Module 4 & 5

Flutter Widgets Fundamentals

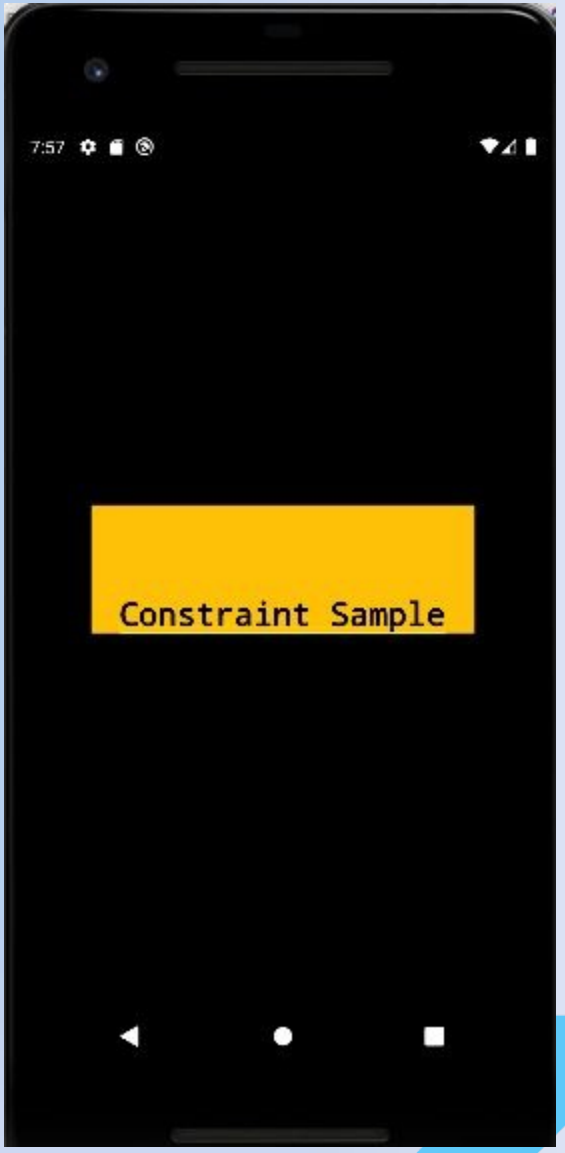
Created By:-

The EasyLearn Academy

What is constraint (rules)?

- A constraint is just **a set of 4 doubles values:**
  - **a minimum and maximum width, and**
  - **a minimum and maximum height**.
- It's about the size of a widget and what are the minimum and maximum height/width.
- Widget gets its constraint or size from its immediate parent. After that, it passes that constraint to its immediate child.
- 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.
- Because, the grand child has its parent that inherits the size from its parent and decides what should be the size of its child.

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

      ),
    );
  }
```

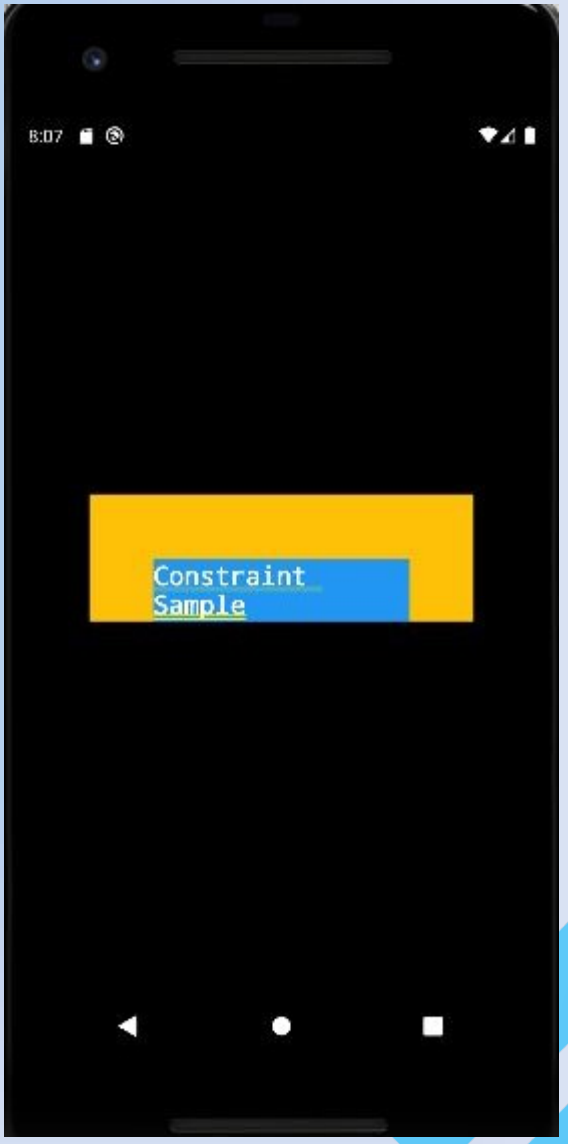


explanation

- In this example container widget is child of a Center widget. the Center widget gets the full size it allows 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.”

Example 2

```
@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,
            ),
          ),
        ),
      ),
    );
  }
```



explanation

- In this example Container with width 300, height 100 and color amber,
- it has a child, which is also 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 Alignment.bottomCenter.
- As a result, the child Container gets the width 200 and height 50 and is placed at the bottom Center position. It happens smoothly because the parent Container's width and height is bigger than the child Container. Therefore, it allocates the exact size required by the child.

Limitations

- A widget can decide its own size only within the constraints given to it by its parent. This means a widget usually **can't have any size it wants**.
- A widget **can't know and doesn't decide its own position in the screen**, since it's the widget's parent who decides the position of the widget.
- Since the parent's size and position, in its turn, also depends on its own parent, it's impossible to precisely define the size and position of any widget without taking into consideration the tree as a whole.
- If a child wants a different size from its parent and the parent doesn't have enough information to align it, then the child's size might be ignored. **Be specific when defining alignment.**

Let us understand some common properoties

1. child: This property is used to store the child widget of the layout.

2. color: This property is used to set the background color of the layout. widget/entire

3. height and width: This property is used to set the layout's height and width according to our needs.

By default, the layout always takes the space based on its child widget.

4. margin: This property is used to surround the empty space around the layout.

5. padding: This property is used to set the distance between the border of the layout (all four directions) and its child widget.

Layout in flutter

- Layout in Flutter defines how the content(widget) expands in a given area as per the requirement of application.
- Layout is important because application need to run on web, mobile devices which has different screen sizes,
- So content/widget has to be dynamically displayed properly.
- In Flutter, the core of the layout is widgets. Even the layout is also widget.
- You can add multiple widgets into Layout as per your need.
- Some special widgets like Center, Align, Container, etc., are also provided in Flutter for laying out the user interface.

Types of Flutter Layout

- Based on its child, the Layout widget in Flutter are divided into 2 broad categories:
- Single child widget
  - Single Child Widget is a type of widget that has only a single widget inside the itself.
  - These single child widgets are very simple to use and make the code readable for the programmer.
- Multiple Child Widget
  - Multiple Child Widgets are the type of widgets that have more than one child widget, and the layout of each child widget is unique.
  - For example, to create a table with rows and columns, a row widget is used, which lays its child object in the horizontal direction, and a column widget, on the other hand, lays its child object in the vertical direction.

Single Child Widget

- **Container** class in flutter is a convenience widget that combines common painting, positioning, and sizing of widgets. We can apply margin, padding, color, width, height & transformation.
- **Center:** This widget is used to center the child in itself. This widget is inherited from the Align class. It is one of the simple yet very useful widgets used in Flutter.
- **Align:** Align class is used to set position of child inside its parent.
- **Padding:** This is one of the most commonly used widgets in order to provide the padding to the child widgets to fit into the layout widget. Padding in Flutter can be provided using the EdgeInserts for the required sides.
- **BaseLine:** This widget is used to position the child widget according to the baseline.

Container

- Container is a simplest type of layout in which we can put single widget, which can we label or textbox or radio button or checkbox or button or any other widget, we can also put other layout inside container like row column stack center etc
- we can optionally apply margin, padding
- we can change colour or we can change shape of the container as per the need means it is possible to display container as a rounded circle instead of rectangular box
- you can also align the child in the container.
- you can also given speecific height & widget as per the needs or even you can rotate container

What is widthfactor and heightfactor?

- heightFactor. If non-null, sets its height to the child's height multiplied by this factor.
- widthFactor. If non-null, sets its width to the child's width multiplied by this factor.

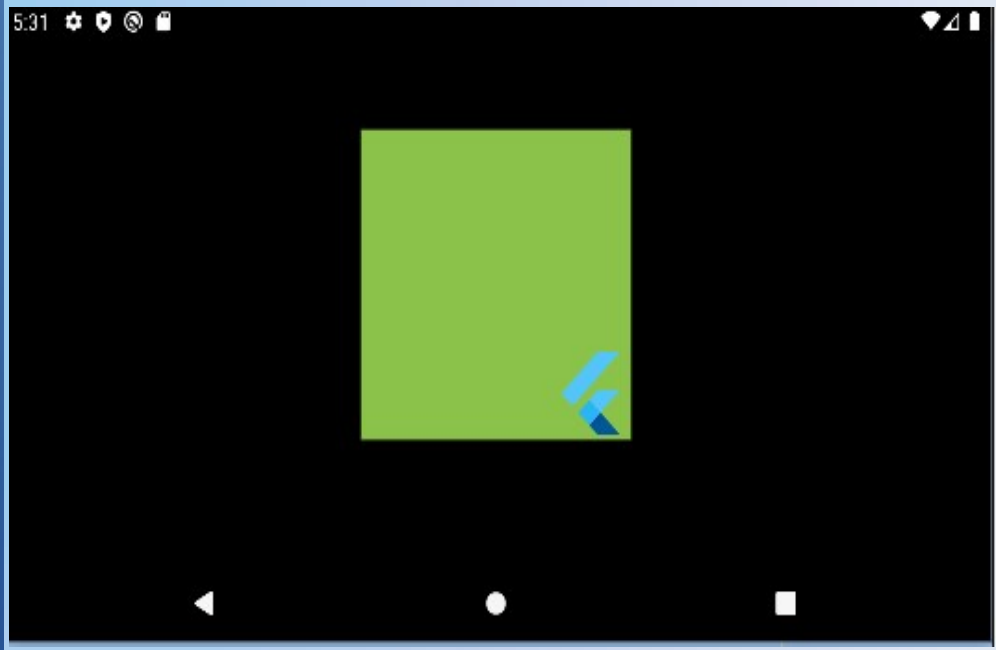
Center

- **Center widget** aligns its child widget to the *center* of the available space on the screen/parent.
- If **widthFactor** is not specified, the width of **Center** will be as large as possible, otherwise, the width of **Center** is equal to the width of the **child** multiplied by the **widthFactor**.
- The **heightFactor** parameter also has the same behavior for the height of the **Center**.
- So by default, the size of the **Center** will be as large as possible.

Align

- align widget set alignment of its child within itself and optionally sizes itself based on the child's size.
- For example, to align a box at the bottom right, you would pass this box a tight constraint that is bigger than the child's natural size, with an alignment of bottomright.
- This widget will be as big as possible if its dimensions are constrained and widthFactor and heightFactor are null.
- If a dimension is unconstrained and the corresponding size factor is null then the widget will match its child's size in that dimension.
- If a size factor is non-null then the corresponding dimension of this widget will be the product of the child's dimension

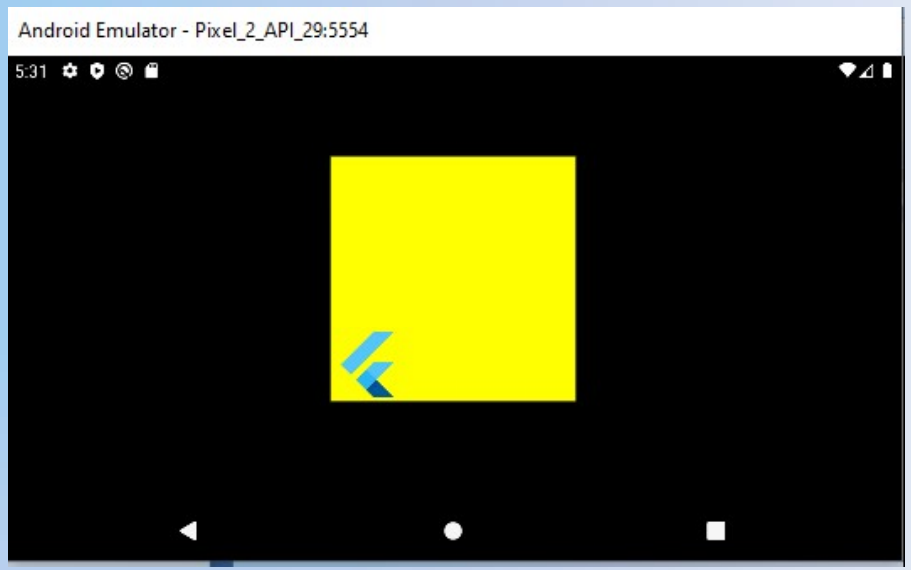
```dart
import 'package:flutter/material.dart';
void main() => runApp(MyAlign());
class MyAlign extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        height: 200,
        width: 200,
        decoration: BoxDecoration(
          color: Colors.lightGreen,
        ),
        child: Align(
          alignment: Alignment.bottomRight,
          child: FlutterLogo(
            size: 60,
          ),
        ),
      ),
    );
  }
}
```



align

- Alignment can be also given in terms of double value. Possible range of alignment can be -1.0 and 1.0 for both width and height.
- Height and width is calculated from center point in such cases.

