

Mobile Computing

Module 6 : Assignment 4 : Flutter

1. **Aim :** Write a program to demonstrate stateless widget.

Objective :

To understand and implement a stateless widget, showcasing its properties and how it differs from stateful widgets.

Theory:

In Flutter, widgets are categorized as either *stateless* or *stateful*. A stateless widget is a widget that does not maintain any state information. This means that it does not change based on user interactions or other data changes. Stateless widgets are suitable for displaying static content or content that does not require user input or interaction to be updated.

Stateless Widgets: These widgets are immutable; once created, their properties cannot change. Examples include **Text**, **Icon**, and custom widgets that do not involve dynamic updates.

Usage: Stateless widgets are often used when you want a widget's display to remain constant, for example, in headers, labels, or icons.

Structure: A stateless widget extends the **StatelessWidget** class, and its content is built through the **build** method. This method returns a widget tree that describes the appearance of the widget.

Code :

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

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

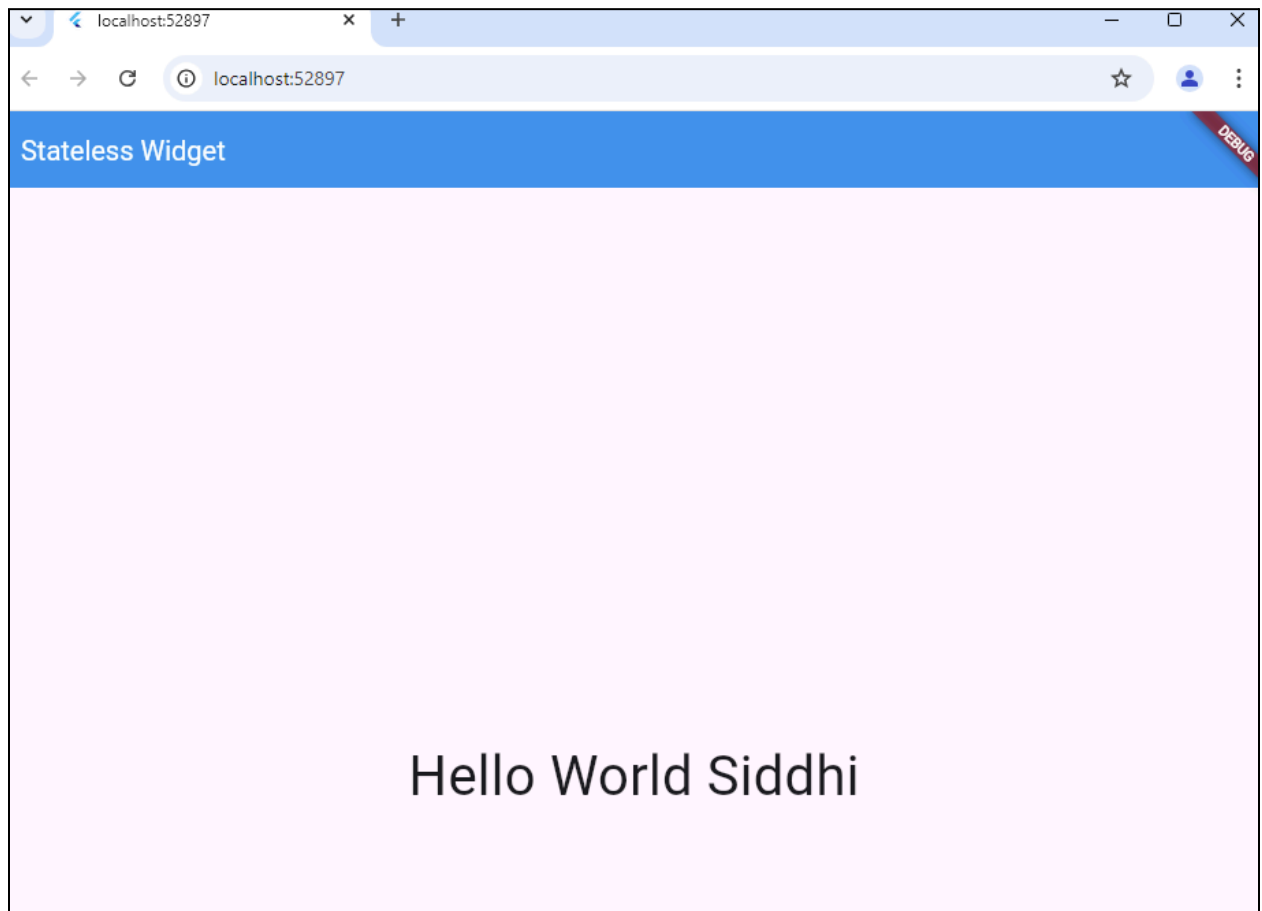
class MyApp extends StatelessWidget{
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("Stateless Widget"),
          backgroundColor: Colors.blue,
          titleTextStyle: const TextStyle(
            color: Colors.white,
```

