**WEEK-1**

**AIM: A)** **Install Flutter and Dart SDK.**

1. To begin the installation of Flutter and Dart SDK on a Windows system, the first step is to ensure that the system meets the necessary requirements. This includes having Windows 10 or later (64-bit), a stable internet connection, and at least 2 GB of free disk space.
2. The process starts by installing Git, which is required for Flutter’s command-line tools. Git can be downloaded from its official website, and after installation, its successful setup can be verified using the git --version command in the Command Prompt.
3. Next, the Flutter SDK is downloaded from the official website (https://flutter.dev) by navigating to the "Get Started" section and selecting the Windows platform.
4. The downloaded ZIP file is then extracted to a suitable directory, such as C:\flutter.
5. Once extracted, the next step is to configure the system’s environment variables. This involves opening the System Properties, accessing Environment Variables, and adding the Flutter bin path (C:\flutter\bin) to the system PATH variable. This step ensures that Flutter commands can be run from any directory in the Command Prompt.
6. After setting the environment variable, the installation is verified by running the flutter doctor command in the Command Prompt. This command checks for missing dependencies and provides setup recommendations, such as installing Android Studio.
7. Android Studio is the recommended IDE for Flutter development as it includes the Android SDK, emulator support, and plugin integration. Once Android Studio is installed, the Flutter and Dart plugins can be added through the Plugins menu. Dart is bundled with Flutter, so it doesn’t need to be installed separately.
8. To run Flutter applications, either an Android Virtual Device (AVD) can be created using the AVD Manager in Android Studio or a real Android device can be connected with USB debugging enabled.
9. Once the setup is complete, a sample Flutter application can be created using the command flutter create demo\_app, and it can be launched using flutter run.
10. After successful execution, developers can start exploring the default app code and begin building custom Flutter applications using Dart language.

**AIM: B)** **Write a simple Dart program to understand the language fundamentals.**

**DESCRIPTION:** This Dart program demonstrates basic programming concepts such as variables, data types, conditionals, loops, functions, lists, and maps. It uses a sample user named John Doe and shows how to display information, perform checks using if-else, loop through numbers, define and call functions, work with a list of fruits, and store subject marks using a map. It serves as a beginner-friendly example to understand Dart language fundamentals.

**PROGRAM:**

// main.dart

void main() {

// 1. Variables and Data Types

String name = "John Doe";

int age = 20;

double height = 5.9;

bool isStudent = true;

print("Name: $name");

print("Age: $age");

print("Height: $height");

print("Is a Student? $isStudent");

// 2. Conditional Statements

if (age >= 18) {

print("You are an adult.");

} else {

print("You are a minor.");

}

// 3. Loops

print("Counting from 1 to 5:");

for (int i = 1; i <= 5; i++) {

print(i);

}

// 4. Functions

greetUser("John Doe");

int sum = addNumbers(10, 20);

print("Sum of 10 and 20 is $sum");

// 5. Lists

List<String> fruits = ["Apple", "Banana", "Mango"];

print("Fruits:");

for (String fruit in fruits) {

print("- $fruit");

}

// 6. Maps

Map<String, int> marks = {

"Math": 90,

"Science": 85,

"English": 88,

};

print("Subject Marks:");

marks.forEach((subject, mark) {

print("$subject: $mark");

});

}

// Function to greet user

void greetUser(String username) {

print("Hello, $username! Welcome to Dart.");

}

// Function to add two numbers

int addNumbers(int a, int b) {

return a + b;

}

**OUTPUT:**

Name: John Doe

Age: 20

Height: 5.9

Is a Student? true

You are an adult.

Counting from 1 to 5:

1

2

3

4

5

Hello, John Doe! Welcome to Dart.

Sum of 10 and 20 is 30

Fruits:

- Apple

- Banana

- Mango

Subject Marks:

Math: 90

Science: 85

English: 88

**WEEK-2**

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

**DESCRIPTION:**

1. Text Widget
2. The Text widget is used to show static or dynamic text on the screen. It supports customization like font size, color, style, etc.
3. Image Widget
4. The Image widget is used to show pictures. It supports various sources such as from the internet, local files, or asset folders.
5. Container Widget
6. The Container widget is one of the most commonly used widgets. It can be used to apply padding, margin, border, color, and even transformations. It often wraps other widgets to give them a styled appearance.

**PROGRAM:**

import 'package:flutter/material.dart';

void main() {

runApp(

MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text('Basic Flutter Widgets')),

body: Column(

children: [

// Text Widget

Text(

'Welcome to Flutter!',

style: TextStyle(fontSize: 24),

),

SizedBox(height: 20),

// Image Widget

Image.network(

'https://ganesh714.github.io/classmate/frontend/logos/logo-dark.png',

height: 100,

),

SizedBox(height: 20),

// Container Widget

Container(

height: 100,

width: 200,

color: Colors.blue,

alignment: Alignment.center,

child: Text(

'This is a Container',

style: TextStyle(color: Colors.white, fontSize: 16),

),

),

],

),

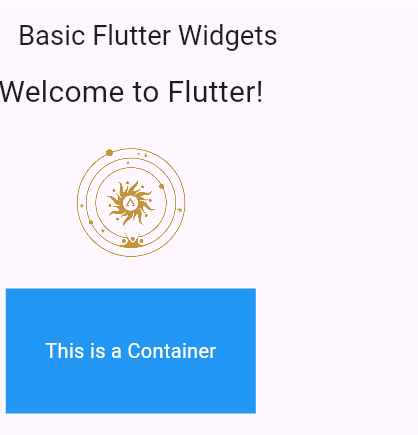
),

),

);

}

**OUTPUT:**

****

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

**DESCRIPTION:**

Row

1. Arranges widgets horizontally (side by side).
2. Useful when you want to place items in a single line from left to right.

Column

1. Arranges widgets vertically (top to bottom).
2. Used when you want to stack items one below another.

Stack

1. Overlaps widgets on top of each other.
2. Helpful for placing widgets in layers, like text over an image or a badge on a profile picture.

**PROGRAM 1:-**

import 'package:flutter/material.dart';

void main() {

runApp(

MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text("Column Example")),

body: Padding(

padding: EdgeInsets.all(16),

child: Column(

children: [

Text(

"Column Widget",

style: TextStyle(fontSize: 20),

),

SizedBox(height: 20),

Text("This is inside a Column"),

Text("Another Text Widget"),

],

),

),

),

),

);

}

**PROGRAM 2:-**

import 'package:flutter/material.dart';

void main() {

runApp(

MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text("Row Example")),

body: Center(

child: Row(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Icon(Icons.star, color: Colors.orange),

SizedBox(width: 10),

Text("This is a Row"),

],

),

),

),

),

);

}

**PROGRAM 3:-**

import 'package:flutter/material.dart';

void main() {

runApp(

MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text("Stack Example")),

body: Center(

child: Stack(

alignment: Alignment.center,

children: [

Container(

width: 120,

height: 120,

color: Colors.blue[200],

),

Text(

"Stack",

style: TextStyle(fontSize: 18, color: Colors.white),

)

],

),

),

),

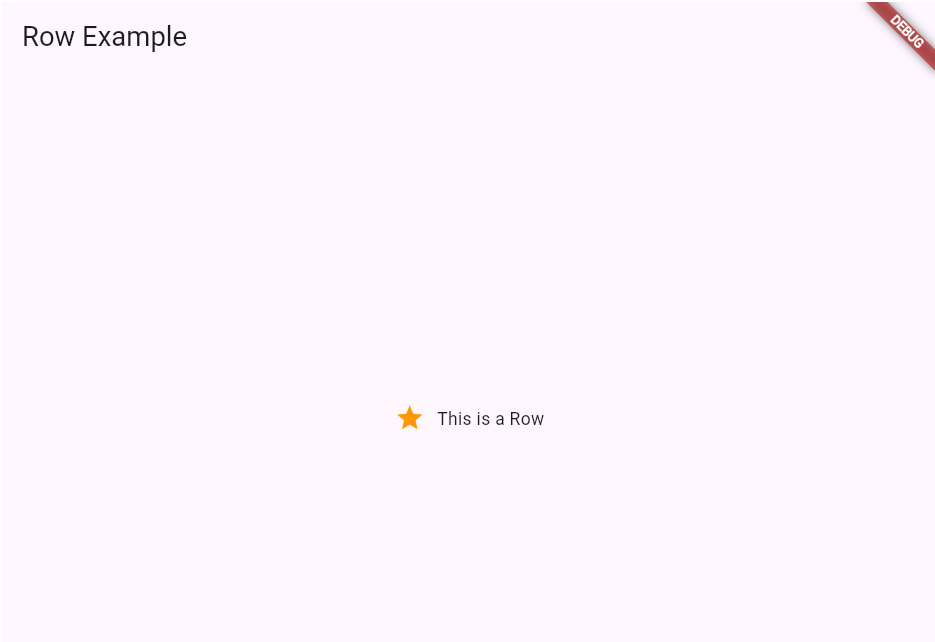
),

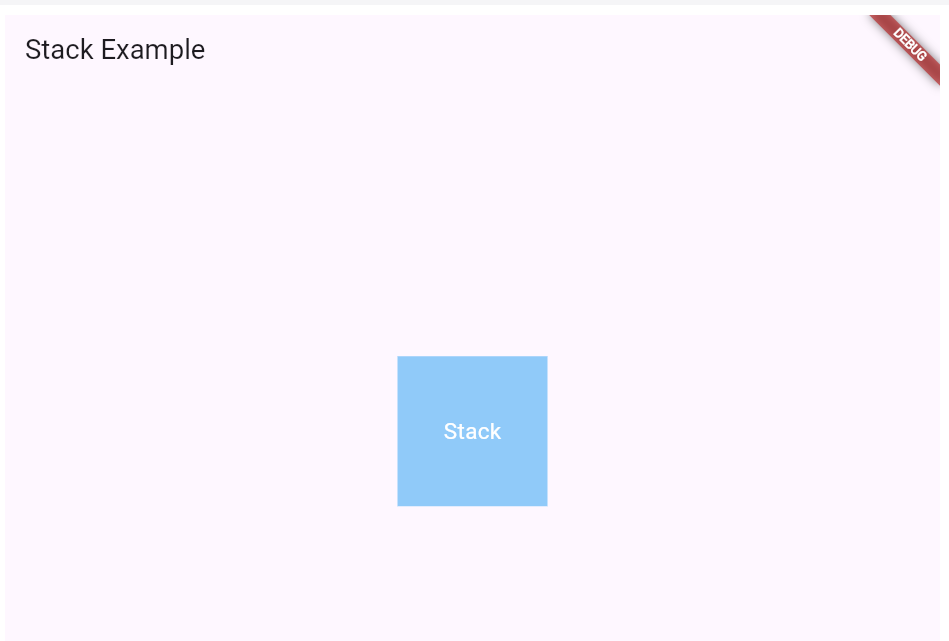
);

}

**OUTPUT:**

****

****

****

**WEEK-3**

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

**DESCRIPTION:**

Responsive UI in Flutter refers to creating user interfaces that automatically adjust their layout and appearance based on the screen size of the device. This ensures a smooth and consistent user experience across smartphones, tablets, and desktops. Flutter offers tools like LayoutBuilder, MediaQuery, and flexible widgets such as Row, Column, and Expanded to help developers design adaptive layouts. These tools detect screen dimensions and allow the UI elements to scale, rearrange, or resize accordingly.

