Chapter 3 : Introduction to Layouts

Objectives

- Type of Layout Widgets
  - Single Child Widgets
  - Multiple Child Widgets
- Advanced Layout Application
- Introduction to Gestures
- Statement Management in Flutter
  - Ephemeral State Management
  - Application State
- Navigation and Routing
- Scoped model

Materials

Type of Layout Widgets

1. Single Child Widgets
2. Multiple Child Widgets

1. Single Child Widgets

Single Child Widgets are those that can contain only one child. These widgets are used to modify or control the behavior of a single widget.

Common Single Child Widgets:

- `Container`:
  - A versatile widget that can be used for layout, styling, and positioning.
  - Can have padding, margin, borders, and background colors.

```
Container(
  padding: EdgeInsets.all(16.0),
  margin: EdgeInsets.all(8.0),
  color: Colors.blue,
  child: Text('Single Child Widget'),
);
```

- `Padding`:
  - Adds padding around a child widget.

```
Padding(
  padding: EdgeInsets.all(16.0),
  child: Text('Padded Text'),
);
```

- `Align`:
  - Aligns a child widget within its parent.

```
Align(
  alignment: Alignment.center,
  child: Text('Centered Text'),
);
```

- `Center`:
  - Centers its child widget within the parent.

```
Center(
  child: Text('Centered Text'),
);
```

- `SizedBox`:
  - Provides a box with a specific size.

```
SizedBox(
  width: 100,
  height: 100,
  child: Text('Sized Box'),
);
```

- `Expanded`:
  - Expands a child widget to fill available space in the parent.

```
Expanded(
  child: Text('Expanded Text'),
);
```

2. Multiple Child Widgets

Multiple Child Widgets can contain more than one child. These widgets are often used to create complex layouts by arranging multiple widgets in a specific order or alignment.

Common Multiple Child Widgets:

- `Column`:
  - Arranges its children in a vertical direction.

```
Column(
  children: <Widget>[
    Text('First'),
    Text('Second'),
    Text('Third'),
  ],
);
```

- `Row`:

- Arranges its children in a horizontal direction.

```
Row(
  children: <Widget>[
    Icon(Icons.star),
    Icon(Icons.star),
    Icon(Icons.star),
  ],
);
```

- `Stack`:
  - Places its children on top of each other, allowing for overlap.

```
Stack(
  children: <Widget>[
    Container(
      width: 100,
      height: 100,
      color: Colors.red,
    ),
    Positioned(
      top: 20,
      left: 20,
      child: Container(
        width: 50,
        height: 50,
        color: Colors.blue,
      ),
    ),
  ],
);
```

- `ListView`:
  - A scrollable list of widgets arranged in a linear fashion (either vertical or horizontal).

```
ListView(
  children: <Widget>[
    ListTile(title: Text('Item 1')),
    ListTile(title: Text('Item 2')),
    ListTile(title: Text('Item 3')),
  ],
);
```

- `GridView`:
  - A scrollable, 2D array of widgets arranged in rows and columns.

```
GridView.count(
  crossAxisCount: 2,
  children: <Widget>[
    Container(color: Colors.red, height: 100),
    Container(color: Colors.blue, height: 100),
    Container(color: Colors.green, height: 100),
    Container(color: Colors.yellow, height: 100),
  ],
);
```

- `Wrap`:
  - Wraps its children into a vertical or horizontal flow, breaking lines as necessary.

```
Wrap(
  spacing: 8.0,
  runSpacing: 4.0,
  children: <Widget>[
    Chip(label: Text('Chip 1')),
    Chip(label: Text('Chip 2')),
    Chip(label: Text('Chip 3')),
  ],
);
```

- `Flex`:

- A widget that arranges its children in a one-dimensional array (similar to `Row` and `Column` but more customizable).

```
Flex(
  direction: Axis.horizontal,
  children: <Widget>[
    Expanded(child: Container(color: Colors.red, height:
50)),
    Expanded(child: Container(color: Colors.blue, heigh
t: 50)),
  ],
);
```

Conclusion:

- **Single Child Widgets** are used when you need to modify or layout a single widget.
- **Multiple Child Widgets** are used for more complex layouts that involve arranging multiple widgets in different patterns or directions.

