# ICT4153

# Mobile Application Development

# *Practical 07*

## Flutter Layouts

**Objective**

Learn and practice different layout techniques in Flutter by implementing various UI structures using Column, Row, Stack, Expanded, Flexible, and Container widgets.

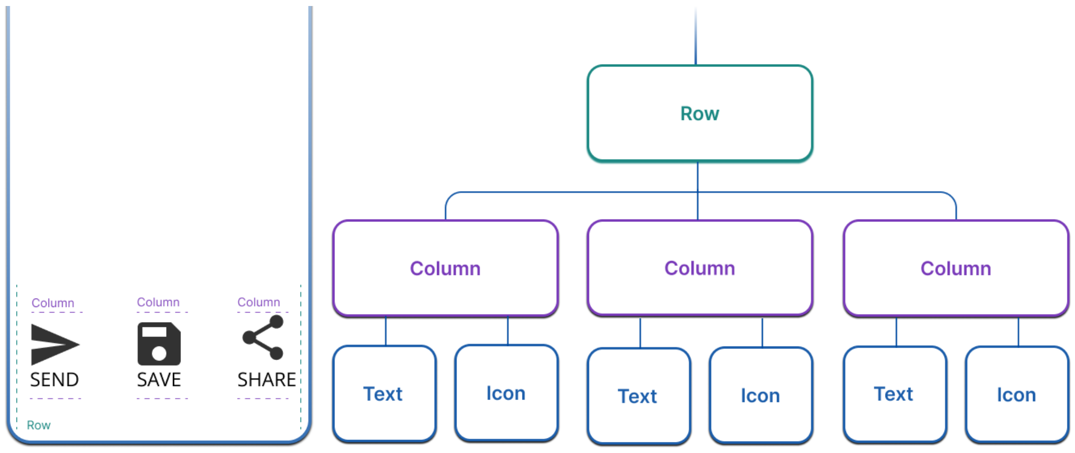
### *Layouts*

The core of Flutter's layout mechanism is widgets. In Flutter, almost everything is a widget — even layout models are widgets. The images, icons, and text that you see in a Flutter app are all widgets. Things you don't see are also widgets, such as the rows, columns, and grids that arrange, constrain, and align the visible widgets.

Flutter uses a widget-based layout system, where each UI component is a widget. The core principle of Flutter's layout system is composition—widgets are combined to create complex UI structures. There are two main types of widgets used in layout design:

- Single-child widgets - Wraps only one child widget (e.g., Container, Center, Padding).
- Multi-child widgets - Can contain multiple child widgets (e.g., Column, Row, Stack).

You create a layout by composing widgets to build more complex widgets. For example, the diagram below shows 3 icons with a label under each one, and the corresponding widget tree:



## Basic Column and Row Layout

Create a simple UI using `Column` and `Row`.

```
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 and Row Layout')),
        body: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Row(
              mainAxisAlignment: MainAxisAlignment.spaceEvenly,
              children: [
                Container(width: 50, height: 50, color: Colors.red),
                Container(width: 50, height: 50, color: Colors.green),
                Container(width: 50, height: 50, color: Colors.blue),
              ],
            ),
            SizedBox(height: 20),
            Row(
              mainAxisAlignment: MainAxisAlignment.center,
              children: [
                Text('Row with Text Widgets', style: TextStyle(fontSize: 20)),
              ],
            ),
          ],
        ),
      ),
    );
  }
}
```

**Stack Widget**

Create a stacked UI using Stack and Positioned widgets.

```
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 Widget')),
        body: Center(
          child: Stack(
            alignment: Alignment.center,
            children: [
              Container(width: 200, height: 200, color: Colors.blue),
              Positioned(bottom: 10, right: 10, child: Icon(Icons.star, size: 50, color: 
Colors.yellow)),
              Positioned(top: 10, left: 10, child: Icon(Icons.favorite, size: 50, color: Colors.red)),
            ],
          ),
        ),
      ),
    );
  }
}
```

## Using Expanded and Flexible

Implement Expanded and Flexible widgets inside a Row.

```
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('Expanded & Flexible')),
        body: Row(
          children: [
            Expanded(
              child: Container(height: 100, color: Colors.red),
            ),
            Flexible(
              child: Container(height: 100, color: Colors.green),
            ),
            Expanded(
              flex: 2,
              child: Container(height: 100, color: Colors.blue),
            ),
          ],
        ),
      ),
    );
  }
}
```

## GridView Layout

Create a GridView layout with 2 columns.

```
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('GridView Example')),
        body: GridView.count(
          crossAxisCount: 2,
          children: List.generate(6, (index) {
            return Container(
              margin: EdgeInsets.all(10),
              color: Colors.primaries[index % Colors.primaries.length],
              child: Center(
                child: Text('Item $index', style: TextStyle(color: Colors.white, fontSize: 20)),
              ),
            );
          }),
        ),
      ),
    );
  }
}
```