**PROGRAM:**

import 'package:flutter/material.dart';

void main()

{

  runApp(

    MaterialApp(

      home : Scaffold(

        appBar : AppBar(title: const Text('Simple Responsive UI')),

        body : LayoutBuilder(

          builder: (contexdt, constraints){

            final width = constraints.maxWidth;

            String text = ' Mobile View';

            IconData icon = Icons.phone\_android;

            double fontSize = 24;

            if ( 600 <= width && width < 1200){

              text = 'Tablet View';

              icon = Icons.tablet;

              fontSize = 28;

            }

            else if(width >= 1200){

              text = 'Desktop View';

              icon = Icons.desktop\_windows;

              fontSize = 32;

            }

            return Center(

              child: Column(

                mainAxisSize: MainAxisSize.min,

                children:[

                  Icon(icon, size: 80),

                  const SizedBox(height: 20),

                  Text(text, style: TextStyle(fontSize : fontSize)),

                ]

              )

            );

          }

        )

      )

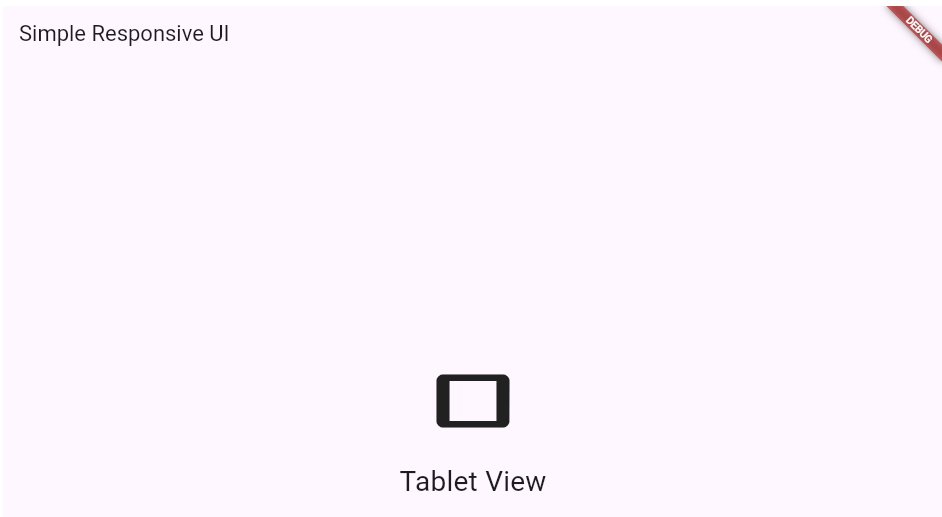
    )

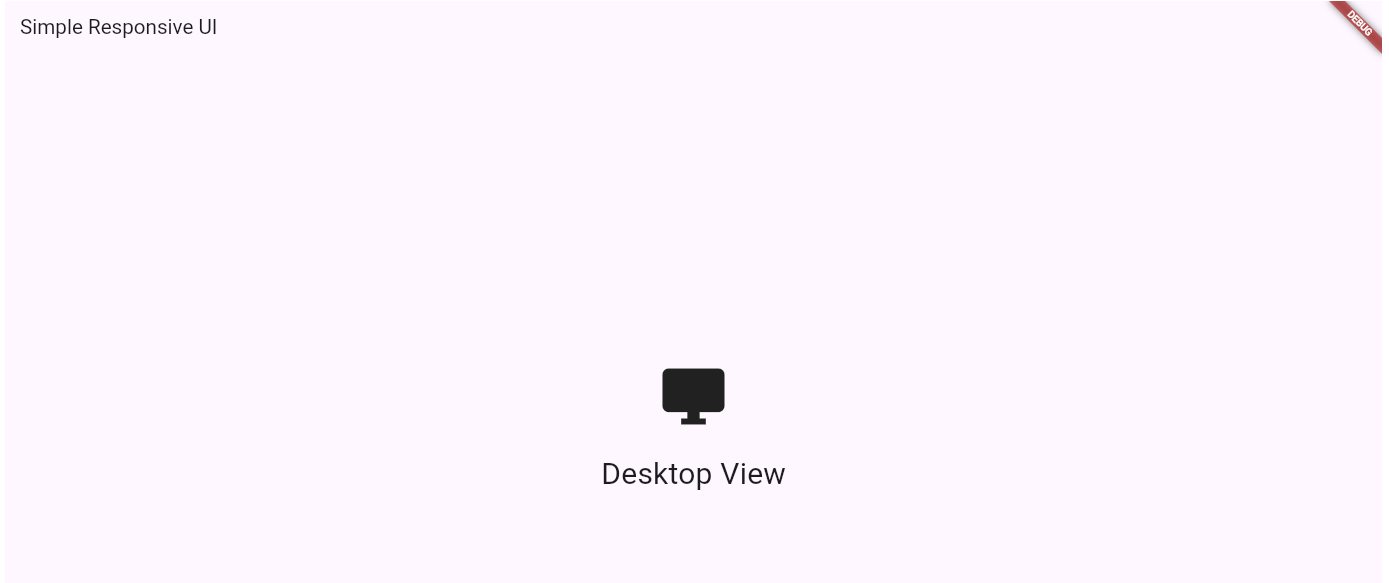
  );

}

**OUTPUT:**

****

****

****

**AIM: B) Implement media queries and breakpoints for responsiveness.**

**DESCRIPTION:** Media queries and breakpoints are essential tools in Flutter for building responsive user interfaces. Using MediaQuery, developers can access information about the device’s screen size, orientation, and pixel density. Breakpoints are specific width thresholds used to define different layouts for mobile, tablet, and desktop views. By checking these breakpoints in the code, Flutter apps can dynamically adjust UI elements—such as font size, widget visibility, or layout structure—to provide an optimal experience on various screen sizes.

**PROGRAM:**

import 'package:flutter/material.dart';

void main()

{

  runApp(

    MaterialApp(

      home : Scaffold(

        appBar : AppBar(title: const Text('Responsive with Media Query')),

        body : Builder(

          builder: (context){

            final width = MediaQuery.of(context).size.width;

            String text = 'Small Screen';

            IconData icon = Icons.phone\_android;

            double fontSize = 24;

            if ( 600 <= width && width < 1200){

              text = 'Medium Screen (Tablet)';

              icon = Icons.tablet;

              fontSize = 35;

            }

            else if(width >= 1200){

              text = 'Large Screnn (Desktop)';

              icon = Icons.desktop\_windows;

              fontSize = 45;

            }

            return Center(

              child: Column(

                mainAxisSize: MainAxisSize.min,

                children:[

                  Icon(icon, size: 80),

                  const SizedBox(height: 20),

                  Text(text, style: TextStyle(fontSize : fontSize)),

                  const SizedBox(height: 10),

                  Text('Screen width: ${width.toStringAsFixed(0)} px')

                ]

              )

            );

          }

        )

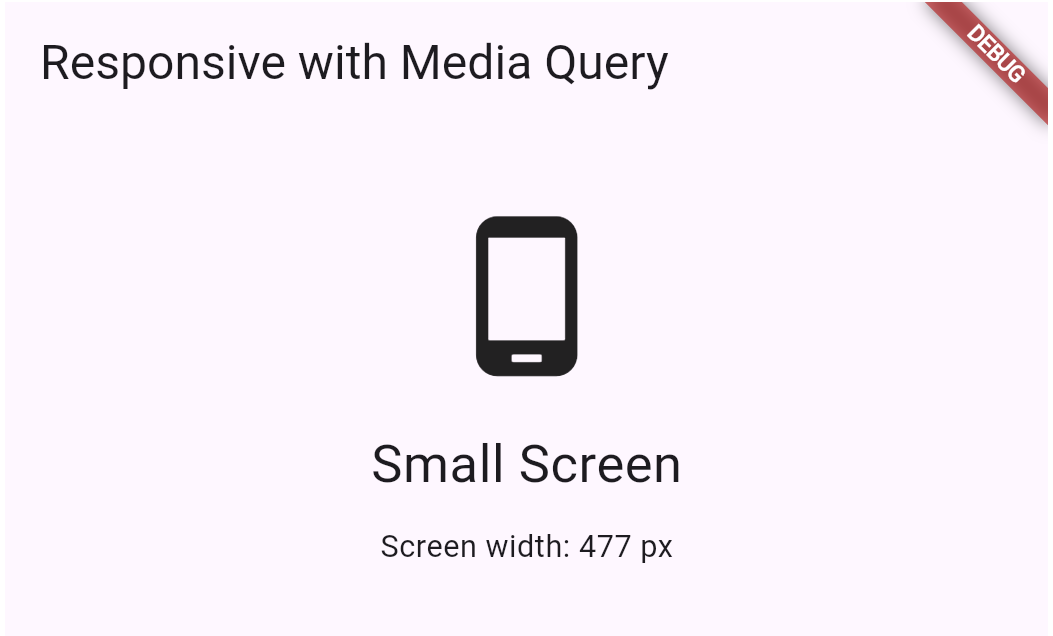
      )

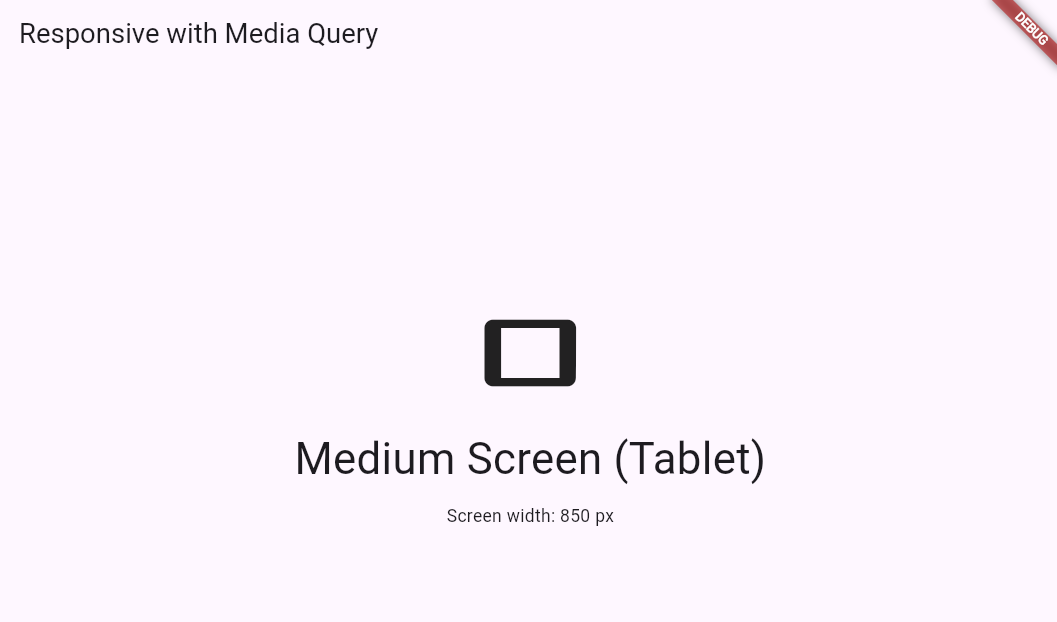
    )

