

Tinkering Lab

(User Interface Design using Flutter)

Experiment 1 a) Install Flutter and Dart SDK.

Aim: To install Flutter and Dart SDK on your local machine.

System Requirements:

Hardware Requirements:

To install and run Flutter, your Windows environment must meet the following hardware requirements:

Requirement	Minimum	Recommended
x86_64 CPU Cores	4	8
Memory in GB	8	16
Display resolution in pixels	WXGA (1366 x 768)	FHD (1920 x 1080)
Free disk space in GB	11.0	60.0

Software Requirements

To write and compile Flutter code for Android, you must have the following version of Windows and the listed software packages.

Operating System

Flutter supports 64-bit version of Microsoft Windows 10 or later. These versions of Windows should include the required Windows PowerShell 5 or later.

Development Tools

Download and install the Windows version of the following packages:

- **Git for Windows 2.27 or later** to manage source code.
- **Android Studio 2023.3.1 (Jellyfish) or later** to debug and compile Java or Kotlin code for Android. Flutter requires the full version of Android Studio.

The developers of the preceding software provide support for those products. To troubleshoot installation issues, consult that product's documentation.

When you run the current version of flutter doctor, it might list a different version of one of these packages. If it does, install the version it recommends.

Configure a Text Editor or IDE

You can build apps with Flutter using any text editor or integrated development environment (IDE) combined with Flutter's command-line tools.

Using an IDE with a Flutter extension or plugin provides code completion, syntax highlighting, widget editing assists, debugging, and other features.

Popular options include:

- **Visual Studio Code 1.77 or later** with the Flutter extension for VS Code.
- **Android Studio 2023.3.1 (Jellyfish) or later** with the Flutter plugin for IntelliJ.
- **IntelliJ IDEA 2023.3 or later** with the Flutter plugin for IntelliJ.

Recommended: The Flutter team recommends installing Visual Studio Code 1.77 or later and the Flutter extension for VS Code. This combination simplifies installing the Flutter SDK.

Procedure

Step 1: Download Flutter SDK

1. Visit the Flutter official website.
2. Choose your operating system (Windows, macOS, Linux).
3. Download the Flutter SDK zip file.

Step 2: Extract Flutter SDK

1. Extract the downloaded zip file to a desired location on your system.
2. Add the flutter/bin directory to your system's PATH environment variable.

Step 3: Verify Installation

1. Open a command line interface.
2. Run the command:

```
flutter doctor
```

3. Follow any additional setup instructions provided by flutter doctor.

Step 4: Install Dart SDK

1. Dart SDK is bundled with Flutter, so no separate installation is required.
2. Verify Dart installation by running:

```
dart --version
```

Conclusion

You have successfully installed Flutter and Dart SDK on your machine. You are now ready to start developing Flutter applications.

Experiment 1 b) Write a simple Dart program to understand the language basics.

Aim: To write a simple Dart program to understand the language basics.

Here is a basic dart program that covers some basic concepts like variables, functions, conditionals, loops, and classes:

```
void main() {  
  // Variables  
  var name = 'Srinivas';  
  int age = 20;  
  double height = 5.9;  
  bool isAdult = age > 18 ? true : false;  
  
  // Print statements  
  print('Name: $name');  
  print('Age: $age');  
  print('Height: $height');  
  
  // Conditional statement  
  if (isAdult) {  
    print('$name is an adult.');  } else {  
    print('$name is not an adult.');  }  
  
  // Function call  
  greet(name);  
  
  // Loop  
  for (int i = 1; i <= 5; i++) {  
    print('Loop iteration $i');  
    if (i == 3) {  
      break;  
    }  
  }  
  
  // Friends array  
  List<String> friends = ['Bhanu', 'Amar', 'Amulya', 'Kiran', 'Sandeep'];  
  
  // Looping through the friends array  
  for (String friend in friends) {  
    print('Hello, $friend!');  }  
  
  // Creating an object of the Person class  
  Person person = Person(name, age, height);  
  person.introduce();  
  
  // Exception handling  
  try {  
    // Code that may throw an exception  
    int result = 10 ~/ 0; // Floor division by zero
```

```
    print('Result: $result');
  } catch (e) {
    // Handling the exception
    print('An error occurred: $e');
  }
}

// Function definition
void greet(String name) {
  print('Welcome to SDC, $name!');
}

// Class definition
class Person {
  String name;
  int age;
  double height;
  bool isAdult;

  // Constructor
  Person(this.name, this.age, this.height) : isAdult = age > 18;

  // Method
  void introduce() {
    print(
      'My name is $name, I am $age years old and my height is $height feet.';
    )
  }
}
```

Explanation:

1. Main Function:

- Entry point of the Dart program.

2. Variables:

- var name = 'Srinivas';: String variable for the name.
- int age = 20;: Integer variable for age.
- double height = 5.9;: Double variable for height.
- bool isAdult = age > 18 ? true : false;: Boolean variable using a ternary operator to check if age is greater than 18.

3. Print Statements:

- print('Name: \$name');: Prints the name.
- print('Age: \$age');: Prints the age.
- print('Height: \$height');: Prints the height.

4. Conditional Statement:

- Checks if isAdult is true.
- If true, prints that the person is an adult.
- Otherwise, prints that the person is not an adult.

5. Function Call:

- Calls the greet function with name as the argument.

6. Loop:

- for (int i = 1; i <= 5; i++): Loop from 1 to 5.
- print('Loop iteration \$i');: Prints the current iteration.
- if (i == 3) break;; Breaks the loop if i equals 3.

7. Friends Array:

- List<String> friends = ['Bhanu', 'Amar', 'Amulya', 'Kiran', 'Sandeep'];: List of friends' names.

8. Looping Through the Friends Array:

- for (String friend in friends): Iterates over each friend in the array.
- print('Hello, \$friend!');: Prints a greeting for each friend.

9. Object Creation:

- Creates an object person of the Person class.
- Calls the introduce method of the Person class.

10. Exception Handling:

- try: Code that may throw an exception.
- int result = 10 ~/ 0;; Floor division by zero (will throw an exception).
- catch (e): Catches the exception and prints an error message.

11. Function Definition:

- void greet(String name): Function that takes a string parameter.
- Prints a welcome message with the provided name.

12. Class Definition:

- class Person: Defines a Person class.
- **Properties:** name, age, height, and isAdult.
- **Constructor:** Initializes the properties and sets isAdult based on the age.
- **Method:** introduce: Prints the person's details.

Explanation:

1. Main Function:

- Entry point of the Dart program.

2. Variables:

- var name = 'Srinivas';: String variable for the name.
- int age = 20;; Integer variable for age.
- double height = 5.9;; Double variable for height.
- bool isAdult = age > 18 ? true : false;; Boolean variable using a ternary operator to check if age is greater than 18.

3. **Print Statements:**

- `print('Name: $name');` Prints the name.
- `print('Age: $age');` Prints the age.
- `print('Height: $height');` Prints the height.

4. **Conditional Statement:**

- Checks if `isAdult` is true.
- If true, prints that the person is an adult.
- Otherwise, prints that the person is not an adult.

5. **Function Call:**

- Calls the `greet` function with `name` as the argument.

6. **Loop:**

- `for (int i = 1; i <= 5; i++):` Loop from 1 to 5.
- `print('Loop iteration $i');` Prints the current iteration.
- `if (i == 3) break;` Breaks the loop if `i` equals 3.

7. **Friends Array:**

- `List<String> friends = ['Bhanu', 'Amar', 'Amulya', 'Kiran', 'Sandeep'];` List of friends' names.

8. **Looping Through the Friends Array:**

- `for (String friend in friends):` Iterates over each friend in the array.
- `print('Hello, $friend!');` Prints a greeting for each friend.

9. **Object Creation:**

- Creates an object `person` of the `Person` class.
- Calls the `introduce` method of the `Person` class.

10. **Exception Handling:**

- `try:` Code that may throw an exception.
- `int result = 10 ~/ 0;` Floor division by zero (will throw an exception).
- `catch (e):` Catches the exception and prints an error message.

11. **Function Definition:**

- `void greet(String name):` Function that takes a string parameter.
- Prints a welcome message with the provided name.

12. **Class Definition:**

- `class Person:` Defines a `Person` class.
- **Properties:** `name`, `age`, `height`, and `isAdult`.
- **Constructor:** Initializes the properties and sets `isAdult` based on the age.
- **Method:** `introduce:` Prints the person's details.

Output:

```
/> dart run
```

Name: Srinivas

Age: 20

Height: 5.9

Srinivas is an adult.

Welcome to SDC, Srinivas!

Loop iteration 1

Loop iteration 2

Loop iteration 3

Hello, Bhanu!

Hello, Amar!

Hello, Amulya!

Hello, Kiran!

Hello, Sandeep!

My name is Srinivas, I am 20 years old and my height is 5.9 feet.

Experiment 2a) Explore various Flutter widgets (Text, Image, Container, etc.).

Aim

To explore various Flutter widgets such as Text, Image, and Container.

Objective

In this lab experiment, we will explore various Flutter widgets such as Text, Image, and Container. Follow the steps below to set up a basic Flutter application and customize it according to your needs.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **IDE:** Visual Studio Code (Supported) or android studio (Supported) or IntelliJ IDEA (Supported).
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

The command creates a Flutter project directory called my_flutter_app that contains a simple demo app that uses Material Components.

2. Change to the Flutter project directory.

```
cd my_flutter_app
```

3. Open the lib/main.dart file in your Flutter project.
4. Replace the existing code with the following code snippet:

Source Code:

```
void main() {  
  
  runApp(const MainApp()); // Run the MainApp widget as the root of the application  
  
}  
  
class MainApp extends StatelessWidget {  
  
  const MainApp({super.key}); // Constructor for the MainApp widget  
  
  @override  
  Widget build(BuildContext context) {  
  
    return MaterialApp( // Create a MaterialApp widget  
  
      home: Scaffold(  

```

```
body: Center(  
  child: Column(  
    mainAxisAlignment: MainAxisAlignment.center, // Center the column vertically  
    children: [  
      const Text( // Display a text widget with the given text  
        'Welcome to Flutter!',  
        style: TextStyle(fontSize: 24),  
      ),  
      const SizedBox(height: 16),  
      Image.asset( // Display an image from the assets folder  
        'assets/images/flutter_logo.png',  
        width: 200, // Set the width, height of the image  
        height: 200,  
      ),  
      const SizedBox(height: 16),  
      Container( // Create a container widget  
        width: 200,  
        height: 50,  
        color: Colors.blue, // Set the background color of the container  
        child: const Center(  
          child: Text( // Display a text widget  
            'Start',  
            style: TextStyle(color: Colors.white), // Set the text color  
          ),  
        ),  
      ),  
    ],  
  ),  
)
```

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

5. Get the image flutter_logo.png from [wikipedia](https://www.wikipedia.org/) and save it in the assets/images/ directory of your Flutter project or replace the image path '**assets/images/flutter_logo.png**' with the actual path to your image file and then save the file.
6. Open the **pubspec.yaml** file in your Flutter project and add the following lines after flutter: to include the image asset in your project:

flutter:

uses-material-design: true

assets: <-- Add this line

- assets/images/flutter_logo.png <-- Add this line

Save the file.

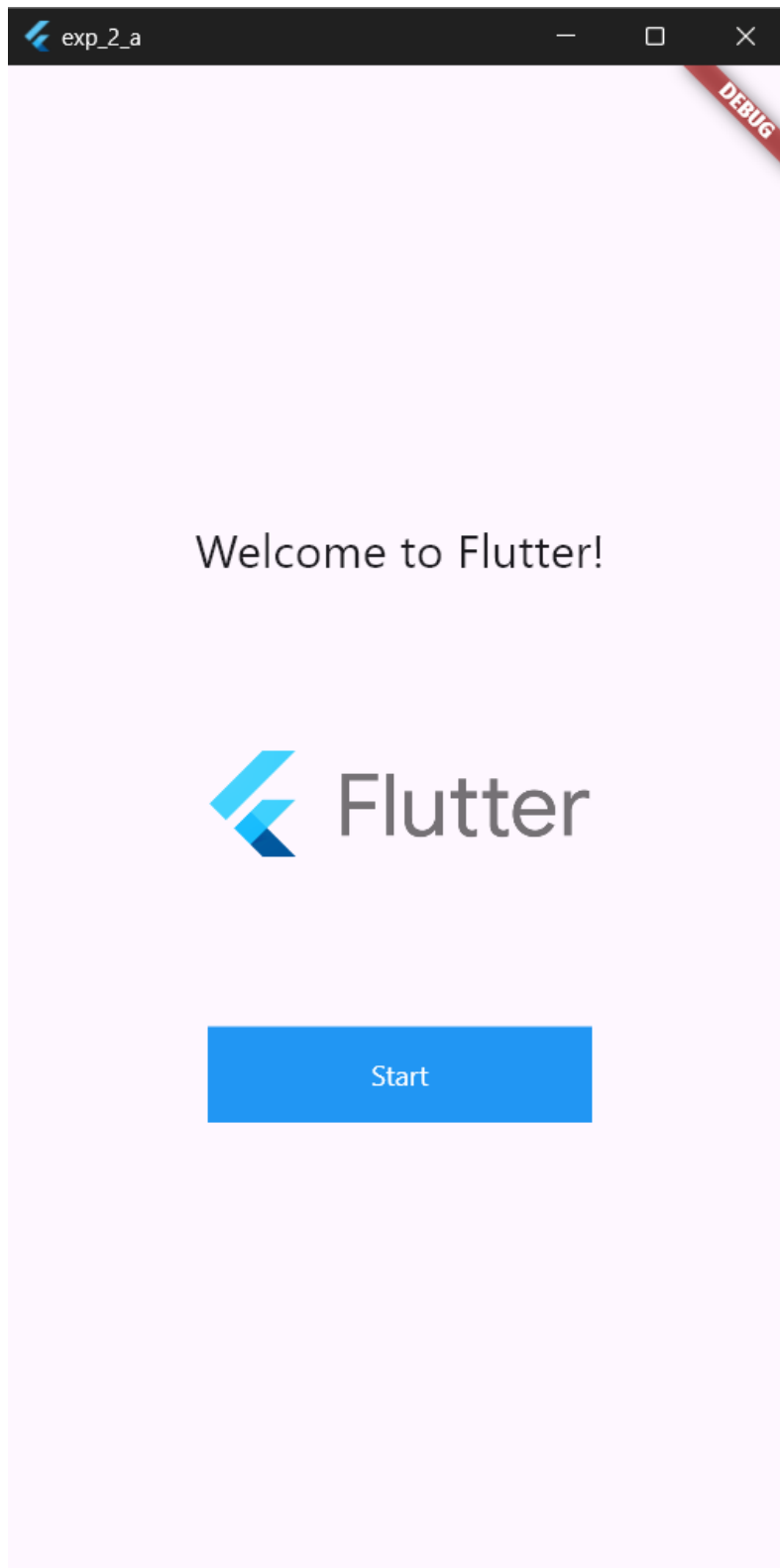
7. Run your Flutter project using the following command:

```
flutter run
```

Select the appropriate device to run the app.

