Colours & Fonts: (Refer App: newbie)

Let us customise our app in a few ways. We need to understand that we are just familiarising ourselves with all the different widgets in Flutter. We’ll learn about all the layouts in Flutter in a moment. We’ll then learn to include all the different widgets that we have learnt in a neat little layout.

Let’s say that we want to customise the colour of our AppBar. We do it by adjusting the backgroundColor property of the AppBar. We set it equal to a ‘Colors.color\_name’. The cool thing here, is that we can directly choose from the beautiful Material.io colour palette! And MY GOD, IS IT AMAZING!

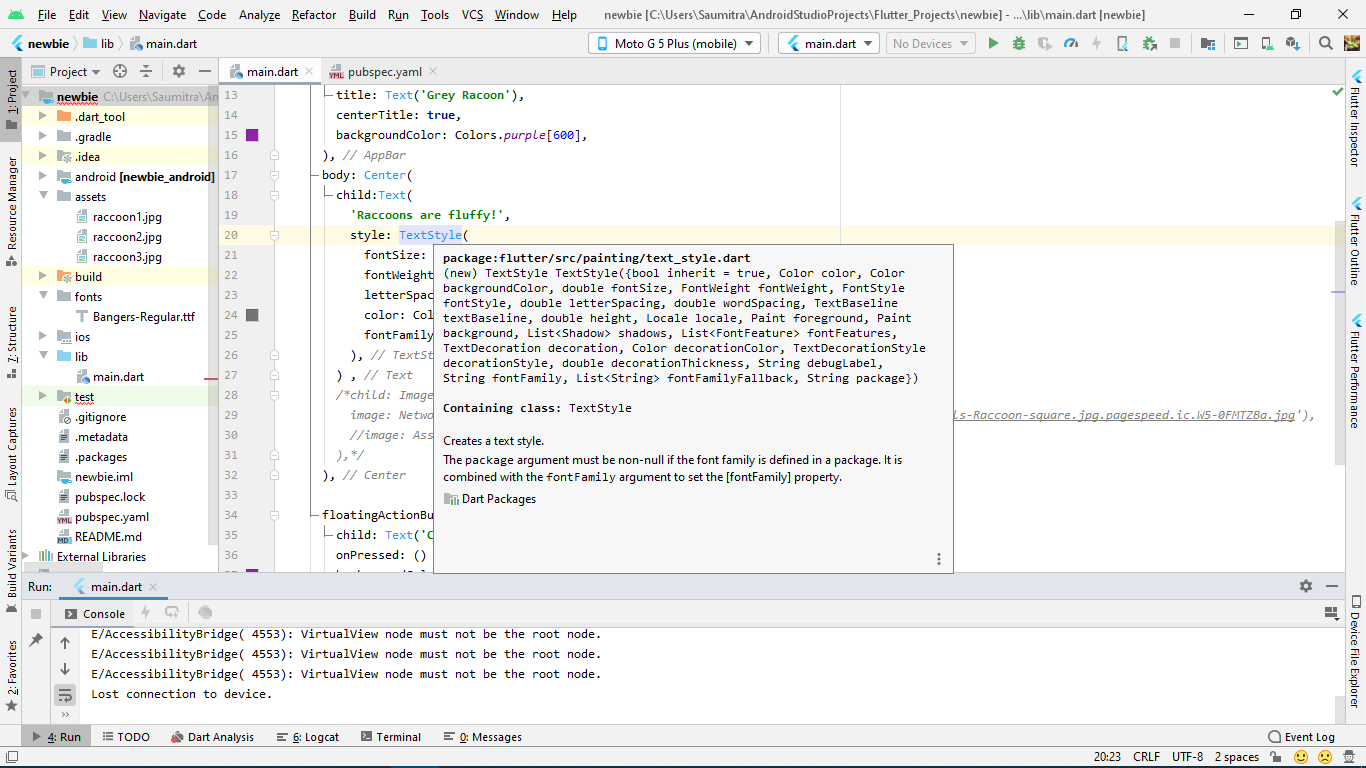
Sneak peek [here](https://material.io/design/color/the-color-system.html#tools-for-picking-colors)!