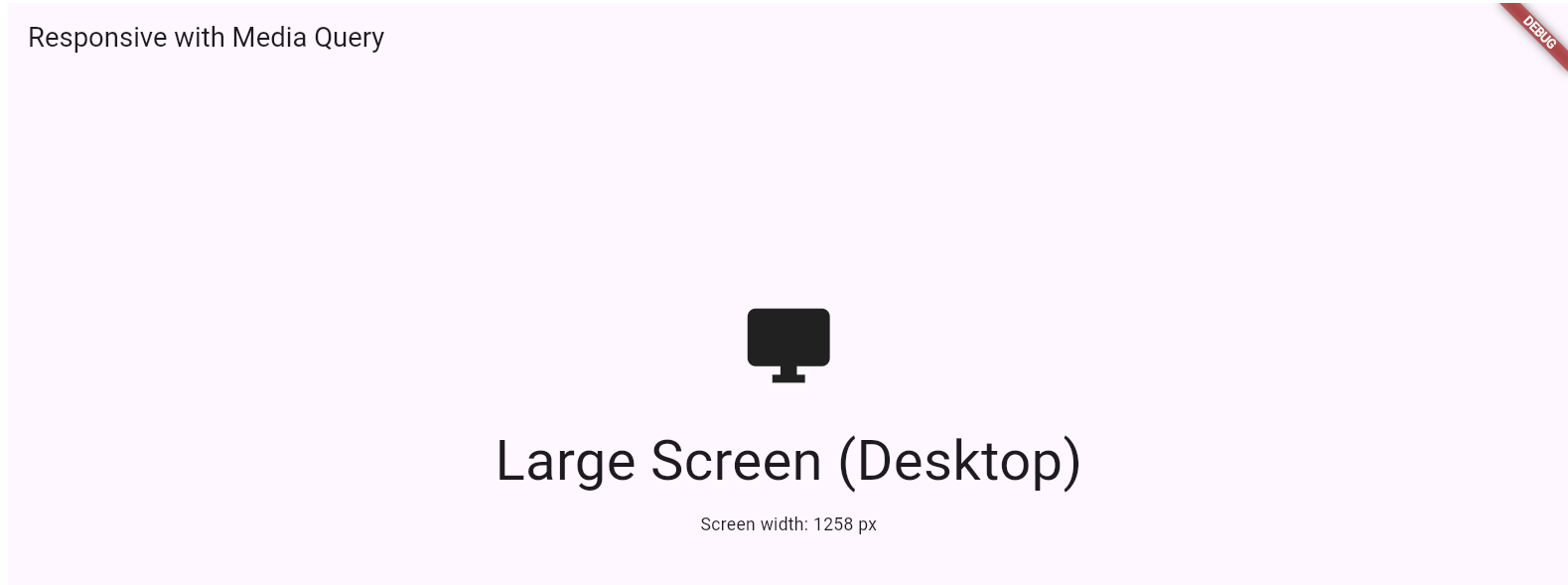
  );

}

**OUTPUT:**

****

****

****

**WEEK-4**

**AIM:** Set up navigation between different screens using Navigator.

**DESCRIPTION:** This Flutter program demonstrates how to navigate between two screens using the Navigator class. It starts with a MaterialApp whose home screen is set to FirstScreen. On this screen, there is a button labeled "Go to Second Screen". When the user taps this button, the Navigator.push() method is called with a MaterialPageRoute that loads the SecondScreen widget. This action places the second screen on top of the navigation stack and displays it. The second screen also contains a button labeled "Back to First Screen". When this button is pressed, the Navigator.pop() method is used to remove the second screen from the stack, taking the user back to the first screen. This simple program illustrates the use of Flutter's Navigator for pushing and popping routes to handle screen transitions in an app.

**PROGRAM:**

import 'package:flutter/material.dart';

void main() => runApp(

MaterialApp(

home: Scaffold(

appBar: AppBar(title: const Text('First Screen')),

body: Builder(

builder: (context) {

return Center(

child: ElevatedButton(

child: const Text('Go to second screen'),

onPressed: () {

Navigator.push(

context,

MaterialPageRoute(

builder: (context) => Scaffold(

appBar: AppBar(title: const Text('Second Screen')),

body: Center(

child: ElevatedButton(

child: const Text('Go back'),

onPressed: () {

Navigator.pop(context);

},

),

),

),

),

);

},

),

);

},

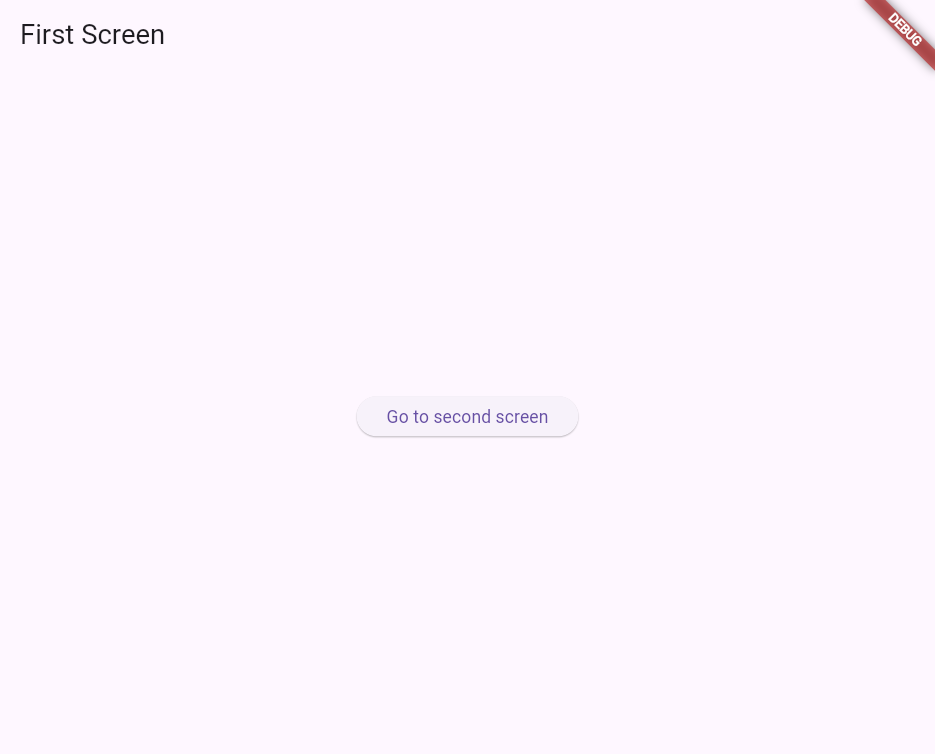
),

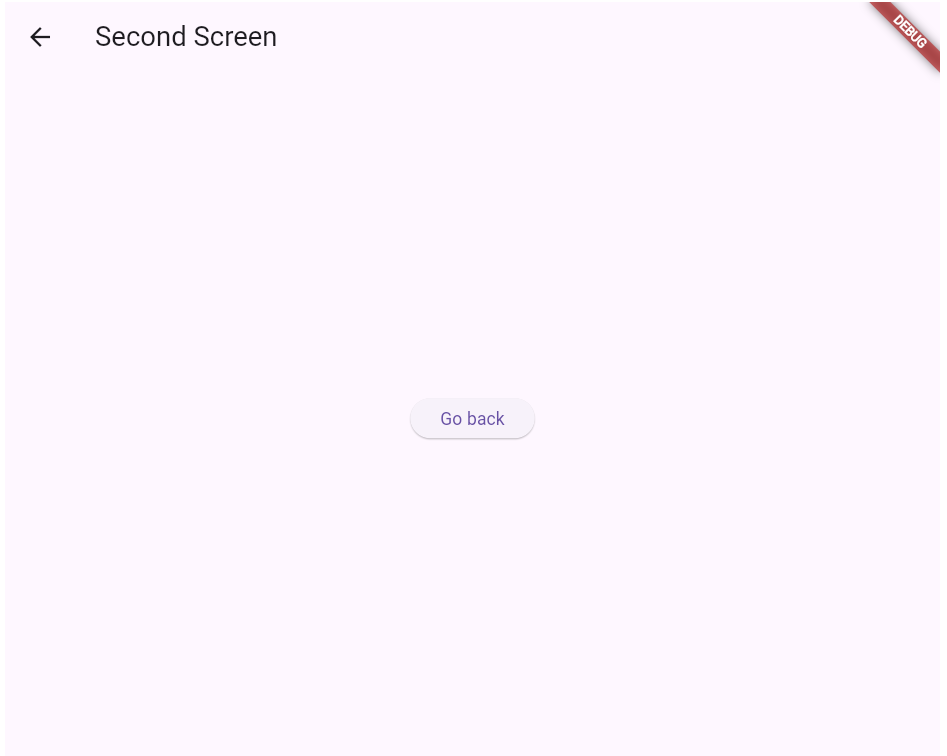
),

),

);

**OUTPUT:**

****

****

**B) AIM:** Implement navigation with named routes.

**DESCRIPTION:**

In Flutter, named routes allow you to define and manage navigation in a centralized way by assigning a string name to each screen (route). Instead of directly creating a MaterialPageRoute inside the Navigator.push() call, you define routes in the MaterialApp widget using the routes property and then navigate using Navigator.pushNamed() with the route's name. This makes the code cleaner, especially in large apps with multiple screens, because all route definitions are maintained in one place. To navigate back, Navigator.pop() is used just like with normal routes. Named routes are useful for structured navigation, easy route management, and when passing arguments between multiple screens.

**PROGRAM:**

import 'package:flutter/material.dart';

void main() => runApp(

MaterialApp(

initialRoute: '/',

routes: {

'/': (context) => Scaffold(

appBar: AppBar(title: const Text('Home Screen')),

body: Center(

child: Column(

mainAxisSize: MainAxisSize.min,

children: [

ElevatedButton(

child: const Text("Go to Screen 1"),

onPressed: () {

Navigator.pushNamed(context, '/screen1');

},

),

const SizedBox(height: 20),

ElevatedButton(

child: const Text("Go to Screen 2"),

onPressed: () {

Navigator.pushNamed(context, '/screen2');

},

),

],

),

),

),

'/screen1': (context) => Scaffold(

appBar: AppBar(title: const Text('Screen 1')),

body: Center(

child: ElevatedButton(

child: const Text('Go Back'),

onPressed: () {

Navigator.pop(context);

},

),

),

),

'/screen2': (context) => Scaffold(

appBar: AppBar(title: const Text('Screen 2')),

body: Center(

child: ElevatedButton(

child: const Text('Go Back'),

onPressed: () {

Navigator.pop(context);

