Presentation 3

25 Nov 2021

OBJECT-ORIENTED PROGRAMMING I IT 411

IT DEPT.

TIU

4RD GRADE

Application Bar, List View and Build A Custom Widget

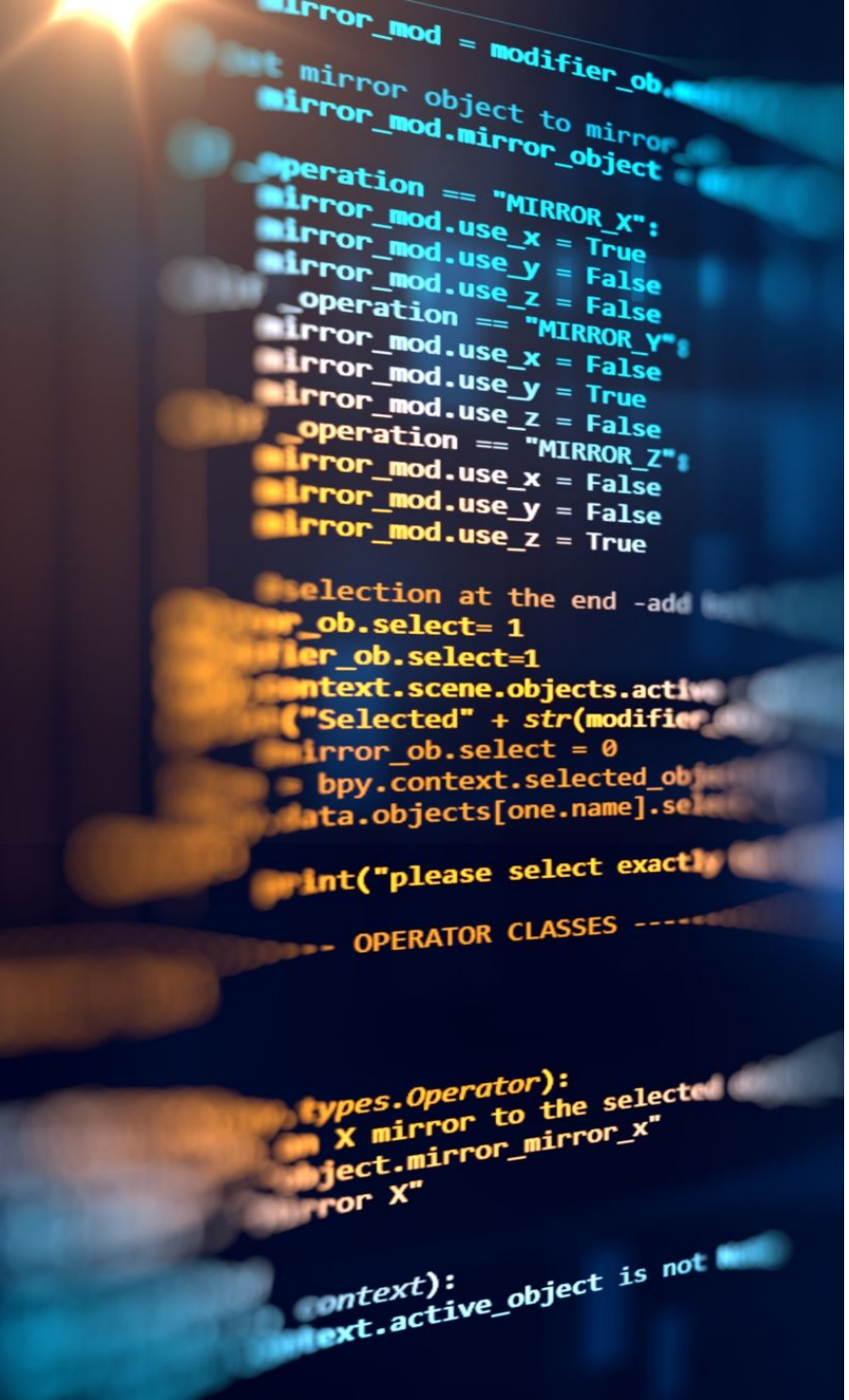
Lect. Mohammad Salim

Thses slide notes are based on many different online resources such as Flutter.dev, Flutter Apprentice Book and The complete Fluter bootcamp course

COURSE CONTENT

- Flutter and OOP

Week	Hour	Date	Topic
1	2	4-7/10/2021	Introduction to OOP , Class diagram
2	2	10-14/10/2021	Introduction to OOP , Class diagram and Dart Packages
3	2	17-21/10/2021	Section 1: Build Your First Flutter App, structure of Flutter projects, create the UI o a Flutter app by Widgets
4	2	24-28/10/2021	Section 2: Everything's a Widget, start to build a full-featured recipe app named Fooderlich
5	2	31/10-4/11/2021	Section 2: Everything's a Widget, start to build a full-featured recipe app named Fooderlich
6	2	7-11/11/2021	Understanding widgets
7	2	14-18/11/2021	Midterm Exam
8	2	21-25/11/2021	Stateless widgets and build our personal profile application (HW2)
9	2	28/11-2/12/2021	Application bar, list view and build a custom widget
10	2	5-9/12/2021	Navigation in Flutte, Stateful Widgets and building an interactive applications
11	2	12-16/12/2021	Material Design, Build for Android and iOS platforms, Colors and Themes
12	2	19-23/12/2021	Handle user input and Handle gestures and responsive design
13	2	26-30/12/2021	Flutter Packages and Plugins, Images, Icons, Fonts
14	2	2-5/1/2022	APIs and how to get data from internet
15	2	9-13/1/2022	Final Exam
16	2	16-20/1/2022	Final Exam



CONTENTS

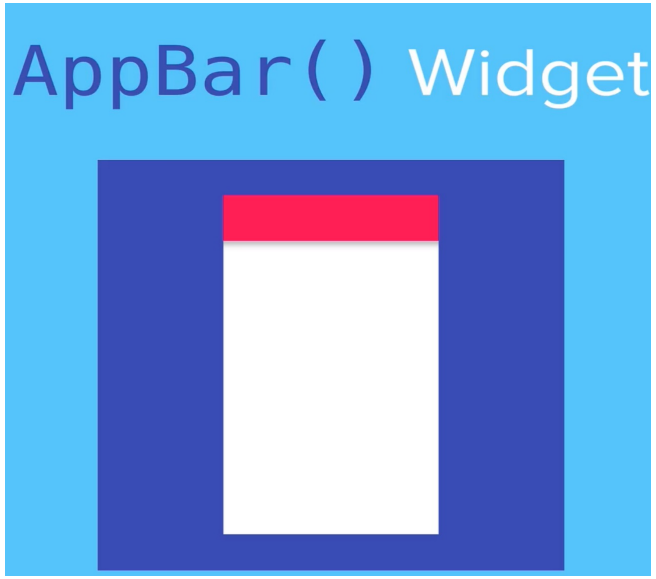
- Application bar
- List view
- Build a custom widget
- Navigation in Flutter
- Stateful Widgets and building an interactive applications