Advanced Layout Application

In this section, let us learn how to create a complex user interface of product listing with

custom design using both **single** and **multiple child layout widgets**.

For this purpose, follow the sequence given below:

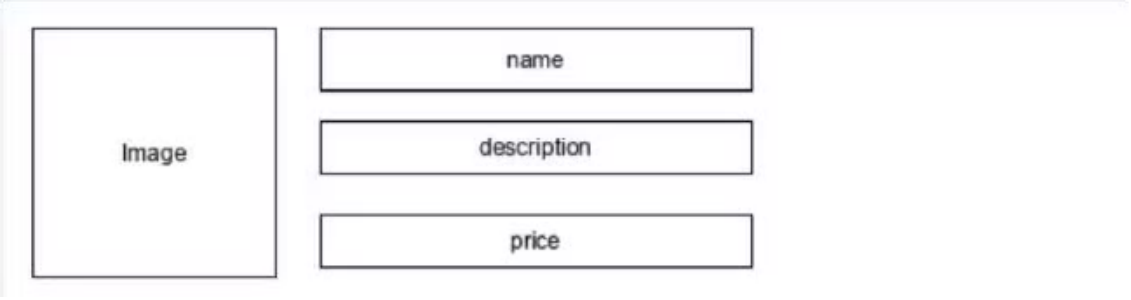
- Create a new Flutter application in VS Code, **store_app**.
- Replace the **main.dart** code with following code:

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

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

class MyApp extends StatelessWidget {

@override  Widget build(BuildContext context) {
return MaterialApp(
title: 'Flutter Demo',
theme: ThemeData(primarySwatch: Colors.blue,),
home: MyHomePage(title: 'Product layout demo home page'),
);
}
}
```

- Here, We have created **MyHomePage** widget by **extending StatelessWidget** instead of default **StatefulWidget** and then removed the relevant code.
- Now, create a new widget, **ProductBox** according to the specified design as shown

below:



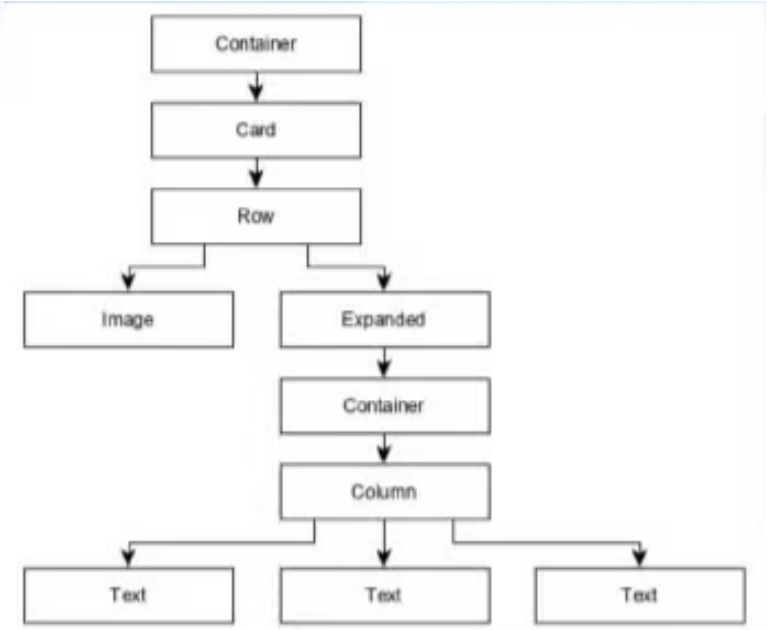
The code for the ProductBox is as follows:

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

class ProductBox extends StatelessWidget {
  const ProductBox(
      {super.key,
      this.name = '',
```

```
      this.description = '',
      this.price = 0,
      this.image = ''});

  final String name;

  final String description;

  final int price;

  final String image;

  @override
  Widget build(BuildContext context) {
    return Container(
      padding: const EdgeInsets.all(2),
      height: 120,
      child: Card(
        child: Row(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: <Widget>[
              Image.asset("assets/appimages/ + $image"),
              Expanded(
                child: Container(
                  padding: const EdgeInsets.all(5),
                  child: Column(
                    mainAxisAlignment: MainAxisAlignment.spac
                    children: <Widget>[
                      Text(name,
                          style: const TextStyle(fontWeight:
                      Text(description),
                      Text("Price: $price"),
                    ],
                  ),
                ),
              )
            ]),
      ),
```

```
    );
  }
}
```

