A Beginner to Pro. Learn how to build Advanced Flutter Apps

BEGINNING FLUTTER 3.0 WITH DART

SANJIB SINHA

Beginning Flutter 3.0 with Dart

A Beginner to Pro. Learn how to build Advanced Flutter 3.0 Apps

Sanjib Sinha

This book is for sale at http://leanpub.com/beginningflutterwithdart

This version was published on 2022-07-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

15. User Interface, Style, Theme and App Design

It really doesn't make any sense if our flutter app does not look good.

Therefore, we must know what are the basic guidelines and concepts behind flutter app designs.

How we can create a custom theme?

How we can style the user interface, so that it looks stunning?

This section is dedicated to in depth study of those design related concepts.

What are constraints in flutter

As a beginner at the very beginning we need to know what are constraints in Flutter .

When you plan to learn Flutter, it starts with Layout. Right? Now, you cannot learn or understand flutter layout without understanding constraints. Therefore, our flutter learning starts by answering this question first – what are constraints in flutter?

Firstly, let me warn you at the very beginning. Flutter layout is not like HTML layout.

Secondly, if you come from HTML or web development background, don't try to apply those CSS rules here. Why so? Because, HTML targets a large screen. Whereas, we're dealing with a Mobile screen. So, layout should not be same. And, there are other reasons too.

Finally, flutter is all about widgets. As a result, we need to understand Flutter layout keeping widgets in our mind.

Let's start with a Material App, and, start building a simple Container widget with a Text widget as its child.

```
1  import 'package:flutter/material.dart';
2
3  class ConstraintSample extends StatelessWidget {
4  const ConstraintSample({Key? key}) : super(key: key);
5
6  @override
7  Widget build(BuildContext context) {
8      return const MaterialApp(
9      title: 'Constraint Sample',
```

```
    debugShowCheckedModeBanner: false,
    home: ConstraintSampleHomme(),
    );
}
}
@override
Widget build(BuildContext context) {
    return Container(
    width: 150,
    height: 200,
    child: const Text('Constraint Sample'),
    );
}
```

Let's run this simple flutter app, and see what happens.

What is the problem here?

We've returned a Container widget mentioning its width and height.

```
return Container(
    width: 150,
    height: 200,
    child: const Text('Constraint Sample'),
    );
```

However, that didn't work at all.

Now we know that the constraint in Flutter is all about the size of the box, which is nothing but a widget.

A size always deals with width and height.

In the above case, the material app takes the width and height of the whole screen and directs its immediate child Container to take that size.

That is why, although we've mentioned the size of the Container widget, it doesn't work.

Therefore, we can conclude that any Widget gets its constraint or size from its immediate parent. After that, it passes that constraint to its immediate child.

And this practice goes on as the number of Widgets increases in the widget tree.

In Flutter, a parent widget always controls the immediate child's size. However, when the parent becomes grand-parent, it cannot affect the constraint or size of the grand-child.

Why?

Because, the grand child has its parent that inherits the size from its parent and decides what should be the size of its child.

We can compare this mechanism with wealth. If a person inherits some wealth from her parent, she is in a position to decide how much of that wealth she will give to her child or children.

And this process goes on.

One after another widget tells its children what their constraints or sizes are.

How do you use constraints in flutter?

Since we have got the idea, now we can apply and see how we can use constraints in flutter.

Our next code snippet is like the following.

```
@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        width: 300,
        height: 100,
        color: Colors.amber,
        alignment: Alignment.bottomCenter,
        child: const Text(
        'Constraint Sample',
        style: TextStyle(
            fontSize: 25,
            fontWeight: FontWeight.bold,
            color: Colors.black,
        ),
        ),

    ),
    );
}
```

Now, we've wrapped the Container widget with a Center widget. As a result, the Center widget gets the full size now. And, after that, it asks immediate child Container widget how big it wants to be.

The Container said, "I want to be 300 in width and 100 in height. Not only that, I want to place my child at the bottom Center position. And, I also want my child should be of color amber."

The Center widget says, "Okay. No problem. Get what you want because I have inherited the whole screen-size from my parent. Take what you need."

As a result, run the code and see the effect on the screen.

Next, we change our code to this:

```
@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        width: 300,
        height: 100,
        color: Colors.amber,
        alignment: Alignment.bottomCenter,
        child: Container(
        width: 200,
        height: 50,
        color: Colors.blue,
        alignment: Alignment.center,
        child: const Text(
            'Constraint Sample',
            style: TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.bold,
            color: Colors.white,
            ),
        ),
        ),
    ),
    );
}
```

Let's see the effect on the screen first. Then we'll discuss the code.

The above image is displayed because the code says that the child Container with width 300, height 100 and color amber has a child, which is another Container and that has color blue, width 200 and height 50. However, the parent Container decides that it will show its child in the bottom Center position.