8. Enter r to hot reload the app and see the changes you made to the code.
Enter q to quit the app.

Output



Conclusion

You have successfully set up a basic Flutter application and explored various Flutter widgets. Feel free to customize the code further and experiment with different widgets to create your desired UI.

Experiment 2 b) Implement different layout structures using Row, Column, and Stack widgets.

Aim: To implement different layout structures using Row, Column, and Stack widgets.

Source Code:

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

void main() {

  runApp(const MainApp());

}

class MainApp extends StatelessWidget {

  const MainApp({super.key});

  @override

  Widget build(BuildContext context) {

    return const MaterialApp(

      home: Scaffold(

        body: LayoutApp(),

      ),

    );

  }

}

class LayoutApp extends StatelessWidget {

  const LayoutApp({super.key});

  @override

  Widget build(BuildContext context) {

    return Column(

      mainAxisAlignment: MainAxisAlignment.center,

      children: [

        const Text('I\'m in a Coloum and Centered. The below is a row.'),

      ],

    );

  }

}
```

```
const SizedBox(height: 20),

Row(

  mainAxisAlignment: MainAxisAlignment.spaceEvenly,

  children: [

    for (var color in [Colors.red, Colors.green, Colors.blue])

      Container(

        width: 100,

        height: 100,

        color: color,

      ),

  ],

),

const SizedBox(height: 20),

Stack(

  alignment: Alignment.center,

  children: [

    Container(

      width: 300,

      height: 200,

      color: Colors.yellow,

    ),

    const Text(

      'Stacked on Yellow Box',

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

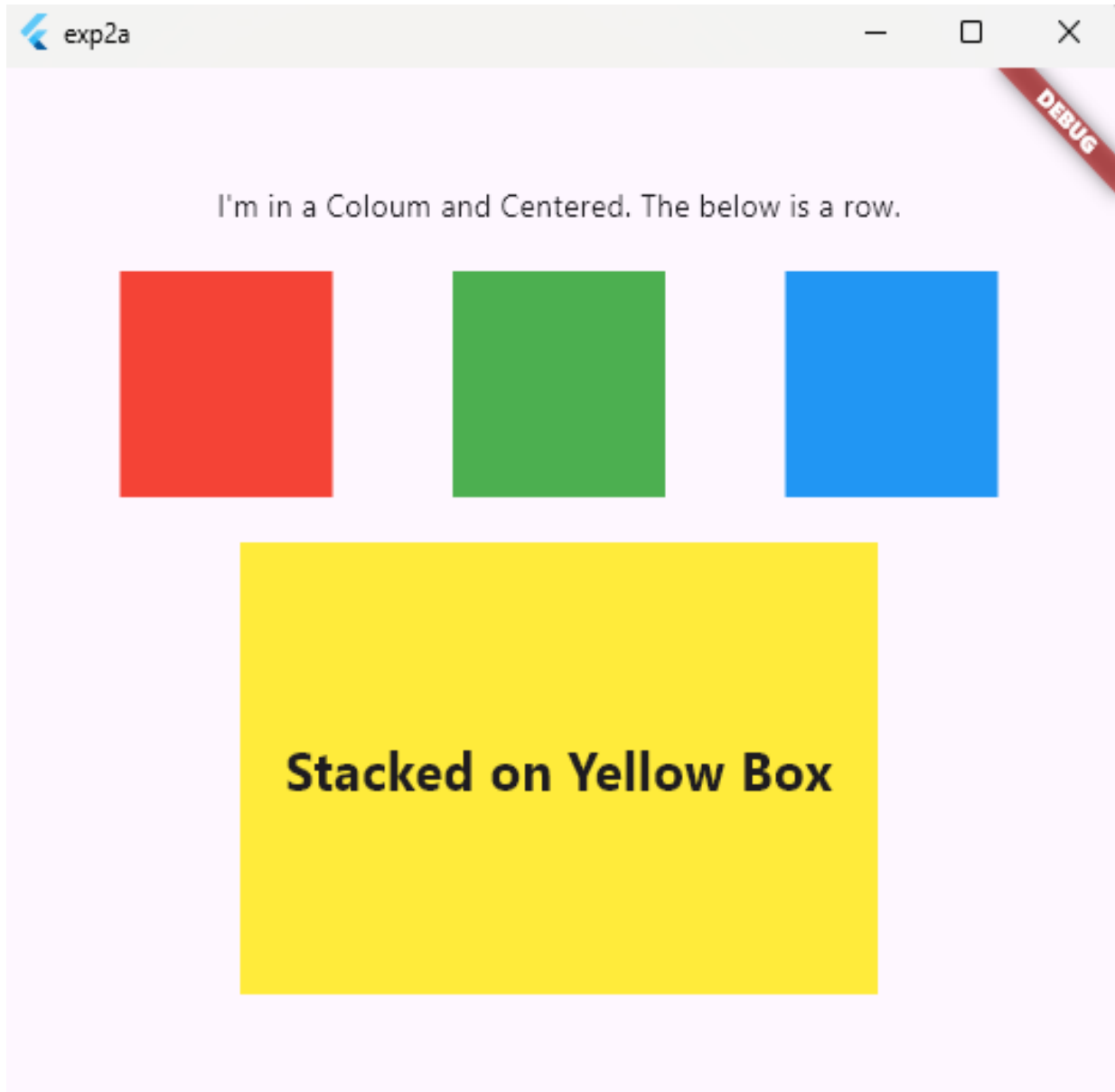
    ),

  ],

),
```

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

Output:



Experiment 3. a) Design a responsive UI that adapts to different screen sizes.

Aim

To design a responsive UI that adapts to different screen sizes.

Objective

The objective of this experiment is to create a Flutter project that demonstrates the concept of responsive UI design. The UI should adjust its layout and appearance based on the screen size of the device.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **IDE:** Visual Studio Code (Supported) or android studio (Supported) or IntelliJ IDEA (Supported).
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

This command will create a Flutter project directory called my_flutter_app that contains a simple demo app using Material Components.

2. Change to the Flutter project directory:
3.

```
cd my_flutter_app
```
4. Open the lib/main.dart file in your Flutter project.
5. Replace the existing code with the following code snippet:

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

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

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
```

```
        title: 'Responsive UI',

        home: ResponsiveHomePage(),

    );
}
}

class ResponsiveHomePage extends StatelessWidget {

  const ResponsiveHomePage({super.key});

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: const Text('Responsive UI'),

      ),

      body: LayoutBuilder(

        builder: (BuildContext context, BoxConstraints constraints) {

          if (constraints.maxWidth >= 600) {

            // For Wide Screen (Tablet/Desktop)

            return _buildWideContainers();

          } else {

            // For Narrow Screen (Phone)

            return _buildNarrowContainers();

          }

        },

      ),

    );

  }

  Widget _buildWideContainers() {

    return Container(

      color: Colors.blue,
```

```
child: const Center(  
  child: Text(  
    'You are using a Wide screen!',  
    style: TextStyle(  
      fontSize: 26, color: Colors.white, fontWeight: FontWeight.bold),  
    ),  
  ),  
);  
}  
  
Widget _buildNarrowContainers() {  
  return Container(  
    color: Colors.green,  
    child: const Center(  
      child: Text(  
        'You are using a Narrow screen!',  
        style: TextStyle(fontSize: 16, color: Colors.white),  
      ),  
    ),  
  );  
}  
}
```

Save the file.

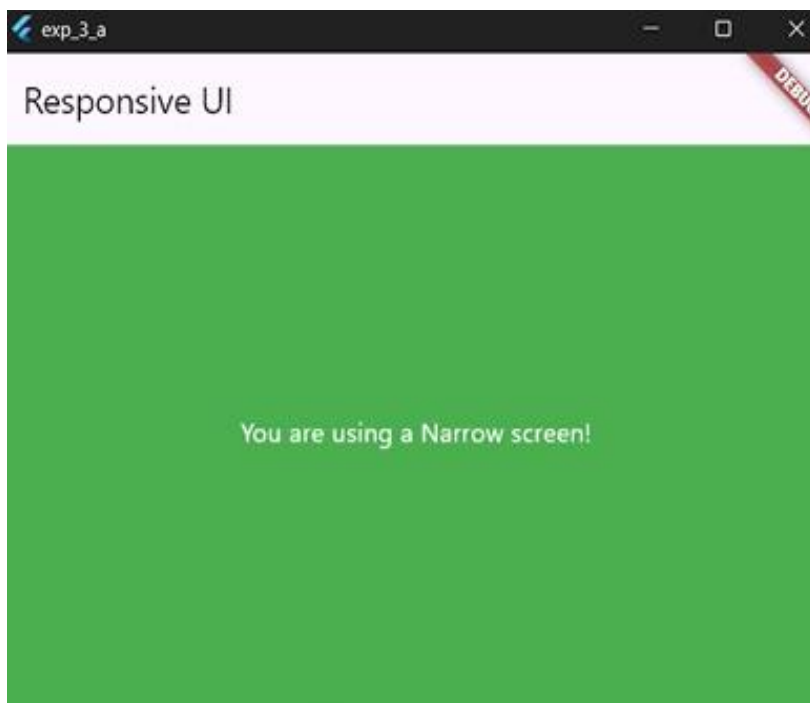
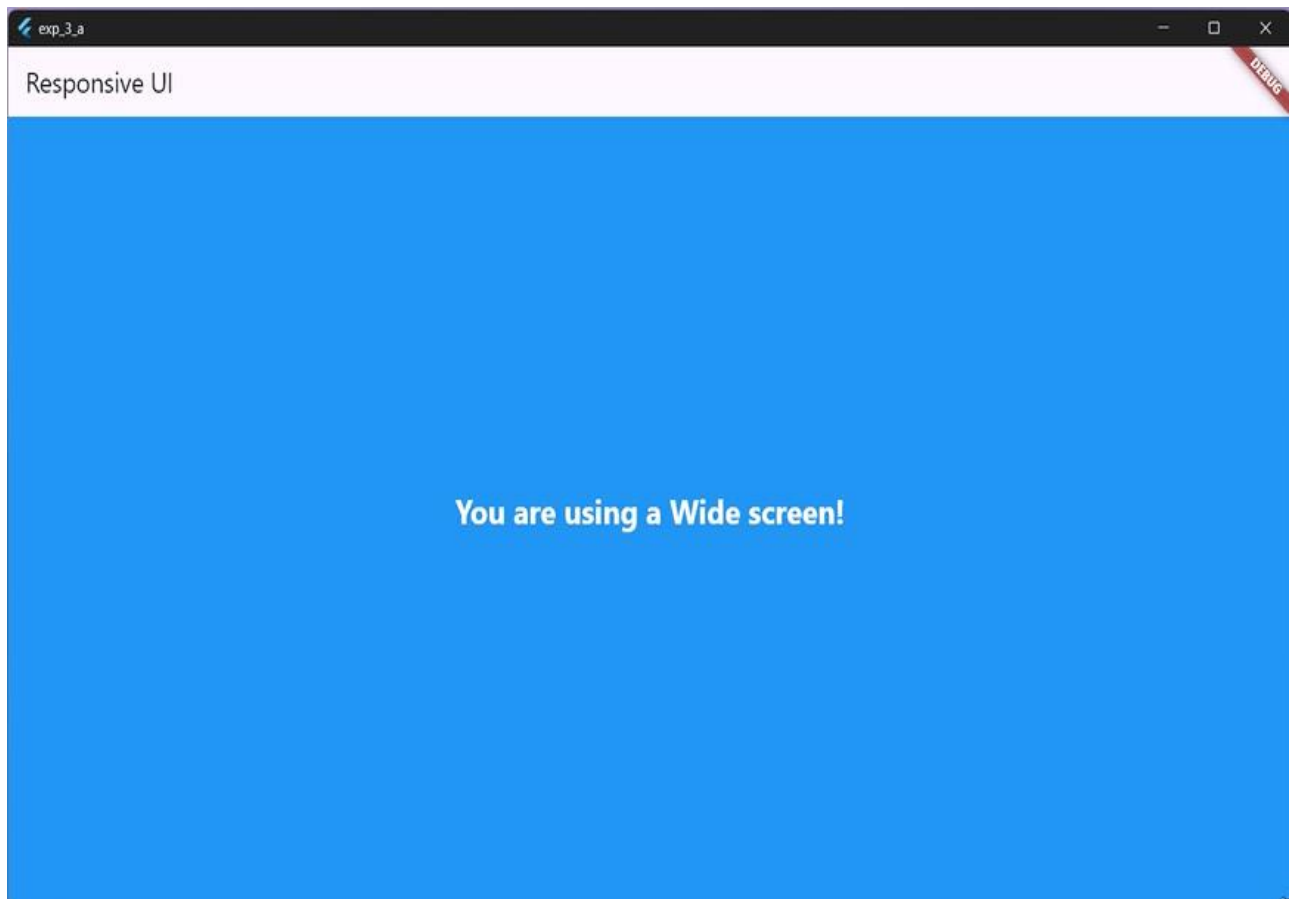
6. Run your Flutter project using the following command:

```
flutter run
```

Select the appropriate device to run the app.

7. To hot reload the app and see the changes you made to the code, enter r. To quit the app, enter q.

Output



Conclusion

In this experiment, we successfully designed a responsive UI that adapts to different screen sizes using Flutter. The UI layout and appearance adjust dynamically based on the screen size of the device, providing an optimal user experience.

Experiment 3 b) Implement media queries and breakpoints for responsiveness.

Aim

The aim of this experiment is to implement media queries and breakpoints in a Flutter project to achieve responsiveness.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **IDE:** Visual Studio Code (Supported) or android studio (Supported) or IntelliJ IDEA (Supported).
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

The command creates a Flutter project directory called my_flutter_app that contains a simple demo app that uses Material Components.