},

),

),

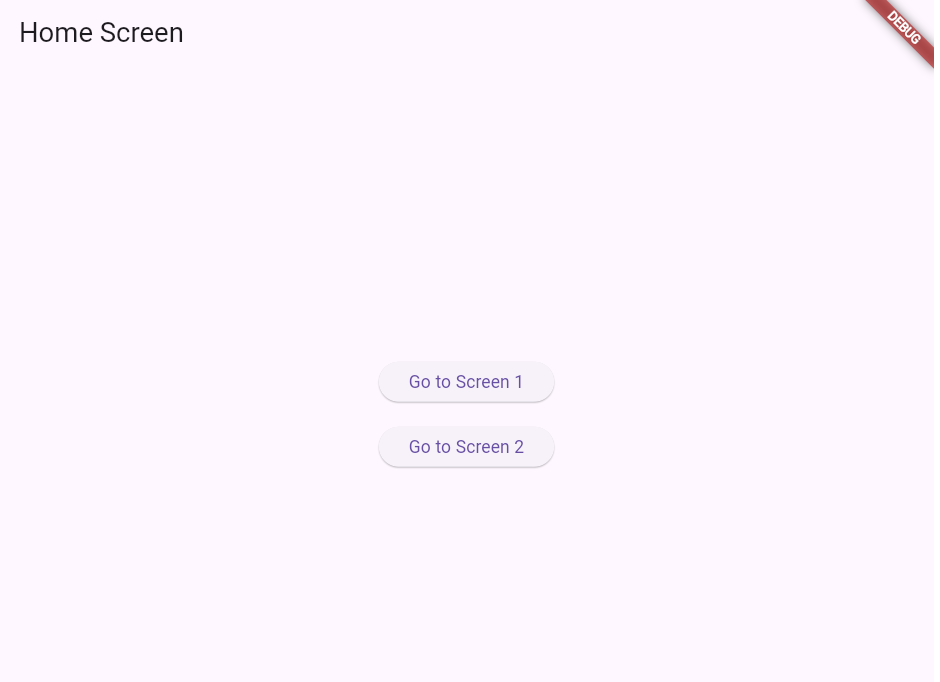
),

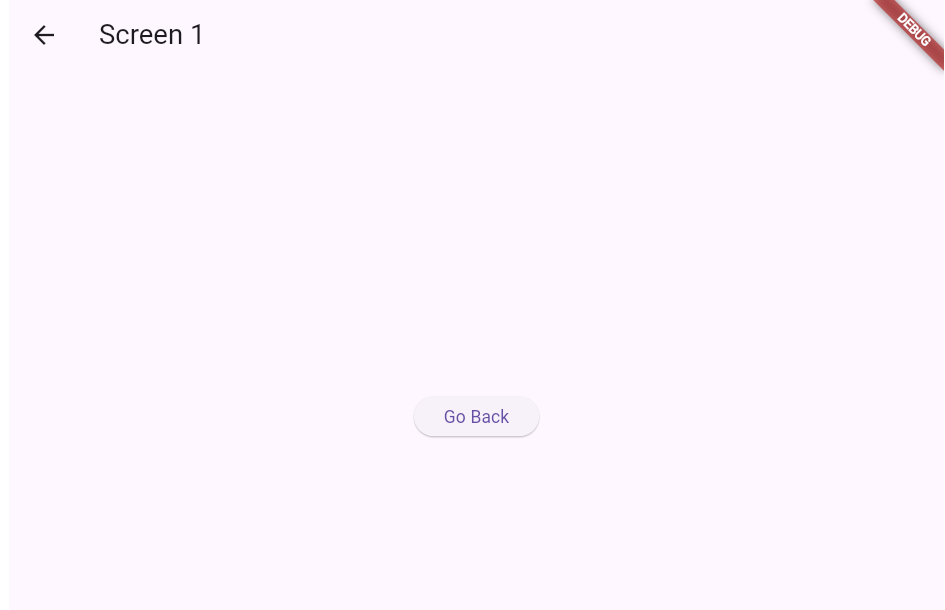
},

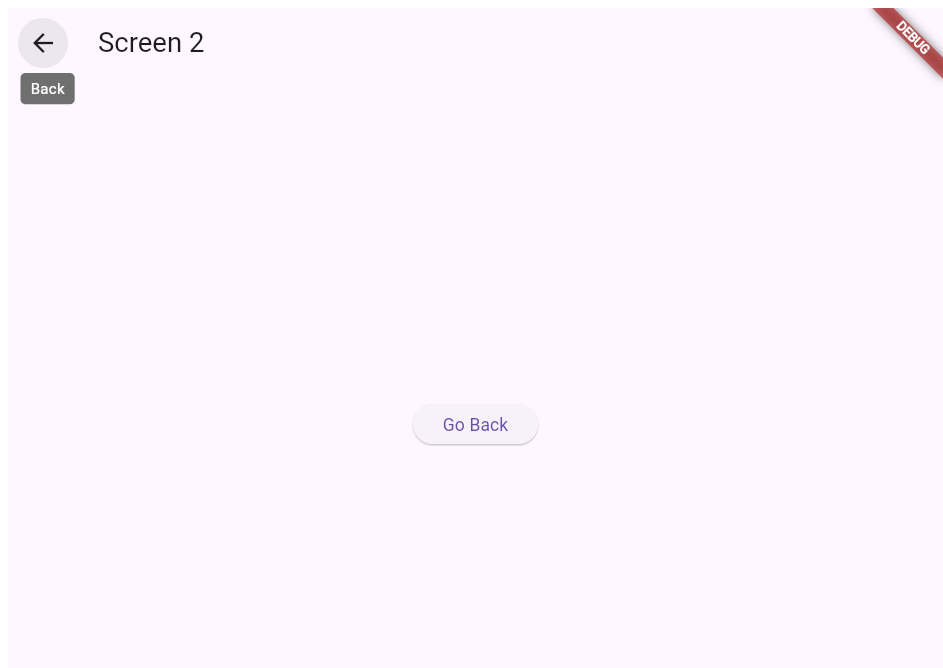
),

);

**OUTPUT:**

****

****

****

**WEEK-5**

**AIM:A)** Learn about stateful and stateless widgets.

**DESCRIPTION:** In Flutter, a Stateless Widget is immutable and displays fixed content that doesn’t change during its lifetime, making it ideal for static UI. A Stateful Widget can change dynamically based on user interaction or data updates, using a State class and setState() to rebuild the UI when needed.

**PROGRAM:**

import 'package:flutter/material.dart';

void main() => runApp(MaterialApp(home: Counter()));

class Counter extends StatefulWidget {

@override

\_CounterState createState() => \_CounterState();

}

class \_CounterState extends State<Counter> {

int count = 0;

@override

Widget build(BuildContext context) {

return Scaffold(

body: Center(

child: TextButton(

onPressed: () => setState(() => count++),

child: Text('Count: $count', style: TextStyle(fontSize: 28)),

),

),

);

}

}

**OUTPUT:**

****

**AIM:B)** Implement state management using set State and Provider.

**DESCRIPTION:** In Flutter, state management controls how data changes and updates the UI. Using setState(), state changes are handled within a single widget by calling setState() to rebuild the UI with updated values. The Provider package offers a more scalable approach, allowing state to be stored and shared across multiple widgets in the widget tree, enabling cleaner code and easier data management for larger applications.

**PROGRAM:**

import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: SetStateCounter(),

);

}

}

class SetStateCounter extends StatefulWidget {

@override

\_SetStateCounterState createState() => \_SetStateCounterState();

}

class \_SetStateCounterState extends State<SetStateCounter> {

int \_count = 0;

void \_increment() {

setState(() {

\_count++;

});

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text('State Management: SetState')),

body: Center(

child: Text('Count: $\_count', style: TextStyle(fontSize: 32)),

),

floatingActionButton: FloatingActionButton(

onPressed: \_increment,

child: Icon(Icons.add),

),

);

}

}

**OUTPUT:**

****

import 'package:flutter/material.dart';

void main() => runApp(

MaterialApp(

theme: ThemeData(

primaryColor: Colors.deepPurple,

iconTheme: IconThemeData(

color: Colors.deepPurple,

size: 36,

),

textTheme: TextTheme(

bodyMedium: TextStyle(

fontSize: 20,

fontWeight: FontWeight.w600,

color: Colors.deepPurple,

),

),

),

home: Center(

child: Row(

mainAxisSize: MainAxisSize.min,

children: [

CustomIcon(),

SizedBox(width: 8),

CustomText(),

],

),

),

),

);

class CustomIcon extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Icon(

Icons.star,

color: Theme.of(context).iconTheme.color,

size: Theme.of(context).iconTheme.size,

);

}

}

class CustomText extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Text(

'Styled By Theme!',

style: Theme.of(context).textTheme.bodyMedium,

);

}

}

-----------------------1------------

import 'package:flutter/material.dart';

void main() => runApp(

MaterialApp(

home: Scaffold(

body: Center(

child: Row(

mainAxisSize: MainAxisSize.min,

children: [

CustomIcon(),

SizedBox(width: 20),

CustomText(),

],

),

),

),

),

);

class CustomIcon extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Icon(

Icons.favorite,

color: Colors.amber,

size: 40,

);

}

}

class CustomText extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Text(

'Hello, Flutter!',

style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),

);

}

}

------------2---------------

import 'package:flutter/material.dart';

void main() => runApp(

MaterialApp(

home: Scaffold(

appBar: AppBar(

title: Text("Icon & Text with Button"),

backgroundColor: Colors.blue,

),

body: Center(

child: Row(

mainAxisSize: MainAxisSize.min,

children: [

CustomIcon(),

SizedBox(width: 20),

CustomText(),

SizedBox(width: 20),

ElevatedButton(

onPressed: () {

print("Button Pressed!");

},

child: Text("Click Me"),

),

],

),

),

),

),

);

class CustomIcon extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Icon(

Icons.favorite,

color: Colors.amber,

size: 40,

);

}

}

class CustomText extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Text(

'Hello, Flutter!',

style: TextStyle(fontSize: 24, color: Colors.blue),

);

}

}

------------------3-----------------

import 'package:flutter/material.dart';

void main() => runApp(

MaterialApp(

home: Scaffold(

body: Center(

child: Row(

mainAxisSize: MainAxisSize.min,

children: [

CustomIcon(),

SizedBox(width: 20),

CustomText(),

],

),

),

),

),

);

class CustomIcon extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Icon(

Icons.favorite,

color: Colors.amber,

);

}

}

class CustomText extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Text(

'Custom Widget!',

style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold),

);

}

}

--------------4----------------

import 'package:flutter/material.dart';

void main() =>runApp(

MaterialApp(

home:Center(

child:Row(

mainAxisSize:MainAxisSize.min,

children:[

CustomIcon(),

SizedBox(width:8),

CustomText(),

SizedBox(width:8),

CustomButton(),

],),

),),

);

//Custom Icon Widget

class CustomIcon extends StatelessWidget{

@override

Widget build(BuildContext context)

{

return Icon(Icons.favorite,color:Colors.red);

}

}

//custom text widget

class CustomText extends StatelessWidget{

@override

Widget build(BuildContext context)

{

return Text(

'Custom Widget!',style:TextStyle(fontSize:18,fontWeight:FontWeight.bold),

);

}

}

//custom button

class CustomButton extends StatelessWidget{

@override

Widget build(BuildContext context)

{

return ElevatedButton(

onPressed: () {

print("Button Pressed!");

},

child: Text("Click Me"),

);

}

}

-----------------5---------------

**WEEK-7**

**AIM:** Design a form with various input fields.

**DESCRIPTION:** In this tinkering activity, a form is designed with various input fields such as text boxes, radio buttons, checkboxes, dropdowns, and submit buttons. The goal is to explore how user input can be collected and processed through an interactive interface. This experiment helps understand basic UI design, data collection, and form validation concepts, encouraging creativity and problem-solving in building user-friendly digital forms.

