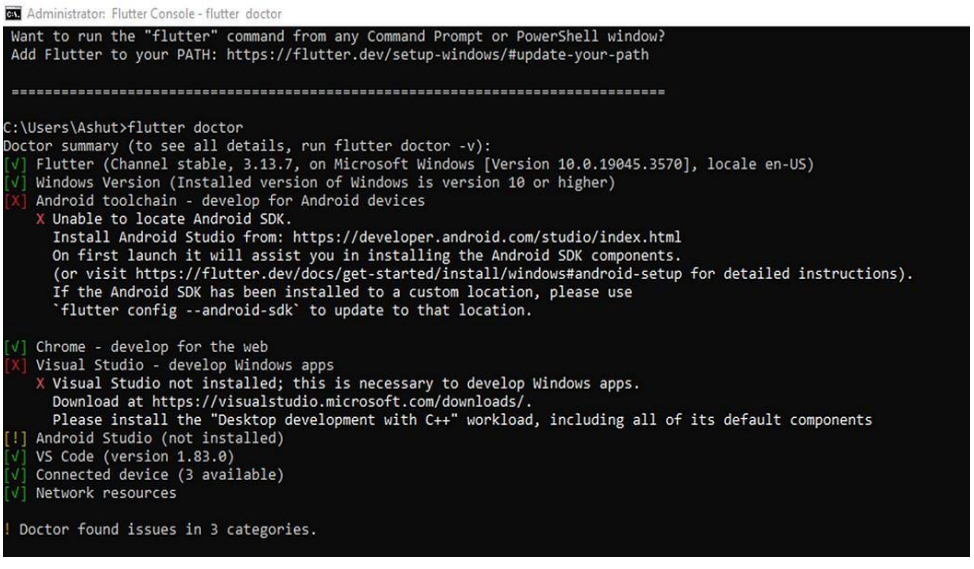
On the next screen, click **New** and add the full path to your *flutter\bin* directory. For this guide, it is shown below. Click OK on both windows to enable running Flutter commands in Windows consoles.



If the entry doesn't exist, create a new user variable named Path with the full path to flutter\bin as its value.

Step 4: Confirm Installed Tools for Running Flutter

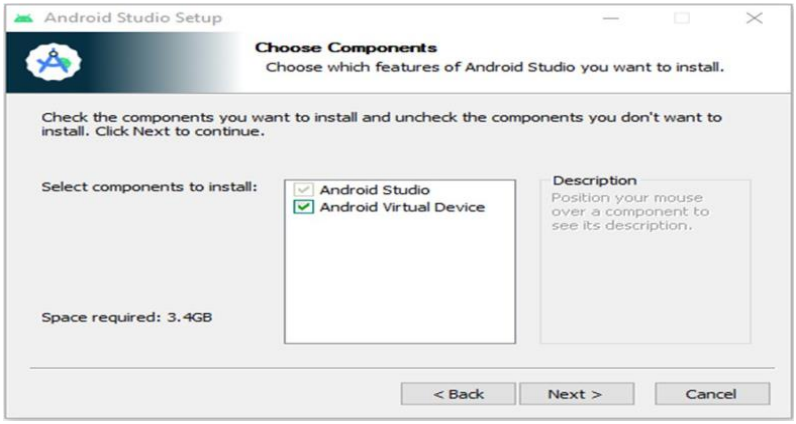
In CMD, run the *flutter doctor* command to confirm the installed tools along with brief descriptions. As visible, several components still need to be installed to complete the installation.



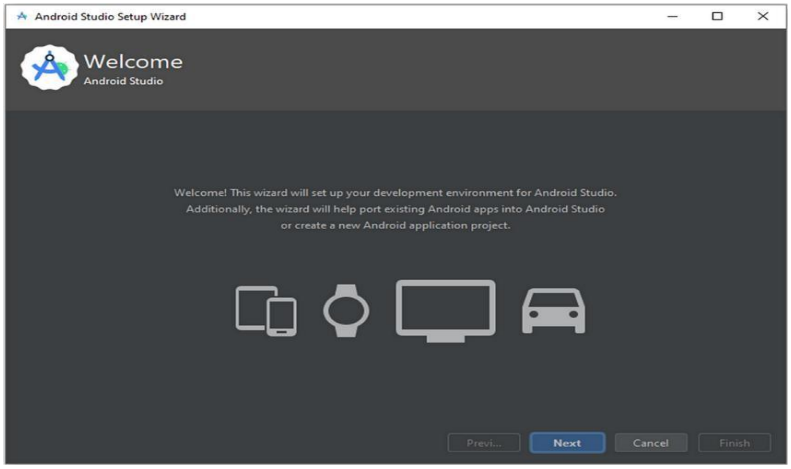
Step 5: Download and Install Android Studio

Visit the official Android Studio download page at https://developer.android.com/studio. Click on the "Download Android Studio" button.

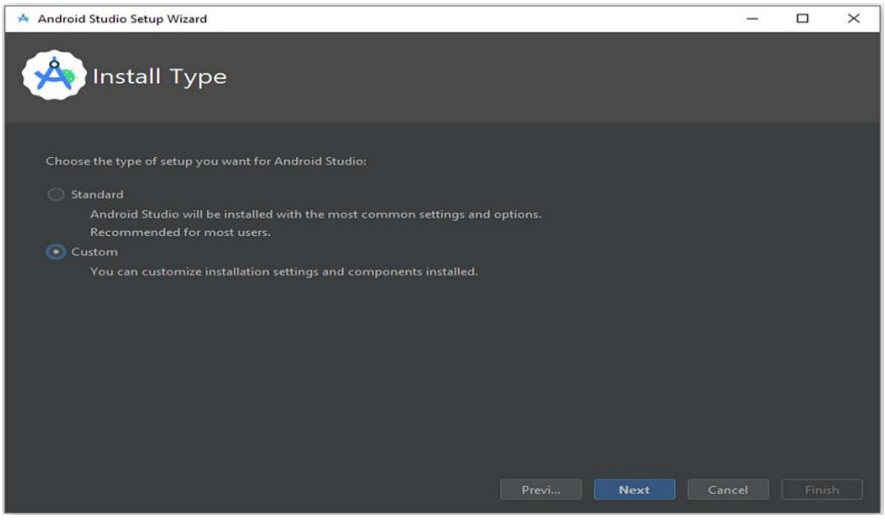
Next, proceed by downloading Android Studio. During the setup, unless you have unique requirements, simply click "Next" on all screens to keep the default settings. On the "Choose Components" screen, be sure to select the "Android Virtual Device" option to enable an Android emulator for your app development needs.



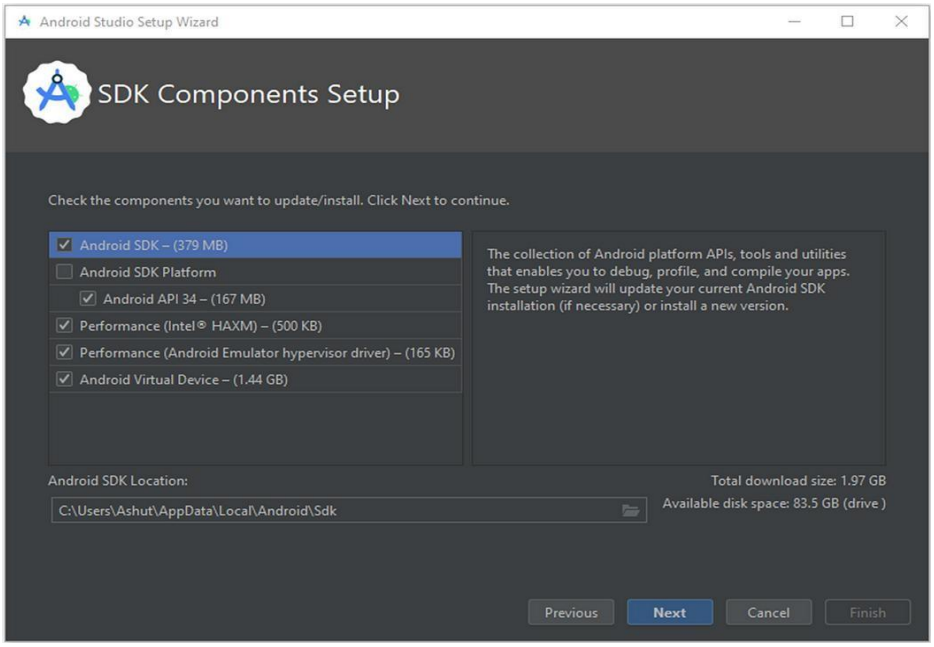
Afterward, The Android Studio Setup Wizard will start and you can proceed by clicking **Next**.



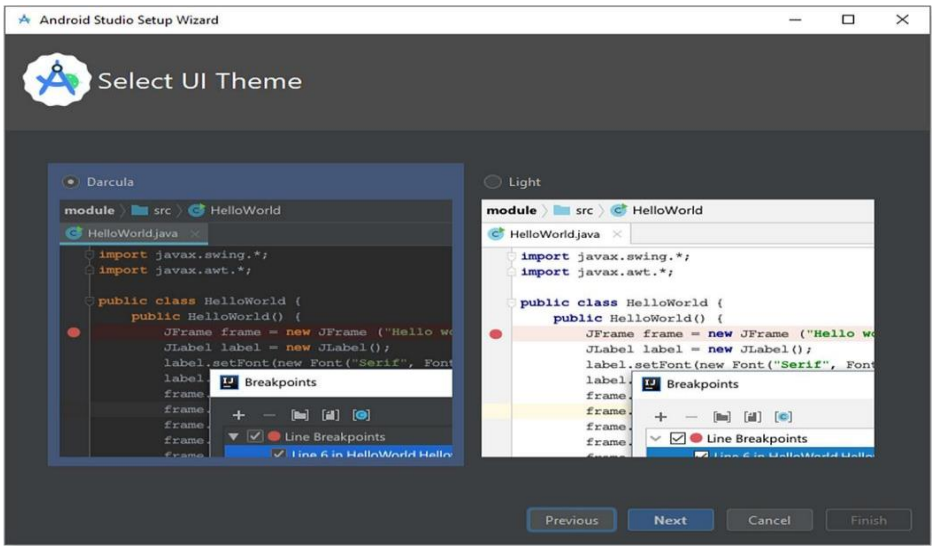
On the Install Type screen, select Custom and click **Next**.



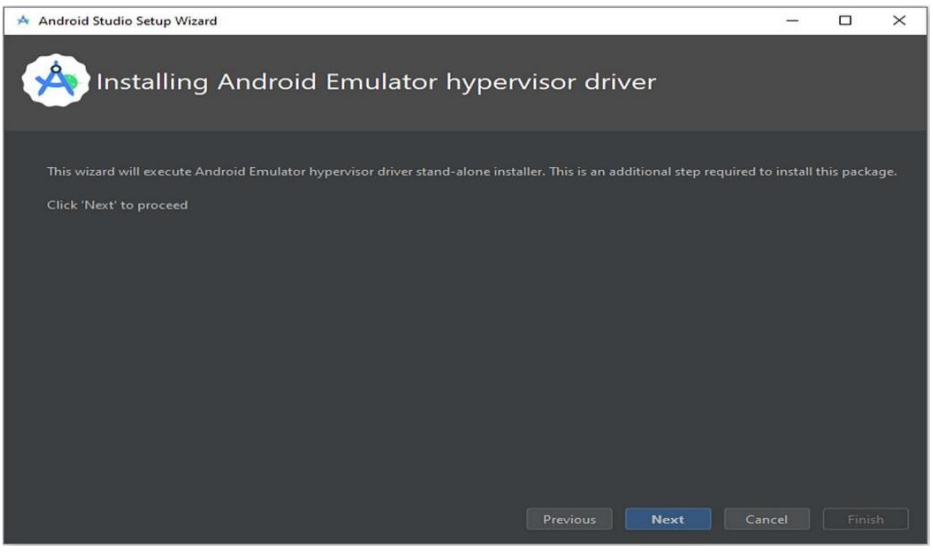
Select the installation location or leave the default path and click **Next**.



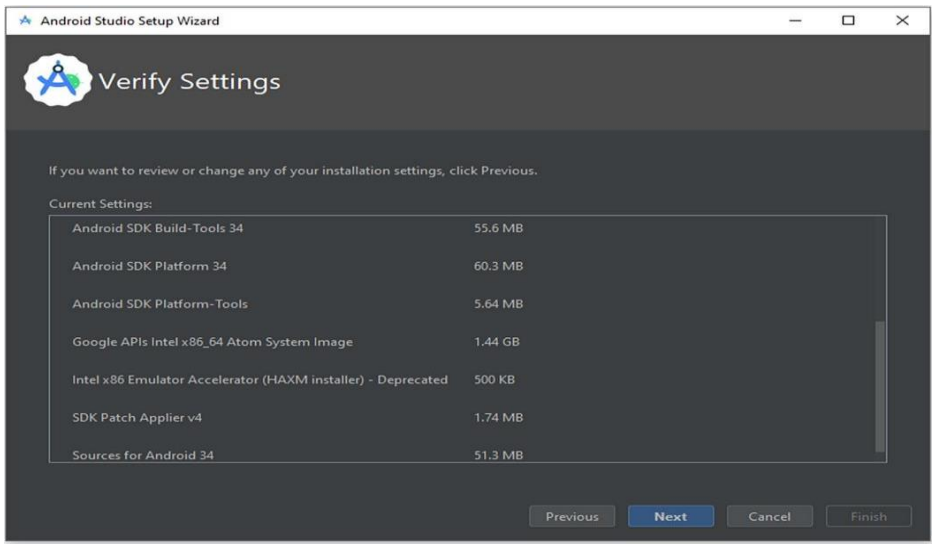
Select your UI theme and click **Next**.



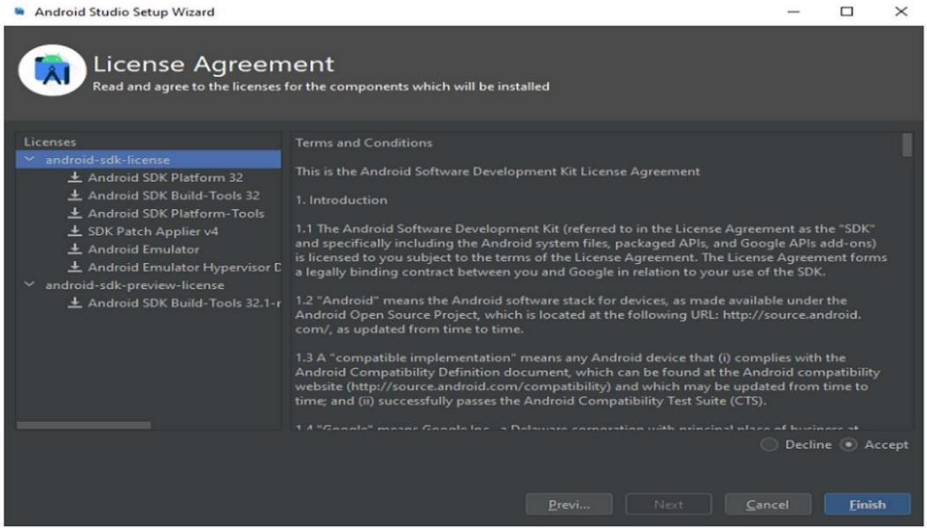
Click Next to proceed for the Android Emulator hypervisor driver stand-alone installer.