```
class MyAlign2 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        height: 200,
        width: 200,
        decoration: BoxDecoration(
          color: Colors.yellowAccent,
        ),
        child: Align(
          alignment: Alignment(-1,1),
          child: FlutterLogo(
            size: 60,
          ),
        ),
      ),
    );
  }
}
```



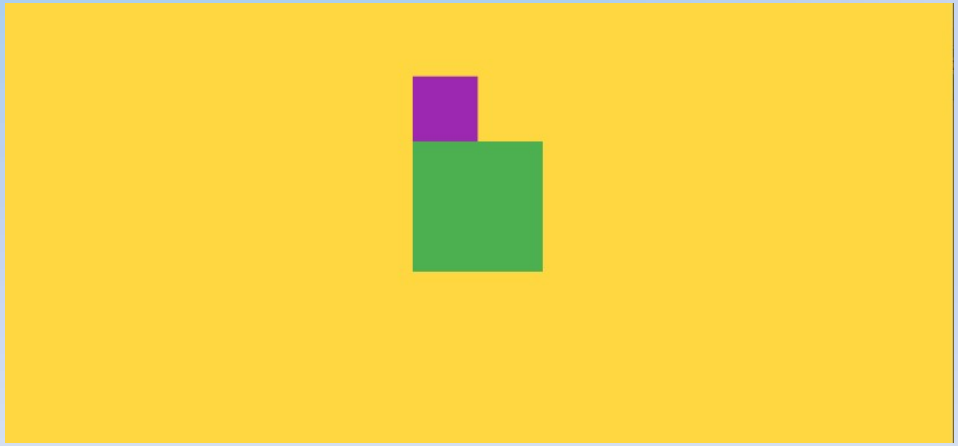
Padding

- **Padding** widget adds padding or empty space around a widget or a bunch of widgets.
- We can apply padding around any widget by placing it as the child of the *Padding* widget.
- The size of the child widget inside padding is constrained by how much space is available after adding empty space around.
- Padding can be either applied using padding class or using Container with padding

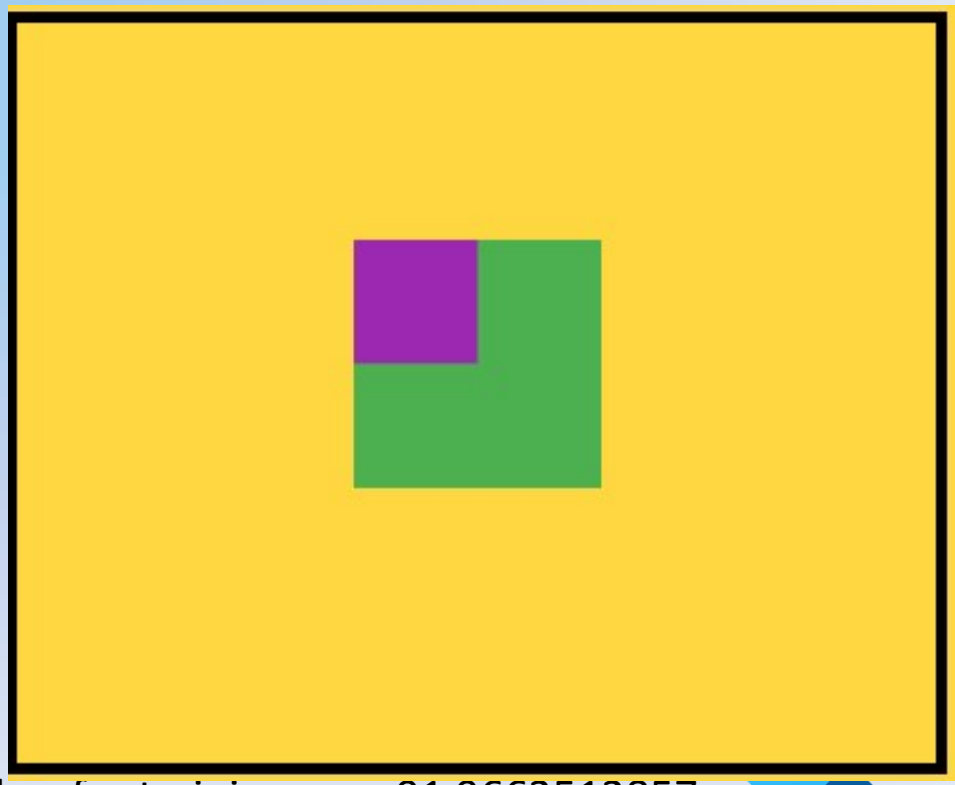
Baseline

- baseline widget tries to shift the Child Container's bottom (baseline) by calculating the distance from the top of the parent Container.
- The Baseline widget has two required parameters. The **baseline**, and the **baselineType**. The second parameter means the type of the baseline.
- we use Baseline widget when we want to position the child widget's bottom in context of the distance from the top of the parent widget.

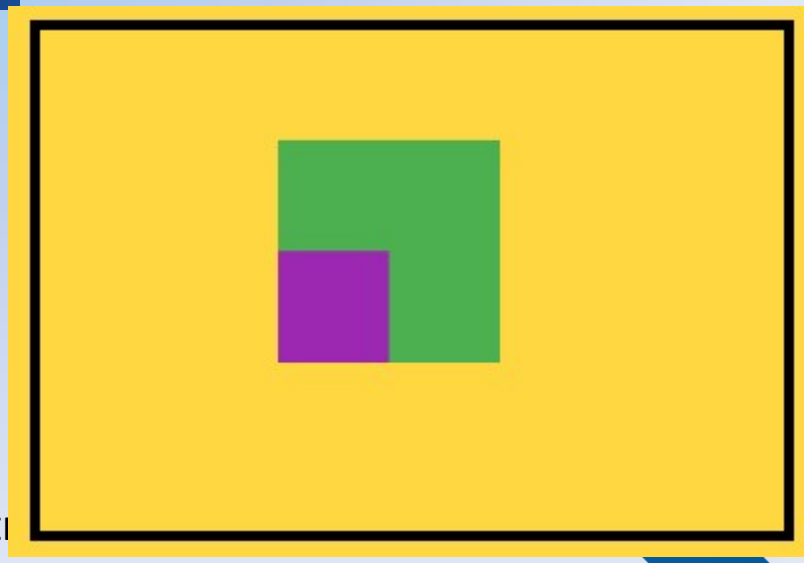
```
return Material(
      color: Colors.amberAccent,
      child: Center(
        child: Container(
          width: 100,
          height: 100,
          color: Colors.green,
          child: Baseline(
            baseline: 0, //0 px away from the top of the parent
            baselineType: TextBaseline.alphabetic,
            child: Container(
              width: 50,
              height: 50,
              color: Colors.purple,
            ),
          ),
        ),
      )
    );
```



```
return Material(
     color: Colors.amberAccent,
     child: Center(
       child: Container(
         width: 100,
         height: 100,
         color: Colors.green,
         child: Baseline(
           baseline: 50, //50 px away from the top of the parent
           baselineType: TextBaseline.alphabetic,
           child: Container(
             width: 50, //child bottom is 50 px away from parent's top
             height: 50,
             color: Colors.purple,
           ),
         ),
       ),
     )
   );
```



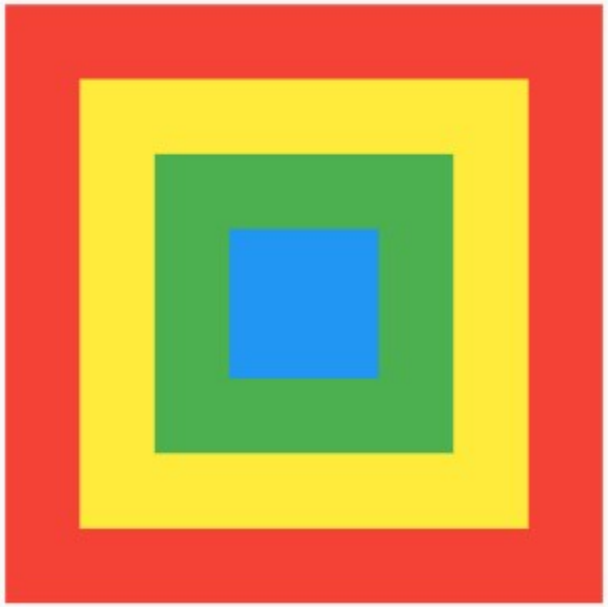
```
return Material(
     color: Colors.amberAccent,
     child: Center(
       child: Container(
         width: 100,
         height: 100,
         color: Colors.green,
         child: Baseline(
           baseline: 100, //100 px away from the top of the parent
           baselineType: TextBaseline.alphabetic,
           child: Container(
             width: 50, //child bottom is 50 px away from parent's top
             height: 50,
             color: Colors.purple,
           ),
         ),
       ),
     )
   );
```



Stack

- The stack is a widget in Flutter that contains a list of widgets and positions them on top of the other.
- In other words, the stack allows developers ***to overlap multiple widgets into a single screen*** and renders them from bottom to top.
- Hence, the **first widget** is the **bottommost** item, and the **last widget** is the **topmost** item.
- The child widget in a stack can be either **positioned** or **non-positioned**.
- Positioned items are wrapped in Positioned widget and must have a one non-null property
- The non-positioned child widgets are aligned itself on the screen based on the stack's alignment.
- The default position of the children is in the top left corner.
- We have to given alignment property in stack widget else we will get error.

```
return Material(
     child: Center(
      child: Container(
        height: 200,
        width: 200,
        color: Colors.red,
        child:  Stack(
          alignment: Alignment.center,
          children: [
             Container(
               height: 150,
               width: 150,
               color: Colors.yellow,
             ),
            Container(
              height: 100,
              width: 100,
              color: Colors.green,
            ),
            Container(
              height: 50,
              width: 50,
              color: Colors.blue,
            ),
          ],
        ),
      ),
    ),
  );
```



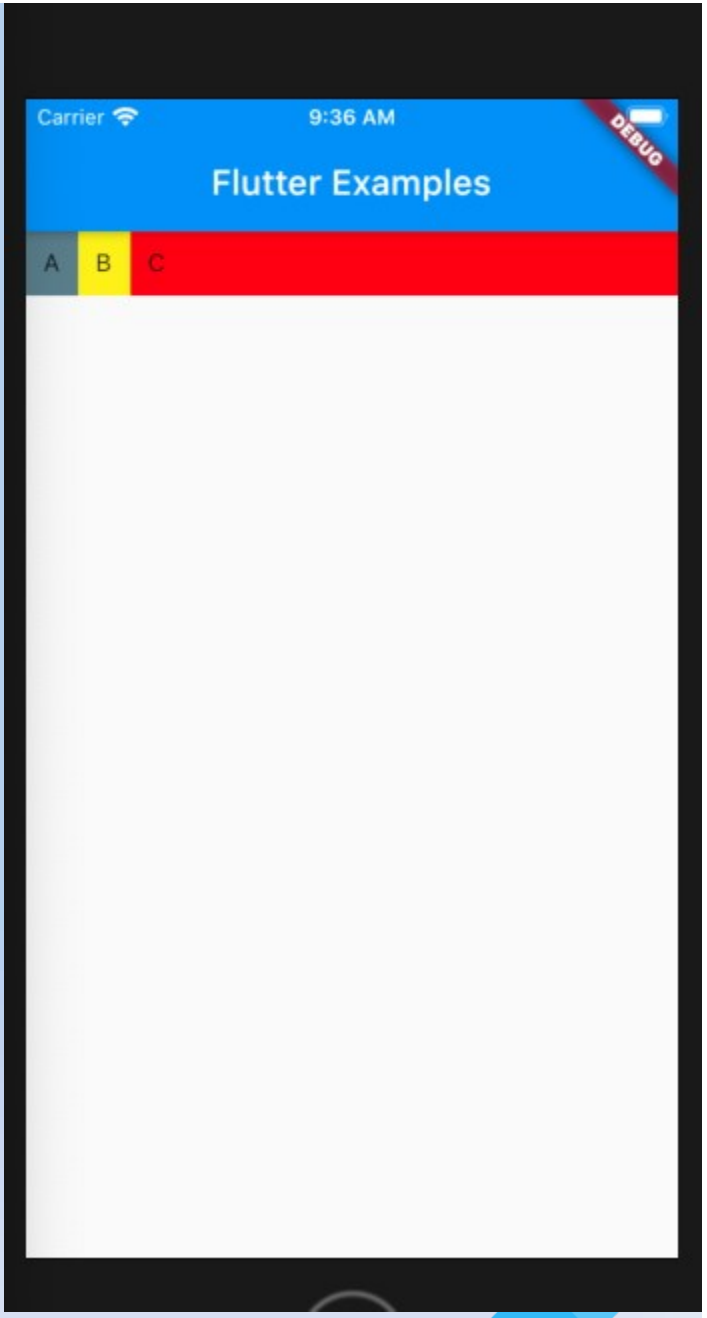
Expanded ....

- Normally Row and Column divide the free space according the size of the children widgets. Children are inflexible. Also this space usage depend on the padding area or mainAxisAlignment property
- But if you want some of the children stretch and fill the extra space then you should use Expanded widget (Wrap the children with Expanded widget) In this situation child become flexible.
- A widget that expands a layout/widget of a Row, Column, or Flex so that the layout/widget fills the available space.
- it does this by filling the available space along the main axis (e.g., horizontally for a Row or vertically for a Column). we can also expand(fill) multiple children, in such case the available space is divided among them according to the flex factor.
- An Expanded widget must be a descendant of a Row, Column, or Flex

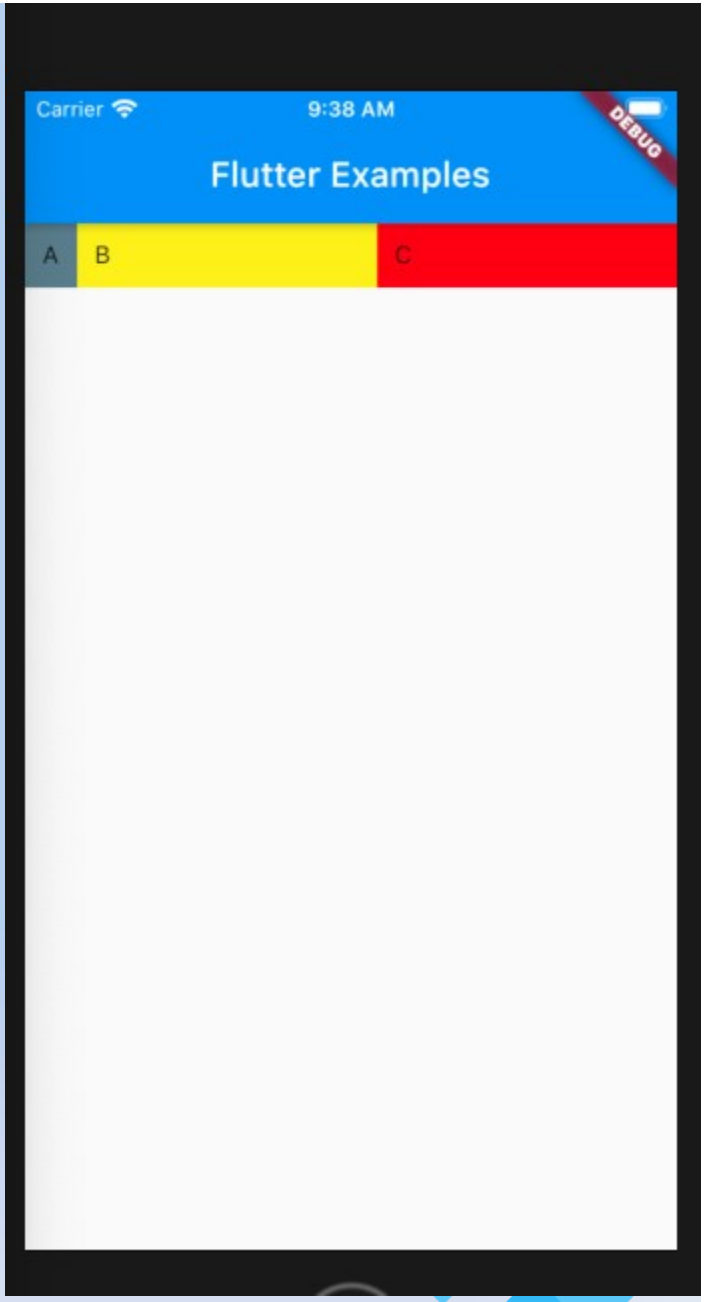
Expand

- it has two important property child and flex.
- Without giving any extra property like flex, Expanded widgets use all remaining space equally. If one child is expanded use all free space, if there are two or more children are expanded, they share all free space equally.(Default flex:1)
- If you want to give some scale to share the free space, you should use flex factor. Flex factor is a property of Expanded widget and determines the usage and sharing of the remaining space between children widgets.
- Let us see example

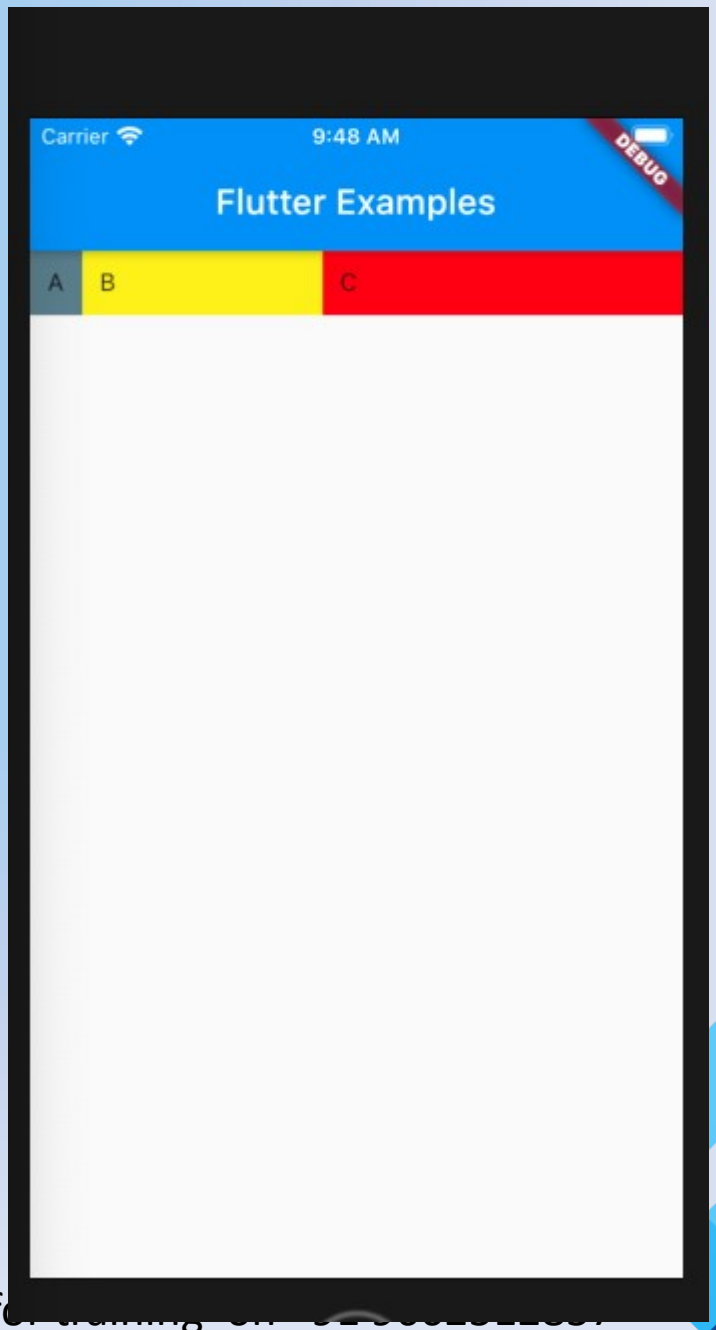
```dart
class Example extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        Container(
          padding: EdgeInsets.all(10),
          color: Colors.blueGrey,
          child: Text('A'),
        ), // Container
        Container(
          padding: EdgeInsets.all(10),
          color: Colors.yellow,
          child: Text('B'),
        ), // Container
        Expanded(
                    child: Container(
            padding: EdgeInsets.all(10),
            color: Colors.red,
            child: Text('C'),
          ), // Container
        ) // Expanded
      ], // <Widget>[]
    ); // Row
  }
}
```