```
        fontSize: 20
      ),
    ),
    body: const Center(
      child: Text("Hello World Siddhi",style: TextStyle(fontSize: 40.0),),
    ),
  ),
);

}
}
```

Output :



2. **Aim :** Write a program to demonstrate stateful widget.

Objective :

To understand and implement a stateful widget, illustrating how it maintains and manages its state across the lifecycle of a Flutter application.

Theory:

In Flutter, widgets are broadly categorized as *stateless* and *stateful*. A stateful widget is a widget that maintains state, which can change over time in response to user interactions or other factors.

Key Concepts:

1. **Stateful Widgets:** Unlike stateless widgets, stateful widgets can rebuild and update their appearance based on user actions or internal changes. This makes them essential for creating interactive applications that react dynamically to user inputs.
2. **Structure of a Stateful Widget:**
 - A stateful widget consists of two classes:
 - A widget class, which extends `StatefulWidget`.
 - A state class, which extends `State<StatefulWidget>` and contains the `build` method. This class holds the mutable state and is where the widget tree is built.
 - The `setState` method is used to trigger a rebuild of the widget, updating the UI whenever the state changes.
3. **Common Uses:** Stateful widgets are ideal for scenarios where data changes in response to user interactions, such as in forms, counters, and animations.

Code :

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

```
void main() {  
  runApp(MaterialApp(  
    title: "Stateful App",  
    home: FavouriteCity(),  
  
  ));  
}
```

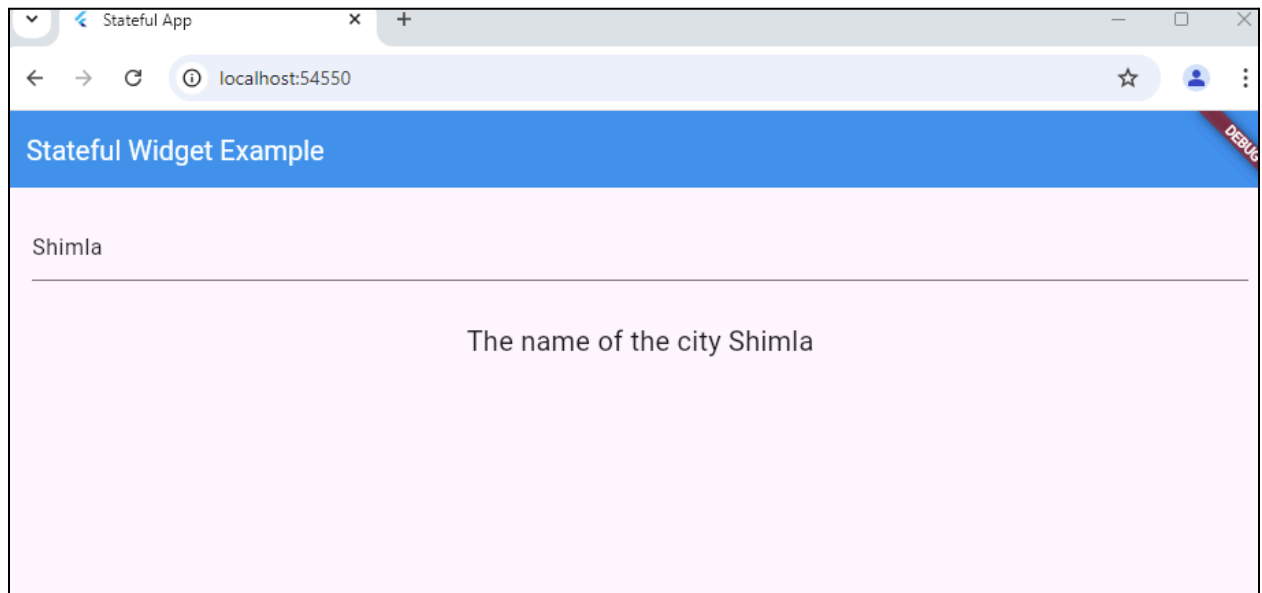
```
class FavouriteCity extends StatefulWidget {  
  const FavouriteCity({super.key});
```

