



Flutter lab manual final

UI Flutter Design (Jawaharlal Nehru Technological University, Hyderabad)



Scan to open on Studocu

Lab Manual
For
UI DESIGN – FLUTTER LAB

III B. TECH I SEMESTER

(R22AUTONOMOUS)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(IoT)



ACE
Engineering College
An Autonomous
Institution

Ghatkesar, Hyderabad - 501 301,

Telangana. Approved by AICTE &

Affiliated to JNTUH



NBA Accredited B.Tech Courses, Accorded NACC A-Grade with 3.20 CGPA

INDEX

S. No	Contents	Page No.
1	Department Vision, Mission, PEOs,POs and PSOs	1
2	Objectives and Course Outcomes of the lab	3
3	COs Mapping with POs and PSOs	4
4	Scheme of Evaluation	4
5	Module Wise Outcome	5
6	Requirements	6
7	List of Experiments	7
8	Introduction to Lab	12
9	Algorithms/Programs with Viva questions	12
10	Additional experiments- beyond syllabus	151
11	References	164

Institute Vision:

To be a leading Technical Institute to prepare high quality Engineers to cater the needs of the stakeholders in the field of Engineering and Technology with global competence fundamental comprehensive analytical skills, critical reasoning, research aptitude, entrepreneur skills, ethical values and social concern.

Institute Mission:

Imparting Quality Technical Education to young Engineers by providing the state-of-the-art laboratories, quality instructions by qualified and experienced faculty and research facilities to meet the requirements of stakeholders in real time usage and in training them to excel in competitive examinations for higher education and employment to interface globally emerging techno- informative challenges in the growth corridor of techno-excellence.

Department Vision:

To be an epicenter of excellence in education by offering thrust courses, research and services for the students and make them to succeed in professional **competitive examinations** globally with an attitude of entrepreneurial skills, ethical values and social concern.

Department Mission:

Imparting quality Technical Education to young Computer Engineer by providing them

M1: Impart quality technical Education with State of-the-art laboratories, Analytical and Technical Skills with International standards by qualified and experienced faculty

M2: Prepare for competitive examinations for higher studies / Employment

M3: Develop professional attitude, Research aptitude, Critical Reasoning and technical consultancy by providing training in cutting edge technologies.

M4: Endorse and Nurture knowledge, Life-long learning, Entrepreneurial practices, ethical values and social concern

Program Educational Objectives (PEOs)

PEO 1: To prepare the students for successful careers in Computer Science and Engineering and fulfill the need by providing training to excel in competitive examinations for higher education and employment.

PEO 2: To provide students a broad-based curriculum with a firm foundation in Computer Science and Engineering, Applied Mathematics & Sciences. To impart high quality technical skills for designing, modeling, analyzing and critical problem solving with global competence.

PEO 3: To inculcate professional, social, ethical, effective communication skills and entrepreneurial practice among their holistic growth.

PEO 4: To provide Computer Science and Engineering students with an academic environment and members associated with student related to professional bodies for multi-disciplinary approach and for lifelong learning.

PEO 5: To develop research aptitude among the students in order to carry out research in cutting edge technologies, solve real world problems and provide technical consultancy services.

Program Educational Objectives (PEOs)

PEOs	Statement
PEO 1	To prepare the students for successful careers in Computer Science and Engineering and fulfill the need by providing training to excel in competitive examinations for higher education and employment.
PEO 2	To provide students a broad-based curriculum with a firm foundation in Computer Science and Engineering, Applied Mathematics & Sciences. To impart high quality technical skills for designing, modeling, analyzing and critical problem solving with global competence.
PEO 3	To inculcate professional, social, ethical, effective communication skills and entrepreneurial practice among their holistic growth.
PEO 4	To provide Computer Science and Engineering students with an academic environment and members associated with student related to professional bodies for multi-disciplinary approach and for lifelong learning.
PEO 5	To develop research aptitude among the students in order to carry out research in cutting edge technologies, solve real world problems and provide technical consultancy services.

Program Outcomes

Program Outcomes	Statement
PO1	An ability to apply knowledge of mathematics, science, and engineering and knowledge of Fundamental Principles.
PO2	An ability to Identify, formulate and solve engineering problems.
PO3	An ability to design a system, component, or process to meet desired needs in Computer Science and Engineering within realistic constraints such as economic, environmental, social, political, ethical, health and safety, manufacturability and sustainability, Design and Modeling.
PO4	An ability to design and conduct experiments, as well as to analyze and interpret data, Experimentation & Interpret/Engineering Analysis.
PO5	An ability to use the techniques, skills and modern Computer Science and Engineering tools necessary for system design with embedded engineering practice.
PO6	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	The broad education necessary to understand the impact of engineering solutions in a global, economic, environmental, and societal context.
PO8	An understanding of professional and ethical responsibility.
PO9	An ability to function on multidisciplinary teams.
PO10	An ability to communicate effectively.
PO11	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	Recognition of the need for, and an ability to engage in life-long learning.

Program Specific Outcomes

Program Specific Outcomes	Statement
PSO1	To prepare the students ready for industry usage by providing required training in cutting edge technologies.
PSO2	An Ability to use the core concepts of computing and optimization techniques to develop more efficient and effective computing mechanisms.
PSO3	Prepare the graduates to demonstrate a sense of societal and ethical responsibility In their professional endeavors and will remain informed and involved as full participants in the profession and our society.

Course Objectives:

The students will earn the following:

CourseObjectives
To learn installation of SDK of Flutter, X code and Android Emulator
Understanding Stateless and Stateful Widgets and Widget Tree
Learning of Dart basics
Application of Animation to app.

Course Outcomes

COURSE CO's	COURSE OUTCOMES	BTL
CO1	Knowledge on installation of various softwares.	L3: Apply
CO2	Understanding of various Widgets.	L4:Analyses
CO3	Application of Animation to Apps.	L4:Analyses

Course Outcomes – Program Outcomes Mapping

C O U R S E C o s	P O 1	P O 2	P O 3	P O 4	P O 5	P O 6	P O 7	P O 8	P O 9	P O 10	P O 11	P O 12
CO1	3	2					1		1	1	1	3
CO2	2	2	3		2			2				2
CO3	2	3	2		2	2				2		2
CO4	2	2	2	3	2				3		3	2
CO5	2	2	2	3	2		1			1		2

Course Outcomes – Program Specific Outcomes Mapping

COURSEC Cos	PSO1	PSO2	PSO3
CO1	3	2	1
CO2	2	2	2
CO3	2	2	3
CO4	1	2	2
CO5	2	1	2

Enter correlation levels 1, 2 or 3 as defined below: 1: Slight (Low) 2: Moderate (Medium) 3: Substantial (High)

Scheme of Evaluation (Autonomous)

Internal Assessment

S. No.	Assessment of work	Evaluation in Marks
1	Lab observation-day to day work	10
2	Viva-Voce/Tutorial/Case Study/Application/Poster Presentation	10
3	Exam	10
4	Laboratory Report/Project Presentation/ App Development	10
Total Marks		40

External Assessment

S. No.	Assessment of work	Evaluation in Marks
1	Write up	10
2	Experiment/Program	15
3	Evaluation of results	15
4	Presentation on another Experiment/Program in the same laboratory course	10
5	Viva	10
Total Marks		60

B.Tech. III Year I Semester Syllabus

UI DESIGN - FLUTTER

CourseCode	Category	Hours/Week			Credits	Maximum Marks		
CS605PC	PC	L	T	P	C	C I A	S E E	Tot al
		0	0	2	1	4 0	60	10 0

LIST OF EXPERIMENTS

Write a Program to Implement the following using Python.

- Install Flutter And Dart SDK
 - Write a simple Dart program to understand the language basics.
- Explore various flutter widgets (Text,Image,Container,etc..)
 - Implement different layout structures using Row,Column,and Stack widgets
- design a responsive UI that adapts to different screen sizes
 - Implement media queries and breakpoints for responsiveness
- Setup navigation between different screens using navigator
 - Implement navigation with named routes
- Learn about stateful and stateless widgets
 - Implement state management using set State and Provider
- Create custom widgets for specific UI elements
 - Apply styling using themes and custom styles
- Design a form with various input fields
 - Implement form validation and error handling
- Add animations to UI elements using flutter's animation framework
 - Experiment with different types of animations like fade,slide,etc.

9.
 - a) Fetch data from REST API
 - b) Display the fetched data in a meaningful way in the UI
10.
 - a) Write unit tests for UI components
 - b) Use Flutter's debugging tools to identify and fix issues

TEXT BOOK:

1. A Marco L.Napoli, Beginning Flutter: A Hands-on Guide to App Development

REFERENCE BOOKS:

1. Flutter for Beginners: An introductory guide to building cross-platform mobile applications with Flutter and Dart 2, Packt Publishing Limited.
2. Rap Payne, Beginning App Development with Flutter: Create Cross-Platform Mobile Apps, 1st edition, Apress.
3. Frank Zammetti, Practical Flutter: Improve your Mobile Development with Google's Latest Open-Source SDK, 1st edition, Apress

1. Introduction

- 1.1 Brief overview of the lab manual
- 1.2 Objectives of the laboratory sessions
- 1.3 Prerequisites for students
- 1.4 Necessary software and hardware requirements

Experiments :

Lab Session 1: a) Install Flutter And Dart SDK

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

Lab Session 2: a) Explore various flutter widgets (Text, Image, Container, etc..)

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

Lab Session 3: a) Design a responsive UI that adapts to different screen sizes

b) Implement media queries and breakpoints for responsiveness

Lab Session 4: a) Setup navigation between different screens using navigator

b) Implement navigation with named routes

Lab Session 5: a) Learn about stateful and stateless widgets

b) Implement state management using set State and Provider

Lab Session 6: a) Create custom widgets for specific UI elements

b) Apply styling using themes and custom styles

Lab Session 7: a) Design a form with various input fields

b) Implement form validation and error handling

Lab Session 8: a) Add animations to UI elements using flutter's animation framework

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

Lab Session 9: a) Fetch data from REST API

b) Display the fetched data in a meaningful way in the UI

Lab Session 10: a) Write unit tests for UI components

b) Use Flutter's debugging tools to identify and fix issues

1.1 Brief overview of the lab manual

The lab manual of UI design with Flutter typically provides a structured guide for students to learn and practice designing user interfaces using Flutter, a popular framework for building cross-platform mobile applications. The manual usually covers various topics, including:

Introduction to Flutter: An overview of Flutter, its features, and benefits for mobile app development.

Setting Up Development Environment: Instructions for setting up Flutter and its dependencies on different platforms like Windows, macOS, and Linux.

Flutter Basics: Fundamentals of Flutter, including widgets, layouts, navigation, state management, and basic UI design principles.

Building UI Components: Step-by-step tutorials for creating common UI components such as buttons, text fields, lists, and forms using Flutter widgets.

Styling and Theming: Guidelines for applying styles, themes, colors, fonts, and other visual elements to enhance the appearance of the app.

Responsive Design: Techniques for designing responsive user interfaces that adapt to different screen sizes and orientations.