- Please observe the following in the code:
  - **ProductBox** has used four arguments as specified below:
    - **name** - Product name
    - **description** - Product description
    - **price** - Price of the product
    - **image** - Image of the product
  - **ProductBox** uses seven build-in widgets as specified below:
    - **Container**
    - **Expanded**
    - **Row**
    - **Column**
    - **Card**
    - **Text**
- **ProductBox** is designed using the above mentioned widget. The arrangement or hierarchy of the widget is specified in the diagram shown below:
- Now, place some dummy image (see below) for product information in the assets folder of the application and configure the assets folder in the **pubspec.yaml** file as shown below:



Definition images in **pubspec.yaml**

- Finally, Use the **ProductBox** widget in the **MyHomePage** widget as specified below:

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

class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key, this.title = ''});

  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("Product Listing")),
      body: ListView(
        shrinkWrap: true,
        padding: const EdgeInsets.fromLTRB(2.0, 10.0, 2.0,
10.0),
        children: <Widget>[
          ProductBox(
              name: "iPhone",
```

```
                description: "iPhone is the stylist phone eve
r",
                price: 1000,
                image: "iPhone.jpg"),
            ProductBox(
                name: "Pixel",
                description: "Pixel is the most featureful ph
one ever",
                price: 800,
                image: "pixel.png"),
            ProductBox(
                name: "Laptop",
                description: "Laptop is most productive devel
opment tool",
                price: 2000,
                image: "laptop.jpg"),
            ProductBox(
                name: "Tablet",
                description: "Tablet is the most useful devic
e ever for meeting",
                price: 1500,
                image: "tablet.jpg"),
            ProductBox(
                name: "Floppy Drive",
                description: "Floppy drive is useful rescue s
torage medium",
                price: 20,
                image: "floppy.jpg"),
          ],
        ),
      );
    }
}
```

Here,we have used **ProductBox** as children of **ListView** widget.

- The complete code (**main.dart**) of the product layout application (**store_app**) is as follows:

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

class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key, this.title = ''});
  final String title;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: const Text("Product Listing")),
        body: ListView(
          shrinkWrap: true,
          padding: const EdgeInsets.fromLTRB(2.0, 10.0, 2.0,
          children: const <Widget>[
            ProductBox(
                name: "iPhone",
                description: "iPhone is the stylist phone eve
                price: 1000,
                image: "iPhone.jpg"),
            ProductBox(
                name: "Pixel",
                description: "Pixel is the most featureful ph
                price: 800,
                image: "pixel.png"),
            ProductBox(
                name: "Laptop",
                description: "Laptop is most productive devel
                price: 2000,
                image: "laptop.jpg"),
            ProductBox(
                name: "Tablet",
                description:
                    "Tablet is the most useful device ever fo
                price: 1500,
                image: "tablet.jpg"),
            ProductBox(
                name: "Floppy Drive",
                description: "Floppy drive is useful rescue s
                price: 20,
```

```
                image: "floppy.jpg"),
          ],
        ));
  }
}

class ProductBox extends StatelessWidget {
  const ProductBox(
      {super.key,
      this.name = '',
      this.description = '',
      this.price = 0,
      this.image = ''});
  final String name;
  final String description;
  final int price;
  final String image;
  @override
  Widget build(BuildContext context) {
    return Container(
        padding: const EdgeInsets.all(2),
        height: 120,
        child: Card(
            child: Row(
                mainAxisAlignment: MainAxisAlignment.spaceEve
                children: <Widget>[
              Image.asset("assets/appimages/$image"),
              Expanded(
                  child: Container(
                      padding: const EdgeInsets.all(5),
                      child: Column(
                        mainAxisAlignment: MainAxisAlignment.
                        children: <Widget>[
                          Text(name,
                              style:
                                  const TextStyle(fontWeight:
                          Text(description),
                          Text("Price: $price"),
```

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

Note: Below 5 Types Are Extra Just For Learning No Need To Prepare For Exam.