Align the child at the bottom Center means, we would pass this child Container a tight constraint that is bigger than the child's natural size, with an alignment of Alignment.bottomCenter.

As a result, the child Container gets the width 200 and height 50 and is placed at the bottom Center alignment. It happens smoothly because the parent Container's width and height is bigger than the child Container. Therefore, it allocates the exact size that it's asked for.

But it didn't happen, if the parent Container didn't set the alignment and said that, "Okay, child container, you can place yourself anywhere you like. Even you may decide your alignment."

So the code is like the following:

```
@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        width: 300,
        height: 100,
        color: Colors.amber,
        child: Container(
        width: 200,
        height: 50,
        color: Colors.blue,
        alignment: Alignment.center,
        child: const Text(
            'Constraint Sample',
            style: TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.bold,
            color: Colors.white,
            ),
        ),
        ),
    ),
    );
}
```

As a result, the child Container decides to looks upward and tries to find if its grand-parent has any alignment already allocated for it.

While looking upward, it finds that the grand-parent is a Center widget itself. Therefore, it decides to take the Center position, as it has the Center alignment itself; and not only that, while doing so, it ignores its own width and height, and it takes the width and height of the immediate parent, which eventually is another Container.

As a result it overlaps completely the parent Container.

To sum up, constraints are basically sizes of width and height that any Widget gets from its parent. However, in case, padding is added, the constraint may change. Although we have not added that feature, still you may test that on your own and see how it affects the sizes of the child.

What are BoxConstraints in Flutter

Imagine each Widget as a box. So it has a size or constraints that defines width and height.

In the previous section we have discussed what constraints are. In Flutter, every widget is rendered by their underlying RenderBox objects. As a result, for each boxes constraints are BoxConstraints.

For a Flutter beginner we need to say one thing at the very beginning. The constraints actually represent sizes of the rendered boxes, which are nothing but widgets.

In that light of the previous discussion, we must try to understand what BoxConstraints are.

We've already seen that widgets pass their constraints, which consist of minimum and maximum width and height, to their children. Moreover, each child may vary in size.

As a result we can say that render tree actually passes a concrete geometry, which is size.

Subsequently the widget tree grows in sizes and for each boxes the constraints are BoxConstraints. And, it consists of four numbers – a minimum width minWidth, a maximum width maxWidth, a minimum height minHeight, and a maximum height maxHeight. Therefore we can set a range of width and height.

As we've said before, the geometry of boxes consists of a Size. Consequently, the Size satisfies the constraints.

Each child in the rendered widget tree, gets BoxConstraints from its parent. After that, the child picks us the size that satisfies the BoxConstraints adjusting with the parent's size.

Certainly, with the size, position changes. A child does not know its position.

Why?

Because, if the parent adds some padding the child's position changes with it.

We'll see that in a minute.

Consider a Flutter app where Scaffold widget acts as the immediate child of Material App widget.

As a result, the Scaffold takes the entire screen from its immediate parent Material App.

Next, the Scaffold passes its constraints to its immediate child Center widget. And then the Center takes the constraints or size from Scaffold. However, as the Scaffold widget allocates some space for the App Bar widget, therefore, Center doesn't get the whole screen.

Let us see the code.

```
1  import 'package:flutter/material.dart';
2
3  class BoxConstraintsSample extends StatelessWidget {
4  const BoxConstraintsSample({Key? key}) : super(key: key);
5
6  @override
7  Widget build(BuildContext context) {
8      return const MaterialApp(
9      title: 'BoxConstraints Sample',
```

```
    debugShowCheckedModeBanner: false,
    home: BoxConstraintsSampleHomme(),
    );
}
}

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

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('BoxConstraints Sample'),
    ),
    body: Center(
        child: Container(
        color: Colors.redAccent,
        padding: const EdgeInsets.all(
            20,
        ),
        child: const Text(
            'Box',
            style: TextStyle(
            fontFamily: 'Allison',
            color: Colors.black38,
            fontSize: 60,
            fontWeight: FontWeight.bold,
            ),
        ),
        constraints: const BoxConstraints(
            minHeight: 70,
            minWidth: 70,
            maxHeight: 200,
            maxWidth: 200,
        ),
        ),
    ),//Center
    );
}
}
```

If we run the code, we'll see the following effect.

The above code tells us about the Container's constraints that point to BoxConstraints, this way.

```
constraints: const BoxConstraints(
            minHeight: 70,
            minWidth: 70,
            maxHeight: 200,
            maxWidth: 200,
        ),
```

The Container widget has a constraints parameter that points to the BoxConstraints widget, which simply defines the minimum and maximum width, height. And the range is between 70px to 200px.

As a result, if we try to make the Text widget bigger, that will not display the whole text, in that case.

Why?

Because, Container passes its constraints to its child Text widget and it gets that exact value. That means, the size of Text widget must remain in between 70px to 200px.