2. Change to the Flutter project directory.

```
cd my_flutter_app
```

3. Open the lib/main.dart file in your Flutter project.

4. Replace the existing code with the following code snippet:

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

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

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: ResponsiveHomePage(),
    );
  }
}
```

```
}

class ResponsiveHomePage extends StatelessWidget {

  const ResponsiveHomePage({super.key});

  static const colorCodes = (

    body: Color(0xFFFF8E287), // Sweet Corn approx.

    navigation: Color(0xFFC5ECCE), // Padua approx.

    pane: Color(0xFFEEE2BC), // Chamois approx

  );

  static const _style = TextStyle(fontSize: 20, fontWeight: FontWeight.bold);

  static const body = Center(child: Text('Body', style: _style));

  static const navigation = Center(child: Text('Navigation', style: _style));

  static const panes = Center(child: Text('Pane', style: _style));

  @override

  Widget build(BuildContext context) {

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

    return Scaffold(

      appBar: AppBar(

        title: () {

          // immediately-invoked function expression (IIFE)

          if (screenWidth < 600) {

            return const Text('Responsive UI - Phone');

          } else if (screenWidth < 840) {

            return const Text('Responsive UI - Tablet');

          } else if (screenWidth < 1200) {

            return const Text('Responsive UI - Landscape');

          } else {

            return const Text('Responsive UI - Large Desktop');

          }

        }(),

      ),
```

```
    ),  
    body: () {  
      // immediately-invoked function expression (IIFE)  
      if (screenWidth < 600) { <-- breakpoint  
        return buildCompactScreen();  
      } else if (screenWidth < 840) { <-- breakpoint  
        return buildMediumScreen();  
      } else if (screenWidth < 1200) { <-- breakpoint  
        return buildExpandedScreen();  
      } else { <-- breakpoint  
        return buildLargeScreen();  
      }  
    }(),  
  );  
}  
  
Widget buildCompactScreen() {  
  return Column(  
    children: [  
      Expanded(  
        child: Container(color: colorCodes.body, child: body),  
      ),  
      Container(height: 80, color: colorCodes.navigation, child: navigation),  
    ],  
  );  
}  
  
Widget buildMediumScreen() {  
  return Row(  
    children: [  
      Container(width: 80, color: colorCodes.navigation),  
    ],  
  );  
}
```

```
Expanded(  
  child: Container(color: colorCodes.body, child: body),  
),  
],  
);  
}  
  
Widget buildExpandedScreen() {  
  return Row(  
    children: [  
      Container(width: 80, color: colorCodes.navigation),  
      Container(width: 360, color: colorCodes.body, child: body),  
      Expanded(  
        child: Container(color: colorCodes.pane, child: panes),  
      ),  
    ],  
  );  
}  
  
Widget buildLargeScreen() {  
  return Row(  
    children: [  
      Container(color: colorCodes.navigation, width: 360, child: navigation),  
      Container(width: 360, color: colorCodes.body, child: body),  
      Expanded(  
        child: Container(color: colorCodes.pane, child: panes),  
      ),  
    ],  
  );  
}  
}
```

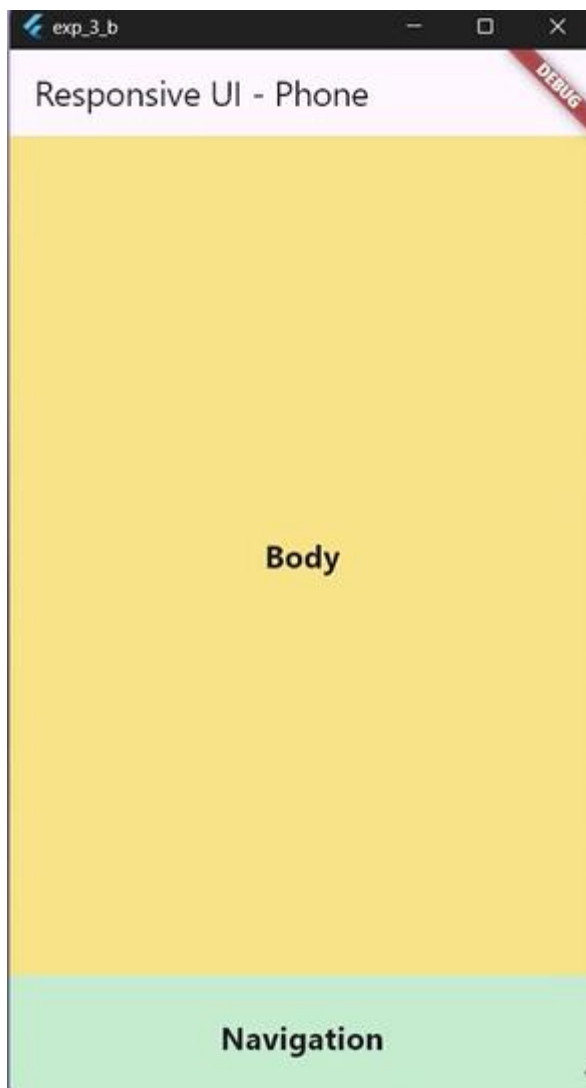
5. Save the file.
6. Run your Flutter project using the following command:

```
flutter run
```

Select the appropriate device to run the app.

7. During the app execution, you can use the following commands:
 - Enter r to hot reload the app and see the changes you made to the code.
 - Enter q to quit the app.

Output:







Conclusion

In this experiment, we learned how to implement media queries and breakpoints in a Flutter project to achieve responsiveness. By using different screen widths, we were able to create a layout that adapts to different devices. This allows our app to provide a consistent user experience across various screen sizes.

Experiment 4. a) Set up navigation between different screens using Navigator.

Aim Set up navigation between different screens using Navigator

Objective

To learn how to navigate between screens in a Flutter app using the Navigator class.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **IDE:** Visual Studio Code (Supported) or android studio (Supported) or IntelliJ IDEA (Supported).
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

The command creates a Flutter project directory called my_flutter_app that contains a simple demo app that uses Material Components.

2. Change to the Flutter project directory.

```
cd my_flutter_app
```

3. Open the lib/main.dart file in your Flutter project.

4. Replace the existing code with the following code snippet:

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

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

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: HomeScreen(),
    ); } }
```

```
class HomeScreen extends StatelessWidget {  
  const HomeScreen({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(title: const Text('Home')),  
      body: Center(  
        child: ElevatedButton(  
          onPressed: () {  
            Navigator.push(  
              context,  
              MaterialPageRoute(builder: (context) => const SecondScreen()),  
            );  
          },  
          child: const Text('Go to Second Screen'),  
        ), ), ); } }  
  
class SecondScreen extends StatelessWidget {  
  const SecondScreen({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(title: const Text('Second Screen')),  
      body: Center(  
        child: ElevatedButton(  
          onPressed: () {  
            Navigator.pop(context);  
          },  
          child: const Text('Go Back'),  
        ), ), ); } }
```

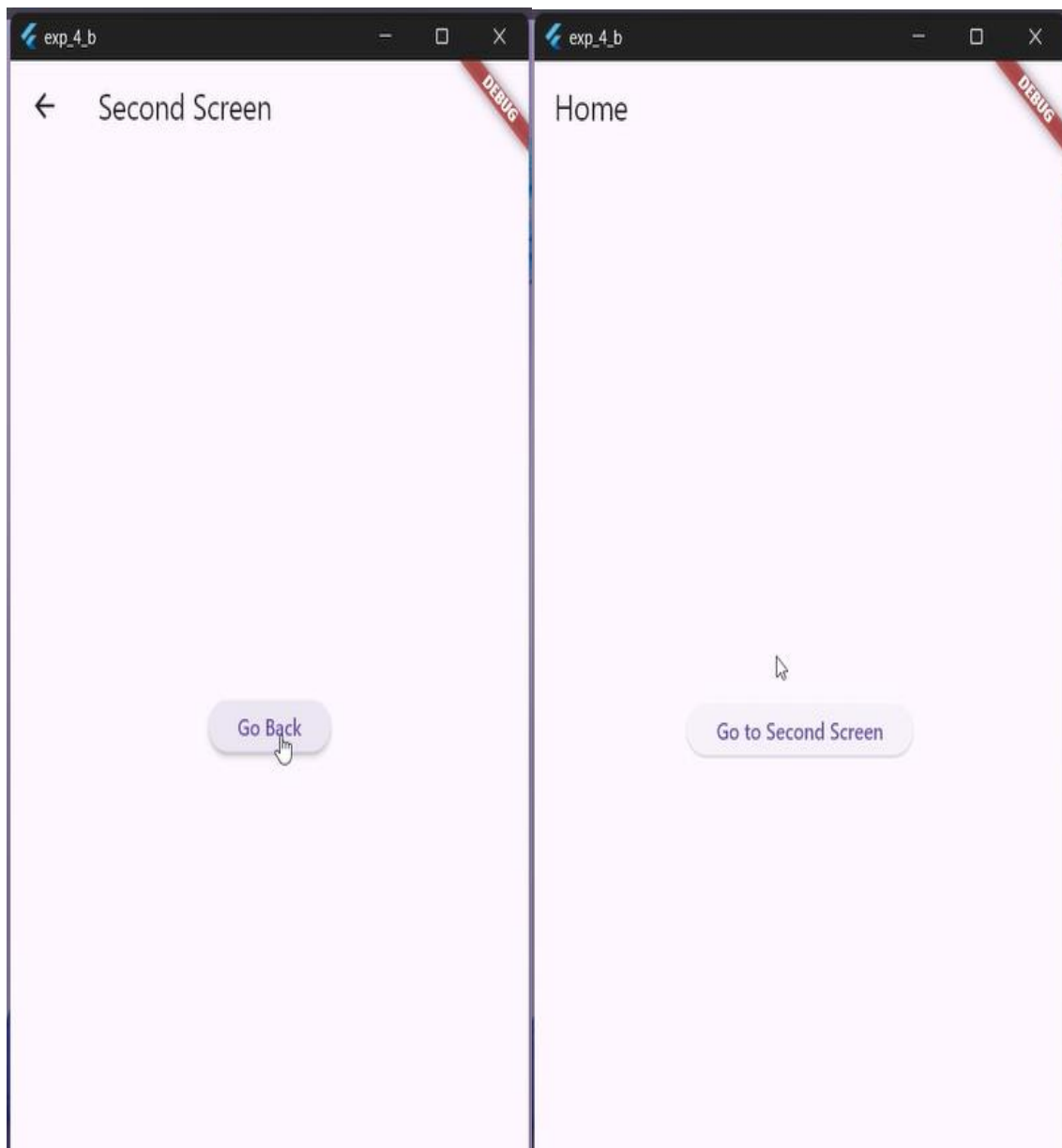
5. Save the file.
6. Run your Flutter project using the following command:

```
flutter run
```

Select the appropriate device to run the app.

7. During the app execution, you can use the following commands:
 - Enter r to hot reload the app and see the changes you made to the code.
 - Enter q to quit the app.

Output



Conclusion

In this lab, we learned how to set up navigation between different screens in a Flutter app using the Navigator class.

Experiment 4b) Implement navigation with named routes.

Aim

Implement navigation with named routes in a Flutter app.

Objective

To understand how to navigate between different screens in a Flutter app using named routes.
Understand deep linking in Flutter apps.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **IDE:** Visual Studio Code (Supported) or android studio (Supported) or IntelliJ IDEA (Supported).
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

The command creates a Flutter project directory called my_flutter_app that contains a simple demo app that uses Material Components.

2. Change to the Flutter project directory.

```
cd my_flutter_app
```

3. Open the lib/main.dart file in your Flutter project.

4. Replace the existing code with the following code snippet:

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

```
void main() {
```

```
  runApp(const MainApp());
```

```
}
```

```
class MainApp extends StatelessWidget {
```

```
  const MainApp({super.key});
```

```
  @override
```

```
  Widget build(BuildContext context) {
```

```
    return MaterialApp(
```

```
      initialRoute: '/',
```

```
routes: {  
  '/': (context) => const HomeScreen(),  
  '/second': (context) => const SecondScreen(),  
},  
);  
}  
}  
  
class HomeScreen extends StatelessWidget {  
  const HomeScreen({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(title: const Text('Home')),  
      body: Center(  
        child: ElevatedButton(  
          onPressed: () {  
            Navigator.pushNamed(context, '/second');  
          },  
          child: const Text('Go to Second Screen'),  
        ),  
      ),  
    );  
  }  
}  
  
class SecondScreen extends StatelessWidget {  
  const SecondScreen({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  

```

```
    appBar: AppBar(title: const Text('Second Screen')),  
    body: Center(  
      child: ElevatedButton(  
        onPressed: () {  
          Navigator.pop(context);  
        },  
        child: const Text('Go Back'),  
      ),  
    ),  
  );  
}
```

5. Save the file.

6. Run your Flutter project using the following command:

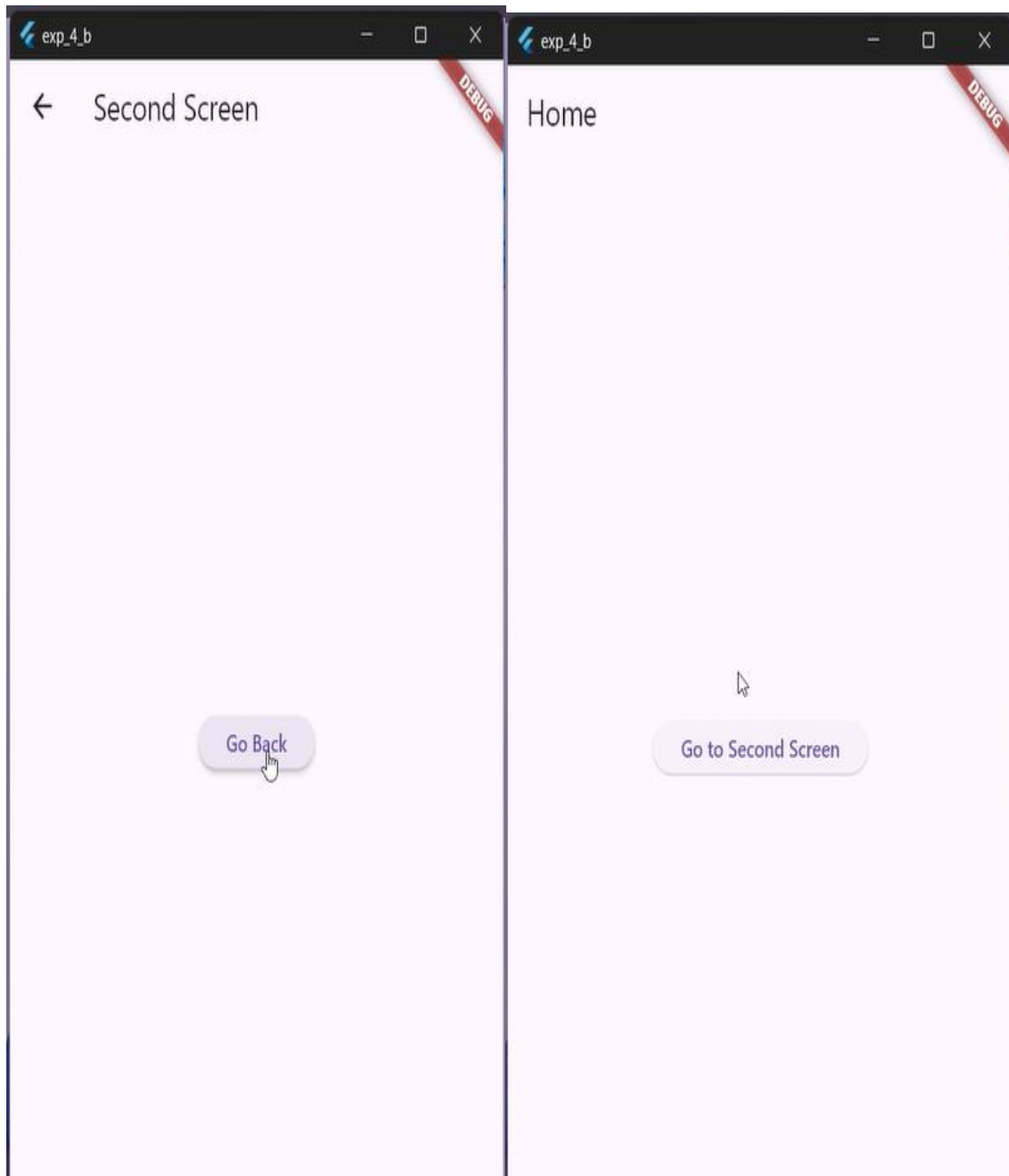
```
flutter run
```

Select the appropriate device to run the app.

