



# **222301504 Cross Platform Mobile Application Development**


# Introduction to Widget Tree

- The widget tree is the root of the widget hierarchy. It is created by the `runApp()` function and contains all the widgets that make up the app's UI.
- An element tree is a representation of the widget tree at a specific point in time. It is created by the Flutter framework when the widget tree is built.
- The render object tree is a representation of the widget tree. The widget tree, which serves as the foundation for the app's user interface, is used to construct the element tree and render object tree.



## **The following are the widgets (usable only with Material Design) that you'll use to create the full and shallow widget tree**

- Scaffold —Implements the Material Design visual layout, allowing the use of Flutter's Material Components widgets
- AppBar —Implements the toolbar at the top of the screen
- CircleAvatar —Usually used to show a rounded user profile photo, but you can use it for any image
- Divider —Draws a horizontal line with padding above and below

- 
- Cupertino you can use two different scaffolds, a page scaffold or a tab scaffold.
  - CupertinoPageScaffold —Implements the iOS visual layout for a page. It works with CupertinoNavigationBar to provide the use of Flutter's Cupertino iOS-style widgets.
  - CupertinoTabScaffold —Implements the iOS visual layout. This is used to navigate multiple pages, with the tabs at the bottom of the screen allowing you to use Flutter's Cupertino iOS-style widgets.
  - CupertinoNavigationBar —Implements the iOS visual layout toolbar at the top of the screen.

# Material Design vs. Cupertino Widgets

MATERIAL DESIGN	CUPERTINO
Scaffold	CupertinoPageScaffold CupertinoTabScaffold
AppBar	CupertinoNavigationBar
CircleAvatar	n/a
Divider	n/a

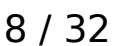
# The following widgets can be used with both Material Design and Cupertino:

- `SingleChildScrollView` —This adds vertical or horizontal scrolling ability to a single child widget.
- `Padding` —This adds left, top, right, and bottom padding.
- `Column` —This displays a vertical list of child widgets.
- `Row` —This displays a horizontal list of child widgets.
- `Container` —This widget can be used as an empty placeholder (invisible) or can specify height, width, color, transform (rotate, move, skew), and many more properties
- `Expanded` —This expands and fills the available space for the child widget that belongs to a `Column` or `Row` widget.
- `Text` —The `Text` widget is a great way to display labels on the screen.
- `Stack` —What a powerful widget! `Stack` lets you stack widgets on top of each other and use a `Positioned` (optional) widget to align each child of the `Stack` for the layout needed.



# BUILDING THE FULL WIDGET TREE

- Need to explain the code
- a combination of Column , Row , Container , CircleAvatar , Divider , Padding , and Text widgets.







# BUILDING A SHALLOW WIDGET TREE

- To make the example code more readable and maintainable, you'll refactor major sections of the code into separate entities. You have multiple refactor options, and the most common techniques are constants, methods, and widget classes.

# Refactoring with a Constant

Refactoring with a constant initializes the widget to a `final` variable. This approach allows you to separate widgets into sections, making for better code readability. When widgets are initialized with a constant, they rely on the `BuildContext` object of the parent widget.

What does this mean? Every time the parent widget is redrawn, all the constants will also redraw their widgets, so you can't do any performance optimization. In the next section, you'll take a detailed look at refactoring with a method instead of a constant. The benefits of making the widget tree shallower are similar with both techniques.

The following sample code shows how to use a constant to initialize the `container` variable as `final` with the `Container` widget. You insert the `container` variable in the widget tree where needed.

```
final container = Container(  
  color: Colors.yellow,  
  height: 40.0,  
  width: 40.0,  
);
```

# Refactoring with a Method

Refactoring with a method returns the widget by calling the method name. The method can return a value by a general widget (`Widget`) or a specific widget (`Container`, `Row`, and others).

The widgets initialized by a method rely on the `BuildContext` object of the parent widget. There could be unwanted side effects if these kinds of methods are nested and call other nested methods/functions. Since each situation is different, do not assume that using methods is not a good choice. This approach allows you to separate widgets into sections, making for better code readability. However, like when refactoring with a constant, every time the parent widget is redrawn, all the methods will also redraw their widgets. That means the widget tree is not optimizable for performance.