What happens if we try to pass the same constraints to another Container, which will act as the immediate child.

Let's change our code.

How do you use box constraints in Flutter?

To use box constraints in Flutter, we must understand how BoxConstraints widget acts, maintaining the range of width and height.

Let's change our above code and it will look like the following, now.

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

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

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'BoxConstraints Sample',
    debugShowCheckedModeBanner: false,
    home: BoxConstraintsSampleHomme(),
    );
}
}
```

```
15
16 class BoxConstraintsSampleHomme extends StatelessWidget {
17 const BoxConstraintsSampleHomme({Key? key}) : super(key: key);
18
19 @override
20 Widget build(BuildContext context) {
21     return Scaffold(
22     appBar: AppBar(
23         title: const Text('BoxConstraints Sample'),
24     ),
25     body: Center(
26         child: Container(
27         color: Colors.redAccent,
28         padding: const EdgeInsets.all(
29             20,
30         ),
31         constraints: const BoxConstraints(
32             minHeight: 170,
33             minWidth: 170,
34             maxHeight: 400,
35             maxWidth: 400,
36         ),
37         child: Container(
38             color: Colors.blueAccent[200],
39             padding: const EdgeInsets.all(
40             20,
41             ),
42             child: const Text(
43             'Box',
44             style: TextStyle(
45                 fontFamily: 'Allison',
46                 color: Colors.white,
47                 fontSize: 60,
48                 fontWeight: FontWeight.bold,
49             ),
50             ),
51             constraints: const BoxConstraints.expand(
52             height: 100,
53             width: 100,
54             ),
55         ),
56         ), //container
57     ), //Center
```

```
    );
}
}
```

In the above code, we find two Container widgets. Both have constraints defined. The first Container have constraints like the following:

```
constraints: const BoxConstraints(
            minHeight: 170,
            minWidth: 170,
            maxHeight: 400,
            maxWidth: 400,
        ),
```

And its child, the second Container has constraints like the following:

```
constraints: const BoxConstraints.expand(
            height: 100,
            width: 100,
            ),
```

The first Container's constraints define a range of width and height. However,the second Container's constraints point to BoxConstraints constructor BoxConstraints.expand.

What does this mean, as long as the size of the child Container is concerned?

We can explain it this way.

Since the child Container receives its constraints from its parent Container. Within that range it can expand its width and height up to 100px. Not more than that.

Moreover, this expansion process starts from the Center.

Understanding how these widgets or boxes handle the constraints in flutter is very important for the beginners.

In general, there are three kind of boxes that we'll encounter while we learn Flutter.

Firstly, the widgets like Center and ListView; they always try to be as big as possible.

Secondly, the widgets like Transform and Opacity always try to take the same size as their children.

And, finally, there are widgets like Image and Text that try to fit to a particular size.

As we'll progress, we'll find, how these constraints vary from widget to widget.

As example, we can remember the role of Center widget. It always maintains the maximum size. The minimum constraint does not work here.

What is widget in Flutter

In Flutter everything is Widget. But in reality it's a class and creates its instances also.

Firstly, widget in flutter is a class that describes how our flutter app should look like by creating its instances.

Secondly, the central idea behind using widgets is to build our user interface using widgets. To clarify, widgets are like boxes on the mobile, tab or desktop screen. Consequently, since each box has a size, every widget has constraints that deal with width and height.

Therefore, finally, we can define a widget as a class that builds or rebuilds its description; and, our flutter app works on that principle.

For a beginner, we need to add something more with this definition.

In Flutter everything is widget. As a result, we must think widget as a central hierarchy in a Flutter framework.

Let us see a minimalist Flutter App to understand this concept first.

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

void main() {
runApp(
    const FirstFlutterApp(),
);
}
```

The above code shows us that the runApp() function takes the given Widget FirstFlutterApp() and makes it the root of the widget tree.

And this is our first custom widget that will sit on the top of the tree.

With reference to the main() function we must also add that to work it properly we need to import Material App library. However, we don't want to dig deep now. Just to make it simple, let's know that we need to have a material design so we can show our widget boxes. Right?

Further, we have created a sub-directory called "widgets" in our "lib" directory and keep the code of FirstFlutterApp(). Remember that also represents a class of hierarchy. Therefore we need to import that local library too.

That is why we need to import that also.

```
import 'widgets/first_flutter_app.dart';
```

Now, we can take a look at the custom widget that we've built.