## ListView Layout

Create a simple ListView with text items.

```
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('ListView Example')),
        body: ListView(
          children: List.generate(10, (index) {
            return ListTile(
              leading: Icon(Icons.label),
              title: Text('Item $index'),
            );
          }),
        ),
      ),
    );
  }
}
```

Tasks to perform:

1. Create a layout with a `Column` containing two `Row` widgets. Each `Row` should contain three colored `Container` widgets.
2. Create a profile card using the `Card` and `ListTile` widgets.
3. Implement a `ListView.builder` to display a list of items dynamically.
4. Display a `GridView` with image tiles.
5. Implement a draggable `Container` that can be dragged to a `DragTarget`.
6. Create a two-screen app where the first screen contains a button to navigate to the second screen.
7. Create an example app using following scenario:

A startup named "QuickBite" is launching a food delivery service. They need a simple, well-structured Flutter mobile application for their customers. Your task is to design and implement the UI of the application using Flutter widgets and layouts. The app should have the following screens:

**Home Screen: Displays the app name and a list of food items available for order.**

**Food Details Screen: Shows detailed information about a selected food item.**

**Cart Screen: Displays selected food items and the total price.**

**Profile Screen: Contains user details such as name, email, and address.**

***Requirements:***

Use Column, Row, Stack, GridView, and ListView widgets effectively.

Implement navigation between screens using Navigator.push().

Apply Card, ListTile, Image, and Text widgets to present content.

Make the UI responsive using Expanded and Flexible widgets.

Add an interactive button to add items to the cart.

# Flutter State Management

***Introduction***

State management is a crucial concept in Flutter, enabling applications to dynamically update their UI in response to changes in data. This tutorial will explore various state management approaches in Flutter, explaining their use cases and providing practical examples.

## 1. Understanding State in Flutter

State represents data that can change over time within a Flutter application. Flutter has two types of state:

1. **Ephemeral State (Local State):**
   - Managed within a single widget.
   - Suitable for UI-related changes (e.g., toggling a switch, incrementing a counter).
2. **App-wide State:**
   - Shared across multiple widgets.
   - Suitable for complex scenarios like authentication, user preferences, and global settings.

## 2. Managing State with setState

The simplest way to manage state in Flutter is by using the `setState` method within a `StatefulWidget`.

Example: Counter App Using `setState`

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: CounterScreen(),
    );
  }
}
class CounterScreen extends StatefulWidget {
  @override
  _CounterScreenState createState() => _CounterScreenState();
}
class _CounterScreenState extends State<CounterScreen> {
  int _counter = 0;
  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Counter App')),
      body: Center(
        child: Text('Counter: $_counter', style: TextStyle(fontSize: 24)),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

*When to Use setState?*

- Best for small-scale applications.
- Suitable for managing state within a single widget.
- Avoid using setState for complex applications with multiple screens.

### 3. Using Provider for State Management

`Provider` is a popular and efficient state management solution that allows sharing state across the widget tree.

**Example: State Management Using Provider**

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

void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CounterModel(),
      child: MyApp(),
    ),
  );
}

class CounterModel extends ChangeNotifier {
  int _counter = 0;

  int get counter => _counter;

  void increment() {
    _counter++;
    notifyListeners();
  }
}

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

class CounterScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final counterModel = Provider.of<CounterModel>(context);

    return Scaffold(
      appBar: AppBar(title: Text('Provider Example')),
      body: Center(
        child: Text('Counter: ${counterModel.counter}', style: TextStyle(fontSize: 24)),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: counterModel.increment,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

*Why Use Provider?*

- Efficient and optimized for performance.
- Reduces boilerplate code compared to `setState`.
- Helps manage app-wide state effectively.

## 4. Other State Management Approaches

Apart from `setState` and `Provider`, Flutter supports several other state management solutions:

1. **Riverpod:**
   - An improved version of `Provider` with a simpler API.
   - Supports dependency injection and reactive programming.
2. **Bloc (Business Logic Component):**
   - Based on Streams and reactive programming.
   - Ideal for complex applications requiring structured state management.
3. **GetX:**
   - Lightweight and easy to use.
   - Provides reactive state management, navigation, and dependency injection.
4. **Redux:**
   - Follows unidirectional data flow.
   - Used in large-scale applications needing predictable state management.

## 5. Choosing the Right State Management Approach

| State Management Approach | Use Case |
|---|---|
| `setState` | Simple local state management within a single widget |
| `Provider` | Medium-scale apps requiring better state sharing |
| `Riverpod` | Advanced version of Provider, useful for dependency injection |
| `Bloc` | Large-scale apps needing structured and reactive state management |
| `GetX` | Lightweight, reactive approach with minimal boilerplate |
| `Redux` | Predictable state management for enterprise-level apps |