Navigation and Routing: How to implement navigation between screens and manage application routes using Flutter's navigation system.

State Management: Introduction to state management techniques in Flutter, including setState, Provider, Bloc, and other state management libraries.

API Integration: Integration of RESTful APIs to fetch and display data in the app, including handling asynchronous operations and error handling.

Testing and Debugging: Strategies for testing UI components, debugging common issues, and using Flutter's debugging tools effectively.

The lab manual typically includes hands-on exercises, code examples, and assignments to reinforce learning and practical skills. It may also provide additional resources such as recommended readings, online tutorials, and reference materials for further exploration of UI design concepts and Flutter development. Overall, the lab manual serves as a comprehensive guide for students to gain proficiency in designing user interfaces with Flutter.

1.2 Objectives of the laboratory sessions

The objectives of laboratory sessions in UI design with Flutter typically include:

Understanding Flutter Framework: Gain familiarity with the Flutter framework, its architecture, and key concepts for building cross-platform mobile applications.

Practical Experience: Provide hands-on experience to students in designing user interfaces using Flutter widgets, layouts, and components.

Applying UI Design Principles: Apply fundamental principles of UI design, such as consistency, clarity, and usability, to create intuitive and visually appealing user interfaces.

Implementing Responsive Design: Learn techniques for designing responsive user interfaces that adapt to different screen sizes and orientations, ensuring a consistent user experience across devices.

Styling and Theming: Understand how to apply styles, themes, colors, and typography to enhance the visual appearance of the app and maintain brand consistency.

Navigation and Routing: Learn how to implement navigation between screens and manage application routes using Flutter's navigation system.

State Management: Explore various state management techniques in Flutter, such as setState, Provider, Bloc, and others, to manage and update the state of UI components.

API Integration: Integrate RESTful APIs to fetch and display dynamic data in the app, including handling asynchronous operations and error handling.

Testing and Debugging: Practice testing UI components, debugging common issues, and using Flutter's debugging tools effectively to identify and fix errors.

Deployment and Publishing: Understand the process of building and deploying Flutter apps to different platforms like Android and iOS, and publishing them to app stores.

Collaboration and Communication: Foster collaboration and communication skills by working in teams on UI design projects, sharing ideas, and providing feedback to peers.

Problem-Solving Skills: Develop problem-solving skills by identifying UI design challenges, brainstorming solutions, and implementing effective design solutions using Flutter.

Explore Algorithmic Complexity and Efficiency: Analyze the time and space complexity of different AI algorithms, and develop an understanding of algorithmic efficiency and optimization strategies.

Enhance Critical Thinking and Problem-Solving Abilities: Engage in critical thinking and analytical reasoning while designing and implementing solutions to complex AI problems, fostering problem-solving skills essential for AI development.

Experiment with Different Approaches: Experiment with various AI approaches, including search algorithms, heuristic methods, machine learning techniques, and optimization algorithms, to understand their strengths, weaknesses, and applications.

Foster Collaboration and Communication: Collaborate with peers on problem-solving tasks, discuss algorithm design choices, and communicate findings effectively, promoting teamwork and effective communication skills.

Build a Strong Foundation in AI Theory and Practice: Develop a strong foundation in both theoretical concepts and practical applications of artificial intelligence, preparing students for further study or professional work in the field.

Encourage Creativity and Innovation: Encourage creativity and innovation in problem-solving, allowing students to explore alternative solutions, adapt existing algorithms, and develop novel approaches to AI challenges.

Promote Lifelong Learning in AI: Cultivate a passion for lifelong learning and exploration in the field of artificial intelligence, inspiring students to continue their education and pursue advancements in AI technology throughout their careers.

1.3 Prerequisites for students

Before participating in laboratory sessions for UI design with Flutter, students should ideally have the following prerequisites:

Basic Programming Skills: Students should have a basic understanding of programming concepts, including variables, data types, control structures, and functions. Knowledge of object-oriented programming (OOP) concepts like classes, objects, and inheritance would be beneficial.

Familiarity with Dart Programming Language: Flutter uses the Dart programming language, so students should have a basic understanding of Dart syntax, features, and concepts. Topics such as variables, functions, classes, and asynchronous programming with Future and Stream would be important.

Understanding of UI Design Principles: Familiarity with fundamental principles of user interface (UI) design, such as consistency, simplicity, visual hierarchy, and usability, would be helpful. Students should have a basic understanding of how to design user-friendly and visually appealing interfaces.

Experience with Mobile Development: While not mandatory, prior experience with mobile app development using frameworks like Flutter, React Native, or native development (Android/iOS) would be advantageous. This includes understanding app lifecycle, navigation patterns, and platform-specific

considerations.

Development Environment Setup: Students should be able to set up and configure their development environment for Flutter development. This includes installing Flutter SDK, Dart SDK, and necessary development tools like IDE (e.g., Android Studio, IntelliJ IDEA, Visual Studio Code) and Flutter plugins/extensions.

Problem-Solving Skills: Students should have strong problem-solving skills and the ability to troubleshoot issues independently. This includes debugging code, researching solutions online, and seeking help when needed.

Critical Thinking and Creativity: UI design involves creative thinking and problem-solving to create intuitive and aesthetically pleasing user interfaces. Students should be able to think critically, analyze design requirements, and propose innovative solutions.

Time Management: Managing time effectively is crucial for completing lab assignments, meeting deadlines, and optimizing productivity during lab sessions. Students should be able to prioritize tasks, allocate time wisely, and stay organized.

Effective Communication: Communication skills, both written and verbal, are essential for collaborating with peers, discussing design ideas, and presenting project outcomes. Students should be able to articulate their thoughts, provide constructive feedback, and work effectively in teams.

While these prerequisites provide a foundation for successful participation in UI design laboratory sessions, students with varying levels of experience and backgrounds can benefit from engaging with the course material and actively participating in hands-on activities and projects.

1.4 Necessary software and hardware requirements

To effectively participate in UI design laboratory sessions with Flutter, students will need the following software and hardware requirements:

Software Requirements:

Flutter SDK: Install the Flutter SDK, which includes the Flutter framework and the Dart programming language. Flutter provides installation instructions for various platforms such as Windows, macOS, and Linux.

Integrated Development Environment (IDE):

Recommended: Android Studio with Flutter plugin or Visual Studio Code with Flutter extension. Both IDEs provide excellent support for Flutter development and include features

like code completion, debugging, and project management tools.

Alternatively, students can use IntelliJ IDEA with Flutter plugin or any other text editor of their choice, but full-featured IDEs are recommended for better productivity.

Dart SDK: The Dart SDK is included with the Flutter SDK, but students can also install it separately if needed. It provides command-line tools for Dart development and is required for

running Dart programs outside of Flutter projects.

Mobile Emulators or Devices:

Install Android Studio and set up Android Virtual Device (AVD) for running Android emulators.

For iOS development, students will need a Mac computer with Xcode installed to run the iOS simulator or test apps on physical iOS devices.

Package Management: Flutter uses Dart's package manager, Pub, to manage dependencies. Students should have access to the internet to download packages from Pub and add them to their Flutter projects.

Hardware Requirements:

Computer: Students will need a computer (desktop or laptop) running Windows, macOS, or Linux to develop Flutter applications. The computer should meet the minimum system requirements for running the chosen IDE and emulators/simulators.

Mobile Devices (Optional): While emulators/simulators are sufficient for testing and development, students may also want to test their apps on physical Android and iOS devices. Ensure compatibility with Flutter by enabling developer mode and USB debugging on the devices.

Internet Connection: A stable internet connection is necessary for downloading Flutter SDK, IDEs, packages from Pub, and accessing online resources and documentation.

Storage: Sufficient storage space on the computer to install Flutter SDK, IDEs, and other development tools, as well as to store project files and dependencies.

By meeting these software and hardware requirements, students can effectively engage in UI design laboratory sessions with Flutter, develop mobile applications, and gain practical experience in building user interfaces for various platforms.

Minimum System requirements:

The minimum system requirements for developing Flutter applications are relatively modest, but they may vary slightly depending on the operating system and the specific development tools being used.

Here are the general minimum system requirements:

For Windows:

Operating System: Windows 7 SP1 or later (64-bit)

Processor: Intel Core i3 or AMD equivalent (64-bit)

Memory (RAM): 4GB RAM or more

Storage: 2GB of free disk space

Graphics Card: Integrated graphics or dedicated GPU with DirectX 9 support

Internet Connection: Required for downloading development tools, SDKs, and dependencies

For macOS:

Operating System: macOS 10.12 (Sierra) or later

Processor: Intel Core i3 or higher

Memory (RAM): 4GB RAM or more

Storage: 2GB of free disk space

Graphics Card: Integrated graphics or dedicated GPU

Internet Connection: Required for downloading development tools, SDKs, and dependencies

For Linux:

Operating System: Ubuntu 16.04 LTS (Xenial Xerus) or later, or another Linux distribution with similar support

Processor: Intel Core i3 or AMD equivalent

Memory (RAM): 4GB RAM or more

Storage: 2GB of free disk space

Graphics Card: Integrated graphics or dedicated GPU

Internet Connection: Required for downloading development tools, SDKs, and dependencies

These are general guidelines, and the actual system requirements may vary depending on factors such as the size and complexity of the Flutter projects, the number of plugins and dependencies used, and the performance of the development tools (IDEs) being used. Additionally, running emulators or simulators for testing on mobile devices may require more system resources.

Lab Session 1:

a) Install flutter and Dart sdk

To install Flutter and the Dart SDK, you can follow these steps:

- a) Download Flutter:** Visit the Flutter website's Get Started page and download the Flutter SDK for your operating system (Windows, macOS, or Linux).
- b) Extract the Flutter SDK:** After downloading, extract the contents of the compressed file to a location on your computer where you want to store the Flutter SDK. For example, you can extract it to C:\flutter on Windows, /Users/<your-username>/flutter on macOS, or ~/flutter on Linux.
- c) Add Flutter to your PATH:** Update your system's PATH variable to include the Flutter bin directory. This step allows you to execute Flutter commands from any directory in your terminal or command prompt. The precise steps for updating the PATH vary depending on your operating system.

Windows:

From the Start search bar, type 'env' and select 'Edit the system environment variables'.

Click on 'Environment Variables'.

Under 'System Variables', find the 'Path' variable, select it, and click 'Edit'.

Click 'New' and add the path to the bin directory inside the Flutter directory (e.g., C:\flutter\bin).

Click 'OK' on all open dialogs to save your changes.

macOS and Linux:

Open a terminal window.

Run the following command to open the profile file associated with your terminal (.bash_profile, .bashrc, .zshrc, or similar):

```
nano ~/.bash_profile
```

Add the following line at the end of the file:

```
export PATH="$PATH:/path/to/flutter/bin"
```

Press Ctrl + X to exit, then Y to save changes, and Enter to confirm.

- d) Verify the Flutter installation:** Open a new terminal window, and run the following command to verify that Flutter is properly installed:

```
flutter --version
```

This command should display the Flutter version and other relevant information if the installation was successful.