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

class FirstFlutterApp extends StatelessWidget {
const FirstFlutterApp({
    Key? key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    home: Center(
        child: Text(
        'Hello, Flutter!',
        ),
    ),
    );
}
}
```

The FirstFlutterApp() extends Stateless widget. Widgets have state. But, don't worry, we'll discuss it later.

The widget tree consists of three more widgets. The MaterialApp, Center widget and its child, the Text widget.

As a consequence, the framework forces the root widget to cover the screen. It places the text "Hello, Flutter" at the Center of the screen.

Since we have used MaterialApp, we don't have to worry about the text direction. The MaterialApp will take care of that.

Since each widget is a Dart class, we need to instantiate each widget and want them to show up at the particular location. This is done through the Element class. This is nothing but an instantiation of a Widget at a particular location in the tree.

What do we see?

A text "Hello, Flutter".

However, it is displayed on the particular position in the widget tree. Now, this widget tree can be a complex one.

As a result, widgets can be inflated into more elements as the tree grows in size.

Before we close down, one thing to remember. A widget is an immutable description of part of a user interface.

We'll discuss that in a minute when we will discuss Element class in detail.

What is element in Flutter

We can locate any widget in a widget tree by its element that is an instantiation of widget.

We have been trying to understand how Flutter works from a beginner's point of view. We've already discussed how Widget, constraints and BoxConstraints are related in our previous sections. Now, we'll try to understand what's the role of Element in Flutter.

So far, we've learned that widget is a class and a root widget might have a tree of other widgets.

That's fine.

However, we haven't known so far, how the instantiation of Widget works. After all, widget is a class. Therefore, that must be instantiated.

Element is the answer. In a widget tree, we can find the instantiation of a widget in a particular location, because that is a unique element.

Suppose we reuse Text widget several times in a widget tree. As a result, Text widget is instantiated several times. But each element is unique. Moreover, we can find each Text widget by that unique Element.

We can add one more comment to the above statement. With the widget tree, an Element tree is also formed.

Since widgets are immutable, any widget can be used to configure multiple sub-trees. However, due to the presence of Element, we can easily find the widget's specific location.

Let us see a screenshot and try to understand how this concept works.

In the above screenshot we have two Text widgets, two Text Button widgets. And at the bottom, we have a Floating action button also.

Let's see the associated code.

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

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

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Constraint Sample',
    debugShowCheckedModeBanner: false,
    home: UnderstandingElementHomme(),
    );
}
```

```
}

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

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Element Sample'),
    ),
    body: Center(
        child: Column(
        children: [
            const Text('Element One: Text Widget'),
            const Text('Element Two: : Text Widget'),
            Row(
            children: [
                TextButton(
                onPressed: () {},
                child: const Text('Element Three: TextButton Widget'),
                ),
                TextButton(
                onPressed: () {},
                child: const Text('Element Four: TextButton Widget'),
                ),
            ],
            )
        ],
        ),
    ),
    floatingActionButton: FloatingActionButton(
        onPressed: () {},
        child: const Icon(Icons.add_a_photo),
    ),
    );
}
}
```

Now, in the above widget tree, how can we find the second Text widget? It has a unique element or instantiation of that Widget, which actually points to a particular location.

Actually that element represents the rendered widget on the screen.

If the parent widget rebuilds and creates a new widget for this location, a widget associated with a given element can also change.

What is Align in Flutter

Align widget allows us to place a child widget anywhere we want inside another widget.

Align is a widget that aligns its child within itself. Moreover, based on the child's size, it optionally sizes itself.

For instance, let us think about a minimal Flutter app that aligns a Flutter logo at top right.

```
Center(
        child: Column(
        children: [
            Container(
            height: 120.0,
            width: 120.0,
            color: Colors.blue[50],
            child: const Align(
                alignment: Alignment.topRight,
                child: FlutterLogo(
                size: 60,
                ),
            ),
            ),
```

Just run the app, and see how it looks like.

The code is quite simple. It gives the Container a light blue color. Besides, it has definite width and height, which are tight constraints.

Now, as a child the Align widget places its child, a Flutter logo at the top right corner inside the Container.

However, we could have placed it with an alignment of Alignment.bottomRight.

To do that we should have given the Container a tight constraint, just like before, and that constraint should be bigger than the Flutter logo.

Next, we consider another piece of code that might place three Flutter logo at three different positions inside the same Container.

As we find when we run the code, three Flutter logos show up at three different positions.

Let's see the full code snippet first. After that, we'll discuss the code.