7. During the app execution, you can use the following commands:

- Enter r to hot reload the app and see the changes you made to the code.
- Enter q to quit the app.

Output:



Conclusion

In this lab, we learned how to navigate between different screens in a Flutter app using named routes.

We created two screens, HomeScreen and SecondScreen, and used the Navigator class to navigate between them. We defined the routes for the screens in the MaterialApp widget using the routes property. We used the Navigator.pushNamed method to navigate to the SecondScreen and the Navigator.pop method to go back to the previous screen.

By following this lab, you should now have a good understanding of how to implement navigation with named routes in a Flutter app.

Experiment 5a) Learn about stateful and stateless widgets.

Stateless Widgets

Stateless widgets are immutable, meaning their properties can't change once they're built. They are typically used for widgets that don't require any mutable state (i.e., they don't change over time or in response to user interactions). Examples of stateless widgets include Text, Icon, and Container.

Key Points:

- Immutable
- Used for static content
- Lightweight and simple to use

Structure:

1. Stateless widget class that extends StatelessWidget.
2. build() method that returns the widget's UI based on the input properties.

```
class MyWidget extends StatelessWidget {  
  const MyWidget({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return /* ... Widget tree */;  
  }  
}
```

Example:

```
import 'package:flutter/material.dart';  
  
void main() => runApp(const MaterialApp(home: MyStatelessWidget()));  
  
class MyStatelessWidget extends StatelessWidget {  
  const MyStatelessWidget({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(title: const Text('Stateless Widget')),  
      body: const Center(  
        child: Text('Hello, I am a stateless widget!'),  
      ),  
    );  
  }  
}
```

Output:



Stateful Widgets

Stateful widgets are dynamic and can change their state during their lifetime. They are used when the widget needs to update based on user interactions or other factors. Examples of stateful widgets include Checkbox, Radio, Slider, and TextField.

Stateful widgets consist of two classes:

1. A StatefulWidget class that is immutable and can be used to create an instance of the widget.
2. A State class that contains the mutable state for that widget.

Key Points:

- Mutable state
- Used for dynamic content
- More complex than stateless widgets

Structure:

1. StatefulWidget class that extends StatefulWidget.
2. State class that extends State and manages the widget's mutable state.
3. createState() method that creates an instance of the State class.
4. build() method in the State class that returns the widget's UI based on the current state.

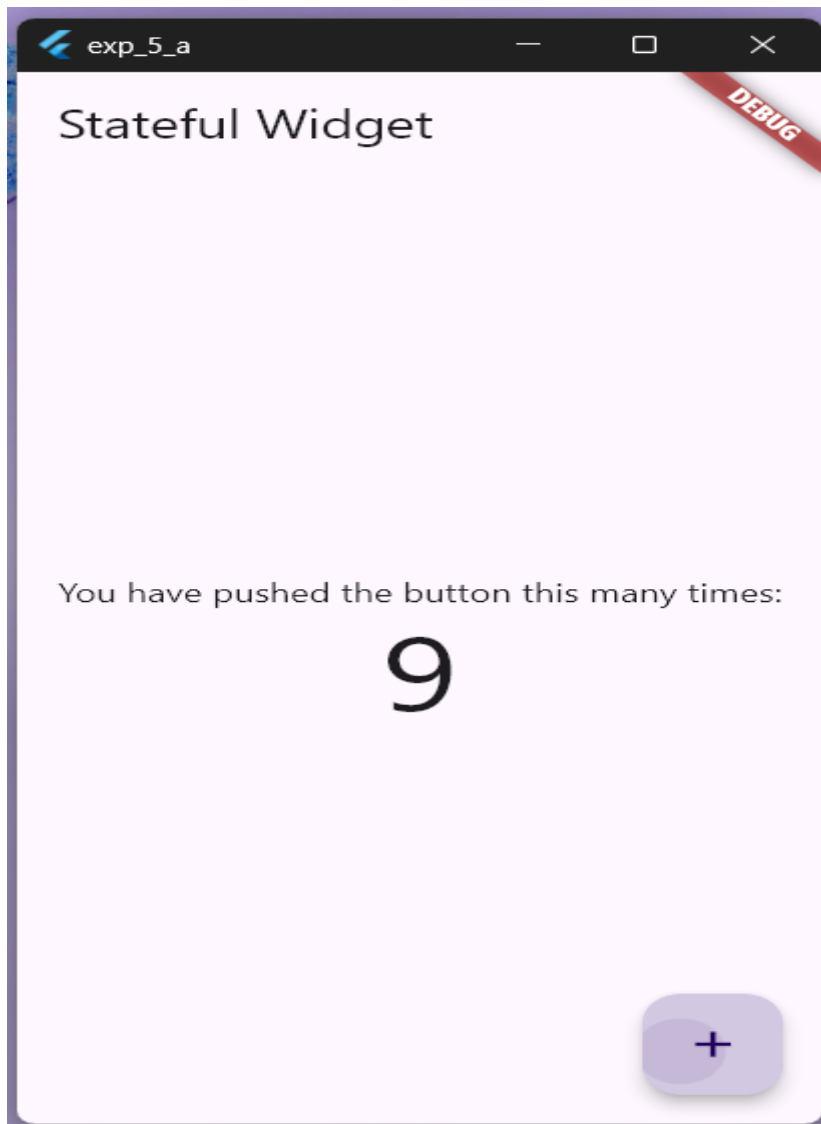
```
class MyWidget extends StatefulWidget {  
  const MyWidget({super.key});  
  
  @override  
  State<MyWidget> createState() => _MyWidgetState();  
}  
  
class _MyWidgetState extends State<MyWidget> {  
  @override  
  Widget build(BuildContext context) {  
    return /* ... Widget tree and stateManagment if needed */;  
  }  
}
```

Example:

```
import 'package:flutter/material.dart';  
  
void main() => runApp(const MaterialApp(home: MyStatefulWidget()));  
  
class MyStatefulWidget extends StatefulWidget {  
  const MyStatefulWidget({super.key});  
  
  @override  
  State<MyStatefulWidget> createState() => _MyStatefulWidgetState();  
}  
  
class _MyStatefulWidgetState extends State<MyStatefulWidget> {  
  int _counter = 0;  
  
  void _incrementCounter() {  
    setState(() {  
      _counter++;  
    });  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(title: const Text('Stateful Widget')),
```

```
body: Center(  
  child: Column(  
    mainAxisAlignment: MainAxisAlignment.center,  
    children: <Widget>[  
      const Text('You have pushed the button this many times:'),  
      Text('$_counter', style: Theme.of(context).textTheme.displayLarge),  
    ],  
  ),  
,  
floatingActionButton: FloatingActionButton(  
  onPressed: _incrementCounter,  
  tooltip: 'Increment',  
  child: const Icon(Icons.add),  
)  
);  
}
```

Output:



Stateful vs Stateless Widgets:

Feature	Stateful Widget	Stateless Widget
Mutability	Mutable	Immutable
State Management	Manages its own state	Does not manage any state
Lifecycle Methods	Complex lifecycle methods (initState(), dispose(), setState())	Simple lifecycle (build())
Reactivity	Can react to changes and update UI dynamically	Cannot react to changes, static UI
Performance	Slightly heavier due to state management	Lighter, more efficient
Use Cases	Dynamic content that changes over time	Static content that does not change
Examples	Forms, animations, interactive elements	Text, icons, containers
Rebuild Frequency	Rebuilt when setState() is called	Rebuilt only when parameters or parent changes
Initialization	Uses createState() to create state object	Directly creates widget through build() method
Complexity	More complex due to state handling	Simpler and easier to use

Key Differences:

- **State Management:** Stateless widgets do not manage any state, while stateful widgets have an internal state that can change over time.
- **Lifecycle:** Stateful widgets have a more complex lifecycle with additional methods such as initState(), setState(), and dispose(), whereas stateless widgets only have the build() method.

Use Cases:

- **Stateless Widgets:** Use these for UI elements that do not change, such as static text, icons, and layout containers.
- **Stateful Widgets:** Use these for UI elements that need to change dynamically based on user input or other factors, such as forms, sliders, and interactive elements.

Experiment 5b) Implement state management using set State and Provider.

Aim

Implement state management using setState and Provider

Objective

To understand how to manage the state of a Flutter app using the setState method and the Provider package.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **IDE:** Visual Studio Code (Supported) or android studio (Supported) or IntelliJ IDEA (Supported).
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

The command creates a Flutter project directory called my_flutter_app that contains a simple demo app that uses Material Components.

2. Change to the Flutter project directory.

```
cd my_flutter_app
```

3. Create a new Dart file called counter_model.dart in the lib directory of your Flutter project.
touch lib/counter_model.dart

4. Open the pubspec.yaml file in your Flutter project and add the following lines after flutter: to include the image asset in your project:

dependencies:

flutter:

 sdk: flutter

provider: ^6.0.0 <-- Add this line

Save the file.

Add the following code snippet to the counter_model.dart file:

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

```
// CounterModel class that extends ChangeNotifier
```

```
class CounterModel extends ChangeNotifier {
```

```
int _counter = 0;

// Getter for the counter value

int get counter => _counter;

// Method to increment the counter and notify listeners

void increment() {

    _counter++;

    notifyListeners();

} }
```

Save the file.

5. Open the lib/main.dart file in your Flutter project.
6. Replace the existing code with the following code snippet:

```
// Import necessary packages

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import 'counter_model.dart';

// Main function

void main() {

    runApp(

        // Wrap the app with ChangeNotifierProvider to provide the CounterModel

        ChangeNotifierProvider(

            create: (context) => CounterModel(),

            child: const MyApp(),

        ),

    );

}

// MyApp widget

class MyApp extends StatelessWidget {

    const MyApp({super.key});

    @override
```

```
Widget build(BuildContext context) {  
  
  return const MaterialApp(  
  
    home: CounterPage(),  
  
  ); } }  
  
// CounterPage widget  
  
class CounterPage extends StatelessWidget {  
  
  const CounterPage({super.key});  
  
  @override  
  
  Widget build(BuildContext context) {  
  
    return Scaffold(  
  
      appBar: AppBar(  
  
        title: const Text('Counter App'),  
  
      ),  
  
      body: Center(  
  
        child: Consumer<CounterModel>(  
  
          builder: (context, counterModel, child) {  
  
            return Column(  
  
              mainAxisAlignment: MainAxisAlignment.center,  
  
              children: [  
  
                const Text('You have pushed the button this many times:'),  
  
                Text(  
  
                  '${counterModel.counter}',  
  
                  style: Theme.of(context).textTheme.displayLarge,  
  
                ), ], ); }, ), ),  
  
      floatingActionButton: FloatingActionButton(  
  
        onPressed: () {  
  
          // Call the increment method of CounterModel using Provider  
  
          Provider.of<CounterModel>(context, listen: false).increment();  
  
        },  
  
      ),  
  
    ),  
  
  ),  
  
); }
```

```
tooltip: 'Increment',  
child: const Icon(Icons.add),  
),  
); } }
```

7. Save the file.

8. Run your Flutter project using the following command:

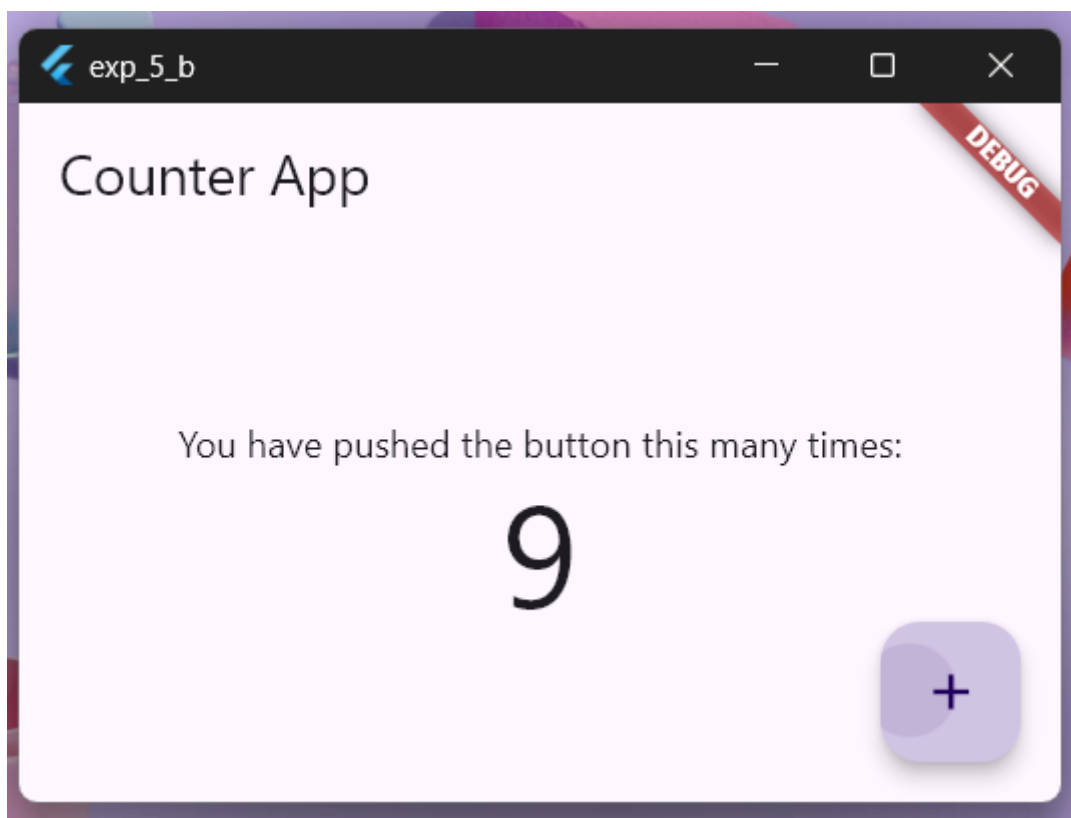
```
flutter run
```

Select the appropriate device to run the app.

9. During the app execution, you can use the following commands:

- Enter r to hot reload the app and see the changes you made to the code.
- Enter q to quit the app.

Output



Conclusion

In this experiment, we learned how to implement state management in a Flutter app using the `setState` method and the `Provider` package. We created a simple counter app that allows the user to increment the counter value by tapping a button. By using the `ChangeNotifierProvider` and `Consumer` widgets from the `Provider` package, we were able to update the UI whenever the counter value changed. This approach provides a scalable and efficient way to manage the state of a Flutter app.