- e) Install Flutter dependencies:** Depending on your development environment, you may need to install additional dependencies, such as Android Studio to fully set up your Flutter development environment.
- f) Download Dart SDK (if not bundled with Flutter):** Flutter comes with the Dart SDK bundled, so if you've installed Flutter, you should have the Dart SDK as well. However, if you need to install Dart separately, you can download it from the Dart "SDK archive".

b) Write a simple dart program to understand the language basics

// Define a main function, which is the entry point of a Dart program.

```
void main() {
```

```
// Variables and data types
```

```
int myNumber = 10;
```

```
double myDouble = 3.14;
```

```
String myString = 'Hello World';
```

```
bool myBool = true;
```

```

// Printing variables
    print('My number is: $myNumber');

print('My double is: $myDouble');
print('My string is: $myString');
print('My boolean is: $myBool');

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

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

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

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

// Maps
Map<String, int> ages = {
    'Kiran': 30,
    'Raj': 25,
    'Alekyia': 35,
};
print('Kiran\'s age: ${ages['Kiran']}');
}

```

Output:

```
My number is: 10
My double is: 3.14
My string is: Hello World
My boolean is: true
Result of addition: 15
myBool is true
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
First element of the list: 1
Length of the list: 5
Kiran's age: 30
```

Lab Session 2:

a) Explore various flutter widgets

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

Layout Widgets:

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

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

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

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

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

Text and Styling Widgets:

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

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

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

Input Widgets:

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

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

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

Button Widgets:

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

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

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

Image and Icon Widgets:

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

Icon: Displays a Material Design icon.

Navigation Widgets:

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

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

Animation Widgets:

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

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

Material Design Widgets:

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

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

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

Cupertino (iOS-style) Widgets:

CupertinoNavigationBar: A navigation bar in the iOS style.

CupertinoButton: A button widget with the iOS style.

CupertinoTextField: A text field widget with the iOS style.

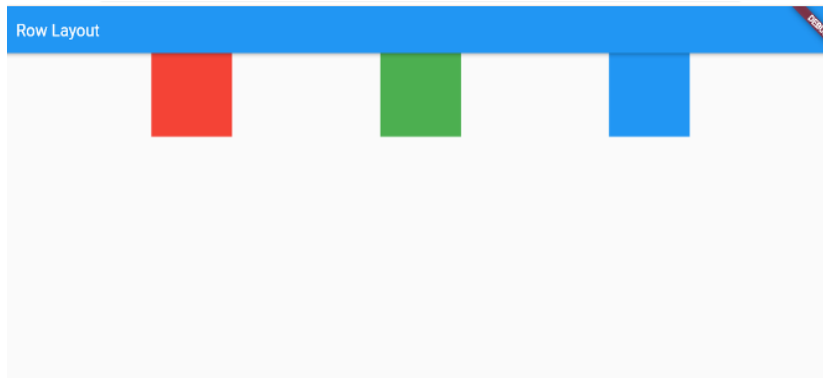
These are just a few examples of the many widgets available in Flutter. Each widget comes with its set of properties and customization options, allowing developers to create highly customizable and responsive user interfaces.

b) User implement different layout structures using Row, Column, and Stack widgets

1. Row Layout:

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

Output:

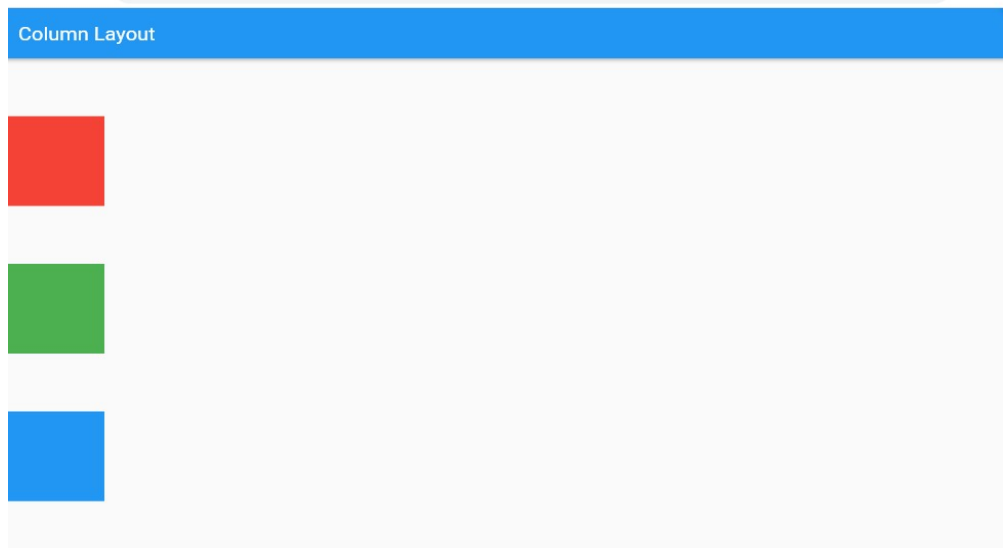


2. Column Layout:

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

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

Output:



3. Stack Layout:

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

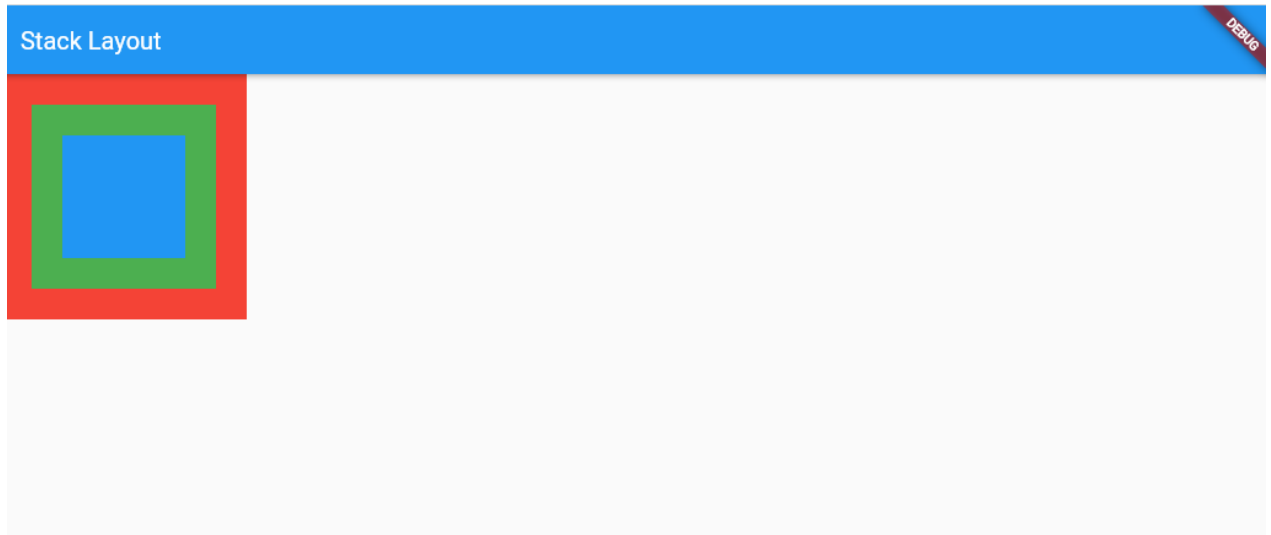
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Stack Layout'),
        ),
        body: Stack(
          alignment: Alignment.center,
          children: <Widget>[

            Container(
              color: Colors.red,
              width: 200,
              height: 200,
            ),

            Container(
              color: Colors.green,
              width: 150,
              height: 150,
            ),
            Container(
              color: Colors.blue,
              width: 100,
              height: 100,
            ),
          ],
        ),
      ),
    );
  }
}
```

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

Output:



Lab Session 3:

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

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

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

class ResponsiveHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Responsive UI Demo'),
      ),
      body: LayoutBuilder(
        builder: (BuildContext context, BoxConstraints constraints) {
          if (constraints.maxWidth < 600) {
            return _buildNarrowLayout();
          } else {
            return _buildWideLayout();
          }
        },
      ),
    );
  }

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

```

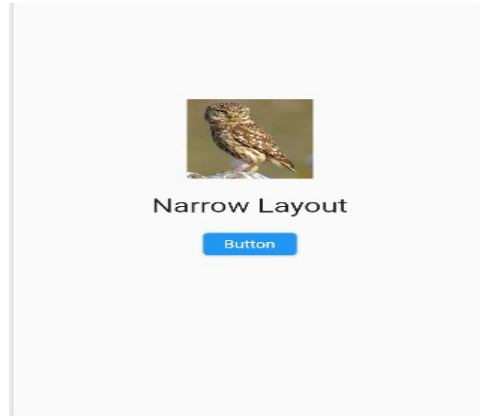
        SizedBox(height: 20),
        ElevatedButton(
          onPressed: () {},
          child: Text('Button'),
        ),
      ],
    ),
  );
}

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

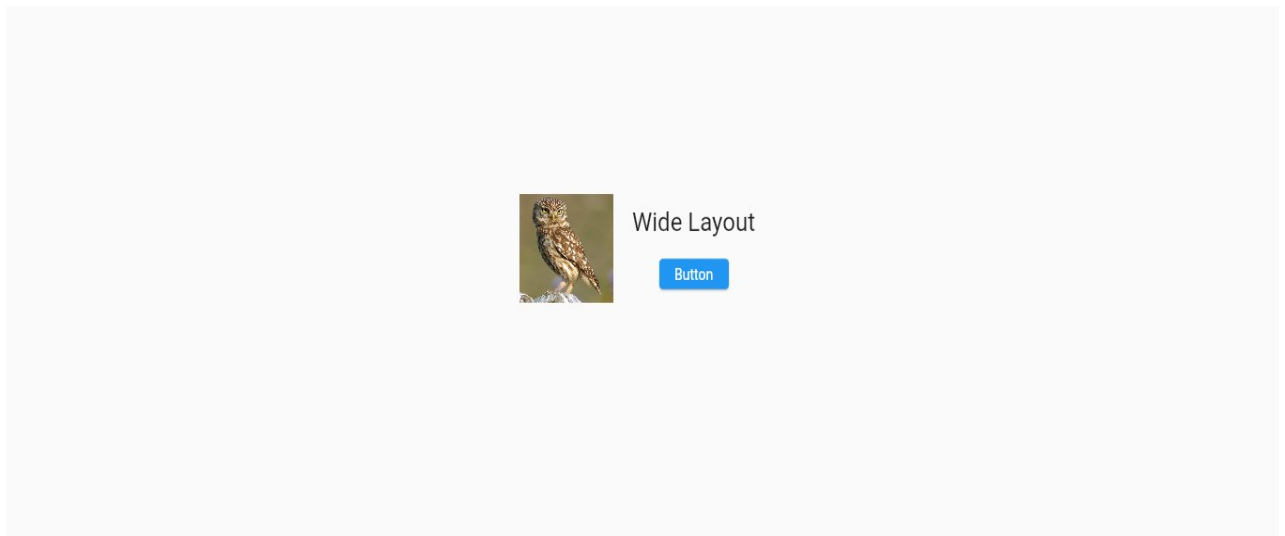
```

Output:

Mobile View:



Desktop View:



b) Implement media queries and breakpoints for responsiveness

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

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

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

class ResponsiveHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Responsive UI with Media Queries'),
      ),
      body: LayoutBuilder(
        builder: (BuildContext context, BoxConstraints constraints) {
          if (constraints.maxWidth < 600) {
            return _buildMobileLayout();
          } else if (constraints.maxWidth < 1200) {
            return _buildTabletLayout();
          } else {
            return _buildDesktopLayout();
          }
        },
      ),
    );
  }

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