The following sample code shows how to use a method to return a `Container` widget. This first method returns the `Container` widget as a general `Widget`, and the second method returns the `Container` widget as a `Container` widget. Both approaches are acceptable. You insert the `_buildContainer()` method name in the widget tree where needed.

```
// Return by general Widget Name
Widget _buildContainer() {
  return Container(
    color: Colors.yellow,
    height: 40.0,
    width: 40.0,
  );
}

// Or Return by specific Widget like Container in this case
Container _buildContainer() {
  return Container(
    color: Colors.yellow,
    height: 40.0,
```

# Refactoring with a Widget Class

- Refactoring with a widget class allows you to create the widget by subclassing the StatelessWidget class.
- You can create reusable widgets within the current or separate Dart file and initiate them anywhere in the application. Notice that the constructor starts with a const keyword, which allows you to cache and reuse the widget. When calling the constructor to initiate the widget, use the const keyword.
- By calling with the const keyword, the widget does not rebuild when other widgets change their state in the tree. If you omit the const keyword, the widget will be called every time the parent widget redraws.



```
class ContainerLeft extends StatelessWidget {  
  const ContainerLeft({  
    Key key,  
  }) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    return Container(  
      color: Colors.yellow,  
      height: 40.0,  
      width: 40.0,  
    );  
  }  
}
```



# Flutter TextField

- TextField is used to get text input from user.
- The default behavior of TextField is that, when you press on it, it gets focus and a keyboard slides from the bottom of the screen.
- When you enter text using keyboard, the string is displayed in the TextField.

```

import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}
class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}
class _MyAppState extends State<MyApp> {
  TextEditingController nameController = TextEditingController();
  String fullName = "";
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter'),
        ),
        body: Center(child: Column(children: <Widget>[
          Container(
            margin: EdgeInsets.all(20),
            child: TextField(
              controller: nameController,
              decoration: InputDecoration(
                border: OutlineInputBorder(),
                labelText: 'Full Name',
              ),
              onChanged: (text) {
                setState(() {
                  fullName = text;
                });
              },
            ),
          ),
          Container(
            margin: EdgeInsets.all(20),
            child: Text(fullName),
          ),
        ])),
      );
  }
}

```

# Flutter - FloatingActionButton

- A floating action button is a circular icon button that hovers over content to promote a primary action in the application.

```
floatingActionButtonLocation: FloatingActionButtonLocation.cente
floatingActionButton: FloatingActionButton(
  // isExtended: true,
  child: Icon(Icons.add),
  backgroundColor: Colors.green,
  onPressed: () {
    setState(() {
      i++;
    });
  },
),
```





# Flat Button

- FlatButton is usually used to display buttons that lead to secondary functionalities of the application like viewing all files of Gallery, opening Camera, changing permissions etc.
- FlatButton does not have an elevation unlike Raised Button. Also, by default, there is no color to the button and text is black.

```

import 'package:flutter/material.dart';

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

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

class _MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter FlatButton - tutorialkart.com'),
        ),
        body: Center(child: Column(children: <Widget>[
          Container(
            margin: EdgeInsets.all(20),
            child: FlatButton(
              child: Text('Login'),
              onPressed: () {},
            ),
          ),
          Container(
            margin: EdgeInsets.all(20),
            child: FlatButton(
              child: Text('Login'),
              color: Colors.blueAccent,
              textColor: Colors.white,
              onPressed: () {},
            ),
          ),
        ])),
      );
  }
}

```

# RaisedButton

- RaisedButton widget is a material design concept of a button with elevation. It provides dimensionality to your UI along z-axis with clues of shadow.
- You can set many properties of RaisedButton like text color, button color, color of button when disabled, animation time, shape, elevation, padding, etc.
- You can also disable the button using enabled property.
- There are callback functions:
  - onPressed() is triggered when user presses this button.
  - onLongPress() is triggered when user long presses on this button.

```

class _MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter RaisedButton - tutorialkart.com'),
        ),
        body: Center(
          child: Column(children: <Widget>[
            Text(
              'Raised Buttons with Different Properties',
              style: TextStyle(fontSize: 16),
            ),
            RaisedButton(child: Text('Disabled Button')),
            RaisedButton(
              child: Text('Default Enabled'),
              onPressed: () {},
            ),
            RaisedButton(
              child: Text('Text Color Changed'),
              textColor: Colors.red,
              onPressed: () {},
            ),
            RaisedButton(
              child: Text('Color Changed'),
              color: Colors.green,
              onPressed: () {},
            ),
            RaisedButton(
              child: Text('Button with Padding'),
              padding: EdgeInsets.all(20),
              onPressed: () {},
            ),