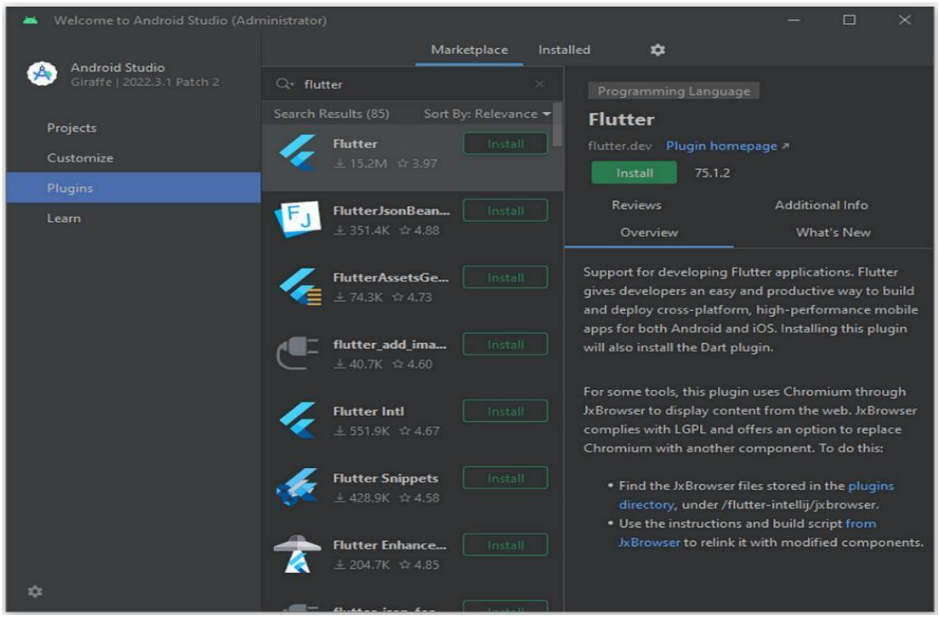
Verify the settings and click **Next**.



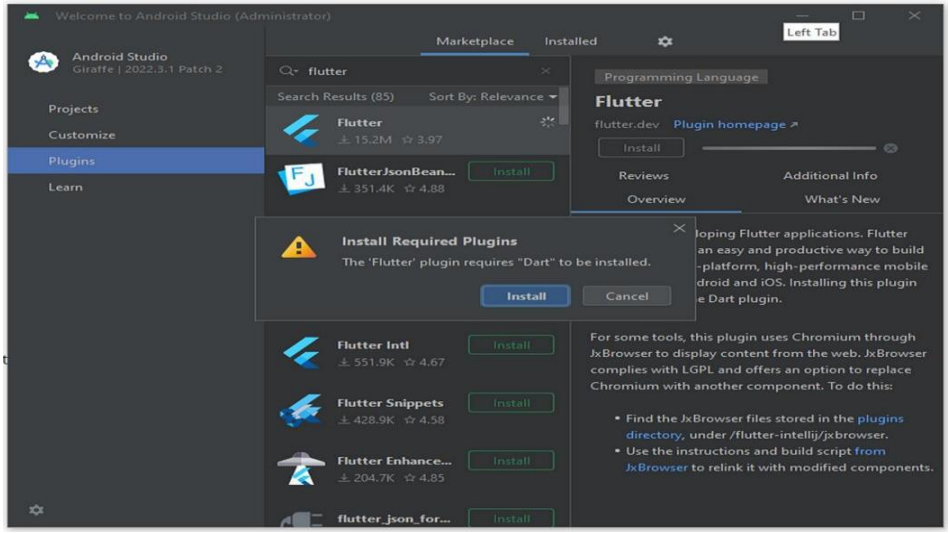
On the next screen, accept the License Agreement and click **Finish**.



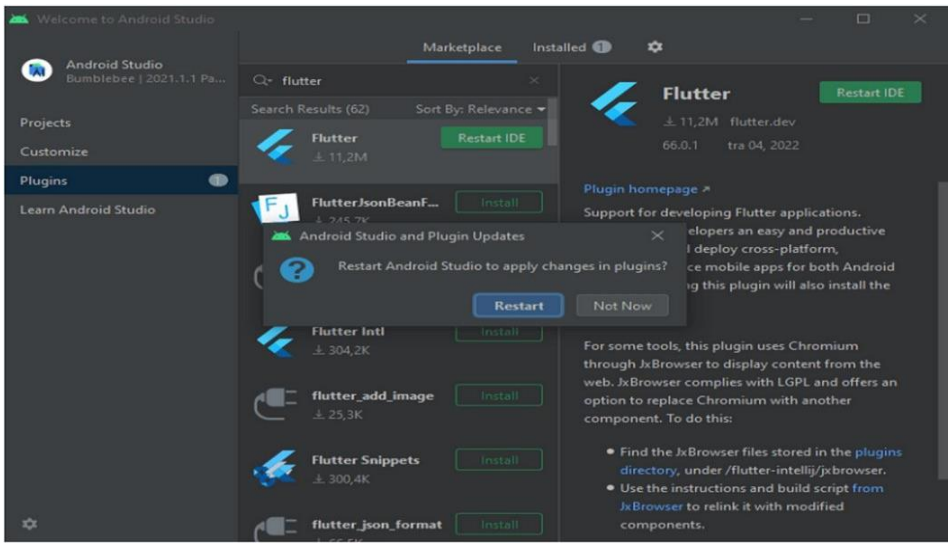
The download of the components will start and Android Studio install. Once completed, click **Finish**. After the installation, start Android Studio. On the left side, click **Plugins**. Search for Flutter and click **Install** to install the Flutter plugin.



It will also prompt you to install Dart, a programming language used to create Flutter apps. Click **Install** at the prompt.



Finally, click **Restart IDE** so that the plugin changes are applied. Click **Restart** at the prompt to confirm this action.



Afterward, run the *flutter doctor* command in CMD to confirm the Android Studio installation.

```
C:\Users\blup>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 2.10.4, on Microsoft Windows [version 10.0.19041.746), locale en-US)
[!] Android toolchain - develop for Android devices (Android SDK version 32.1.0-rc1)
    ! Some Android licenses not accepted. To resolve this, run: flutter doctor --android-licenses
[√] Chrome - develop for the web
[X] Visual Studio - develop for Windows
    X Visual Studio not installed; this is necessary for Windows development.
      Download at https://visualstudio.microsoft.com/downloads/.
      Please install the "Desktop development with C++- workload, including all of its default components
[√] Android Studio (version 2021.1)
[√] Connected device (2 available)
[√] HTTP Host Availability

! Doctor found issues in 2 categories.
```

Android Studio was successfully installed; however, it found an issue with Android licenses. This issue is fairly common and is mitigated by running the following command in CMD.

flutter doctor --android-licenses

When asked, input **y** to all prompts, to accept licenses.

```
C:\Users\blup>flutter doctor --android-licenses
5 of 7 SDK package licenses not accepted. 100% Computing updates...
Review licenses that have not been accepted (y/N)? y
```

Running the *Flutter Doctor* command again shows the issue resolved.

```
C:\Users\blup>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 2.10.4, on Microsoft Windows [Version 10.0.19041.746], locale en- US)
[√] Android toolchain - develop for Android devices (Android SDK version 32.1.0-rc1)
[√] Chrome - develop for the web
[X] Visual Studio - develop for Windows
    X Visual Studio not installed; this is necessary for Windows development.
      Download at https://visualstudio.microsoft.com/downloads/.
      Please install the "Desktop development with C++" workload, including all of its default
components
[√] Android Studio (version 2021.1)
[√] Connected device (2 available)
[√] HTTP Host Availability

! Doctor found issues in 1 category.
```

Step 6: Install Visual Studio (Optional)

The above output also shows that Visual Studio is not installed. Visual Studio is not needed unless you want to use Flutter for Windows desktop development.

If you need to use it, you can download Microsoft's Visual Studio 2022 with C++ (URL: https://visualstudio.microsoft.com/downloads/). Once the *VisualStudioSetup.exe* file is downloaded, open it and proceed with the installation by agreeing to all default installation options.

This installation requires at least 20 GB of free disk space. After the installation completes, run the *flutter doctor* command in CMD to confirm the Visual Studio installation.

```
C:\Users\blup>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 2.10.4, on Microsoft Windows [Version 10.0.19041.746], locale en-
US)
[√] Android toolchain - develop for Android devices (Android SDK version 32.1.0-rc1)
[√] Chrome - develop for the web
[√] Visual Studio - develop for Windows (Visual Studio Community 2022 17.1.3)
[√] Android Studio (version 2021.1)
[√] Connected device (2 available)
[√] HTTP Host Availability

• No issues found!
```

At this point, all the tools for Flutter projects are ready to be used for the development of Flutter apps. Depending on your needs, you can start your projects in Android Studio or Visual Studio.

Experiment - 1(b)

AIM: Write a simple Dart program to understand the language basics.

SOLUTION:

```
// Define a main function, which is the entry point of a Dart program.
void main()
{
  // Variables and data types
  int myNumber = 10;
  double myDouble= 3.14;
  String myString = 'Hello World';
  bool myBool = true;

  // Printing variables
  print('My number is: $myNumber');
  print('My double is: $myDouble');
  print('My string is: $myString');
  print('My boolean is: $myBool');

  // Basic arithmetic operations
  int result= myNumber + 5;
  print('Result of addition: $result');

  // Conditional statements
  if (myBool)
  {
    print('myBool is true');
  }
  else
  {
    print('myBool is false');
  }

  // Loops
  for (int i = 0; i < 5; i++)
  {
    print('Iteration $i');
  }

  // Lists
  List numbers = [1, 2, 3, 4, 5];
  print('First element of the list: ${numbers[0]}');
  print('Length of the list: ${numbers.length}');

  // Maps
  Map branch_codes = { 'CSE': 05,'AIML': 42,'DS': 44};
  print('List of Engineering branches with codes: $branch_codes');
  print('List of branches: ${branch_codes.keys}');
  print('List of branch codes: ${branch_codes.values}');
  print('AIML Branch code: ${branch_codes['AIML']}');

}
```

Output:

My number is: 10
My double is: 3.14
My string is: Hello World
My boolean is: true
Result of addition: 15
myBool is true
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
First element of the list: 1
Length of the list: 5
List of Engineering branches with codes: {CSE: 5, AIML: 42, DS: 44}
List of branches: (CSE, AIML, DS)
List of branch codes: (5, 42, 44)
AIML Branch code: 42

Experiment - 1(c)

AIM: Write a Dart console program that prints your name, checks age with conditionals, uses a loop to count from 1 to 5, and defines a function to return the sum of two numbers.

SOLUTION:

```
void main()
{
  // 1. Print your name
  String name = "Your Name";
  print("Hello, my name is $name.");

  // 2. Check age with conditionals
  int age = 18;
  if (age >= 18)
  {
    print("You are an adult.");
  }
  else
  {
    print("You are a minor.");
  }

 // 3. Loop to count from 1 to 5
print("Counting from 1 to 5:");
for (int i = 1; i <= 5; i++)
{
   print(i);
}

// 4. Function to return the sum of two numbers
int a = 10;
int b = 15;
int result = addNumbers(a, b);
print("Sum of $a and $b is $result.");
}

// Function definition
int addNumbers(int num1, int num2)
{
  return num1 + num2;
}
```

OUTPUT:

```
Hello, my name is Your Name. You
are an adult.
Counting from 1 to 5:
1
2
3
4
5
Sum of 10 and 15 is 25.
```

EXPERIMENT – 2

AIM:

a) Explore various Flutter widgets (Text, Image, Container, etc.).

b) Implement different layout structures using Row, Column, and Stack widgets.

c) Create a Flutter app with a Text widget showing a counter value, and an ElevatedButton that increments the counter using set State ().

DESCRIPTION:

Flutter provides a rich set of widgets to build user interfaces for mobile, web, and desktop applications. These widgets help in creating visually appealing and interactive UIs. Here are some of the commonly used Flutter widgets categorized by their functionalities:

Layout Widgets:

Container: A versatile widget that can contain other widgets and provides options for alignment, padding, margin, and decoration.

Row and Column: Widgets that arrange their children in a horizontal or vertical line respectively.

Stack: Allows widgets to be stacked on top of each other, enabling complex layouts.

ListView and GridView: Widgets for displaying a scrollable list or grid of children, with support for various layouts and scrolling directions.

Scaffold: Implements the basic material design layout structure, providing app bars, drawers, and floating action buttons.

Text and Styling Widgets:

Text: Displays a string of text with options for styling such as font size, color, and alignment.

RichText: Allows for more complex text styling and formatting, including different styles within the same text span.

TextStyle: A class for defining text styles that can be applied to Text widgets.

Input Widgets:

TextField: A widget for accepting user input as text, with options for customization and validation.

Checkbox and Radio: Widgets for selecting from a list of options, either through checkboxes or radio buttons.

DropdownButton: Provides a dropdown menu for selecting from a list of options.

Button Widgets:

ElevatedButton and TextButton: Widgets for displaying buttons with different styles and customization options.

IconButton: A button widget that displays an icon and responds to user taps.

GestureDetector: A versatile widget that detects gestures such as taps, swipes, and drags, allowing for custom interactions.

Image and Icon Widgets:

Image: Widget for displaying images from various sources, including assets, network URLs, and memory.

Icon: Displays a Material Design icon.

Navigation Widgets:

Navigator: Manages a stack of route objects and transitions between different screens or pages in the app.

PageRouteBuilder: A customizable widget for building page transitions and animations.

Animation Widgets:

AnimatedContainer: An animated version of the Container widget, with support for transitioning properties over a specified duration.

AnimatedOpacity, AnimatedPositioned, AnimatedBuilder: Widgets for animating opacity, position,and custom properties respectively.

Material Design Widgets:

AppBar: A material design app bar that typically contains a title, leading and trailing widgets, and actions.

BottomNavigationBar: Provides a navigation bar at the bottom of the screen for switching betwee different screens or tabs.

Card: Displays content organized in a card-like structure with optional elevation and padding.

Cupertino (iOS-style) Widgets:

CupertinoNavigationBar: A navigation bar in the iOS style.

CupertinoButton: A button widget with the iOS style.

CupertinoTextField: A text field widget with the iOS style.

These are just a few examples of the many widgets available in Flutter. Each widget comes with its set of properties and customization options, allowing developers to create highly customizable and responsive user interfaces.

Experiment - 2(a)

AIM: Explore various Flutter widgets (Text, Image, Container, etc.).

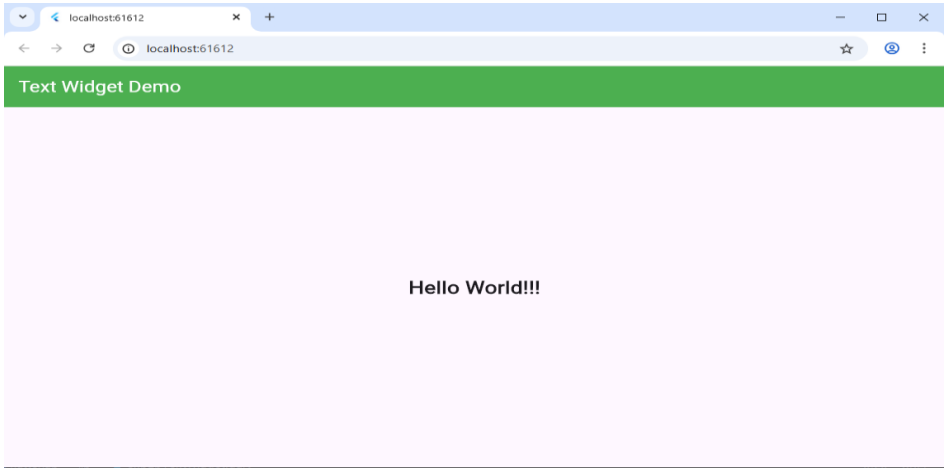
SOLUTION:

Flutter Text widget:

```
import 'package:flutter/material.dart';
void main()
{
  runApp(MyApp());
}
class MyApp extends StatelessWidget
{
  @override
  Widget build(BuildContext context)
  {
    return MaterialApp(
       debugShowCheckedModeBanner: false,
       home: HomePage()
    );
  }
}
class HomePage extends StatelessWidget
{
  const HomePage({super.key});
  @override
  Widget build(BuildContext context)
  {
    return Scaffold(
      appBar: AppBar(

        // Set the background color of the app bar
        backgroundColor: Colors.green,

        // Set the foregroundColor color of the app bar
        foregroundColor: Colors.white,

        // Set the title of the app bar
        title: Text("Text Widget Demo"),
      ),

      // The main body of the scaffold
      body: Center(
        // Display a centred text widget

        child: Text( 'Hello World!!!',
```

```
        // Apply text styling
        style: TextStyle(
                fontSize: 24,
                fontWeight: FontWeight.bold,
        ),
       ),
      ),
    );
  }
}
```

Output:



Flutter Image widget:

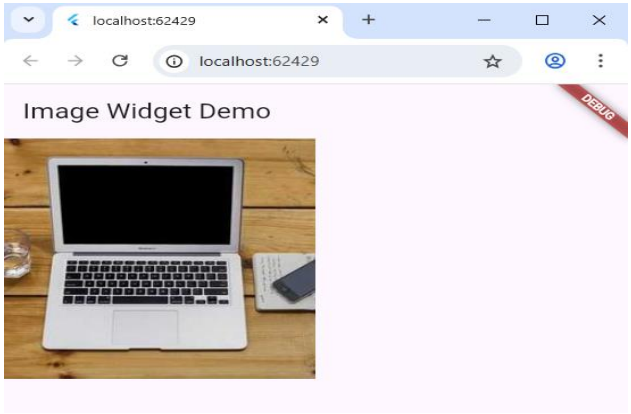
Display images from the internet: To work with images from a URL, use the Image.network() constructor.