APP BAR: SCAFFOLDING A MATERIAL APP

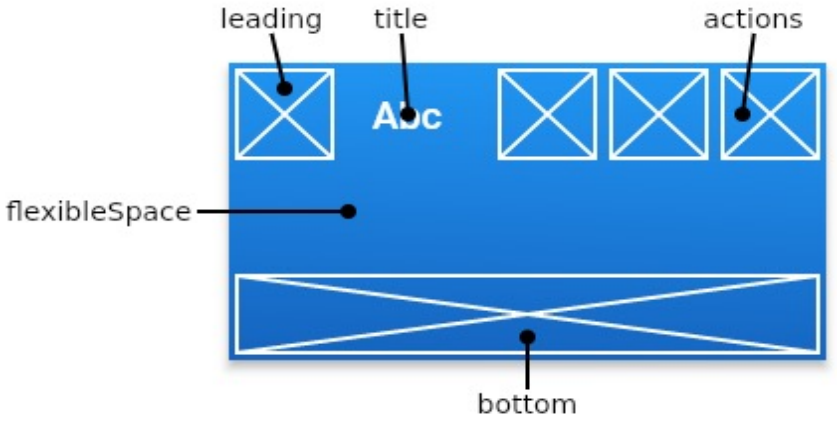
- By using **MaterialApp** widget we can get **Scaffold** widget.
- Scaffold help us to get many attributes like **appBar** and **BottomBar** and many more others.
- The **scaffold** will expand to fill the available space. That usually means that it will occupy its entire window or device screen.

APP BAR: SCAFFOLDING A MATERIAL APP

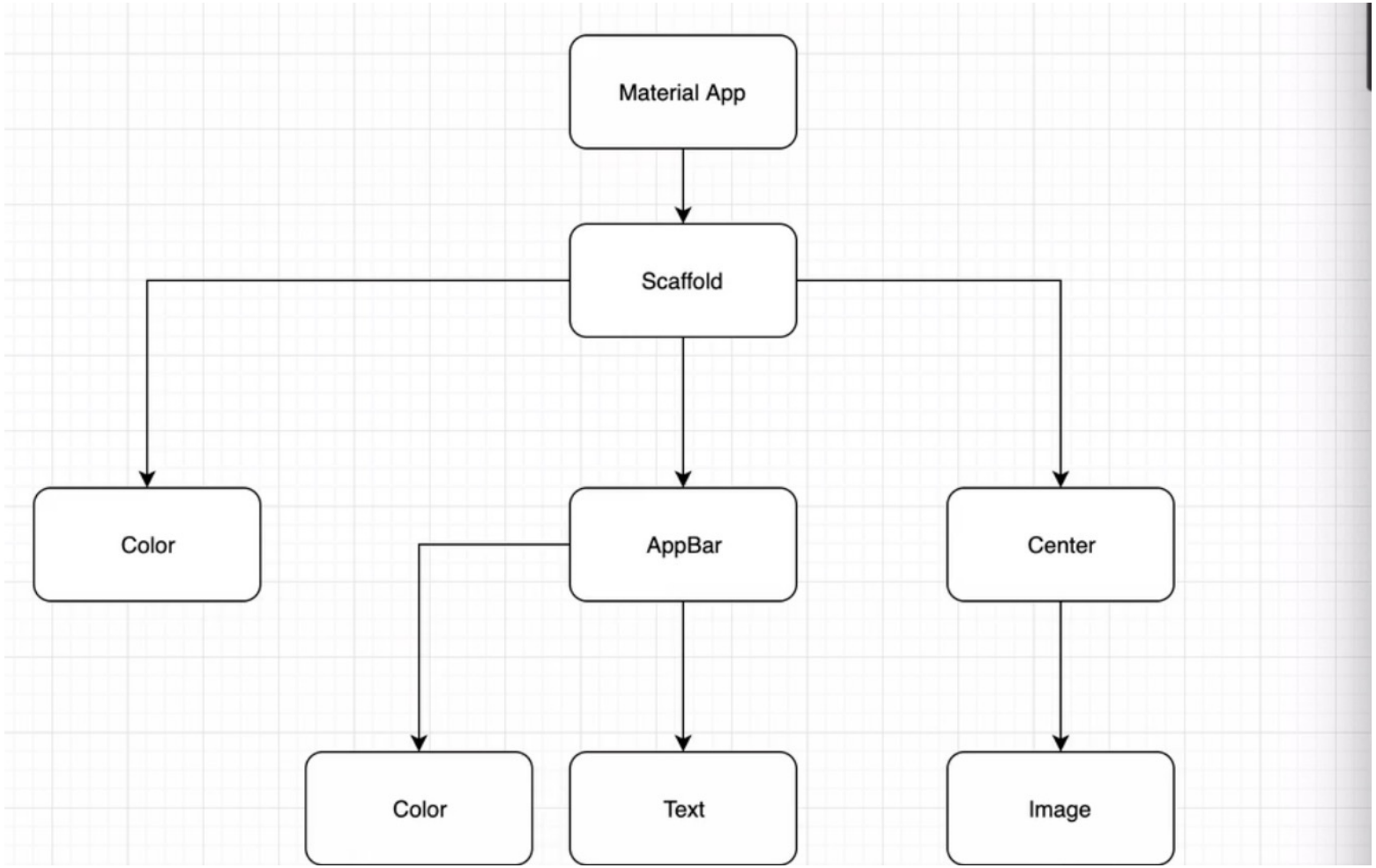
- An app bar consists of a toolbar and potentially other widgets, such as a TabBar and a FlexibleSpaceBar.
- App bars are typically used in the Scaffold.appBar property, which places the app bar as a fixed-height widget at the top of the screen.
- The AppBar displays the toolbar widgets, leading, title, and actions, above the bottom (if any). The bottom is usually used for a TabBar.