1. Responsive Layouts

MediaQuery

- `MediaQuery` is a widget that provides information about the size and orientation of the screen. It’s useful for building layouts that adapt to different screen sizes.

```
dartCopy code
@override
Widget build(BuildContext context) {
  var screenWidth = MediaQuery.of(context).size.width;
  var isLandscape = MediaQuery.of(context).orientation =
= Orientation.landscape;

  return Scaffold(
    body: Center(
      child: isLandscape
          ? Row(
              children: [Text('Landscape Mode')],
            )
          : Column(
              children: [Text('Portrait Mode')],
            ),
    ),
  );
}
```

LayoutBuilder

- `LayoutBuilder` is a widget that helps you build a layout depending on the parent widget's size. It provides constraints that can be used to adapt the UI accordingly.

```
dartCopy code
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: LayoutBuilder(
      builder: (BuildContext context, BoxConstraints con
straints) {
        if (constraints.maxWidth > 600) {
          return Row(
            children: [
              Expanded(child: Container(color: Colors.re
```

```
d)),
              Expanded(child: Container(color: Colors.bl
ue)),
            ],
          );
        } else {
          return Column(
            children: [
              Expanded(child: Container(color: Colors.re
d)),
              Expanded(child: Container(color: Colors.bl
ue)),
            ],
          );
        }
      },
    ),
  );
}
```

2. Slivers

Slivers are advanced scrollable widgets that offer highly customizable scroll behaviors and layouts.

SliverAppBar

- A `SliverAppBar` is a material design app bar that integrates with custom scroll views. It can scroll with the content or stay pinned at the top.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: CustomScrollView(
      slivers: <Widget>[
        SliverAppBar(
```

```
          expandedHeight: 200.0,
          flexibleSpace: FlexibleSpaceBar(
            title: Text('SliverAppBar'),
            background: Image.network(
              'https://flutter.dev/assets/homepage/carou
sel/slide_1-layer_0-1a111fa8b65e018f8987c4205c84a4a8.pn
g',
              fit: BoxFit.cover,
            ),
          ),
          floating: false,
          pinned: true,
        ),
        SliverList(
          delegate: SliverChildBuilderDelegate(
            (context, index) => ListTile(title: Text('It
em #$index')),
            childCount: 50,
          ),
        ),
      ],
    ),
  );
}
```

SliverList and SliverGrid

- `SliverList` and `SliverGrid` allow for more customized scrolling layouts, which can be integrated into a `CustomScrollView`.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: CustomScrollView(
      slivers: <Widget>[
        SliverList(
```

```
          delegate: SliverChildBuilderDelegate(
            (context, index) => ListTile(title: Text('It
em #$index')),
            childCount: 20,
          ),
        ),
        SliverGrid(
          delegate: SliverChildBuilderDelegate(
            (context, index) => Container(
              color: index.isEven ? Colors.blue : Color
s.green,
            ),
            childCount: 20,
          ),
          gridDelegate: SliverGridDelegateWithFixedCross
AxisCount(
            crossAxisCount: 3,
          ),
        ),
      ],
    ),
  );
}
```

3. Advanced Grid Layouts

GridView with Custom Layouts

- `GridView` can be customized extensively to create complex grid layouts.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: GridView.custom(
      gridDelegate: SliverGridDelegateWithMaxCrossAxisEx
```

```
tent(
        maxCrossAxisExtent: 200,
        mainAxisSpacing: 10,
        crossAxisSpacing: 10,
        childAspectRatio: 3 / 2,
      ),
      childrenDelegate: SliverChildBuilderDelegate(
        (context, index) {
          return Container(
            color: index.isEven ? Colors.red : Colors.bl
ue,
          );
        },
        childCount: 30,
      ),
    ),
  );
}
```

4. Animated Layouts

AnimatedContainer

- `AnimatedContainer` allows you to create smooth transitions between different layouts.