SOLUTION:

```
import 'package:flutter/material.dart';
void main()
{
  runApp(MyApp());
}
class MyApp extends StatelessWidget
{
@override
Widget build(BuildContext context)
{
  return MaterialApp(
      home: HomePage()
  );
}
}
class HomePage extends StatelessWidget
{
const HomePage({super.key});
```

```
@override
Widget build(BuildContext context)
{
  return Scaffold(
    appBar: AppBar(title: Text('Image Widget Demo')),
    body: Image.network('https://picsum.photos/250?image=9'),
  );
}
}
```

OUTPUT:



Display images from the Assets: A flutter app when built has both assets (resources) and code. Assets are available and deployed during runtime. The asset is a file that can include static data, configuration files, icons, and images. The Flutter app supports many image formats, such as JPEG, WebP, PNG, GIF, animated WebP/GIF, BMP, and WBMP.

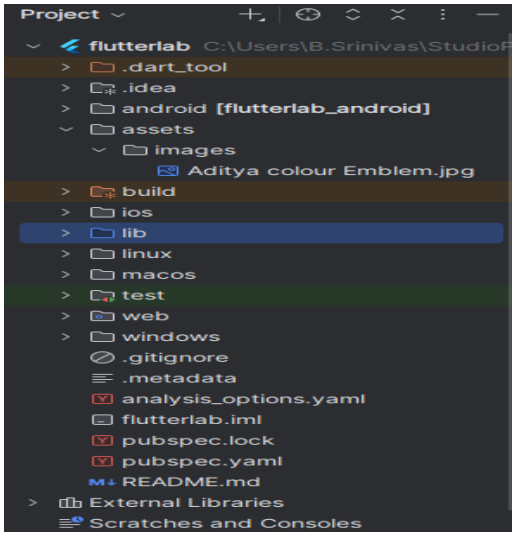
Syntax:

Image.asset('image name')

Steps to Add an Image:

Step 1: Create a new folder

- It should be at the root of your flutter project. You can name it whatever you want, but *assets* are preferred.
- If you want to add other assets to your app, like fonts, it is preferred to make another subfolder named *images*.



Step 2: Now you can copy your image to *images* sub-folder. The path should look like *assets/images/yourImage*. Before adding images also check the above-mentioned supported image formats.

Step 3: Register the assets folder in *pubspec.yaml* file and update it (click Pub get).
a) To add images, write the following code:

```
flutter:
  assets:
       - assets/images/yourFirstImage.jpg
       - assets/image/yourSecondImage.jpg
```

b) If you want to include all the images of the assets folder then add this:

```
flutter:
  assets:
        - assets/images/
```

Note: *Take care of the indentation, assets should be properly indented to avoid any error.*

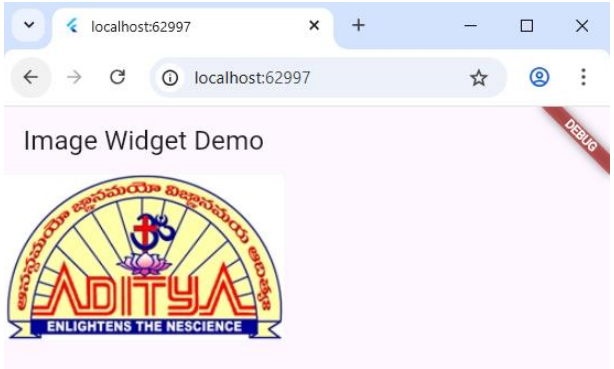
Step 4: Insert the image code in the file, where you want to add the image.

```
Image.asset('assets/images/Aditya colour Emblem.jpg')
```

SOLUTION:

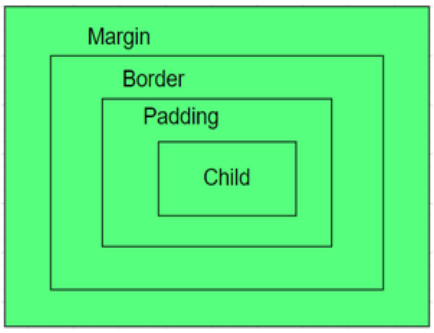
```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: HomePage()
    );
  }
}
class HomePage extends StatelessWidget {
  const HomePage({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Image Widget Demo')),
      body: Image.asset('assets/images/Aditya colour Emblem.jpg'),
    );
  }
}
```

OUTPUT:



Container class in Flutter

Container class in flutter is a convenience widget that combines common painting, positioning, and sizing of widgets. A Container class can be used to store one or more widgets and position them on the screen according to our convenience. Basically, a container is like a box to store contents. A basic container element that stores a widget has a **margin**, which separates the present container from other contents. The total container can be given a **border** of different shapes, for example, rounded rectangles, etc. A container surrounds its child with **padding** and then applies additional constraints to the padded extent (incorporating the width and height as constraints, if either is non-null).

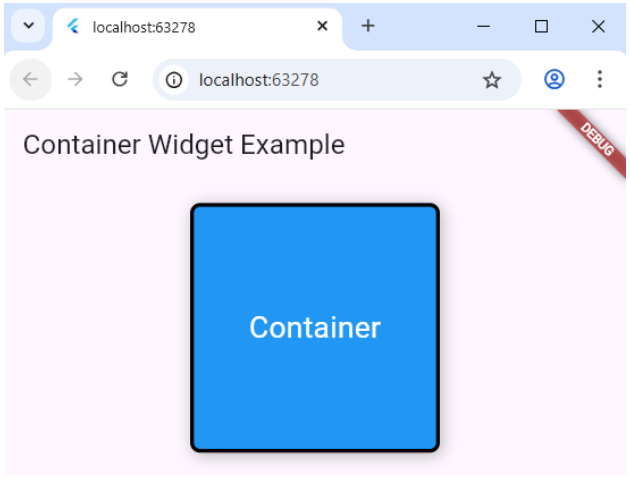


SOLUTION:

```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: HomePage(),
    );
  }
}
class HomePage extends StatelessWidget {
  const HomePage({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Container Widget Example')),
      body: Center(
        child: Container(
          width: 200,
          height: 200,
          padding: EdgeInsets.all(16),
          margin: EdgeInsets.all(16),
          decoration: BoxDecoration(
            color: Colors.blue,
            borderRadius: BorderRadius.circular(8),
            border: Border.all(color: Colors.black, width: 3),
            boxShadow: [
```

```
            BoxShadow(
              color: Colors.black26,
              blurRadius: 10,
              offset: Offset(2, 2),
            ),
          ],
        ),
        child: Center(
          child: Text(
            'Container',
            style: TextStyle(color: Colors.white, fontSize: 24),
          ),
        ),
      ),
    ),
  );
 }
}
```

OUTPUT:



Experiment - 2(b)

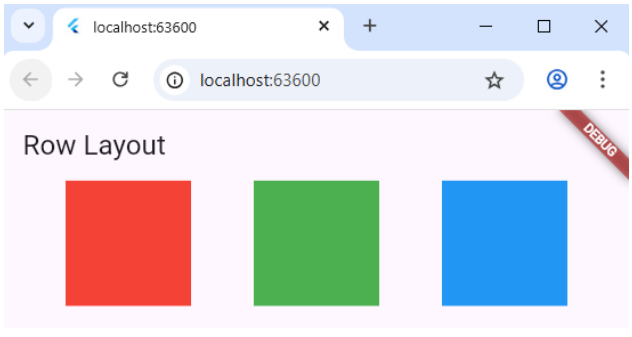
AIM: Implement different layout structures using Row, Column, and Stack widgets.

SOLUTION:

1. Row Layout:

```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Row Layout'),
        ),
        body: Row(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: <Widget>[
            Container(
            color: Colors.red,
            width: 100,
            height: 100,
          ),
            Container(
              color: Colors.green,
              width: 100,
              height: 100,
            ),
            Container(
              color: Colors.blue,
              width: 100,
              height: 100,
            ),
          ],
        ),
      ),
    );
  }
}
```

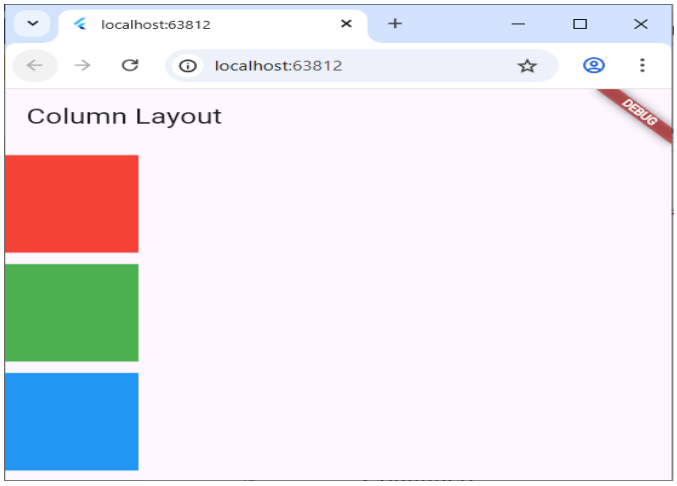
OUTPUT:



2. Column Layout:

```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Column Layout'),
        ),
        body: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: <Widget>[
            Container(
              color: Colors.red,
              width: 100,
              height: 100,
            ),
            Container(
              color: Colors.green,
              width: 100,
              height: 100,
            ),
            Container(
              color: Colors.blue,
              width: 100,
              height: 100,
            ),
          ],
        ),
      ),
    );
  }
}
```

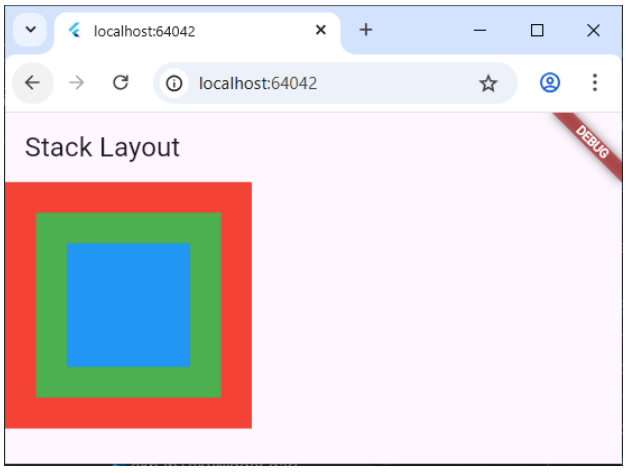
OUTPUT:



3. Stack Layout:

```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Stack Layout'),
        ),
        body: Stack(
          alignment: Alignment.center,
          children: <Widget>[
            Container(
              color: Colors.red,
              width: 200,
              height: 200,
            ),
            Container(
              color: Colors.green,
              width: 150,
              height: 150,
            ),
            Container(
              color: Colors.blue,
              width: 100,
              height: 100,
            ),
          ],
        ),
      ),
    );
  }
}
```

OUTPUT:



Experiment - 2(c)

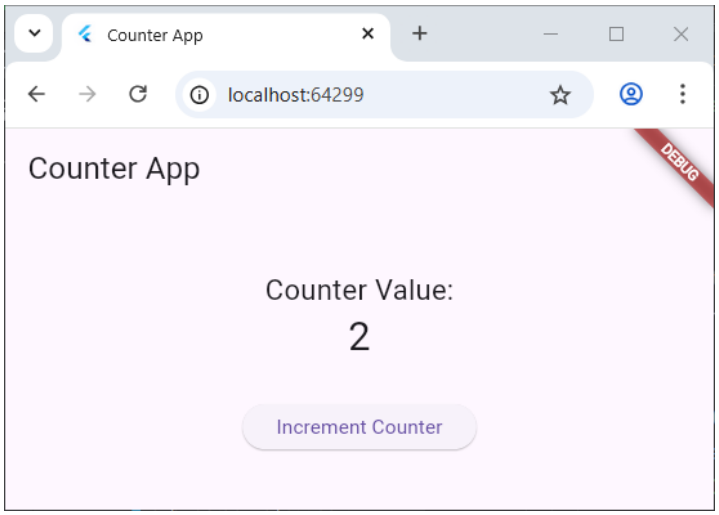
AIM: Create a Flutter app with a Text widget showing a counter value, and an ElevatedButton that increments the counter using set State ().

SOLUTION:

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Counter App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(),
    );
  }
}
class MyHomePage extends StatefulWidget {
  const MyHomePage({super.key});

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0; // Initialize the counter value

  void _incrementCounter() {
    setState(() {
      // Increment the counter and update the UI
      _counter++;
    });
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Counter App'),
      ),
      body: Center(
```

```
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'Counter Value:',
              style: TextStyle(fontSize: 20),
            ),
            Text(
              '$_counter', // Display the current counter value
              style: Theme.of(context).textTheme.headlineMedium,
            ),
            const SizedBox(height: 30), // Add some spacing
            ElevatedButton(
              onPressed: _incrementCounter, // Call the increment function on button press
              child: const Text('Increment Counter'),
            ),
          ],
        ),
      ),
    );
  }
}
```

OUTPUT:



EXPERIMENT – 3

AIM:

a) Design a responsive UI that adapts to different screen sizes.

b) Implement media queries and breakpoints for responsiveness.

DESCRIPTION:

LayoutBuilder:

In Flutter, LayoutBuilder is a powerful widget used to build a widget tree based on the constraints of its parent widget. It is particularly useful for creating responsive layouts that adapt to different screen sizes, orientations, or available space.

LayoutBuilder takes a builder function as a parameter. This function provides two arguments:

- BuildContext context: The build context of the widget.
- BoxConstraints constraints: An object containing the maximum and minimum width and height constraints passed down from the parent widget.

Inside the builder function, you can access the maxWidth, minWidth, maxHeight, and minHeight properties of the BoxConstraints object. This information allows you to conditionally render different layouts or adjust widget properties based on the available space.

Experiment - 3(a)

AIM: Design a responsive UI that adapts to different screen sizes.

SOLUTION:

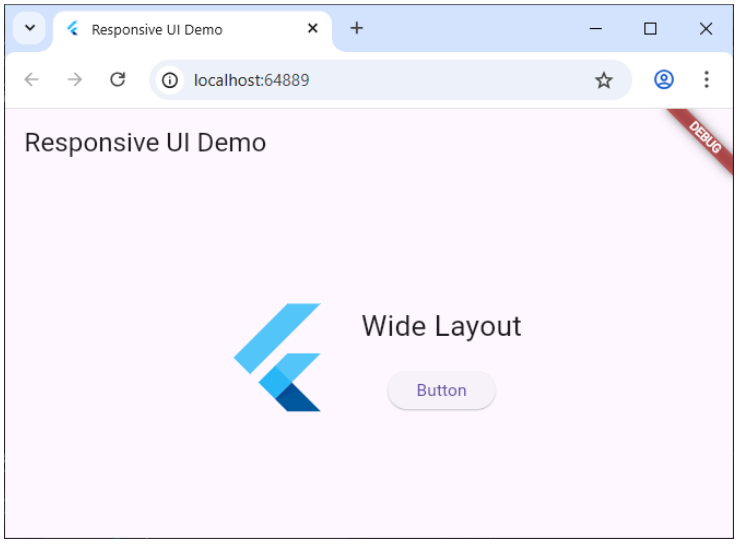
```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Responsive UI Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: ResponsiveHomePage(),
    );
  }
}
class ResponsiveHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Responsive UI Demo'),
```