Experiment 6. a) Create custom widgets for specific UI elements.

Aim

Create custom widgets for specific UI elements.

Objective

To understand how to create custom widgets for specific UI elements.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **IDE:** Visual Studio Code (Supported) or android studio (Supported) or IntelliJ IDEA (Supported).
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

The command creates a Flutter project directory called my_flutter_app that contains a simple demo app that uses Material Components.

2. Change to the Flutter project directory.

```
cd my_flutter_app
```

3. Open the lib/main.dart file in your Flutter project.

4. Replace the existing code with the following code snippet:

```
import 'package:flutter/foundation.dart'; // Required for kDebugMode
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('Custom Button Widget')),  
  body: Center(  
    child: CustomButton(  
      text: 'Click Me',  
      icon: Icons.circle_outlined,  
      color: Colors.lightBlue[600],  
      onPressed: () {  
        if (kDebugMode) {  
          // Check if the app is running in debug mode  
          print('Button Pressed!');  
        }  
      },  
    ), // CustomButton  
  ), // Center  
), // Scaffold  
); // MaterialApp  
}  
}  
  
// Custom Button Widget  
  
class CustomButton extends StatelessWidget {  
  final String text;  
  final IconData icon;  
  final VoidCallback onPressed;  
  final Color? color, textColor;  
  const CustomButton(  
    {super.key,  
    required this.text,  
    required this.icon,
```

```
        required this.onPressed,  
        this.color = Colors.blueAccent,  
        this.textColor = Colors.white));  
  
@override  
Widget build(BuildContext context) {  
    return ElevatedButton.icon(  
        onPressed: onPressed,  
        icon: Icon(icon, color: textColor),  
        label: Text(text, style: TextStyle(color: textColor)),  
        style: ElevatedButton.styleFrom(  
            backgroundColor: color,  
            shape: RoundedRectangleBorder(  
                borderRadius: BorderRadius.circular(8.0),  
            ), // RoundedRectangleBorder  
            padding: const EdgeInsets.symmetric(horizontal: 16.0, vertical: 12.0),  
        ),  
    ); // ElevatedButton.icon  
}  
}
```

5. Save the file.

6. Run your Flutter project using the following command:

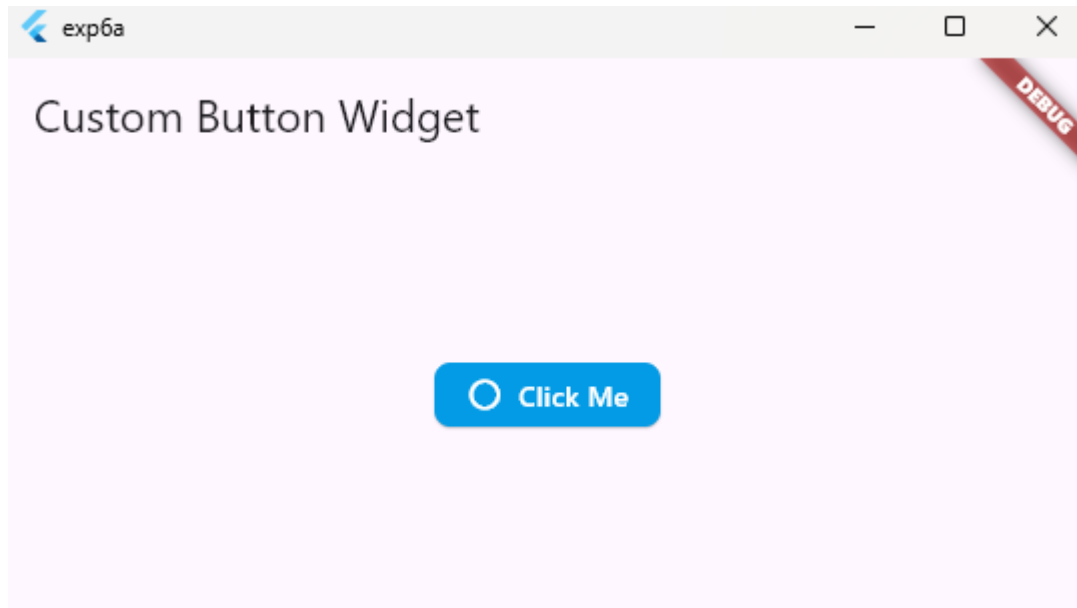
```
flutter run
```

Select the appropriate device to run the app.

7. During the app execution, you can use the following commands:

- Enter r to hot reload the app and see the changes you made to the code.
- Enter q to quit the app.

Output:



```
C:\Users\LAB10-STUDENT03\exp6a>flutter run
Connected devices:
Windows (desktop) • windows • windows-x64 • Microsoft Windows [Version 10.0.22631.5624]
Chrome (web) • chrome • web-javascript • Google Chrome 141.0.7390.108
Edge (web) • edge • web-javascript • Microsoft Edge 141.0.3537.92
[1]: Windows (windows)
[2]: Chrome (chrome)
[3]: Edge (edge)
Please choose one (or "q" to quit): 1
Launching lib\main.dart on Windows in debug mode...
Building Windows application... 10.0s
✓ Built build\windows\x64\runner\Debug\exp6a.exe
Syncing files to device Windows... 94ms

Flutter run key commands.
r Hot reload.
R Hot restart.
h List all available interactive commands.
d Detach (terminate "flutter run" but leave application running).
c Clear the screen
q Quit (terminate the application on the device).

A Dart VM Service on Windows is available at: http://127.0.0.1:63489/szKMJw1oSxg=/
The Flutter DevTools debugger and profiler on Windows is available at:
http://127.0.0.1:9100?uri=http://127.0.0.1:63489/szKMJw1oSxg=/
Button Pressed!
Button Pressed!
Button Pressed!
```

Conclusion

In this experiment, you learned how to create custom widgets for specific UI elements in Flutter.

Experiment 6 b) Apply styling using themes and custom styles.

Aim

To understand how to apply themes and custom styles to a Flutter app.

Objective

In this experiment, you will learn how to:

- Create a Flutter app with a custom theme.
- Apply custom styles to the app.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **IDE:** Visual Studio Code (Supported) or android studio (Supported) or IntelliJ IDEA (Supported).
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

The command creates a Flutter project directory called my_flutter_app that contains a simple demo app that uses Material Components.

Change to the Flutter project directory.

```
cd my_flutter_app
```

Open the lib/main.dart file in your Flutter project.

Replace the existing code with the following code snippet:

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

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

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

  @override
  State<MainApp> createState() => _MainAppState();
}

class _MainAppState extends State<MainApp> {
```

```
bool isDarkTheme = false;

@override

Widget build(BuildContext context) {

  return MaterialApp(

    theme: isDarkTheme

      ? ThemeData.dark()

      : ThemeData.light().copyWith(

        colorScheme: ColorScheme.fromSeed(seedColor: Colors.redAccent)),

    home: Builder(builder: (BuildContext context) {

      return Scaffold(

        appBar:

          AppBar(title: Text(isDarkTheme ? 'Dark Theme' : 'Light Theme')),

        body: Column(

          // mainAxisAlignment: MainAxisAlignment.center, <-- Uncomment to center the widgets

          children: <Widget>[

            OutlinedButton.icon(

              onPressed: () {},

              label: const Text('Button'),

              icon: isDarkTheme

                ? const Icon(Icons.dark_mode_rounded)

                : const Icon(Icons.light_mode_rounded)),

            const SizedBox(height: 20),

            SwitchListTile(

              title: const Text('Dark Theme'),

              value: isDarkTheme,

              onChanged: (value) {

                setState() {

                  isDarkTheme = value;

                }

              });

          ]

        );

      }

    );

  }

}
```

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

Save the file.

Run your Flutter project using the following command:

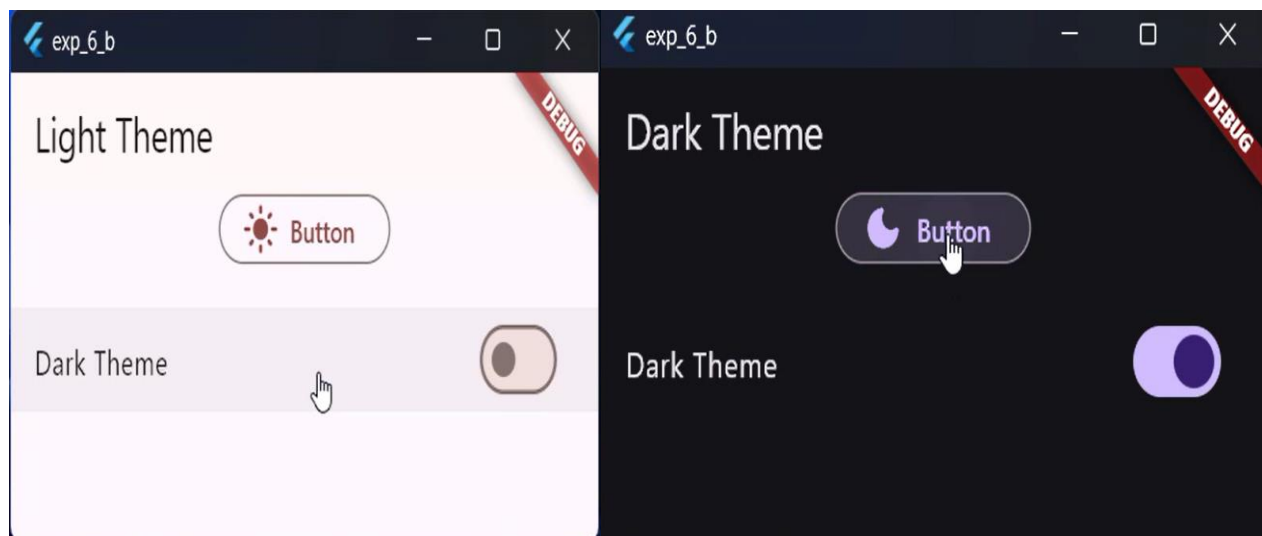
```
flutter run
```

Select the appropriate device to run the app.

During the app execution, you can use the following commands:

- Enter r to hot reload the app and see the changes you made to the code.
- Enter q to quit the app.

Output:



Conclusion

In this experiment, you learned how to create a Flutter app with a custom theme and apply custom styles to the app.

Experiment 7. a) Design a form with various input fields.

Aim

To create a Flutter application that includes a form with various input fields and a submit button.

Objective

To understand how to create and use custom text fields in a Flutter application and handle user input format.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **IDE:** Visual Studio Code (Supported) or android studio (Supported) or IntelliJ IDEA (Supported).
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

The command creates a Flutter project directory called my_flutter_app that contains a simple demo app that uses Material Components.

2. Change to the Flutter project directory.

```
cd my_flutter_app
```

3. Open the lib/main.dart file in your Flutter project.

4. Replace the existing code with the following code snippet:

```
import 'package:flutter/material.dart';  
  
import 'package:flutter/services.dart';  
  
void main() => runApp(const MainApp());  
  
class MainApp extends StatelessWidget {  
  
  const MainApp({super.key});  
  
  @override  
  
  Widget build(BuildContext context) {  
  
    return MaterialApp(  
  
      home: Scaffold(  
  
        appBar: AppBar(title: const Text('Form Application')),
```

```
body: Column(
  children: <Widget>[
    Image.asset('images/3d_avatar_21.png', width: 100, height: 100),
    // Custom text fields for user input
    const CustomTextField(label: 'First Name'),
    const CustomTextField(label: 'Last Name'),
    const CustomTextField(label: 'Email', suffixText: '@mlritm.ac.in'),
    const CustomTextField(
      prefixText: '+91 ',
      label: 'Phone Number',
      keyboardType: TextInputType.phone,
      maxLength: 10,
    ),
    const Divider(indent: 8, endIndent: 8), // Divider
    const CustomTextField(label: 'Username'),
    const CustomTextField(label: 'Password', obscureText: true),
    const CustomTextField(label: 'Confirm Password', obscureText: true),
    ElevatedButton(
      onPressed: () {},
      child: const Text('Register'),
    ), ], ), ), );}}
// Custom text field widget
class CustomTextField extends StatelessWidget {
  final String label;
  final TextInputType? keyboardType;
  final bool obscureText;
  final String? prefixText, suffixText;
  final int? maxLength;
  const CustomTextField({
```

```
super.key,  
required this.label,  
this.keyboardType,  
this.suffixText,  
this.prefixText,  
this.maxLength,  
this.obscureText = false,  
});  
  
@override  
Widget build(BuildContext context) {  
  return Padding(  
    padding: const EdgeInsets.symmetric(vertical: 8, horizontal: 16),  
    child: TextFormField(  
      keyboardType: keyboardType,  
      obscureText: obscureText,  
      inputFormatters: maxLength != null  
        ? [LengthLimitingTextInputFormatter(maxLength)]  
        : null,  
      decoration: InputDecoration(  
        border: const OutlineInputBorder(),  
        labelText: label,  
        suffixText: suffixText,  
        prefixText: prefixText,  
      ), ), ); } }
```

Save the file.

5. Get the image 3d_avatar from 3d_avatar_21.png and save it as the images/3d_avatar_21.png in your Flutter project.
6. Open the pubspec.yaml file in your Flutter project and add the following lines after flutter: to include the image asset in your project:

flutter:

```
# ...
```

assets:

- images/3d_avatar_21.png <-- Add this line

Save the file.

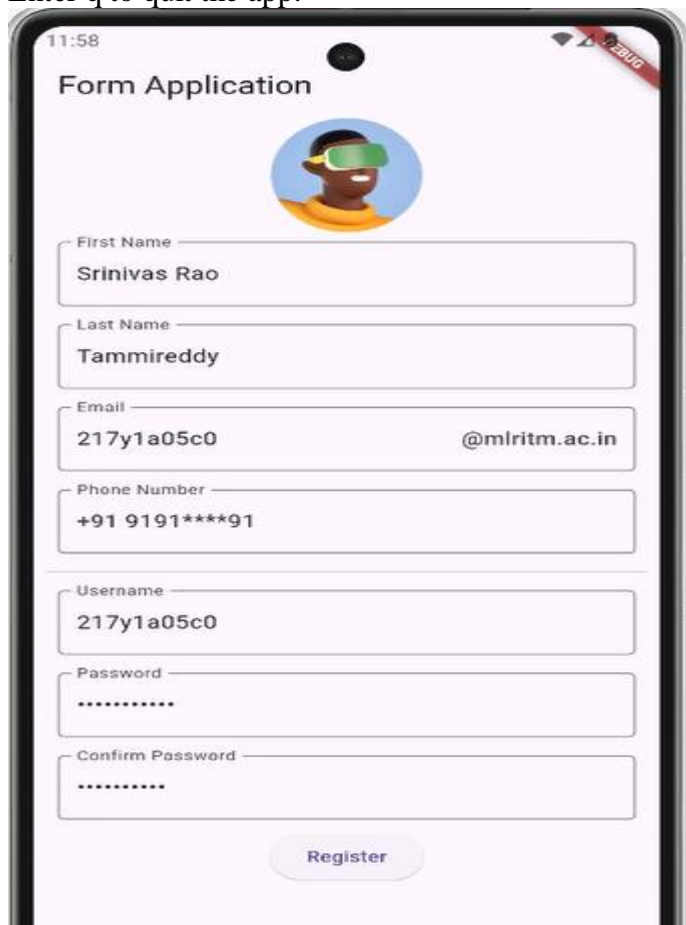
7. Run your Flutter project using the following command:

```
flutter run
```

Select the appropriate device to run the app.

8. During the app execution, you can use the following commands:

- Enter r to hot reload the app and see the changes you made to the code.
- Enter q to quit the app.



○

Conclusion

By following the above steps, you have successfully created a Flutter application with a form that includes custom text fields and a submit button. This exercise helps in understanding the creation and usage of custom widgets in Flutter.

Experiment 7 b) Implement form validation and error handling.

Aim

To design and implement a Flutter app that performs form validation and error handling.

Objective

To understand the concept of form validation and error handling in Flutter.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **Dart SDK:** version 2.12.0 or higher
- **IDE:** Visual Studio Code (latest version) or android studio (latest version)
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create exp_7_b
```

The command creates a Flutter project directory called exp_7_b that contains a simple demo app that uses Material Components.