```

        SizedBox(height: 20),
        ElevatedButton(
          onPressed: () {},
          child: Text('Button'),
        ),
      ],
    ),
  );
}

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

```

```

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

```

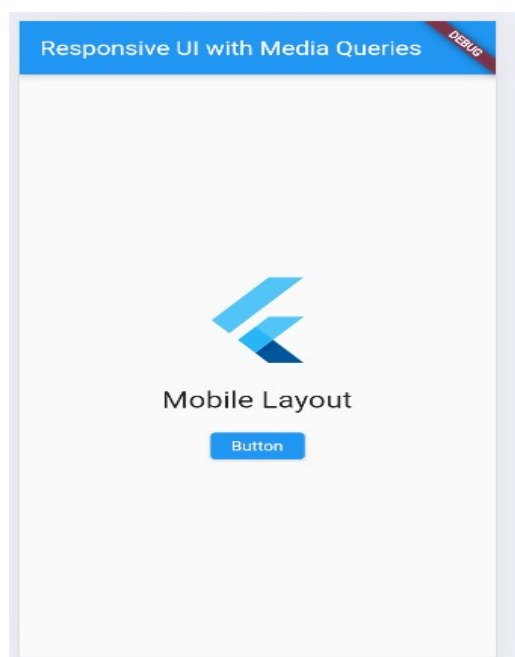
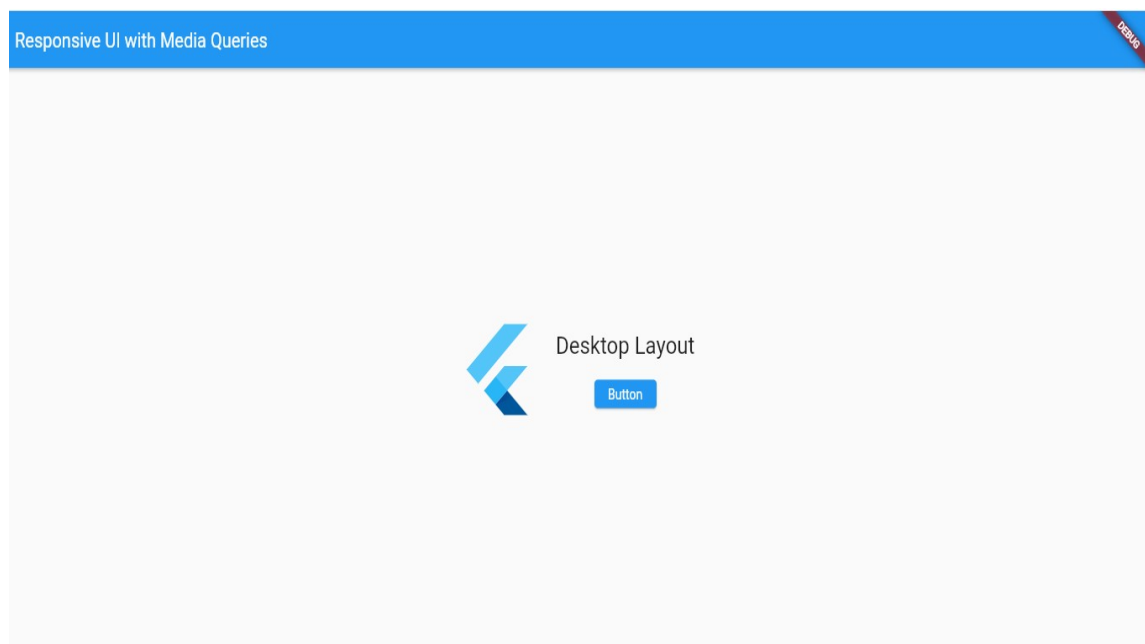


```

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

```

Output:



Lab Session 4:

a) Setup navigation between different screens using navigator

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

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

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

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

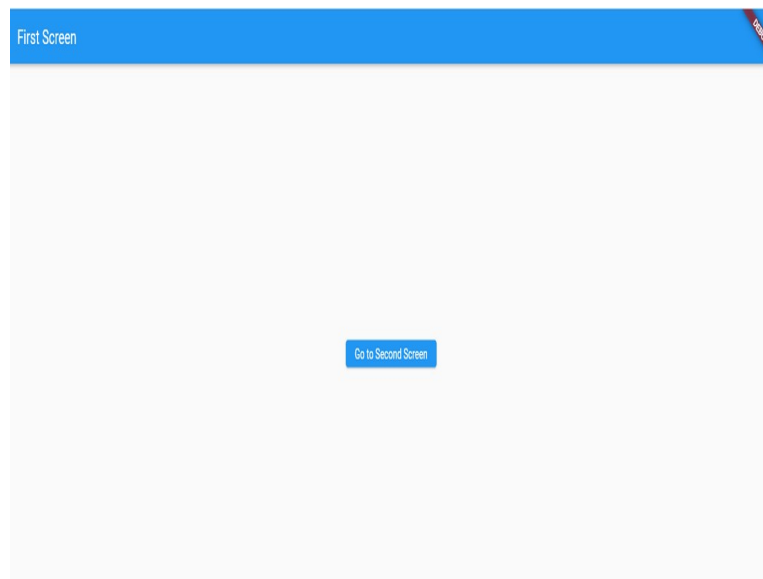
class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
```

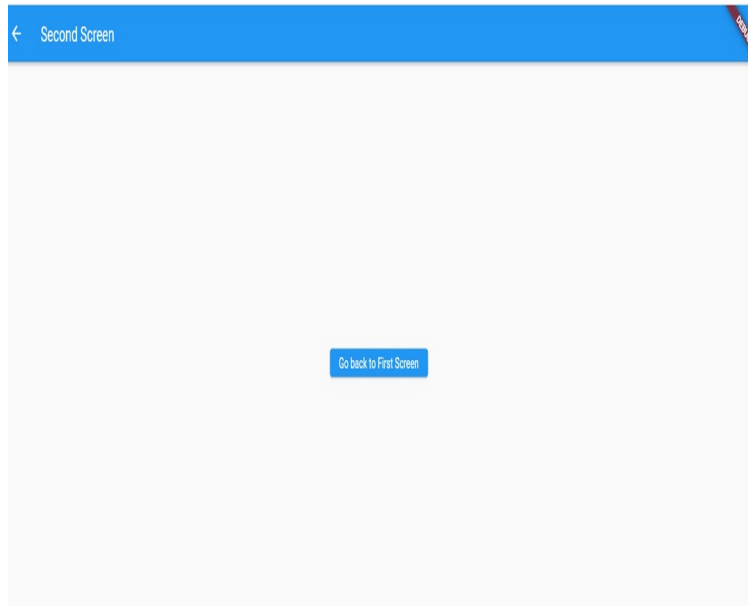
```

appBar: AppBar(
  title: Text('Second Screen'),
),
body: Center(
  child: ElevatedButton(
    onPressed: () {
      // Navigate back to the first screen
      Navigator.pop(context);
    },
    child: Text('Go back to First Screen'),
  ),
),
);
}
}

```

Output:





b) Implement navigation with named routes

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Named Routes Demo',
      initialRoute: '/',
      routes: {
        '/': (context) => HomeScreen(),
        '/second': (context) => SecondScreen(),
        '/third': (context) => ThirdScreen(),
      },
    );
  }
}

class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Home Screen'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
```

```

        Navigator.pushNamed(context, '/second');
    },
    child: Text('Go to Second Screen'),
  ),
),
);
}
}

```

```

class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Second Screen'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {

            Navigator.pushNamed(context, '/third');
          },
          child: Text('Go to Third Screen'),
        ),
      ),
    );
  }
}

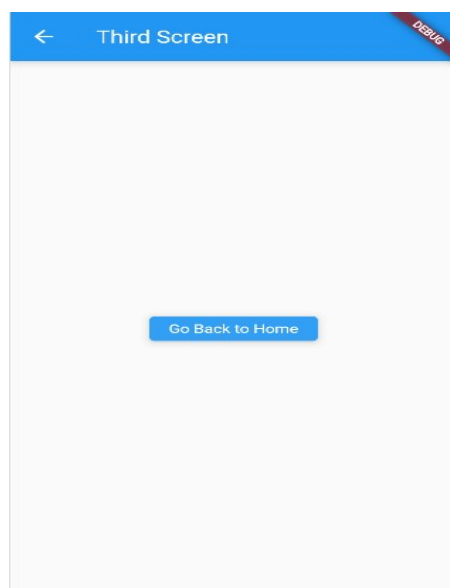
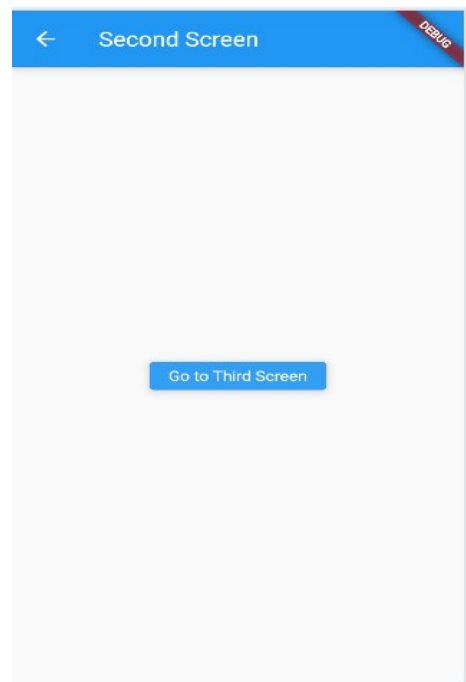
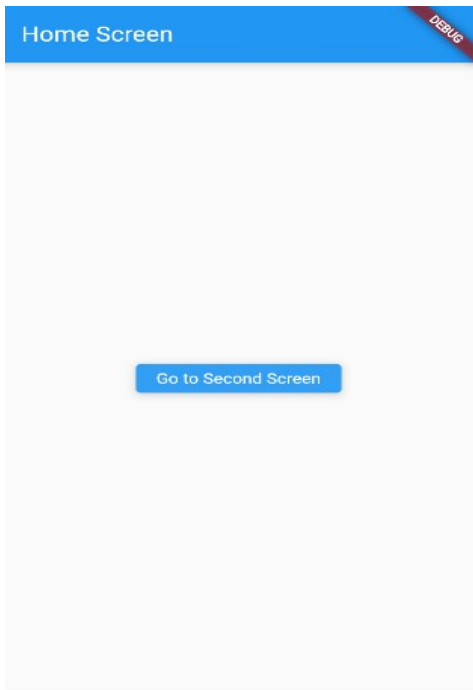
```