```
    ),
    body: LayoutBuilder(
     builder: (BuildContext context, BoxConstraints constraints) {
      if (constraints.maxWidth < 600) {
       return _buildNarrowLayout();
      } else {
       return _buildWideLayout();
      }
     },
    ),
   );
}

Widget _buildNarrowLayout() {
  return Center(
    child: Column(
     mainAxisAlignment: MainAxisAlignment.center,
     children:
     <Widget>[
     FlutterLogo(size: 100),
     SizedBox(height: 20),
     Text('Narrow Layout',
      style: TextStyle(fontSize: 24),
     ),
     SizedBox(height: 20),
     ElevatedButton( onPressed:
        () {},
      child: Text('Button'),
     ),
    ],
    ),
  );
}

Widget _buildWideLayout() {
  return Center(
    child: Row(
     mainAxisAlignment: MainAxisAlignment.center,
     children: <Widget>[
      FlutterLogo(size: 100),
      SizedBox(width: 20),
      Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
        Text(
         'Wide Layout',
```

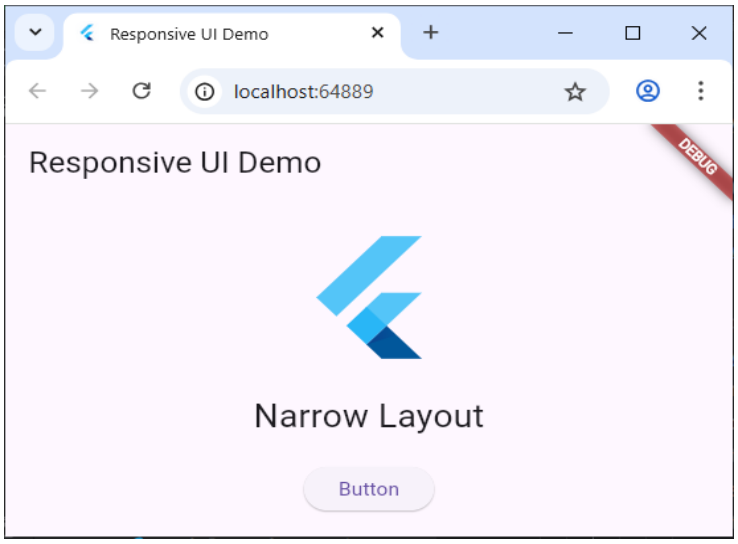
```
              style: TextStyle(fontSize: 24),
            ),
            SizedBox(height: 20),
            ElevatedButton( onPressed: ()
            {},
              child: Text('Button'),
            ),
          ],
        ),
      ],
    ),
  );
 }
}
```

OUTPUTS:

DESKTOP VIEW



MOBILE VIEW



Experiment - 3(b)

AIM: Implement media queries and breakpoints for responsiveness.

DESCRIPTION:

In Flutter, **MediaQuery** is a widget that provides information about the device's screen and user preferences. It allows developers to build responsive and adaptive user interfaces by giving access to various details about the screen's size, orientation, pixel density, and more.

It is quite easy to use MediaQuery, just use the ***of()*** method and you are good to go, the only requirement is **BuildContext.** MediaQuery can be used anywere in the widget tree where the **BuildContext** is available.

Next, in order to access the data provided by **MediaQuery.of(context)** you can create a variable of type **MediaQueryData,** let's say **mediaQuery**. Now once we have instance of MediaQueryData, just use " **mediaQuery. "** and you will be able to see everything MediaQueryData has to offer.(Assuming you are using vscode, Android Studio or IntelliJ IDE)

SOLUTION:

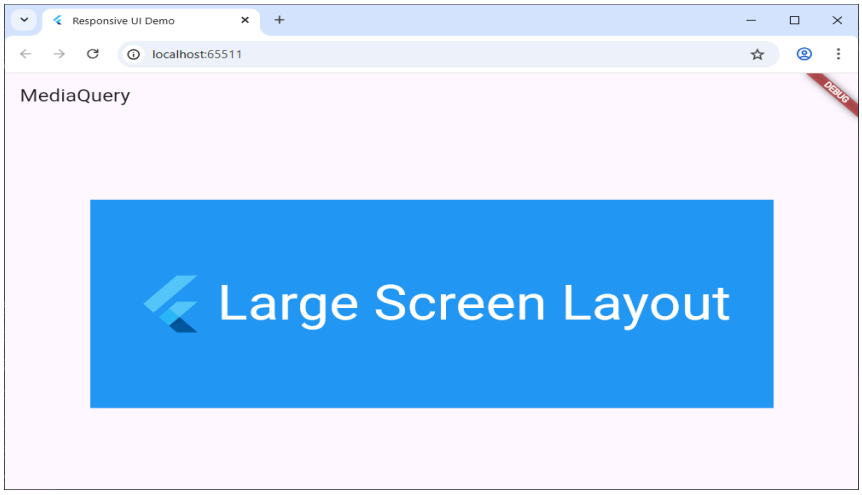
```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());

}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Responsive UI Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MediaQueryWidget(),
    );
  }
}

class MediaQueryWidget extends StatelessWidget {
  const MediaQueryWidget({super.key});
  @override
  Widget build(BuildContext context) {
    // Use media query to get the screen size
    var screenSize = MediaQuery.of(context).size;

    // Define breakpoints for different screen sizes
    double breakpointSmall = 600.0;
    double breakpointMedium = 900.0;

    // Choose the appropriate layout based on screen width
    Widget content;
    if (screenSize.width < breakpointSmall) {
      content = buildSmallLayout();
    } else if (screenSize.width < breakpointMedium) {
      content = buildMediumLayout();
```

```
  } else {
    content = buildLargeLayout();
  }

  return SafeArea(
    child: Scaffold(
      appBar: AppBar(
        title: Text('MediaQuery'),
      ),
      body: Center(
        child: Container(
          width: screenSize.width * 0.8,
          height: screenSize.height * 0.5,
          color: Colors.blue,
          child: Center(child: content),
        ),
      ),
    ),
  );
}

Widget buildSmallLayout() {
  return const Text(
    'Small Screen Layout',
    textAlign: TextAlign.center,
    style: TextStyle(fontSize: 40, color: Colors.white),
  );
}

Widget buildMediumLayout() {
  return Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
      Text(
        'Medium Screen Layout',
        textAlign: TextAlign.center,
        style: TextStyle(fontSize: 60, color: Colors.white),
      ),
      SizedBox(height: 10.0),
      FlutterLogo(size: 80),
    ],
  );
}

Widget buildLargeLayout() {
  return Row(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
      FlutterLogo(size: 80),
      SizedBox(width: 10.0),
      Text(
        'Large Screen Layout',
        textAlign: TextAlign.center,
```

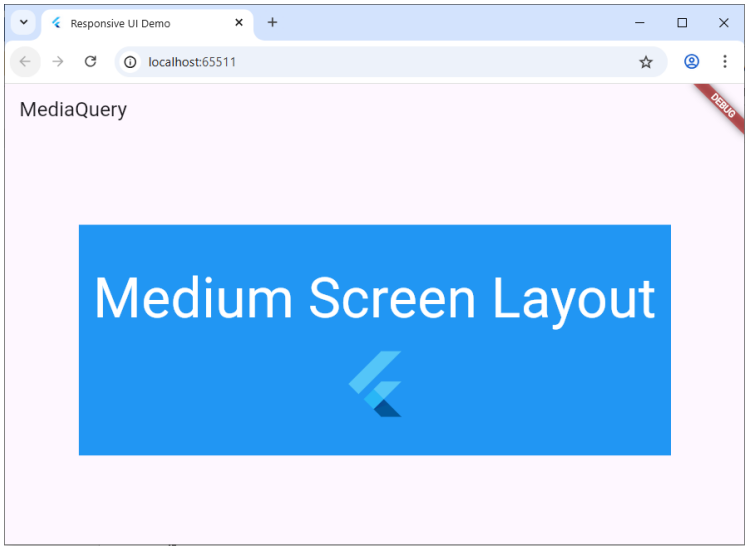
```
          style: TextStyle(fontSize: 60, color: Colors.white),
        ),
      ],
    );
  }
}
```

OUTPUTS:

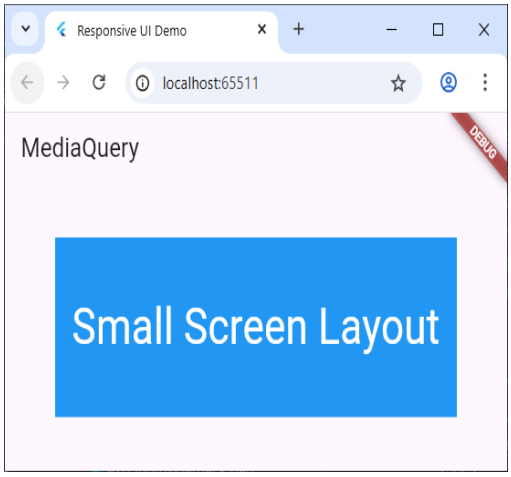
DESKTOP SCREEN SIZE



TABLET SCREEN SIZE



MOBILE SCREEN SIZE



EXPERIMENT – 4

AIM:

a) Setup navigation between different screens using navigator.

b) Implement navigation with named routes.

Experiment - 4(a)

AIM: Setup navigation between different screens using navigator.

DESCRIPTION:

In Flutter, Navigator.push() is a method used for navigating to a new screen or "route" within an application. It works by pushing a new Route onto the navigator's stack, effectively placing the new screen on top of the current one. This creates a visual transition to the new screen, and the previous screen remains in the stack, allowing for easy navigation back.

- Navigator.push() is fundamental for implementing navigation flows in Flutter applications.
- It utilizes a stack-based approach to manage the order of screens.
- MaterialPageRoute is frequently used with Navigator.push() to create routes for new screens.
- To return from a pushed screen, Navigator.pop() is used.
- To return to first screen from the current screen

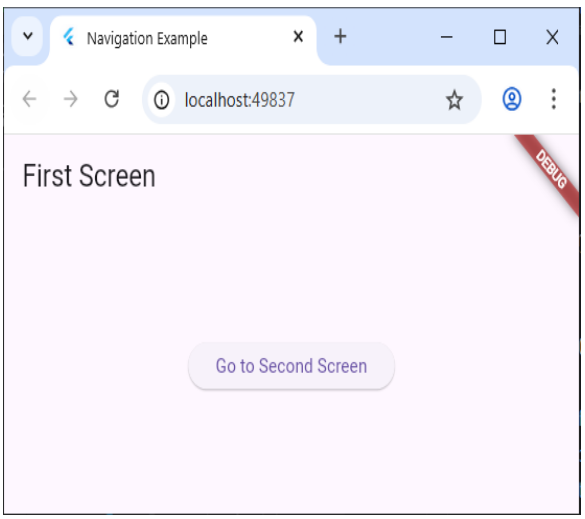
  Navigator.popUntil(context, ModalRoute.withName('/'));

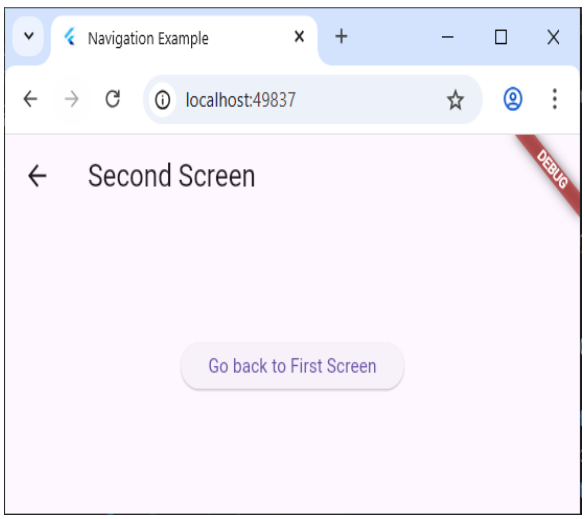
SOLUTION:

```
import 'package:flutter/material.dart';
void main()
{
 runApp(MyApp());
}
class MyApp extends StatelessWidget
{
 @override
 Widget build(BuildContext context)
 {
  return MaterialApp(
   title: 'Navigation Example',
   theme: ThemeData(
    primarySwatch: Colors.blue,
   ),
   home: FirstScreen(),
  );
 }
}
class FirstScreen extends StatelessWidget
{
 @override
 Widget build(BuildContext context)
 {
  return Scaffold(
   appBar: AppBar(
    title: Text('First Screen'),
```

```
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: ()
          {
            // Navigate to the second screen
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => SecondScreen()),
            );
            },
          child: Text('Go to Second Screen'),
        ),
      ),
    );
  }
}
class SecondScreen extends StatelessWidget
{
  @override
  Widget build(BuildContext context)
  {
    return Scaffold(
    appBar: AppBar(
    title: Text('Second Screen'),
    ),
    body: Center(
      child: ElevatedButton( onPressed: () {
      // Navigate back to the first screen
        Navigator.pop(context);
        },
        child: Text('Go back to First Screen'),
      ),
    ),
  );
  }
}
```

OUTPUT:





Experiment - 4(b)

AIM: Implement navigation with named routes.

DESCRIPTION:

Navigator.pushNamed( ) in Flutter is a method used for navigating to a new screen (route) by its registered name. This approach offers advantages over Navigator.push( ) when dealing with multiple routes, as it promotes better organization and maintainability.

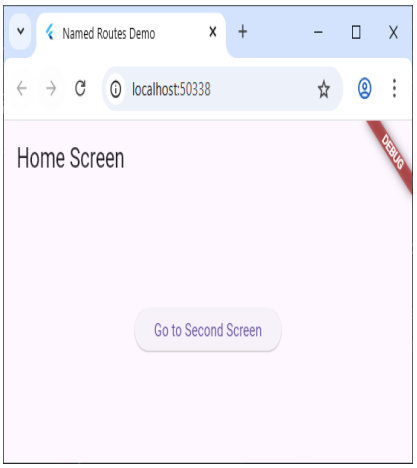
To use pushNamed, the named routes must be declared in the routes property of the MaterialApp widget. This routes property is a map where keys are the route names (strings) and values are builder functions that return the widget associated with that route.

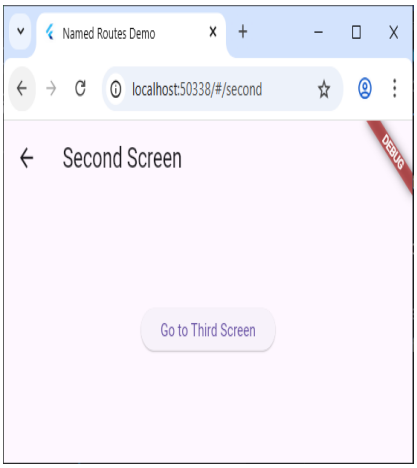
SOLUTION:

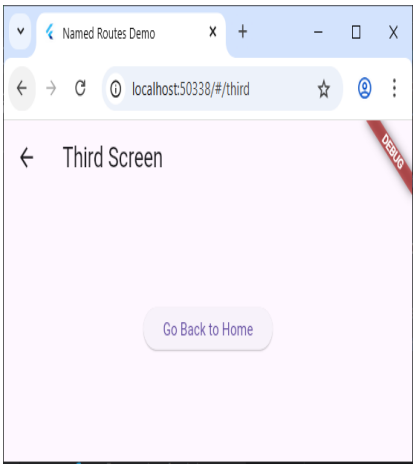
```
import 'package:flutter/material.dart';
void main()
{
 runApp(MyApp());
}
class MyApp extends StatelessWidget
{
 @override
 Widget build(BuildContext context)
 {
  return MaterialApp(
   title: 'Named Routes Demo',
   initialRoute: '/',
   routes: {
    '/': (context) => HomeScreen(),
    '/second': (context) => SecondScreen(),
    '/third': (context) => ThirdScreen(),
   },
  );
 }
}
class HomeScreen extends StatelessWidget
{
 @override
 Widget build(BuildContext context)
 {
  return
   Scaffold(
    appBar: AppBar(
     title: Text('Home Screen'),
    ),
    body: Center(
     child: ElevatedButton( onPressed: () {
      Navigator.pushNamed(context, '/second');
      },
      child: Text('Go to Second Screen'),
     ),
    ),
   );
 }
```