We select a colour, and press ctrl+q to open the DART documentation for more information. Here we can set different intensities of the colour by including the variant inside the square bracket. Let’s choose purple.

backgroundColor: Colors.*purple*[600];

In a similar manner, we can also style the FloatingActionButton by setting the backgroundColor property to Colors.*purple*[600].

Similarly, let’s style the text that we wrote earlier. For this we shall use the ‘style’ property, and set it equal to the ‘TextStyle’ widget. This widget in turn presents itself with a host of different properties to style our text and customise it accordingly. To have a look, highlight the TextStyle widget and press ctrl+q.



We’ll adjust the fontSize, fontWeight, letterSpacing, and colour properties of the text.

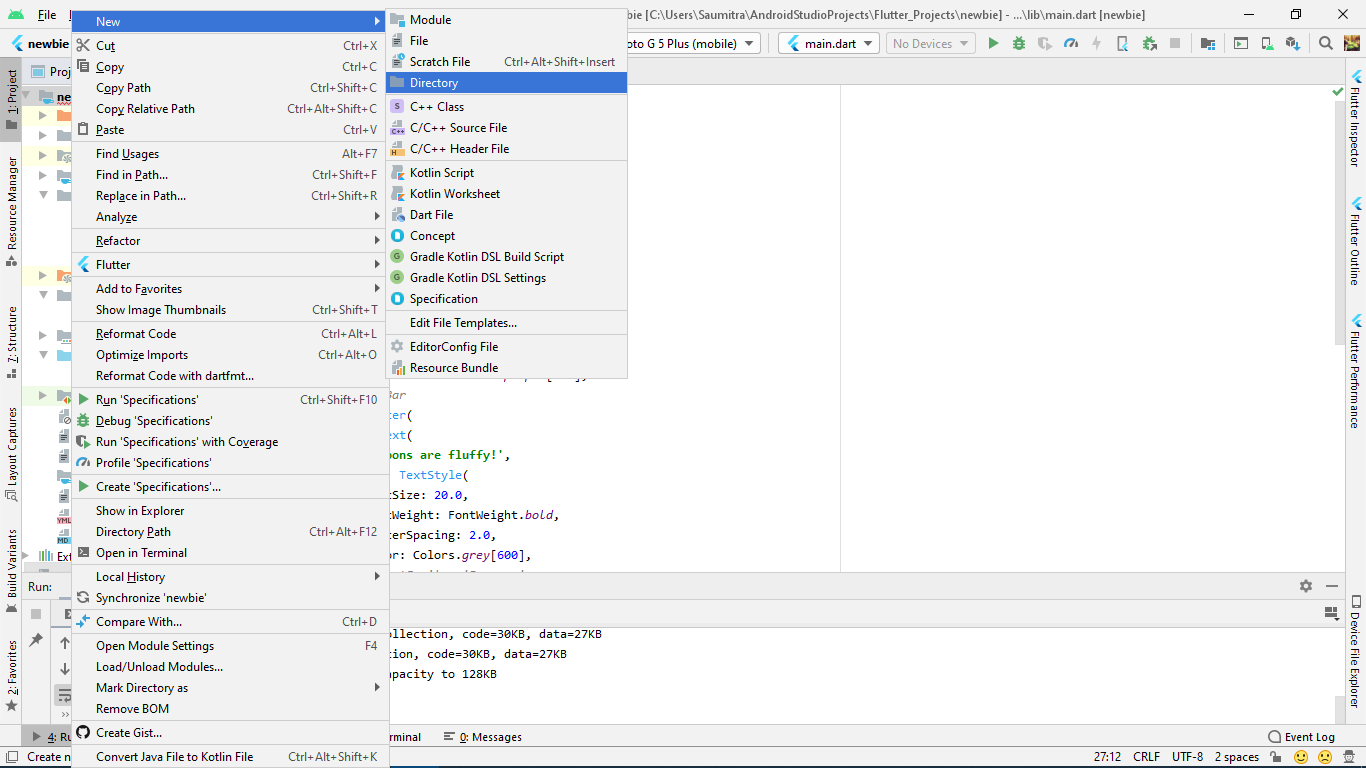
body: Center(  
 child:Text(  
 **'Raccoons are fluffy!'**,  
 style: TextStyle(  
 fontSize: 20.0,  
 fontWeight: FontWeight.*bold*,  
 letterSpacing: 2.0,  
 colour: Colors.*grey*[600],  
  