**PROGRAM:**

import 'package:flutter/material.dart';

void main() => runApp(MaterialApp(home: SimpleForm()));

class SimpleForm extends StatefulWidget {

@override

\_SimpleFormState createState() => \_SimpleFormState();

}

class \_SimpleFormState extends State<SimpleForm> {

final \_formKey = GlobalKey<FormState>();

String name = '';

String email = '';

String password = '';

String gender = 'Male';

bool accepted = false;

final genders = ['Male', 'Female', 'Other'];

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text('Simple Form')),

body: Padding(

padding: EdgeInsets.all(16),

child: Form(

key: \_formKey,

child: ListView(

children: [

TextFormField(

decoration: InputDecoration(labelText: 'Name'),

onSaved: (val) => name = val ?? '',

validator: (val) => val!.isEmpty ? 'Enter name' : null,

),

TextFormField(

decoration: InputDecoration(labelText: 'Email'),

keyboardType: TextInputType.emailAddress,

onSaved: (val) => email = val ?? '',

validator: (val) => val!.contains('@') ? null : 'Enter valid email',

),

TextFormField(

decoration: InputDecoration(labelText: 'Password'),

obscureText: true,

onSaved: (val) => password = val ?? '',

validator: (val) => val!.length < 6 ? 'Min 6 characters' : null,

),

DropdownButtonFormField<String>(

value: gender,

items: genders

.map<DropdownMenuItem<String>>(

(g) => DropdownMenuItem<String>(

value: g,

child: Text(g),

))

.toList(),

onChanged: (String? val) => setState(() => gender = val ?? 'Male'),

decoration: InputDecoration(labelText: 'Gender'),

),

CheckboxListTile(

title: Text('Accept Terms'),

value: accepted,

onChanged: (bool? val) => setState(() => accepted = val ?? false),

),

ElevatedButton(

child: Text('Submit'),

onPressed: () {

if (\_formKey.currentState!.validate() && accepted) {

\_formKey.currentState!.save();

showDialog(

context: context,

builder: (\_) => AlertDialog(

title: Text('Form Submitted'),

content: Text(

'Name: $name\nEmail: $email\nGender: $gender'),

),

);

} else if (!accepted) {

ScaffoldMessenger.of(context).showSnackBar(

SnackBar(content: Text('Please accept terms')),

);

}

},

),

],

),

),

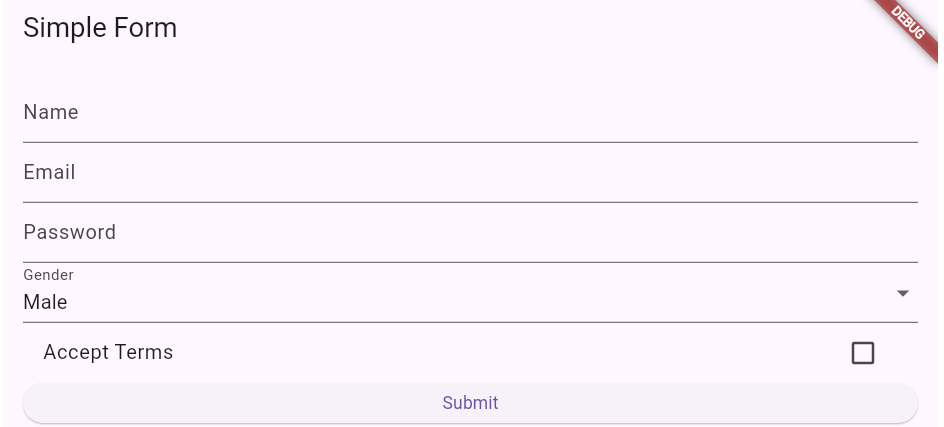
),

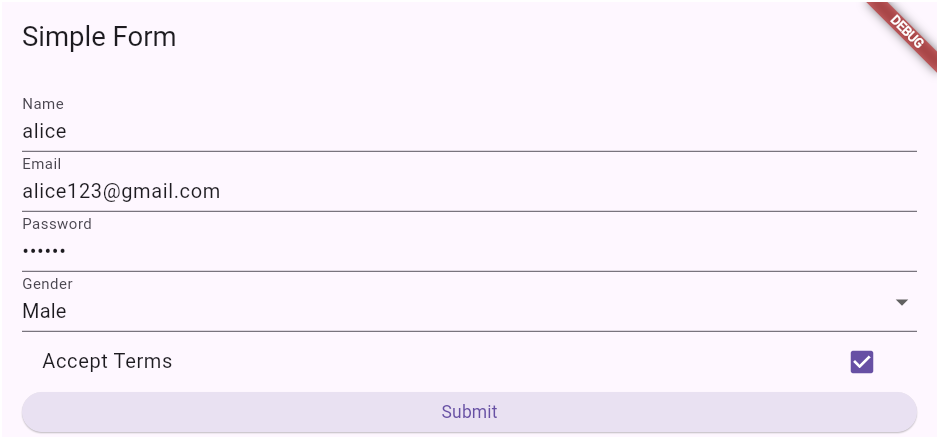
);

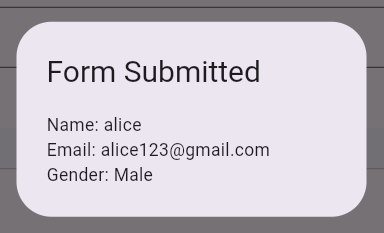
}

}

**OUTPUT:**

****

****

****

**B) AIM:** Implement input validation and error messages using Form and validator

**DESCRIPTION:** In this tinkering activity, a Flutter Form is created with multiple input fields that include validation and error messages using the Form widget and validator functions. Each field checks for specific input rules, such as non-empty names, valid email formats, and minimum password length. If inputs are invalid, real-time error messages guide users to correct them. This helps learners understand form validation, user feedback, and data reliability in app development.

**PROGRAM:**

import 'package:flutter/material.dart';

void main() => runApp(MaterialApp(home: ValidationForm()));

class ValidationForm extends StatefulWidget {

@override

\_ValidationFormState createState() => \_ValidationFormState();

}

class \_ValidationFormState extends State<ValidationForm> {

final \_formKey = GlobalKey<FormState>();

String email = '';

String password = '';

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text('Validation Form')),

body: Padding(

padding: EdgeInsets.all(16),

child: Form(

key: \_formKey,

child: Column(

children: [

// Email input with validator

TextFormField(

decoration: InputDecoration(labelText: 'Email'),

keyboardType: TextInputType.emailAddress,

validator: (val) {

if (val == null || val.isEmpty) return 'Email is required.';

if (!val.contains('@')) return 'Enter a valid email.';

return null;

},

onSaved: (val) => email = val ?? '',

),

SizedBox(height: 16),

// Password input with validator

TextFormField(

decoration: InputDecoration(labelText: 'Password'),

obscureText: true,

validator: (val) {

if (val == null || val.isEmpty) return 'Password is required.';

if (val.length < 6) return 'Password must be at least 6 characters.';

return null;

},

onSaved: (val) => password = val ?? '',

),

SizedBox(height: 24),

ElevatedButton(

child: Text('Submit'),

onPressed: () {

if (\_formKey.currentState!.validate()) {

// If valid, save the form fields

\_formKey.currentState!.save();

showDialog(

context: context,

builder: (\_) => AlertDialog(

title: Text('Success'),

content: Text('Email: $email\nPassword: $password'),

),

);

}

},

),

],

),

),

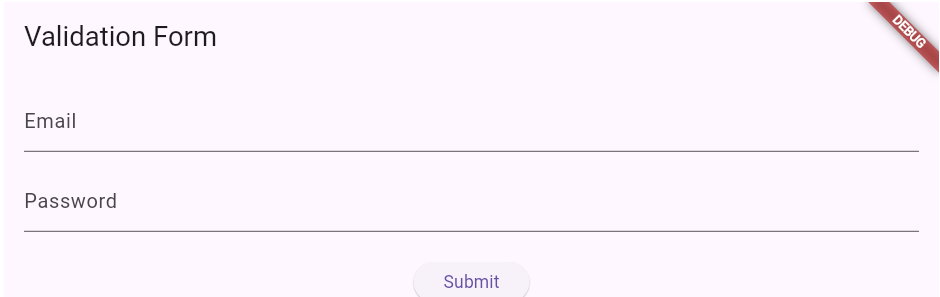
),

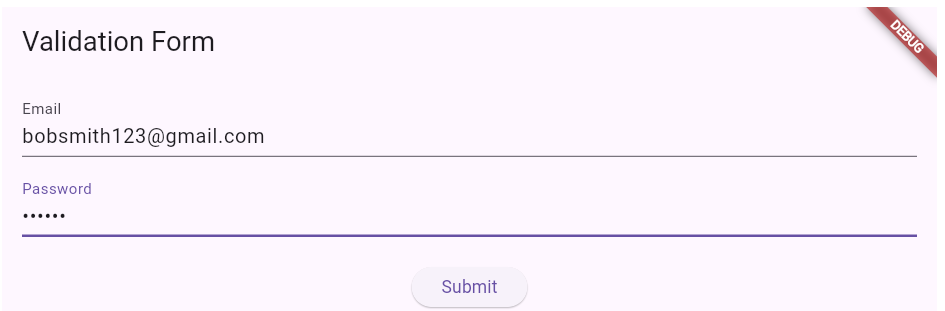
);

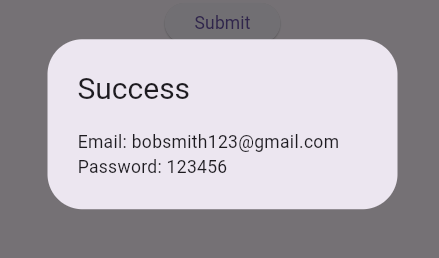
}

}

**OUTPUT:**

****

****

****

**WEEK-8**

**AIM:** Add animations to UI elements using Flutter's animation framework

**DESCRIPTION:** In this tinkering activity, animations are added to UI elements using Flutter’s built-in animation framework to make the interface more dynamic and engaging. Widgets like AnimatedContainer, AnimatedOpacity, or AnimationController are used to create smooth transitions, motion effects, and interactive visuals. This helps learners understand the basics of implicit and explicit animations, improving the user experience and visual appeal of Flutter applications.