```
class AnimatedLayoutExample extends StatefulWidget {
  @override
  _AnimatedLayoutExampleState createState() => _Animated
LayoutExampleState();
}

class _AnimatedLayoutExampleState extends State<Animated
LayoutExample> {
  bool _isExpanded = false;
```

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: GestureDetector(
          onTap: () {
            setState(() {
              _isExpanded = !_isExpanded;
            });
          },
          child: AnimatedContainer(
            width: _isExpanded ? 200 : 100,
            height: _isExpanded ? 200 : 100,
            color: _isExpanded ? Colors.blue : Colors.re
d,
            alignment: _isExpanded ? Alignment.center :
Alignment.topLeft,
            duration: Duration(seconds: 1),
            curve: Curves.easeInOut,
            child: Text('Tap to Expand'),
          ),
        ),
      ),
    );
  }
}
```

Introduction To Gestures

In Flutter, gestures are a way to detect and respond to user interactions like taps, swipes, and drags. Flutter provides a robust system to handle gestures, making it easy to create interactive and responsive user interfaces. The primary widget used for handling gestures in Flutter is the `GestureDetector`.

Key Concepts of Gestures

1. **GestureDetector**:
   - The `GestureDetector` widget is used to capture a wide range of gestures. It wraps around another widget and intercepts gestures like taps, double taps, long presses, swipes, and more.
2. **Common Gestures**:
   - **Tap**: A single quick press.
   - **Double Tap**: Two quick presses in succession.
   - **Long Press**: A press that lasts for more than a second.
   - **Swipe**: A quick drag motion in a particular direction.
   - **Drag**: A press followed by movement.
3. **Using GestureDetector**:
   - The `GestureDetector` widget has properties like `onTap`, `onDoubleTap`, `onLongPress`, `onPanUpdate`, and others, each corresponding to a specific gesture.

Example of Using GestureDetector

Let's create a simple example where a container changes its color when the user taps on it:

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

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

class GestureExample extends StatefulWidget {
  @override
  _GestureExampleState createState() => _GestureExampleStat
e();
}

class _GestureExampleState extends State<GestureExample> {
```

```
  Color _color = Colors.blue;

  void _changeColor() {
    setState(() {
      _color = _color == Colors.blue ? Colors.red : Colors.
blue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Gesture Example'),
        ),
        body: Center(
          child: GestureDetector(
            onTap: _changeColor,
            child: Container(
              width: 200,
              height: 200,
              color: _color,
              child: Center(
                child: Text(
                  'Tap me',
                  style: TextStyle(color: Colors.white, fon
tSize: 24),
                ),
              ),
            ),
          ),
        ),
      ),
    );
  }
}
```

Explanation:

- The `GestureDetector` wraps a `Container` widget.
- The `onTap` callback is used to change the color of the container when the user taps on it.
- The `_changeColor` method is called when the container is tapped, toggling its color between blue and red.

Advanced Gestures

Beyond simple taps and presses, Flutter allows you to detect more complex gestures:

1. **Drag Gesture**:
   - You can detect dragging using the `onPanUpdate` callback. This is useful for interactive elements like sliders or custom scroll views.

```
dartCopy code
GestureDetector(
  onPanUpdate: (details) {
    // Handle drag
  },
  child: Container(
    color: Colors.green,
    width: 200,
    height: 200,
  ),
);
```

2. **Swipe Gesture**:
   - Detecting swipes involves using `onHorizontalDragEnd` or `onVerticalDragEnd` callbacks. These are triggered when the user swipes in a particular direction.

```
dartCopy code
GestureDetector(
  onHorizontalDragEnd: (details) {
    // Handle horizontal swipe
```

```
  },
  child: Container(
    color: Colors.orange,
    width: 200,
    height: 200,
  ),
);
```

3. **Pinch and Zoom**:
   - For more advanced interactions like pinch-to-zoom, you would typically use the `onScaleUpdate` callback.

```
dartCopy code
GestureDetector(
  onScaleUpdate: (details) {
    // Handle pinch-to-zoom
  },
  child: Container(
    color: Colors.purple,
    width: 200,
    height: 200,
  ),
);
```

State Management in Flutter

State management is a critical concept in Flutter that involves managing the state, or the data that your UI depends on. In Flutter, state refers to any data that can change over time, such as user input, the result of an asynchronous operation, or even the appearance of a widget.

Flutter provides various ways to manage state, depending on the complexity and scope of the application. These methods can generally be categorised into **Ephemeral State Management** and **Application State Management**.

1. Ephemeral State Management

Ephemeral state (also known as UI state or local state) is state that is local to a single widget. This type of state is usually simple and short-lived, like whether a checkbox is checked or the current page of a `PageView`. Ephemeral state can be managed directly within the widget itself using the `State` class.

Example of Ephemeral State Management