 ),  
 ) ,

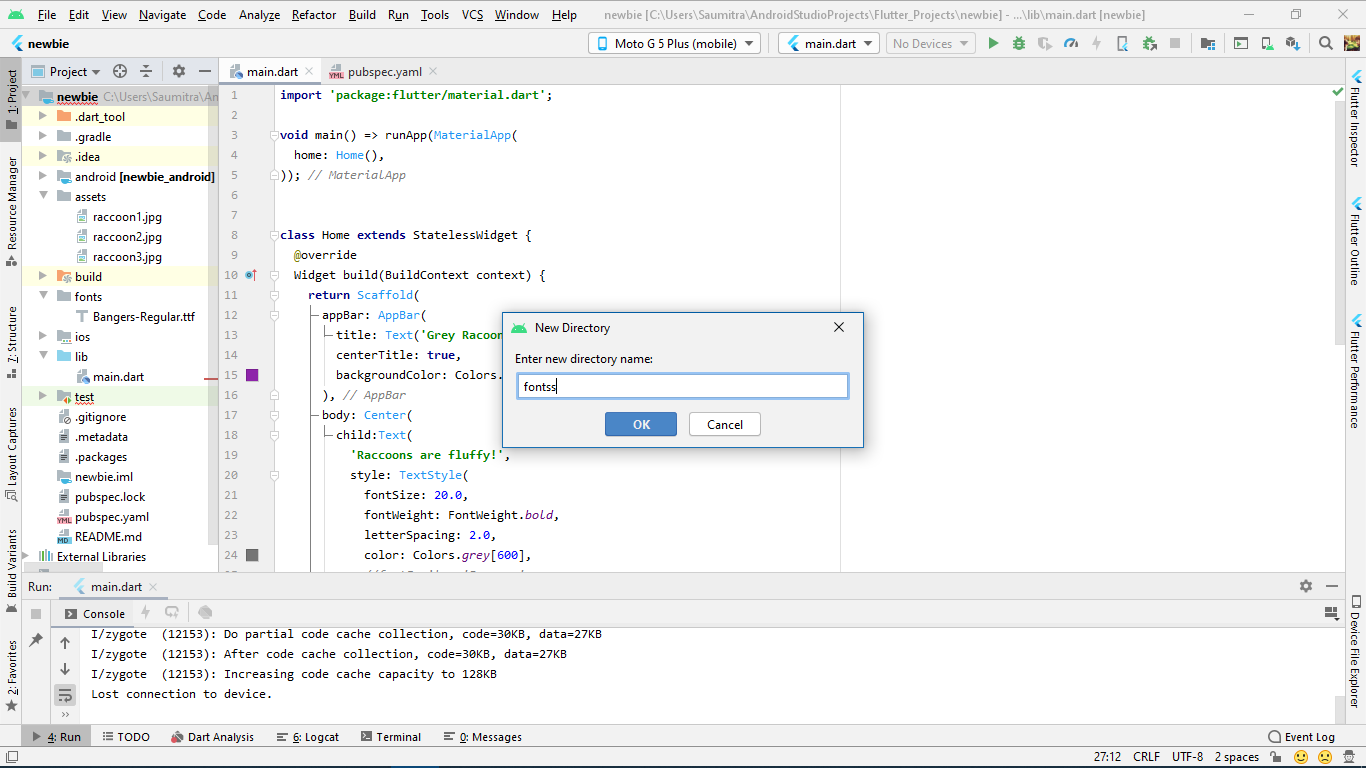
),

Our App looks like:



Now, let’s change the font of our text. The process for doing this is slightly different. We head to <https://fonts.google.com/> and select a font that we like, and download it to our local drive.

We then create a new directory in our project folder—call it ‘fonts’, and paste the font file in it.



In order to make flutter know about the font file that we have used, we head on to the pubspec.yaml file. Under the fonts subheading, we add the font family name as well as the asset used for the font.