2. Change to the Flutter project directory.

```
cd exp_7_b
```

3. Open the lib/main.dart file in your Flutter project.

4. Replace the existing code with the following code snippet:

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

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

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text('Form Validation')),
        body: const SingleChildScrollView(
```

```
        child: RegisterForm(),

        ), ), );} }

class RegisterForm extends StatefulWidget {

  const RegisterForm({super.key});

  @override

  State<RegisterForm> createState() => _RegisterFormState();

}

class _RegisterFormState extends State<RegisterForm> {

  final _formKey = GlobalKey<FormState>();

  final _nameController = TextEditingController(),

    _emailController = TextEditingController();

  @override

  Widget build(BuildContext context) {

    return Form(

      key: _formKey,

      child: Column(

        children: <Widget>[

          Padding(

            padding: const EdgeInsets.symmetric(vertical: 8, horizontal: 16),

            child: TextFormField(

              controller: _nameController,

              decoration: const InputDecoration(labelText: 'Name'),

              validator: (value) {

                if (value == null || value.isEmpty) {

                  return 'Please enter your name';

                }

                if (!RegExp(r'^[a-zA-Z][a-zA-Z\s]*$').hasMatch(value)) {

                  return 'Name can only contain letters and spaces in between';

                }

              }

            )

          ]

        )

      )

    );
```

```
        return null;

    },

),

),

Padding(

padding: const EdgeInsets.symmetric(vertical: 8, horizontal: 16),

child: TextFormField(

controller: _emailController,

decoration: const InputDecoration(labelText: 'Email'),

validator: (value) {

    if (value == null || value.isEmpty) {

        return 'Please enter your email';

    }

    if (!RegExp(

        r'^[a-zA-Z0-9](?!.*\+|\+)[a-zA-Z0-9._%+-]*[a-zA-Z0-9]@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-9-]+)*$')

        .hasMatch(value)) {

        return 'Please enter a valid email address';

    }

    return null;

},

),

),

const SizedBox(height: 20),

ElevatedButton(

onPressed: () {

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

        ScaffoldMessenger.of(context).showSnackBar(

            const SnackBar(content: Text('Data is valid')),
```

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

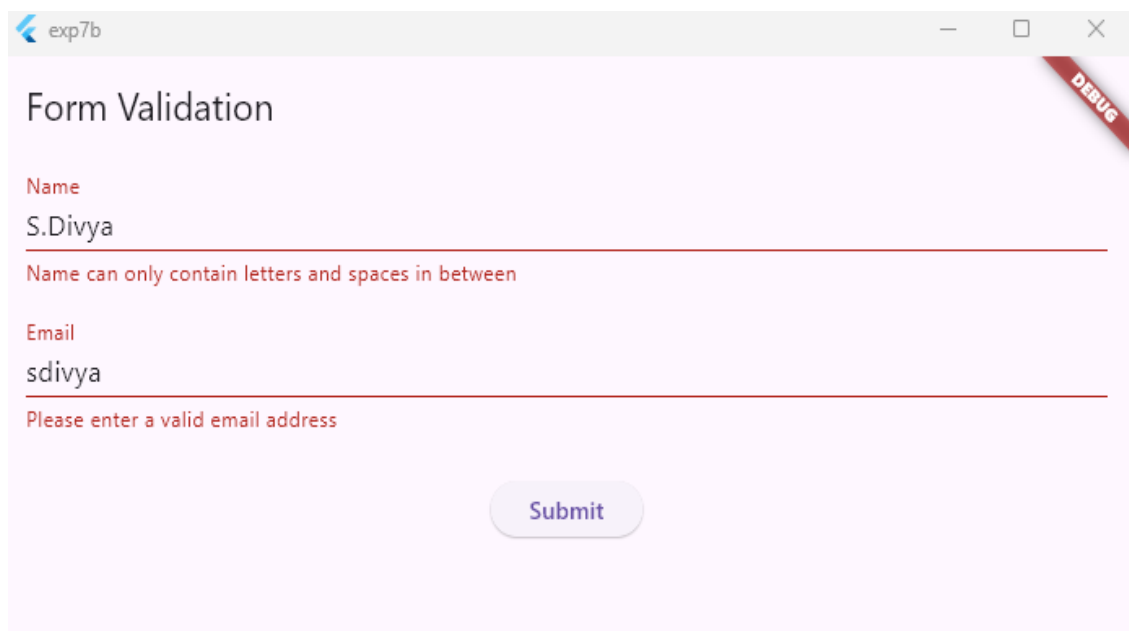
5. Save the file.
6. Run your Flutter project using the following command:

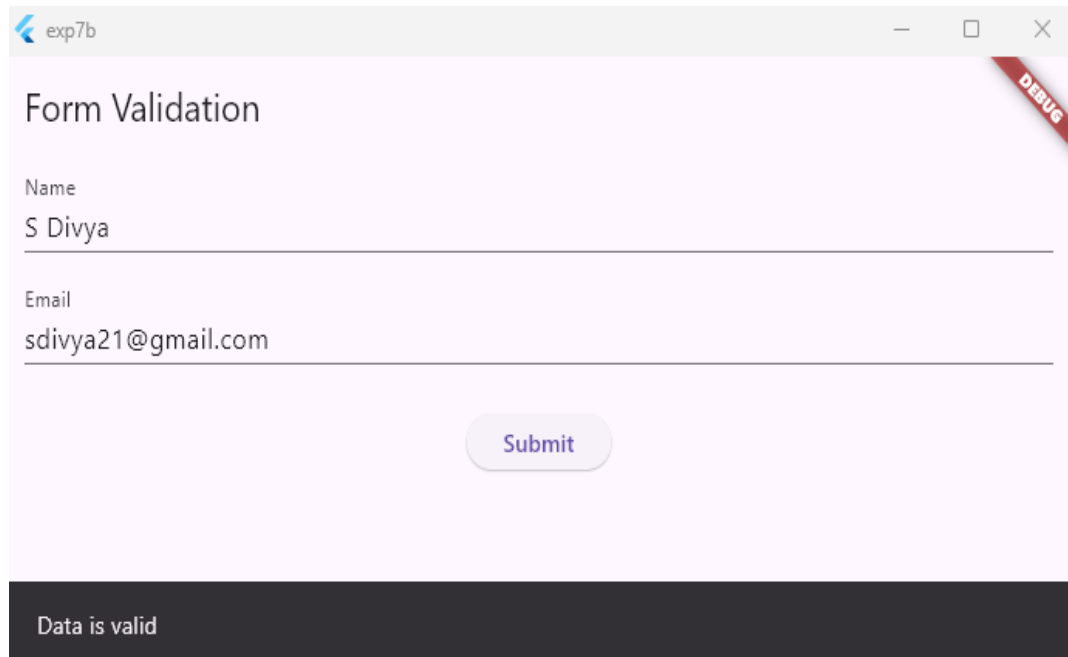
```
flutter run
```

Select the appropriate device to run the app.

7. During the app execution, you can use the following commands:
 - Enter r to hot reload the app and see the changes you made to the code.
 - Enter q to quit the app.

Output:





Conclusion

In this experiment, we learned how to perform form validation and error handling in a Flutter app. We created a simple form that takes the user's name and email address as input and validates the input data. If the input data is valid, a success message is displayed; otherwise, an error message is displayed.

Experiment 8. a) Add animations to UI elements using Flutter's animation framework.

Aim

To add animations to UI elements using Flutter's animation framework.

Objective

To understand how to add animations to UI elements using Flutter's animation framework.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **Dart SDK:** version 2.12.0 or higher
- **IDE:** Visual Studio Code (latest version) or android studio (latest version)
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

The command creates a Flutter project directory called my_flutter_app that contains a simple demo app that uses Material Components.

2. Change to the Flutter project directory.

```
cd my_flutter_app
```

3. Open the lib/main.dart file in your Flutter project.

4. Replace the existing code with the following code snippet:

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

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

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

  @override
  State<MainApp> createState() => _MainAppState();
}

class _MainAppState extends State<MainApp> {
  bool _isBig = false;
```

```
@override

Widget build(BuildContext context) {

  return MaterialApp(

    debugShowCheckedModeBanner: false,

    home: Scaffold(

      appBar: AppBar(

        title: const Text('Beginner Animation'),

      ),

      body: Center(

        child: Column(

          mainAxisAlignment: MainAxisAlignment.center,

          children: [

            const Text(

              'Tap the box to animate',

            ),

            GestureDetector(

              onTap: () {

                setState(() {

                  _isBig = !_isBig;

                });

              },

              child: AnimatedContainer(

                decoration: BoxDecoration(

                  borderRadius: BorderRadius.circular(8),

                  color: _isBig ? Colors.deepPurpleAccent : Colors.blueAccent,

                ),

                duration: const Duration(seconds: 1),

                curve: Curves.easeInOutBack,

                width: _isBig ? 250 : 100,
```

```
        height: _isBig ? 500 : 100,  
        alignment: Alignment.center,  
        child: const Text(  
          'Box',  
          style: TextStyle(  
            color: Colors.white, fontWeight: FontWeight.bold),  
          ),  
        ),  
      ),  
    ],  
  ),  
),  
),  
),  
);  
}  
}
```

5. Save the file.

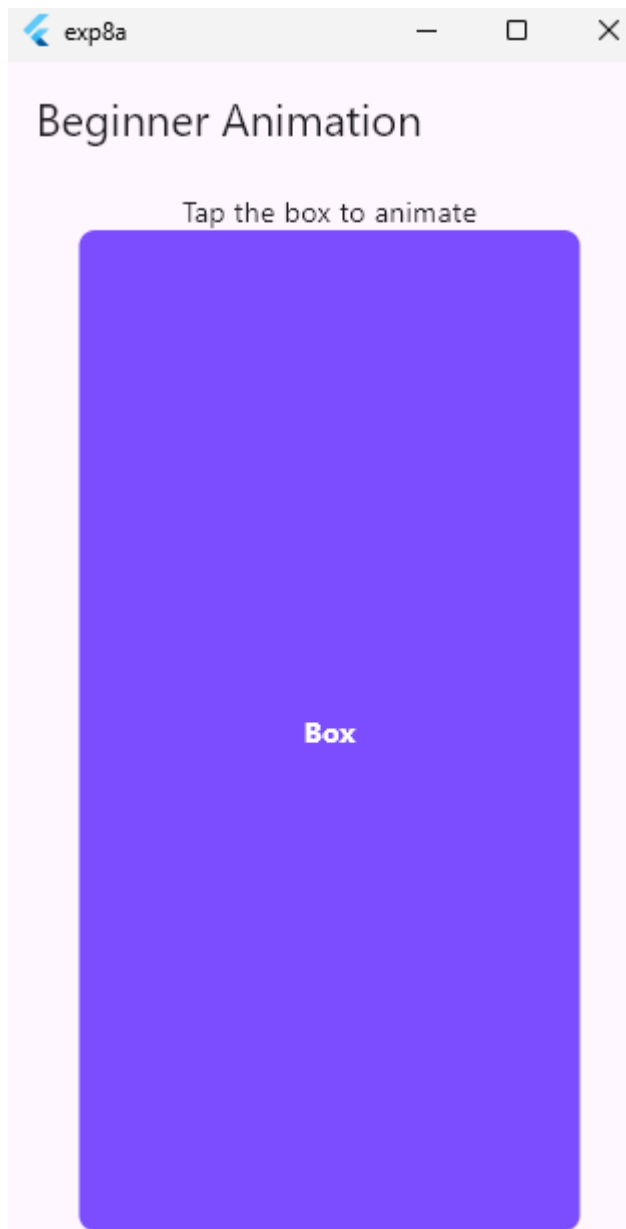
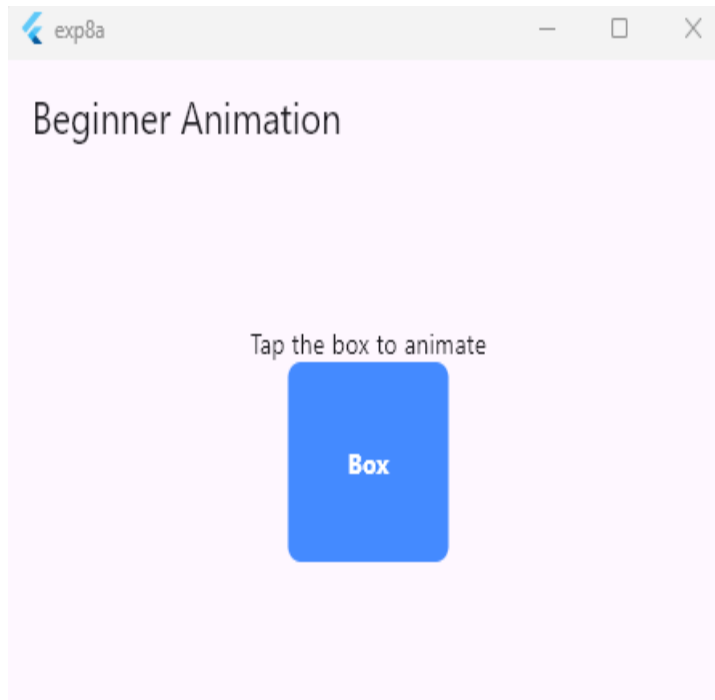
6. Run your Flutter project using the following command:

```
flutter run
```

Select the appropriate device to run the app.

7. During the app execution, you can use the following commands:

- Enter r to hot reload the app and see the changes you made to the code.
- Enter q to quit the app.



Experiment 8 b) Experiment with different types of animations (fade, slide, etc.).

Aim

To experiment with different types of animations in Flutter.

Objective

To understand the different types of animations in Flutter and how to implement them in a Flutter app.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **Dart SDK:** version 2.12.0 or higher
- **IDE:** Visual Studio Code (latest version) or android studio (latest version)
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create my_flutter_app
```

The command creates a Flutter project directory called my_flutter_app that contains a simple demo app that uses Material Components.

2. Change to the Flutter project directory.