```
dartCopy code
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('Ephemeral State Example'),
        ),
        body: CounterWidget(),
      ),
    );
  }
}

class CounterWidget extends StatefulWidget {
  @override
  _CounterWidgetState createState() => _CounterWidgetState
();
}

class _CounterWidgetState extends State<CounterWidget> {
  int _counter = 0;
```

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

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'You have pushed the button this many times:',
          ),
          Text(
            '$_counter',
            style: Theme.of(context).textTheme.headline4,
          ),
          ElevatedButton(
            onPressed: _incrementCounter,
            child: Text('Increment'),
          ),
        ],
      ),
    );
  }
}
```

Explanation:

- In this example, the `_counter` variable is managed within the `CounterWidget`'s state class (`_CounterWidgetState`).
- This state is ephemeral because it only affects the `CounterWidget` and is not needed elsewhere in the app.
- When the button is pressed, `setState` is called, which tells Flutter to rebuild the widget with the updated state.

2. Application State Management

Application state (also known as shared state or global state) refers to state that needs to be shared across multiple parts of the application. This kind of state can include user preferences, authentication status, or data fetched from an API that needs to be accessible throughout the app.

Managing application state can be more complex because it often involves sharing data between different parts of the app. Flutter provides several approaches to managing application state, including `InheritedWidget`, `Provider`, `Bloc`, `Riverpod`, and more.

Step 1: Define the App's State in a StatefulWidget

You can use a `StatefulWidget` to hold the application state (in this case, a counter) and manage its changes.

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

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

class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  int _counter = 0;

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'App State Management',
```

```
      theme: ThemeData(primarySwatch: Colors.blue),
      home: HomeScreen(
        counter: _counter,
        incrementCounter: _incrementCounter,
      ),
    );
  }
}
```

- Here, `_counter` is our application-wide state. We have a function `_incrementCounter()` that updates the state.
- `setState()` is used to rebuild the widget tree when the state changes.

Step 2: Create the HomeScreen and Pass the State

The `HomeScreen` widget will display the counter value and allow navigation to a second screen where the counter can be updated.

```
class HomeScreen extends StatelessWidget {
  final int counter;
  final VoidCallback incrementCounter;

  HomeScreen({required this.counter, required this.incremen
tCounter});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Home Screen'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'Counter value:',
```

```
              style: TextStyle(fontSize: 20),
            ),
            Text(
              '$counter',
              style: Theme.of(context).textTheme.headline4,
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: () {
                Navigator.push(
                  context,
                  MaterialPageRoute(
                    builder: (context) => SecondScreen(
                      counter: counter,
                      incrementCounter: incrementCounter,
                    ),
                  ),
                );
              },
              child: Text('Go to Second Screen'),
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

- The `HomeScreen` takes the current counter value and the `incrementCounter` function as parameters.
- The button navigates to the `SecondScreen` while passing the same state and callback function to allow state management across screens.

Step 3: Create the SecondScreen and Access the Shared State

The `SecondScreen` will display the counter value and allow the user to increment the counter.

```
class SecondScreen extends StatelessWidget {
  final int counter;
  final VoidCallback incrementCounter;

  SecondScreen({required this.counter, required this.increm
entCounter});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Second Screen'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'Counter value:',
              style: TextStyle(fontSize: 20),
            ),
            Text(
              '$counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
```

```
      );
    }
  }
```

Explanation:

- `CounterModel` is a `ChangeNotifier` that holds the application state (in this case, the counter value).
- The `ChangeNotifierProvider` makes `CounterModel` available to the entire widget tree.
- `CounterWidget` listens to changes in `CounterModel` using `context.watch<CounterModel>()` and updates the UI accordingly.
- The `FloatingActionButton` increments the counter, demonstrating how application state can be modified and shared across different parts of the app.

Summary

In this example:

- **State is managed by a parent** `StatefulWidget` (`MyApp`).
- The state (counter value) and a function to update the state (`_incrementCounter`) are passed to multiple screens (`HomeScreen` and `SecondScreen`).
- Each screen can display and modify the same state without using any external dependencies or libraries like `Provider`.