```
@override
```

```
State<FavouriteCity> createState() => _FavouriteCityState();  
}
```

```
class _FavouriteCityState extends State<FavouriteCity> {  
  String nameCity="";  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text("Stateful Widget Example"),  
        backgroundColor: Colors.blue,  
        titleTextStyle: TextStyle(  
          color: Colors.white,  
          fontSize: 20,  
        ),  
      ),  
      body: Container(  
        margin: EdgeInsets.all(20),  
        child: Column(  
          children: <Widget>[  
            TextField(  
              onChanged: (String userInput){  
                setState(() {  
                  nameCity=userInput;  
                });  
              },  
            ),  
            Padding(padding: EdgeInsets.all(30.0),  
              child: Text("The name of the city $nameCity",  
                style: TextStyle(fontSize: 20),),  
            ),  
          ],  
        ),  
      ),  
    );  
  }  
}
```

Output :



3. Aim : Write a program to demonstrate ListView control in Flutter.

Objective :

To understand and implement the ListView widget in Flutter for creating scrollable lists and displaying items efficiently

Theory:

The ListView widget in Flutter is a powerful tool for displaying a scrollable list of widgets. It is commonly used to present a collection of items that users can scroll through, making it suitable for lists of text, images, and complex custom widgets.

Key Concepts:

1. ListView Basics:

- **ListView** is a widget that arranges its children in a scrollable list. It can be customized to display items in a horizontal or vertical scroll direction.
- **Default Behavior:** The default scroll direction for ListView is vertical, but it can be set to horizontal using the **scrollDirection** property.

2. Types of ListView:

- **ListView():** The basic constructor for ListView. Requires the full list of widgets to be provided upfront, which can be memory-intensive if handling large datasets.
- **ListView.builder:** Builds items on-demand as they come into view, making it more efficient for handling large datasets.
- **ListView.separated:** Provides a builder for both the list items and the separators between items, allowing for custom separators.
- **ListView.custom:** Offers full control over the rendering and layout, suitable for custom scroll views.

Code :

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

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

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

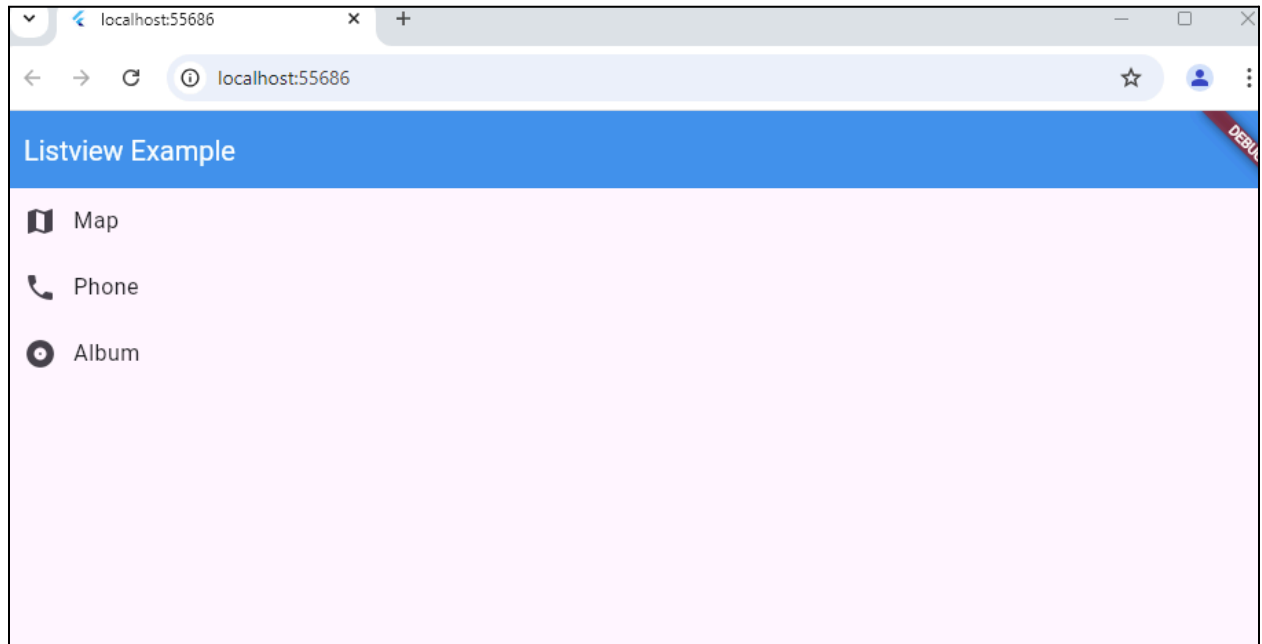
```
  @override  
  Widget build(BuildContext context) {
```