```dart
class Example extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        Container(
          padding: EdgeInsets.all(10),
          color: Colors.blueGrey,
          child: Text('A'),
        ), // Container
        Expanded(
          child: Container(
            padding: EdgeInsets.all(10),
            color: Colors.yellow,
            child: Text('B'),
          ), // Container
        ), // Expanded
        Expanded(
          child: Container(
            padding: EdgeInsets.all(10),
            color: Colors.red,
            child: Text('C'),
          ), // Container
        ) // Expanded
      ], // <Widget>[]
    ); // Row
  }
}
```



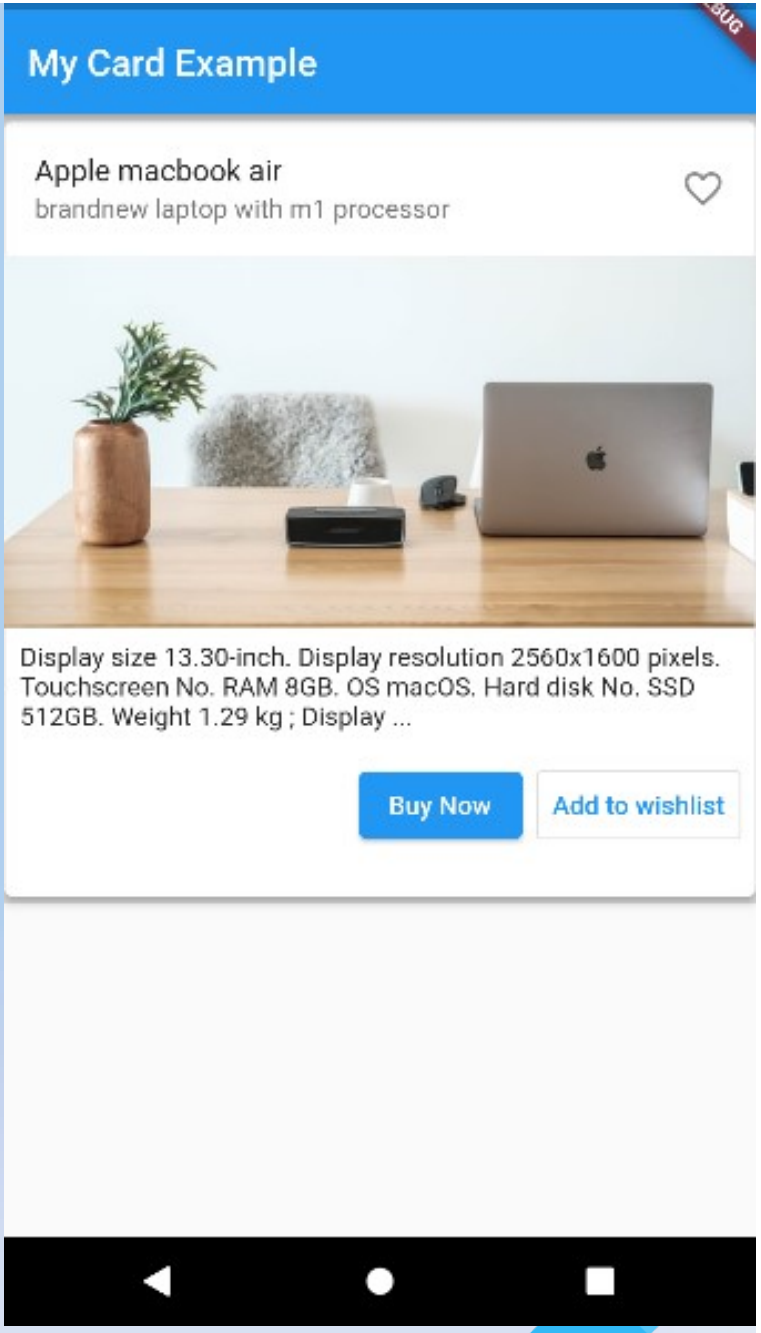
```dart
class Example extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        Container(
          padding: EdgeInsets.all(10),
          color: Colors.blueGrey,
          child: Text('A'),
        ), // Container
        Expanded(
          flex: 2,
          child: Container(
            padding: EdgeInsets.all(10),
            color: Colors.yellow,
            child: Text('B'),
          ), // Container
        ), // Expanded
        Expanded(
          flex: 3,
          child: Container(
            padding: EdgeInsets.all(10),
            color: Colors.red,
            child: Text('C'),
          ), // Container
        ) // Expanded
      ], // <Widget>[]
    ); // Row
  }
}
```



Card

- A material design card is a panel with slightly rounded corners and an elevation shadow.
- A card is a sheet of Material used to show related information, for example an album, product, contact details, etc.
- Card must be used inside material.
- To read more about card refer below link
- https://material.io/components/cards/flutter#card

```dart
Card buildCard() {
  var heading = 'Apple macbook air';
  var subheading = 'brandnew laptop with m1 processor ';
  var cardImage = NetworkImage(
      'https://source.unsplash.com/random/600x300?macbook');
  var supportingText =
      'Display size 13.30-inch. Display resolution 2560x1600 pixels. '
      'Touchscreen No. RAM 8GB. OS macOS. Hard disk No. SSD 512GB. Weight 1.29 kg ; Display ...';

  return Card(
      elevation: 4.0,
      child: Column(
        children: [
          ListTile(
            title: Text(heading),
            subtitle: Text(subheading),
            trailing: Icon(Icons.favorite_outline),
          ),
          Container(
            height: 200.0,
            child: Ink.image(
              image: cardImage,
              fit: BoxFit.cover,
            ),
          ),
          Container(
            padding: EdgeInsets.all(8.0),
            child: Text(supportingText),
          ),
          ButtonBar(
            children: [
              ElevatedButton(
                child: const Text('Buy Now'),
                onPressed: () {},
              ),
              OutlineButton(
                child: const Text('Add to wishlist'),
                onPressed: () {},
              )
            ],
          )
        ],
      ));
}
```

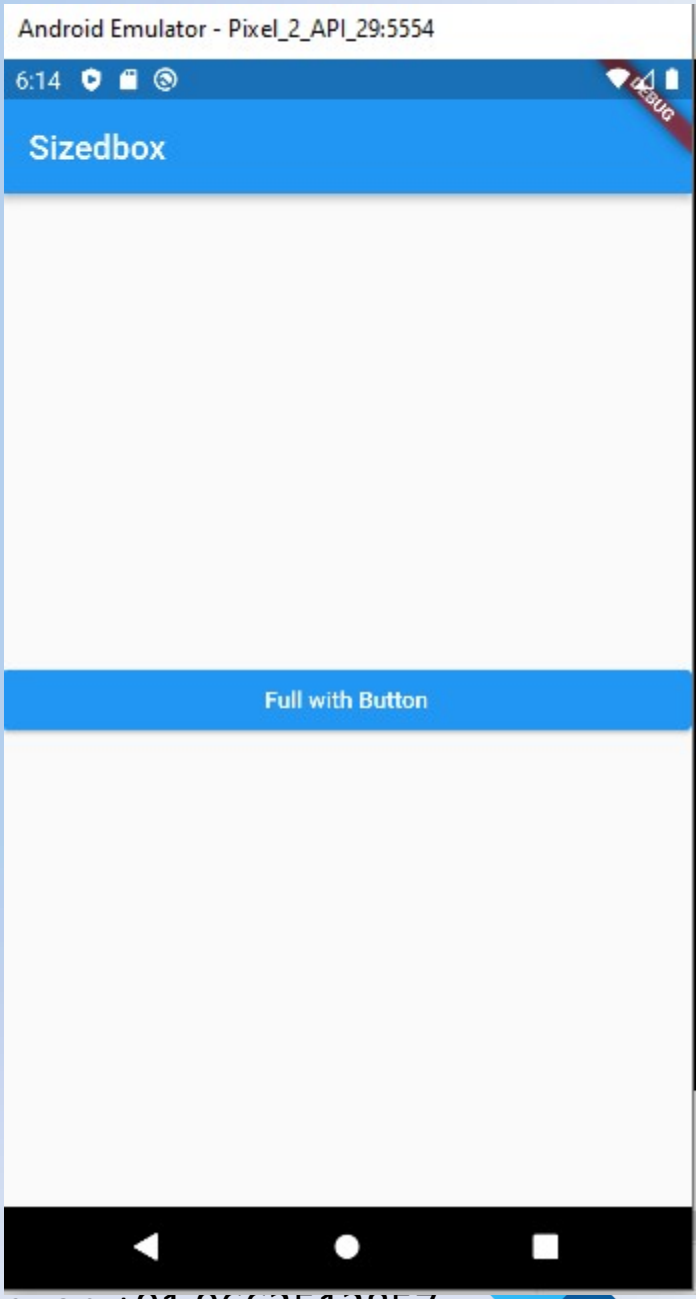


SizedBox

- **SizedBox Widget** is a simple box with a desired height and width.
- It has 3 important optional properties height, width and child.
- If you specify a particular size for the **SizedBox**, that size will also apply to its child **widget**.
- if the width of **SizedBox** is not specified or **null** then its child widget will have width by its own setting or equal to 0 (if not set). height also has similar behavior.

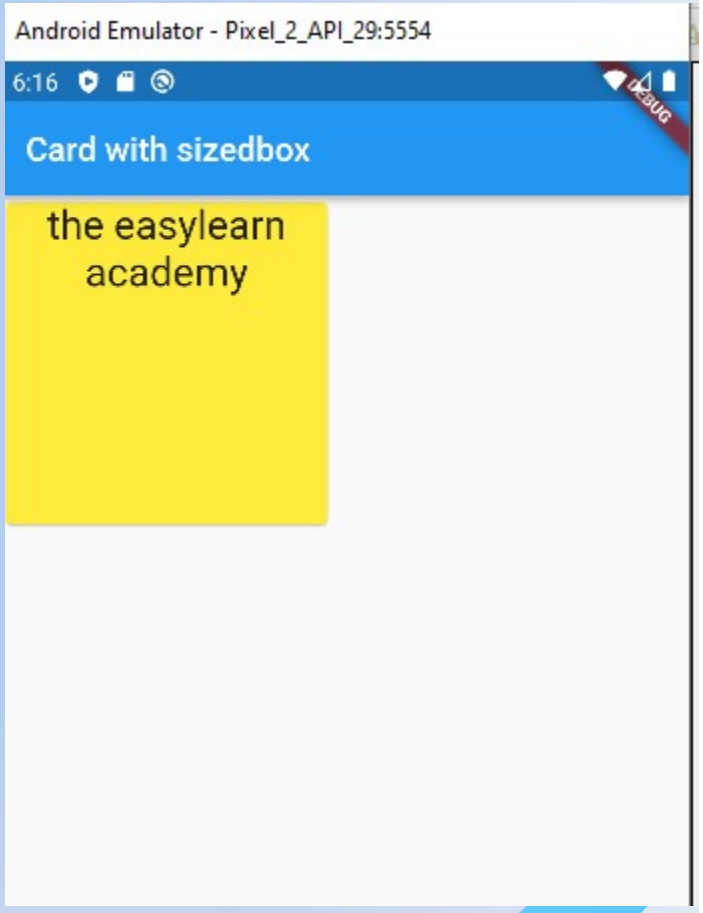
First example