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

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

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Align Sample',
    debugShowCheckedModeBanner: false,
    home: AlignSampleHomme(),
    );
}
}

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

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Align Sample'),
    ),
    body: Center(
        child: Column(
        children: [
            Container(
            height: 120.0,
            width: 120.0,
            color: Colors.blue[50],
            child: const Align(
                alignment: Alignment.topRight,
                child: FlutterLogo(
                size: 60,
                ),
            ),
            ),
            const SizedBox(
            height: 10,
            ),
            Container(
            height: 120.0,
```

```
44              width: 120.0,
45              color: Colors.yellow[50],
46              child: const Align(
47                  alignment: Alignment(0.2, 0.6),
48                  child: FlutterLogo(
49                  size: 60,
50                  ),
51              ),
52              ),
53              const SizedBox(
54              height: 10,
55              ),
56              Container(
57              height: 120.0,
58              width: 120.0,
59              color: Colors.red[50],
60              child: const Align(
61                  alignment: FractionalOffset(0.2, 0.6),
62                  child: FlutterLogo(
63                  size: 60,
64                  ),
65              ),
66              ),
67          ],
68          ),
69      ),
70      floatingActionButton: FloatingActionButton(
71          onPressed: () {},
72          child: const Icon(Icons.add_a_photo),
73      ),
74      );
75  }
76  }
```

The alignment property describes a point in the child's coordinate system and a different point in the coordinate system of this widget.

After that, the Align widget positions the child, here a Flutter logo, in a way so that both points are lined up on top of each other.

In the first case, the Align widget uses one of the defined constants from Alignment, which is Alignment.topRight.

```
Container(
            height: 120.0,
            width: 120.0,
            color: Colors.blue[50],
            child: const Align(
                alignment: Alignment.topRight,
                child: FlutterLogo(
                size: 60,
                ),
            ),
            ),
```

As a result, this constant value places the FlutterLogo at the top right corner of the parent blue Container.

However, the second case is different, where the Alignment defines a single point.

```
Container(
            height: 120.0,
            width: 120.0,
            color: Colors.yellow[50],
            child: const Align(
                alignment: Alignment(0.2, 0.6),
                child: FlutterLogo(
                size: 60,
                ),
            ),
            ),
```

It calculates the position of the Flutter logo in a different way.

The formula is, the result of (0.2 * width of FlutterLogo/2 + width of FlutterLogo/2) comes to a whole number, that is 36.0.

And this is a point in the coordinate system of Flutter logo.

The next point is defined by this formula – (0.6 * height of FlutterLogo/2 + height of FlutterLogo/2), which is equal to 48.0. Moreover, it's point in the coordinate system of the Align widget.

As a result Align will place the FlutterLogo at (36.0, 48.0) according to this coordinate system.

Although in the third example, the calculation goes in the same direction; yet the result differs with the previous one.

```
alignment: FractionalOffset(0.2, 0.6)
```

Consequently, the position of Flutter logo changes.

How to use aspect ratio widget

The AspectRatio Widget tries to find the best size to maintain aspect ratio of a child widget.

The AspectRatio widget attempts to size the child to a specific aspect ratio.

Suppose we have a Container widget with width 100, and height 100. In that case the aspect ratio would be 100/100; that is, 1.0.

Now, each Widget has its own constraints. As a result, the AspectRatio Widget tries to find the best size to maintain aspect ratio. However, while doing so it respects it's layout constraints.

Let's run the code and see the effect where we have used three different types of aspect ratio.

A Container widget has an AspectRatio widget, which has a child Container in a different color.

As a result, we see different types of color combination.

Let's see the full code now.

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

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

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'AspectRatio Sample',
    debugShowCheckedModeBanner: false,
    home: AspectRatioSampleHomme(),
    );
}
}

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

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
```

```
        title: const Text('AspectRatio Sample'),
    ),
    body: Center(
        child: Column(
        children: [
            Container(
            color: Colors.red,
            alignment: Alignment.center,
            padding: const EdgeInsets.all(10),
            width: 100.0,
            height: 100.0,
            child: AspectRatio(
                aspectRatio: 2.0,
                child: Container(
                width: 50.0,
                height: 50.0,
                color: Colors.yellow,
                ),
            ),
            ),
            const SizedBox(
            height: 10,
            ),
            Container(
            color: Colors.blue,
            alignment: Alignment.center,
            width: 100.0,
            height: 100.0,
            child: AspectRatio(
                aspectRatio: 2.0,
                child: Container(
                width: 80.0,
                height: 70.0,
                color: Colors.white,
                ),
            ),
            ),
            const SizedBox(
            height: 10,
            ),
            Container(
            color: Colors.green,
            alignment: Alignment.center,
```

```
66              width: 100.0,
67              height: 100.0,
68              child: AspectRatio(
69                  aspectRatio: 0.5,
70                  child: Container(
71                  width: 100.0,
72                  height: 50.0,
73                  color: Colors.black26,
74                  ),
75              ),
76              ),
77          ],
78          ),
79      ),
80      floatingActionButton: FloatingActionButton(
81          onPressed: () {},
82          child: const Icon(Icons.add_a_photo),
83      ),
84      );
85  }
86  }
```

Remember, in each case, the AspectRatio widget tries to find the best possible size and adjusts the child accordingly.

It comes to our help, when we try to change the size of an image on the fly.

What is Baseline in Flutter

When we try to position a child widget inside or outside of parent widget, Baseline helps us.

Baseline is a widget that positions its child according to the child widget's baseline.