```

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

```

Output:



Lab Session 5:

a) Learn about stateful and stateless widgets

In Flutter, widgets can be categorized into two main types based on their behavior regarding state management: stateful widgets and stateless widgets.

Stateless Widgets:

Definition: Stateless widgets are widgets that do not have any mutable state. Once created, their properties (configuration) cannot change.

Characteristics:

They are immutable and lightweight.

They only depend on their configuration and the build context provided during construction.

Their appearance (UI) is purely a function of their configuration.

They are ideal for UI elements that do not change over time, such as static text labels, icons, or simple buttons.

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

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

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text('Cards Example'),  
        ),  
        body: CardList(),  
      ),  
    );  
  }  
}
```

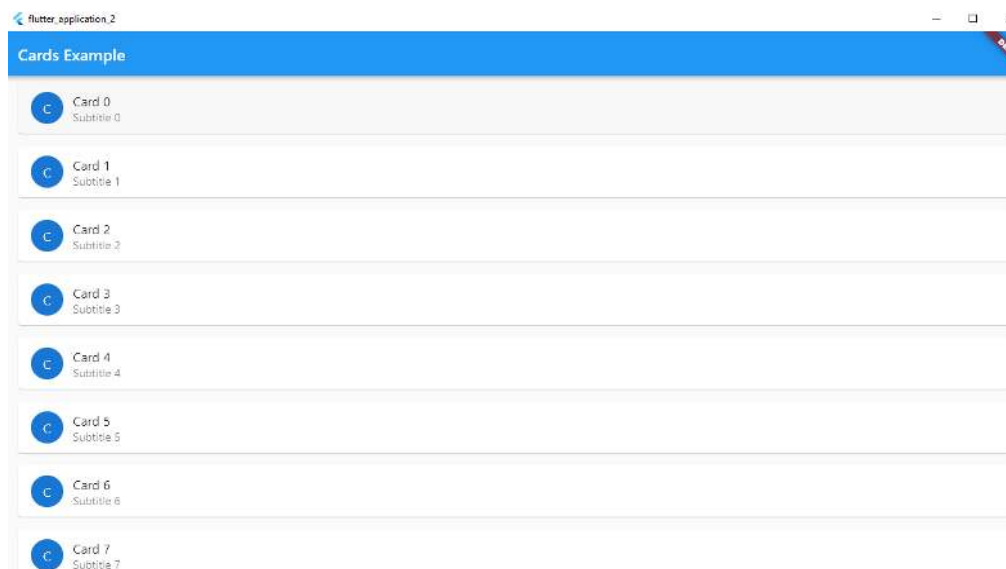
```
class CardList extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return ListView.builder(  
      itemCount: 10,  
      itemBuilder: (context, index) {  
        return CardItem(  
          title: 'Card $index',  
          subtitle: 'Subtitle $index',  
        );  
      },  
    );  
  }  
}
```

```
class CardItem extends StatelessWidget {  
  final String title;  
  final String subtitle;  
  
  const CardItem({  
    Key key,  
    @required this.title,  
    @required this.subtitle,  
  }) : super(key: key);
```

```
  @override
```

```
Widget build(BuildContext context) {
  return Card(
    margin: EdgeInsets.symmetric(horizontal: 16, vertical: 8),
    child: ListTile(
      title: Text(title),
      subtitle: Text(subtitle),
      leading: CircleAvatar(
        child: Text('${title.substring(0, 1)}'),
      ),
      onTap: () {
        // Handle card tap
      },
    ),
  );
}
```

Output:



b) Implement state management using set state and provider

Stateful Widgets:

Definition: Stateful widgets are widgets that maintain state, allowing them to change and update over time in response to user actions, network events, or other factors.

Characteristics:

They have an associated mutable state that can change during the widget's lifetime. The state is stored in a separate class that extends `State` and is associated with the stateful widget. Changes to the state trigger a rebuild of the widget's UI, allowing dynamic updates. They are ideal for UI elements that need to change or react to user interactions, such as input forms, animations, or scrollable lists.

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

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

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: CounterApp(),  
    );  
  }  
}
```

```
class CounterApp extends StatefulWidget {  
  @override  
  _CounterAppState createState() => _CounterAppState();  
}
```

```
class _CounterAppState extends State<CounterApp> {  
  int _counter = 0;
```

```
  void _incrementCounter() {  
    setState(() {  
      _counter++;  
    });  
  }  
}
```

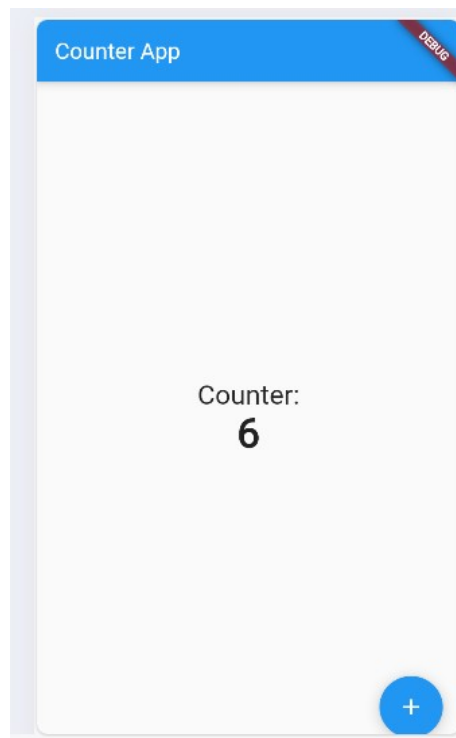
```
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text('Counter App'),  
      ),  
      body: Center(  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: <Widget>[  
         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

```

    ],
  ),
),
floatingActionButton: FloatingActionButton(
  onPressed: _incrementCounter,
  tooltip: 'Increment',
  child: Icon(Icons.add),
),
);
}
}

```

Output:



Stateful widgets are composed of two classes: the stateful widget itself (which extends StatefulWidget) and its corresponding state class (which extends State). The state class is responsible for maintaining the widget's mutable state and updating the UI accordingly via the `setState()` method.

stateless widgets are static and immutable, while stateful widgets are dynamic and can change over time by managing their internal state. Understanding the difference between these two types of widgets is essential for designing and building efficient and responsive Flutter UIs.

State Management using `setState()`:

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

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

```
class MyApp extends StatelessWidget {
  @override
```

```

Widget build(BuildContext context) {
  return MaterialApp(
    home: CounterPage(),
  );
}

class CounterPage extends StatefulWidget {
  @override
  _CounterPageState createState() => _CounterPageState();
}

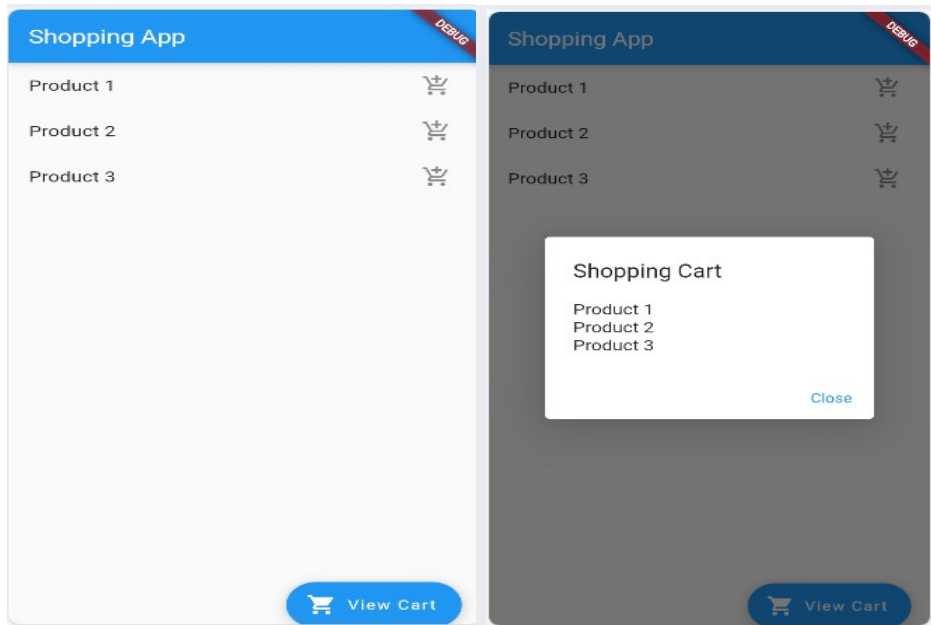
class _CounterPageState extends State<CounterPage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Counter Example (setState)'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'Counter Value:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}

```

Output:



State Management using provider package:

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

import 'provider/movie_provider.dart';

import 'screens/home_screen.dart';

void main() {
  runApp(ChangeNotifierProvider<MovieProvider>(
    child: const MyApp(),
    create: (_) => MovieProvider(), // Create a new ChangeNotifier object
  ));
}

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

  @override
  Widget build(BuildContext context) {

    return MaterialApp(
      // Remove the debug banner
      debugShowCheckedModeBanner: false,
      title: 'State Management using provider',
      theme: ThemeData(
        primarySwatch: Colors.indigo,
      ),

      home: const HomeScreen(),
    );
  }
}
```

```
}
```

create a model folder for models and create file movie.dart

```
class Movie {  
  final String title;  
  final String? runtime; // how long this movie is (in minute)  
  
  Movie({required this.title, this.runtime});  
}
```

Create a provider folder and create movie_provider.dart inside the provider folder

```
// provider/movie_provider.dart  
import 'package:flutter/material.dart';  
import 'dart:math';  
import '../models/movie.dart';  
  
// A list of movies  
final List<Movie> initialData = List.generate(  
  50,  
  (index) => Movie(  
    title: "Movie $index",  
    runtime: "${Random().nextInt(100) + 60} minutes");  
  
class MovieProvider with ChangeNotifier {  
  // All movies (that will be displayed on the Home screen)  
  final List<Movie> _movies = initialData;  
  
  // Retrieve all movies  
  List<Movie> get movies => _movies;  
  
  // Favorite movies (that will be shown on the MyList screen)  
  final List<Movie> _myList = [];  
  
  // Retrieve favorite movies  
  List<Movie> get myList => _myList;  
  
  // Adding a movie to the favorites list  
  void addToList(Movie movie) {  
    _myList.add(movie);  
    notifyListeners();  
  }  
  
  // Removing a movie from the favorites list  
  void removeFromList(Movie movie) {  
    _myList.remove(movie);  
    notifyListeners();  
  }  
}
```

Create a screens folder for screens

Create home_screen.dart for home screen page

```
// screens/home_screen.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import '../provider/movie_provider.dart';
import 'my_list_screen.dart';

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

  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
  @override
  Widget build(BuildContext context) {

    var movies = context.watch<MovieProvider>().movies;
    var myList = context.watch<MovieProvider>().myList;

    return Scaffold(
      appBar: AppBar(
        title: const Text('State Management using provider'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(15),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.stretch,
          children: [
            ElevatedButton.icon(
              onPressed: () {
                Navigator.of(context).push(
                  MaterialPageRoute(
                    builder: (context) => const MyListScreen(),
                  ),
                );
              },
              icon: const Icon(Icons.favorite),
              label: Text(
                "Go to my list (${myList.length})",
                style: const TextStyle(fontSize: 24),
              ),
            ),

            style: ElevatedButton.styleFrom(
              primary: Colors.red,
              padding: const EdgeInsets.symmetric(vertical: 20)),
          ),
        ],
      ),
    );
```