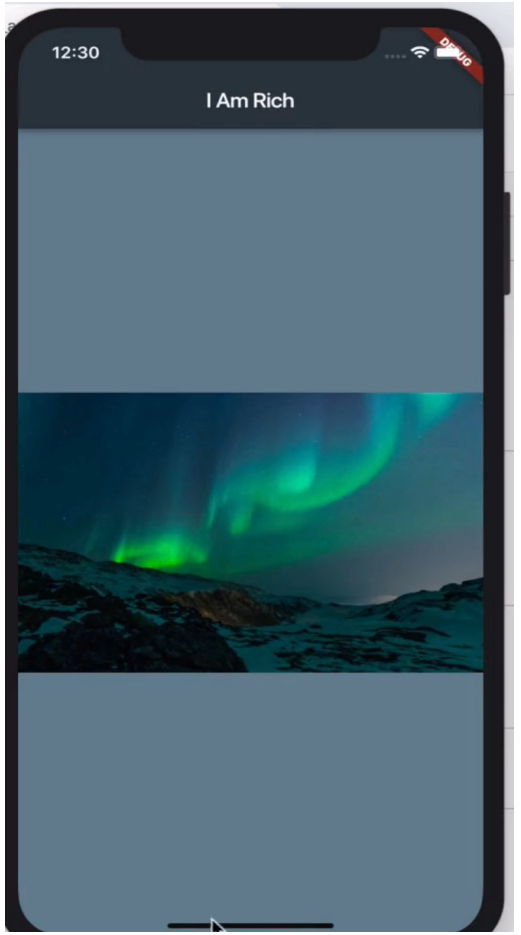



```
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('AppBar Demo'),
```



APP BAR: SCAFFOLDING A MATERIAL APP

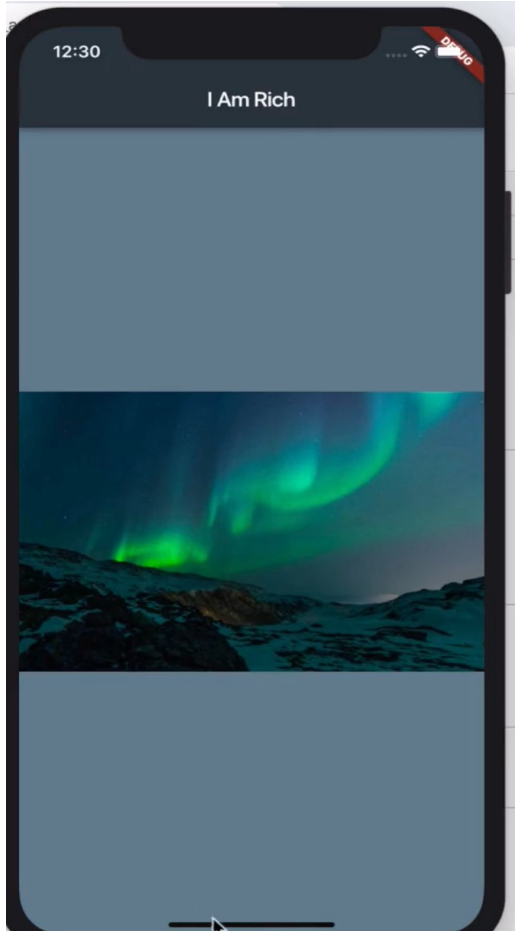




APP BAR: SCAFFOLDING A MATERIAL APP

main.dart

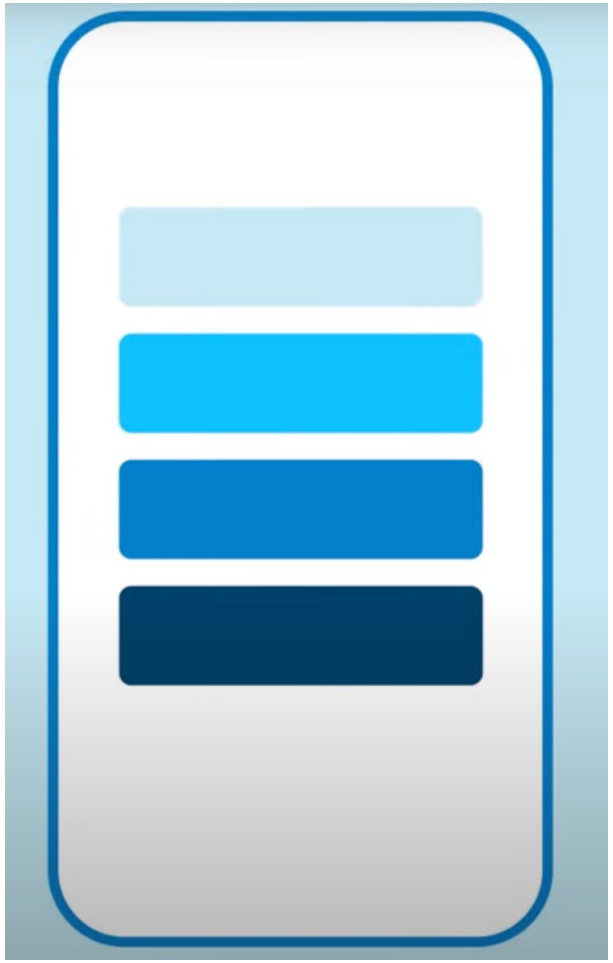
```dart
import 'package:flutter/material.dart';

//The main function is the entrance point for all our Flutter apps.
void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        backgroundColor: Colors.blueGrey,
        appBar: AppBar(
          title: Text('I Am Rich'),
          backgroundColor: Colors.blueGrey[900],
        ), // AppBar
        body: Center(
          child: Image(
            image:
                NetworkImage('https://www.w3schools.com/w3css/img_lights.jpg'),
          ), // Image
        ), // Center
      ), // Scaffold
    ), // MaterialApp
  );
}
```



LIST VIEW

- ListView is the most commonly used scrolling widget. It displays its children one after another in the scroll direction. In the cross axis, the children are required to fill the ListView.
- There are four options for constructing a ListView, however we will cover only two of them:

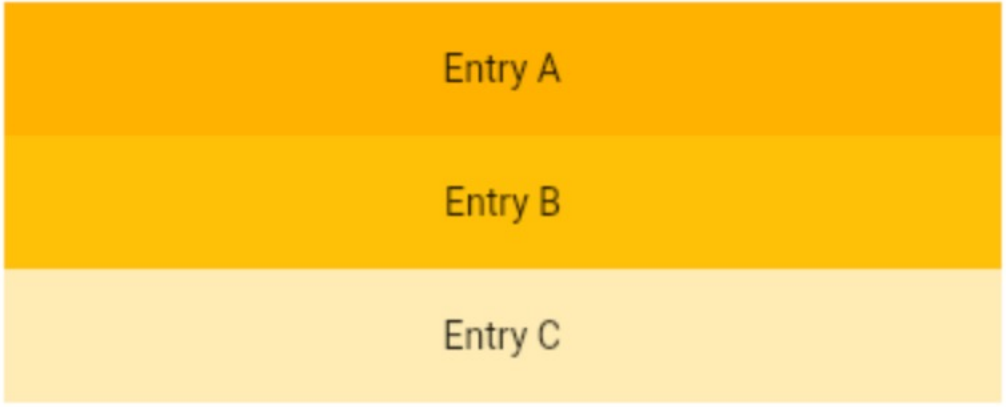
1. The default constructor takes an explicit List<Widget> of children. This constructor is appropriate for list views with a small number of children because constructing the List requires doing work for every child that could possibly be displayed in the list view instead of just those children that are actually visible.
2. The ListView.builder constructor takes an IndexedWidgetBuilder, which builds the children on demand. This constructor is appropriate for list views with a large (or infinite) number of children because the builder is called only for those children that are actually visible.



LIST VIEW

```
ListView(
  padding: const EdgeInsets.all(8),
  children: <Widget>[
    Container(
      height: 50,
      color: Colors.amber[600],
      child: const Center(child: Text('Entry A')),
    ),
    Container(
      height: 50,
      color: Colors.amber[500],
      child: const Center(child: Text('Entry B')),
    ),
    Container(
      height: 50,
      color: Colors.amber[100],
      child: const Center(child: Text('Entry C')),
    ),
  ],
)
```

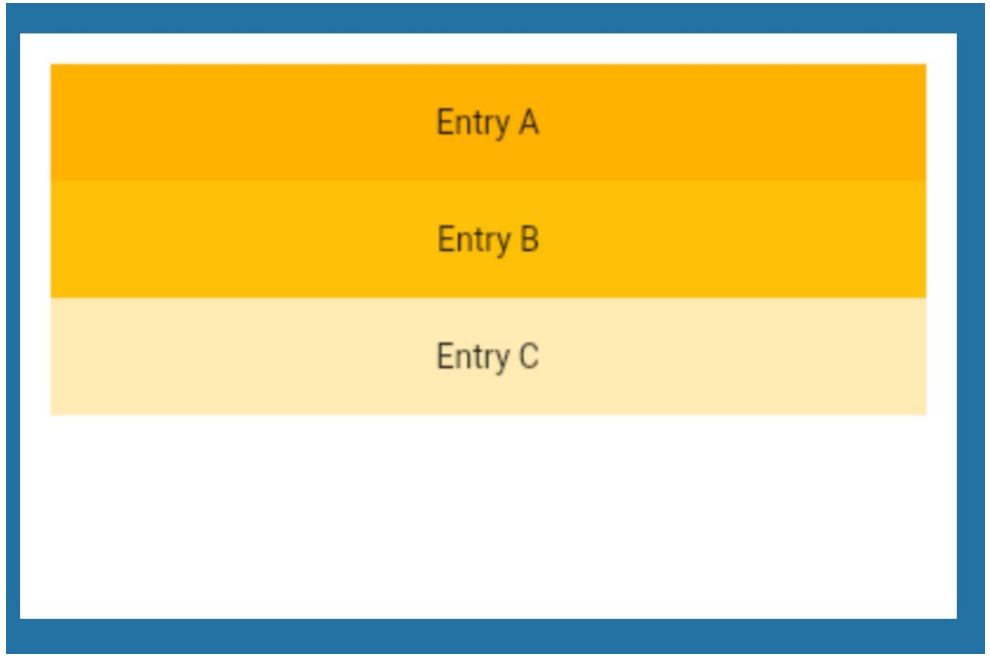
This example uses the default constructor for ListView which takes an explicit List<Widget> of children. This ListView's children are made up of Containers with Text.



LIST VIEW

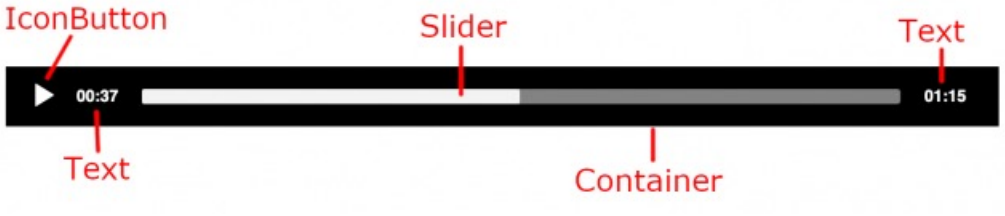
```
final List<String> entries = <String>['A', 'B', 'C'];
final List<int> colorCodes = <int>[600, 500, 100];

ListView.builder(
  padding: const EdgeInsets.all(8),
  itemCount: entries.length,
  itemBuilder: (BuildContext context, int index) {
    return Container(
      height: 50,
      color: Colors.amber[colorCodes[index]],
      child: Center(child: Text('Entry ${entries[index]}')),
    );
  }
);
```

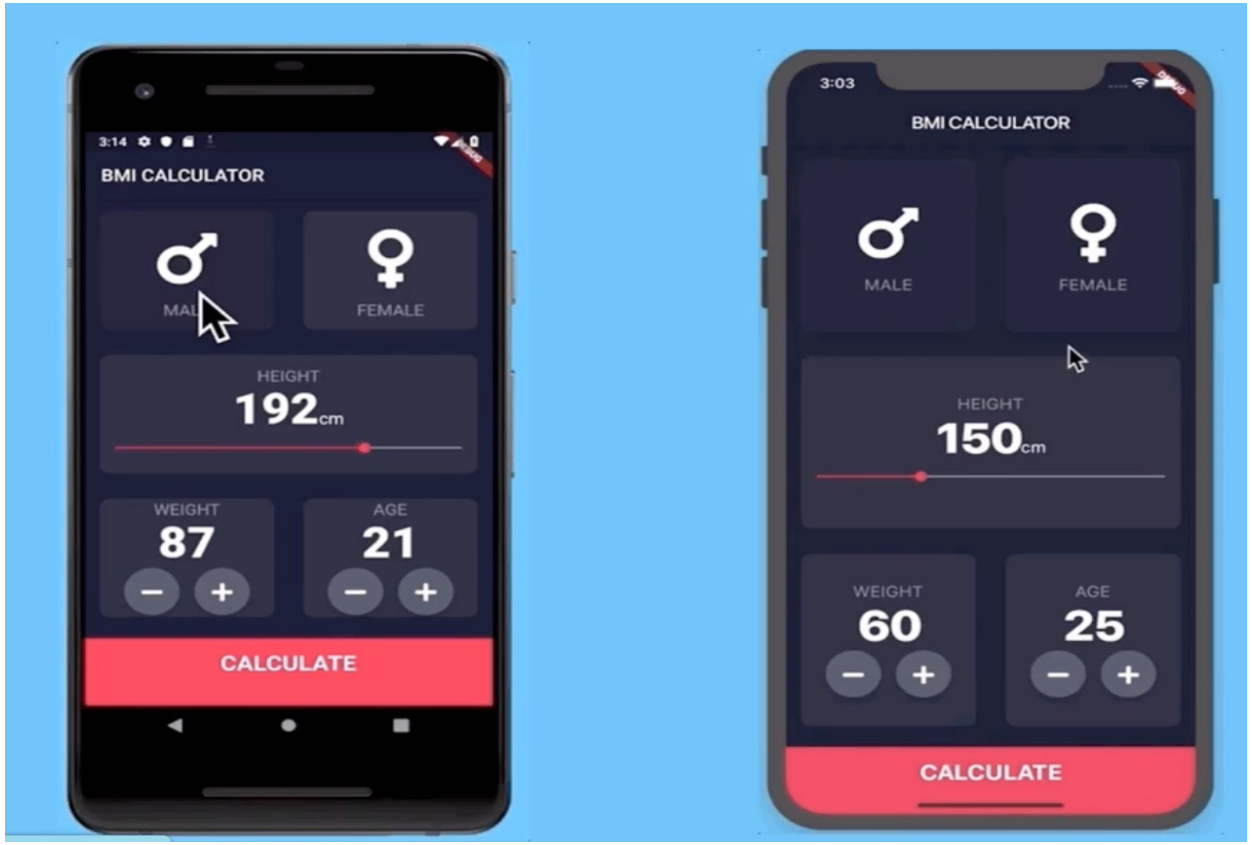
This example mirrors the previous one, creating the same list using the ListView.builder constructor. Using the IndexedWidgetBuilder, children are built lazily and can be infinite in number.

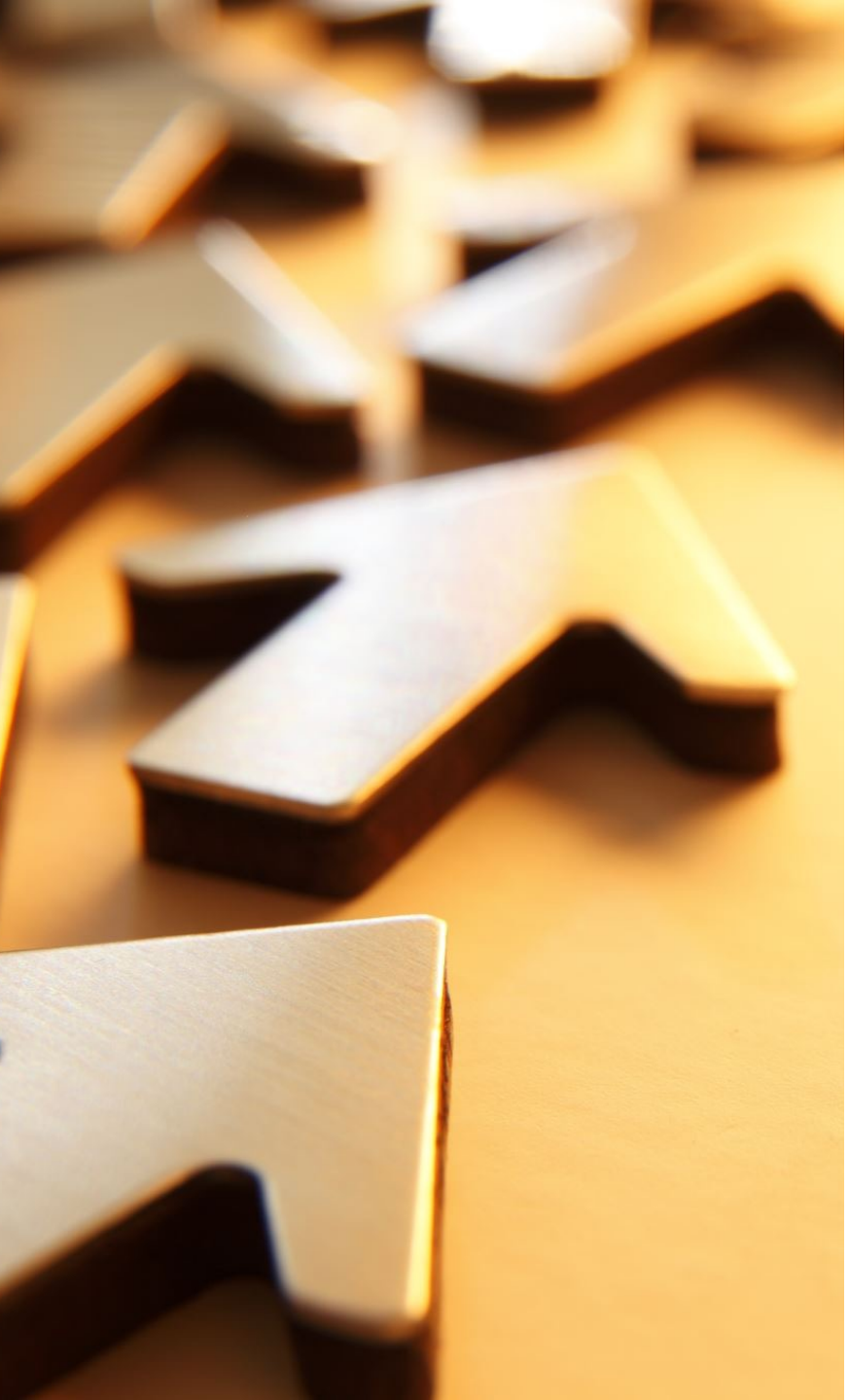


BUILD A CUSTOM WIDGET

- Everything's a widget in Flutter… so wouldn't it be nice to know how to make your own?
- There are several methods to create custom widgets, but the most basic is to combine simple existing widgets into the more complex widget that you want,
- This is called **composition**
- In Practical steps (Put your cursor on **Any Nested Widget** and **right-click** to show the context menu. Then choose **Refactor ▸ Extract ▸ Extract Flutter Widget…**.)







NAVIGATION IN FLUTTER

- Flutter has an imperative routing mechanism, the **Navigator** widget, and a more **idiomatic** declarative routing mechanism (which is similar to build methods as used with widgets), the **Router** widget.
- The two systems can be used together (indeed, the declarative system is built using the imperative system).

1. Typically, small applications are served well by just using the Navigator API, via the **MaterialApp** constructor's **MaterialApp.routes** property.

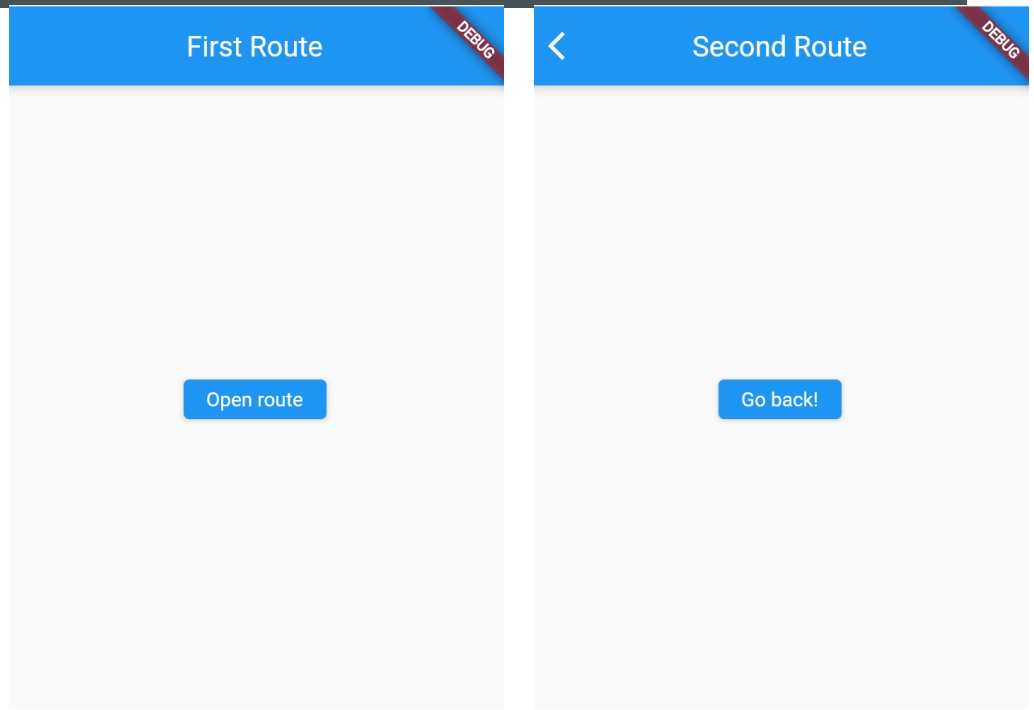
To learn about Navigator and its imperative API, see the **Navigation recipes** in the **Flutter cookbook**, and the **Navigator** API docs.

2. More elaborate applications are usually better served by the **Router** API, via the **MaterialApp.router** constructor.

NAVIGATE TO A NEW SCREEN AND BACK

Most apps contain several screens for displaying different types of information. For example, an app might have a screen that displays products. When the user taps the image of a product, a new screen displays details about the product.

- **Terminology**: In Flutter, *screens* and *pages* are called ***routes***. The remainder of this recipe refers to routes.
- In Android, a route is equivalent to an **Activity**. In iOS, a route is equivalent to a **ViewController.** In Flutter, a route is just a widget.
- This coming example uses the **Navigator** to navigate to a new route.



The next few sections show how to navigate between two routes, using these steps:

1. Create two routes.
2. Navigate to the second route using Navigator.push().
3. Return to the first route using Navigator.pop().

NAVIGATE TO A NEW SCREEN AND BACK

1. Create two routes:

- First, create two routes to work with. Since this is a basic example, each route contains only a single button.
- Tapping the button on the first route navigates to the second route.
- Tapping the button on the second route returns to the first route. First, set up the visual structure:

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('First Route'),
      ),
      body: Center(
        child: ElevatedButton(
          child: Text('Open route'),
          onPressed: () {
            // Navigate to second route when tapped.
          },
        ),
      ),
    );
  }
}
```

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Second Route"),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Navigate back to first route when tapped.
          },
          child: Text('Go back!'),
        ),
      ),
    );
  }
}
```

NAVIGATE TO A NEW SCREEN AND BACK

2. Navigate to the second route using Navigator.push()

- To switch to a new route, use the **Navigator.push()** method. The push() method adds a **Route** to the stack of routes managed by the **Navigator**. Where does the Route come from? You can create your own, or use a **MaterialPageRoute,** which is useful because it transitions to the new route using a platform-specific animation.
- In the **build()** method of the **FirstRoute** widget, update the **onPressed()** callback:

```
// Within the `FirstRoute` widget
onPressed: () {
  Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => SecondRoute()),
  );
}
```

NAVIGATE TO A NEW SCREEN AND BACK

3. Return to the first route using **Navigator.pop()**

- How do you close the second route and return to the first? By using the Navigator.pop() method. The pop() method removes the current Route from the stack of routes managed by the Navigator.
- To implement a return to the original route, update the onPressed() callback in the SecondRoute widget:

```
// Within the SecondRoute widget
onPressed: () {
  Navigator.pop(context);
}
```

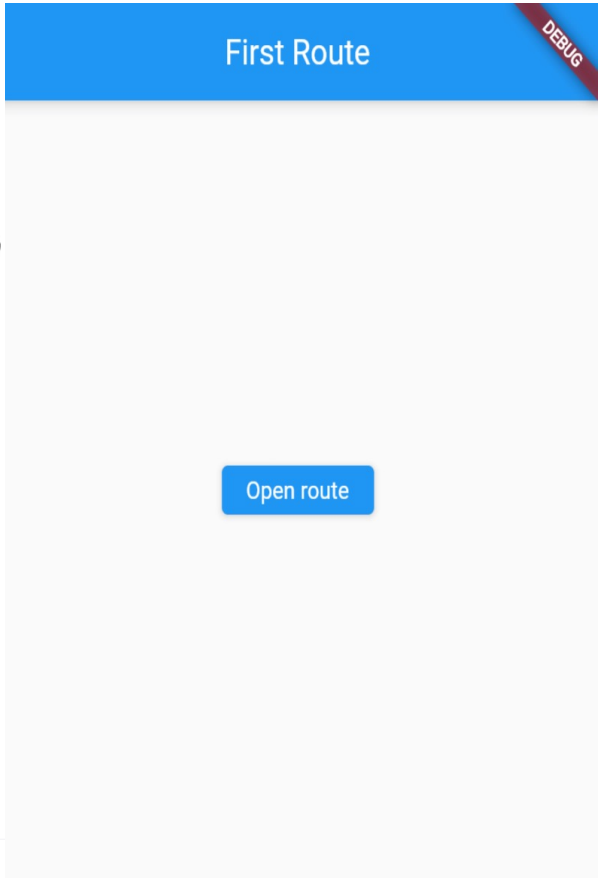
INTERACTIVE EXAMPLE

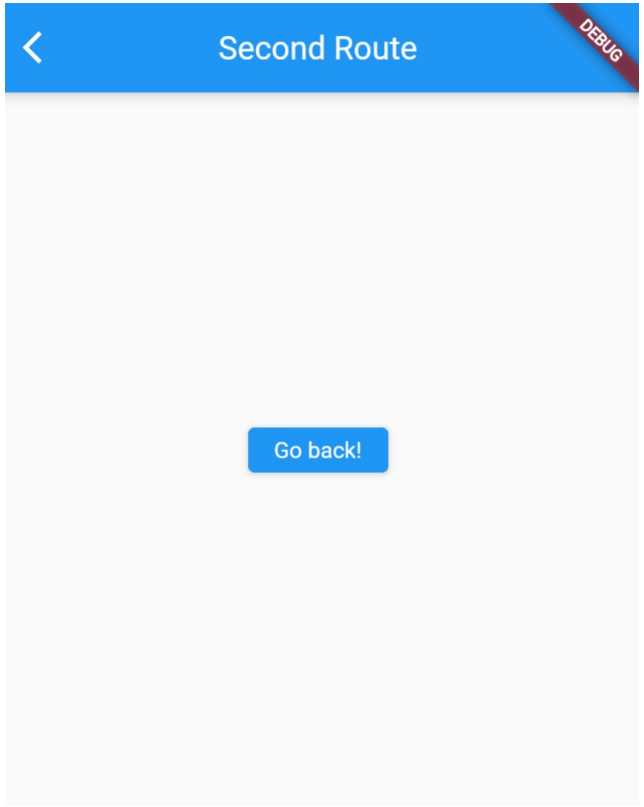
```dart
import 'package:flutter/material.dart';

void main() {
  runApp(const MaterialApp(
    title: 'Navigation Basics',
    home: FirstRoute(),
  ));
}
```

```dart
class FirstRoute extends StatelessWidget {
  const FirstRoute({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Route'),
      ),
      body: Center(
        child: ElevatedButton(
          child: const Text('Open route'),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => const SecondRoute()),
            );
          },
        ),
      ),
    );
  }
}
```

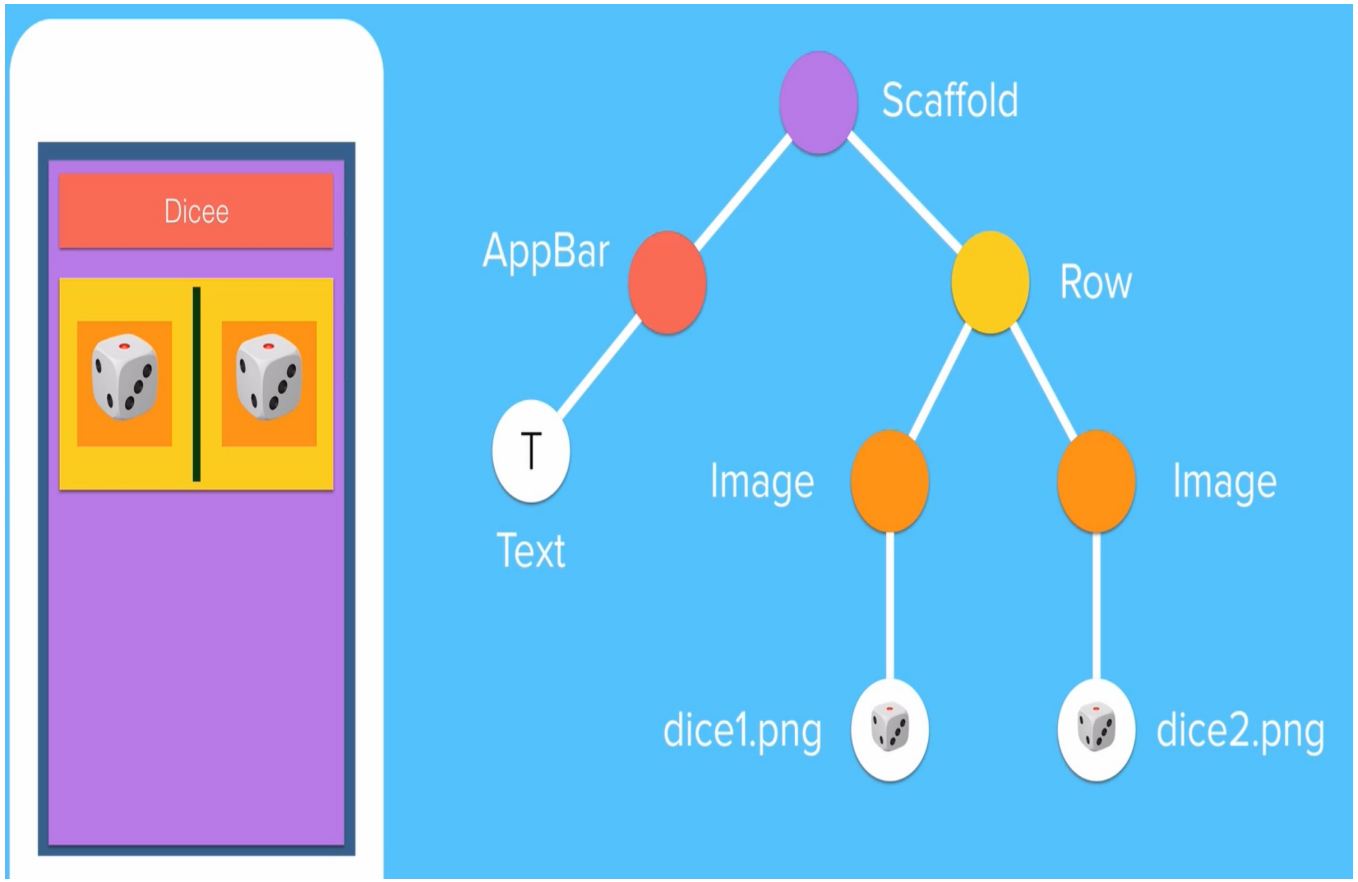
INTERACTIVE EXAMPLE

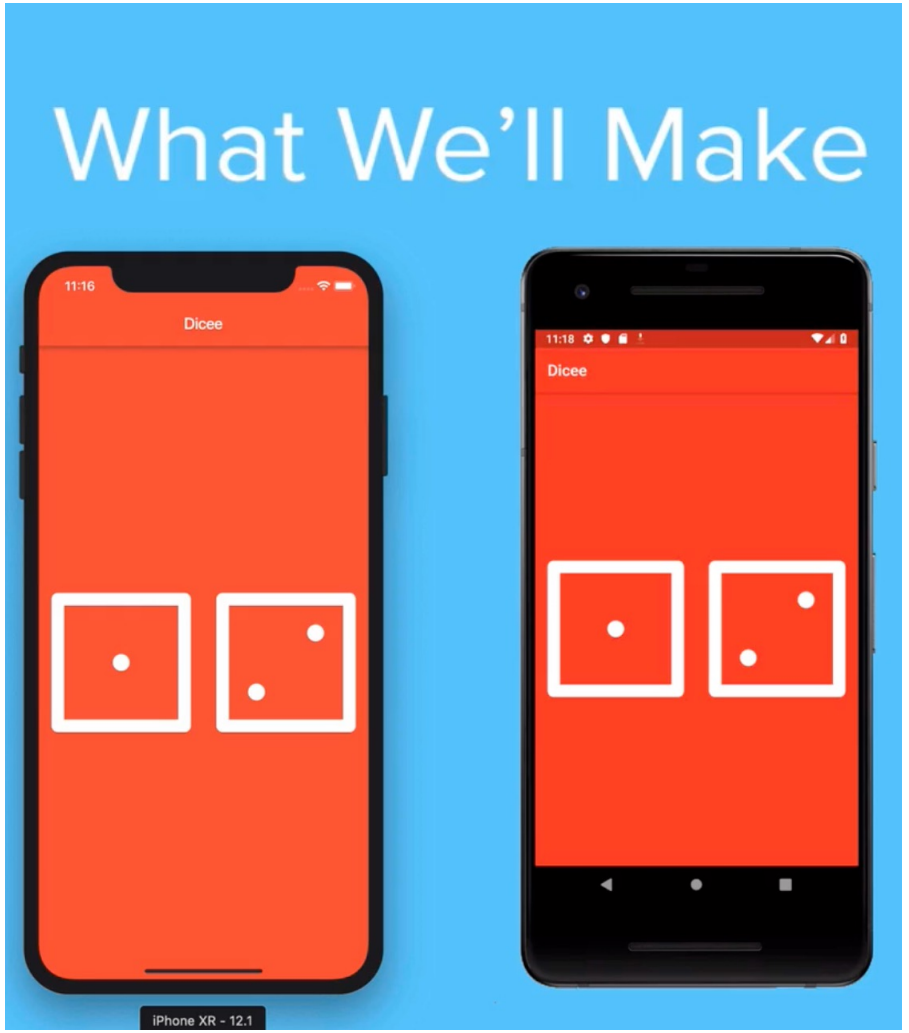
```dart
class SecondRoute extends StatelessWidget {
  const SecondRoute({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("Second Route"),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: const Text('Go back!'),
        ),
      ),
    );
  }
}
```





STATEFUL WIDGETS AND BUILDING AN INTERACTIVE APPLICATIONS



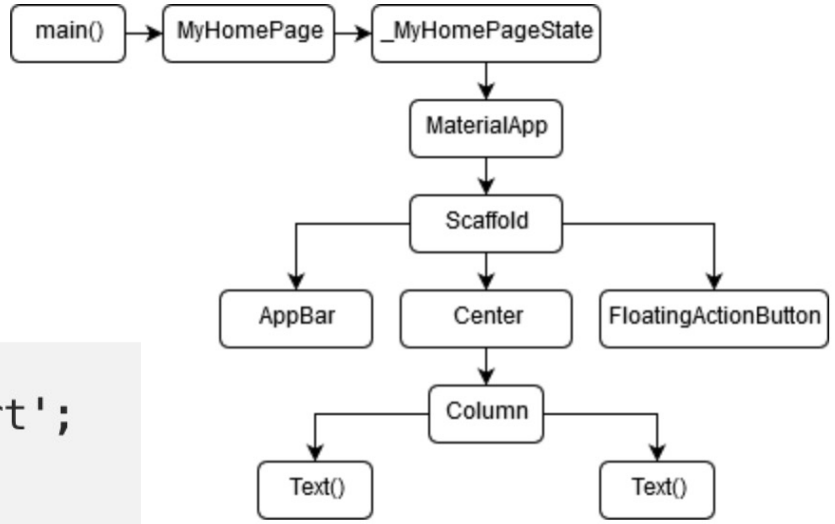


STATEFUL WIDGETS AND BUILDING AN INTERACTIVE APPLICATIONS

- A widget is either stateful or stateless. If a widget can change—when a user interacts with it, for example—it's stateful.
- A *stateless* widget never changes. Icon, IconButton, and Text are examples of stateless widgets. Stateless widgets subclass StatelessWidget.
- A *stateful* widget is dynamic: for example, it can change its appearance in response to events triggered by user interactions or when it receives data. Checkbox, Radio, Slider, InkWell, Form, and TextField are examples of stateful widgets. Stateful widgets subclass StatefulWidget.
- A widget's state is stored in a State object, separating the widget's state from its appearance.
- The state consists of values that can change, like a slider's current value or whether a checkbox is checked. When the widget's state changes, the state object calls **setState**(), telling the framework to redraw the widget.

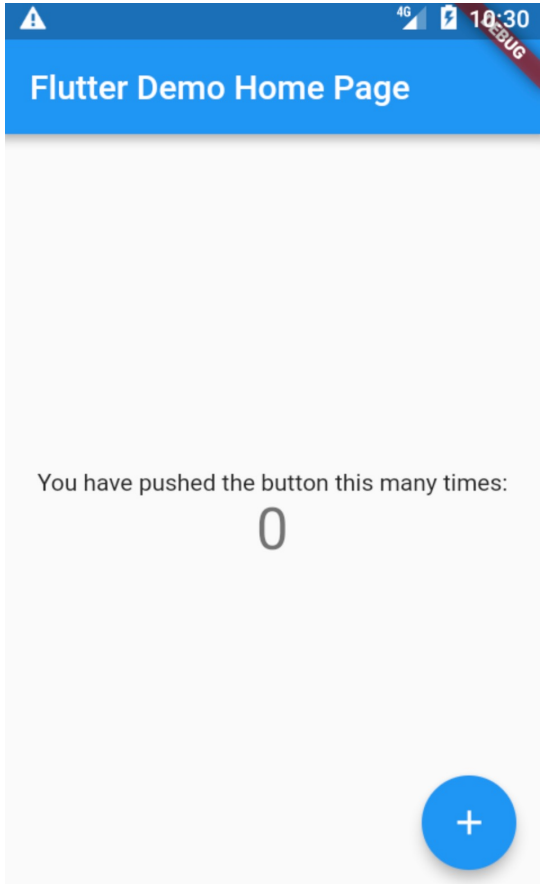
STATEFUL WIDGETS AND BUILDING AN INTERACTIVE APPLICATIONS

The starting point...




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

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

```
class MyHomePage extends StatefulWidget {
    @override
    _MyHomePageState createState() => _MyHomePageState();
}
```



STATEFUL WIDGETS AND BUILDING AN INTERACTIVE APPLICATIONS

You may notice that we have returned an object of **_MyHomePageState** Class. This class is of type ***State<MyHomePage>*** and has a **build**() method that returns the **Widget** that should appear on the screen.

```
class _MyHomePageState extends State<MyHomePage> {
    @override
    Widget build(BuildContext context) {
        return null;
    }
}
```

In **_MyHomePageState** we will now have a global variable that will represent state of our app and a method that will change the state of our app on every click of **FloatingActionButton**.

```
int _counter = 0;
void _incrementCounter() {
     setState(() {
         _counter++;
     });
}
```

So here is our full Widget tree.

```
@override
Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Hello Flutter'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Text('You have pushed the button this many times:',),
              Text('$_counter'),
            ],
          ),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: _incrementCounter,
          tooltip: 'Increment',
          child: Icon(Icons.add),
        ),
      ),
    );
}
```

KEY POINTS

- Flutter has widget for almost everything you need, and it allows you to build your own custom widget if you want to build your own brand widgets.
- A **ListView** and **ListTile** widgets are very useful when you want your app has any kind of list.
- The navigation in Flutter is done by **Navigator API** from **MaterialApp** widget.
- You should always start by creating **StatelessWidgets** and only use **StatefulWidgets** when you need to manage and maintain the state of your widget.
- Most of the time you can use Stateless widgets unless you want to update data on the UI !