```
return MaterialApp(  
  home: Scaffold(  
    appBar: AppBar(  
      title: const Text("Listview Example"),  
      backgroundColor: Colors.blue,  
      titleTextStyle: const TextStyle(  
        color: Colors.white,  
        fontSize: 20,  
      ),  
    ),  
    body: const ListData(),  
  ),  
);  
}  
}
```

```
class ListData extends StatelessWidget {  
  const ListData({super.key}); // Fixed constructor
```

```
  @override  
  Widget build(BuildContext context) {  
    return ListView(  
      children: const <Widget>[  
        ListTile(  
          leading: Icon(Icons.map),  
          title: Text("Map"),  
        ),  
        ListTile(  
          leading: Icon(Icons.phone),  
          title: Text("Phone"),  
        ),  
        ListTile(  
          leading: Icon(Icons.album),  
          title: Text("Album"),  
        ),  
      ],  
    );  
  }  
}
```

Output :



4. **Aim :** Write a program to demonstrate Bottom Navigation Bar control in Flutter.

Objective :

To understand and implement the Bottom Navigation Bar widget in Flutter, enabling seamless navigation between different sections or views within a mobile application.

Theory:

The Bottom Navigation Bar in Flutter is a commonly used UI component that allows users to navigate between multiple screens or sections within an app. This widget is typically used in mobile applications where quick navigation between primary sections is essential, such as in social media, e-commerce, or dashboard applications.

Code :

> **main.dart**

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

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

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

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

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

  @override
  State<BottomNavigationBarExample> createState() => _BottomNavigationBarExampleState();