This approach is simple and works well for small to medium-sized apps where you want to manage global or shared state without adding complexity. For larger apps, you might consider more sophisticated state management solutions.

- **Ephemeral State Management**: Suitable for simple, short-lived state that is confined to a single widget. Managed using `State` objects within the widget.

- **Application State Management**: Used for state that needs to be shared across multiple parts of the application. Managed using more complex patterns like `Provider`, `Bloc`, or other state management solutions.

Navigation and Routing in Flutter

Navigation and routing in Flutter allow you to move between different screens (or pages) in your app, creating a smooth user experience. Flutter provides a robust set of APIs to handle navigation and routing, making it easy to build complex applications with multiple screens.

Key Concepts

1. **Route**:
   - A `Route` in Flutter is an abstraction for a screen or page. When you navigate to a new screen, you are pushing a new `Route` onto the navigation stack.
2. **Navigator**:
   - The `Navigator` is a widget that manages a stack of `Route` objects. It provides methods to navigate between routes, such as `push`, `pop`, and others.
3. **Routing**:
   - Routing refers to the process of defining and handling the routes in your app. This includes defining named routes and handling dynamic routing for more complex scenarios.

Basic Navigation

1. Pushing a New Screen

To navigate to a new screen, you use the `Navigator.push` method. This method takes a `Route` and adds it to the navigation stack, displaying the new screen.

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

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

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

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

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

```
        child: ElevatedButton(
          child: Text('Go Back'),
          onPressed: () {
            Navigator.pop(context);
          },
        ),
      ),
    );
  }
}
```

Explanation:

- `Navigator.push`: This method is used to navigate to a new screen by adding a new route to the stack.
- `MaterialPageRoute`: This creates a route that uses a platform-specific animation to transition between screens.
- `Navigator.pop`: This removes the top route from the stack, returning to the previous screen.

2. Named Routes

Named routes allow you to define routes centrally and refer to them by name when navigating. This is useful for larger apps where route management becomes complex.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      // Define the named routes
      routes: {
        '/': (context) => FirstScreen(),
        '/second': (context) => SecondScreen(),
      },
```

```
      initialRoute: '/',
    );
  }
}

class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('First Screen')),
      body: Center(
        child: ElevatedButton(
          child: Text('Go to Second Screen'),
          onPressed: () {
            Navigator.pushNamed(context, '/second');
          },
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Second Screen')),
      body: Center(
        child: ElevatedButton(
          child: Text('Go Back'),
          onPressed: () {
            Navigator.pop(context);
          },
        ),
      ),
    );
  }
}
```

Explanation:

- **Named Routes**: Routes are defined in a map within the `MaterialApp` widget using the `routes` property.
- `Navigator.pushNamed`: This method is used to navigate to a route using its name, which simplifies the navigation process, especially in larger apps.

Passing Data Between Screens

You can pass data between screens when navigating by using the constructor of the screen you're navigating to or by using arguments in named routes.

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

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

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

class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('First Screen')),
      body: Center(
        child: ElevatedButton(
          child: Text('Go to Second Screen with Data'),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(
                builder: (context) => SecondScreen(data: 'H
```

```
ello from First Screen!'),
              ),
            );
          },
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  final String data;

  SecondScreen({required this.data});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Second Screen')),
      body: Center(
        child: Text(data),
      ),
    );
  }
}
```

Explanation:

- The `data` is passed to the `SecondScreen` through its constructor. This allows the `SecondScreen` to access and display the data.

Summary

- **Basic Navigation**: Use `Navigator.push` and `Navigator.pop` for simple navigation between screens.
- **Named Routes**: Define routes centrally and use `Navigator.pushNamed` for cleaner, more maintainable navigation.

- **Passing Data**: You can pass data to and return data from routes easily using constructors or arguments.
- **Advanced Navigation**: Flutter's `Navigator` and `Route` system is flexible enough to handle complex navigation patterns, including nested navigation and dynamic routing.

Navigation and routing are crucial parts of building a Flutter application, and mastering these concepts will allow you to create a smooth and intuitive user experience.