```
cd my_flutter_app
```

3. Open the lib/main.dart file in your Flutter project.

4. Replace the existing code with the following code snippet:

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

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

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
```

```
        home: AnimationDemo(),  
    );  
}  
}  
  
class AnimationDemo extends StatefulWidget {  
    const AnimationDemo({super.key});  
  
    @override  
    State<AnimationDemo> createState() => _AnimationDemoState();  
}  
  
class _AnimationDemoState extends State<AnimationDemo>  
    with TickerProviderStateMixin {  
    late AnimationController _scaleController;  
    late Animation<double> _scaleAnimation;  
    late AnimationController _fadeController;  
    late Animation<double> _fadeAnimation;  
    late AnimationController _slideController;  
    late Animation<Offset> _slideAnimation;  
    late AnimationController _rotateController;  
    late Animation<double> _rotateAnimation;  
  
    bool _visible = false;  
    bool _sliding = false;  
    bool _rotating = false;  
    bool _scaling = true;  
  
    @override  
    void initState() {  
        super.initState();  
  
        _fadeController = AnimationController(  
            vsync: this,  
            duration: const Duration(seconds: 1),  
        );  
    }  
}
```

```
);

_fadeAnimation = Tween<double>(begin: 1, end: 0).animate(_fadeController);

_slideController = AnimationController(

  vsync: this,

  duration: const Duration(seconds: 1),

);

_slideAnimation = Tween<Offset>(

  begin: const Offset(-1.0, 0.0),

  end: const Offset(1.0, 0.0),

).animate(_slideController);

_rotateController = AnimationController(

  vsync: this,

  duration: const Duration(seconds: 2),

);

_rotateAnimation =

  Tween<double>(begin: 0, end: 1).animate(_rotateController);

_scaleController = AnimationController(

  vsync: this,

  duration: const Duration(seconds: 2),

)..repeat(reverse: true);

_scaleAnimation = CurvedAnimation(

  parent: _scaleController,

  curve: Curves.easeInOut,

);

}

@override

void dispose() {

  _scaleController.dispose();

  _fadeController.dispose();
```

```
_slideController.dispose();

_rotateController.dispose();

super.dispose();
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Animation Demo'),
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          FadeTransition(
            opacity: _fadeAnimation,
            child: SlideTransition(
              position: _slideAnimation,
              child: RotationTransition(
                turns: _rotateAnimation,
                child: ScaleTransition(
                  scale: _scaleAnimation,
                  child: const FlutterLogo(size: 100),
                ),
              ),
            ),
          ),
          ),
        ),
        ),
      child: SwitchListTile(
        title: const Text('Fade'),
```

```
        value: _visible,

        onChanged: (bool value) {

          setState(() {

            _visible = value;

            _visible

              ? _fadeController.forward()

              : _fadeController.reverse();

          });

        },

      ),

      SwitchListTile(

        title: const Text('Slide'),

        value: _sliding,

        onChanged: (bool value) {

          setState(() {

            _sliding = value;

            _sliding

              ? _slideController.repeat(reverse: true)

              : _slideController.stop();

          });

        },

      ),

      SwitchListTile(

        title: const Text('Rotate'),

        value: _rotating,

        onChanged: (bool value) {

          setState(() {

            _rotating = value;

            _rotating
```

```
        ? _rotateController.repeat()
        : _rotateController.stop();

    });

  },

),

SwitchListTile(

  title: const Text('Scale'),

  value: _scaling,

  onChanged: (bool value) {

    setState() {

      _scaling = value;

      _scaling

      ? _scaleController.repeat(reverse: true)

      : _scaleController.stop();

    });

  },

),

],

),

),

);

}

}
```

5. Save the file.

6. Run your Flutter project using the following command:

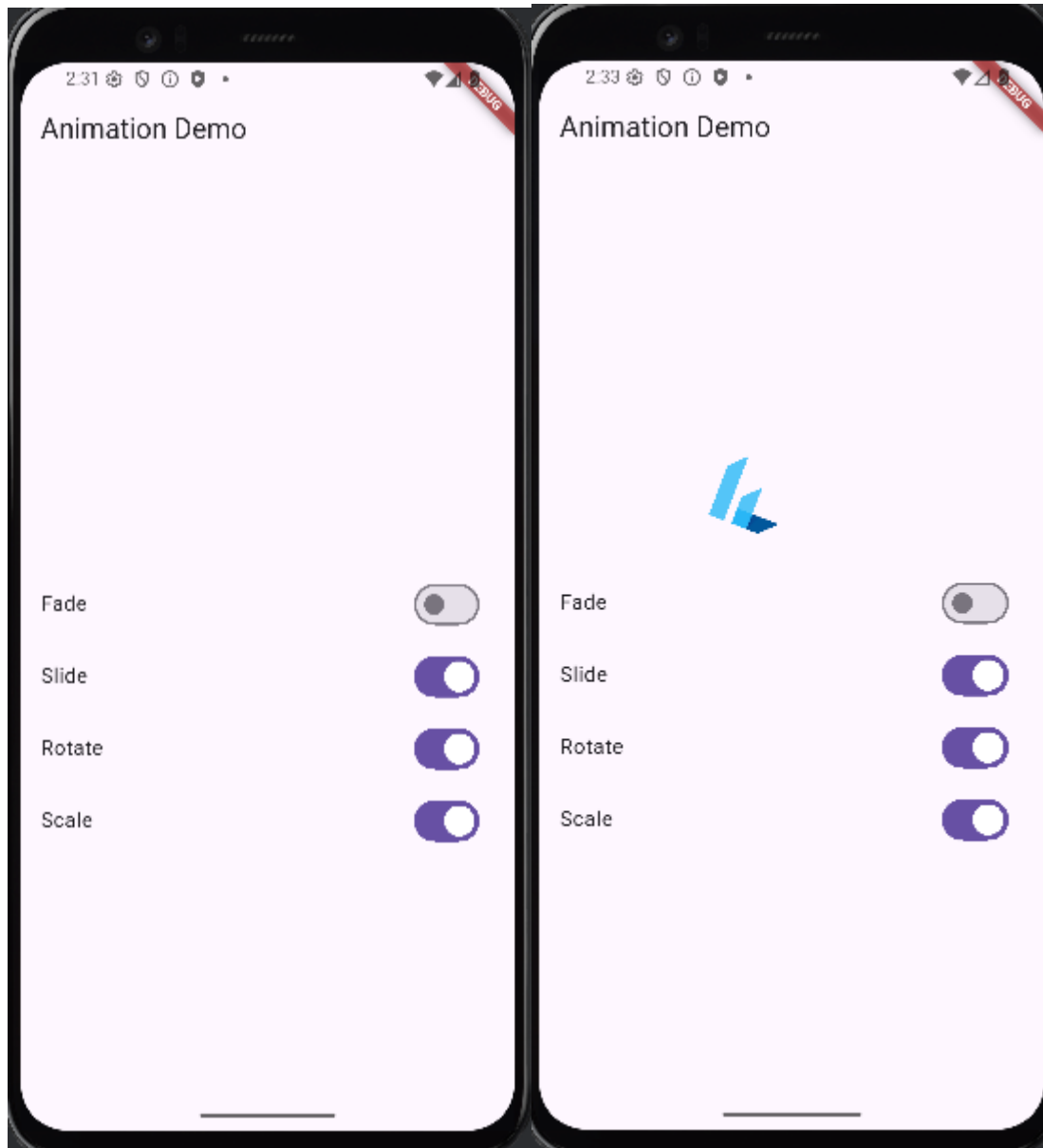
```
flutter run
```

Select the appropriate device to run the app.

7. During the app execution, you can use the following commands:

- Enter r to hot reload the app and see the changes you made to the code.

- Enter q to quit the app.



Conclusion

In this experiment, we learned how to implement different types of animations in a Flutter app. We used the FadeTransition, Slide Transition, RotationTransition, and ScaleTransition widgets to create fade, slide, rotate, and scale animations, respectively. We also used the Animation Controller class to control the animations and the Tween class to define the animation values. By combining these widgets and classes, we were able to create a simple animation demo app that demonstrates the use of different types of animations in Flutter.

Experiment 9. a) Fetch data from a REST API.

Aim

To fetch data from a REST API in Dart.

Objective

To understand how to fetch data from a REST API in Dart using the http package.

System Requirements

- **Dart SDK:** version 2.12.0 or higher
- **IDE:** Visual Studio Code (latest version) or any text editor
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Dart console project by running the following command in your terminal:

```
dart create console_app
```

The command creates a Dart project directory called console_app that contains a simple demo app in the \bin directory.

2. Change to the Dart project directory.

```
cd console_app
```

3. Add the http package to your project by adding the following dependency to the pubspec.yaml file:
dependencies:

```
http: ^1.2.2
```

Save the file.

4. Run the following command to get the dependencies:

```
dart pub get
```

This command downloads the http package and adds it to your project.

5. Create a new Dart file called git_hub_api.dart in the lib directory of your project.

6. Add the following code to the git_hub_api.dart file:

```
// Utilizing GitHub API service
```

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

```
import 'dart:convert';
```

```
Future<List<dynamic>> fetchRepos(String username) async {
```

```
    final response = await http.get(Uri.parse('https://api.github.com/users/$username/repos'));
```

```
if (response.statusCode == 200) {  
    return json.decode(response.body);  
} else {  
    throw Exception('Failed to load repos');  
} }
```

7. Open the bin/console_app.dart file in your Dart project.

8. Replace the existing code with the following code snippet:

```
// Utilizing GitHub API service from console_app  
import 'package:console_app/git_hub_api.dart' as git_hub_api;  
import 'dart:io';  
  
void main(List<String> arguments) async {  
    String? username = arguments.isNotEmpty ? arguments.first : null;  
  
    if (username == null) {  
        stdout.write('Enter GitHub username: ');  
        username = stdin.readLineSync();  
    }  
  
    if (username != null && username.isNotEmpty) {  
        try {  
            print('Fetching repositories for user: $username');  
  
            List<dynamic> repos = await git_hub_api.fetchRepos(username);  
  
            // Sort by 'updated_at' in descending order  
            repos.sort(  
                (a, b) => DateTime.parse(b['updated_at']).compareTo(  
                    DateTime.parse(a['updated_at']),  
                ),  
            );  
  
            print('First 5 recent repositories:');  
  
            for (var repo in repos.take(5)) {  
                print(repo['name']);  
            }  
        }  
    }  
}
```

```
    }  
  
    } catch (e) {  
  
        print('Error: $e');  
  
    }  
  
    } else {  
  
        print('Username cannot be empty');  
  
    } }  

```

9. Save the file.

10. Run your Dart project using the following command:

```
dart run
```

Output

```
C:\path\to\console_app> dart run  
Building package executable...  
Built exp_9_a:exp_9_a.  
Enter GitHub username: divya  
Fetching repositories for user: divya  
First 5 recent repositories:  
Flutter-Lab  
217Y1A05C0-ML-Lab  
LaTex-Resume  
FSD-Lab  
learning-dashboard  
C:\path\to\console_app>
```

command line arguments

```
C:\path\to\console_app> dart run .\bin\console_app.dart srinu2003  
Fetching repositories for user: divya  
First 5 recent repositories:  
Flutter-Lab  
217Y1A05C0-ML-Lab  
LaTex-Resume  
FSD-Lab  
learning-dashboard  
C:\path\to\console_app>
```

Conclusion

We have successfully fetched data from a REST API in Dart using the http package.

Experiment 9b) Display the fetched data in a meaningful way in the UI.

Aim

To fetch data from an API and display it in a meaningful way using Flutter widgets.

Objective

In this lab experiment, we will learn how to fetch data from an API using the http package and display the data in a user-friendly format using Flutter widgets such as ListView and FutureBuilder.

System Requirements

- **Flutter SDK:** version 2.0.0 or higher
- **Dart SDK:** version 2.12.0 or higher
- **IDE:** Visual Studio Code (latest version) or Android Studio (latest version)
- **Operating System:** Windows (7 or higher), macOS (10.12 or higher), or Linux (Ubuntu, Debian, Fedora, CentOS, or similar)

Procedure

1. Create a new Flutter project by running the following command in your terminal:

```
flutter create fetch_data_app
```

The command creates a Flutter project directory called fetch_data_app.

2. Change to the Flutter project directory.

```
cd fetch_data_app
```

3. Add the http package to your project by updating the pubspec.yaml file:

4. dependencies:
http: ^0.15.0

Run flutter pub get to install the package.

5. Create a new file lib/src/services/api_service.dart and implement the API service to fetch data:
// filepath: lib/src/services/api_service.dart

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

```
import 'dart:convert';
```

```
class ApiService {
```

```
Future<List<dynamic>> fetchData() async {
```

```
    final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));
```

```
    if (response.statusCode == 200) {
```

```
        return json.decode(response.body);
```

```
    } else {
```

```
        throw Exception('Failed to load data');  
    }  
}  
}
```

6. Create a new file lib/src/screens/home_screen.dart and implement the UI to display the fetched data:

// filepath: lib/src/screens/home_screen.dart

```
import 'package:flutter/material.dart';  
import '../services/api_service.dart';  
  
class HomeScreen extends StatefulWidget {  
    const HomeScreen({super.key});  
  
    @override  
    State<HomeScreen> createState() => _HomeScreenState();  
}  
  
class _HomeScreenState extends State<HomeScreen> {  
    late Future<List<dynamic>>> _data;  
  
    @override  
    void initState() {  
        super.initState();  
        _data = ApiService().fetchData();  
    }  
  
    @override  
    Widget build(BuildContext context) {  
        return Scaffold(  
            appBar: AppBar(  
                title: const Text('Fetched Data'),  
            ),  
            body: FutureBuilder<List<dynamic>>>(  
                future: _data,  
                builder: (context, snapshot) {
```

```
if (snapshot.connectionState == ConnectionState.waiting) {  
    return const Center(child: CircularProgressIndicator());  
} else if (snapshot.hasError) {  
    return Center(child: Text('Error: ${snapshot.error}'));  
} else if (snapshot.hasData) {  
    return ListView.separated(  
        itemCount: snapshot.data!.length,  
        separatorBuilder: (context, index) => const Divider(),  
        itemBuilder: (context, index) {  
            final item = snapshot.data![index];  
            return Padding(  
                padding: const EdgeInsets.all(16.0),  
                child: Column(  
                    crossAxisAlignment: CrossAxisAlignment.start,  
                    children: [  
                        Text(  
                            item['title'],  
                            style: Theme.of(context).textTheme.titleLarge?.copyWith(  
                                fontWeight: FontWeight.bold,  
                                fontSize: 20,  
                            )),  
                        const SizedBox(height: 8),  
                        Text(  
                            item['body'],  
                            style: Theme.of(context).textTheme.bodyMedium?.copyWith(  
                                fontSize: 16,  
                            )),  
                    ],  
                ),  
            ),  
        ),  
    );  
}
```

```
        ],  
        ),  
    );  
    },  
    );  
    } else {  
        return const Center(child: Text('No data available'));  
    }  
    },  
    ),  
    );  
    }  
}
```