}

class _BottomNavigationBarExampleState extends State<BottomNavigationBarExample> {
  int _selectedIndex = 0;
```

```
static const TextStyle optionStyle = TextStyle(
  fontSize: 18,
);

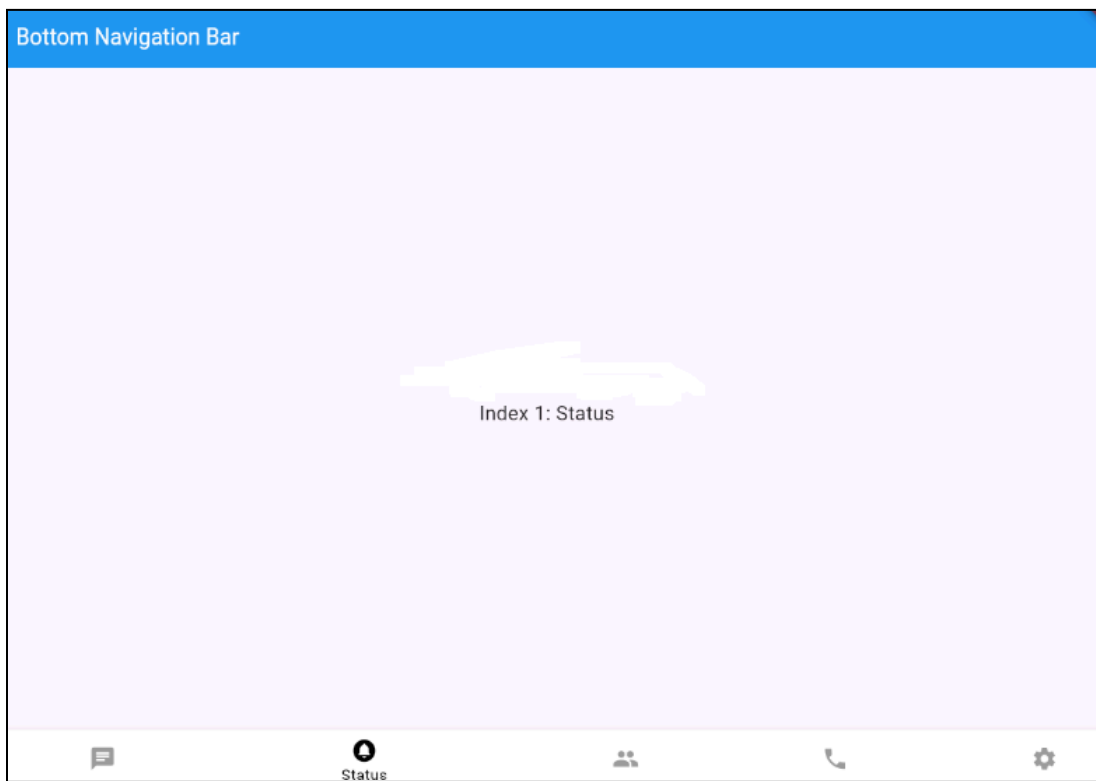
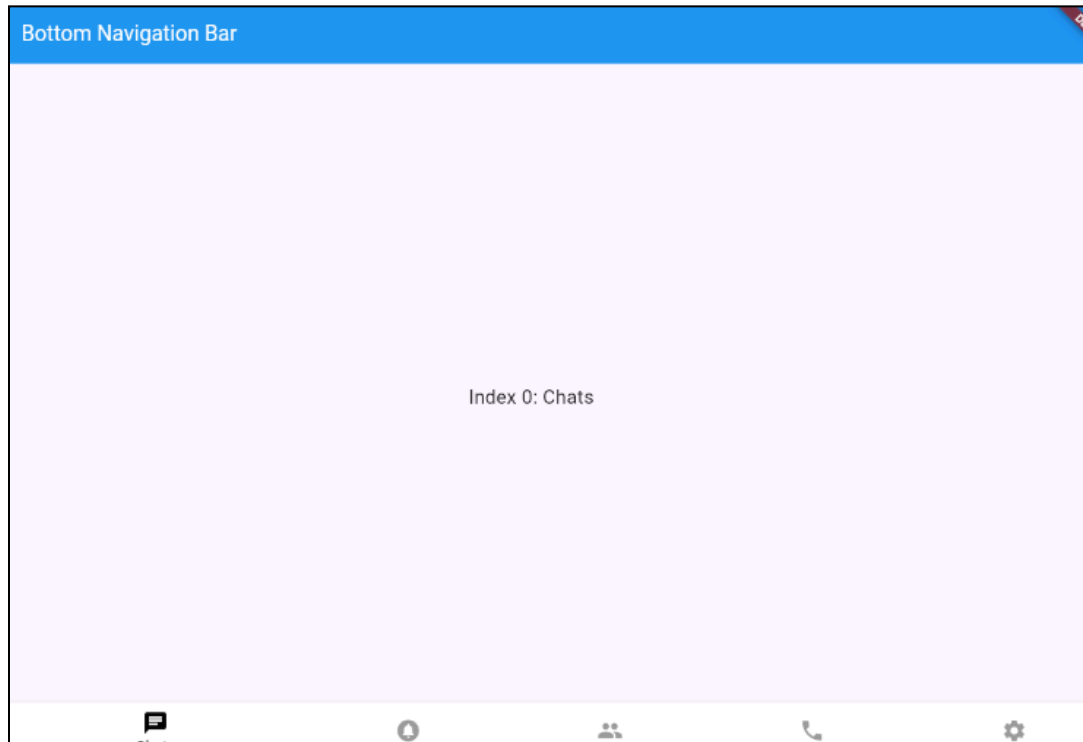
static const List<Widget> _widgetOptions = <Widget>[
  Text("Index 0: Chats", style: optionStyle),
  Text("Index 1: Status", style: optionStyle),
  Text("\nIndex 2: Communities", style: optionStyle),
  Text("\nIndex 3: Calls", style: optionStyle),
  Text("\nIndex 4: Settings", style: optionStyle),
];

void _onItemTapped(int index) {
  setState(() {
    _selectedIndex = index;
  });
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Bottom Navigation Bar"),
      backgroundColor: Colors.blue,
      titleTextStyle: TextStyle(
        color: Colors.white,
        fontSize: 20,
      ),
    ),
    body: Center(
      child: _widgetOptions.elementAt(_selectedIndex),
    ),
    bottomNavigationBar: BottomNavigationBar(
      items: const <BottomNavigationBarItem>[
        BottomNavigationBarItem(
          icon: Icon(Icons.chat),
          label: "Chats",
          backgroundColor: Colors.white,
        ),
        BottomNavigationBarItem(
          icon: Icon(Icons.circle_notifications),
          label: "Status",
          backgroundColor: Colors.white,
        ),
        BottomNavigationBarItem(
```

```
        icon: Icon(Icons.people),
        label: "Communities",
        backgroundColor: Colors.white,
    ),
    BottomNavigationBarItem(
        icon: Icon(Icons.call_sharp),
        label: "Calls",
        backgroundColor: Colors.white,
    ),
    BottomNavigationBarItem(
        icon: Icon(Icons.settings),
        label: "Settings",
        backgroundColor: Colors.white,
    ),
],
currentIndex: _selectedIndex,
selectedItemColor: Colors.black,
unselectedItemColor: Colors.grey,
onTap: _onItemTapped,
),
);
}
```