```

```

            RaisedButton(
              child: Text('More Rounded Corners'),
              color: Colors.purpleAccent,
              shape: RoundedRectangleBorder(
                borderRadius: BorderRadius.all(Radius.circular(16.0))),
              onPressed: () {},
            ),
            RaisedButton(
              child: Text('Elevation increased'),
              elevation: 5,
              onPressed: () {},
            ),
            RaisedButton(
              child: Text('Splash Color as red'),
              splashColor: Colors.red,
              onPressed: () {},
            ),
            RaisedButton(
              child: Text('Zero Elevation'),
              elevation: 0,
              onPressed: () {},
            ),
            RaisedButton(
              onPressed: () {},
              textColor: Colors.white,
              padding: const EdgeInsets.all(0.0),
              child: Container(
                decoration: const BoxDecoration(
                  gradient: LinearGradient(
                    colors: <Color>[
                      Color(0xFF0D47A1),
                      Color(0xFF1976D2),
                      Color(0xFF42A5F5),
                    ],

```



# IconButton


- Flutter IconButton acts just like a button, but with an icon instead of an usual button.
- You can execute a set of statements when the IconButton is pressed using onPressed property. Also, you get the animations like splash when you click this IconButton, just like a regular button.
- If you do not specify onPressed property (not even null), the IconButton is displayed as disabled button.

```
class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('TutorialKart - Flutter IconButton'),
      ),
      body: Column(children: <Widget>[
        Container(
          padding: EdgeInsets.all(50),
          alignment: Alignment.center,
          child: IconButton(
            icon: Icon(
              Icons.directions_transit,
            ),
            iconSize: 50,
            color: Colors.green,
            splashColor: Colors.purple,
            onPressed: () {},
          ),
        ),
      ]),
    );
  }
}
```



# PopupMenuButton

- Displays a menu when pressed and calls `onSelected` when the menu is dismissed because an item was selected.
- The value passed to `onSelected` is the value of the selected menu item.
- If we focus on an Application, We can see in every Application there is a Pop Up Menu button that will do some work. So in this article, we will implement the pop-up menu button.



```


home: Scaffold(
  // appbar with title text
  appBar: AppBar(
    title: Text('AppBar Popup Menu Button'),
    // in action widget we have PopupMenuButton
    actions: [
      PopupMenuButton<int>(
        itemBuilder: (context) => [
          // PopupMenuItem 1
          PopupMenuItem(
            value: 1,
            // row with 2 children
            child: Row(
              children: [
                Icon(Icons.star),
                SizedBox(
                  width: 10,
                ),
                Text("Get The App")
              ],
            ),
          ],
        ),
      ],
    ),
  ),

```

```

// PopupMenuItem 2
PopupMenuitem(
  value: 2,
  // row with two children
  child: Row(
    children: [
      Icon(Icons.chrome_reader_mode),
      SizedBox(
        width: 10,
      ),
      Text("About")
    ],
  ),
),

```








# ButtonBar

- The ButtonBar widget is basically a container that arranges a group of buttons in a horizontal manner. This widget makes creating a row of buttons easier without hassling over additional alignment and padding issues.
- The ButtonBar widget has various applications, but it is mainly used in situations where multiple buttons are needed to be displayed together in a uniform manner, such as in a toolbar or a form.

- 
- The ButtonBar widget proposes a number of various properties for modification according to the user's needs. These properties can be specified by you in any manner you want the widget to perform and look
  - Some of these properties include:
    - Alignment
    - Button height
    - Padding
    - Children
    - Main Axis Size

- Example at:  
<https://www.educative.io/answers/how-to-use-buttonbar-in-flutter>

```
ButtonBar(  
  alignment: // MainAxisAlignment value,  
  mainAxisSize: // MainAxisSize value,  
  children: <Widget>[  
    // Buttons or other widgets  
  ],  
)
```

# Image

- Flutter Image widget displays an image in our Flutter Application.
- ```
const Image(  
  image:  
    NetworkImage('https://www.abc.com/img/lion  
    .jpg'),  
)
```

```
class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(
        children: <Widget>[
          Container(
            margin: const EdgeInsets.all(20),
            child: const Image(
              image: NetworkImage(
                'https://www.tutorialkart.com/img/lion.jpg')
            ),
          ),
        ],
      ),
    );
  }
}
```



# Flutter Icon

- Icons can be used as a representative symbol for a quick understanding of the functionality, or path of the navigation, etc.
- Following are the list of examples we shall discuss.
- Basic Icon widget example, with default values.
- Change Icon Size using size property.
- Change Icon Color using color property.
- Icon with a Text Label below it with the help of Column widget



```
class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('TutorialKart - Icon Tutorial'),
      ),
      body: Column(children: <Widget>[
        Center(child: Icon(Icons.directions_transit)),
      ]),
    );
  }
}
```



# More Examples at

- <https://www.tutorialkart.com/flutter/flutter-appbar/#gsc.tab=0>
- <https://www.educative.io>