The font family is nothing but the name that we’ll use for the font. The asset is nothing but the font file we downloaded from the internet. We type in the path for the font file under the asset heading. Care must be taken to ensure that the pubspec.yaml file is indented with ONLY 2 SPACES.

We chose the font called ‘Bangers-Regular’ and we’ll name it’s family as ‘Bangers’ for our convenience. We can name it anything we want. Our pubspec file now has text which looks like:

**fonts**:  
 - **family**: Bangers  
 **fonts**:  
 - **asset**: fonts/Bangers-Regular.ttf

We again head to our main.dart file, and add the fontFamily property to the TextStyle widget. We simply write the font family name (Bangers in our case) in front of it.

fontFamily: **'Bangers'**,



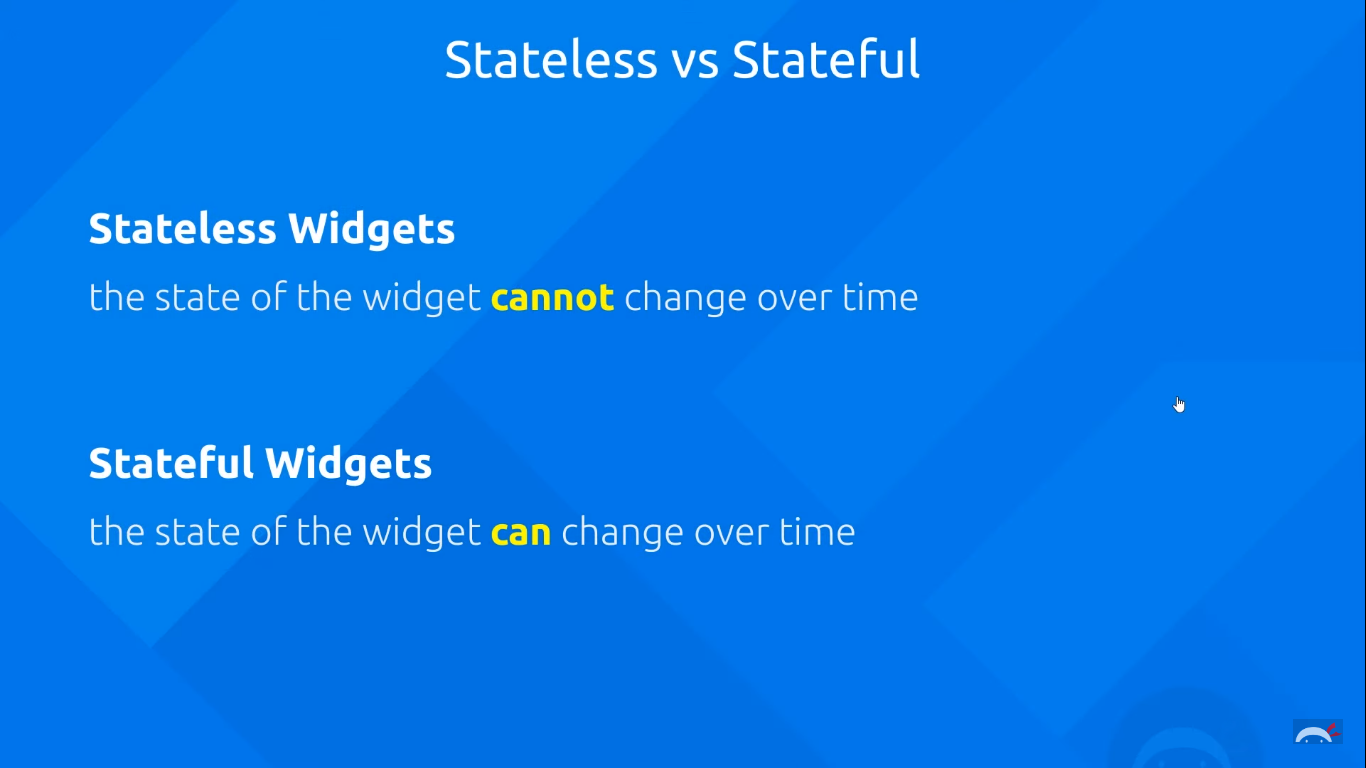
Hot Reload:

We read that one of the advantages of Flutter was the facility of Hot Reload. Which means that changes are deployed to our ‘on-the-go’. As the app runs, the widgets are updated accordingly.