```
class Sizebox1 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
   return  MaterialApp(
     home: Scaffold(
       appBar: AppBar(
         title: Text("Sizedbox"),
       ),
       body: Material(
         child: Center(
           child: SizedBox(
             width: double.infinity,
             child: ElevatedButton(
               onPressed: () {
                 print("Hello");
               },
               child: Text("Full with Button",textDirection: TextDirection.ltr,),
             ),
           ),
         ),
       ),
     ),
   );
  }
}
```



Second example

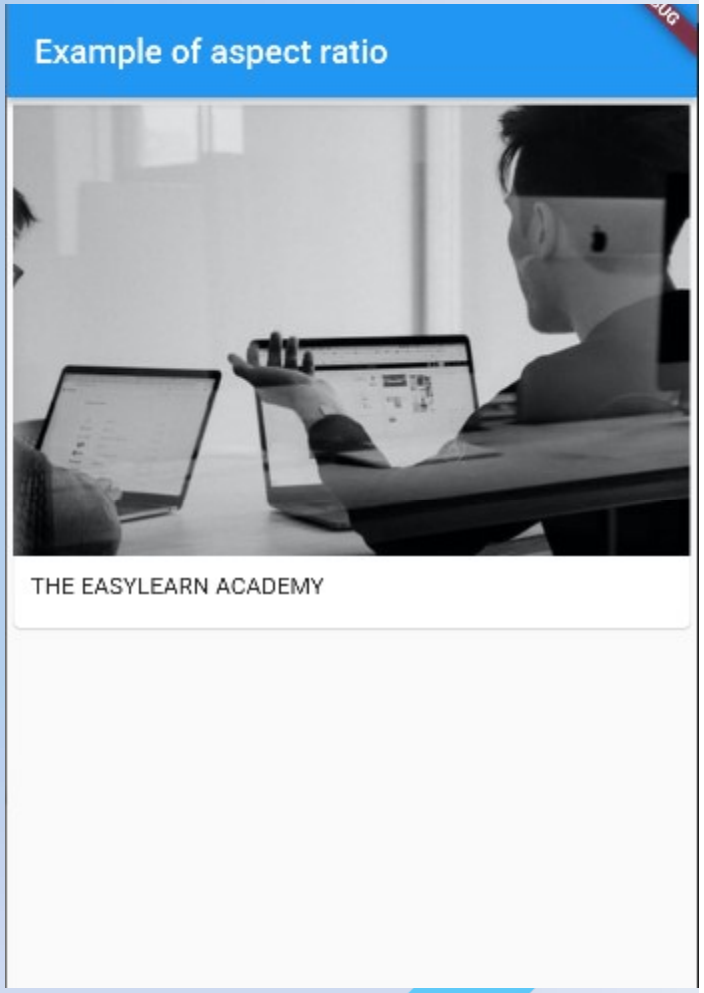
```
class SizedBox2 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Sized Box 2",
      home: Scaffold(
        appBar: AppBar(title: Text("Card with sizedbox"),),
        body: Material(
          child: Container(
            child: SizedBox(
              height: 200,
              width: 200,
              child: Card(
                color: Colors.yellow,
                  child: Text("the easylearn academy",
                    textAlign: TextAlign.center,
                    textDirection: TextDirection.ltr,
                    style: TextStyle(
                      fontSize: 24,
                    ),),
                )
                ,)
              ,),
          ),
        ),
    );
  }
}
```



AspectRatio

- Sometimes, you may care about the proportions of a widget more than its size.
- For example, you want the width to be always twice as large as the height. In such case use widget called AspectRatio.
- It requires you to pass a named argument aspectRatio whose type is double.
- The passed value is for the width, height ratio will be calculated & applied to the Widget passed as child.
- For better understanding, you are recommended to pass the aspectRatio value as a fraction instead of a decimal.

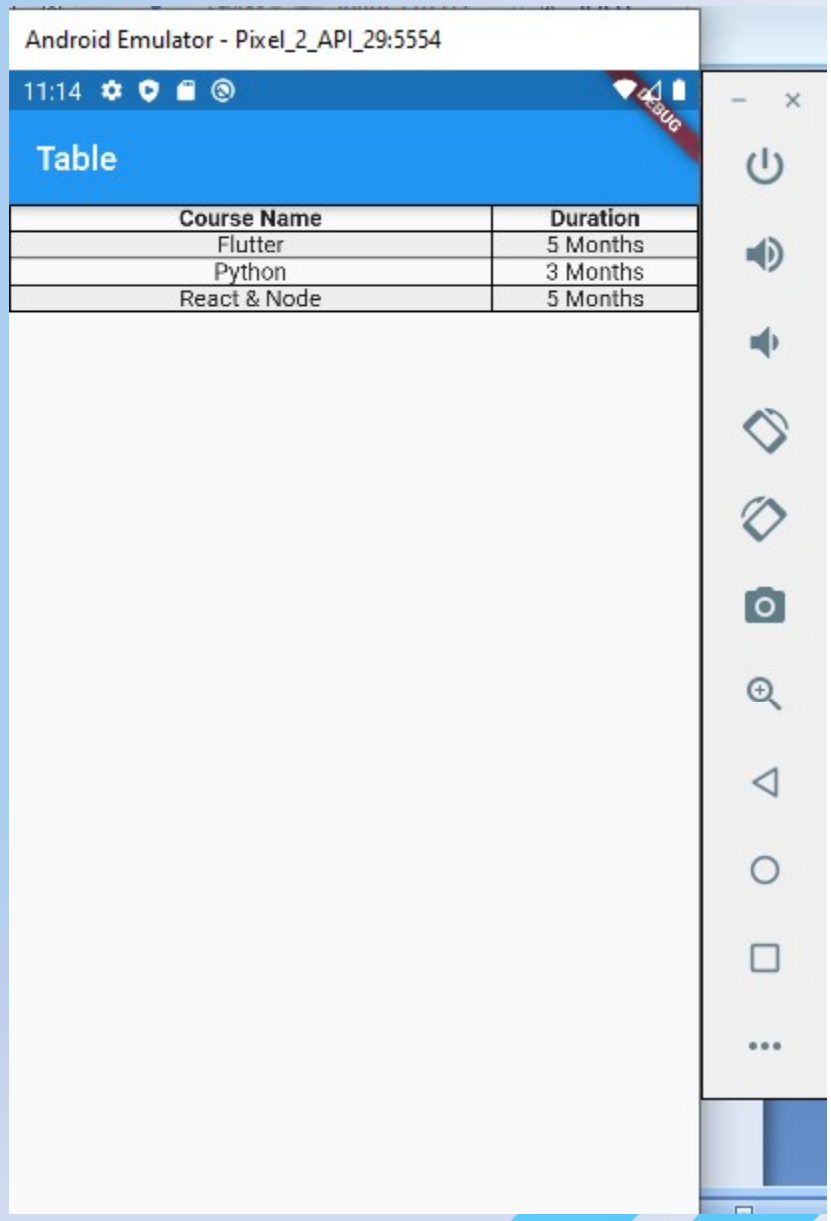
```dart
class aspectratio1 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Aspect ratio",
      home: Scaffold(
        appBar: AppBar(title: Text("Example of aspect ratio"),),
        body: Material(
          child: SizedBox(
            width: double.infinity,
            height: 320,
            child: Card(
              child: Column(
                children: [
                  AspectRatio(
                    aspectRatio: 3/2,
                    child: Ink.image(
                      image: NetworkImage(
                          'https://source.unsplash.com/random/600x300?training'),
                      fit: BoxFit.cover,
                    ),
                  ),
                  Container(
                    padding: EdgeInsets.all(10),
                    alignment: Alignment.topLeft,
                    child: Text("THE EASYLEARN ACADEMY"),
                  )
                ],
              ),
            ),
          )
        ),
      ),
    );
  }
}
```



Table

- **Table** widget is used to display items in a table layout without using Rows and Columns to create a table.
- If we have multiple rows with the same width of columns then *Table widget* is the right approach.
- The height of rows in the *Table* widget is dependent on the content inside them. But the width of the column can be changed by specifying *columnWidths* property.