7. Update the lib/main.dart file to set the HomeScreen as the home widget:

// filepath: lib/main.dart

```
import 'package:flutter/material.dart';  
import 'package:google_fonts/google_fonts.dart';  
import 'src/screens/home_screen.dart';  
  
void main() {  
    runApp(const MainApp());  
}  
  
class MainApp extends StatelessWidget {  
    const MainApp({super.key});  
  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            theme: ThemeData(  
                useMaterial3: true,  
                colorScheme: ColorScheme.fromSeed(seedColor: Colors.teal),
```

```
textTheme: GoogleFonts.lobsterTwoTextTheme(  
  ThemeData.light().textTheme,  
),  
),  
home: const HomeScreen(),  
);  
}  
}
```

8. Save all files.

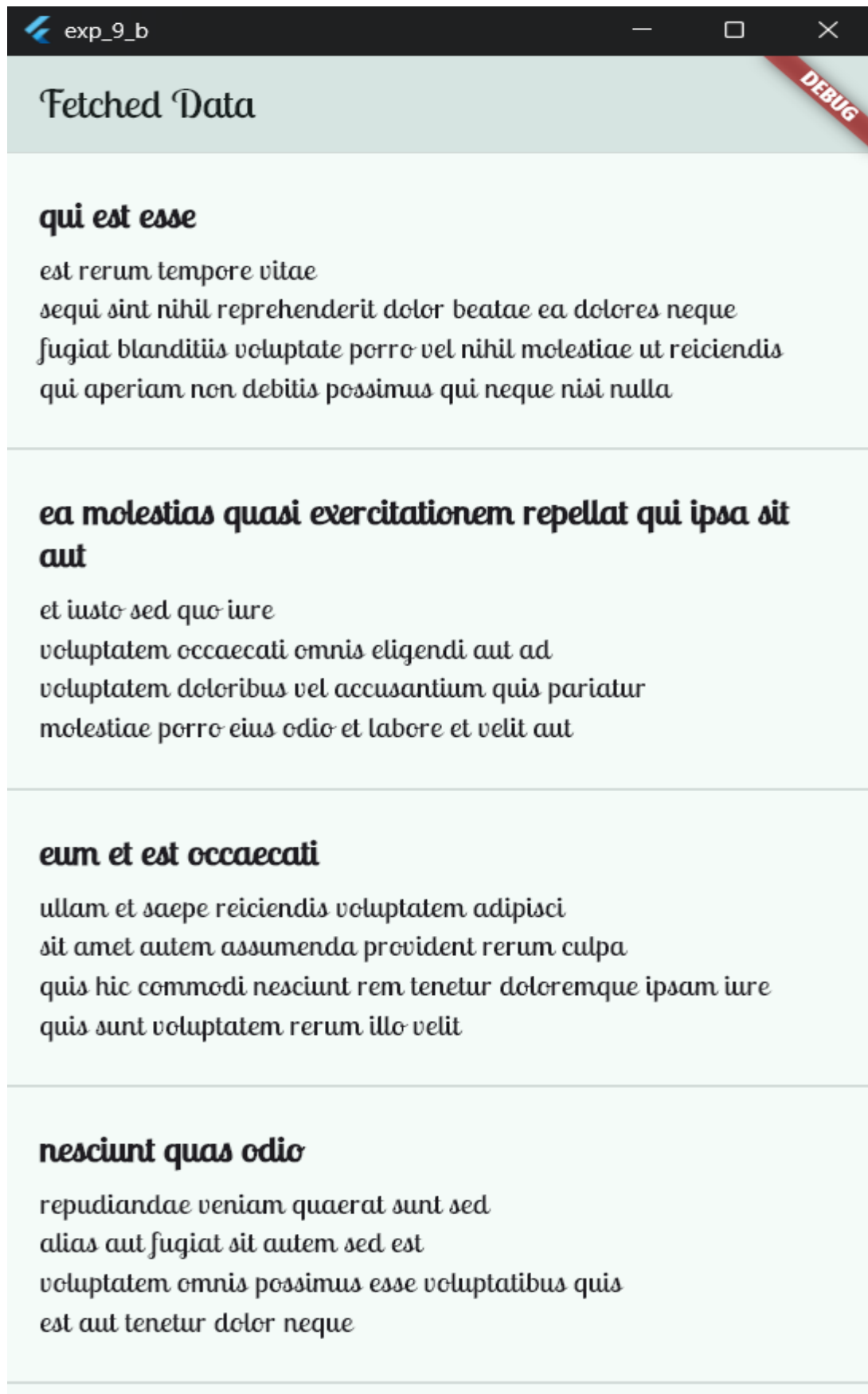
9. Run your Flutter project using the following command:

```
flutter run
```

Select the appropriate device to run the app.

10. During the app execution, you can use the following commands:

- Enter r to hot reload the app and see the changes you made to the code.
- Enter q to quit the app.



Conclusion

In this lab experiment, we successfully fetched data from an API and displayed it in a meaningful way using Flutter widgets. This approach allows us to build dynamic and interactive apps that consume external data.

Experiment 10. a) Write unit tests for UI components.

Unit Testing UI Components in Flutter:

Why We Test UI Components?

In software development, a "unit" refers to the smallest testable part of an application. In Flutter, this unit is often a single widget, or a small group of widgets. Unit testing for UI components is the process of writing code that automatically verifies a widget renders and behaves exactly as you intend, in an isolated and controlled environment.

The main goals of this practice are:

Confidence: To be certain that our UI is correct and won't break when we make other changes.

Reliability: To catch bugs early in the development cycle, before they reach users.

Clarity: To provide living documentation of how our widgets are supposed to work.

The Test Structure: Arrange, Act, Assert (AAA)

Most well-structured tests follow the "Arrange, Act, Assert" pattern.

Arrange: Set up the test environment. You build the widget you want to test and provide it with all the necessary data.

Act: Perform a specific action on the widget, such as tapping a button, typing text, or scrolling.

Assert: Verify that the action had the expected result. You check for changes in the UI or in the state of the app.

This pattern makes tests easy to read and understand.

Core Syntax for UI Component Testing

The `flutter_test` package provides all the tools you need to perform widget testing. Here are the most important components:

testWidgets(): The function used to define a widget test. It gives you a `WidgetTester` object.

WidgetTester (tester): Your main tool for interacting with the widget tree. It allows you to build, tap, and find widgets.

tester.pumpWidget(widget): This method "inflates" or renders a widget in the test environment.

tester.tap(finder): This simulates a user's tap gesture on a found widget.

tester.pump(): After an action that changes the widget's state (like setState()), you must call this method to rebuild the UI and reflect the changes.

find: A class with static methods to locate widgets in the widget tree.

find.text('...'): Finds a widget that contains a specific string.

find.byIcon(Icons.add): Finds a widget by its icon.

expect(finder, matcher): The core assertion function. It checks if the result of a find operation matches what you expect.

findsOneWidget: Asserts that exactly one widget was found.

findsNothing: Asserts that no widgets were found.

For the first test: PostCard displays title and body

Arrange: This is the setup phase.

The line `await tester.pumpWidget(const MaterialApp(home: PostCard(...)));` is the "Arrange" step. We are preparing the testing environment by building our PostCard widget on a virtual screen.

Act: The action being tested.

Since PostCard is a stateless widget with no buttons or user inputs, there is no explicit user interaction in this test. The "Act" is the rendering of the widget itself. The test's purpose is to verify the state of the UI immediately after it has been built.

Assert: The verification step.

The two expect statements, `expect(find.text("Test Title"), findsOneWidget);` and `expect(find.text("Test Body"), findsOneWidget);`, are the "Assert" steps. We are verifying that the widget has rendered the correct text as a result of the "Act" (rendering).

For the second test: PostCard widget has correct styling

This one is very similar, but the "Act" step is slightly more defined:

Arrange: Again, the `await tester.pumpWidget(...)` line is the "Arrange" step, setting up the widget in the test environment.

Act: The lines `final titleText = tester.widget<Text>(find.text("Test Title"));` and `final bodyText = tester.widget<Text>(find.text("Test Body"));` are the "Act" steps. We are performing an action (finding and getting a reference to the widgets) in preparation for the assertion.

Assert: The series of expect statements are the "Assert" steps, where we verify the style properties like `fontSize` and `fontWeight` match what we expect.

Post card example:

Post_card.dart:

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

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: const EdgeInsets.all(16.0),

      child: Padding(

        padding: const EdgeInsets.all(16.0),

        child: Column(

          crossAxisAlignment: CrossAxisAlignment.start,
```

```
children: [

  Text(

    title,

    style: const TextStyle(

      fontSize: 18.0,

      fontWeight: FontWeight.bold,

    ),

  ),

  const SizedBox(height: 8.0),

  Text(

    body,

    style: const TextStyle(

      fontSize: 16.0,

    ),

  ),

],

),

),

);

}

}

Post_card_test.dart:

import 'package:flutter/material.dart';

import 'package:flutter_test/flutter_test.dart';

import 'package:flutter_application_4/post_card.dart'; // Import the widget from your app's lib
folder

void main() {
```

```
// Test case to verify that the PostCard displays the correct title and body text.

testWidgets('PostCard displays title and body', (WidgetTester tester) async {

// Build our widget with a MaterialApp wrapper. This is necessary because some
// widgets (like Card) require a Material ancestor to render correctly.

await tester.pumpWidget(

const MaterialApp(

home: PostCard(

title: 'Test Title',

body: 'Test Body',

),

),

);

// Use `find.text()` to locate widgets with the exact text content.

// `findOneWidget` asserts that exactly one widget with that text was found.

expect(find.text('Test Title'), findsOneWidget);

expect(find.text('Test Body'), findsOneWidget);

});

// Test case to verify that the styling on the text widgets is correct.

testWidgets('PostCard widget has correct styling', (WidgetTester tester) async {

// Build the widget with the desired content.

await tester.pumpWidget(

const MaterialApp(

home: PostCard(

title: 'Test Title',

body: 'Test Body',

),

),

),
```

```
);

// Find the Text widgets using their keys or text content.

// Use `tester.widget()` to get a reference to the actual widget instance.

final titleText = tester.widget<Text>(find.text('Test Title'));

final bodyText = tester.widget<Text>(find.text('Test Body'));

// Assert that the style properties match our expectations.

// We can check for font size, font weight, color, etc.

expect(titleText.style?.fontSize, 18.0);

expect(titleText.style?.fontWeight, FontWeight.bold);

expect(bodyText.style?.fontSize, 16.0);

});

}
```

Basic Flutter app example:

main.dart:

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

void main() {

  runApp(const MyApp());

}

class MyApp extends StatelessWidget {

  const MyApp({super.key});

  @override

  Widget build(BuildContext context) {

    return const MaterialApp(

      title: 'Flutter Demo',

      home: MyHomePage(),

    );

  }

}
```

```
}

class MyHomePage extends StatefulWidget {

  const MyHomePage({super.key});

  @override

  State<MyHomePage> createState() => _MyHomePageState();

}

class _MyHomePageState extends State<MyHomePage> {

  int _counter = 0;

  void _incrementCounter() {

    setState(() {

      _counter++;

    });

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: const Text('Flutter Demo Home Page'),

      ),

      body: Center(

        child: Column(

          mainAxisAlignment: MainAxisAlignment.center,

          children: [

            const Text(

              'You have pushed the button this many times:',

            ),

            Text(
```

```
'$_counter',  
  
style: Theme.of(context).textTheme.headlineMedium, // Using the updated text style  
  
,  
  
],  
  
,  
  
,  
  
floatingActionButton: FloatingActionButton(  
  
onPressed: _incrementCounter,  
  
tooltip: 'Increment',  
  
child: const Icon(Icons.add),  
  
,  
  
);  
  
}  
  
}
```

Testing:

```
import 'package:flutter/material.dart';  
  
import 'package:flutter_test/flutter_test.dart';  
  
import 'package:testing/main.dart'; // Correct package name as in your example  
  
void main() {  
  
  // This test checks a complete user interaction flow.  
  
  testWidgets('Counter increments smoke test', (WidgetTester tester) async {  
  
    // 1. Arrange: Build our app in the test environment.  
  
    await tester.pumpWidget(const MyApp());  
  
    // 2. Assert (Initial State): Verify the initial state of the counter.  
  
    // The counter starts at 0, so we expect to find a widget with the text '0'.  
  
    expect(find.text('0'), findsOneWidget);  
  
    // We also make sure that '1' is not on the screen yet.
```

```
expect(find.text('1'), findsNothing);

// 3. Act: Tap the floating action button to increment the counter.

// `find.byIcon(Icons.add)` locates the button by its icon.

await tester.tap(find.byIcon(Icons.add));

// `tester.pump()` triggers a new frame to rebuild the widget after the state has changed.

await tester.pump();

// 4. Assert (Final State): Verify the counter has incremented.

// We expect the text '0' to be gone.

expect(find.text('0'), findsNothing);

// And we expect the text '1' to now be on the screen.

expect(find.text('1'), findsOneWidget);

});
}
```

Experiment 10.b) Use flutter's debugging tools to identify and fix issues.

```
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() {

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

Now, let's use Flutter's debugging tools to identify and fix the issue:

Widget Inspector: First, let's inspect the widget tree to see if the "+" button is correctly wired to the `_incrementCounter` method. We can do this by running the app in debug mode and enabling the widget inspector. You can do this by clicking the "Open DevTools" button in your IDE (Android Studio/IntelliJ IDEA or Visual Studio Code) or running the following command in your terminal:

flutter run --debug

Once the app is running, click on the "Toggle Widget Inspector" button in the top-right corner of your app. Then, select the FloatingActionButton widget representing the "+" button. Ensure that the onPressed callback is correctly set to _incrementCounter.

Debugging Console: If everything looks fine in the widget inspector, we can add some debug print statements to the _incrementCounter method to see if it's being called when the button is pressed.

Modify the _incrementCounter method as follows:

```
void _incrementCounter() { print('Incrementing counter');  
  _counter++; }
```

Now, run the app again in debug mode and observe the console output when you press the "+" button.

If you don't see the "Incrementing counter" message in the console, it means the _incrementCounter method is not being called.

Breakpoints: As a final step, let's set a breakpoint in the _incrementCounter method and debug the app to see if it's being hit. Add a breakpoint by clicking on the left margin of the _incrementCounter method in your code editor.

Then, run the app in debug mode and press the "+" button. The app should pause at the breakpoint, allowing you to inspect the current state and variables. You can step through the code to see if there are any issues with the execution flow.

By using Flutter's debugging tools, you should be able to identify the issue with the counter app and fix it accordingly. In this case, if the debugging process reveals that the _incrementCounter method is not being called, you can double-check the onPressed callback of the FloatingActionButton to ensure it's correctly wired to the _incrementCounter method.