Does it not make any sense?

Well, truly it didn't make any sense to me also, when I first encountered this widget.

In fact, the above statement doesn't really make any sense if we don't turn this abstraction into a concrete example.

Therefore, firstly, let's see one screenshot of a simple Flutter app where we've used Baseline widget.

Secondly, we'll look into the code and try to understand how this widget works.

In the above image we see three Baseline examples. The first one consists of two Container widgets.

Let's see the code.

```
Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 0,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
                ),
            ),
            ),
```

The Baseline widget has two required parameters. The baseline, and the baselineType. The second parameter means the type of the baseline.

As a result, we need to supply value to those parameters.

The baseline parameter plays the most important role, of course. It requires a double value.

In the above case, the baseline is zero.

So it sits on top of the parent Container.

How does it happen?

It happens because the Baseline widget tries to shift the Child Container's bottom or baseline by calculating the distance from the top of the parent Container. Since it's zero, it cannot shift it. So it sits on its top.

As a result, it cannot enter the parent Container.

In the second case, the Child Container's baseline is 50.

```
Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 50,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
```

```
                ),
            ),
            ),
```

Therefore, the baseline logical pixels below the top of the parent Container is 50px. As a result, the bottom of the child Container shifts 50px inside the parent Container.

And the parent Container contains the child in the middle.

Take a look at the screenshot above, you'll understand how it works.

How do you use baseline in flutter?

Most importantly, we use Baseline widget when we want to position the child widget's bottom according to the distance from the top of the parent widget.

When the child Container's baseline is 100px, it moves its bottom 100px exactly, from the top of the parent Container.

As a consequence, the child's bottom merges with the parent's bottom. The above screenshot displays the same thing.

```
Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 100,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
                ),
            ),
            ),
```

If we start increasing the value of the baseline, and make it 110px, what happens?

If we run the code we'll see that the child Container moves outside the parent Container.

Moreover, from the top of the parent Container's to the bottom of the child Container, the distance is 110px exact.

Let's take a look at the full code finally.

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

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

@override
Widget build(BuildContext context) {
    return const MaterialApp(
    title: 'Baseline Sample',
    debugShowCheckedModeBanner: false,
    home: BaselineSampleHomme(),
    );
}
}

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

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: const Text('Baseline Sample'),
    ),
    body: Center(
        child: Column(
        children: [
            const SizedBox(
            height: 100,
            ),
            Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 0,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
                ),
            ),
```

```
            ),
            const SizedBox(
            height: 20,
            ),
            Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 50,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
                ),
            ),
            ),
            const SizedBox(
            height: 20,
            ),
            Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Baseline(
                baseline: 110,
                baselineType: TextBaseline.alphabetic,
                child: Container(
                width: 50,
                height: 50,
                color: Colors.purple,
                ),
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

We can provide a negative value to the baseline of the child Container. Try to make it -50.

At that instance, the bottom of the child Container moves away upward and goes out of the parent Container in the upward direction.

Try it. Happy fluttering.

How to use theme in Flutter

We can create a custom global theme and apply that across the entire flutter app.

Layout of any Flutter app greatly depends on a global theme that we can apply to any widget in the widget tree.

In this section we'll learn how we can create a global theme object that can help us to change or modify our style globally, across the whole flutter app.

To do that, we will create a new ThemeData object first.

```
final globalTheme = ThemeData(
primarySwatch: Colors.deepOrange,
textTheme: const TextTheme(
    bodyText1: TextStyle(
    fontSize: 22,
    height: 1.2,
    ),
    bodyText2: TextStyle(
    color: Colors.blue,
    fontSize: 20,
    fontWeight: FontWeight.bold,
    height: 1.0,
    ),
    caption: TextStyle(
    fontSize: 16,
    fontWeight: FontWeight.bold,
    fontStyle: FontStyle.italic,
    height: 1.2,
    ),
    headline1: TextStyle(
    color: Colors.deepOrange,
    fontFamily: 'Allison',
    fontWeight: FontWeight.bold,
    fontSize: 60,
    ),
    headline2: TextStyle(
    color: Colors.black38,
```

```
    fontSize: 30,
    fontWeight: FontWeight.bold,
    ),
),
appBarTheme: const AppBarTheme(
    backgroundColor: Colors.amber,
    // This will control the "back" icon
    iconTheme: IconThemeData(color: Colors.red),
    // This will control action icon buttons that locates on the right
    actionsIconTheme: IconThemeData(color: Colors.blue),
    centerTitle: false,
    elevation: 15,
    titleTextStyle: TextStyle(
    color: Colors.deepPurple,
    fontFamily: 'Allison',
    fontWeight: FontWeight.bold,
    fontSize: 40,
    ),
),
);
```

Usually the ThemeData class comes with a default configuration. With the help of context we can apply the Material App theme property across the flutter app.