**PROGRAM:**

import 'package:flutter/material.dart';

void main() => runApp(MaterialApp(home: SimpleAnimation()));

class SimpleAnimation extends StatefulWidget {

const SimpleAnimation({super.key});

@override

State<SimpleAnimation> createState() => \_SimpleAnimationState();

}

class \_SimpleAnimationState extends State<SimpleAnimation> {

bool big = false;

@override

Widget build(BuildContext context) {

return Scaffold(

body: Center(

child: AnimatedContainer(

width: big ? 200:100,

height: big ? 200:100,

color: big ? Colors.orange:Colors.blue,

duration: const Duration(seconds: 1),

),

),

floatingActionButton: FloatingActionButton(onPressed: () => setState(() => big = !big),

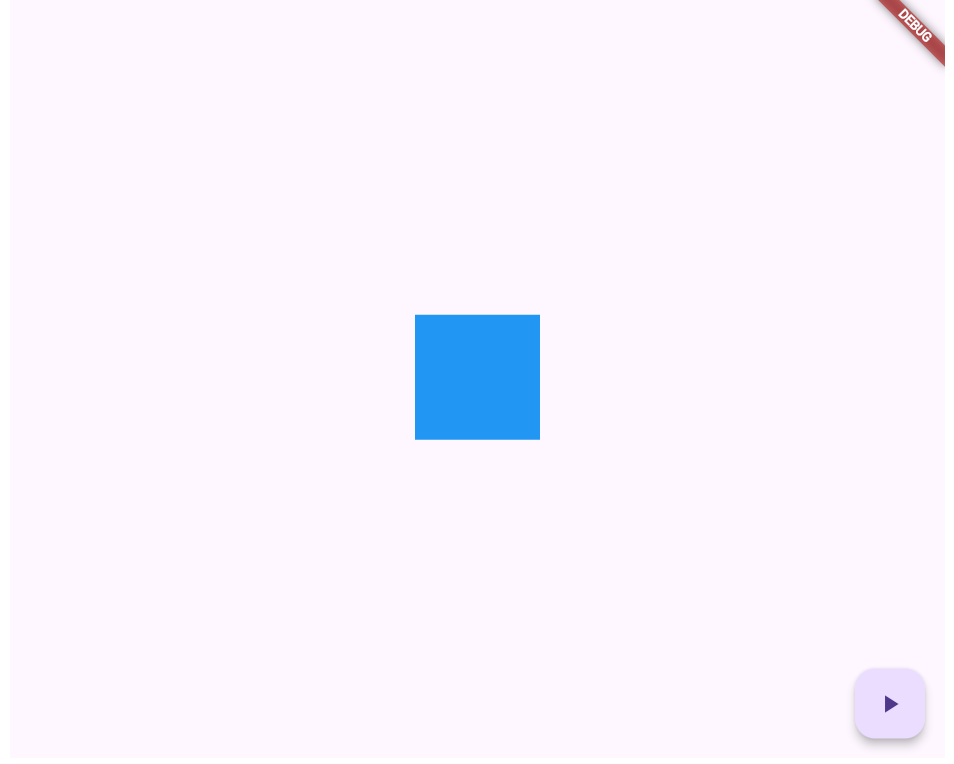
child: const Icon(Icons.play\_arrow),),

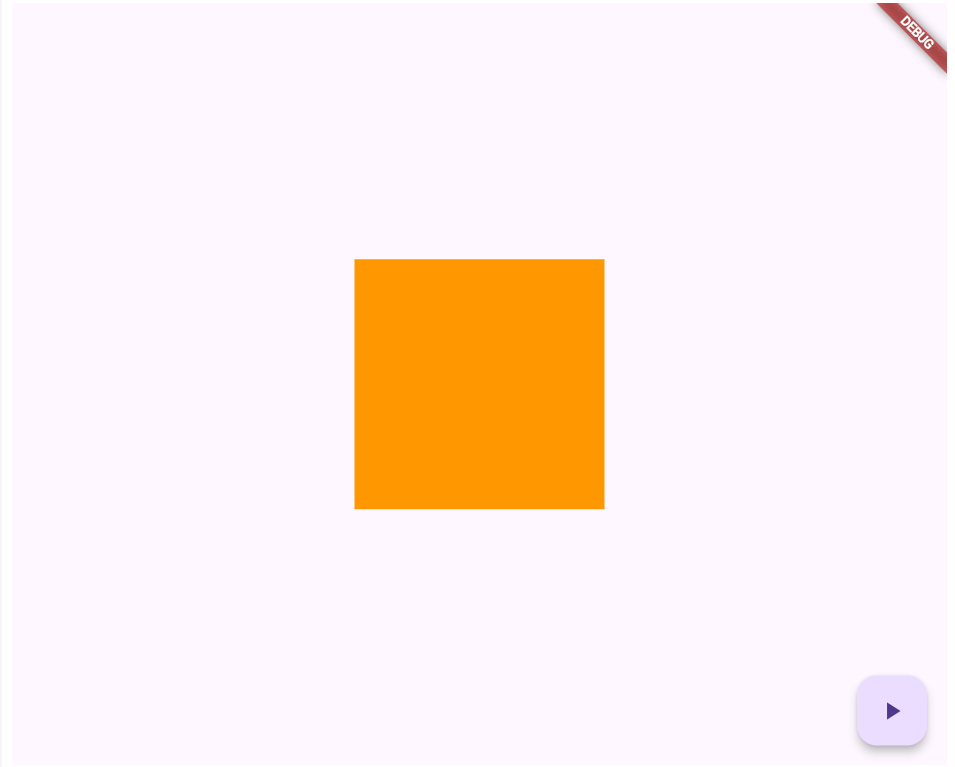
);

}

}

**OUTPUT:**

****

****

**B) AIM:** Experiment with different types of animations (fade, slide, etc.).

**DESCRIPTION:** In this tinkering activity, different types of animations such as fade, slide, scale, and rotation are implemented using Flutter’s animation widgets. The experiment demonstrates how motion can enhance user interaction and make UI transitions smoother. By adjusting animation duration, curves, and effects, learners explore how various animations influence the overall look, feel, and responsiveness of an app’s interface.

**PROGRAM:**

import 'package:flutter/material.dart';

void main() => runApp(const MaterialApp(home: SimpleAnimation()));

class SimpleAnimation extends StatefulWidget {

const SimpleAnimation({super.key});

@override

State<SimpleAnimation> createState() => \_SimpleAnimationState();

}

class \_SimpleAnimationState extends State<SimpleAnimation> {

bool big = false; // using this instead of 'on'

// Helper widget for reusable boxes

Widget \_box(Color color) => Container(width: 80, height: 80, color: color);

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text("Simple Animation Example")),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

// Fade animation

AnimatedOpacity(

opacity: big ? 1 : 0,

duration: const Duration(seconds: 1),

child: \_box(Colors.blue),

),

const SizedBox(height: 20),

// Slide animation

AnimatedSlide(

offset: big ? Offset.zero : const Offset(-1, 0),

duration: const Duration(seconds: 1),

child: \_box(Colors.green),

),

const SizedBox(height: 20),

// Scale animation

AnimatedScale(

scale: big ? 1 : 0.5,

duration: const Duration(seconds: 1),

child: \_box(Colors.orange),

),

],

),

),

floatingActionButton: FloatingActionButton(

onPressed: () => setState(() => big = !big),

child: const Icon(Icons.play\_arrow),

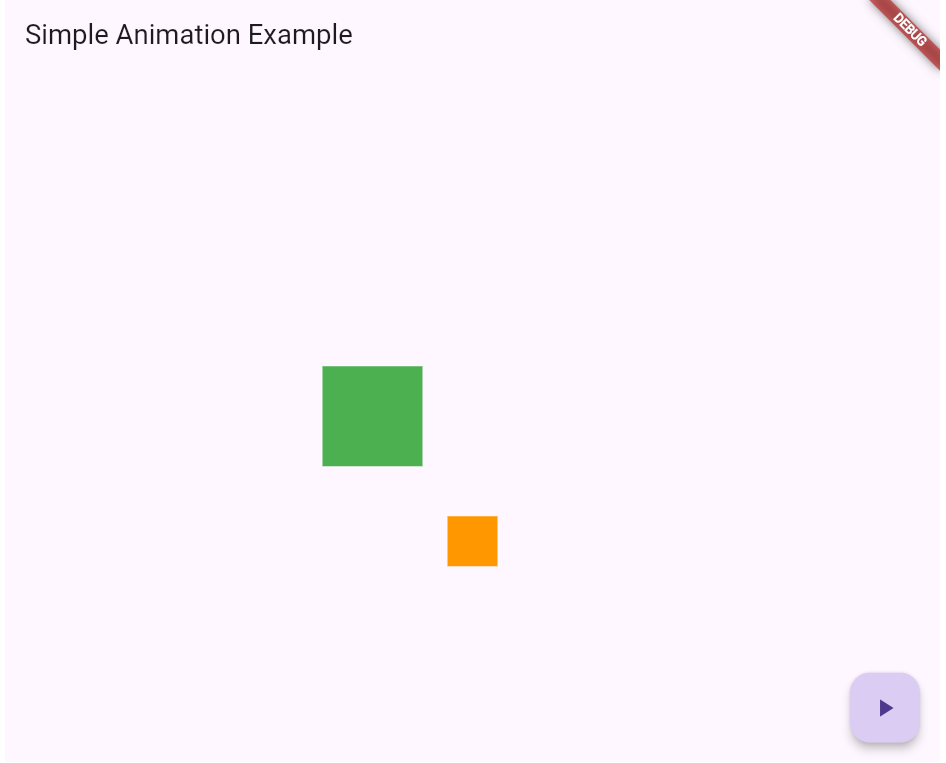
),

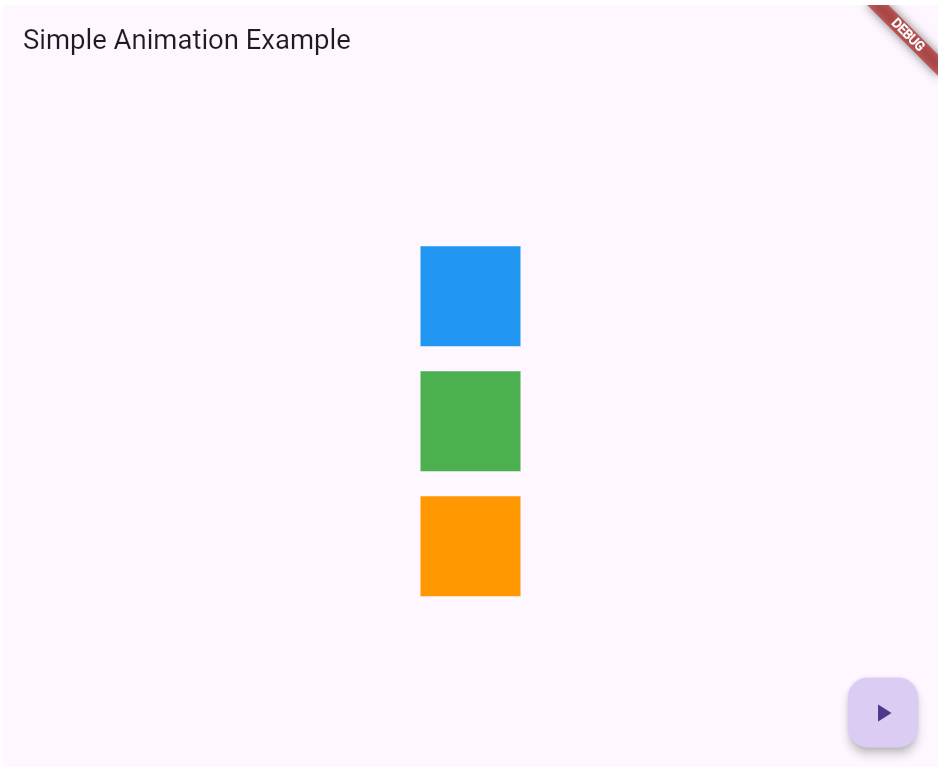
);

}

}

**OUTPUT:**

****

****

**WEEK-9**

**AIM:** Fetch data from a REST API

**DESCRIPTION:** Fetching data from a REST API is one of the most essential operations in modern mobile app development, allowing applications to communicate with external web services and display dynamic, real-time data. In Flutter, this process is commonly achieved using the http package, which provides easy-to-use methods for making network requests such as GET, POST, PUT, and DELETE. When fetching data from a REST API, the app sends an HTTP request to a server endpoint, and the server responds with data—typically in JSON (JavaScript Object Notation) format. This data is then decoded using Dart’s jsonDecode() method and mapped to custom model classes for easy use within the app’s UI