```dart
class table1 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Table",
      home: Scaffold(
        appBar: AppBar(
          title: Text("Table"),
        ),
        body: Material(
          child: Table(
            border: TableBorder.all(),
            defaultVerticalAlignment: TableCellVerticalAlignment.top,
              columnWidths: {
                0: FractionColumnWidth(0.7),
                1: FractionColumnWidth(0.3)
              },
            children: [
              TableRow(
                  children: [
                Text('Course Name', textAlign: TextAlign.center, style: TextStyle(fontWeight:
FontWeight.bold)),
                Text('Duration', textAlign: TextAlign.center, style: TextStyle(fontWeight:
FontWeight.bold)),
              ]),
              TableRow(
                  decoration: BoxDecoration(color: Colors.grey[200]),
                  children: [
                Text('Flutter', textAlign: TextAlign.center),
                Text('5 Months', textAlign: TextAlign.center),
              ]),
              TableRow(children: [
                Text('Python', textAlign: TextAlign.center),
                Text('3 Months', textAlign: TextAlign.center),
              ]),
              TableRow(
                  decoration: BoxDecoration(color: Colors.grey[200]),
                  children: [
                Text('React & Node', textAlign: TextAlign.center),
                Text('5 Months', textAlign: TextAlign.center),
              ]),
            ],
          ),
        ),
      ),
    );
  }
}
```

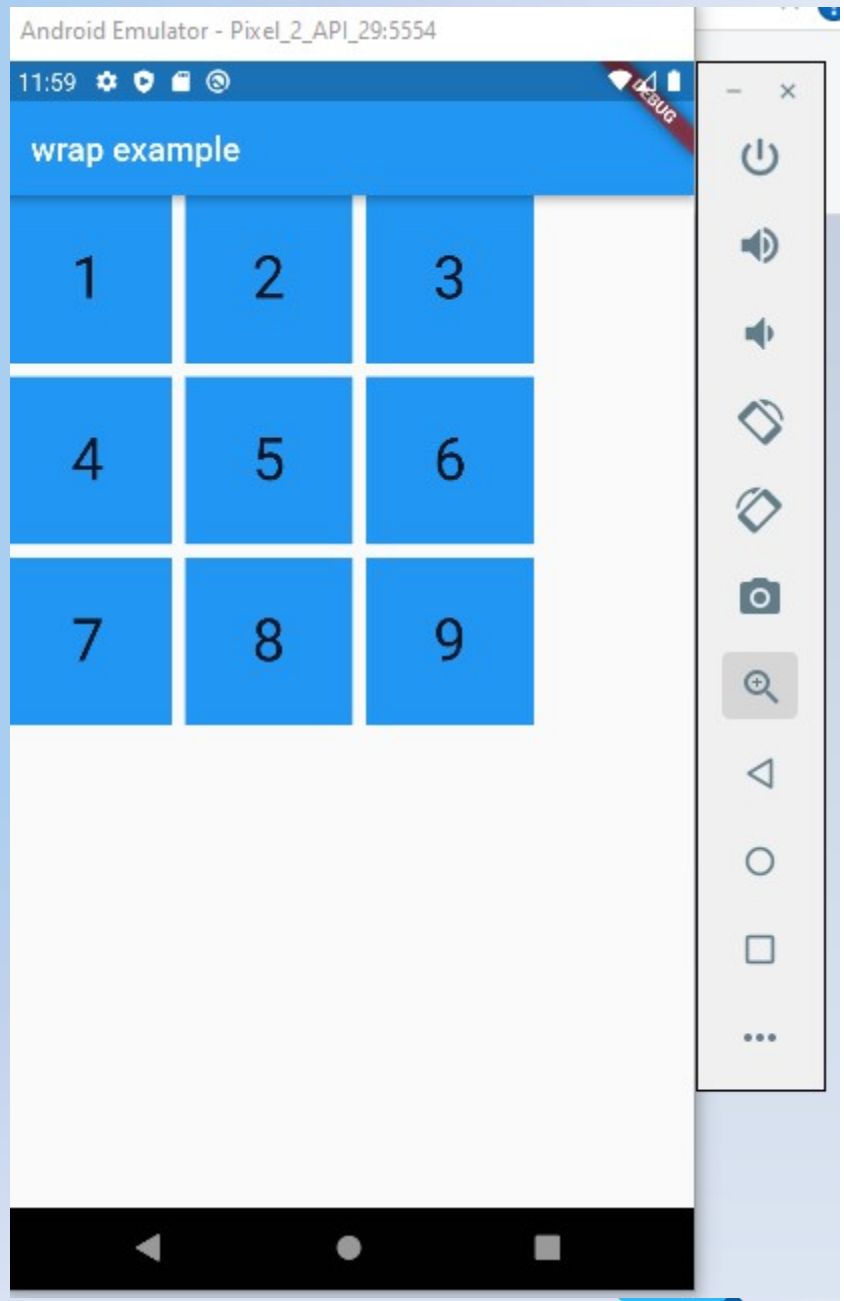


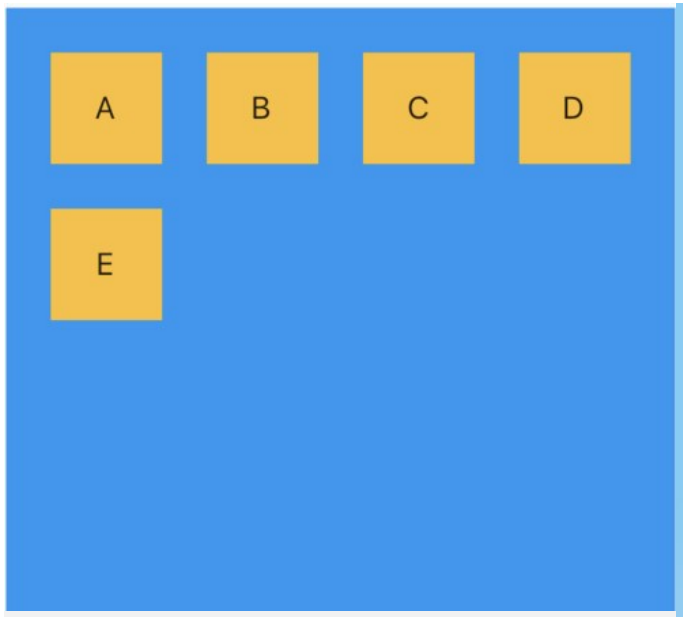
wrap

- Normally when you want to layout multiple widgets horizontally or vertically you can use a row or column.
- But if there is not enough room to display all the children then content gets clipped and you get the yellow and black overflow warning.
- To fix that you can use a Wrap widget instead of a Row.
- The default is to wrap horizontally in rows, but if you want to wrap vertically, you can set the direction.
- You can also set the alignment and spacing between the widgets. The spacing is the added space before the next widget. The runSpacing is the added space between rows or columns.

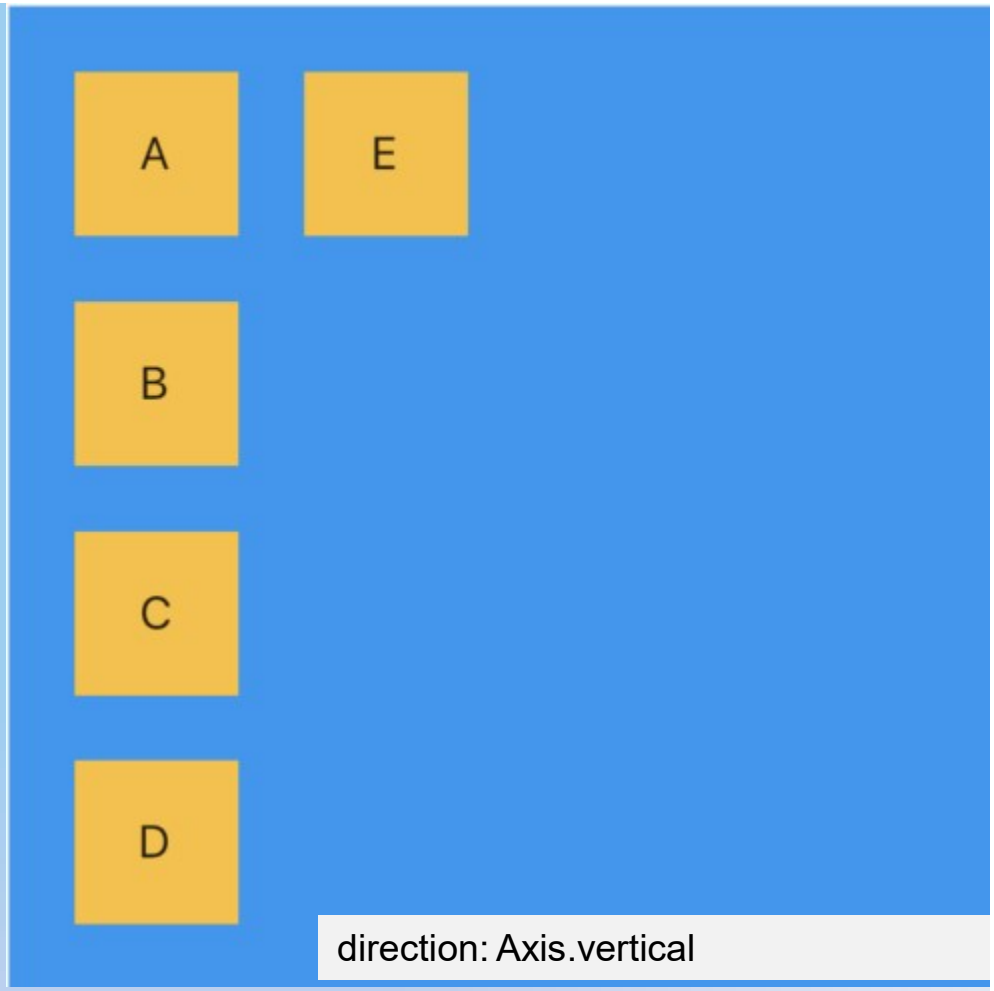
wrap

```dart
class wrap1 extends StatelessWidget {
  Container MyContainer(String letter){
    return Container(
        color: Colors.blue, width: 100, height: 100,
        child:Center(child: Text(letter,textScaleFactor: 2.5,))
    );
  }
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "wrap example",
      home: Scaffold(
        appBar: AppBar(
          title: Text("wrap example"),
        ),
        body: Material(
          child: Wrap(
            // direction: Axis.vertical,
            spacing:8.0, //horizontal spacing
            runSpacing: 8.0, //vertical spacing
             children: <Widget>[
              MyContainer('1'),
              MyContainer('2'),
              MyContainer('3'),
              MyContainer('4'),
              MyContainer('5'),
              MyContainer('6'),
              MyContainer('7'),
              MyContainer('8'),
              MyContainer('9'),