However, we can also override the default configuration and modify it according to our choices.

Exactly that's what we've done here, in the above code.

We've defined the configuration of the overall visual Theme for a MaterialApp or a widget sub-tree within the app.

That's why we've included a Theme widget at the top of the sub-tree.

Normally, if we don't create our own theme object, we can obtain the default configuration with Theme.of. Material components that typically depend on the colorScheme and textTheme. These properties are always provided with non-null values.

As a result, with the help of context of the nearest BuildContext ancestor the static Theme.of method finds the ThemeData value.

In our case, however, we apply the ThemeData object the following way.

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

@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
    title: 'Global Theme Sample',
    theme: globalTheme,
    home: const GlobalThemeSampleHome(),
    );
}
}
```

Once we have declared overall visual Theme for our MaterialApp, we can apply the properties across the widget tree.

It starts with AppBar theme, right way.

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

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(
        'Global Theme Sample',
        style: globalTheme.appBarTheme.titleTextStyle,
        ),
    ),
    body: const GlobalThemeBody(),
    );
}
}
```

Next, the rest global theme properties are applied across the widget tree.

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

@override
Widget build(BuildContext context) {
    DateTime now = DateTime.now();
    String stringDate = DateFormat('yyyy-MM-dd – kk:mm').format(now);
    return Center(
    child: Column(
        children: [
        const SizedBox(
            height: 10,
        ),
        Container(
            height: 70,
            width: 70,
            color: Colors.blue[50],
            child: const Align(
            alignment: Alignment.topCenter,
            child: FlutterLogo(
                size: 60,
            ),
            ),
        ),
        const SizedBox(
            height: 10,
        ),
        Text(
            'Headline 1',
            style: globalTheme.textTheme.headline1,
        ),
        const SizedBox(
            height: 10,
        ),
        Text(
            'Headline 2',
            style: globalTheme.textTheme.headline2,
        ),
        Container(
            margin: const EdgeInsets.all(5),
            padding: const EdgeInsets.all(5),
            child: Text(
            'Body Text 1: Here goes some introduction about yourself.',
```

```
            style: globalTheme.textTheme.bodyText1,
            ),
        ),
        Container(
            margin: const EdgeInsets.all(5),
            padding: const EdgeInsets.all(5),
            child: Text(
            'Body Text 2: Here goes some more information regarding your works.',
            style: globalTheme.textTheme.bodyText2,
            ),
        ),
        Container(
            margin: const EdgeInsets.all(5),
            padding: const EdgeInsets.all(5),
            child: Text(
            stringDate,
            style: globalTheme.textTheme.caption,
            ),
        ),
        ],
    ),
    );
}
}
```

Now we can run the app and see the output.

Certainly, we can apply more style. Moreover, we can create a custom Theme class and with the help of Provider package we can give user a chance to change theme anytime.

We'll discuss that in a separate section.

- All related code snippets - https://github.com/sanjibsinha/build_blog_with_flutter/tree/how_-provider_works[1]

How to use theme with Provider on flutter

Using a custom theme across the Flutter app, can also be done through Provider package.

In our previous section we've discussed how we can use a global theme in Flutter and apply that style across the Widget tree. However, there are better ways to do that.

Certainly, using Provider package to pass ThemeData object is one of them.

[1] https://github.com/sanjibsinha/build_blog_with_flutter/tree/how_provider_works

Moreover, it makes our flutter app more performant.

However, we can use Provider to share a Theme across an entire app in many ways. One of them is to provide a unique ThemeData to the MaterialApp constructor.

In this section, we'll see how to use class Constructors to pass the provided value or ThemeData object. The next section will show you a more robust and easy way to use Provider in providing ThemeData object with less line of code.

Sharing colors, and a unique font style throughout a Flutter app, is always a great idea. However, using the Provider package to use that ThemeData object across the app, will be more interesting.

Therefore, let's jump in and start building our global theme in a different way.

Let us create three folders in our lib folder first. Model, View and Controller.

In model, we create a class first.

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

class GlobalTheme with ChangeNotifier {
final globalTheme = ThemeData(
    primarySwatch: Colors.deepOrange,
    textTheme: const TextTheme(
    bodyText1: TextStyle(
        fontSize: 22,
    ),
    bodyText2: TextStyle(
        color: Colors.blue,
        fontSize: 18,
        fontWeight: FontWeight.bold,
    ),
    caption: TextStyle(
        fontSize: 16,
        fontWeight: FontWeight.bold,
        fontStyle: FontStyle.italic,
    ),
    headline1: TextStyle(
        color: Colors.deepPurple,
        fontSize: 50,
        fontFamily: 'Allison',
    ),
    headline2: TextStyle(
        color: Colors.deepOrange,
        fontSize: 30,
        fontWeight: FontWeight.bold,
```

```
    ),
    ),
    appBarTheme: const AppBarTheme(
    backgroundColor: Colors.amber,
    // This will control the "back" icon
    iconTheme: IconThemeData(color: Colors.red),
    // This will control action icon buttons that locates on the right
    actionsIconTheme: IconThemeData(color: Colors.blue),
    centerTitle: false,
    elevation: 15,
    titleTextStyle: TextStyle(
        color: Colors.deepPurple,
        fontWeight: FontWeight.bold,
        fontFamily: 'Allison',
        fontSize: 40,
    ),
    ),
);
}
```

We've created a globalTheme property that defines a unique global theme for us, using the ThemeData widget.