```

const SizedBox(
  height: 15,
),
Expanded(
  child: ListView.builder(
    itemCount: movies.length,
    itemBuilder: (_, index) {

      final currentMovie = movies[index];
      return Card(
        key: ValueKey(currentMovie.title),
        color: Colors.amberAccent.shade100,
        elevation: 4,
        child: ListTile(
          title: Text(currentMovie.title),
          subtitle:
            Text(currentMovie.runtime ?? 'No information'),
          trailing: IconButton(
            icon: Icon(
              Icons.favorite,
              color: myList.contains(currentMovie)
                ? Colors.red
                : Colors.white,

              size: 30,
            ),
            onPressed: () {
              if (!myList.contains(currentMovie)) {
                context
                  .read<MovieProvider>()
                  .addToList(currentMovie);
              } else {
                context
                  .read<MovieProvider>()
                  .removeFromList(currentMovie);
              }
            },
          ),
        ),
      );
    }
  ),
);

```

create my_list_screen.dart inside the screens folder

```
// screens/my_list_screen.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import '../provider/movie_provider.dart';

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

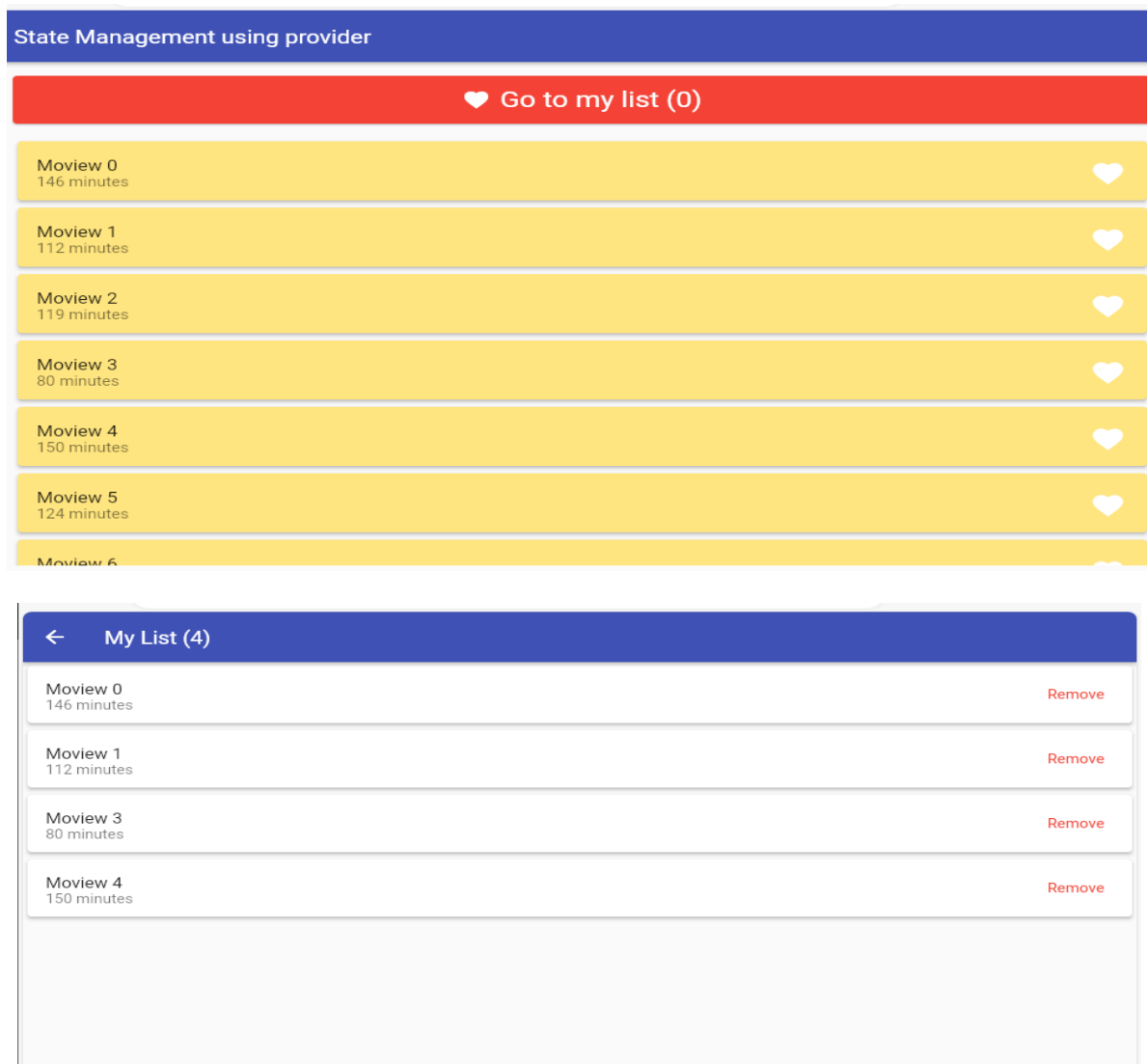
  @override
  State<MyListScreen> createState() => _MyListScreenState();
}

class _MyListScreenState extends State<MyListScreen> {
  @override
  Widget build(BuildContext context) {
    final myList = context.watch<MovieProvider>().myList;
    return Scaffold(
      appBar: AppBar(
        title: Text("My List (${myList.length})"),
      ),
      body: ListView.builder(
        itemCount: myList.length,
        itemBuilder: (_, index) {
          final currentMovie = myList[index];

          return Card(
            key: ValueKey(currentMovie.title),
            elevation: 4,

            child: ListTile(
              title: Text(currentMovie.title),
              subtitle: Text(currentMovie.runtime ?? ""),
              trailing: TextButton(
                child: const Text(
                  'Remove',
                  style: TextStyle(color: Colors.red),
                ),
                onPressed: () {
                  context.read<MovieProvider>().removeFromList(currentMovie);
                },
              ),
            ),
          );
        },
      ),
    );
  }
}
```


Output:



we use the provider package to manage state. We define a Counter class that extends `ChangeNotifier`, and the counter value is stored inside it. Whenever the counter is incremented, we call `notifyListeners()` to inform the listeners about the change.

Lab Session 6:

a) Create custom widgets for specific UI elements

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Custom Widget Example'),
        ),
        body: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Padding(
              padding: const EdgeInsets.all(8.0),
              child: CustomTextField(
                hintText: 'Enter your name',
                onChanged: (value) {
                  print('Name changed: $value');
                },
              ),
            ),
            SizedBox(height: 20),
            Padding(
              padding: const EdgeInsets.all(8.0),
              child: CustomTextField(
                hintText: 'Enter Email',
                onChanged: (value) {
                  print('Name changed: $value');
                },
              ),
            ),
            SizedBox(height: 20),
            Padding(
              padding: const EdgeInsets.all(8.0),
              child: CustomTextField(
                hintText: 'Enter Roll Number',
                onChanged: (value) {
                  print('Name changed: $value');
                },
              ),
            ),
            SizedBox(height: 20),
            CustomButton(
              text: 'Press Me',
```

```

        onPressed: () {
          print('Button pressed!');
        },
      ),
    ],
  ),
),
);
}
}

```

```

class CustomButton extends StatelessWidget {
  final String? text;
  final VoidCallback? onPressed;

  const CustomButton({
    Key? key,
    @required this.text,
    @required this.onPressed,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      child: Text(text!),
    );
  }
}

```

```

class CustomTextField extends StatelessWidget {
  final String hintText;
  final ValueChanged<String> onChanged;

  const CustomTextField({
    Key? key,
    required this.hintText,
    required this.onChanged,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return TextField(
      onChanged: onChanged,
      decoration: InputDecoration(
        hintText: hintText,
        border: OutlineInputBorder(),
      ),
    );
  }
}

```

Output:

b) Apply styling using themes

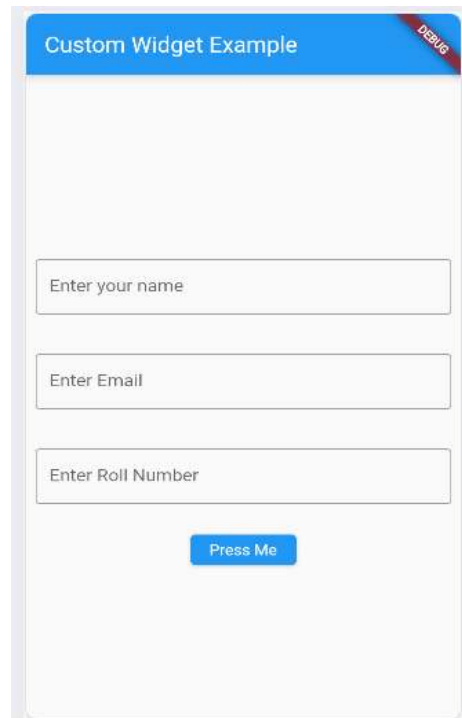
In Flutter, you can apply styling to custom styles to maintain consistency visually appealing.

import

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

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData(
        // Define the overall theme of the app
        primaryColor: Colors.blue,
        accentColor: Colors.orange,
        fontFamily: 'Roboto',
        textTheme: TextTheme(
          headline1: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),
          bodyText1: TextStyle(fontSize: 16),
        ),
        elevatedButtonTheme: ElevatedButtonThemeData(
          style: ElevatedButton.styleFrom(
            primary: Colors.blue,
            textStyle: TextStyle(fontSize: 18),
            padding: EdgeInsets.symmetric(horizontal: 20, vertical: 15),
            shape: RoundedRectangleBorder(
              borderRadius: BorderRadius.circular(10),
            ),
          ),
        ),
      ),
      home: HomePage(),
    );
  }
}
```

```
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```



and custom styles

your widgets using themes and
and make your UI more

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

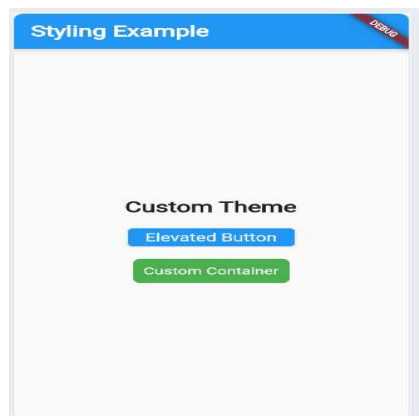
```
StatelessWidget {
{
```

```

return Scaffold(
  appBar: AppBar(
    title: Text('Styling Example'),
  ),
  body: Center(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text(
          'Welcome to MyApp',
          style: Theme.of(context).textTheme.headline1,
        ),
        SizedBox(height: 20),
        ElevatedButton(
          onPressed: () {},
          child: Text('Get Started'),
        ),
      ],
    ),
  ),
);
}

```

Output:



In this example:

We define a custom theme using ThemeData and apply it to the entire app using the theme property of MaterialApp.