            ],
          ),
        ),
      ),
    );
  }
}
```



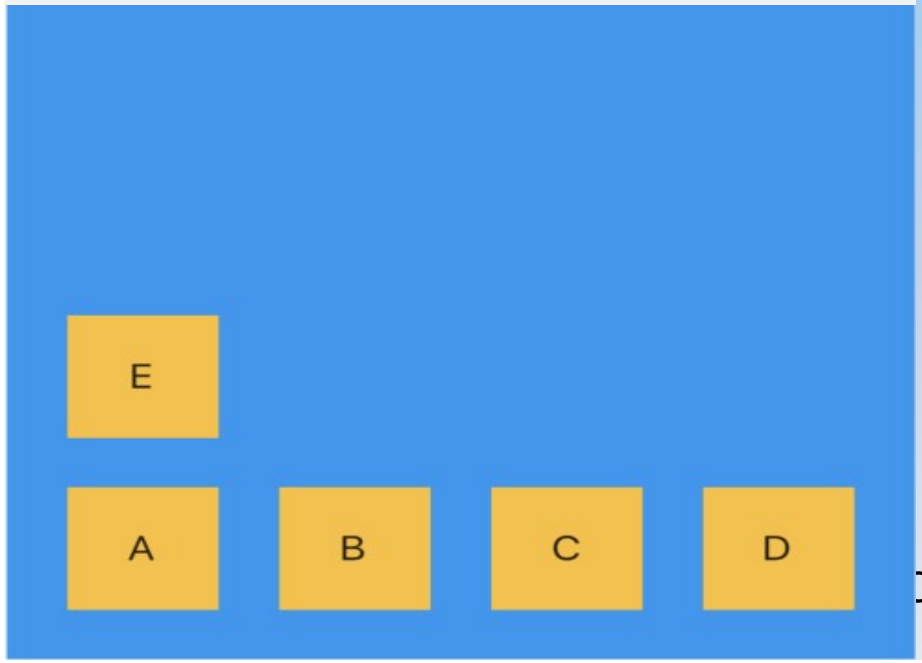


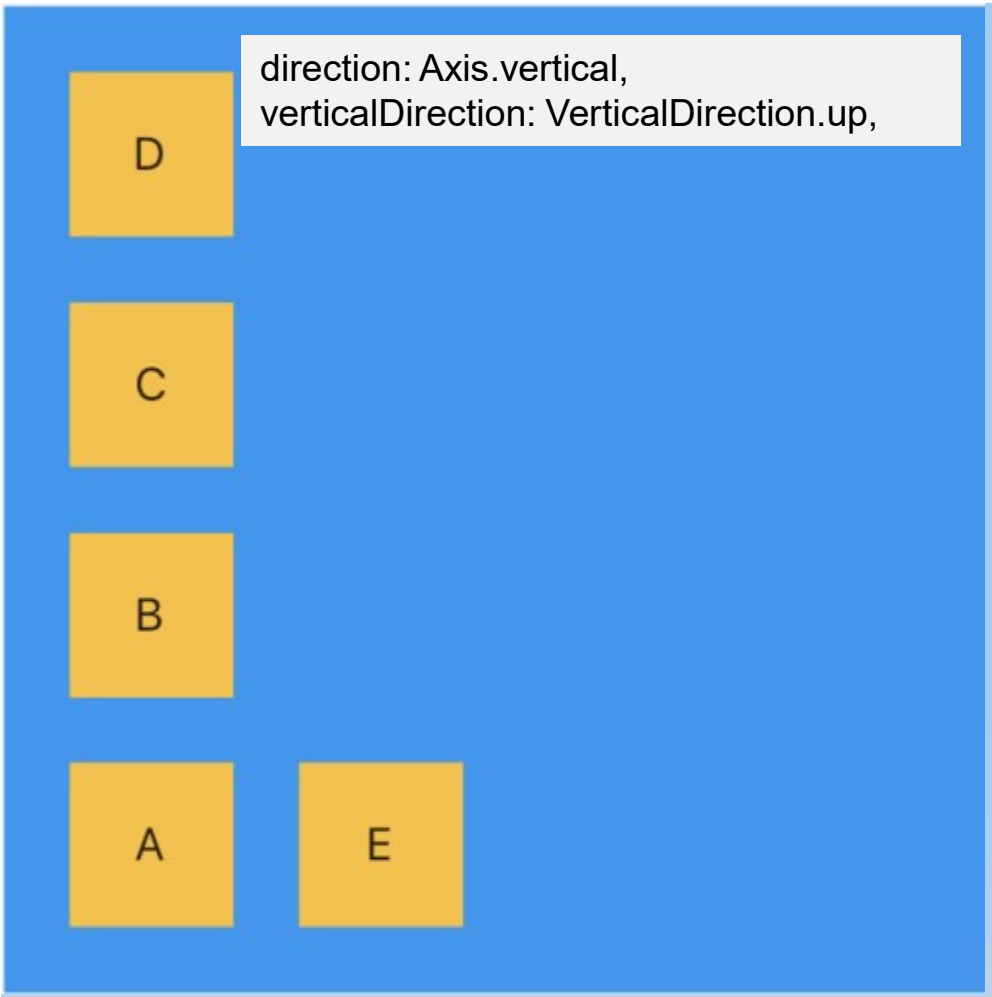
direction: Axis.horizontal, // default

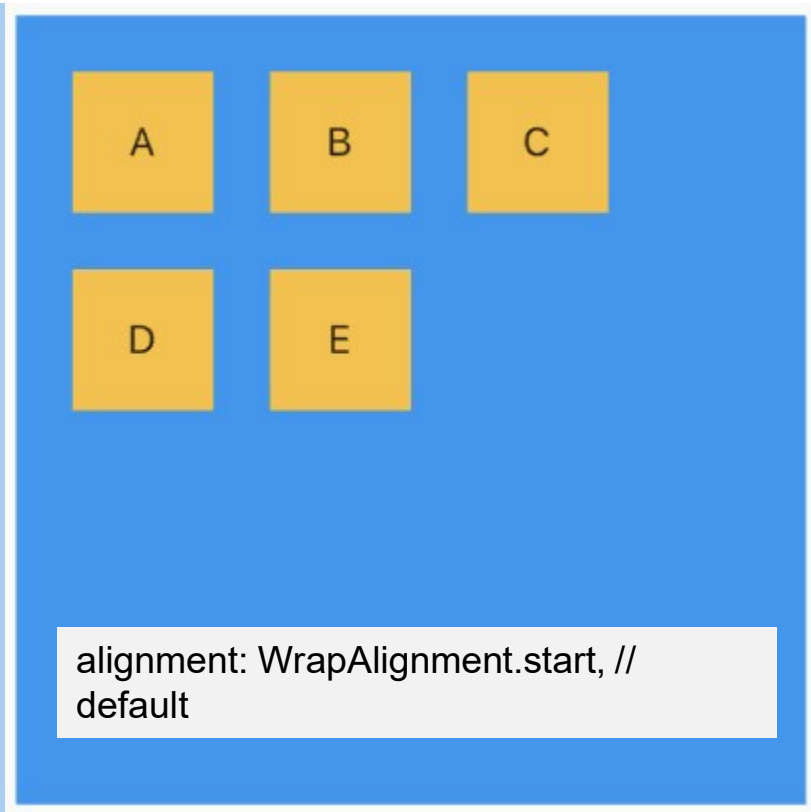


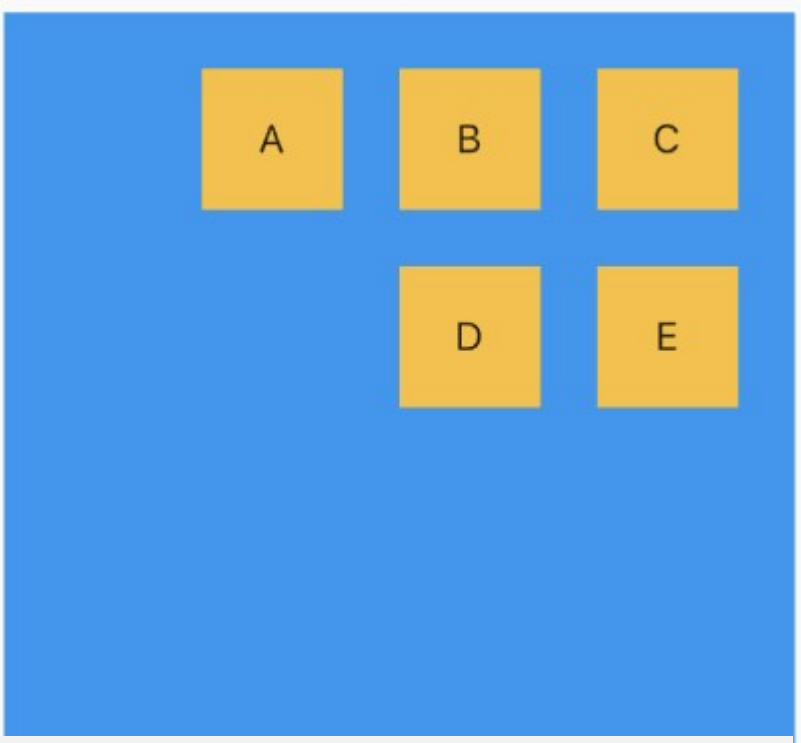
direction: Axis.vertical

verticalDirection: VerticalDirection.up,

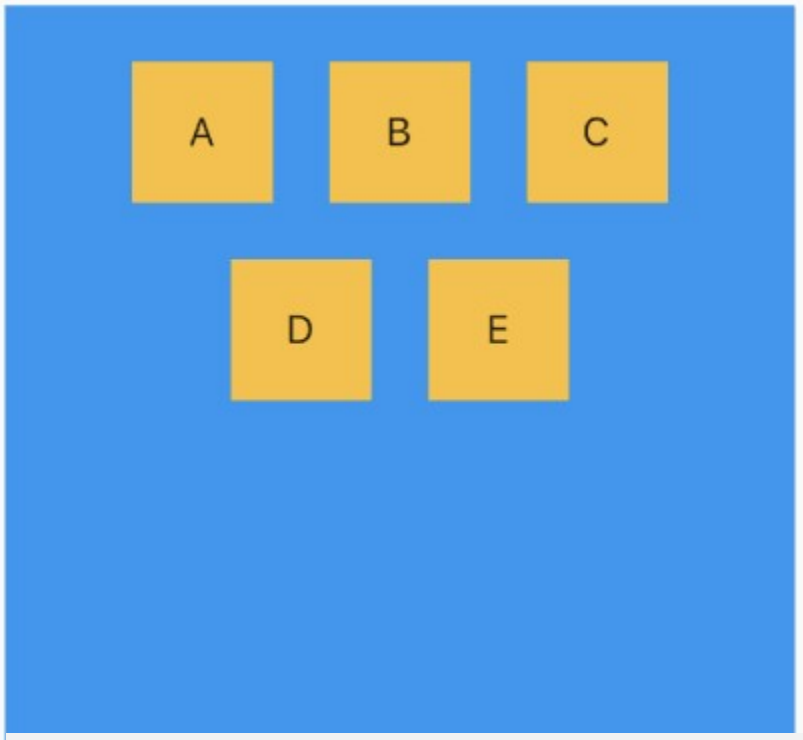



direction: Axis.vertical,
verticalDirection: VerticalDirection.up,
D
C
B
A
E

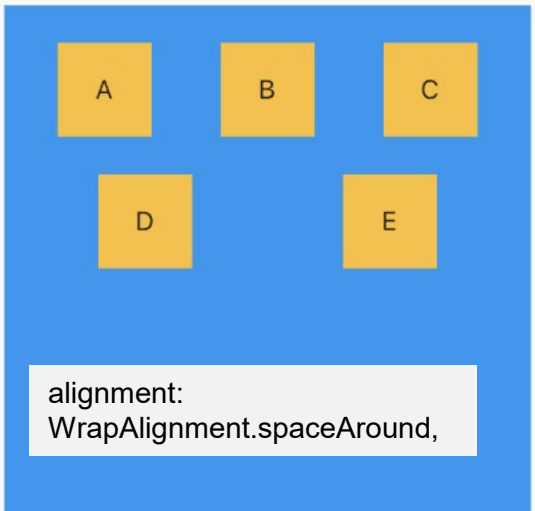

A
B
C
D
E
alignment: WrapAlignment.start, //
default



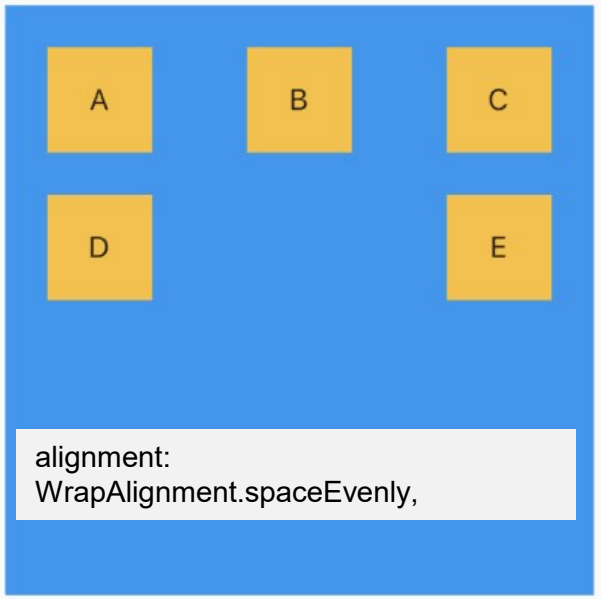
alignment: WrapAlignment.end,



alignment: WrapAlignment.center,

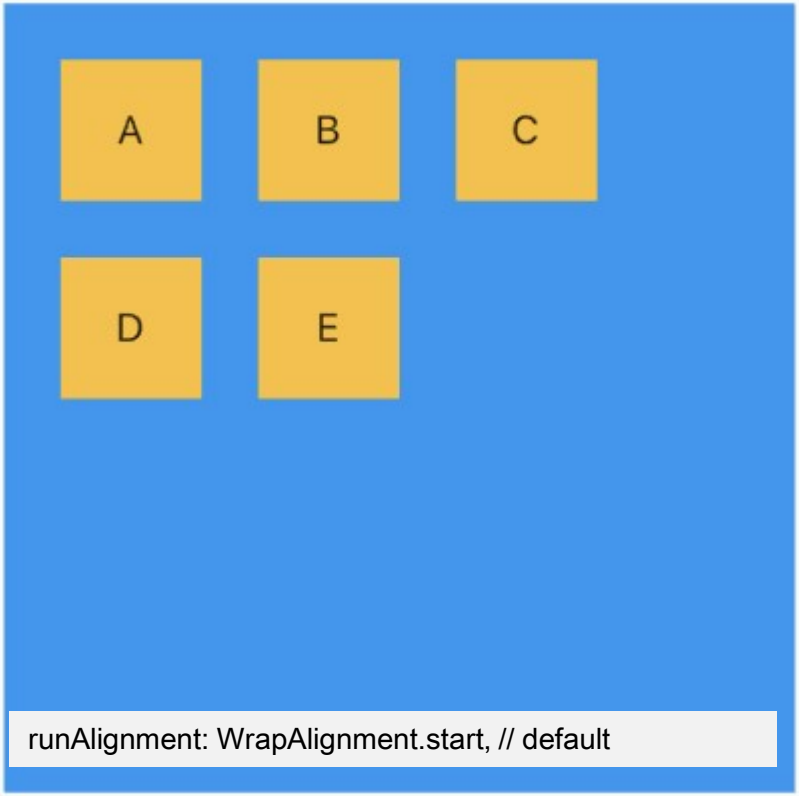


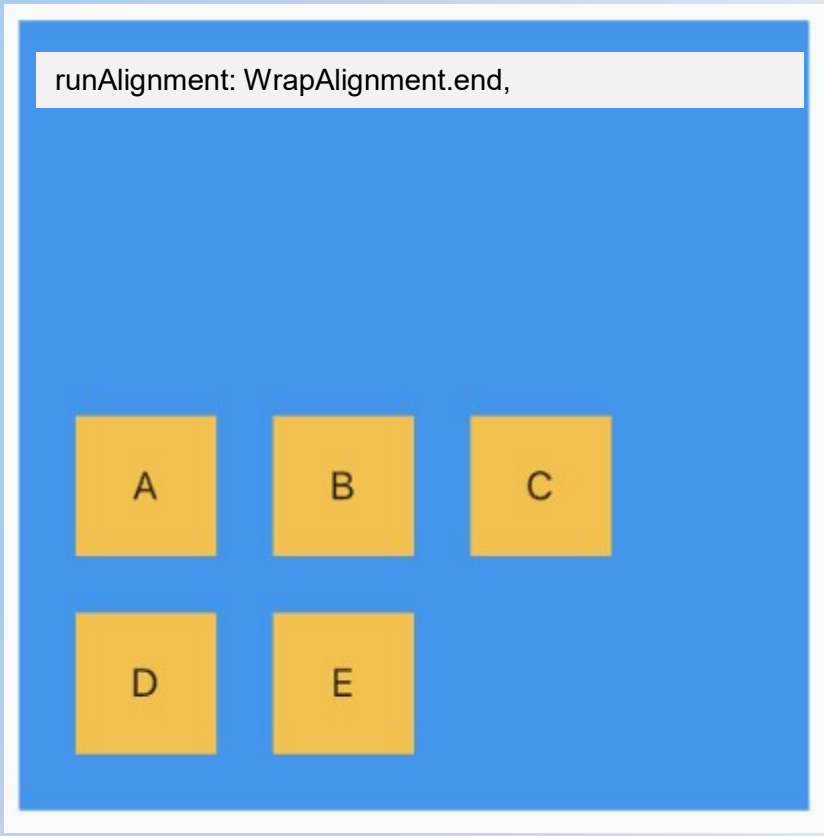
alignment: WrapAlignment.spaceAround,

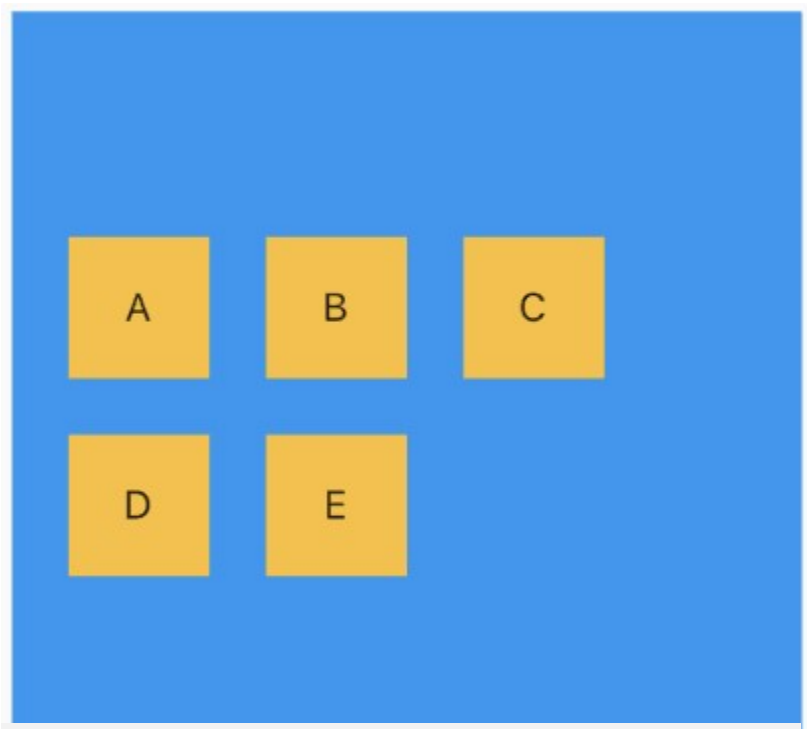


alignment: WrapAlignment.spaceEvenly,

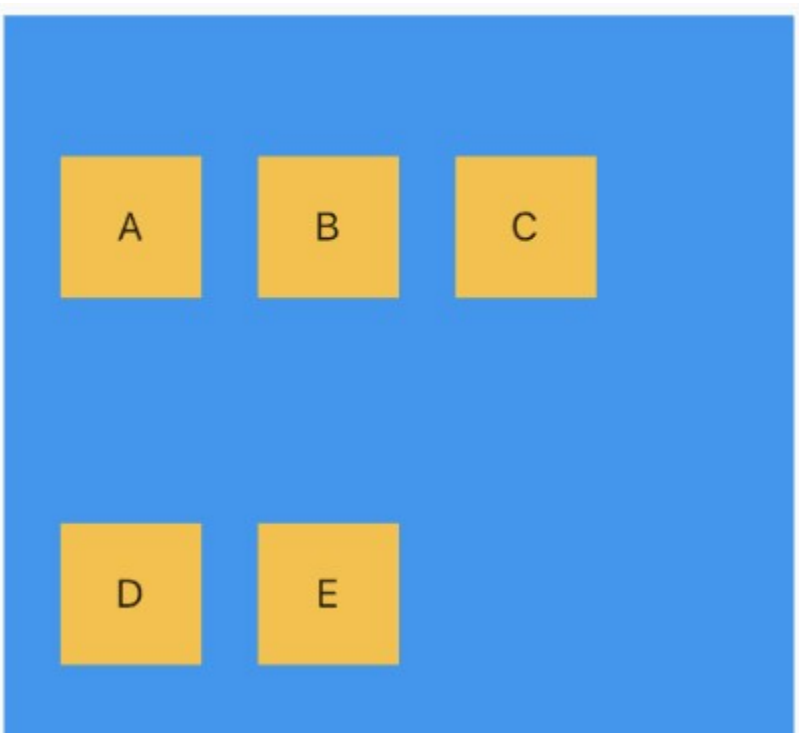
Run Alignment (vertical alignment)



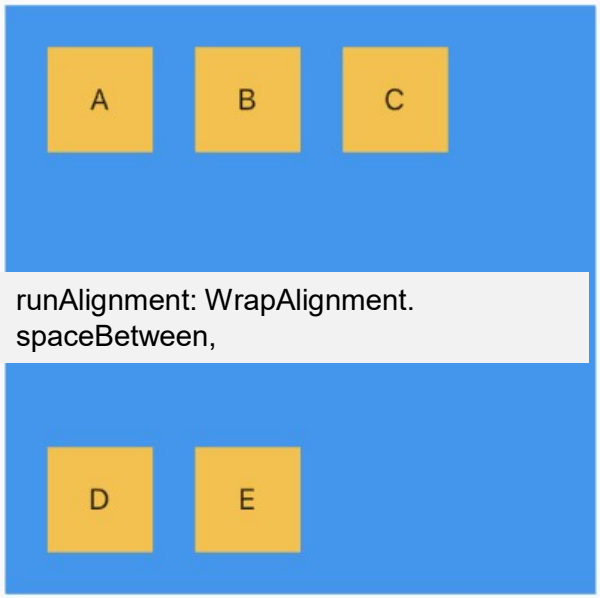




runAlignment: WrapAlignment.center,

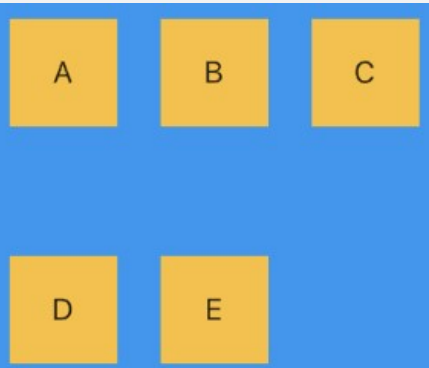


runAlignment: WrapAlignment.spaceAround,



runAlignment: WrapAlignment.
spaceBetween,

runAlignment: WrapAlignment.
spaceEvenly,



on +91 9662512857

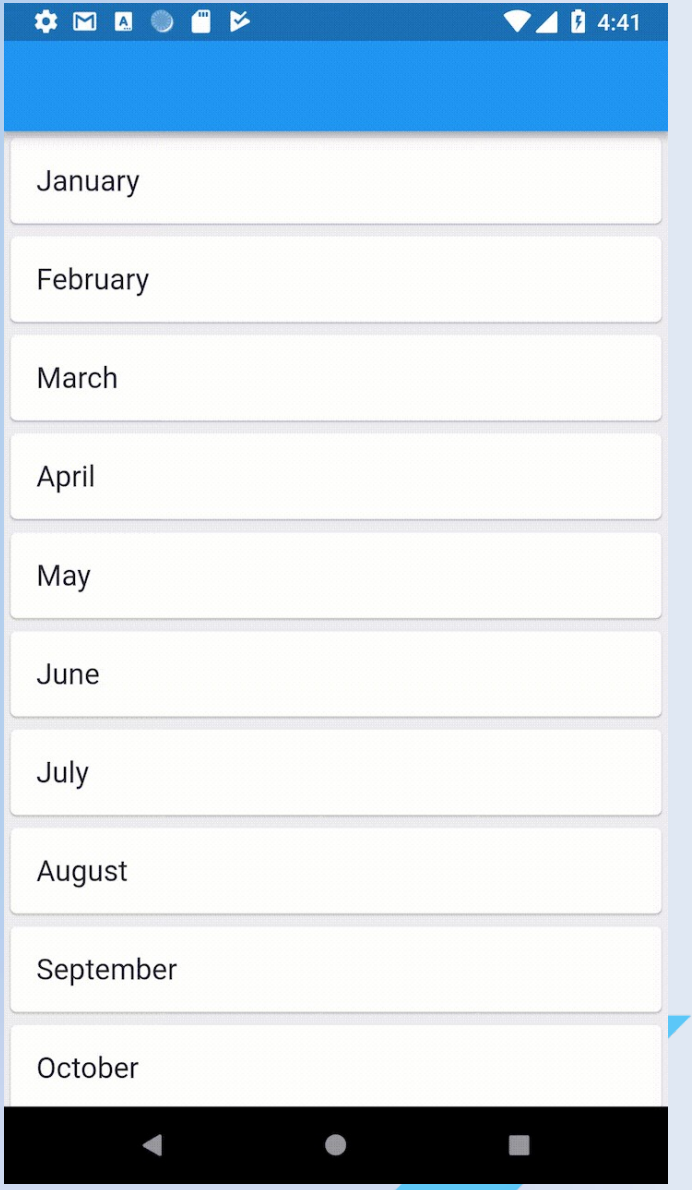
GridView

- GridView is a widget that displays a list of items as a 2D array. It means, the items are shown in a table format.
- Unlike a normal list, in which items are rendered only in one direction, GridView renders items both horizontally and vertically
- It can contain **text, images, icons**, etc depending on the user requirement. Since it is scrollable, we can specify the direction only in which it scrolls.
- The gridview can be created using any one of the various contructor, which are given below:

1. count()
2. builder()
3. custom()
4. extent()

ListView

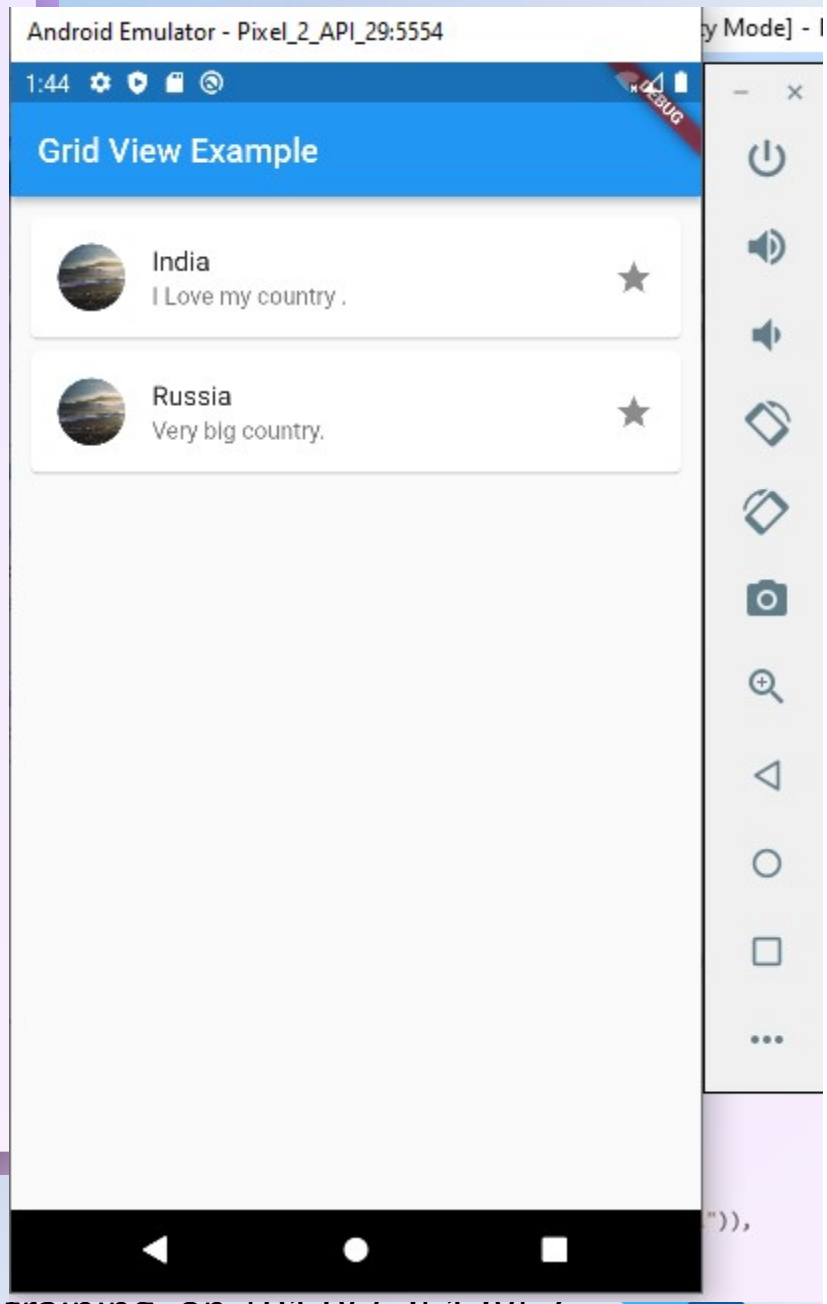
- ListView is used to group of several items in an array and display them in a scrollable list.
- The list can be scrolled vertically, horizontally, or displayed in a grid:



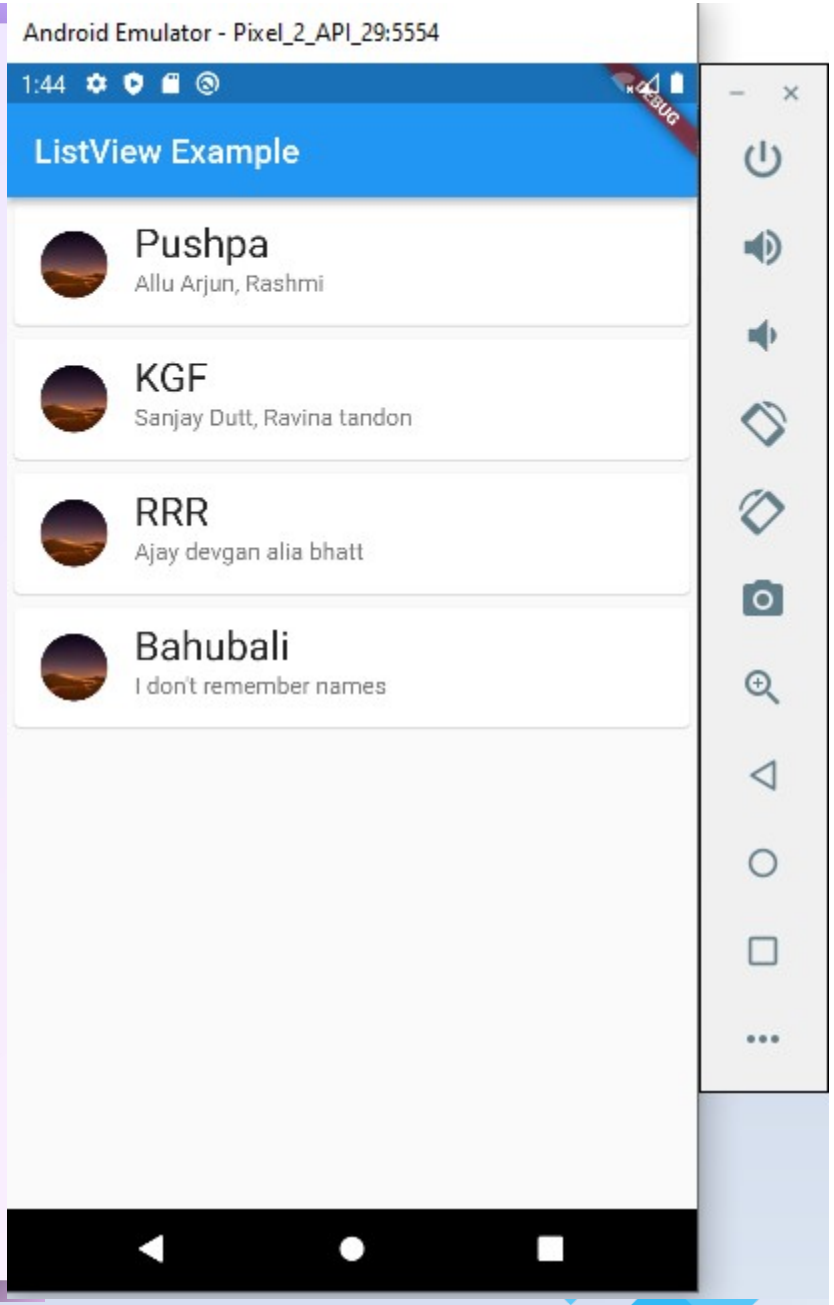
ListView properties

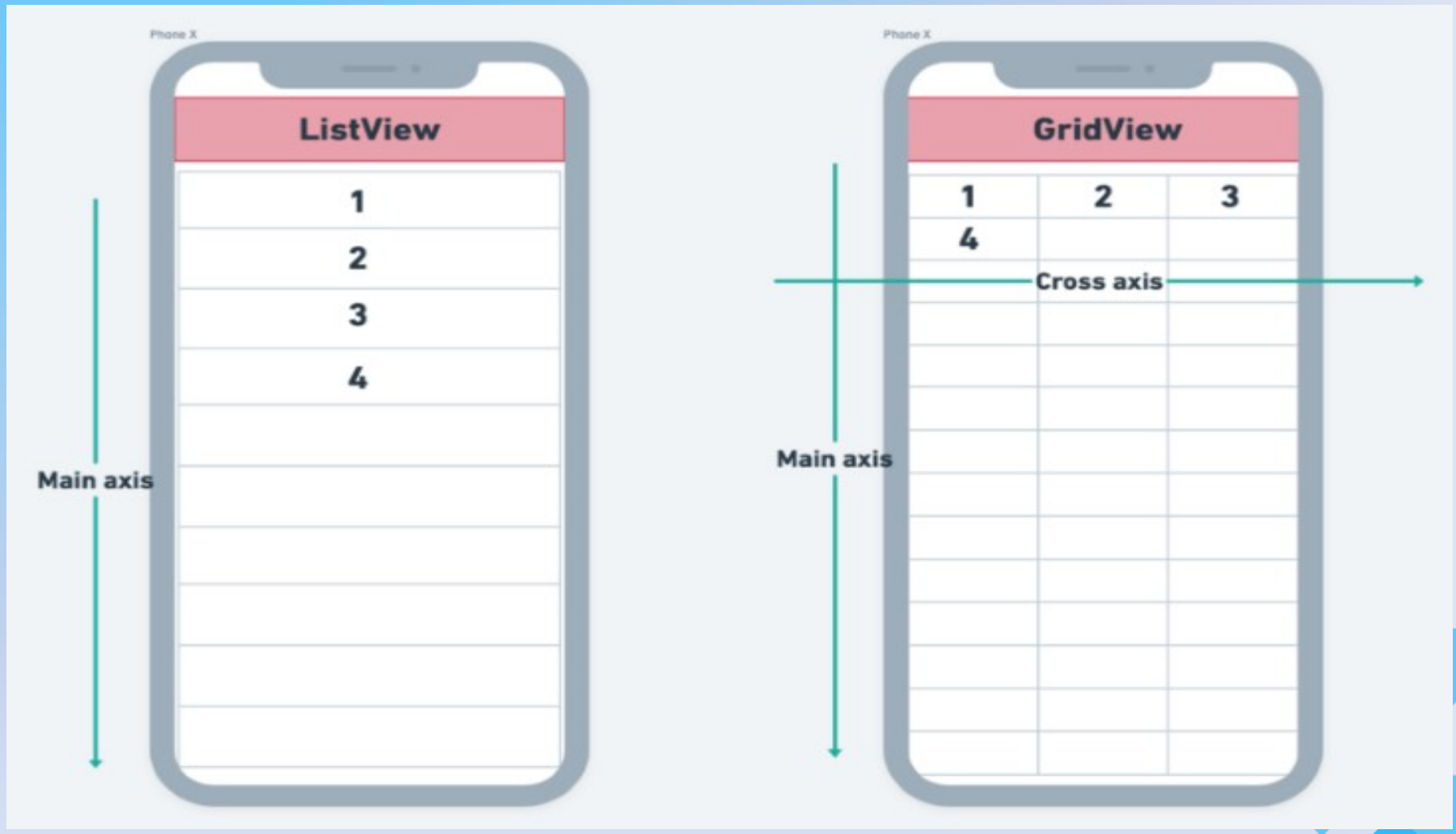
- **childrenDelegate:** This property takes *SliverChildDelegate* as the object. It serves as a delegate that provided the children for the *ListView*.
- **padding:** It holds *EdgeInsetsGeometryI* as the object to give space between the Listview and its children.
- **scrollDirection:** This property takes in *Axis enum* as the object to decide the direction of the scroll on the *ListView.*
- **shrinkWrap:** This property takes in a boolean value as the object to decide whether the size of the scrollable area will be determined by the contents inside the *ListView.*