But currently our app does nothing if we try to hot reload. This is because we did not specify which widgets are stateful and which are stateless. So, flutter treats all widgets as stateful and reloads all of them with each run command. As a result, we need to reload the entire app (which is stupid).

Why not take advantage of the awesome ‘hot reload’ that flutter offers?

Before we do that, we need to understand what are ‘stateless’ and ‘stateful’ widgets.



As the above picture depicts, in flutter we can have either stateless or stateful widgets. Right now we're using a stateless widget. This basically means that the state of the widget cannot change over time. For example, the layout or the colours or any data we use inside that widget has to be final and it cannot change over time as we use the app. It CAN contain data, but that data can't change after the widget has been initialized.

Stateful widgets on the other hand can contain ‘States’ which can change over time. So things like its colour, its layout or any data inside it can change over time. For example, we can have some kind of counting widget that displays the number of flies that you swat on an app that would have changing data over time. As more flies appear on the screen the number would go up. So, we would use a stateful widget for that.

Depending on what we need in our apps for the different parts we'd either choose a stateless widget or a stateful widget.

In order to create our own stateless widget we need to create our own class. Android studio provides a neat little shortcut for quickly creating a stateless widget. Just type in ‘stless’ and press ctrl+space, and voila! We have our readymade code snippet for creating a stateless widget class.

Let’s name this class as ‘Home’, as that is what we want to display on the home screen of the app. This class extends the ‘StatelessWidget’ class, which has been predefined in Flutter. Here we override a method called ‘build’ which returns an object of the Widget class—meaning that it essentially creates a widget and returns it.

We copy THE ENTIRE code that we have written till now after the ‘home’ property (i.e. the Scaffold widget) of the MaterialApp widget, and place it after the return statement. This essentially means that we are consolidating whatever we wrote on the home-screen and compiling into a SINGLE STATELESS WIDGET, and calling the widget as ‘Home’ (i.e. the class name). In this manner, we have successfully made our own widget!

**class** Home **extends** StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 **return** Scaffold(  
 appBar: AppBar(  
 title: Text(**'Grey Racoon'**),  
 centerTitle: **true**,  
 backgroundColor: Colors.*purple*[600],  
 ),  
 body: Center(  
 child:Text(  
 **'Raccoons are fluffy!'**,  
 style: TextStyle(  
 fontSize: 20.0,  
 fontWeight: FontWeight.*bold*,  
 letterSpacing: 2.0,  
 color: Colors.*grey*[600],  
 fontFamily: **'Bangers'**,  
 ),  
 ) ,  
   
 ),  
  
 floatingActionButton:FloatingActionButton(  
 child: Text(**'Click!'**),  
 onPressed: () {},  
 backgroundColor: Colors.*purple*[600],  
 ),  
 );  
 }  
}

All this is now a single widget called ‘Home’

We, thus add a single widget to our home property of the MaterialApp widget.

**void** main() => runApp(MaterialApp(  
 home: Home(),  
));

This doesn’t change the output in any way. But it makes whatever changes that we make during development appear blazing fast, thanks to ‘Hot Reload!’

Images & Assets:

Let’s add images to our app!

There are 2 ways of adding images to our app. We can either add them from the internet using what is known as a ‘NetworkImage’ widget. Or, we can add them locally by adding them as assets to our app using the ‘AssetImage’ widget.

Before we do that, let’s comment the text that we had in our app body under the child property of the Center widget.