Output :



5. **Aim :** Write a program to demonstrate layout in Flutter.

Objective :

To understand and apply different layout widgets in Flutter for arranging and structuring UI elements in a mobile application.

Theory:

Layouts in Flutter are essential for designing responsive and organized user interfaces. Flutter offers a variety of layout widgets to help arrange UI components, including containers, rows, columns, stacks, and grids. Proper layout design ensures that elements are visually aligned and adapt to different screen sizes and orientations.

Code :

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: "Card Layout",
      home: ProductBox(),
    ); // MaterialApp
  }
}

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


class ProductBox extends StatelessWidget {

  @override
```

```
Widget build(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(  
      title: const Text("Card Layout"),  
      backgroundColor: Colors.blue,  
      titleTextStyle: TextStyle(  
        color: Colors.white,  
        fontSize: 20,  
      ),  
    ),  
    body: Center(  
      child: Container(  
        padding: const EdgeInsets.all(2),  
        child: Card(  
          child: Row(  
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
            children: <Widget>[  
              Image(  
                image: AssetImage("images/iPhone13.jpg"),  
              ),  
              Expanded(  
                child: Column(  
                  crossAxisAlignment: CrossAxisAlignment.start,  
                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
                  children: <Widget>[  
                    Text("Siddhi Kotre", style: textStyle),  
                    Text("Product: Apple iPhone 13", style: textStyle),  
                    Text("Description:", style: textStyle),  
                    Text("Apple iPhone 13: Featuring a 6.1-inch Super Retina XDR display and  
powered by the A15 Bionic chip, it offers stunning photography with a dual-camera system and  
exceptional battery life, all in a sleek design.", style: textStyle),  
                    Text("Price: Rs. 47,999", style: textStyle),  
                  ],  
                ),  
              ),  
            ],  
          ),  
        ),  
      ),  
    );  
  }  
}
```

Output :

Card Layout

A white Apple iPhone 13 is shown from a three-quarter perspective. The front screen displays a blue and green abstract wallpaper. The back of the phone shows the dual-camera system and the Apple logo.

Product: Apple iPhone 13

Description:

Apple iPhone 13: Featuring a 6.1-inch Super Retina XDR display and powered by the A15 Bionic chip, it offers stunning photography with a dual-camera system and exceptional battery life, all in a sleek design.

Price: Rs. 47,999

6. **Aim :** Write a Flutter program based on RestAPI

Objective :

To understand and implement data retrieval from a REST API in Flutter, using HTTP requests to fetch and display the data within a mobile app.

Theory:

REST (Representational State Transfer) is an architectural style that uses standard HTTP requests to interact with data. APIs that adhere to REST principles allow data to be accessed, modified, and deleted across platforms through stateless calls. In a RESTful setup, data is usually exchanged in JSON format, making it lightweight and compatible with mobile apps.

Flutter supports REST API integration through libraries like [http](#) and [dio](#). The [http](#) package is commonly used for simpler GET, POST, PUT, and DELETE operations. When working with REST APIs in Flutter:

1. Setting up dependencies: Add [http](#) to your [pubspec.yaml](#) file to perform HTTP requests.
2. Making API calls: Using functions like [http.get\(\)](#) or [http.post\(\)](#) to retrieve or send data to endpoints.
3. Parsing JSON data: The JSON response is parsed into Dart objects, which can then be rendered in Flutter widgets.
4. State management: Stateful widgets or state management libraries like Provider can manage data fetched from APIs to ensure efficient updating of the user interface.

Code :

main.dart

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