```dart
class GridViewExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "",
      home: Scaffold(
          appBar: AppBar(
            title: Text("Grid View Example"),
          ),
          body: ListView(
            padding: const EdgeInsets.all(8),
            children: <Widget>[
              Card(
                  child: ListTile(
                      title: Text("India"),
                      subtitle: Text("I Love my country ."),
                      leading: CircleAvatar(
                          backgroundImage:
                              NetworkImage("http://picsum.photos/64?random=1")),
                      trailing: Icon(Icons.star))),
              Card(
                  child: ListTile(
                      title: Text("Russia"),
                      subtitle: Text("Very big country."),
                      leading: CircleAvatar(
                          backgroundImage:
                              NetworkImage("http://picsum.photos/64?random=1")),
                      trailing: Icon(Icons.star))),
            ],
          )),
    );
  }
}
```



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

class GridViewExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "",
      home: Scaffold(
          appBar: AppBar(
            title: Text("ListView Example"),
          ),
          body: ListViewHome()
      )
    );
  }
}
class ListViewHome extends StatelessWidget {
  final titles = ["Pushpa", "KGF", "RRR", "Bahubali",];
  final subtitles = ["Allu Arjun, Rashmi",
    "Sanjay Dutt, Ravina tandon", "Ajay devgan alia bhatt","I don't remember names"];
  @override
  Widget build(BuildContext context) {
    return ListView.builder(
        itemCount: titles.length,
        itemBuilder: (context, index) {
          return Card(
              child: ListTile(
                  title: Text(titles[index],textScaleFactor: 1.5,),
                  subtitle: Text(subtitles[index]),
                  leading: CircleAvatar(
                      backgroundImage: NetworkImage(
                          "https://picsum.photos/64")),
                  ));
        });
  }
}
```