1. NETWORK IMAGE:

Adding a network image is very simple. We simply select the link of the image that we want to add. Then under the child property of the Center widget, we add a widget called as ‘Image’. It again has a property called as ‘Image’ under which we add a widget called ‘NetworkImage’. Inside it we specify the URL of the image we want to display.

child: Image(  
 image: NetworkImage(**'https://curiodyssey.org/wp-content/uploads/bb-plugin/cache/xMammals-Raccoon-square.jpg.pagespeed.ic.W5-0FMTZBa.jpg'**),  
  
),

A neat shortcut for adding network images to directly write:

child: Image.network(**'https://curiodyssey.org/wp-content/uploads/bb-plugin/cache/xMammals-Raccoon-square.jpg.pagespeed.ic.W5-0FMTZBa.jpg'**);

Our entire code looks like:

**class** Home **extends** StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 **return** Scaffold(  
 appBar: AppBar(  
 title: Text(**'Grey Racoon'**),  
 centerTitle: **true**,  
 backgroundColor: Colors.*purple*[600],  
 ),  
 body: Center(  
 */\*child:Text(  
 'Raccoons are fluffy!',  
 style: TextStyle(  
 fontSize: 20.0,  
 fontWeight: FontWeight.bold,  
 letterSpacing: 2.0,  
 color: Colors.grey[600],  
 fontFamily: 'Bangers',  
 ),  
 ) ,\*/* child: Image(  
 image: NetworkImage(**'https://curiodyssey.org/wp-content/uploads/bb-plugin/cache/xMammals-Raccoon-square.jpg.pagespeed.ic.W5-0FMTZBa.jpg'**),  
   
 ),  
 ),  
  
 floatingActionButton:FloatingActionButton(  
 child: Text(**'Click!'**),  
 onPressed: () {},  
 backgroundColor: Colors.*purple*[600],  
 ),  
 );  
 }  
}

And for the output:



1. ASSET IMAGE:

For the asset image, we first need to create a new directory in our project which will hold our images. Unlike the drawable directory of the Java world, we need to manually create a directory for the assets of our image. Let’s call our directory as ‘Assets’. We copy the images that we want to load in our app to this directory.

We then head to the pubspec.yaml file in order to tell flutter that we indeed are using some assets in our app. Under the assets section, we add the filenames of the images.

*# To add assets to your application, add an assets section, like this:***assets**:  
 - assets/raccoon1.jpg  
 - assets/raccoon2.jpg  
 - assets/raccoon3.jpg

OR, in a better way, we simply ADD THE ENTIRE ASSETS FOLDER under the assets section. This is usually used in case we have a lot of assets in our app.

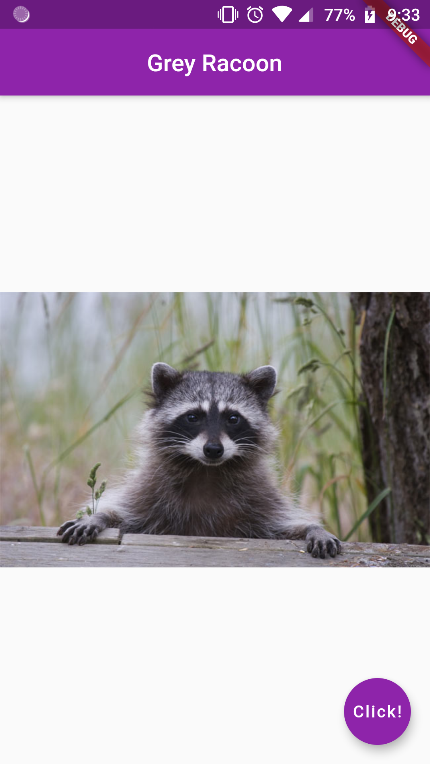
*# To add assets to your application, add an assets section, like this:***assets**:  
 - assets/

Then we head to the child property of the Center widget, and we again add a widget called as ‘Image’. It again has a property called as ‘image’ under which we add a widget called ‘AssetImage’. Inside it we specify the path of the image.

child: Image(  
 image: AssetImage(**'assets/raccoon2.jpg'**),  
 ),  
),

The entire code looks like:

**import 'package:flutter/material.dart'**;  
  
**void** main() => runApp(MaterialApp(  
 home: Home(),  
));  
  
  
**class** Home **extends** StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 **return** Scaffold(  
 appBar: AppBar(  
 title: Text(**'Grey Racoon'**),  
 centerTitle: **true**,  
 backgroundColor: Colors.*purple*[600],  
 ),  
 body: Center(  
 */\*child:Text(  
 'Raccoons are fluffy!',  
 style: TextStyle(  
 fontSize: 20.0,  
 fontWeight: FontWeight.bold,  
 letterSpacing: 2.0,  
 color: Colors.grey[600],  
 fontFamily: 'Bangers',  
 ),  
 ) ,\*/* child: Image(  
 image: AssetImage(**'assets/raccoon2.jpg'**),  
 ),  
 ),  
  
 floatingActionButton:FloatingActionButton(  
 child: Text(**'Click!'**),  
 onPressed: () {},  
 backgroundColor: Colors.*purple*[600],  
 ),  
 );  
 }  
}