The theme specifies primary and accent colors, a custom font family, and text styles for different text elements (headline6 and bodyText2).

We customize the appearance of the elevated buttons using ElevatedButtonThemeData and ElevatedButton.styleFrom.

In the HomePage widget, we use Theme.of(context) to access the custom theme properties and apply them to various widgets such as Text and ElevatedButton.

We also demonstrate custom styling for a Container widget with a custom background color and border radius.

Using themes and custom styles like this helps maintain a consistent visual identity throughout your app and makes it easier to manage and update styling across multiple widgets.

Lab Session 7:

a) Design a form with various input fields

form with various input fields such as text fields, checkboxes, radio buttons, and a dropdown menu
import 'package:flutter/material.dart';

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

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Form Example',
```

```

        theme: ThemeData(
          primarySwatch: Colors.blue,
        ),
        home: FormPage(),
      );
    }
  }

class FormPage extends StatefulWidget {
  @override
  _FormPageState createState() => _FormPageState();
}

class _FormPageState extends State<FormPage> {
  final _formKey = GlobalKey<FormState>();

  String _name;
  String _email;

  bool _subscribeToNewsletter = false;
  String _selectedCountry = 'USA';

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Form Example'),
      ),

      body: Padding(
        padding: EdgeInsets.all(20.0),
        child: Form(
          key: _formKey,
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: <Widget>[
              TextFormField(
                decoration: InputDecoration(labelText: 'Name'),
                onSave: (value) {
                  _name = value;
                },
              ),
              SizedBox(height: 20),
              TextFormField(
                decoration: InputDecoration(labelText: 'Email'),
                onSave: (value) {
                  _email = value;
                },
              ),
              SizedBox(height: 20),
              Row(
                children: <Widget>[

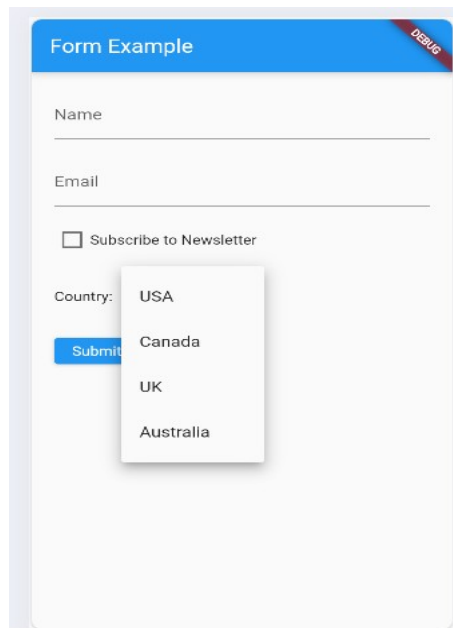
```

```

Checkbox(
  value: _subscribeToNewsletter,
  onChanged: (value) {
    setState(() {
      _subscribeToNewsletter = value;
    });
  },
),
Text('Subscribe to Newsletter'),
],
),
 SizedBox(height: 20),
Row(
  children: <Widget>[
    Text('Country: '),
    SizedBox(width: 20),
    DropdownButton<String>(
      value: _selectedCountry,
      onChanged: (value) {
        setState(() {
          _selectedCountry = value;
        });
      },
      items: <String>['USA', 'Canada', 'UK', 'Australia']
        .map<DropdownMenuItem<String>>((String value) {
          return DropdownMenuItem<String>(
            value: value,
            child: Text(value),
          );
        }).toList(),
    ),
  ],
),
 SizedBox(height: 20),
ElevatedButton(
  onPressed: () {
    _formKey.currentState.save();
    // Submit the form data
    print('Name: $_name');
    print('Email: $_email');
    print('Subscribe to Newsletter: $_subscribeToNewsletter');
    print('Country: $_selectedCountry');
  },
  child: Text('Submit'),
),
],
),
),
),
);
}
}

```


Output:

A screenshot of a mobile application interface titled "Form Example". The form contains several input fields: a text field for "Name", a text field for "Email", a checkbox labeled "Subscribe to Newsletter", and a dropdown menu for "Country". The dropdown menu is currently open, showing a list of countries: "USA", "Canada", "UK", and "Australia". Below the dropdown is a blue "Submit" button. The form is styled with a light gray background and a blue header bar.

b) Implement form validation and error handling

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Form Example'),
        ),

        body: SingleChildScrollView(
          padding: EdgeInsets.all(16),
          child: FormWidget(),
        ),
      ),
    );
  }
}

class FormWidget extends StatefulWidget {
  @override
  _FormWidgetState createState() => _FormWidgetState();
}

class _FormWidgetState extends State<FormWidget> {
```

```

final _formKey = GlobalKey<FormState>();

String _name;
String _email;
String _password;
String _phone;
String _address;

@override
Widget build(BuildContext context) {
  return Form(
    key: _formKey,
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: <Widget>[
        TextFormField(
          decoration: InputDecoration(labelText: 'Name'),
          validator: (value) {
            if (value.isEmpty) {
              return 'Please enter your name';
            }
            return null;
          },
          onSaved: (value) => _name = value,
        ),
        SizedBox(height: 16),
        TextFormField(
          decoration: InputDecoration(labelText: 'Email'),
          keyboardType: TextInputType.emailAddress,
          validator: (value) {
            if (value.isEmpty) {
              return 'Please enter your email';
            }
            // Add more complex email validation logic if needed
            return null;
          },
          onSaved: (value) => _email = value,
        ),
        SizedBox(height: 16),
        TextFormField(
          decoration: InputDecoration(labelText: 'Password'),
          obscureText: true,
          validator: (value) {
            if (value.isEmpty) {
              return 'Please enter a password';
            }
            // Add more complex password validation logic if needed
            return null;
          },
          onSaved: (value) => _password = value,
        ),

```

```

    SizedBox(height: 16),
    TextFormField(
      decoration: InputDecoration(labelText: 'Phone'),
      keyboardType: TextInputType.phone,
      validator: (value) {
        if (value.isEmpty) {
          return 'Please enter your phone number';
        }
        // Add more complex phone number validation logic if needed
        return null;
      },
      onSaved: (value) => _phone = value,
    ),
    SizedBox(height: 16),
    TextFormField(
      decoration: InputDecoration(labelText: 'Address'),
      maxLines: 3,
      validator: (value) {
        if (value.isEmpty) {
          return 'Please enter your address';
        }
        return null;
      },
      onSaved: (value) => _address = value,
    ),
    SizedBox(height: 16),
    ElevatedButton(
      onPressed: _submitForm,
      child: Text('Submit'),
    ),
  ],
),
);
}

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

    // Perform form submission with the saved form data
    print('Form submitted:');
    print('Name: $_name');

    print('Email: $_email');
    print('Password: $_password');
    print('Phone: $_phone');
    print('Address: $_address');
  }
}
}

```

Output:

Form Example

Design

Name

Email

Password

Phone

Address

Submit

Lab Session 8:

a) Add animations to UI elements using flutter's animation framework

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Animation Example'),
        ),
        body: AnimationWidget(),
      ),
    );
  }
}

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

class _AnimationWidgetState extends State<AnimationWidget>
  with SingleTickerProviderStateMixin {
  AnimationController _controller;
  Animation<double> _animation;

  @override
  void initState() {
    super.initState();
    _controller = AnimationController(
      duration: Duration(seconds: 1),
      vsync: this,
    );
    _animation = Tween<double>(begin: 0, end: 300).animate(_controller)
      ..addListener() {
        setState(() {}); // Trigger rebuild when animation value changes
      };
  }

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
```

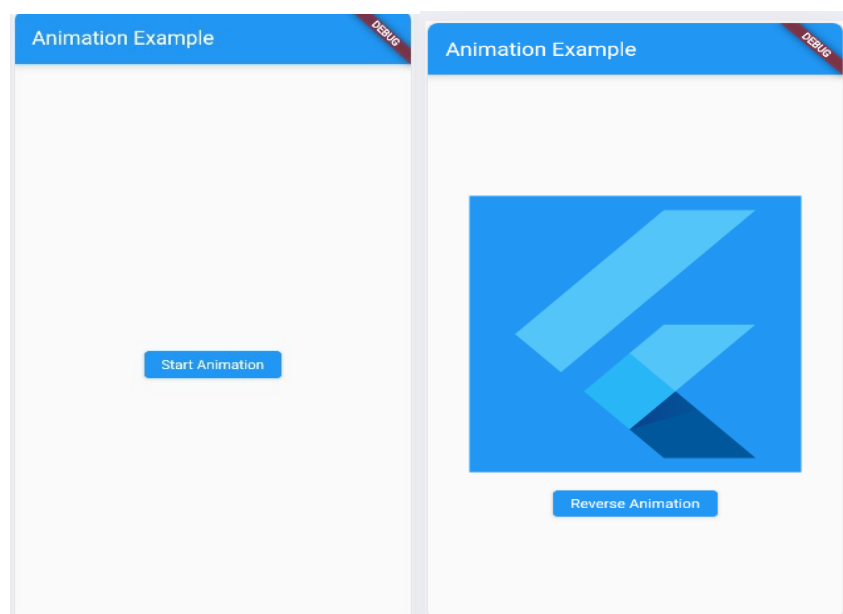
```

Container(
  width: _animation.value,
  height: _animation.value,
  color: Colors.blue,
  child: FlutterLogo(size: 100),
),
),
),
],
),
);
}

@override
void dispose() {
  _controller.dispose();
  super.dispose();
}
}

```

Output:



We define an Animation object with Tween to define the range of values for the animation.

I

inside the initState() method, we initialize the animation controller and define the animation. We use addListener() to trigger a rebuild when the animation value changes.

In the build method, we use the animated value _animation.value to control the size of the Container, which contains the FlutterLogo.

The ElevatedButton toggles the animation between forward and reverse based on the status of the animation controller.

You can customize the animation further by adjusting the duration, adding curves, or chaining multiple animations together.

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

Fade Animation:

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

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

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text('Fade Animation Example'),  
        ),  
        body: FadeAnimation(),  
      ),  
    );  
  }  
}
```

```
class FadeAnimation extends StatefulWidget {  
  @override  
  _FadeAnimationState createState() => _FadeAnimationState();  
}
```

```
class _FadeAnimationState extends State<FadeAnimation>  
  with SingleTickerProviderStateMixin {  
  AnimationController _controller;  
  Animation<double> _animation;
```

```
  @override  
  void initState() {  
    super.initState();  
    _controller = AnimationController(  
      vsync: this,  
      duration: Duration(seconds: 2),  
    );  
    _animation = Tween<double>(  
      begin: 0.0,
```

```

        end: 1.0,
      ).animate(_controller);

      _controller.forward();
    }

    @override
    Widget build(BuildContext context) {
      return Center(
        child: FadeTransition(
          opacity: _animation,
          child: Container(
            width: 200,
            height: 200,
            color: Colors.blue,
          ),
        ),
      );
    }

    @override
    void dispose() {
      _controller.dispose();
      super.dispose();
    }
  }
}