Next, we'll add the dependency in our pubspec.yaml file, so that we can use Provider package to provide that model class object throughout our app.

```
dependencies:
cupertino_icons: ^1.0.2
flutter:
    sdk: flutter
intl: ^0.17.0
provider: ^6.0.1
```

Now, we should keep the Provider above the root folder and use multi provider and ChangeNotifier-Provider, so that we can later use other Providers as well.

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'model/global_theme.dart';
import 'view/home.dart';

void main() {
runApp(
    /// Providers are above [Root App] instead of inside it, so that tests
    /// can use [Root App] while mocking the providers
    MultiProvider(
    providers: [
        ChangeNotifierProvider(create: (_) => GlobalTheme()),
    ],
    child: const Home(),
    ),
);
}
```

In our Home widget, we can get the unique and custom ThemeData object using Provider.of(context).

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

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

@override
Widget build(BuildContext context) {
    final ThemeData globalTheme = Provider.of<GlobalTheme>(context).globalTheme;
    return MaterialApp(
    title: 'Flutter Demo',
    theme: globalTheme,
    home: HomePage(
        homeTheme: globalTheme,
    ),
    );
}
}
```

Next step is much easier. Just pass the same ThemeData object, to child widgets in the Widget tree through the class constructors.

In fact, app-wide themes are now just ThemeData object provided by the Provider. Once the theme parameter of Material App takes that unique ThemeData object, we can use that across the app. Because Themes widgets created at the root of an app by the MaterialApp can be used anywhere.

Since we've defined our custom Theme, we can use it within our own widgets. Flutter's Material widgets also use our custom Theme to set the background colors and font styles for AppBars, Buttons and more.

Now we'll pass the custom ThemeData object quite easily and use in our child widget.

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

class HomePage extends StatelessWidget {
final ThemeData homeTheme;
const HomePage({Key? key, required this.homeTheme}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(
        'Testing Global Theme with Provider',
        style: homeTheme.appBarTheme.titleTextStyle,
        ),
    ),
    body: HomeBody(
        homeTheme: homeTheme,
    ),
    );
}
}

class HomeBody extends StatelessWidget {
final ThemeData homeTheme;
const HomeBody({Key? key, required this.homeTheme}) : super(key: key);

@override
Widget build(BuildContext context) {
    DateTime now = DateTime.now();
    String stringDate = DateFormat('yyyy-MM-dd – kk:mm').format(now);
    return Center(
    child: Column(
        children: [
```

```
35          Container(
36              margin: const EdgeInsets.all(5),
37              padding: const EdgeInsets.all(5),
38              child: Text(
39              'Headline 2 theme style provided by provider',
40              style: homeTheme.textTheme.headline2,
41              ),
42          ),
43          Container(
44              margin: const EdgeInsets.all(5),
45              padding: const EdgeInsets.all(5),
46              child: Text(
47              'Headline 1 theme style provided by provider',
48              style: homeTheme.textTheme.headline1,
49              ),
50          ),
51          Container(
52              margin: const EdgeInsets.all(5),
53              padding: const EdgeInsets.all(5),
54              child: Text(
55              'Body Text 2: Here goes some introduction about yourself. Theme by provi\
56  der.',
57              style: homeTheme.textTheme.bodyText2,
58              ),
59          ),
60          Container(
61              margin: const EdgeInsets.all(5),
62              padding: const EdgeInsets.all(5),
63              child: Text(
64              'Body Text 1: Here goes some more information regarding your works. Them\
65  e by provider.',
66              style: homeTheme.textTheme.bodyText1,
67              ),
68          ),
69          Container(
70              margin: const EdgeInsets.all(5),
71              padding: const EdgeInsets.all(5),
72              child: Text(
73              'Datetime theme style provided by provider: $stringDate',
74              style: homeTheme.textTheme.caption,
75              ),
76          ),
77          ],
```

```
78      ),
79      );
80  }
81  }
```

As a result, our app uses that custom theme from AppBar to the body.

- All related code snippets - https://github.com/sanjibsinha/build_blog_with_flutter/tree/theme-with-provider[2]

[2] https://github.com/sanjibsinha/build_blog_with_flutter/tree/theme-with-provider