```
}
class SecondScreen extends StatelessWidget
{
  @override
  Widget build(BuildContext context)
  {
    return Scaffold(
      appBar: AppBar(
        title: Text('Second Screen'),
      ),
      body: Center(
        child: ElevatedButton( onPressed: () {
        Navigator.pushNamed(context, '/third');
      },
        child: Text('Go to Third Screen'),
      ),
    ),
  );
}
}
class ThirdScreen extends StatelessWidget
{
  @override
  Widget build(BuildContext context)
  {
    return Scaffold(
      appBar: AppBar(
      title: Text('Third Screen'),
      ),
      body: Center(
        child: ElevatedButton( onPressed: () {
        Navigator.popUntil(context, ModalRoute.withName('/'));
        },
          child: Text('Go Back to Home'),
        ),
      ),
    );
  }
}
```

OUTPUT:







EXPERIMENT – 5

AIM:

a) Learn about stateful and stateless widgets.

b) Implement state management using set State and Provider.

DESCRIPTION:

What are Widgets?

Each element on the screen of the Flutter app is a widget. The view of the screen completely depends upon the choice and sequence of the widgets used to build the apps. The structure of the code of apps is a tree of widgets.

Types of Widgets

There are broadly two types of widgets in the flutter:

- Stateless Widget
- Stateful Widget

1. Stateless Widget

Stateless Widget is a type of widget which once built, then it's properties and state can't be changed. These widgets are immutable, once created can't be modified.

Note: *These are used for static content or UI content that don't need a change after time.*

Key Characteristics of Stateless Widgets are: Immutable, No State and Lightweight.

Examples: Display Text, Icons, Images, etc.

2. Stateful Widget

Stateful Widgets is a type of widget that can change state. It can maintain and update the appearance in the response to change in state.

Note: *These are used for dynamic change in the properties and appearance over the time.*

Key Characteristics of Stateful Widgets are: Mutable State, State Lifecycle and Dynamic Updates.

Examples: Buttons, Sliders, Text Fields, etc.

Experiment - 5(a)

AIM: Learn about stateful and stateless widgets.

SOLUTION:

Stateless Multi-Card Widget

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
```

```
      home: Scaffold(
        appBar: AppBar(title: const Text('Stateless Multi-Card Example')),
        body: CardListScreen(),
      ),
    );
  }
}

class CardListScreen extends StatelessWidget {
  CardListScreen({super.key});
  final List<bool> _cardSelections = List.generate(5, (index) => false);
  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: 5,
      itemBuilder: (context, index) {
        return Card(
          margin: const EdgeInsets.all(8.0),
          color: _cardSelections[index] ? Colors.blueAccent.shade100 : Colors.white,
          elevation: _cardSelections[index] ? 8.0 : 2.0,
          child: Padding(
            padding: const EdgeInsets.all(16.0),
            child: Column(
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
                Text(
                  'Card ${index + 1}',
                  style: const TextStyle(
                    fontSize: 20,
                    fontWeight: FontWeight.bold,
                  ),
                ),
                const SizedBox(height: 8.0),
                Text(
                  _cardSelections[index]
                      ? 'This card is selected!'
                      : 'Tap to select this card.',
                  style: TextStyle(
                    color: _cardSelections[index] ? Colors.white : Colors.black87,
                  ),
                ),
              ],
            ),
          ),
        );
      },
```

```
    );
  }
}
```

Output:

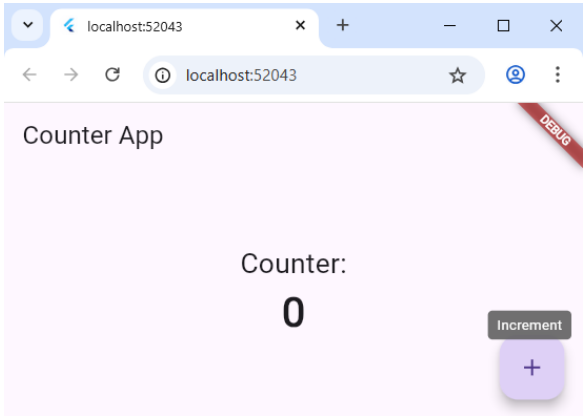
Stateful Widgets:

Stateful button-widget

```
import 'package:flutter/material.dart';
void main()
{
  runApp(MyApp());
}
class MyApp extends StatelessWidget
{ @override
Widget build(BuildContext context)
{
  return MaterialApp(
  home: CounterApp(),
  );
}
}
class CounterApp extends StatefulWidget
{ @override
_CounterAppState createState() => _CounterAppState();
}
class _CounterAppState extends State<CounterApp>
{
  int _counter = 0;
  void _incrementCounter()
  {
```

```
    setState(() {
  _counter++;});
  }
@override
Widget build(BuildContext context)
{
  return
  Scaffold (
    appBar: AppBar(
      title: Text('Counter App'),
    ),
    body: Center(
      child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text( 'Counter:',
            style: TextStyle(fontSize: 24),
          ),
          Text( '$_counter',
            style: TextStyle(fontSize: 36, fontWeight: FontWeight.bold),
          ),
        ],
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed:
      _incrementCounter, tooltip: 'Increment',
      child: Icon(Icons.add),
    ),
  );
}
}
```

OUTPUT:



Experiment - 5(b)

AIM: Implement state management using setState and Provider.

SOLUTION:

Stateful Multi-Card Widget using setState():

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text('Stateful Multi-Card Example')),
        body: const CardListScreen(),
      ),
    );
  }
}

class CardListScreen extends StatefulWidget {
  const CardListScreen({super.key});
  @override
  State<CardListScreen> createState() => _CardListScreenState();
}

class _CardListScreenState extends State<CardListScreen> {
  // A list to store the selection state of each card
  final List<bool> _cardSelections = List.generate(5, (index) => false);
  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: _cardSelections.length,
      itemBuilder: (context, index) {
        return GestureDetector(
          onTap: () {
            setState(() {
              // Toggle the selection state of the tapped card
              _cardSelections[index] = !_cardSelections[index];
            });
          },
          child: Card(
            margin: const EdgeInsets.all(8.0),
            color: _cardSelections[index] ? Colors.blueAccent.shade100 : Colors.white,
            elevation: _cardSelections[index] ? 8.0 : 2.0,
            child: Padding(
              padding: const EdgeInsets.all(16.0),
              child: Column(
```

```
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              Text(
                'Card ${index + 1}',
                style: const TextStyle(
                  fontSize: 20,
                  fontWeight: FontWeight.bold,
                ),
              ),
              const SizedBox(height: 8.0),
              Text(
                _cardSelections[index]
                    ? 'This card is selected!'
                    : 'Tap to select this card.',
                style: TextStyle(
                  color: _cardSelections[index] ? Colors.white : Colors.black87,
                ),
              ),
            ],
          ),
        ),
      ),
    );
  },
 );
 }
}
```

OUTPUT:

State Management using provider package:

ChangeNotifier

It is a class that provides **notifications for changes to its listeners**. It is a simpler way to use for a small number of listeners. It uses the **notifyListeners()** method to notify its listeners about changes in the model.

For example, let us create a class **CounterModel** which will extend **ChangeNotifier**, and it will have a function **increment()** which will increment the counter and notify its listeners about the changes using **notifyListeners()**, and UI will get updated.

Step 1: Add Provider to Your Project

- First, add the Provider package to your pubspec.yaml file:

```
dependencies:
  flutter:
    sdk: flutter
  provider: ^6.1.4
```

- Then, run flutter pub get to install the package.

Step 2: Create a simple model class to hold your app's state. For this example, counter_model.dart contains a class named CounterModel that extends ChangeNotifier

counter_model.dart:

```
import 'package:flutter/material.dart';
class CounterModel with ChangeNotifier {
  int _count = 0;
  int get count => _count;
  void increment() {
    _count++;
    notifyListeners(); // Notify listeners that the state has changed
  }
}
```

Step 3: Set Up the Provider

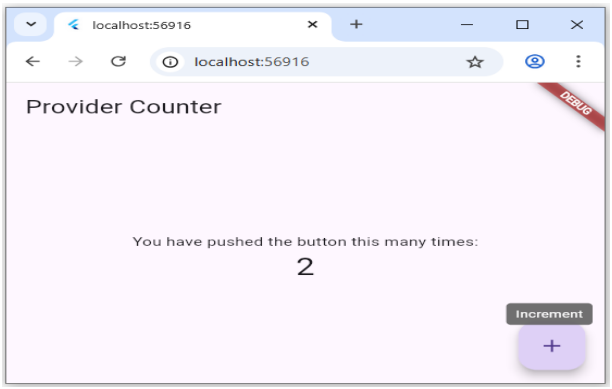
Wrap your app with a ChangeNotifierProvider to provide the Counter model to the widget tree:

exp5bproviderpack.dart:

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'counter_model.dart'; // Assuming counter_model.dart contains CounterModel
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CounterModel(),
      child: const MyApp(),
    ),
  );
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
```

```
      home: const CounterScreen(),
    );
  }
}
class CounterScreen extends StatelessWidget {
  const CounterScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Provider Counter'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ),
            Consumer<CounterModel>( // Use Consumer to rebuild only the necessary part
              builder: (context, counter, child) {
                return Text(
                  '${counter.count}',
                  style: Theme.of(context).textTheme.headlineMedium,
                );
              },
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          Provider.of<CounterModel>(context, listen: false).increment();
        },
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Output:



EXPERIMENT – 6

AIM:
a) Create custom widgets for specific UI elements.
b) Apply styling using themes and custom styles.

Experiment - 6(a)

AIM: Create custom widgets for specific UI elements.

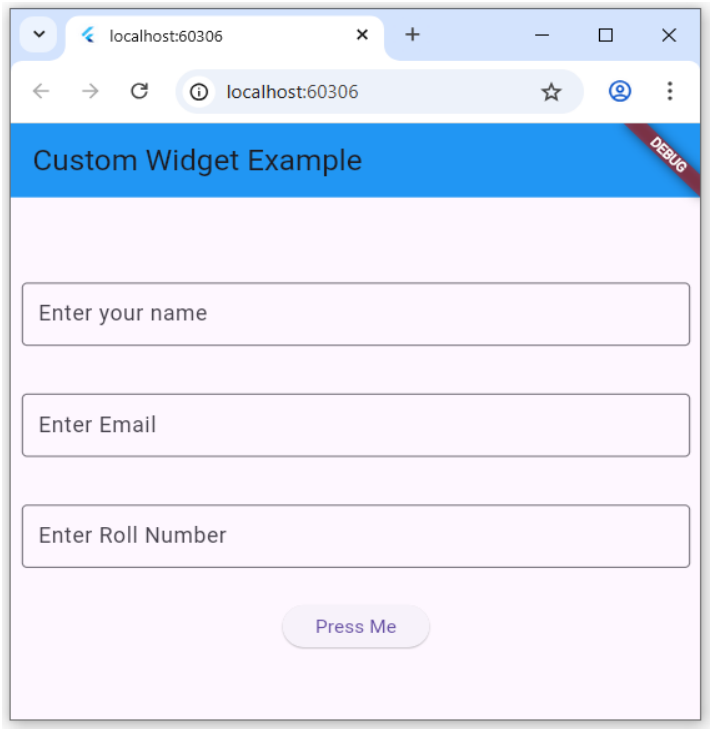
SOLUTION:

```
import 'package:flutter/material.dart';
void main()
{
 runApp(MyApp());
}
class MyApp extends StatelessWidget
{
 @override
 Widget build(BuildContext context)
 {
  return MaterialApp(
   home: Scaffold(
    appBar: AppBar(
    title: Text('Custom Widget Example'),
    backgroundColor: Colors.blue,
    ),
    body: Column(
     mainAxisAlignment: MainAxisAlignment.center, children:
    <Widget>[
     Padding(
      padding: const EdgeInsets.all(8.0),
      child: CustomTextField(
      hintText: 'Enter your name',
      onChanged: (value) { print('Name changed: $value'); },
      ),
     ),
     SizedBox(height: 20),
     Padding(
      padding: const EdgeInsets.all(8.0),
      child: CustomTextField(
       hintText: 'Enter Email',
       onChanged: (value) { print('Name changed: $value');},
      ),
     ),
     SizedBox(height: 20),
     Padding(
      padding: const EdgeInsets.all(8.0),
```

```
            child: CustomTextField(
              hintText: 'Enter Roll Number',
              onChanged: (value) { print('Name changed: $value');},
            ),
          ),
          SizedBox(height: 20),
          CustomButton(
            text: 'Press Me',
            onPressed: () { print('Button pressed!'); },
          ),
        ],
        ),
      ),
    );
  }
}
class CustomButton extends StatelessWidget
{
  final String? text;
  final VoidCallback? onPressed;
  const CustomButton({ Key? key, required this.text, required this.onPressed,
  }) : super(key: key);
  @override
  Widget build(BuildContext context)
  {
    return ElevatedButton(
      onPressed: onPressed,
      child: Text(text!),
    );
  }
}
class CustomTextField extends StatelessWidget
{
  final String hintText;
  final ValueChanged<String> onChanged;
  const CustomTextField({ Key? key, required this.hintText,
    required this.onChanged,}) : super(key: key);
  @override
  Widget build(BuildContext context)
  {
    return TextField(
      onChanged: onChanged,
      decoration: InputDecoration(
        hintText: hintText,
        border: OutlineInputBorder(),
      ),
```

```
    );
  }
}
```

OUTPUT:


localhost:60306
localhost:60306
Custom Widget Example
DEBUG
Enter your name
Enter Email
Enter Roll Number
Press Me

Experiment - 6(b)

AIM: Apply styling using themes and custom styles.

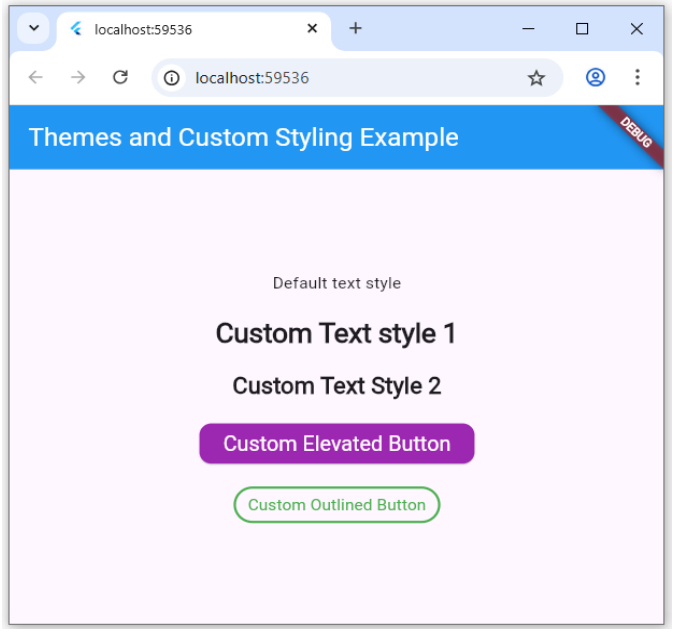
DESCRIPTION:
In Flutter, you can apply styling to your widgets using themes and custom styles to maintain consistency and make your UI more visually appealing.

SOLUTION:

```
import 'package:flutter/material.dart';
void main()
{ runApp(MyApp());
}
class MyApp extends StatelessWidget
{ @override
Widget build(BuildContext context)
{ return MaterialApp(
  theme: ThemeData(
   // Define the overall theme of the app
    primaryColor: Colors.blue,
    fontFamily: 'Roboto',
       // Text Theme
    textTheme: TextTheme(
      displayLarge: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),
      headlineMedium: TextStyle(fontSize: 20, fontWeight: FontWeight.bold),
      bodyLarge: TextStyle(fontSize: 16),
    ),
        // Elevated Button Theme
    elevatedButtonTheme: ElevatedButtonThemeData(
      style: ElevatedButton.styleFrom(
        backgroundColor: Colors.purple,
        foregroundColor: Colors.white,
        textStyle: TextStyle(fontSize: 18),
        padding: EdgeInsets.symmetric(horizontal: 20, vertical: 15),
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(10),
        ),
      ),
    ),
       // Outlined Button Theme
    outlinedButtonTheme: OutlinedButtonThemeData(
      style: OutlinedButton.styleFrom(
        foregroundColor: Colors.green, // Text/Icon color
        side: const BorderSide(color: Colors.green, width: 2), // Border
        padding: const EdgeInsets.all(12),
      ),
    ),
  ),
  home: HomePage(),
);
  }
}
class HomePage extends StatelessWidget {
```

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Themes and Custom Styling Example'),
        backgroundColor: Colors.blue,
        foregroundColor: Colors.white,
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children:
          <Widget>[
            Text('Default text style'),
            SizedBox(height: 20),
            Text('Text style 1', style: Theme.of(context).textTheme.displayLarge),
            SizedBox(height: 20),
            Text('Text Style 2', style: Theme.of(context).textTheme.headlineMedium),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: () {},
              child: Text('Custom Elevated Button'),
            ),
            SizedBox(height: 20),
            OutlinedButton(
              onPressed: () {},
              child: const Text('Custom Outlined Button'),
            ),
          ],
        ),
      ),
    );
  }
}
```

OUTPUT:



EXPERIMENT – 7

AIM:
a) Design a form with various input fields.
b) Implement form validation and error handling.