Scoped Model in Flutter

`ScopedModel` is a state management solution in Flutter that allows you to share state across your application in a way that is simple, efficient, and effective. It was popular before more modern solutions like `Provider` and `Riverpod` became mainstream, but it's still useful for understanding how state can be managed in a structured manner.

Key Concepts of Scoped Model

1. **Model**:
   - A `Model` is a class that holds the application state. It extends the `Model` class from the `scoped_model` package. Any changes to the model can trigger a rebuild of widgets that depend on it.
2. **ScopedModel**:
   - The `ScopedModel` widget is used to provide the `Model` to the widget tree. It acts as a provider of the state and can be accessed by any widget in the subtree.
3. **ScopedModelDescendant**:
   - `ScopedModelDescendant` is a widget that listens to changes in the `Model`. When the state in the `Model` changes, only those `ScopedModelDescendant` widgets that depend on the model will rebuild.

Example of Scoped Model in Flutter

Let's create a simple counter application using `ScopedModel`.

Step 1: Add the `scoped_model` Dependency

First, add the `scoped_model` package to your `pubspec.yaml` file:

```
yamlCopy code
dependencies:
  flutter:
    sdk: flutter
  scoped_model: ^1.0.1
```

Then, run `flutter pub get` to install the package.

Step 2: Define the Model

Create a `CounterModel` class that will hold the counter value and provide methods to modify it.

```
import 'package:scoped_model/scoped_model.dart';

class CounterModel extends Model {
  int _counter = 0;

  int get counter => _counter;

  void increment() {
    _counter++;
    notifyListeners(); // Notifies all listening widgets to
rebuild
  }
}
```

Step 3: Provide the Model Using `ScopedModel`

Wrap your `MaterialApp` or `Scaffold` with a `ScopedModel` to provide the model to the widget tree.

```
import 'package:flutter/material.dart';
import 'package:scoped_model/scoped_model.dart';
import 'counter_model.dart'; // Import the model

void main() {
  runApp(
    ScopedModel<CounterModel>(
      model: CounterModel(),
      child: MyApp(),
    ),
  );
}

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

Step 4: Consume the Model Using `ScopedModelDescendant`

Use `ScopedModelDescendant` to access and react to changes in the model.

```
dartCopy code
import 'package:flutter/material.dart';
import 'package:scoped_model/scoped_model.dart';
import 'counter_model.dart';

class CounterScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
```

```
        title: Text('Scoped Model Example'),
      ),
      body: Center(
        child: ScopedModelDescendant<CounterModel>(
          builder: (context, child, model) {
            return Text(
              'Counter: ${model.counter}',
              style: Theme.of(context).textTheme.headline4,
            );
          },
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          ScopedModel.of<CounterModel>(context).increment
();
        },
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Explanation of the Code

- **CounterModel**: The `CounterModel` class holds the counter value and a method to increment it. It extends `Model` and calls `notifyListeners()` whenever the counter value changes. This ensures that all widgets listening to this model get rebuilt with the updated state.
- **ScopedModel**: The `ScopedModel<CounterModel>` widget provides the `CounterModel` to the entire widget tree. Any widget within this tree can access the model and react to changes.
- **ScopedModelDescendant**: This widget is used to listen to changes in `CounterModel`. When the `CounterModel` updates, the `ScopedModelDescendant` will rebuild, displaying the new counter value.

- **FloatingActionButton**: The button triggers the `increment()` method in the `CounterModel` to increase the counter value and notify listeners to update the UI.

Advantages of Scoped Model

- **Simplicity**: It's easy to understand and use, making it suitable for small to medium-sized applications.
- **Separation of Concerns**: Helps to separate the UI from the business logic and state management.
- **Efficient Rebuilds**: Only the widgets that depend on the model are rebuilt when the state changes.

Limitations

- **Scalability**: While good for small projects, `ScopedModel` can become difficult to manage in larger applications with more complex state requirements.
- **Less Community Support**: Over time, `Provider` has become the preferred state management solution, so `ScopedModel` has less community support and fewer resources.

Conclusion

`ScopedModel` is a straightforward state management approach in Flutter, ideal for small to medium-sized applications. It provides a clear and effective way to manage and share state across your widget tree. However, for more complex applications, modern solutions like `Provider` or `Riverpod` are often preferred