**PROGRAM:**

import 'package:flutter/material.dart';

import 'dart:convert';

import 'package:http/http.dart' as http;

void main() => runApp(MaterialApp(home: ApiDemo()));

class ApiDemo extends StatefulWidget {

  @override

  \_ApiDemoState createState() => \_ApiDemoState();

}

class \_ApiDemoState extends State<ApiDemo> {

  Map<String, dynamic>? data;

  @override

  void initState() {

    super.initState();

    http

        .get(Uri.parse('https://jsonplaceholder.typicode.com/posts/1'))

        .then((r) => setState(() => data = jsonDecode(r.body) as Map<String, dynamic>?));

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(title: const Text('API Demo')),

      body: Center(

        child: data == null

            ? const CircularProgressIndicator()

            : Text(data!['title'].toString()),

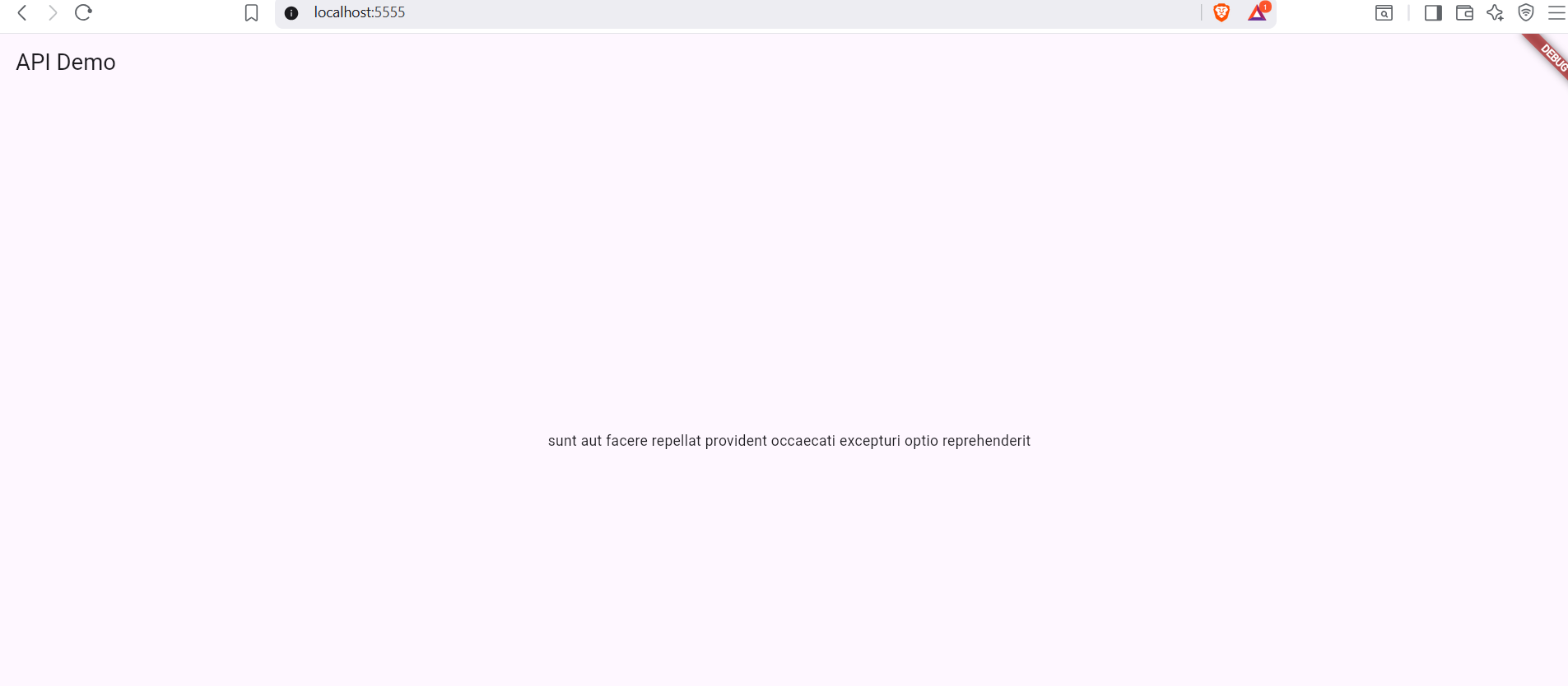
      ),

    );

  }

}

**OUTPUT:**

****

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

**DESCRIPTION:** In Flutter, displaying fetched data in a meaningful way involves fetching the data asynchronously from an API, local database, or any other source using Future, Stream, or async/await mechanisms. Once the data is retrieved, it can be displayed using scrollable widgets like ListView, GridView, or CustomScrollView, depending on the design requirements. To handle asynchronous operations efficiently, widgets like FutureBuilder or StreamBuilder are used, allowing the UI to reflect different states such as loading, error, or empty data. Each data item can be presented in a visually appealing way using widgets like Card, ListTile, or custom widgets, and enhanced with icons, images, or interactive elements. Styling and layout are managed using Container, Row, Column, and text formatting to make the content readable and user-friendly. This approach ensures that the data is displayed clearly, dynamically, and provides a smooth user experience in the Flutter application.

**PROGRAM:**

import 'package:flutter/material.dart';

import 'dart:convert';

import 'package:http/http.dart' as http;

void main() => runApp(MaterialApp(home: ApiDemo()));

class ApiDemo extends StatefulWidget {

  @override

  \_ApiDemoState createState() => \_ApiDemoState();

}

class \_ApiDemoState extends State<ApiDemo> {

  Map<String, dynamic>? data;

  @override

  void initState() {

    super.initState();

    fetchData();

  }

  void fetchData() async {

    final response =

        await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts/1'));

    setState(() {

      data = jsonDecode(response.body);

    });

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(title: const Text('API Data Display')),

      body: Center(

        child: data == null

            ? const CircularProgressIndicator()

            : Padding(

                padding: const EdgeInsets.all(16),

                child: Column(

                  mainAxisSize: MainAxisSize.min,

                  children: [

                    Text('ID: ${data!['id']}'),

                    const SizedBox(height: 8),

                    Text('Title: ${data!['title']}',

                        style: const TextStyle(fontWeight: FontWeight.bold)),

                    const SizedBox(height: 8),

                    Text('Body: ${data!['body']}'),

                  ],

                ),

              ),

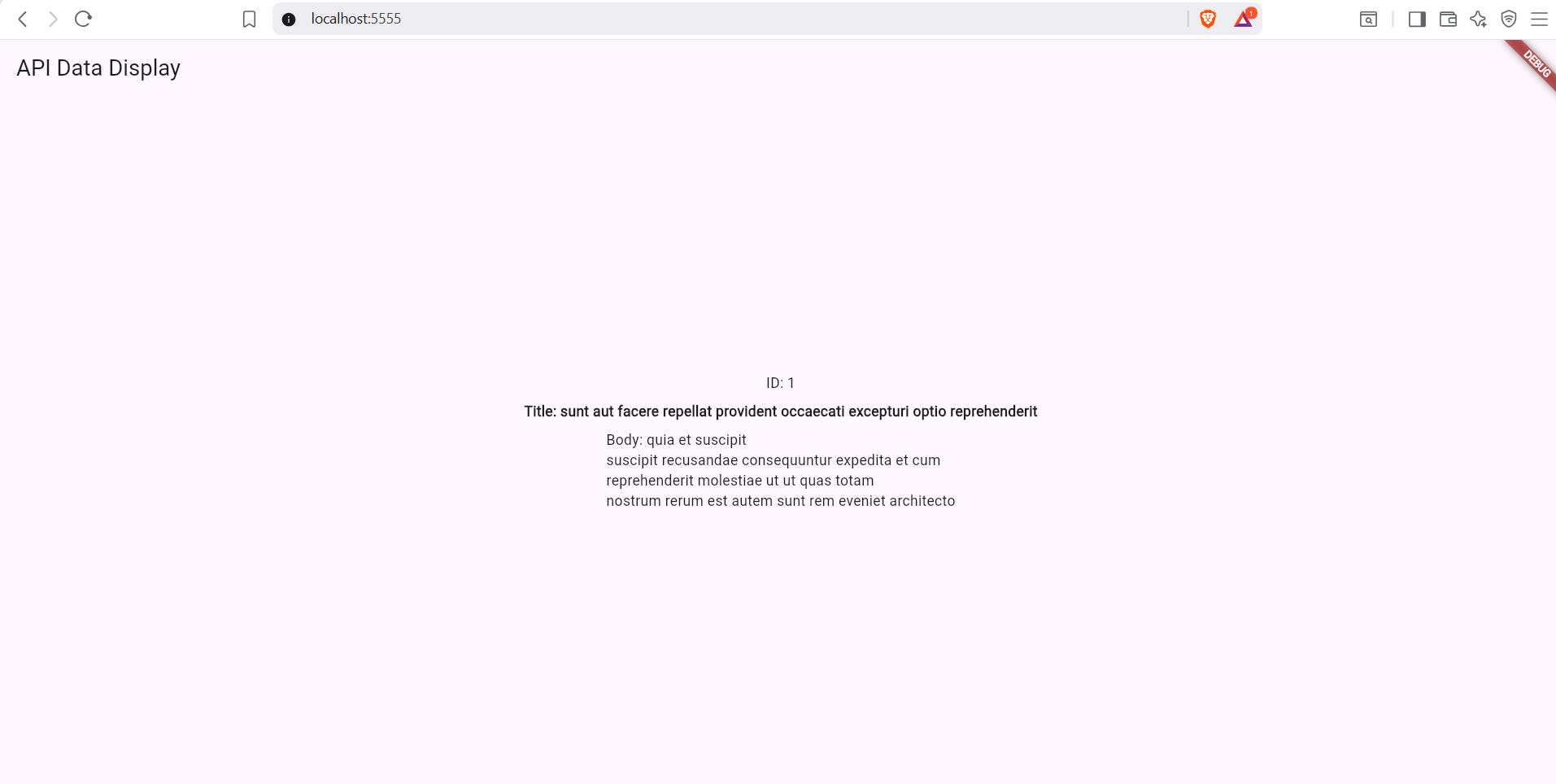
      ),

    );

  }

}

**OUTPUT:**

****

**WEEK-10**

**AIM:** Write unit tests for UI components.

**DESCRIPTION:** Unit testing UI components in Flutter involves verifying that individual widgets behave as expected in isolation, without running the full application. This is typically done using the flutter\_test package, which provides tools like testWidgets and WidgetTester to build widgets in a test environment, simulate user interactions, and check their rendered output. Developers write tests to ensure that widgets display the correct content, respond to input events, and handle different states such as loading, error, or empty data. By asserting the presence of specific text, buttons, or other UI elements, unit tests help catch errors early, maintain UI consistency, and improve the reliability of the application during development. Overall, unit testing UI components ensures that each widget performs its intended function and provides a stable, predictable user experience.