Experiment - 7(a)

AIM: Design a form with various input fields.

DESCRIPTION:
Design a form with various input fields such as text fields, checkboxes, radio buttons, and a dropdown menu.

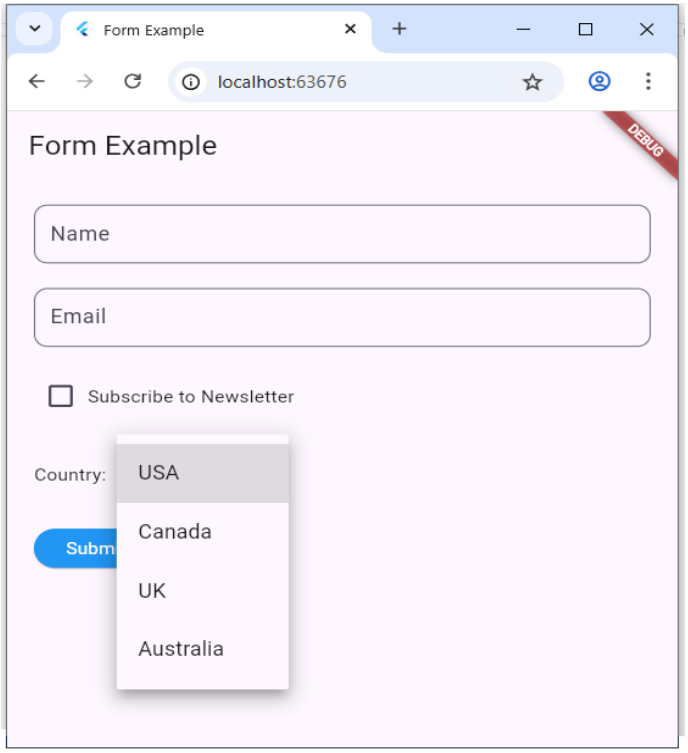
SOLUTION:

```
import 'package:flutter/material.dart';
void main()
{
  runApp(MyApp());
}
class MyApp extends StatelessWidget
{
  @override
  Widget build(BuildContext context)
  {
    return MaterialApp(
      title: 'Form Example',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: FormPage(),
    );
  }
}
class FormPage extends StatefulWidget {
  const FormPage({super.key});
  @override
  State<FormPage> createState() => _FormPageState();
}

class _FormPageState extends State<FormPage>
{
  final _formKey = GlobalKey<FormState>();
  String ? _name;
  String ? _email;
  bool ? _subscribeToNewsletter = false;
  String ? _selectedCountry = 'USA';
  @override
  Widget build(BuildContext context)
  {
    return Scaffold(
      appBar: AppBar(
        title: Text('Form Example'),
```

```
),
body: Padding(
  padding: EdgeInsets.all(20.0),
  child: Form(
    key: _formKey,
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: <Widget>[
        TextFormField(
          decoration: InputDecoration(
            border: OutlineInputBorder(
              borderRadius: BorderRadius.circular(10.0),
            ),
            labelText: 'Name'),
          onSaved: (value) { _name = value;  },
        ),
        SizedBox(height: 20),
        TextFormField(
          decoration: InputDecoration(
            border: OutlineInputBorder(
              borderRadius: BorderRadius.circular(10.0),
            ),
            labelText: 'Email'
          ),
          onSaved: (value) { _email = value; },
        ),
        SizedBox(height: 20),
        Row(
          children: <Widget>[
            Checkbox(
              value: _subscribeToNewsletter,
              onChanged: (value) {
                setState(() { _subscribeToNewsletter = value; }
                );
                },
            ),
            Text('Subscribe to Newsletter'),
          ],
        ),
        SizedBox(height: 20),
        Row(
          children: <Widget>[
            Text('Country: '),
            SizedBox(width: 20),
            DropdownButton<String>(
              value: _selectedCountry,
              onChanged: (value) { setState(() {
                _selectedCountry = value; });
                },
              items: <String>['USA', 'Canada', 'UK', 'Australia']
                  .map<DropdownMenuItem<String>>((String value)
              { return DropdownMenuItem<String>(
                value: value,
```

```
                child: Text(value),
              );
              }).toList(),
            ),
          ],
        ),
        SizedBox(height: 20),
        ElevatedButton(
          style: ElevatedButton.styleFrom(
            backgroundColor: Colors.blue,
            foregroundColor: Colors.white,
          ),
            onPressed: () {
            _formKey.currentState!.save();
            // Submit the form data
            print('Name: $_name');
            print('Email: $_email');
            print('Subscribe to Newsletter: $_subscribeToNewsletter');
            print('Country: $_selectedCountry');
            },

          child: Text('Submit'),
        ),
      ],
    ),
  ),
 );
 }
}
```

OUTPUT:



Experiment - 7(b)

AIM: Implement form validation and error handling.

SOLUTION:

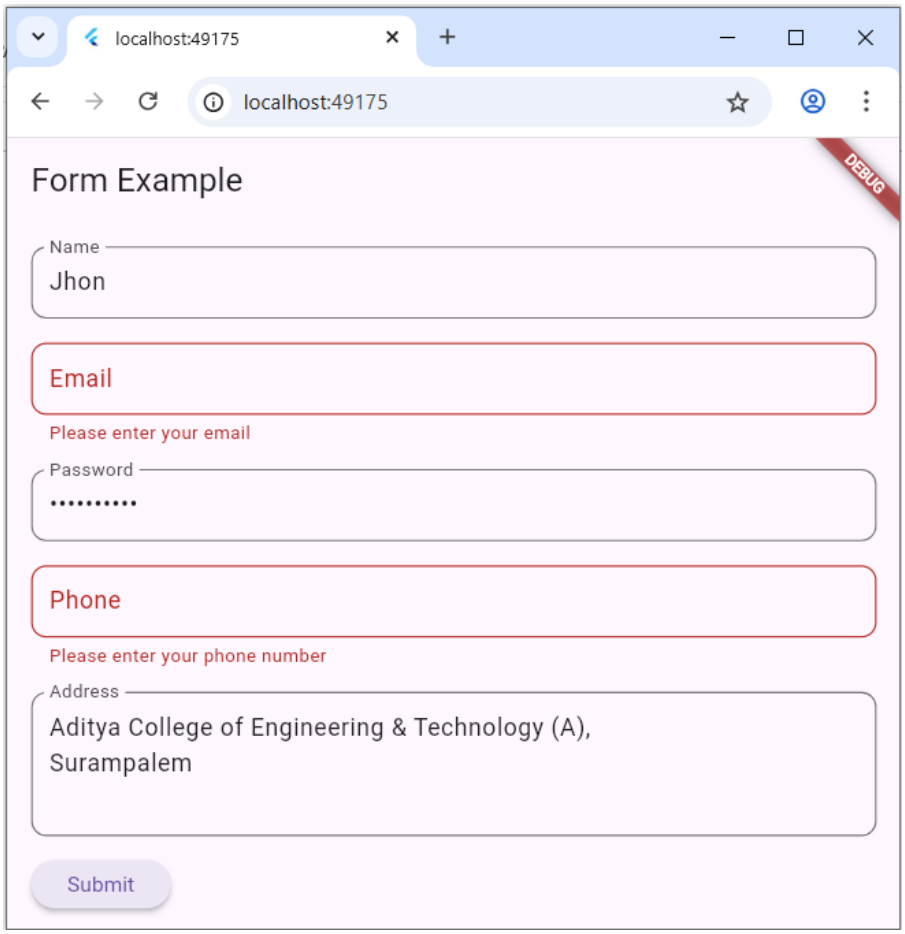
```
import 'package:flutter/material.dart';
void main()
{
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Form Example'),
        ),
        body: SingleChildScrollView(
          padding: EdgeInsets.all(16),
          child: FormWidget(),
        ),
      ),
    );
  }
}
class FormWidget extends StatefulWidget
{
  const FormWidget({super.key});
  @override
  State<FormWidget> createState() => _FormWidgetState();
}

class _FormWidgetState extends State<FormWidget>
{
  final _formKey = GlobalKey<FormState>();
  String ? _name;
  String ? _email;
  String ? _password;
  String ? _phone;
  String ? _address;
  @override
  Widget build(BuildContext context)
  {
    return Form(
      key: _formKey,
      child: Column(
```

```
crossAxisAlignment: CrossAxisAlignment.start,
children: <Widget>[
 TextFormField(
  decoration: InputDecoration(
     border: OutlineInputBorder(
       borderRadius: BorderRadius.circular(10.0),
     ),
     labelText: 'Name'),
  validator: (value) {
   if (value!.isEmpty) {
    return 'Please enter your name';
   }
   return null;
   },
  onSaved: (value) => _name = value,
 ),
 SizedBox(height: 16),
 TextFormField(
  decoration: InputDecoration(
     border: OutlineInputBorder(
       borderRadius: BorderRadius.circular(10.0),
     ),
     labelText: 'Email'
  ),
  keyboardType: TextInputType.emailAddress,
  validator: (value)
  {
   if (value!.isEmpty) {
    return 'Please enter your email';
   }
   // Add more complex email validation logic if needed
   return null;
  },
  onSaved: (value) => _email = value,
 ),
 SizedBox(height: 16),
 TextFormField(
  decoration: InputDecoration(
     border: OutlineInputBorder(
       borderRadius: BorderRadius.circular(10.0),
     ),
     labelText: 'Password'
  ),
  obscureText: true,
  validator: (value)
  {
```

```
    if (value!.isEmpty) {
      return 'Please enter a password';
    }
    // Add more complex password validation logic if needed
    return null;
    },
  onSaved: (value) => _password = value,
),
SizedBox(height: 16),
TextFormField(
  decoration: InputDecoration(
      border: OutlineInputBorder(
        borderRadius: BorderRadius.circular(10.0),
      ),
      labelText: 'Phone'),
  keyboardType: TextInputType.phone,
  validator: (value)
  {
    if (value!.isEmpty) {
      return 'Please enter your phone number';
    }
    // Add more complex phone number validation logic if needed
    return null;
    },
  onSaved: (value) => _phone = value,
),
SizedBox(height: 16),
TextFormField(
  decoration: InputDecoration(
      border: OutlineInputBorder(
        borderRadius: BorderRadius.circular(10.0),
      ),
      labelText: 'Address'),
  maxLines: 3,
  validator: (value)
  {
    if (value!.isEmpty)
    {
      return 'Please enter your address';
    }
    return null;
    },
  onSaved: (value) => _address = value,
),
SizedBox(height: 16),
ElevatedButton(
```

```
            onPressed: _submitForm,
            child: Text('Submit'),
      ),
    ],
),
);
}
void _submitForm() {
  if (_formKey.currentState!.validate()) {
    _formKey.currentState!.save();
    // Perform form submission with the saved form data
    print('Form submitted:');
    print('Name: $_name');
    print('Email: $_email');
    print('Password: $_password');
    print('Phone: $_phone');
    print('Address: $_address');
    }
  }
}
```

OUTPUT:



Experiment - 8

AIM:
a) Add animations to UI elements using flutter's animation framework.
b) Experiment with different types of animations like fade, slide, etc.

DESCRIPTION:
Flutter's animation framework offers both implicit and explicit animations to enhance UI elements.

- **Implicit animations:** are suitable for simple property changes and when you prioritize ease of use. Examples: AnimatedContainer, AnimatedOpacity, AnimatedPositioned, AnimatedCrossFade).
- **Explicit animations:** are necessary for complex, custom animations, sequential animations, or when you need precise control over the animation's behaviour. Explicit animations are also prebuilt animation effects, but require an Animation object in order to work.
  Examples: SizeTransition, ScaleTransition or PositionedTransition.

Key Components:

- **Animation**: is a class that represents a running or stopped animation, and is composed of a value representing the target value the animation is running to, and the status, which represents the current value the animation is displaying on screen at any given time. It is a subclass of ***Listenable***, and notifies its listeners when the status changes while the animation is running.
- **AnimationController:** Manages the animation's duration, playback (forward, reverse, repeat), and status. Requires a *TickerProviderStateMixin* in the StatefulWidget.
- **Tween:** Defines the range of values an animated property will transition between (e.g., Tween<double>(begin: 0.0, end: 1.0)).
- **CurvedAnimation:** Applies a non-linear curve to the animation's progress, creating different easing effects (e.g., Curves.easeOut).
- **AnimatedBuilder** or **AnimatedWidget:** Rebuilds the UI based on the current animation value without rebuilding the entire widget tree, optimizing performance.

Experiment - 8(a)

AIM: Add animations to UI elements using flutter's animation framework.