```
void main() {  
  runApp(const MaterialApp(  
    home: HomePage(),  
  ));  
}
```

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

```
@override
```



```
State<HomePage> createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  List data = []; // Initialized as an empty list

  Future<void> getData() async {
    var response = await http.get(
      Uri.parse("https://jsonplaceholder.typicode.com/posts"),
      headers: {"Accept": "application/json"},
    );
    setState(() {
      data = json.decode(response.body);
    });
  }

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("REST API using ListView"),
        backgroundColor: Colors.blue,
      ),
      body: Column(
        children: [
          const SizedBox(height: 20), // Space between AppBar and text
          const Center(
            child: Text(
              "Siddhi Kotre",
              style: TextStyle(
                fontSize: 20,
              ),
            ),
          ),
          const SizedBox(height: 20), // Space between text and ListView
          Expanded(
            child: data.isEmpty
              ? const Center(child: CircularProgressIndicator()) // Show loading spinner
              : ListView.builder(
                  itemCount: data.length,
```

```
        itemBuilder: (BuildContext context, int index) {  
            return Card(  
                child: ListTile(  
                    title: Text(data[index]["title"]),  
                    subtitle: Text(data[index]["body"]),  
                ),  
            );  
        },  
    ),  
),  
],  
),  
);  
}  
}
```

Output :

REST API using ListView
<p>sunt aut facere repellat provident occaecati excepturi optio reprehenderit quia et suscipit suscipit recusandae consequuntur expedita et cum reprehenderit molestiae ut ut quas totam nostrum rerum est autem sunt rem eveniet architecto</p>
<p>qui est esse est rerum tempore vitae sequi sint nihil reprehenderit dolor beatae ea dolores neque fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis qui aperiam non debitis possimus qui neque nisi nulla</p>
<p>ea molestias quasi exercitationem repellat qui ipsa sit aut et iusto sed quo iure voluptatem occaecati omnis eligendi aut ad voluptatem doloribus vel accusantium quis pariatur molestiae porro eius odio et labore et velit aut</p>

7. **Aim :** Write a program to perform serialization and deserialization of JSON in flutter.

Objective :

To understand the process of converting JSON data to and from Dart objects using serialization and deserialization in Flutter applications, enabling effective data handling for API-based applications.

Theory:

Serialization is the process of converting an object into a format that can be easily stored or transmitted, while deserialization is the reverse process—converting serialized data back into a usable object. In Flutter, JSON (JavaScript Object Notation) is commonly used as the format for exchanging data between an app and a backend server, especially with REST APIs.

Code :

UserModel.dart

```
class UserModel {  
  late String id;  
  late String fullname;  
  late String email;  
  late String percentage;  
  
  UserModel({  
    required this.id,  
    required this.fullname,  
    required this.email,  
    required this.percentage,  
  });  
  
  UserModel.fromMap(Map<String, dynamic> map) {  
    id = map["id"];  
    fullname = map["fullname"];  
    email = map["email"];  
    percentage = map["percentage"];  
  }  
  
  Map<String, dynamic> toMap() {  
    return {  
      "id": id,  
      "fullname": fullname,  
      "email": email,  
      "percentage": percentage,  
    };  
  }  
}
```

```
};  
}  
}
```

> **main.dart**

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

```
import 'UserModel.dart';
```

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

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

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

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