**PROGRAM 1:-**

**(Main.dart)**

import 'package:flutter/material.dart';

void main() => runApp(MaterialApp(home: GreetingWidget(name: 'Alice')));

class GreetingWidget extends StatelessWidget {

  final String name;

  const GreetingWidget({Key? key, required this.name}) : super(key: key);

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(title: const Text('Greeting Test Example')),

      body: Center(

        child: Text(

          'Hello, $name!',

          style: const TextStyle(fontSize: 24),

        ),

      ),

    );

  }

}

**PROGRAM 2:-**

**(Widget\_test.dart)**

import 'package:flutter\_test/flutter\_test.dart';

import 'package:flutter/material.dart';

import 'package:flutter\_application\_1/main.dart';

void main() {

  testWidgets('GreetingWidget shows correct text', (WidgetTester tester) async {

    // Build our widget and trigger a frame

    await tester.pumpWidget(const MaterialApp(

      home: GreetingWidget(name: 'Alice'),

    ));

    // Verify that the text is displayed correctly

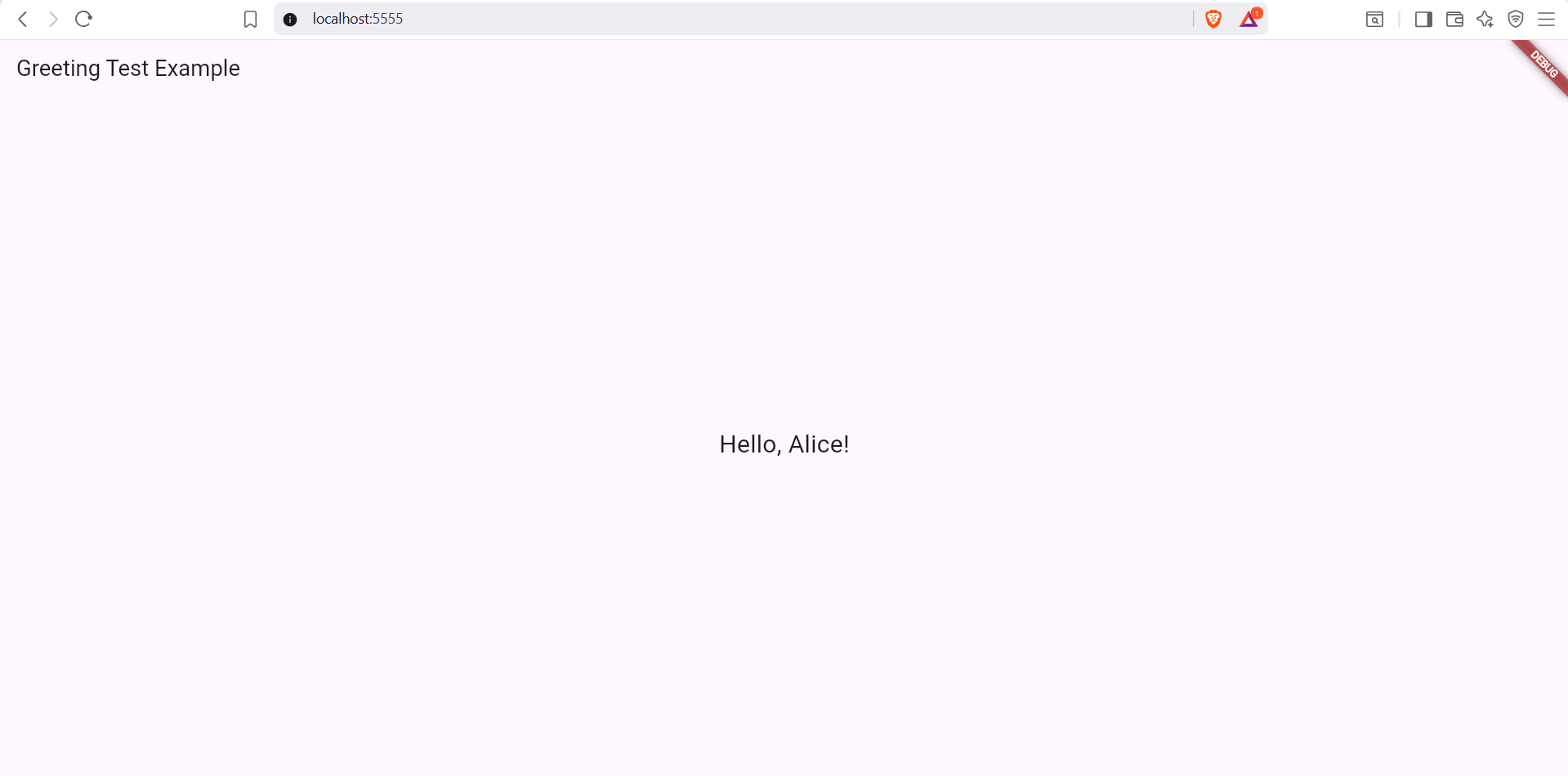
    expect(find.text('Hello, Alice!'), findsOneWidget);

    expect(find.text('Hello, John!'), findsNothing);

  });

}

**OUTPUT:**

****

**PS D:\5th sem\Flutter\flutter\_application\_1> flutter test**

**00:02 +1: All tests passed!**

**B) AIM:** Use Flutter's debugging tools to identify and fix issues.

**DESCRIPTION**: Flutter’s debugging tools help developers quickly identify and fix issues in their apps. Using debugPrint() and assertions, developers can track state changes and catch errors during development. The Flutter DevTools, especially the Flutter Inspector, allow inspection of the widget tree, layout, and rebuild counts, while the Debug Console displays logs and exceptions. These tools provide real-time insights, making it easier to understand app behavior and resolve problems efficiently.

**PROGRAM:**

import 'package:flutter/material.dart';

void main() {

  // Enable debugging print statements

  debugPrint('App starting...');

  runApp(MaterialApp(home: DebugDemo()));

}

class DebugDemo extends StatefulWidget {

  @override

  \_DebugDemoState createState() => \_DebugDemoState();

}

class \_DebugDemoState extends State<DebugDemo> {

  int counter = 0;

  void \_incrementCounter() {

    setState(() {

      counter++;

      // Debug print for tracking value changes

      debugPrint('Counter updated to: $counter');

      // Example assertion: Counter should never exceed 5

      assert(counter <= 5, 'Counter value exceeded 5!');

    });

  }

  @override

  Widget build(BuildContext context) {

    // Print widget rebuilds in console for debugging

    debugPrint('Widget rebuilt with counter = $counter');

    return Scaffold(

      appBar: AppBar(title: const Text('Debugging Demo')),

      body: Center(

        child: Column(

          mainAxisAlignment: MainAxisAlignment.center,

          children: [

            Text(

              'Counter Value: $counter',

              style: const TextStyle(fontSize: 24),

            ),

            const SizedBox(height: 20),

            ElevatedButton(

              onPressed: \_incrementCounter,

              child: const Text('Increment Counter'),

            ),

          ],

        ),

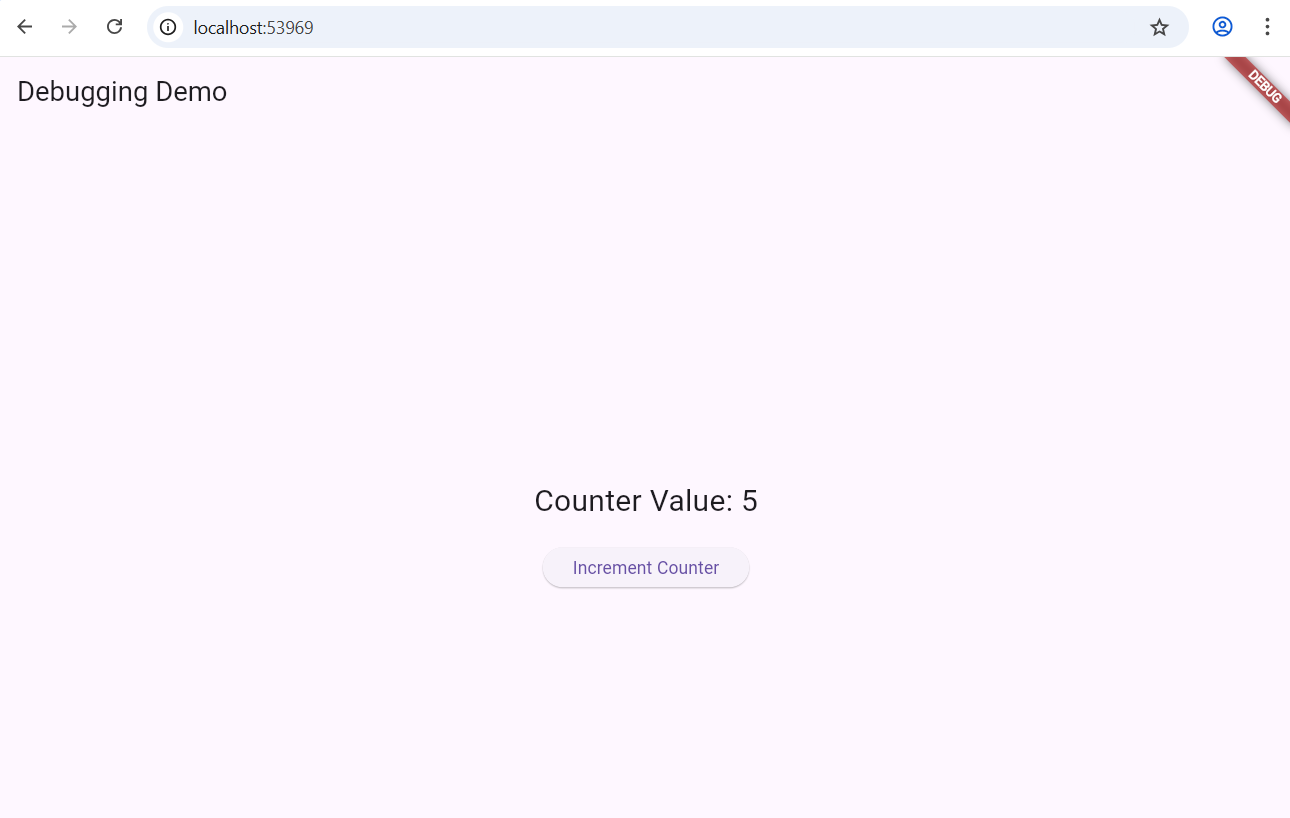
      ),

    );

  }

}

**OUTPUT:**

****

**PS D:\5th sem\Flutter\flutter\_application\_1> flutter run**

**A Dart VM Service on Chrome is available at: http://127.0.0.1:64735/F-MO2mqjs00=**

**App starting...**

**Widget rebuilt with counter = 0**

**The Flutter DevTools debugger and profiler on Chrome is available at: http://127.0.0.1:9101?uri=http://127.0.0.1:64735/F-MO2mqjs00=**

**Counter updated to: 1**

**Widget rebuilt with counter = 1**

**Counter updated to: 2**

**Widget rebuilt with counter = 2**

**Counter updated to: 3**

**Widget rebuilt with counter = 3**

**Counter updated to: 4**

**Widget rebuilt with counter = 4**

**Counter updated to: 5**

**Widget rebuilt with counter = 5**

**Counter updated to: 6**

**══╡ EXCEPTION CAUGHT BY GESTURE ╞═══════════════════════════════════════════════════════════════════**

**The following assertion was thrown while handling a gesture:**

**Assertion failed: file:///D:/5th%20sem/Flutter/flutter\_application\_1/lib/main.dart:20:14**

**counter <= 5**

**"Counter value exceeded 5!"**