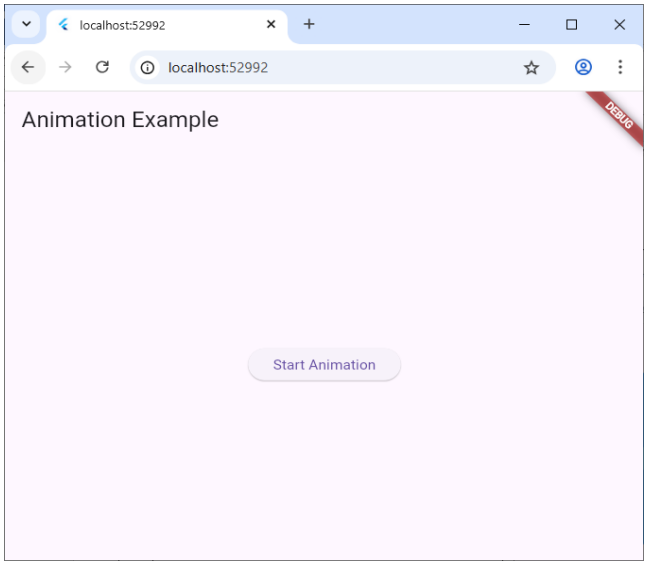
SOLUTION:

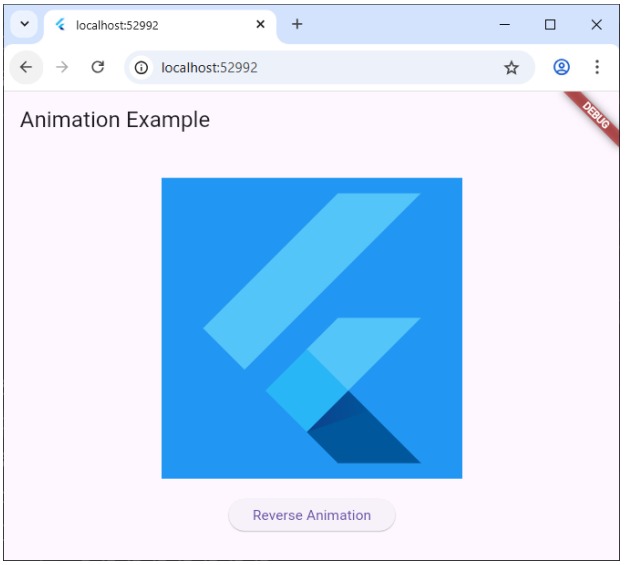
```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget
{
  @override
  Widget build(BuildContext context)
  {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Animation Example'),
        ),
        body: AnimationWidget(),
      ),
    );
  }
```

```
}
class AnimationWidget extends StatefulWidget
{
  @override
  _AnimationWidgetState createState() => _AnimationWidgetState();
}

class _AnimationWidgetState extends State<AnimationWidget>
    with SingleTickerProviderStateMixin {
  late AnimationController _controller;
  late Animation<double> _animation;
  @override
  void initState()
  {
    super.initState();
    _controller = AnimationController(
      duration: Duration(seconds: 1),
      vsync: this,
    );
    _animation = Tween<double>(begin: 0, end: 300).animate(_controller)
      ..addListener(()
      {
        setState(() {}); // Trigger rebuild when animation value changes
      });
}
@override
Widget build(BuildContext context)
{
  return Center(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Container(
          width:  _animation.value,
          height: _animation.value,
          color: Colors.blue,
          child: FlutterLogo(size: 100),
        ),
        SizedBox(height: 20),
        ElevatedButton(
          onPressed: ()
          {
            if (_controller.status == AnimationStatus.completed)
            { // Restart animation
              _controller.reverse();
            }
            else
            { // Start animation
              _controller.forward();
            }
          },
          child: Text(
            _controller.status == AnimationStatus.completed
```

```
            ? 'Reverse Animation'
            : 'Start Animation',
          ),
        ),
      ],
    ),
  );
}
@override
void dispose()
{ // Clean up controller when widget is disposed
  _controller.dispose();
  super.dispose();
}
}
```

Output:





Experiment - 8(b)

AIM: Experiment with different types of animations like fade, slide, etc.

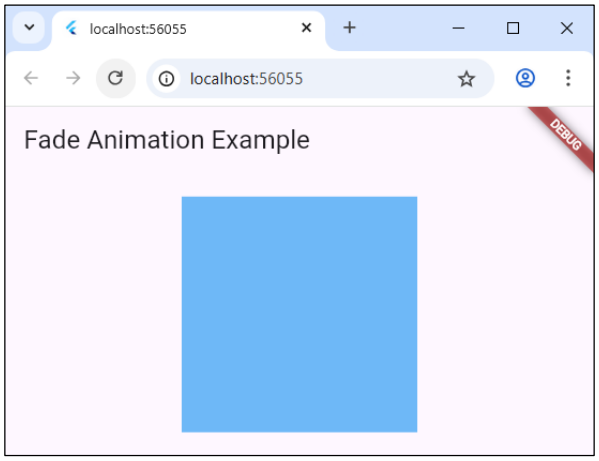
SOLUTION:

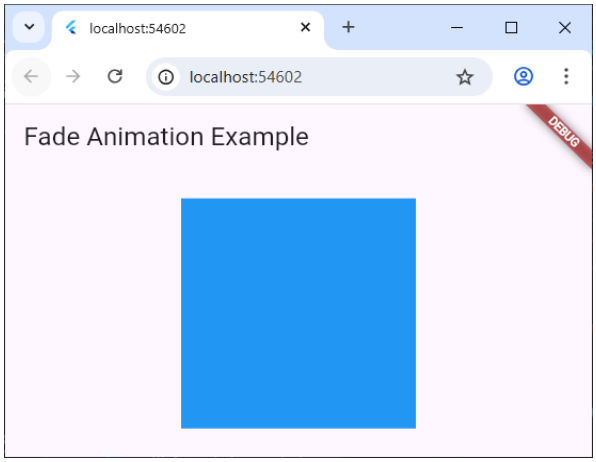
Fade Animation:

```
import 'package:flutter/material.dart';
void main()
{
  runApp(MyApp());
}
class MyApp extends StatelessWidget
{
  @override
  Widget build(BuildContext context)
  {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Fade Animation Example'),
        ),
        body: FadeAnimation(),
      ),
    );
  }
}
class FadeAnimation extends StatefulWidget
{
  @override
  _FadeAnimationState createState() => _FadeAnimationState();
}
class _FadeAnimationState extends State<FadeAnimation>
    with SingleTickerProviderStateMixin {
  late AnimationController _controller;
  late Animation<double> _animation;
  @override
  void initState()
  {
    super.initState();
    _controller = AnimationController(
      duration: Duration(seconds: 2),
      vsync: this,
    );
    _animation = Tween<double>( begin: 0, end: 2).animate(_controller);
    _controller.forward(); // Start animation automatically
  }
  @override
  Widget build(BuildContext context)
  {
    return
      Center(
        child: FadeTransition(
          opacity: _animation,
```

```
          child: Container(
            width: 200,
            height: 200,
            color: Colors.blue,
          ),
        ),
      );
  }
  @override
  void dispose()
  { // Clean up controller when widget is disposed
    _controller.dispose();
    super.dispose();
  }
}
```

Output:



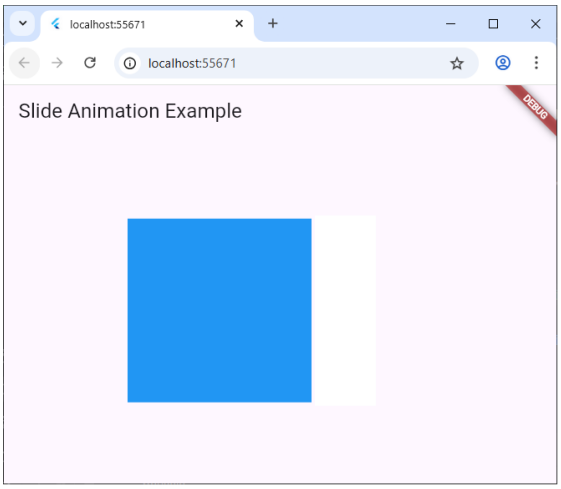


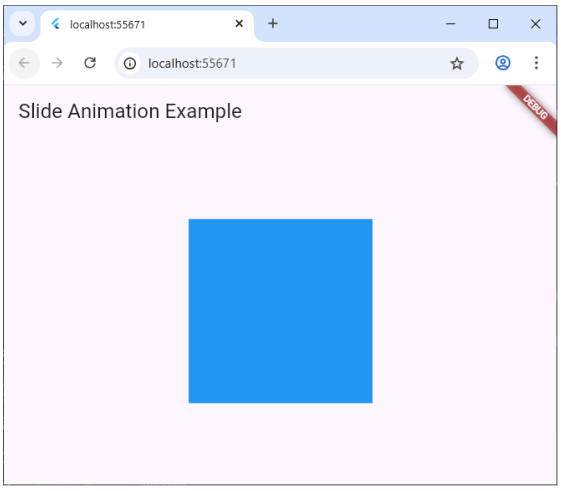
Slide Animation:

```
import 'package:flutter/material.dart';
void main()
{
  runApp(MyApp());
}
class MyApp extends StatelessWidget
{
  @override
  Widget build(BuildContext context)
  {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Slide Animation Example'),
        ),
        body: SlideAnimation(),
      ),
    );
  }
}
class SlideAnimation extends StatefulWidget
{
  @override
  _SlideAnimationState createState() => _SlideAnimationState();
}
class _SlideAnimationState extends State<SlideAnimation>
    with SingleTickerProviderStateMixin {
  late AnimationController _controller;
  late Animation<Offset> _animation;
  @override
  void initState()
  {
    super.initState();
    _controller = AnimationController(
      duration: Duration(seconds: 2),
      vsync: this,
    );
    _animation = Tween<Offset>(
        begin: Offset(-1.0, 0.0),
        end: Offset(0.0, 0.0)
    ).animate(_controller);
    _controller.forward(); // Start animation automatically
  }
  @override
  Widget build(BuildContext context)
  {
    return
      Center(
        child: SlideTransition(
          position: _animation,
          child: Container(
            width: 200,
```

```
            height: 200,
            color: Colors.blue,
          ),
        ),
      );
  }
  @override
  void dispose()
  { // Clean up controller when widget is disposed
    _controller.dispose();
    super.dispose();
  }
}
```

Output:



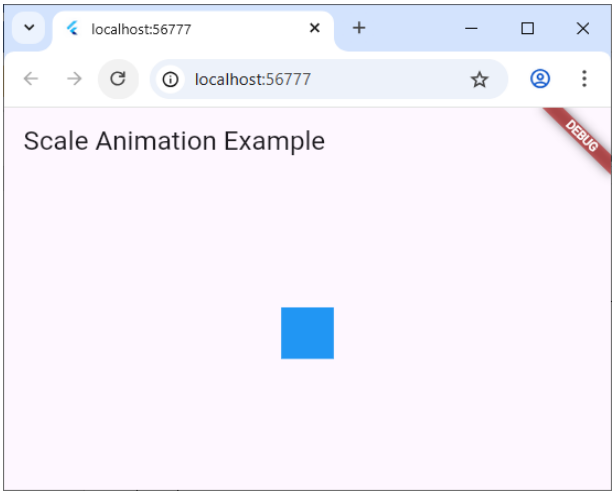


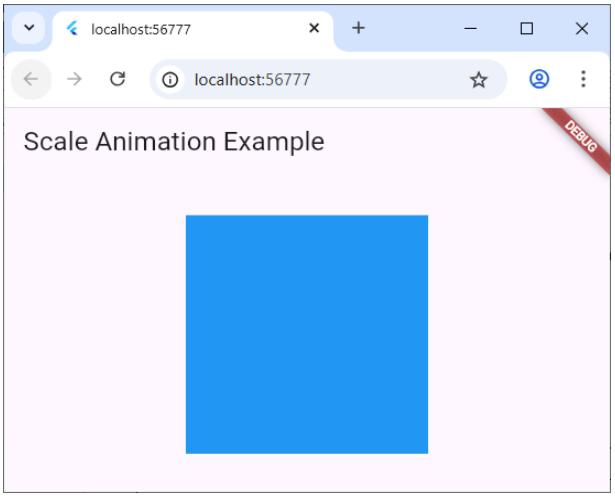
Scale Animation:

```
import 'package:flutter/material.dart';
void main()
{
  runApp(MyApp());
}
class MyApp extends StatelessWidget
{
  @override
  Widget build(BuildContext context)
  {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Scale Animation Example'),
        ),
        body: ScaleAnimation(),
      ),
    );
  }
}
class ScaleAnimation extends StatefulWidget
{
  @override
  _ScaleAnimationState createState() => _ScaleAnimationState();
}
class _ScaleAnimationState extends State<ScaleAnimation>
    with SingleTickerProviderStateMixin {
  late AnimationController _controller;
  late Animation<double> _animation;
  @override
  void initState()
  {
    super.initState();
    _controller = AnimationController(
      duration: Duration(seconds: 2),
      vsync: this,
    );
    _animation = Tween<double>( begin: 0.0, end: 1.0).animate(_controller);
    _controller.forward(); // Start animation automatically
  }
  @override
  Widget build(BuildContext context)
  {
    return Center(
      child: ScaleTransition(
        scale: _animation,
        child: Container(
          width: 200,
          height: 200,
          color: Colors.blue,
        ),
      ),
```

```
    );
  }
  @override
  void dispose()
  { // Clean up controller when widget is disposed
    _controller.dispose();
    super.dispose();
  }
}
```

Output:





Experiment - 9

AIM:
a) Fetch data from REST API.
b) Display the fetched data in a meaningful way in the UI.

DESCRIPTION:
A REST API in Flutter refers to the integration and utilization of a Representational State Transfer (REST) Application Programming Interface (API) within a Flutter application. APIs are often used to fetch data from a server or a database. HTTP requests, specifically GET requests, are commonly employed to retrieve data from APIs. When a Flutter app sends a GET request to an API endpoint, it receives a response containing the requested data, typically in JSON format.

Flutter applications use the http package (or other third-party packages like Dio) to make HTTP requests to a REST API. The process typically involves:

- **Making HTTP Requests:**
  Using methods like http.get(), http.post(), http.put(), or http.delete() to send requests to specific API endpoints.
- **Handling Responses:**
  The server responds with data, usually in JSON format. Flutter applications parse this JSON data into Dart objects.
- **Updating UI:**
  The parsed data is then used to update the user interface of the Flutter application, displaying information fetched from the server or reflecting changes made through API calls.

Fetch Data from a REST API in Flutter:
STEP-1: Include the http package in your pubspec.yaml file

```
dependencies:
      flutter:
        sdk: flutter
      http: ^1.2.1 # Use the latest version
```

STEP-2: run `flutter pub get` in your terminal. Import the http package.

STEP-3: Create a Dart file with http packages imported and an asynchronous function to fetch data that makes the HTTP GET request and handles the response.

Experiment – 9(a)

AIM: Fetch data from REST API.

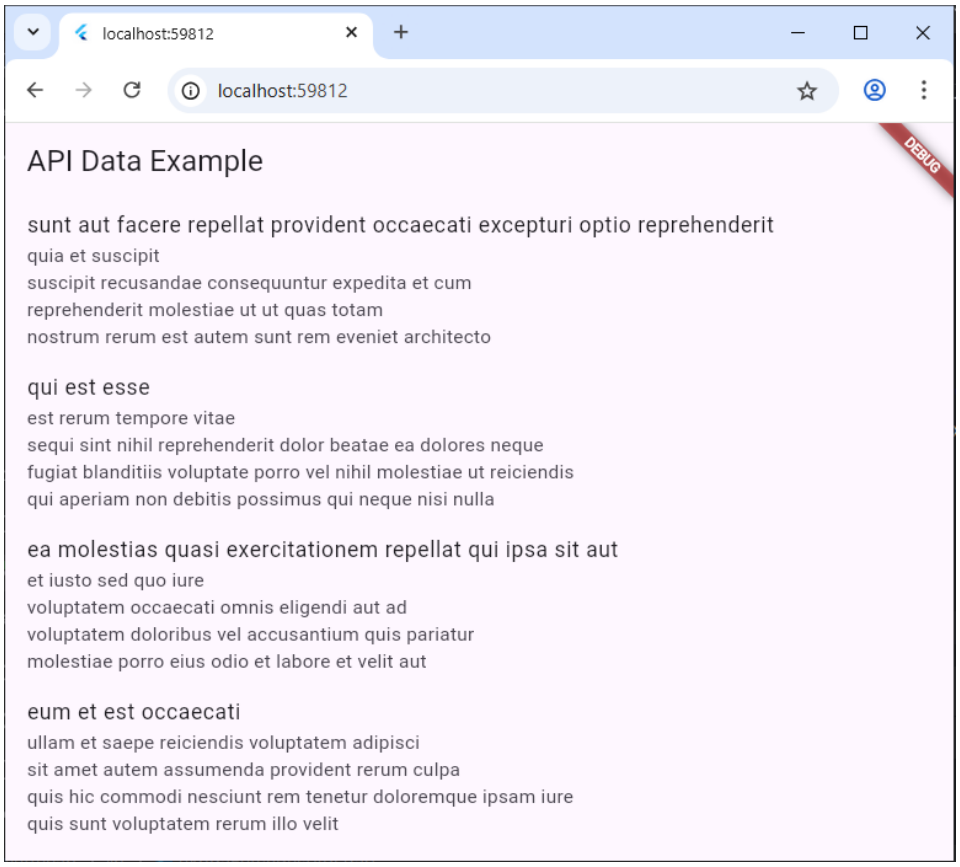
SOLUTION:

```
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
void main()
{
 runApp(MyApp());
}
class MyApp extends StatelessWidget
{
 @override
 Widget build(BuildContext context)
 {
```

```
    return MaterialApp(
      home: HomePage(),
    );
  }
}
class HomePage extends StatefulWidget
{
  @override
  _HomePageState createState() => _HomePageState();
}
class _HomePageState extends State<HomePage> {
  List<dynamic> _data = [];
  bool _isloading = false;
  @override
  void initState() {
    super.initState();
    _fetchDataFromApi();
  }

  Future<void> _fetchDataFromApi() async
  {
    setState(() {
      _isloading = true;
    });
    final response = await http.get(
        Uri.parse('https://jsonplaceholder.typicode.com/posts'));
    if (response.statusCode == 200) { // If the server returns a 200 OK response, parse the JSON.
      setState(() {
        _data = json.decode(response.body);
        _isloading = false;
      }
      );
    }
    else { // If the server did not return a 200 OK response,
      // throw an exception.
      throw Exception('Failed to load data');
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('API Data Example'),
      ),
      body: ListView.builder(
```

```
        itemCount: _data.length,
        itemBuilder: (context, index) {
          return ListTile(
            title: Text(_data[index]['title']),
            subtitle: Text(_data[index]['body']),
          );
        },
      ),
    );
  }
}
```

Output:



Experiment – 9(b)

AIM: Display the fetched data in a meaningful way in the UI.