ROWS COLUMNS & PADDING: (Refer App: lumpylettuce)

It’s time to learn a key concept now. Till now we were just putting up single widgets onto the body of the app. Let’s now learn about placing multiple widgets on the app screen.

The home screen can divided into rows and columns, and essentially we can consider it as a type of “GridLayout”.

Before we learn about rows and columns, we need to grasp the concept of “Padding and Containers”.

One of the simplest widget that flutter has to offer is the “container” widget. It does as it says—it acts as a container for other widgets, and therefore wraps around them. When no widget is inserted inside it, the container occupies the entire screen.

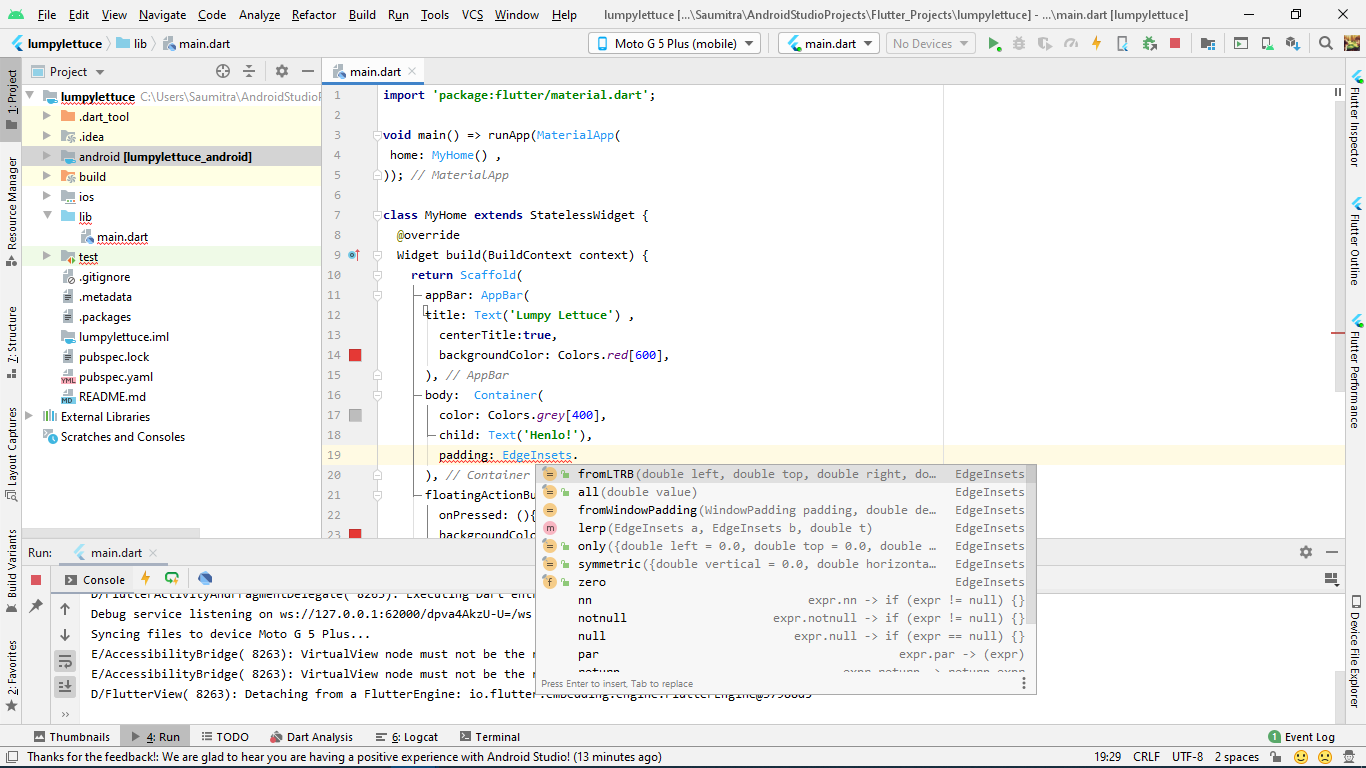
We insert other widgets inside it using the “child” property. Let’s take the example of a simple text widget inserted in a container. We give the background color as grey to it in order to identify it on the white screen.