```
  @override  
  State<HomePage> createState() => _HomePageState();  
}
```

```
class _HomePageState extends State<HomePage> {  
  UserModel userObject = UserModel(  
    id: "14",  
    fullname: "Siddhi Kotre",  
    email: "s1062230050@timscdrmbai.in",  
    percentage: "80%",  
  );
```

```
    String userJson = '{"id": "50", "fullname": "Siddhi Kotre", "email":  
"s1062230050@timscdrmbai.in", "percentage": "80%"}';
```

```
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  

```

```
appBar: AppBar(  
  title: const Text("Json Operations"),  
  backgroundColor: Colors.blue,  
  titleTextStyle: const TextStyle(  
    color: Colors.white,  
    fontSize: 20,  
  ),  
,  
body: Center(  
  child: Column(  
    mainAxisAlignment: MainAxisAlignment.center,  
    children: <Widget>[  
      const Text(  
        "Siddhi Kotre",  
        style: TextStyle(  
          fontSize: 20,  
        ),  
      ),  
      const SizedBox(height: 20), // Space between text and buttons  
      Row(  
        mainAxisAlignment: MainAxisAlignment.center,  
        children: <Widget>[  
          ElevatedButton(  
            onPressed: () {  
              Map<String, dynamic> userMap = userObject.toMap();  
              var json = jsonEncode(userMap);  
              print("Serialized JSON: $json");  
            },  
            child: const Text("Serialize", style: TextStyle(fontSize: 20)),  
          ),  
          const SizedBox(width: 20),  
          ElevatedButton(  
            onPressed: () {  
              var decode = jsonDecode(userJson);  
              Map<String, dynamic> userMap = decode;  
              UserModel newUser = UserModel.fromMap(userMap);  
              print("Deserialized - Name: ${newUser.fullname}, Percentage:  
${newUser.percentage}");  
            },  
            child: const Text("Deserialize", style: TextStyle(fontSize: 20)),  
          ),  
        ],  
      ),  
    ],  
  ),  
,  
)
```

```
);  
}  
}
```

Output :

Json Operations

Serialize

Deserialize

8. **Aim :** Write a note on Introduction swift programming concept, objective c and comparison between objective C and Swift Programming

Objective :

To understand the fundamental concepts of Swift programming, gain insights into Objective-C, and compare the two languages to understand their respective strengths and use cases in iOS and macOS app development.

Theory:

Swift is a modern programming language developed by Apple in 2014 for iOS, macOS, watchOS, and tvOS app development. It was designed to be faster, safer, and more user-friendly than Objective-C, the previous standard for Apple platforms. Swift's syntax is concise and expressive, making it easier for developers to write, read, and maintain code. Swift supports advanced programming concepts such as optionals, closures, generics, and type inference, which make it powerful yet accessible.

Code :

Hello World Program in Swift:

```
import Foundation
print("Hello, World!")
```

Hello World Program in Objective-C:

```
#import <Foundation/Foundation.h>
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSLog(@"Hello, World!");
    }
    return 0;
}
```

Feature	Swift	Objective-C
Syntax	Concise, modern, user-friendly	Verbose, influenced by Smalltalk
Typing	Strongly typed with type inference	Dynamic typing, more flexibility
Memory Management	ARC enabled by default	ARC, manual management in some cases

Error Handling	Uses try, catch, throw	Uses NSError , manual checking
Interoperability	Interoperable with Objective-C	Interoperable with C and C++
Optionals	Supports optionals for safer code	No native support for optionals
Closures	Simple, concise syntax	Verbose block syntax
Performance	Generally faster with optimizations	Slightly slower in some cases
Syntax Style	Dot syntax, modern, consistent	Bracketed syntax, unique to Obj-C
Functional Programming	Strong support for functional paradigms	Limited support, less functional syntax

Conclusion:

Both Swift and Objective-C are powerful programming languages for iOS and macOS development, but they differ significantly in syntax, performance, and modern features. Swift, with its concise syntax, strong typing, and built-in error handling, is generally favored for new iOS projects due to its improved readability and safety features. On the other hand, Objective-C, while more verbose and reliant on manual memory management, continues to be relevant due to its deep integration with legacy codebases and extensive interoperability with C and C++.