DESCRIPTION:

To display the fetched data in a meaningful way in the UI, we can use a more structured layout rather than just displaying the data in a list. We'll create a custom widget to represent each post fetched from the API, and display them in a scrollable list.

We've added a loading indicator (CircularProgressIndicator) to indicate when data is being fetched. The fetched data is displayed as a list of PostCard widgets, each representing a post from the API. The PostCard widget displays the title and body of each post in a structured manner using a Card layout.

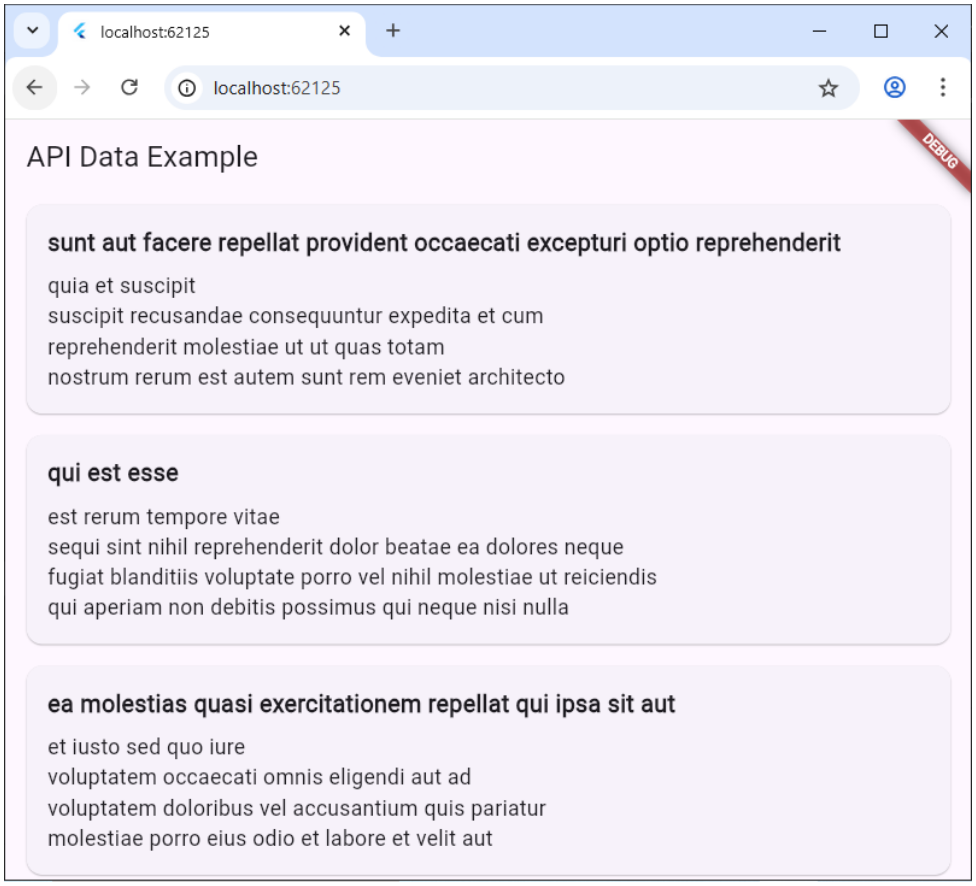
SOLUTION:

```
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
void main()
{
  runApp(MyApp());
}
class MyApp extends StatelessWidget
{
  @override
  Widget build(BuildContext context)
  {
    return MaterialApp(
      home: HomePage(),
    );
  }
}
class HomePage extends StatefulWidget
{
  @override
  _HomePageState createState() => _HomePageState();
}
class _HomePageState extends State<HomePage> {
  List<dynamic> _data = [];
  bool _isloading = false;
  @override
  void initState() {
    super.initState();
    _fetchDataFromApi();
  }

  Future<void> _fetchDataFromApi() async
  {
    setState(() {
      _isloading = true;
    });
    final response = await http.get(
        Uri.parse('https://jsonplaceholder.typicode.com/posts'));
    if (response.statusCode == 200)
    { // If the server returns a 200 OK response, parse the JSON.
```

```
      setState(() {
        _data = json.decode(response.body);
        _isloading = false;
      }
      );
    }
    else { // If the server did not return a 200 OK response,
      // throw an exception.
      throw Exception('Failed to load data');
    }
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('API Data Example'),
      ),
      body: _isloading ? Center(
        child: CircularProgressIndicator(),
      )
      : ListView.builder(
        itemCount: _data.length,
        itemBuilder: (context, index) {
          return PostCard(
            title: _data[index]['title'],
            body:  _data[index]['body'],
          );
        },
      ),
    );
  }
}
class PostCard extends StatelessWidget {
  final String title;
  final String body;
  const PostCard({
    Key? key,
    required this.title,
    required this.body,
  }) : super(key: key);
  @override
  Widget build(BuildContext context)
  {
    return Card(
      margin: EdgeInsets.symmetric(horizontal: 16, vertical: 8),
      child: Padding(
        padding: EdgeInsets.all(16),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: <Widget>[
            Text( title,
              style: TextStyle(
                fontSize: 18,
```

```
              fontWeight: FontWeight.bold
            ),
          ),
          SizedBox(height: 8),
          Text(body,
            style: TextStyle(fontSize: 16
            ),
          ),
        ],
      ),
    ),
  );
 }
}
```

Output:



Experiment – 10

AIM:
a) Write unit tests for UI components.
b) Use flutter's debugging tools to identify and fix issues.

Experiment – 10(a)

AIM: Write unit tests for UI components.

DESCRIPTION:
In Flutter, testing UI components is primarily achieved through Widget Tests, not traditional unit tests. Widget tests are specifically designed to verify the behavior and appearance of UI components (widgets).

Here's how to write widget tests for UI components in Flutter:

- Add the **flutter_test** dependency: This package is included by default in new Flutter projects.

```
dev_dependencies:
  flutter_test:
    sdk: flutter
```

- run "flutter pub get" to install the flutter_test package. (Not required if flutter_test dependency is already available)
- **Create a test file:** Create a new Dart file in your **test** folder, typically named [widget_name]_test.dart
- **Import necessary packages:**

```
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
// Import your widget file
import 'package:your_app_name/widgets/my_widget.dart';
```

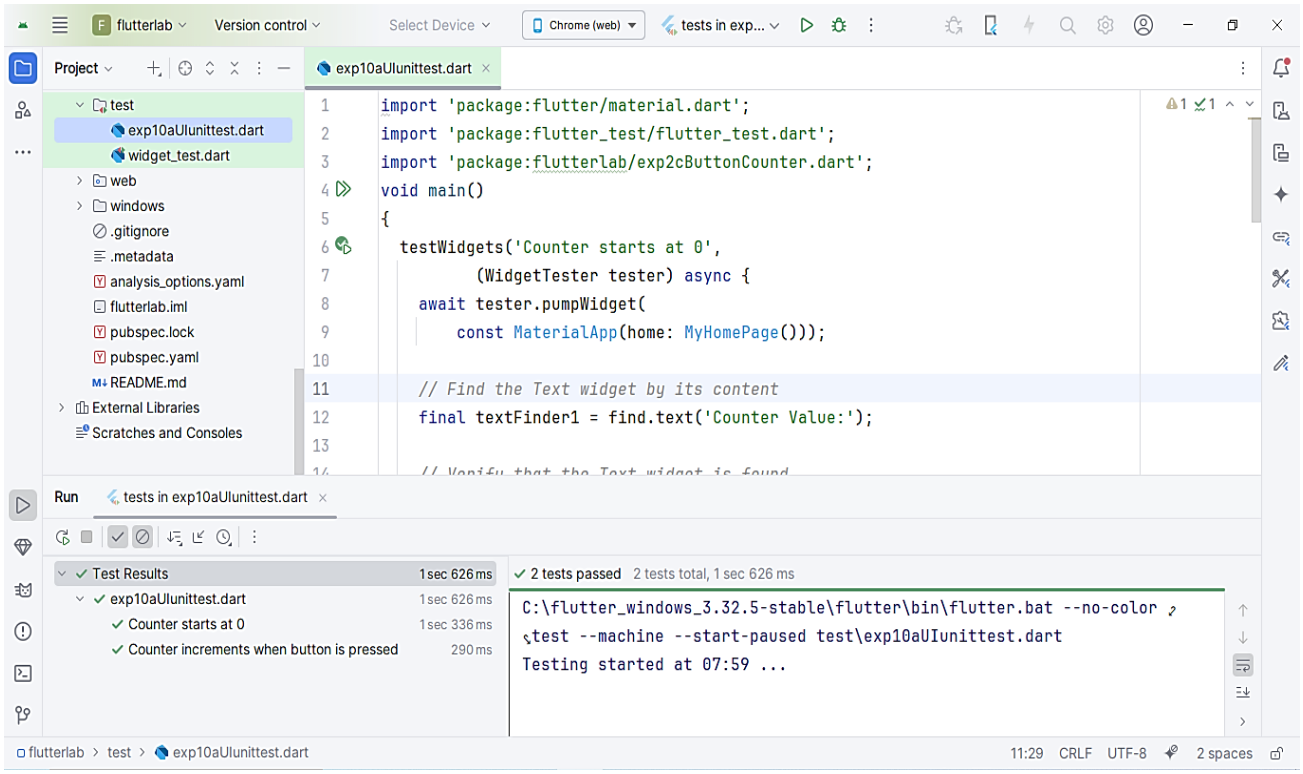
- **Write your widget test:** Use the **testWidgets** function to define your test cases. This functionprovides a **WidgetTester** object that allows you to interact with and inspect your widget tree.
  - ✓ **Key elements of widget testing:**
    - **tester.pumpWidget(widget):** Renders the provided widget in an isolated test environment.
    - **find methods**: Used to locate widgets in the widget tree (e.g., find.text(), find.byType(), find.byKey()).
    - **expect and matchers**: Used to assert conditions (e.g., findsOneWidget, findsNWidgets, findsNothing).
    - **tester.tap(finder)**: Simulates a tap on a widget.
    - **tester.pump()**: Rebuilds the widget tree, reflecting any state changes after user interactions or animations.
- **Run the tests:** You can run your tests from your IDE (e.g., VS Code, IntelliJ) or by executing flutter test in your terminal.

SOLUTION:

```
// a test file in the test directory
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
// Import your widget file
import 'package:flutterlab/exp2cButtonCounter.dart';
void main()
{
  testWidgets('Counter starts at 0',
        (WidgetTester tester) async {
    await tester.pumpWidget(
       const MaterialApp(home: MyHomePage()));

    // Find the Text widget by its content
    final textFinder1 = find.text('Counter Value:');

    // Verify that the Text widget is found
    expect(textFinder1, findsOneWidget);

    // Find the Text widget by its content
    final textFinder2 = find.text('0');

    // Verify that the Text widget is found
    expect(textFinder2, findsOneWidget);
  }
  );
  testWidgets('Counter increments when button is pressed',
        (WidgetTester tester) async {
      await tester.pumpWidget(
        const MaterialApp(home: MyHomePage()));

      // Find the button or tappable area
      final buttonFinder = find.text('Increment Counter');
      expect(buttonFinder, findsOneWidget);

      // Simulate a tap
```

```
    await tester.tap(buttonFinder);
    await tester.pump(); // Rebuild the widget tree after the tap

    // Verify the expected change (e.g., text changes, state updates)
    expect(find.text('0'), findsNothing);
    expect(find.text('1'), findsOneWidget);
  });
}
```

Output:



Experiment – 10(b)

AIM: Use flutter's debugging tools to identify and fix issues.

DESCRIPTION:
Flutter offers a comprehensive suite of debugging tools to help identify and fix issues in your applications. These tools are primarily integrated into your IDE (like VS Code or Android Studio) and accessible through Dart DevTools.

1. Dart DevTools:
Dart DevTools is a web-based suite of tools for debugging and profiling Flutter and Dart applications. It can be launched from your IDE during a debug session. Key features include:

- **Widget Inspector:** Visually inspect the widget tree, understand layout, and diagnose rendering issues. You can select widgets in the running app and see their properties and position in the tree, or vice-versa.
- **Performance View:** Analyse application performance, including frame rendering times, CPU usage, and network activity. Identify bottlenecks and optimize for smoother UI.
- **Memory Profiler:** Detect and analyse memory leaks, track memory allocation, and optimize memory usage to prevent performance degradation.
- **CPU Profiler:** Profile CPU usage to identify performance bottlenecks in your code and optimize execution efficiency.
- **Network Profiler:** Monitor network requests and responses, analyze network performance, and optimize API calls.
- **Logging View:** View and filter logs from your application, helping to understand the flow of execution and identify errors.

2. IDE Debugging (VS Code/Android Studio):
Your IDE provides powerful built-in debugging capabilities:

- **Breakpoints:** Set breakpoints in your Dart code to pause execution at specific lines, allowing you to inspect variable values, call stacks, and the overall state of your application at that point.
- **Step-through Debugging:** Step through your code line by line, entering or stepping over functions, to meticulously follow the execution flow and understand how your program behaves.
- **Variable Inspection:** Examine the values of variables in scope at breakpoints, helping to pinpoint incorrect data or unexpected changes.
- **Debug Console:** View application logs and print statements, providing valuable insights into the program's execution.

3. Other Useful Tools and Techniques:

- **flutter doctor:** This command line tool checks your Flutter development environment for common issues like missing dependencies or incorrect SDK versions. Running it first can often resolve setup-related problems.
- **Print Statements and Logging:** Simple print() statements or more structured logging using the dart:developer library can help you track variable values, execution paths, and debug specific sections of your code.
- **Assertions:** Use assert() statements to validate assumptions in your code during development. These assertions are only active in debug mode and can help catch logical errors early.
- **Hot Reload:** While not strictly a debugging tool, Hot Reload is invaluable for quickly iterating on UI changes and seeing the effects of your code modifications without restarting the entire application, speeding up the debugging process.

SOLUTION:

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

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

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

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0; // Initialize the counter value

  void _incrementCounter() {
    setState(() {
      // Increment the counter and update the UI
      _counter++;
    });
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Counter App'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'Counter Value:',
              style: TextStyle(fontSize: 20),
            ),
            Text(
              '$_counter', // Display the current counter value
```

```
            style: Theme.of(context).textTheme.headlineMedium,
          ),
          const SizedBox(height: 30), // Add some spacing
          ElevatedButton(
            onPressed: _incrementCounter, // Call the increment function on button press
            child: const Text('Increment Counter'),
          ),
        ],
      ),
    ),
  );
 }
}
```

Output: