Week 5

2 Nov 2025

IT DEPT.

TIU

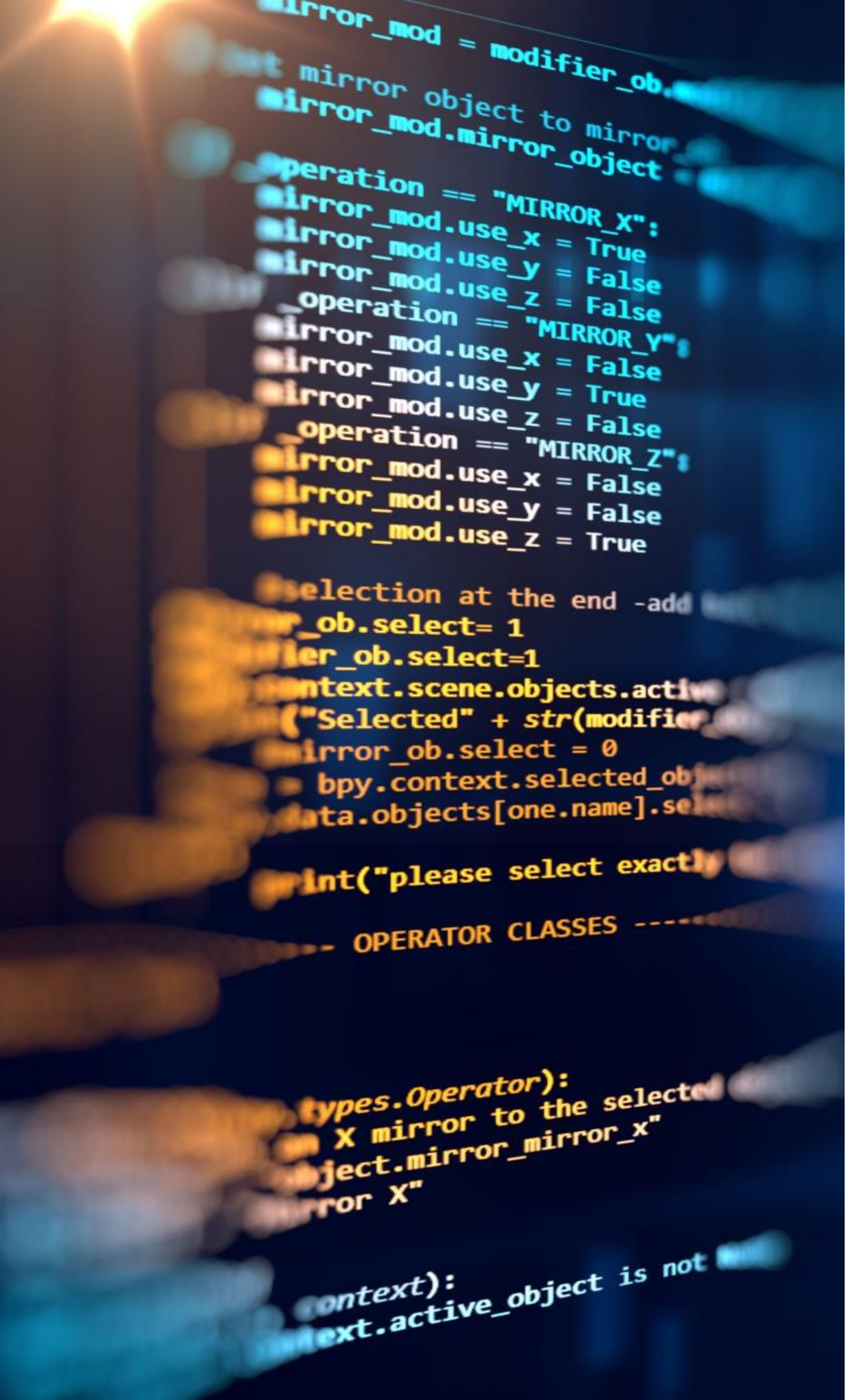
3RD GRADE

MOBILE APPLICATIONS IT 319 (OOPI EARLIER)

Application Bar, List View and Build A Custom Widget
2025-2026

Lect. Mohammad Salim

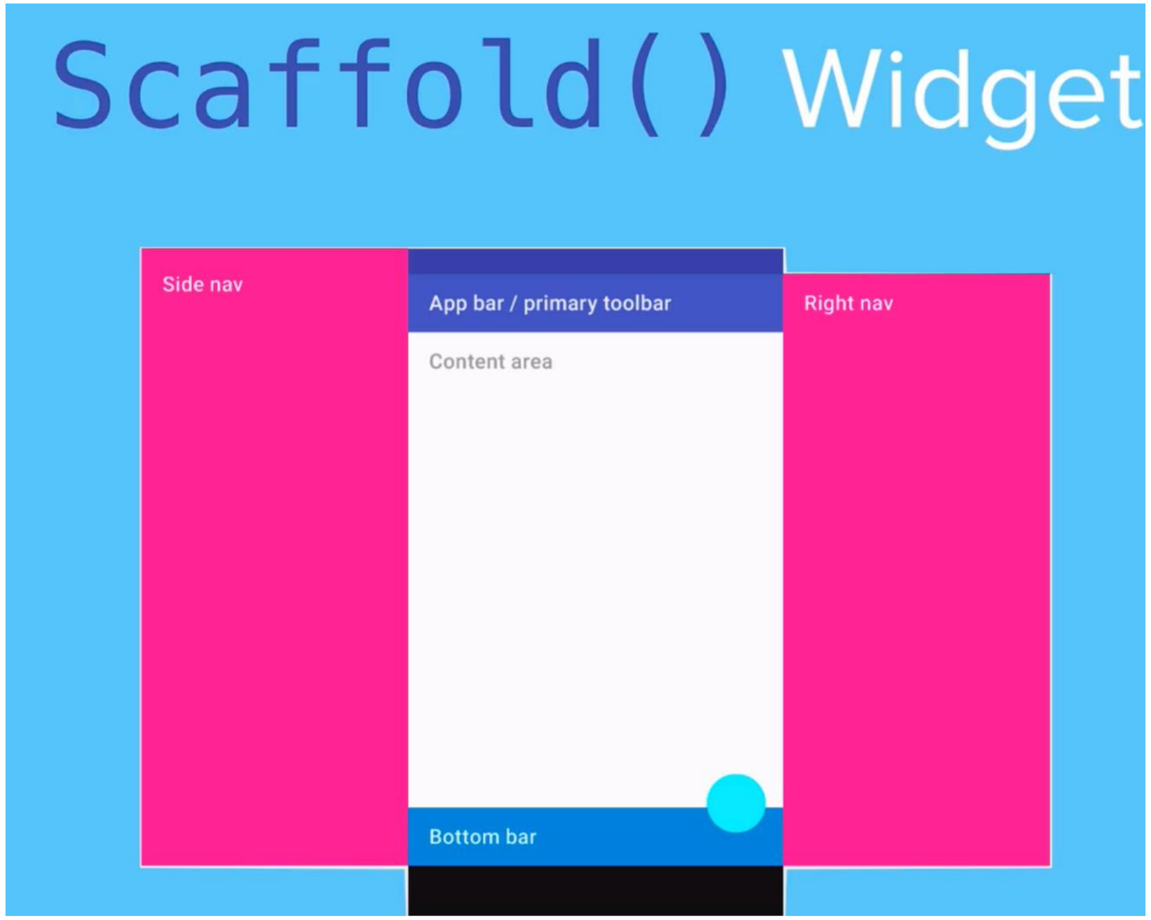
Thses slide notes are based on many different online resources such as Flutter.dev, Flutter Apprentice Book and The complete Fluter bootcamp course



CONTENTS

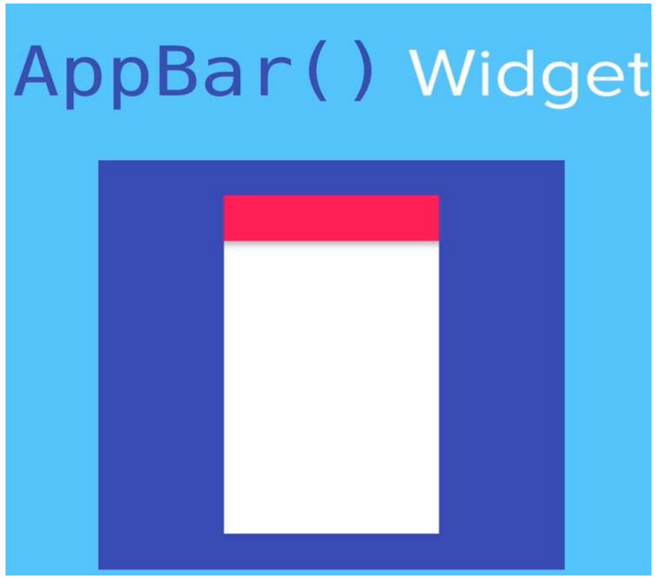
APP BAR: SCAFFOLDING A MATERIAL APP

- 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.

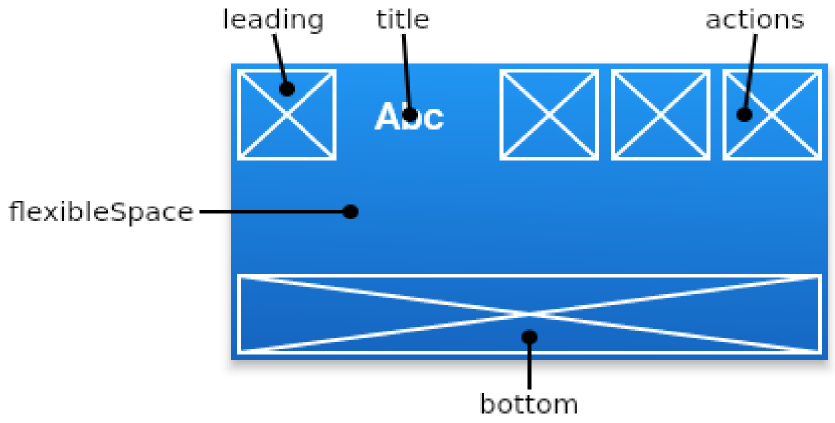


APP BAR: SCAFFOLDING A MATERIAL APP

- 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.



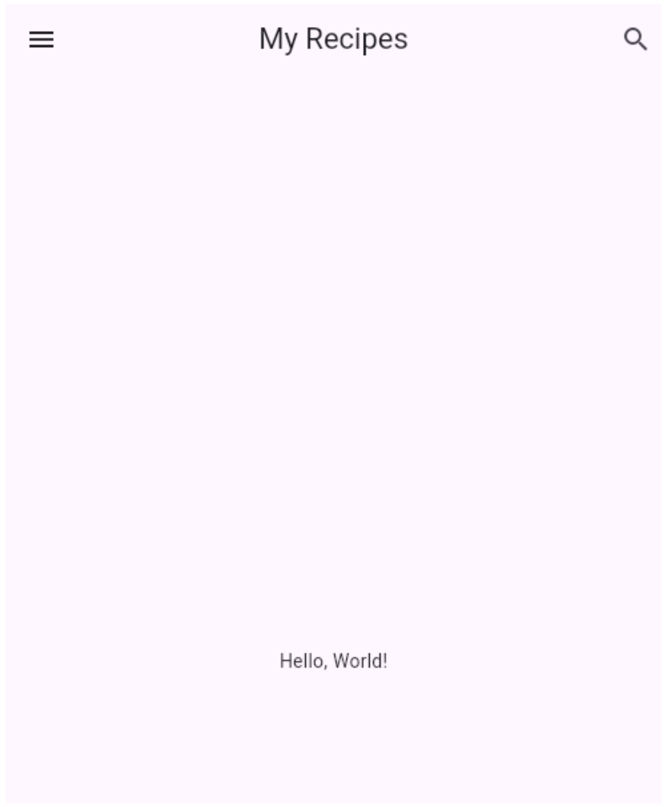
```
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);

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

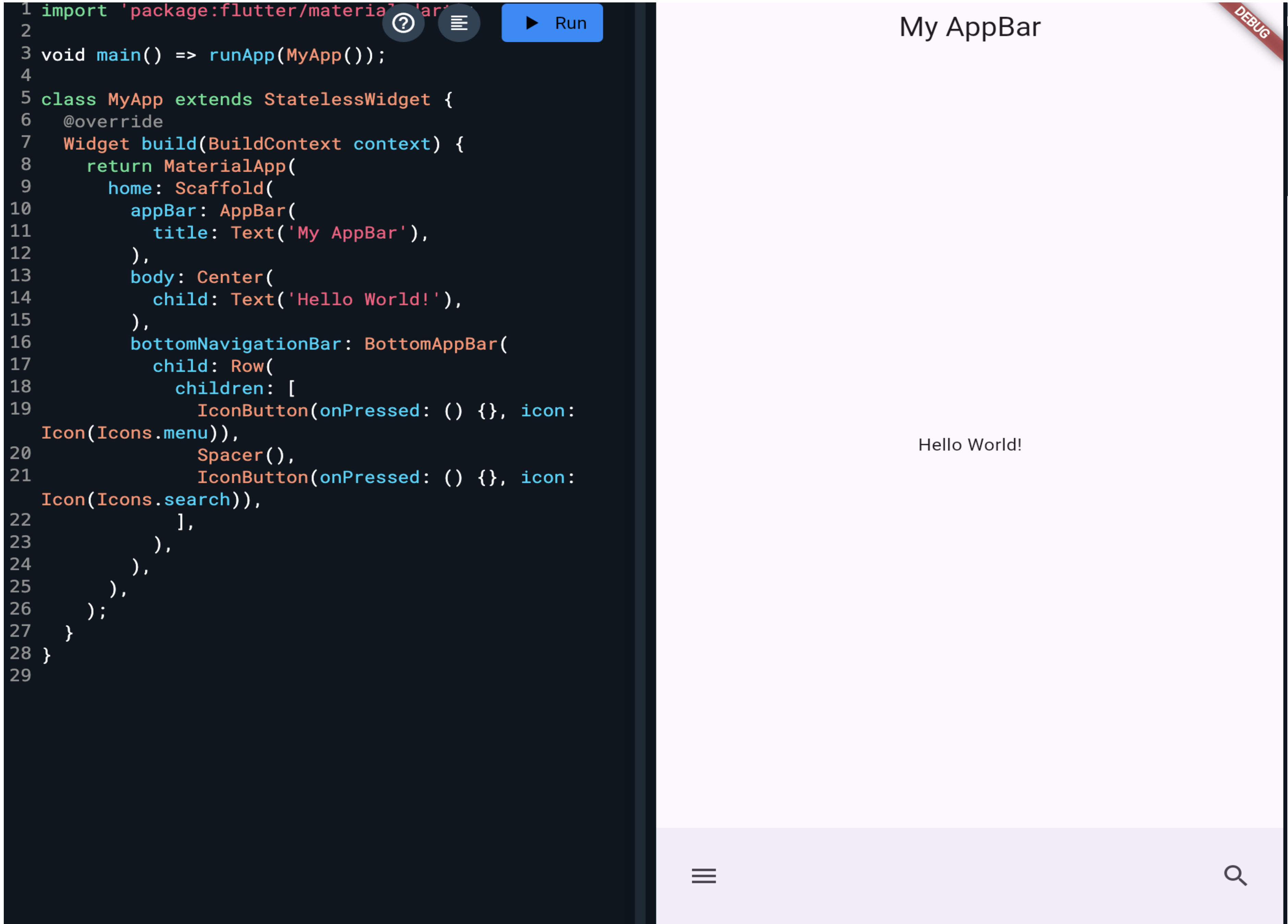


APPBAR EXAMPLES

dart

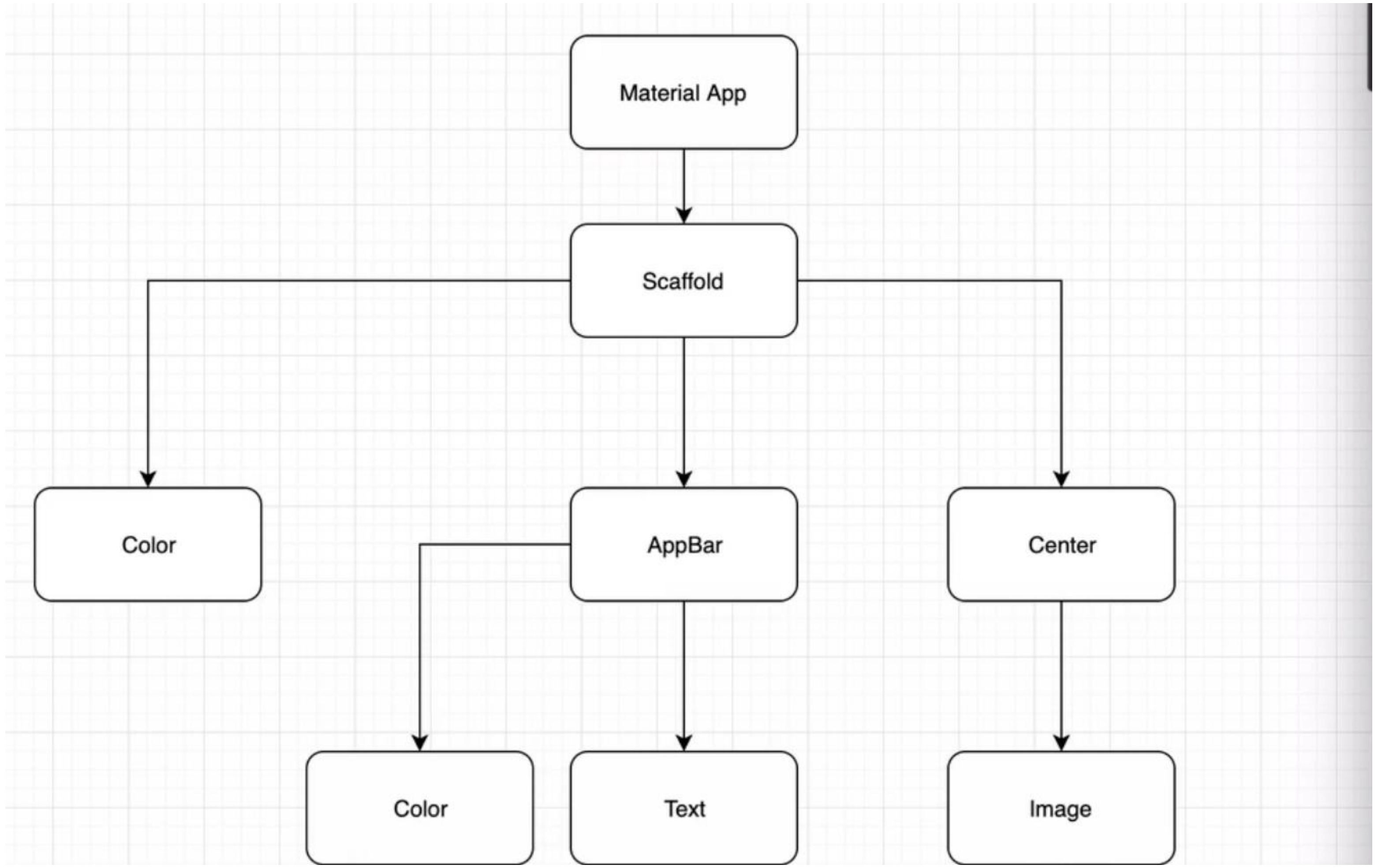
```
// Real example with leading + title + actions
AppBar(
  leading: IconButton(icon: Icon(Icons.menu), onPressed: () {}),
  title: Text('My Recipes'),
  actions: [
    IconButton(icon: Icon(Icons.search), onPressed: () {}),
  ],
)
```

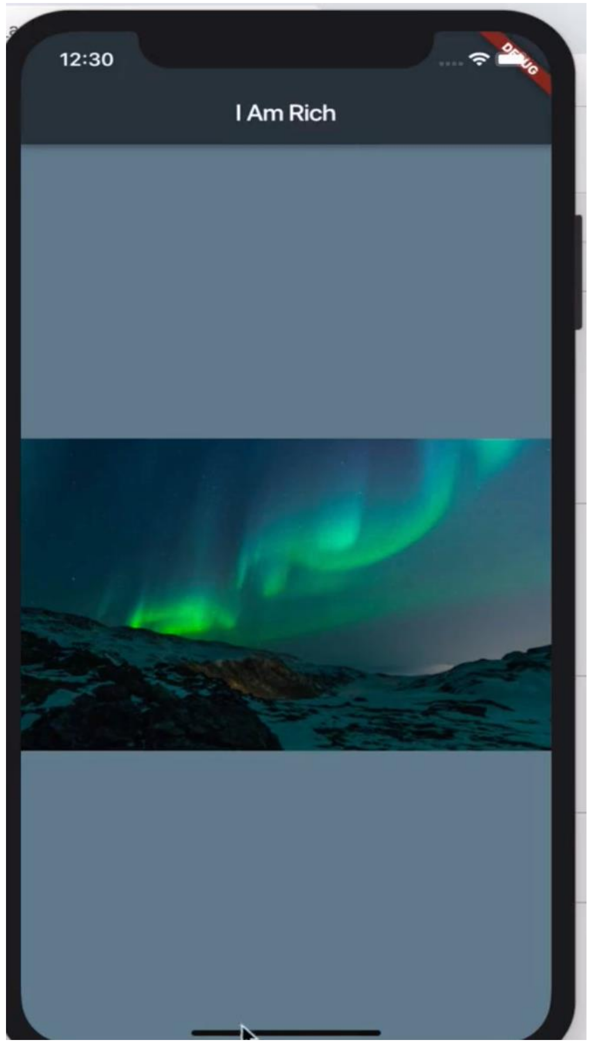




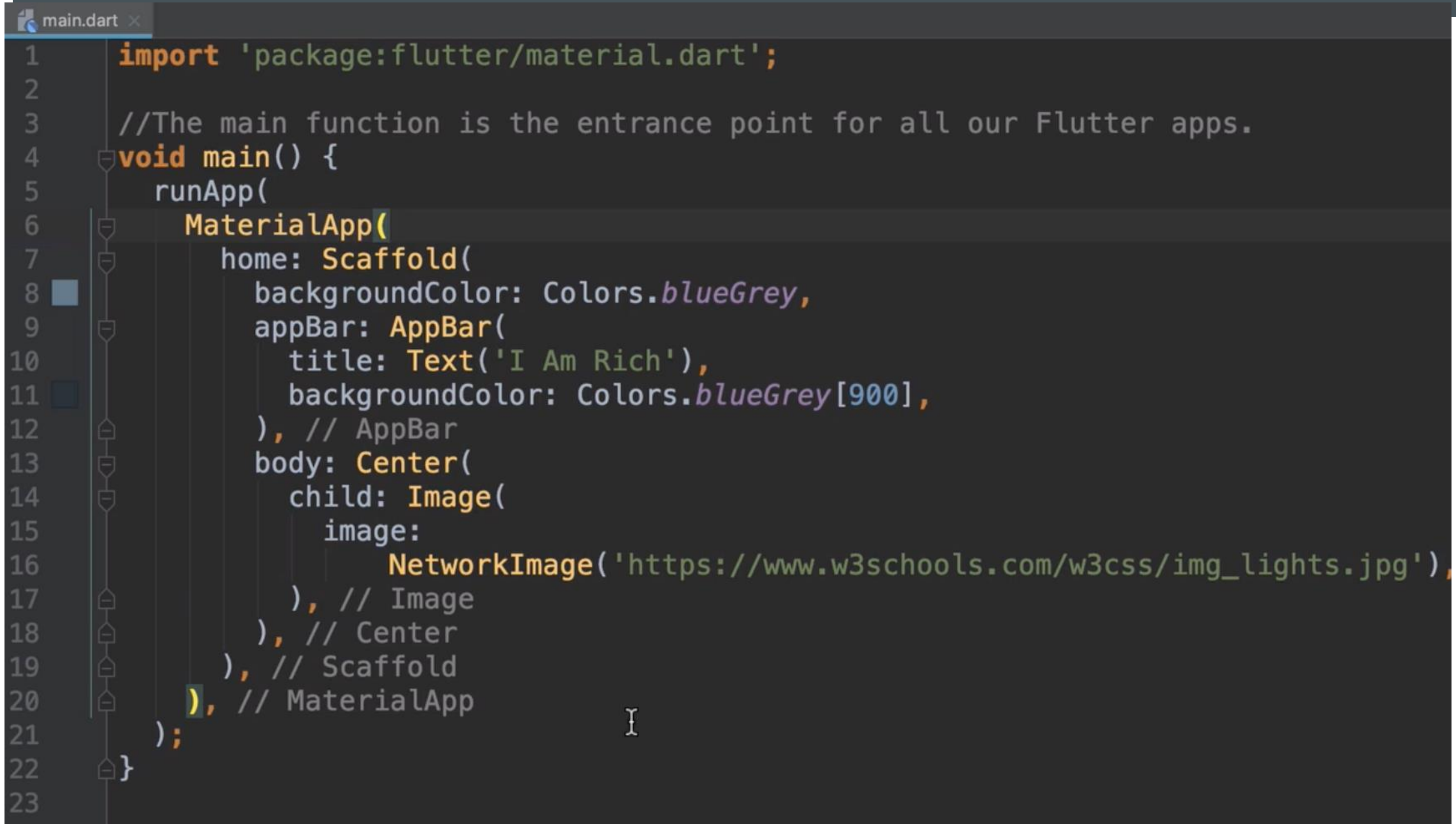
```dart
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('My AppBar'),
        ),
        body: Center(
          child: Text('Hello World!'),
        ),
        bottomNavigationBar: BottomAppBar(
          child: Row(
            children: [
              IconButton(onPressed: () {}, icon: Icon(Icons.menu)),
              Spacer(),
              IconButton(onPressed: () {}, icon: Icon(Icons.search)),
            ],
          ),
        ),
      ),
    );
  }
}
```

APP BAR: SCAFFOLDING A MATERIAL APP



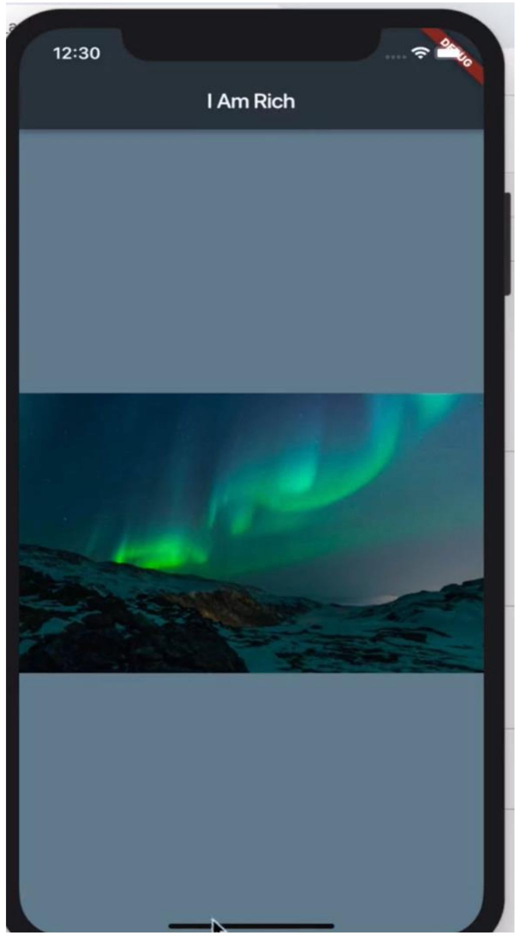


APP BAR: SCAFFOLDING A MATERIAL APP



main.dart

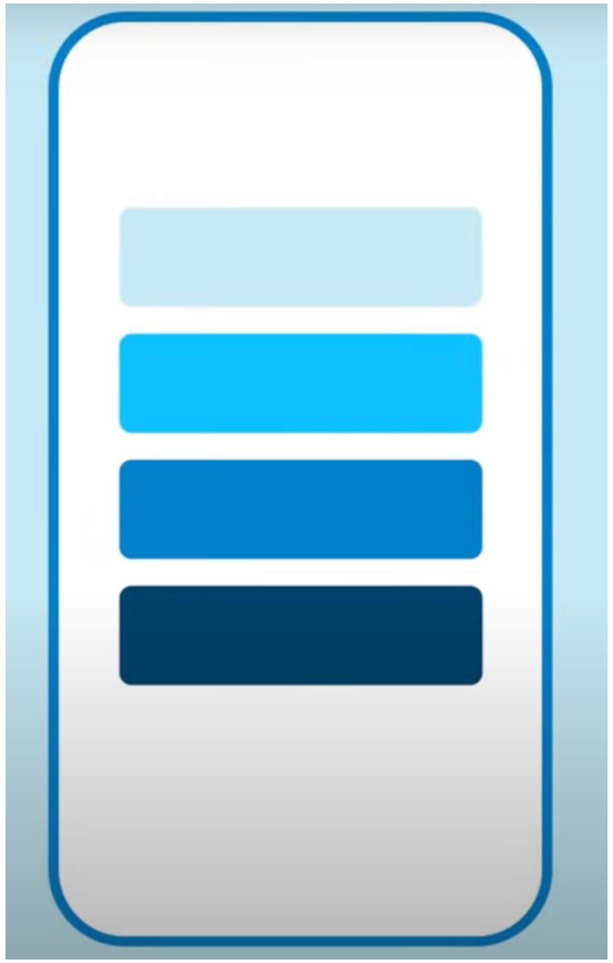
```
import 'package:flutter/material.dart';

//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:

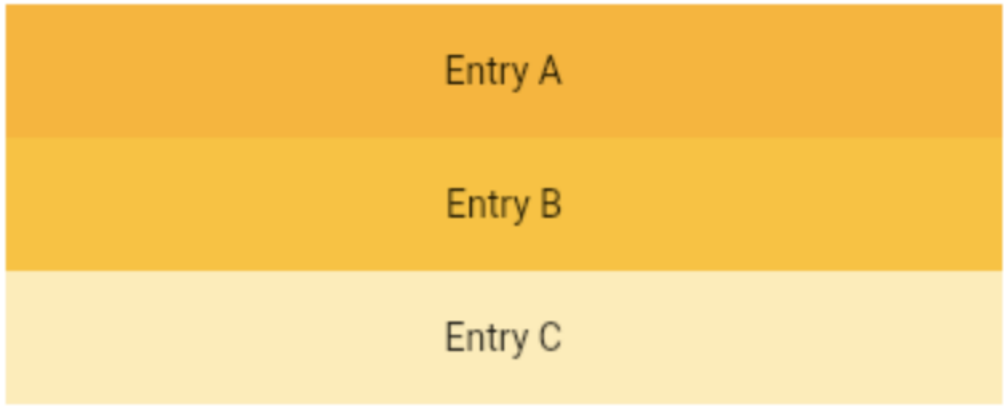
1. The default constructor takes an explicit List<Widget> of children. This constructor is appropriate for list views with a small number of children because constructing the List requires doing work for every child that could possibly be displayed in the list view instead of just those children that 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')),
    ),
  ],
)
```

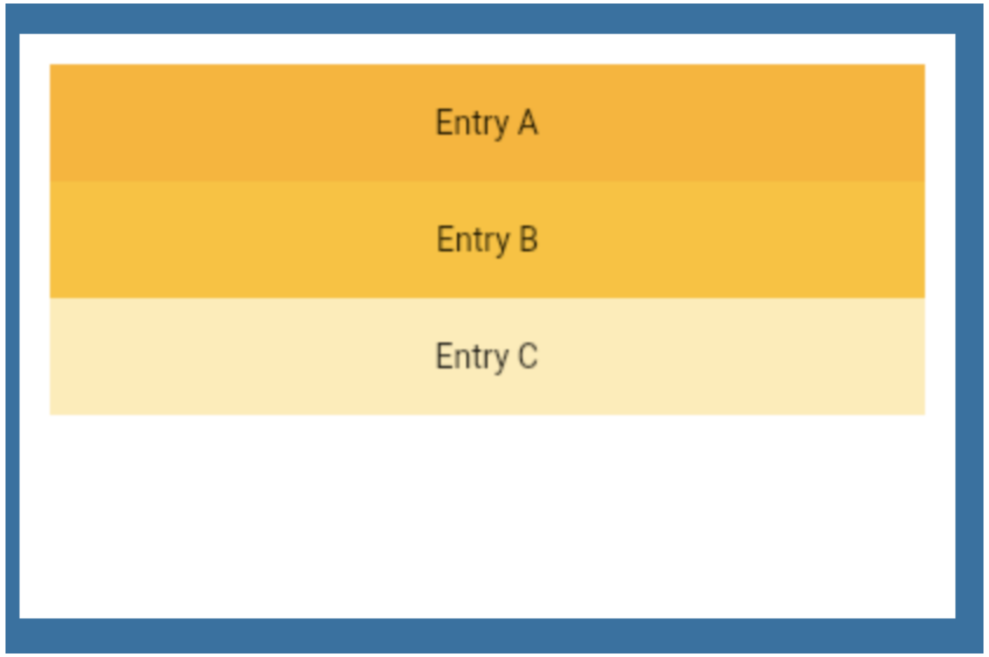
This example uses the default constructor for ListView which takes an explicit List<Widget> of children. This ListView's children are made up of Containers with Text.



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.



Example for Default Constructor:

```dart
ListView(
  children: [
    ListTile(
      leading: Icon(Icons.map),
      title: Text('Map'),
    ),
    ListTile(
      leading: Icon(Icons.photo),
      title: Text('Photos'),
    ),
  ],
);
```

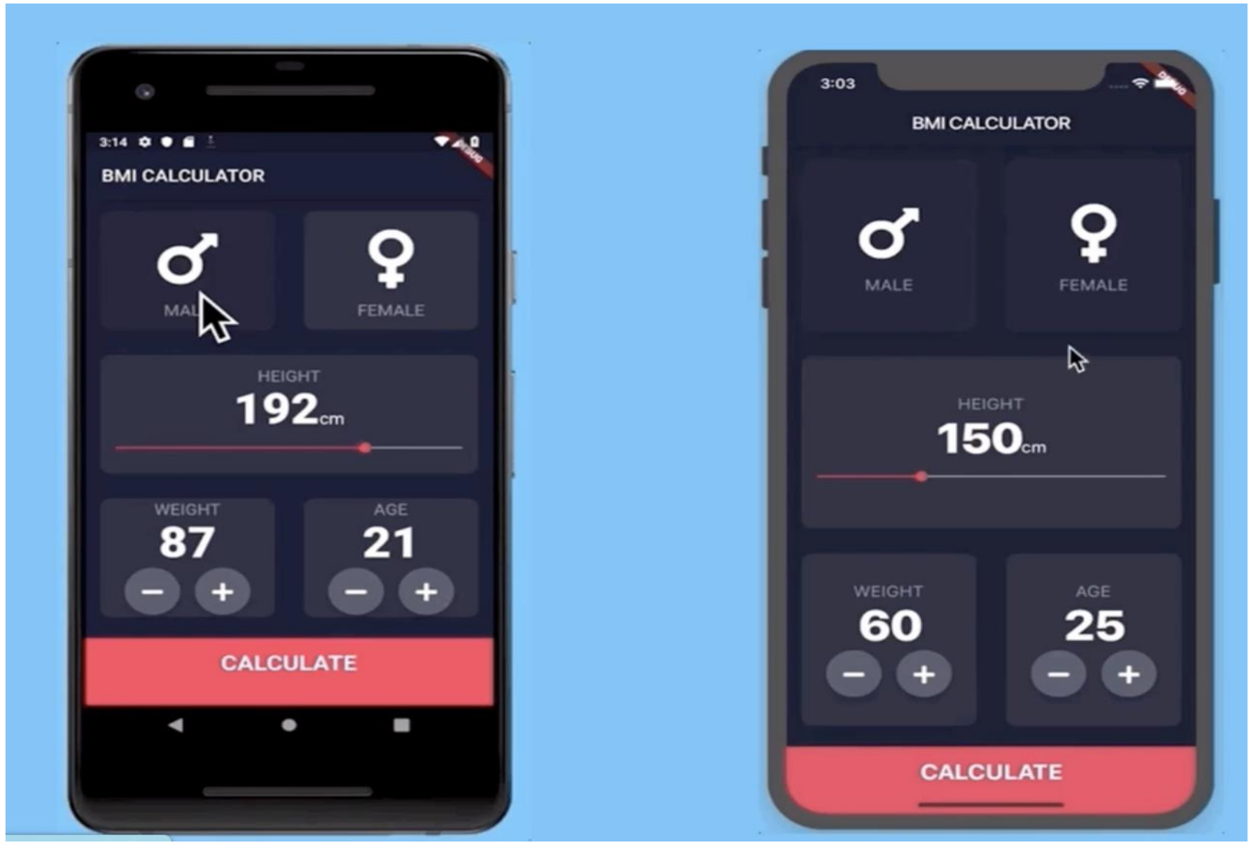
Example for `ListView.builder`:

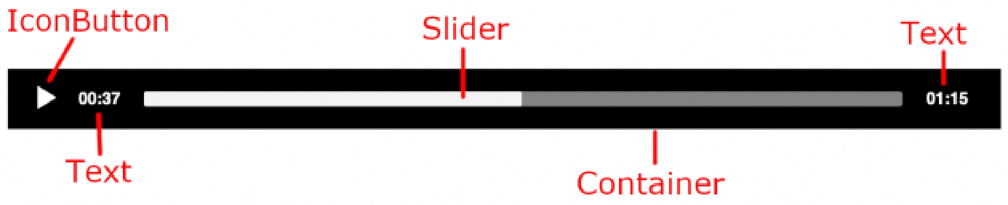
```dart
ListView.builder(
  itemCount: 10,
  itemBuilder: (context, index) {
    return ListTile(
      title: Text('Item $index'),
    );
  },
);
```

// Use this when: List might be large, from API, or infinite

BUILD A CUSTOM WIDGET

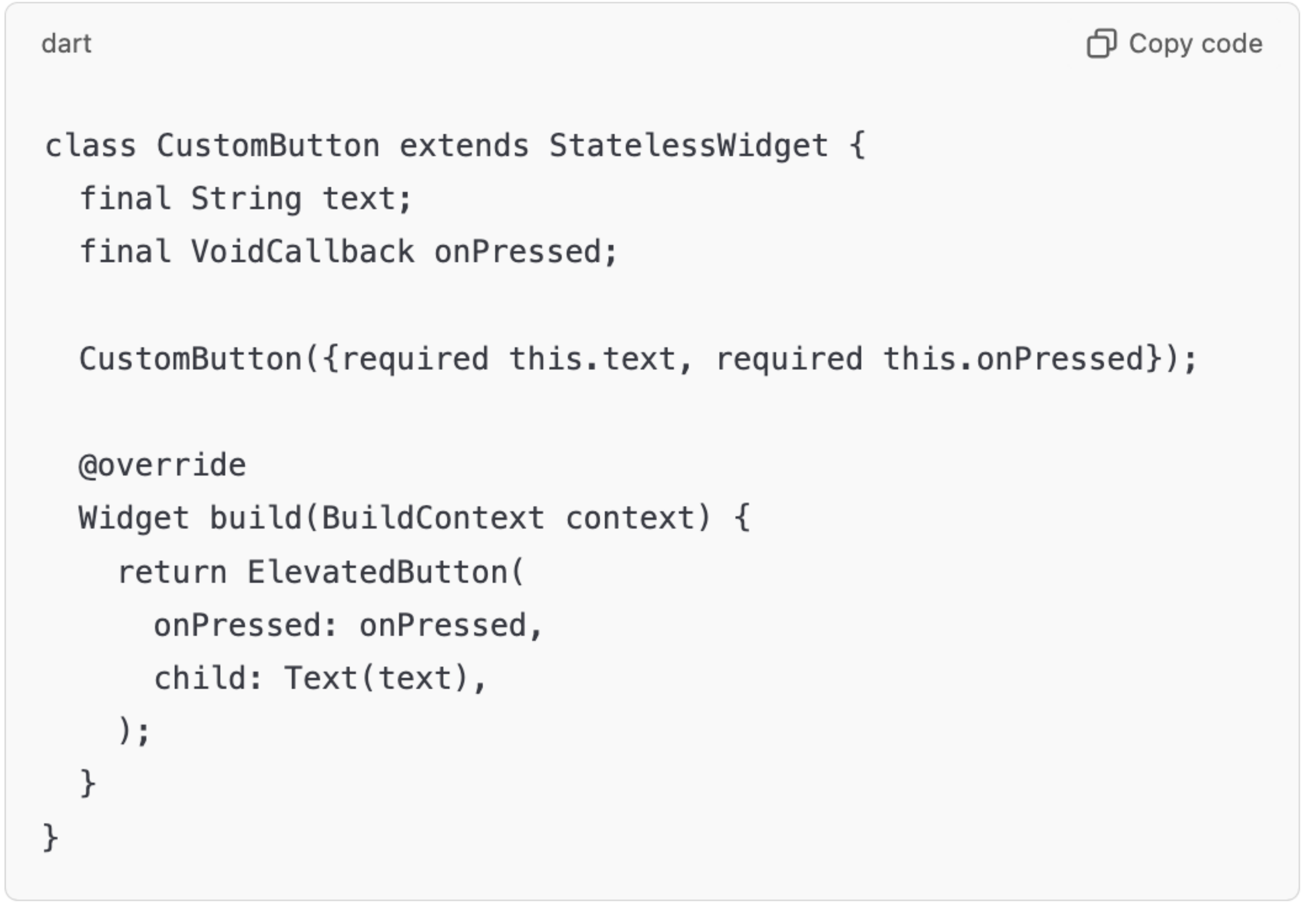
- Everything's a widget in Flutter… so wouldn't it be nice to know how to make your own?
- There are several methods to create custom widgets, but the most basic is to combine simple existing widgets into the more complex widget that you want,
- This is called **composition**
- In Practical steps (Put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





CUSTOM WIDGETS

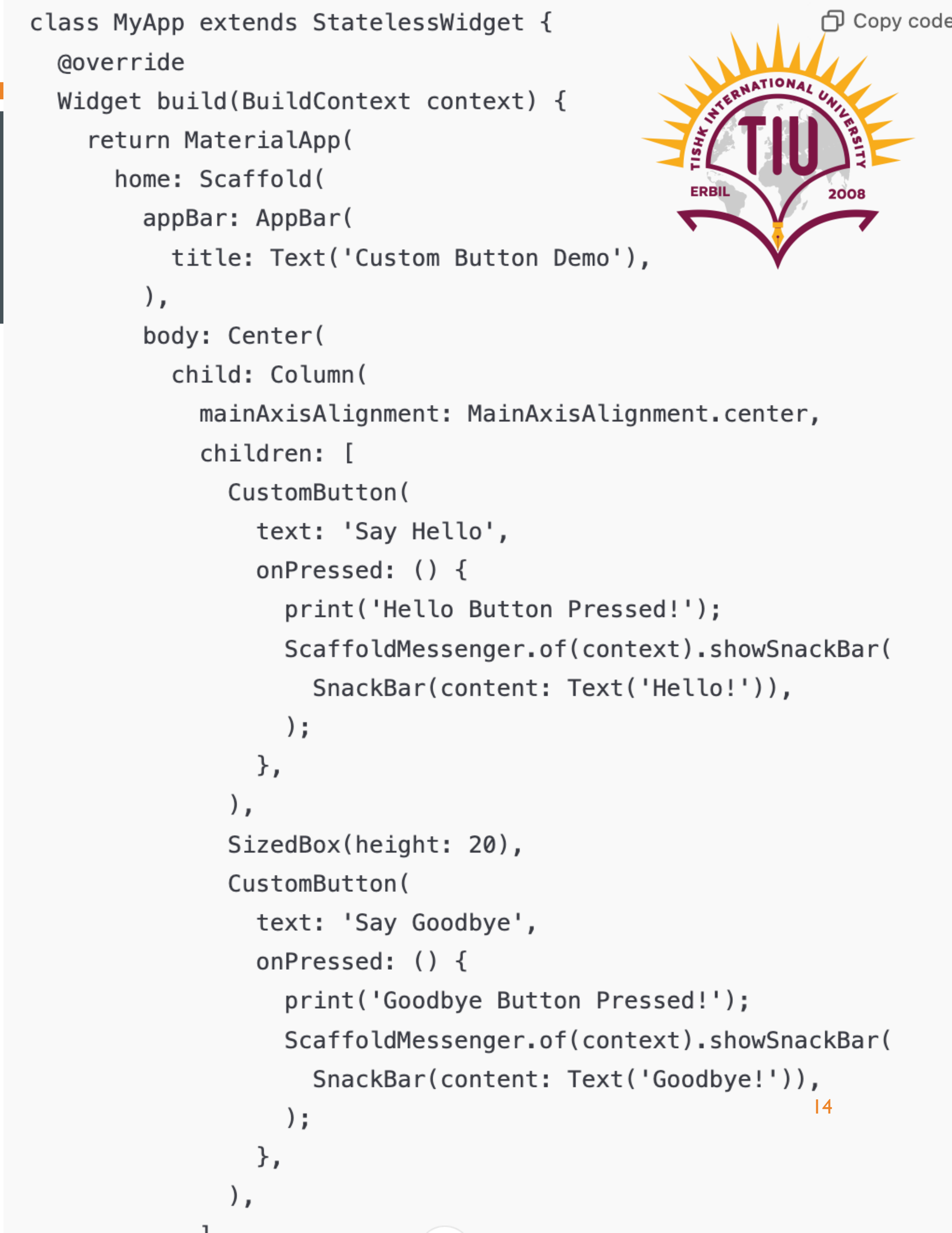
Example Code:



dart

Copy code

```dart
class CustomButton extends StatelessWidget {
  final String text;
  final VoidCallback onPressed;

  CustomButton({required this.text, required this.onPressed});

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



Copy code

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Custom Button Demo'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              CustomButton(
                text: 'Say Hello',
                onPressed: () {
                  print('Hello Button Pressed!');
                  ScaffoldMessenger.of(context).showSnackBar(
                    SnackBar(content: Text('Hello!')),
                  );
                },
              ),
              SizedBox(height: 20),
              CustomButton(
                text: 'Say Goodbye',
                onPressed: () {
                  print('Goodbye Button Pressed!');
                  ScaffoldMessenger.of(context).showSnackBar(
                    SnackBar(content: Text('Goodbye!')),
                  );
                },
              ),
```

HOW IT WORKS:

1. **CustomButton Reusability**:
    1. The **CustomButton** widget is reused twice: once for the "Say Hello" button and once for the "Say Goodbye" button.
    2. The text and **onPressed** properties allow for customization.
2. **Interactivity**:
    1. Each button performs a different action when pressed, demonstrating reusability with distinct behaviors.
3. **Output**:
    1. Clicking the "Say Hello" button prints "Hello Button Pressed!" in the console and shows a SnackBar saying "Hello!".
    2. Clicking the "Say Goodbye" button does the same with "Goodbye!".

PERFORMANCE TIP: USE CONST

dart

```dart
// ✓ GOOD - const prevents unnecessary rebuilds
const Icon(Icons.add)
const SizedBox(height: 16)
const Text('Label')

// In custom widgets, use const constructor:
const RecipeCard(
  name: 'Pasta',
  servings: 2,
  onTap: null,
)

// Why? Const widgets don't rebuild when parent rebuilds
// If you have 100 list items, 100 const saves memory!
```

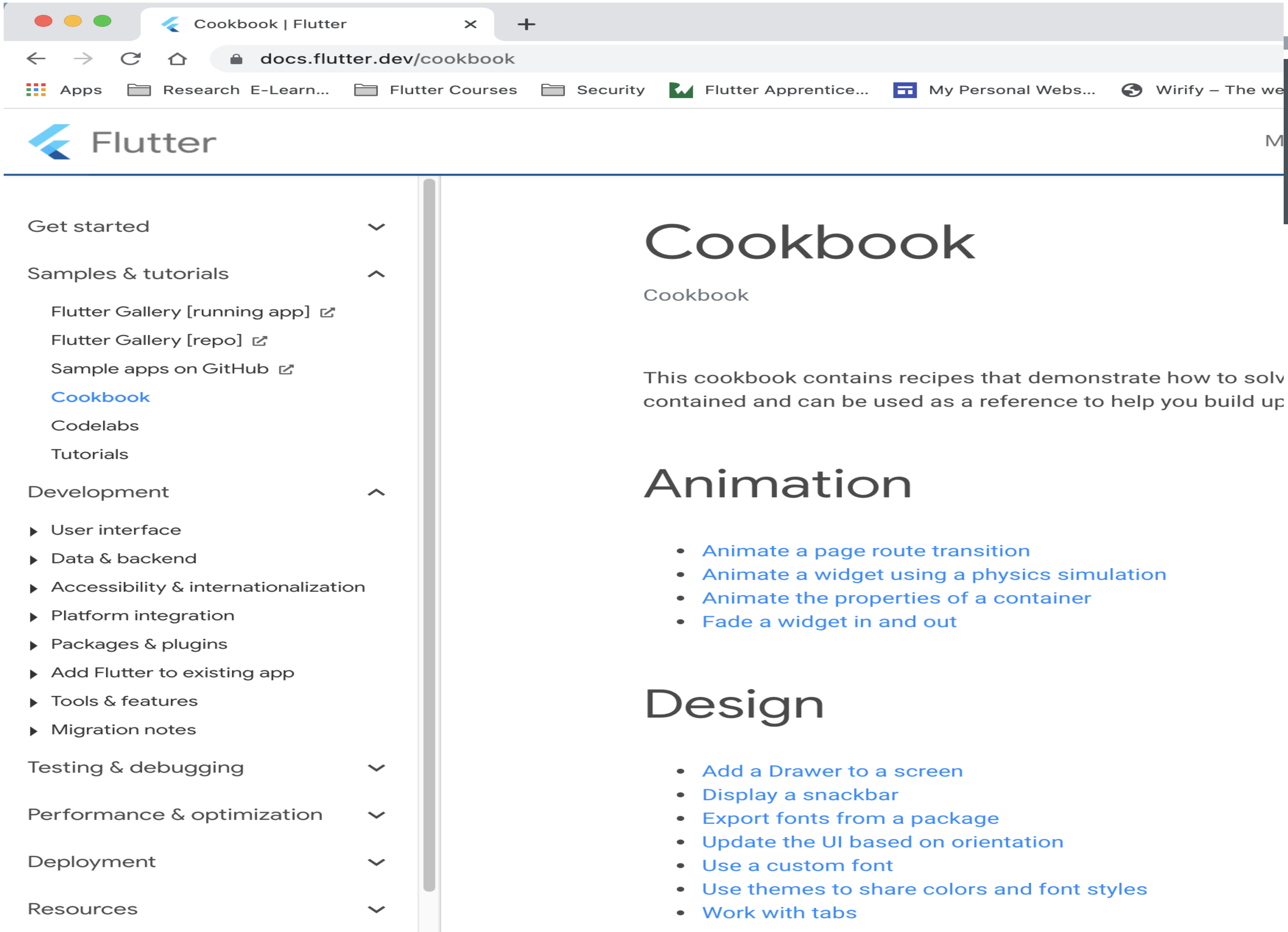
NAVIGATION IN FLUTTER

- 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.


Cookbook | Flutter

docs.flutter.dev/cookbook

Apps · Research · E-Learn... · Flutter Courses · Security · Flutter Apprentice... · My Personal Webs... · Wirify – The we

Flutter

- Get started
- Samples & tutorials
  - Flutter Gallery [running app]
  - Flutter Gallery [repo]
  - Sample apps on GitHub
  - Cookbook
  - Codelabs
  - Tutorials
- Development
  - User interface
  - Data & backend
  - Accessibility & internationalization
  - Platform integration
  - Packages & plugins
  - Add Flutter to existing app
  - Tools & features
  - Migration notes
- Testing & debugging
- Performance & optimization
- Deployment
- Resources

Cookbook

Cookbook

This cookbook contains recipes that demonstrate how to solv
contained and can be used as a reference to help you build up

Animation

- Animate a page route transition
- Animate a widget using a physics simulation
- Animate the properties of a container
- Fade a widget in and out

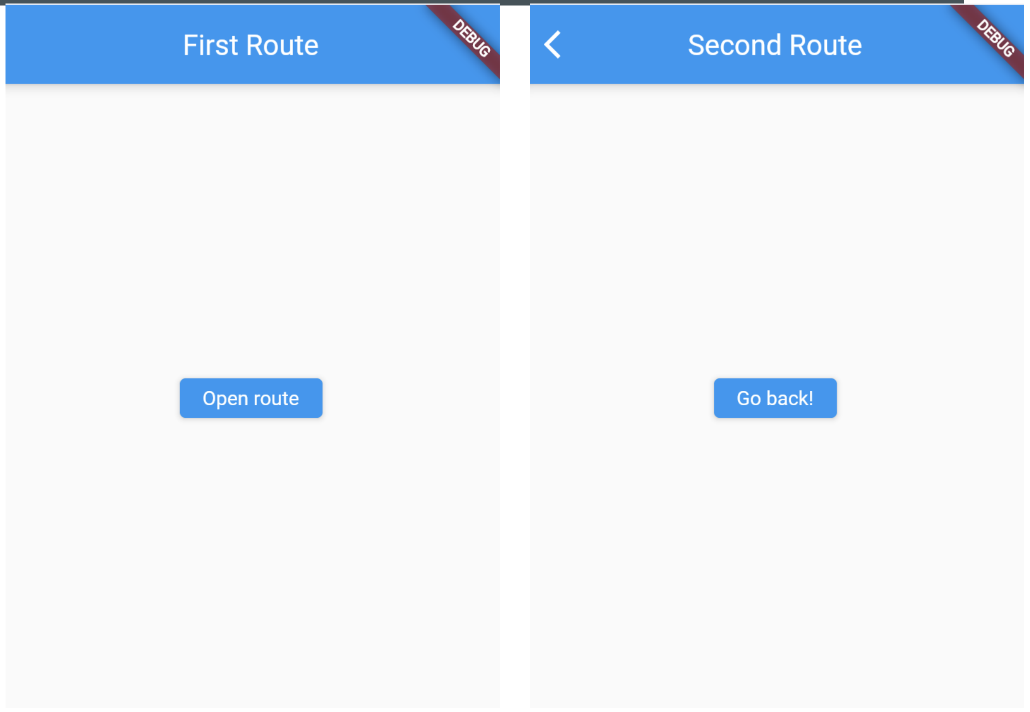
Design

- Add a Drawer to a screen
- Display a snackbar
- Export fonts from a package
- Update the UI based on orientation
- Use a custom font
- Use themes to share colors and font styles
- Work with tabs

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 example 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:

```dart
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.
          },
        ),
      ),
    );
  }
}
```

```dart
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()

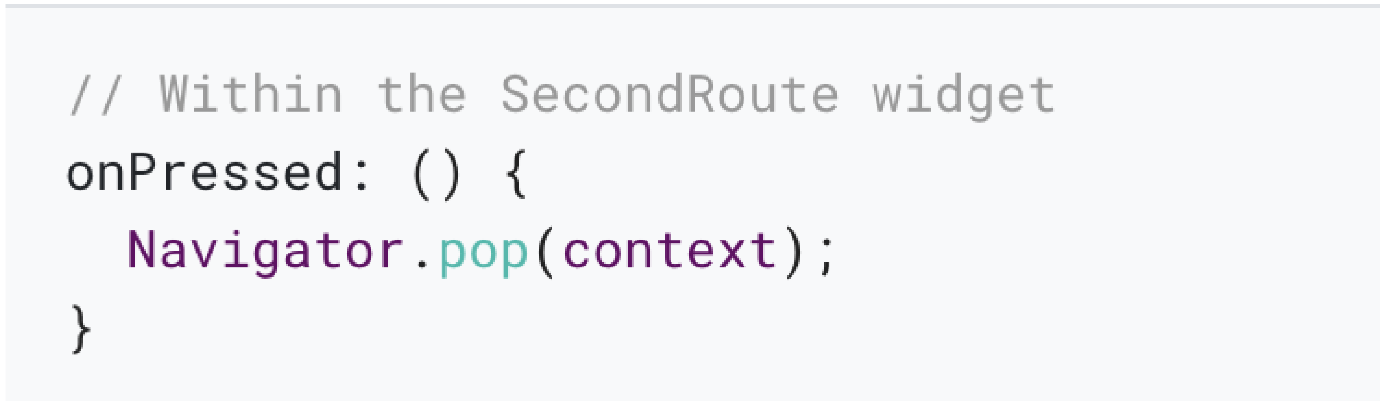
- To switch to a new route, use the **Navigator.push()** method. The push() method adds a **Route** to the stack of routes managed by the **Navigator**. Where does the Route come from? You can create your own, or use a **MaterialPageRoute,** which is useful because it transitions to the new route using 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);
}
```

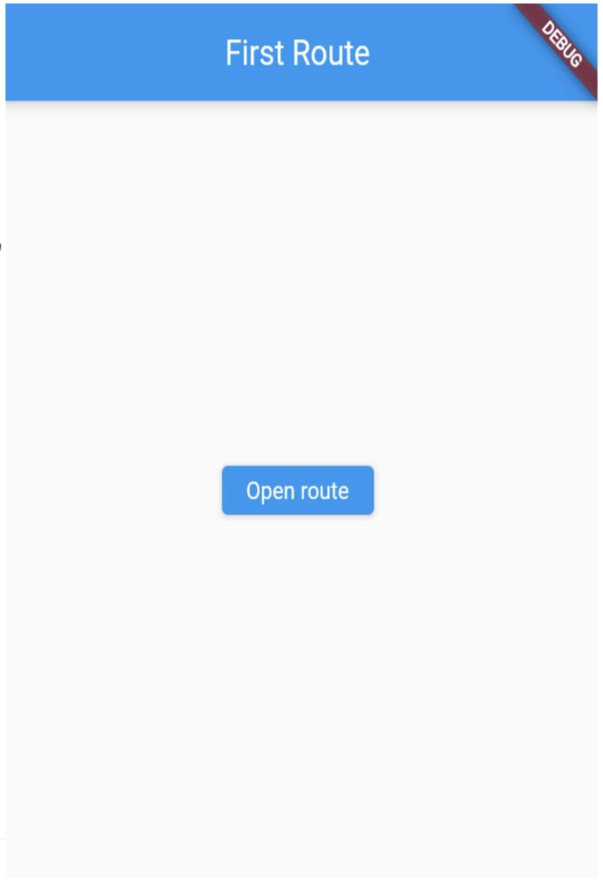
INTERACTIVE EXAMPLE

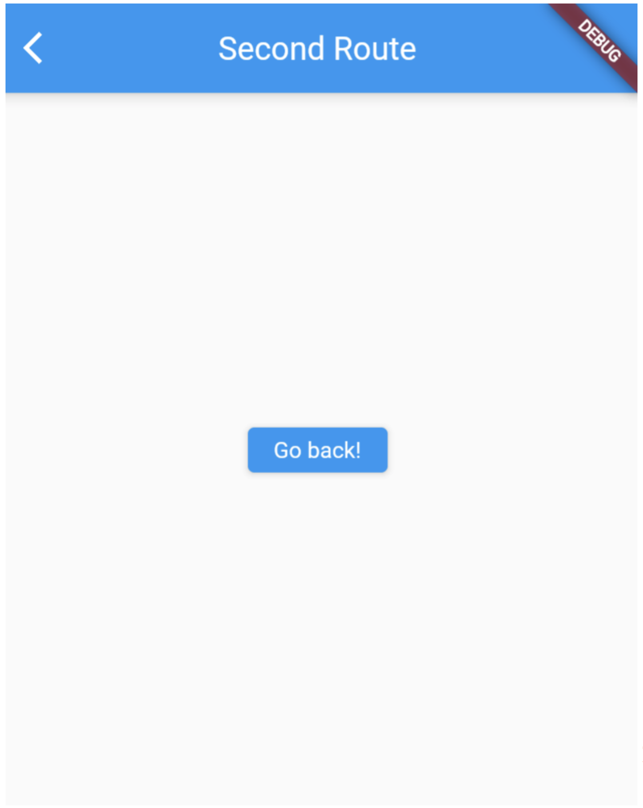
```dart
import 'package:flutter/material.dart';

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()),
            );
          },
        ),
      ),
    );
  }
}
```

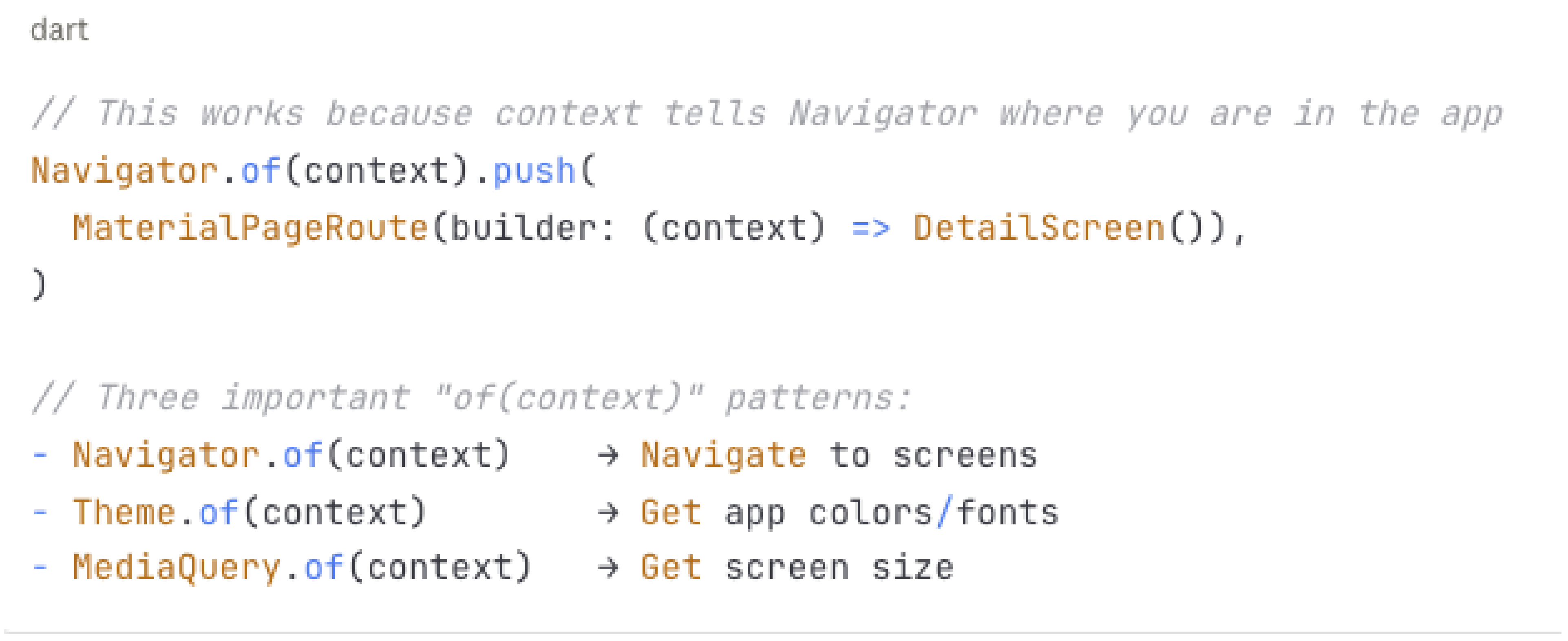
INTERACTIVE EXAMPLE

```
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!'),
        ),
      ),
    );
  }
}
```





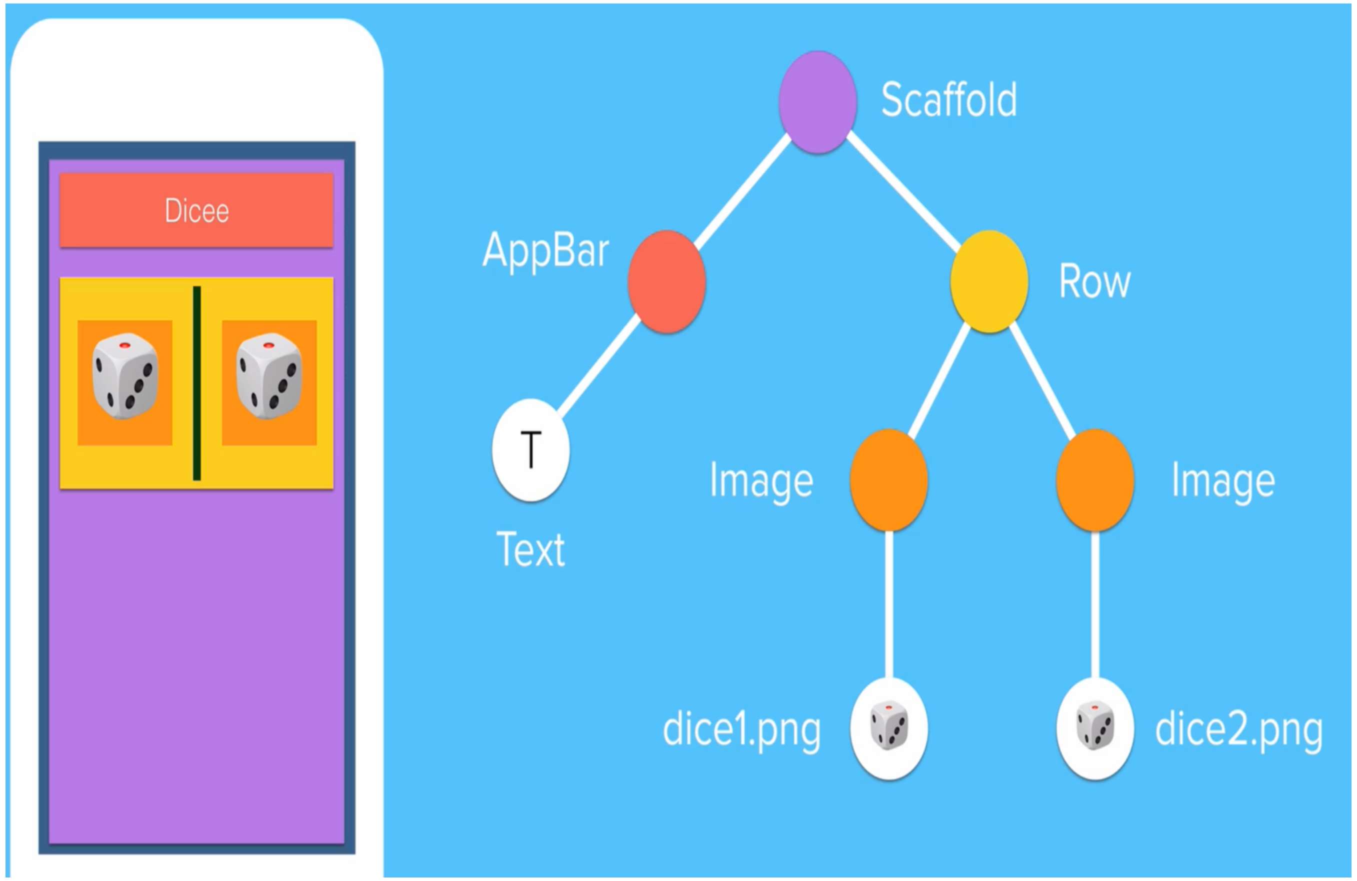
WHY NAVIGATION NEEDS CONTEXT

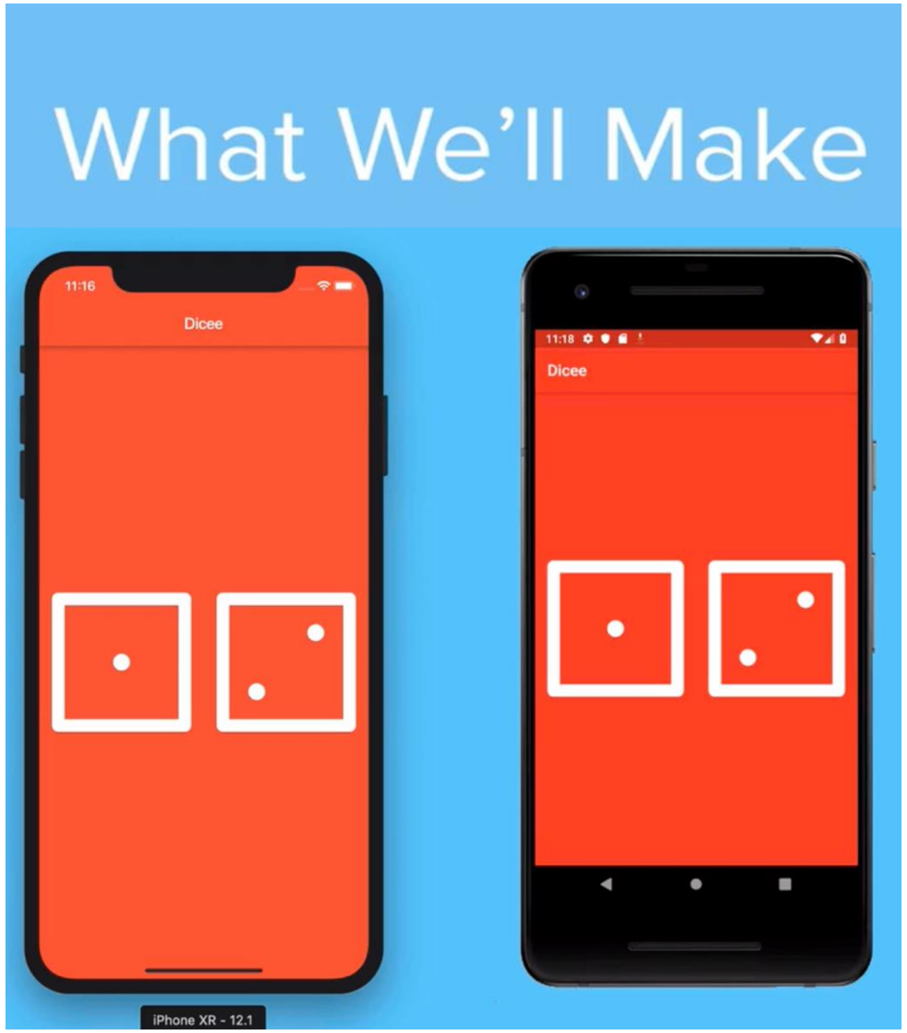


dart

```dart
// This works because context tells Navigator where you are in the app
Navigator.of(context).push(
  MaterialPageRoute(builder: (context) => DetailScreen()),
)

// Three important "of(context)" patterns:
- Navigator.of(context)     → Navigate to screens
- Theme.of(context)         → Get app colors/fonts
- MediaQuery.of(context)    → Get screen size
```

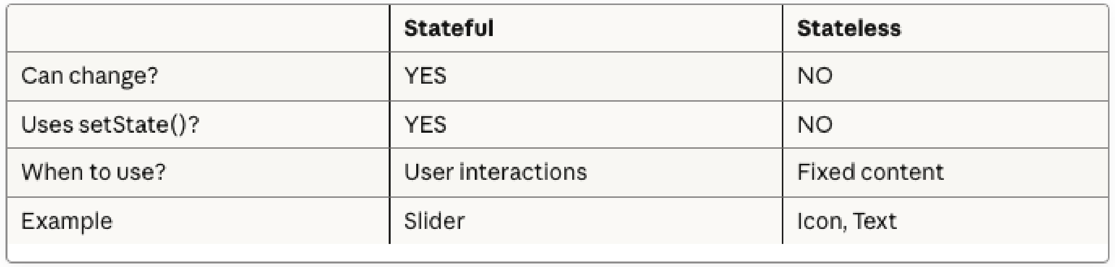
STATEFUL WIDGETS AND BUILDING AN INTERACTIVE APPLICATIONS





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	Stateless
Can change?	YES	NO
Uses setState()?	YES	NO
When to use?	User interactions	Fixed content
Example	Slider	Icon, Text

WIDGET LIFECYCLE: WHY SETSTATE() REBUILDS

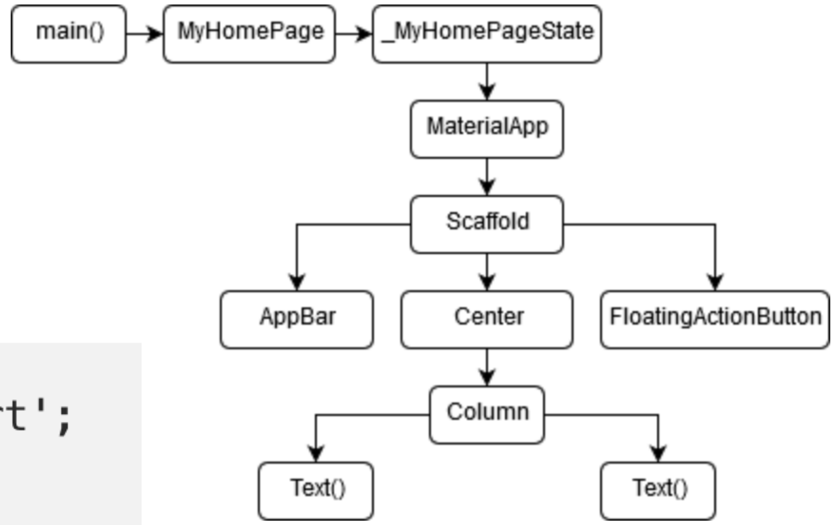
- When you call setState():
- 1. setState() tells Flutter the widget changed
- 2. Flutter calls build() again for THAT widget
- 3. Widget rebuilds and appears on screen
- 4. Parent/other widgets DON'T rebuild (efficient!)
- Example:
- - You change servings with Slider
- - Only that widget rebuilds
- - AppBar stays the same (no rebuild)
- - Rest of app stays the same (no rebuild) Key: setState() rebuilds ONLY the Stateful widget's subtree

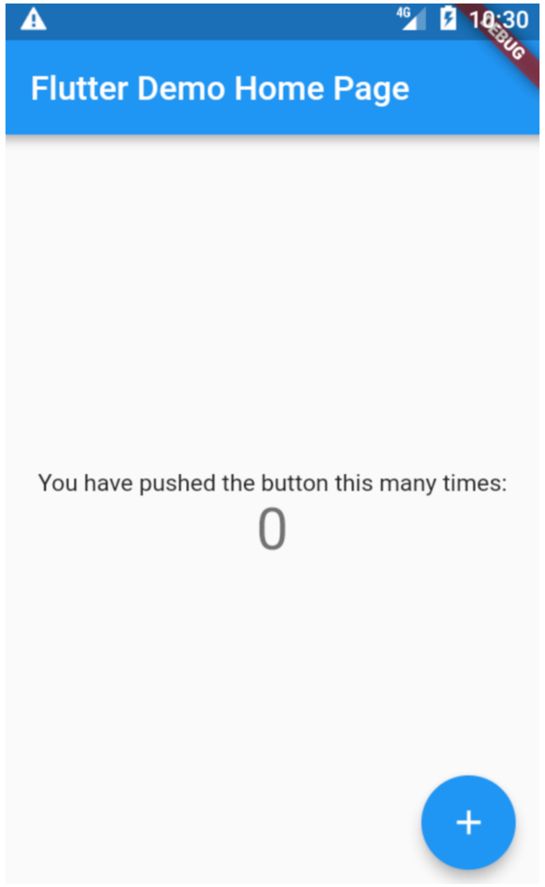
User changes state → setState() called → build() called → UI updates

STATEFUL WIDGETS AND BUILDING AN INTERACTIVE APPLICATIONS

The starting point...

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

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





You have pushed the button this many times:
0

```
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**.

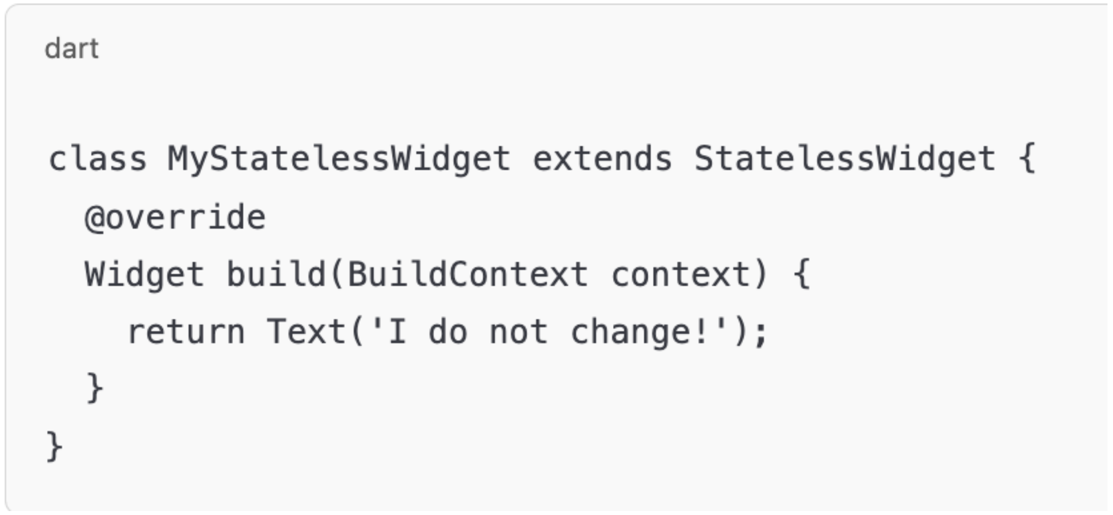
```
int _counter = 0;
void _incrementCounter() {
     setState(() {
         _counter++;
     });
}
```

So here is our full Widget tree.

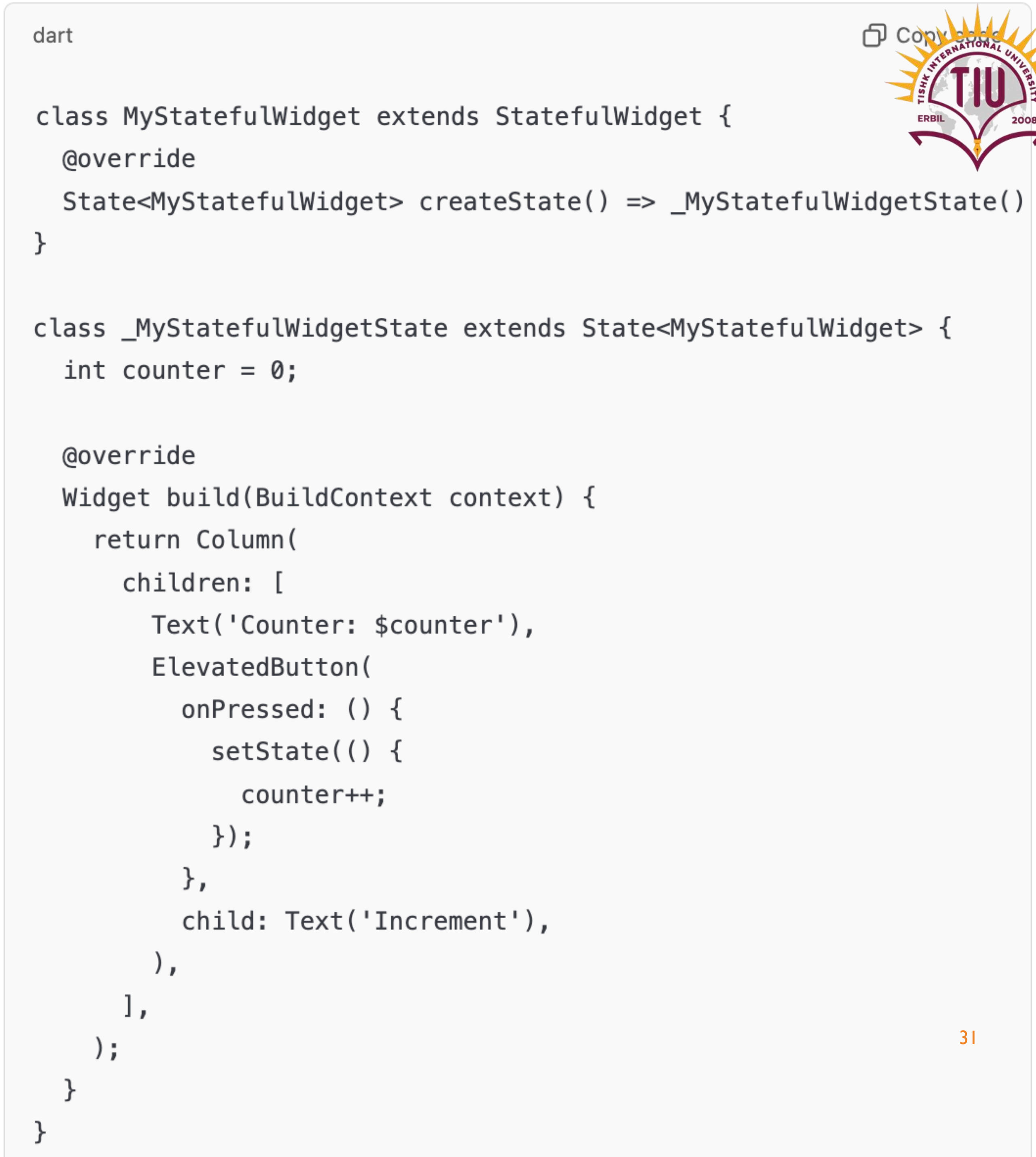
```
@override
Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Hello Flutter'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[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

STATFUL VS STATELESS WIDGETS

StatelessWidget Example:


dart

```dart
class MyStatelessWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text('I do not change!');
  }
}
```


dart Copy code

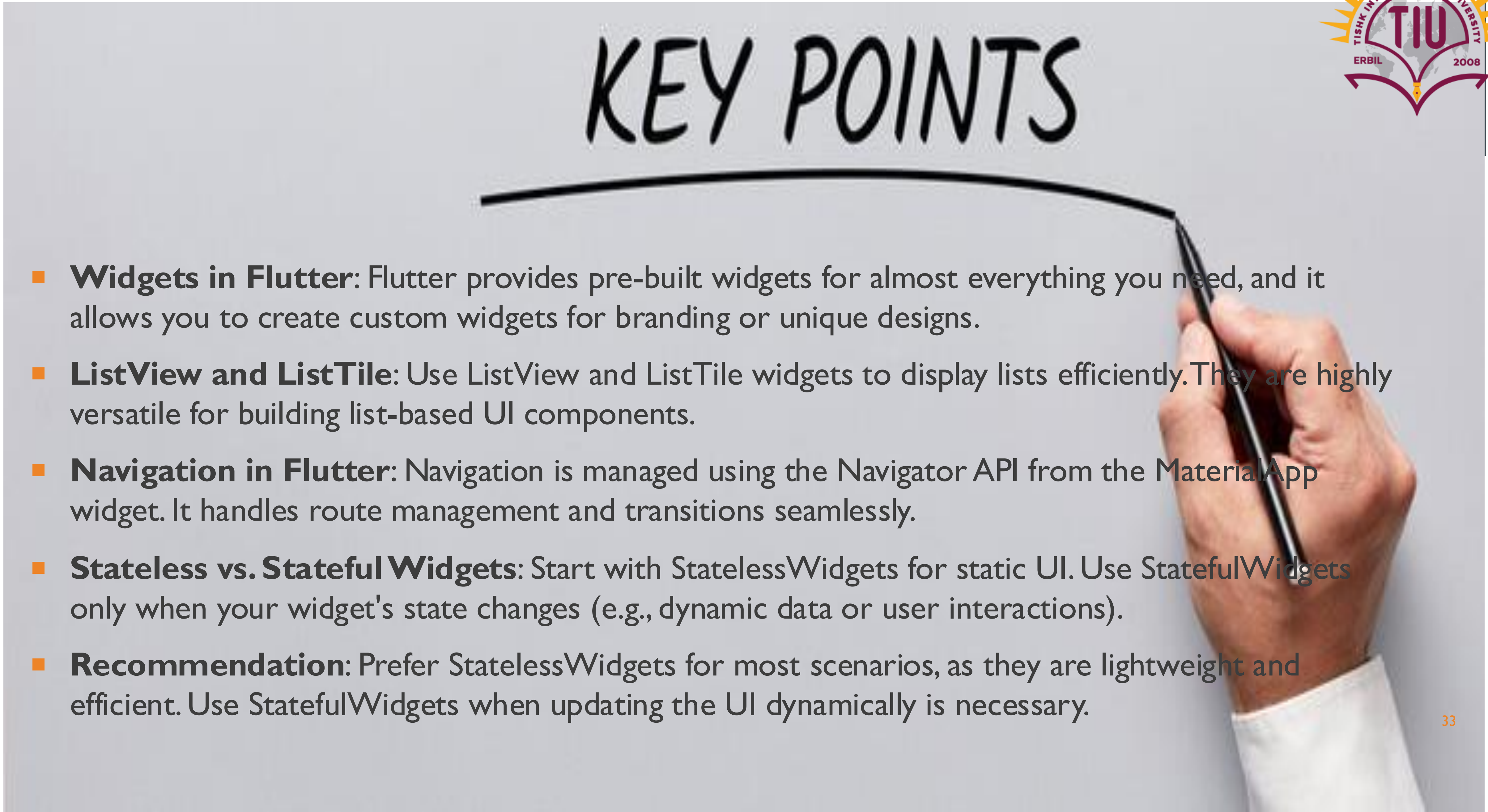
```dart
class MyStatefulWidget extends StatefulWidget {
  @override
  State<MyStatefulWidget> createState() => _MyStatefulWidgetState()
}

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  int counter = 0;

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Counter: $counter'),
        ElevatedButton(
          onPressed: () {
            setState(() {
              counter++;
            });
          },
          child: Text('Increment'),
        ),
      ],
    );
  }
}
```

SIMPLE EXERCISES

EXERCISE: Build a Recipe List App REQUIREMENTS:

- ✓ AppBar with title + 1 action button (e.g., search)
- ✓ ListView with 5 recipes (use ListView.builder)
- ✓ Extract recipe items into a custom widget
- ✓ Tap a recipe → Navigate to detail screen (new screen)
- ✓ Detail screen shows full recipe + back button

Time: 1-2 hours Submission: Working Flutter project



- **Widgets in Flutter**: Flutter provides pre-built widgets for almost everything you need, and it allows you to create custom widgets for branding or unique designs.
- **ListView and ListTile**: Use ListView and ListTile widgets to display lists efficiently. They are highly versatile for building list-based UI components.
- **Navigation in Flutter**: Navigation is managed using the Navigator API from the MaterialApp widget. It handles route management and transitions seamlessly.
- **Stateless vs. Stateful Widgets**: Start with StatelessWidgets for static UI. Use StatefulWidgets only when your widget's state changes (e.g., dynamic data or user interactions).
- **Recommendation**: Prefer StatelessWidgets for most scenarios, as they are lightweight and efficient. Use StatefulWidgets when updating the UI dynamically is necessary.