body: Container(  
 color: Colors.*grey*[400],  
 child: Text(**'Henlo!'**),  
),



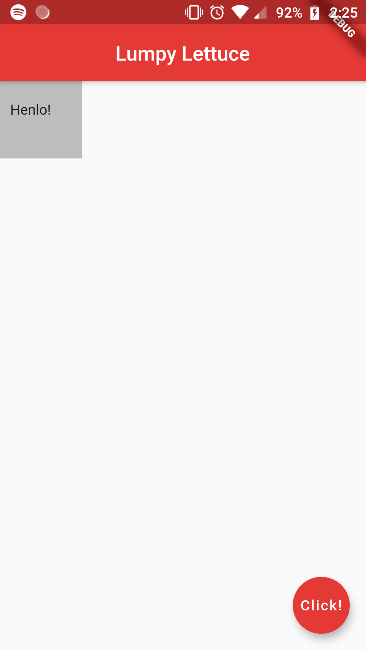
CONTAINER PADDING:

One of the advantages of the container widget, is that we can assign padding as well as margins to it. This enables us to better position elements on our app screen. We adjust the padding of the container using the “padding” property. Its value is equal to a class called “EdgeInsets” and there are various options available to us here:



1. fromLTRB: It stands for “from Left Top Right Bottom”. As the name suggests, it accepts 4 double values for the left, top, right, and bottom padding of the widget contained inside.

padding: EdgeInsets.fromLTRB(10.0,20.0, 30.0, 40.0),



1. all: As the name suggests, we specify “one” double value for the padding on all sides.

padding: EdgeInsets.all(20.0)



1. Symmetric: It accepts 2 values as properties—the padding for the top and bottom (vertical padding) and the padding for the left and right (horizontal padding).

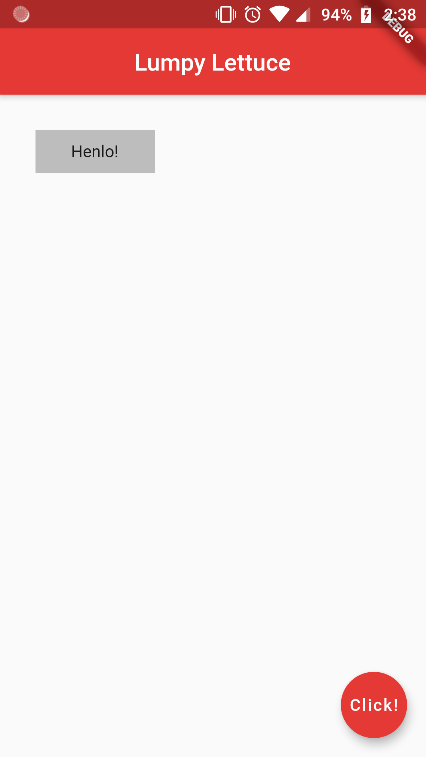
padding: EdgeInsets.symmetric(horizontal: 30.0, vertical: 10.0),



CONTAINER MARGINS:

Similar to padding, we also have a “margin” property. It has the EXACT SAME values as that padding. We here again deal with the “EdgeInsets” property with the same 4 options. As an example, let’s see the EdgeInsets.all property:

margin: EdgeInsets.all(30.0)



THE PADDING WIDGET:

In order to quickly add padding to a particular widget without bothering to insert it into a container widget first, we use the “Padding” widget. This widget has only one adjustable property called “padding”. The procedure is the same, the only difference is that we can ONLY adjust the padding here, NOT the margins, NOT the colour, etc. ONLY PADDING.

body: Padding(  
 child: Text(**'Padding Text!'**),  
 padding: EdgeInsets.all(90.0),  
),