```

Slide Animation:

```

import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Slide Animation Example'),
        ),
        body: SlideAnimation(),
      ),
    );
  }
}

class SlideAnimation extends StatefulWidget {
  @override
  _SlideAnimationState createState() => _SlideAnimationState();
}

```



```

class _SlideAnimationState extends State<SlideAnimation>
with SingleTickerProviderStateMixin {
  AnimationController _controller;
  Animation<Offset> _animation;
  @override
  void initState() {
    super.initState();
    _controller = AnimationController(
      vsync: this,
      duration: Duration(seconds: 2),
    );
    _animation = Tween<Offset>(
      begin: Offset(-1.0, 0.0),
      end: Offset(0.0, 0.0),
    ).animate(_controller);
    _controller.forward();
  }

  @override
  Widget build(BuildContext context) {
    return Center(
      child: SlideTransition(
        position: _animation,
        child: Container(
          width: 200,
          height: 200,
          color: Colors.blue,
        ),
      ),
    );
  }

  @override
  void dispose() {
    _controller.dispose();
    super.dispose();
  }
}

```

Scale Animation:

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

```

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

```

```

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Scale Animation Example'),
        ),
      ),
    );
  }
}

```

```

        body: ScaleAnimation(),
    ),
);

}
}

class ScaleAnimation extends StatefulWidget {
  @override
  _ScaleAnimationState createState() => _ScaleAnimationState();
}

class _ScaleAnimationState extends State<ScaleAnimation>
  with SingleTickerProviderStateMixin {
  AnimationController _controller;
  Animation<double> _animation;

  @override
  void initState() {
    super.initState();
    _controller = AnimationController(
      vsync: this,
      duration: Duration(seconds: 2),
    );
    _animation = Tween<double>(
      begin: 0.0,
      end: 1.0,
    ).animate(_controller);
    _controller.forward();
  }

  @override
  Widget build(BuildContext context) {
    return Center(
      child: ScaleTransition(
        scale: _animation,
        child: Container(
          width: 200,
          height: 200,
          color: Colors.blue,
        ),
      ),
    ),
  );
}

@override
void dispose() {
  _controller.dispose();
  super.dispose();
}
}

```

Output:



Lab Session 9:

a) Fetch data from REST API

add dependency in pubspec.yaml:

dependencies:

flutter:

 sdk: flutter

http: ^0.13.3

enable internet permissions in your AndroidManifest.xml file for Android apps like below.

```
<uses-permission android:name="android.permission.INTERNET"
```

```
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
```

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

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: HomePage(),
    );
  }
}
```

```
class HomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}
```

```
class _HomePageState extends State<HomePage> {
  List<dynamic> _data = [];
  @override
  void initState() {
    super.initState();
    _fetchDataFromApi();
  }
```

```
Future<void> _fetchDataFromApi() async {
  final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));

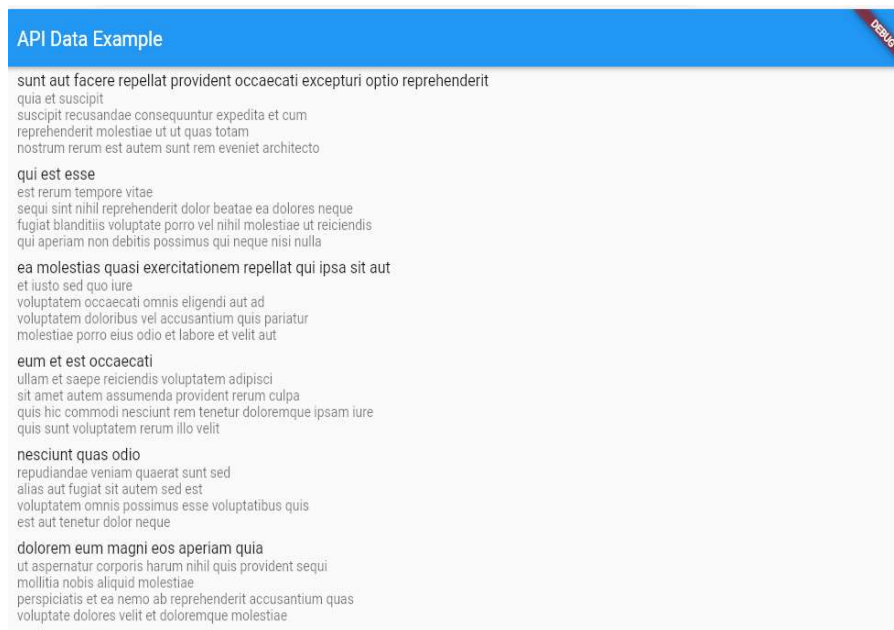
  if (response.statusCode == 200) {
    setState(() {
      _data = json.decode(response.body);
    });
  } else {
    throw Exception('Failed to load data');
  }
}
```

```

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('API Data Example'),
    ),
    body: ListView.builder(
      itemCount: _data.length,
      itemBuilder: (context, index) {
        return ListTile(
          title: Text(_data[index]['title']),
          subtitle: Text(_data[index]['body']),
        );
      },
    ),
  );
}

```

Output:



b) Display the fetched data in a meaningful way in the UI

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

```
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: HomePage(),
    );
  }
}

class HomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  List<dynamic> _data = [];
  bool _isLoading = false;

  @override
  void initState() {
    super.initState();
    _fetchDataFromApi();
  }

  Future<void> _fetchDataFromApi() async {
    setState(() {
      _isLoading = true;
    });

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

    if (response.statusCode == 200) {
      setState(() {
        _data = json.decode(response.body);
        _isLoading = false;
      });
    } else {
      throw Exception('Failed to load data');
    }
  }
}
```

```

    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('API Data Example'),
      ),
      body: _isLoading
        ? Center(
            child: CircularProgressIndicator(),
          )
        : ListView.builder(
            itemCount: _data.length,
            itemBuilder: (context, index) {
              return PostCard(
                title: _data[index]['title'],
                body: _data[index]['body'],
              );
            },
          ),
    );
  }
}

```

```

class PostCard extends StatelessWidget {
  final String title;
  final String body;

  const PostCard({
    Key key,
    @required this.title,
    @required this.body,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Card(
      margin: EdgeInsets.symmetric(horizontal: 16, vertical: 8),
      child: Padding(
        padding: EdgeInsets.all(16),

        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: <Widget>[
            Text(
              title,
              style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
            ),
            SizedBox(height: 8),
            Text(
              body,

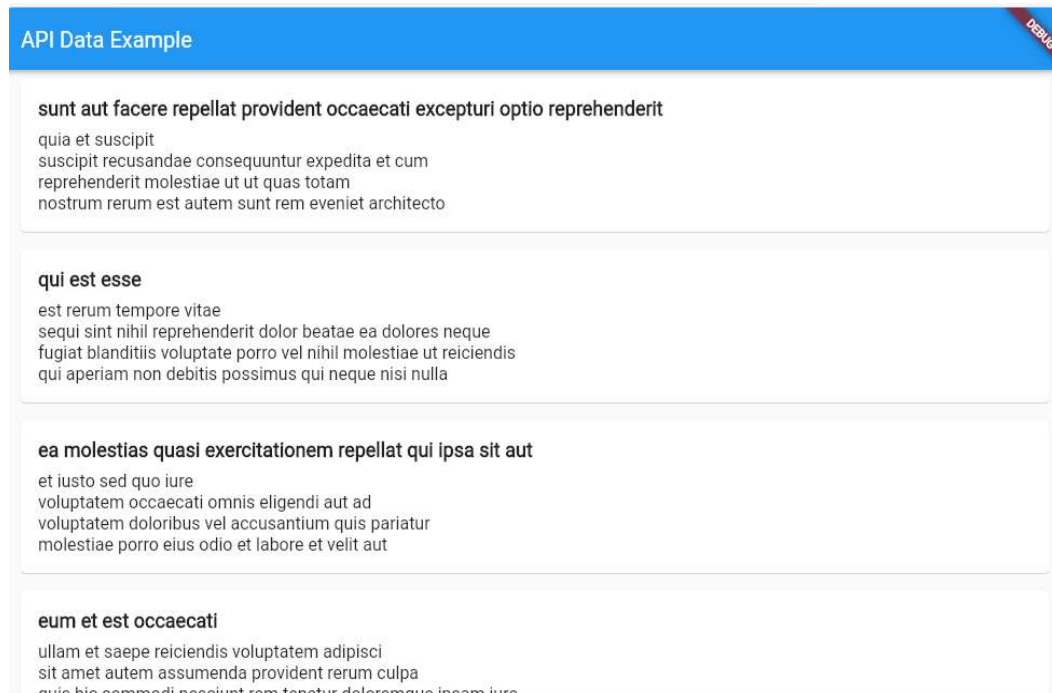
```

```

        style: TextStyle(fontSize: 16),
      ),
    ],
  ),
),
);
}
}

```

Output:



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

Lab Session 10:

a) Write unit tests for UI components

Unit tests for UI components in Flutter typically involves using the flutter_test package along with the test package for writing tests.

Here's how we can write unit tests for the PostCard widget:

```
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:your_app/post_card.dart'; // Import your widget file

void main() {
  testWidgets('PostCard displays title and body', (WidgetTester tester) async {
    // Build our widget and trigger a frame.
    await tester.pumpWidget(
      MaterialApp(
        home: PostCard(
          title: 'Test Title',
          body: 'Test Body',
        ),
      ),
    );

    // Verify that the title and body are displayed correctly.
    expect(find.text('Test Title'), findsOneWidget);
    expect(find.text('Test Body'), findsOneWidget);
  });

  testWidgets('PostCard widget has correct styling', (WidgetTester tester) async {
    // Build our widget and trigger a frame.
    await tester.pumpWidget(
      MaterialApp(
        home: PostCard(
          title: 'Test Title',
          body: 'Test Body',
        ),
      ),
    );

    // Verify that the text styles are applied correctly.
    final titleText = tester.widget<Text>(find.text('Test Title'));
    expect(titleText.style.fontSize, 18);
    expect(titleText.style.fontWeight, FontWeight.bold);

    final bodyText = tester.widget<Text>(find.text('Test Body'));
    expect(bodyText.style.fontSize, 16);
  });
}
```

In this test:

We use the testWidgets function from the flutter_test package to define our test cases.

In the first test case, we ensure that the PostCard widget correctly displays the provided title and body text.

In the second test case, we verify that the text styles applied to the title and body texts are as expected. We use expect statements to assert that the expected UI elements are found on the screen and that their properties match the expected values. Make sure to replace `your_app` with the appropriate package name where your PostCard widget resides.

Run the tests using the following command:

flutter test

This will execute all the tests defined in your project, including the ones written for the UI components.

Make sure to provide different test scenarios and edge cases to ensure comprehensive test coverage for your UI components.

b) Use flutter's debugging tools to identify and fix issues?

demonstrate the use of Flutter's debugging tools, let's consider a scenario where we have a simple counter app, but there's a bug where the counter is not incrementing when the "+" button is pressed. We'll use Flutter's debugging tools to identify and fix this issue.

Here's the code for the counter app:

```
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.