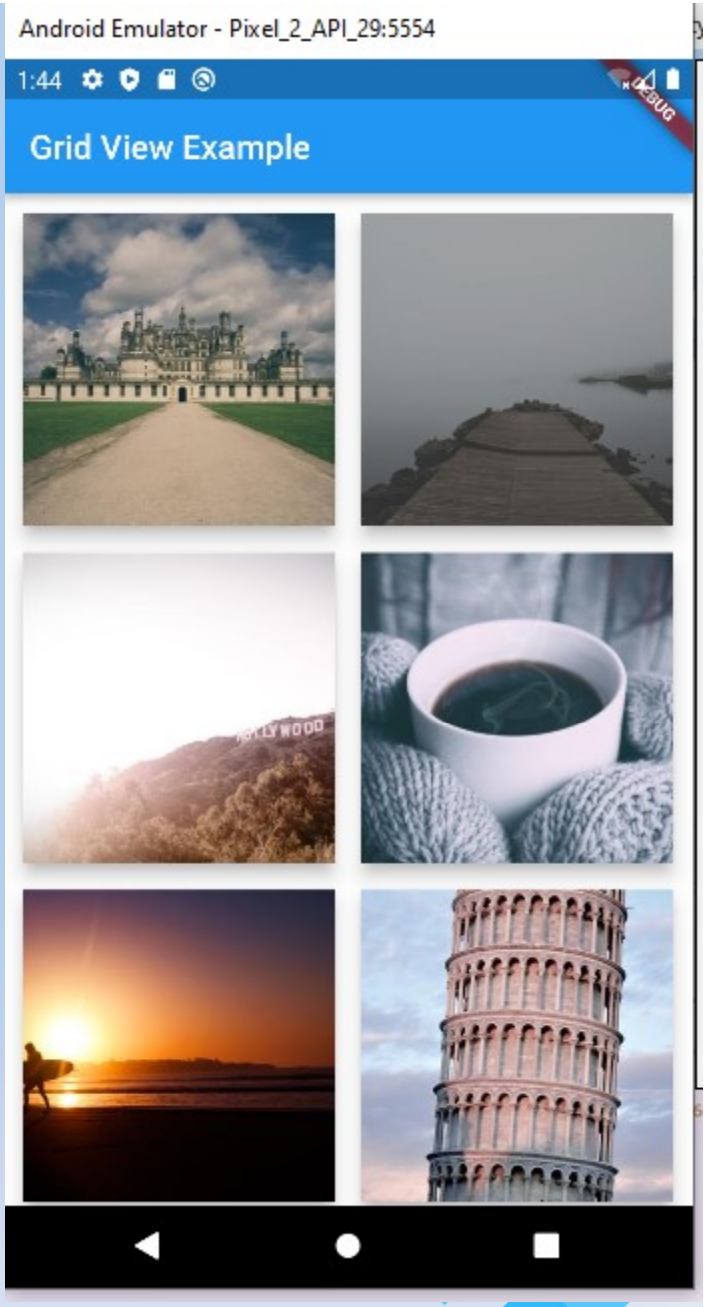



Phone X
ListView
1
2
3
4
Main axis
Phone X
GridView
1
2
3
4
Cross axis
Main axis

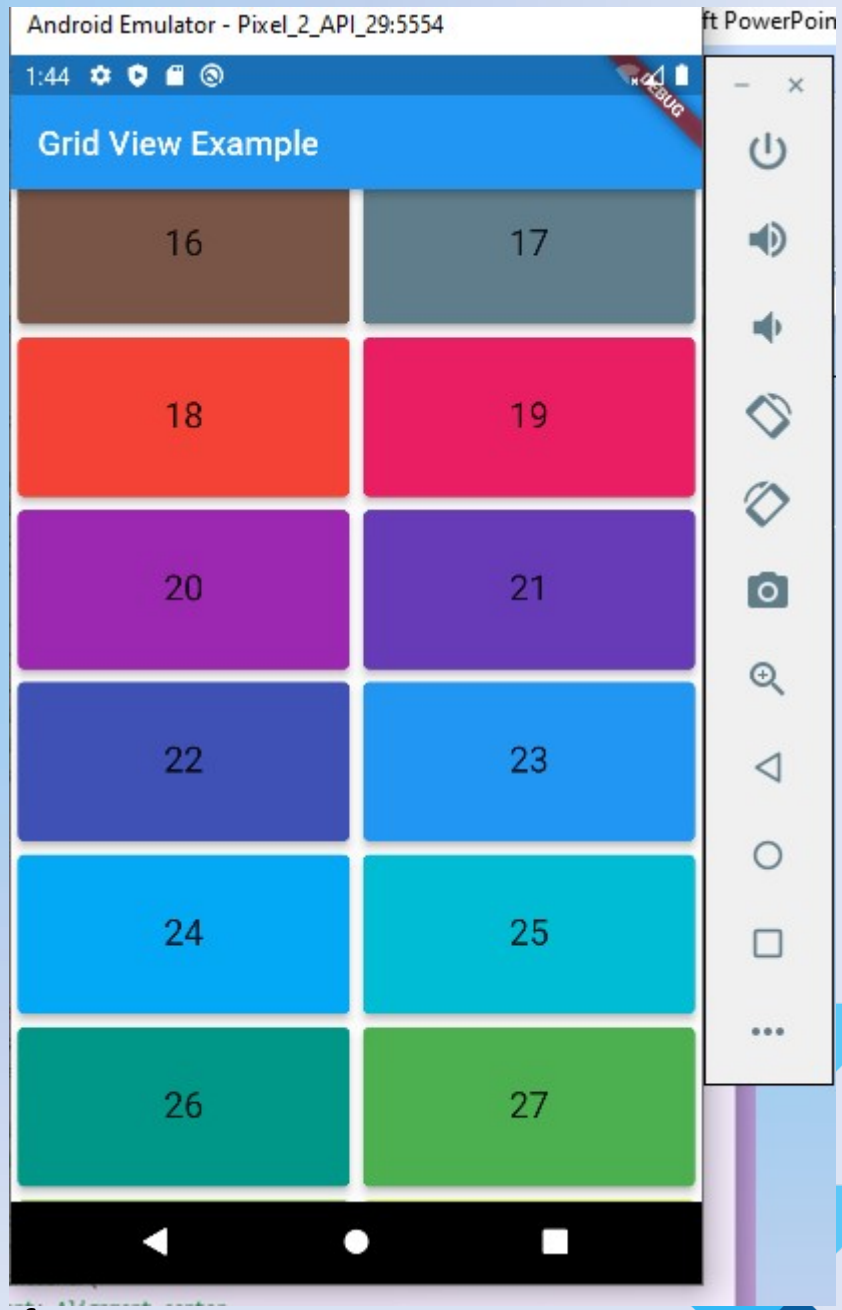
Important properties of GridView constructor

- **gridDelegate** is a property that controls how items are shown in a list. normally it SliverGridDelegateWithFixedCrossAxisCount() with crossAxisCount set to 2 or 3 or 4.
- That means we want to display 2/3/4 items horizontally if the scroll direction is vertical. The default scroll direction for any list is vertical only, so the items are shown horizontally.
- **children** refers to the list of items given here. It accepts a list of any widgets so you can show anything you would like to appear on the screen
- **crossAxisSpacing** this property allows you to place a space between items on the cross axis. That means space will appear horizontally if the scroll direction is vertical.
- **mainAxisSpacing** refers space between two items vertically.

```dart
class GridViewExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "", home: Scaffold(
        appBar: AppBar(title: Text("Grid View Example"),
        ),
        body: Material(color: Colors.white30,
          child: GridView(
            padding: EdgeInsets.all(8),
            gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
              crossAxisCount: 2, crossAxisSpacing: 8, mainAxisSpacing: 8,
            ),
              children: [
                Card(elevation:8, child: Image.network('https://picsum.photos/300?random=1')),

                Card(elevation:8, child: Image.network('https://picsum.photos/300?random=2')),

                Card(elevation:8, child: Image.network('https://picsum.photos/300?random=3')),

                Card(elevation:8, child: Image.network('https://picsum.photos/300?random=4')),

                Card(elevation:8, child: Image.network('https://picsum.photos/300?random=5')),

                Card(elevation:8, child: Image.network('https://picsum.photos/300?random=6')),

              ],
          ),
        ),
      ),
    );
  }
}
```

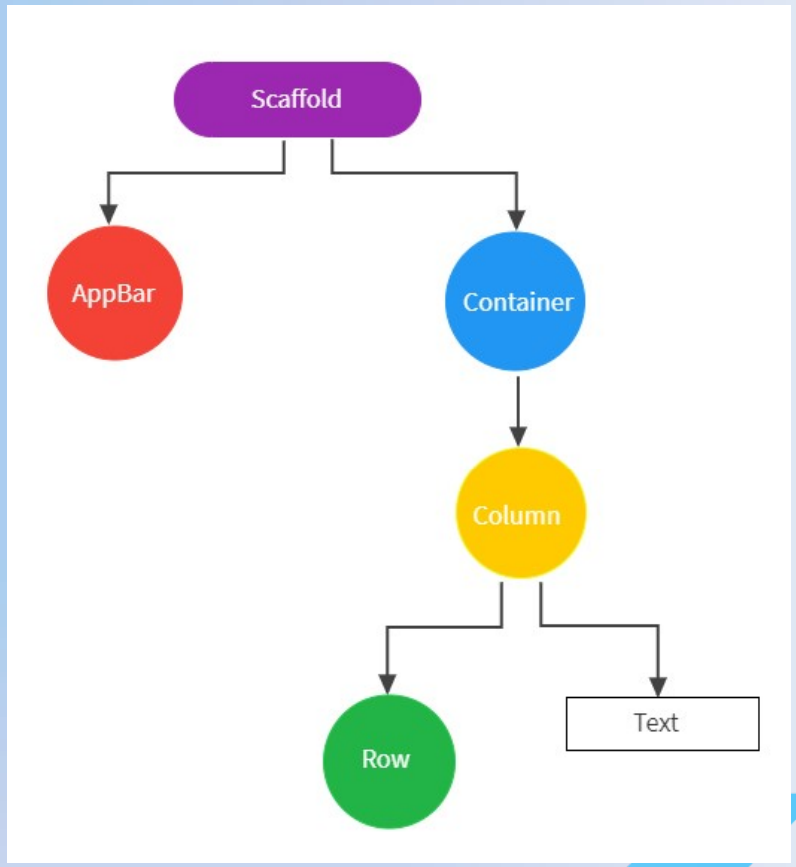


```dart
class GridViewExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "", home: Scaffold(
        appBar: AppBar(title: Text("Grid View Example"),
        ),
        body: Material(color: Colors.white30,
          child: GridView.builder(
            itemCount: 30,
            itemBuilder: (context, index) => ItemTile(index),
            gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
              crossAxisCount: 2, childAspectRatio: 2,
            ),
          )
        ),
      ),
    );
  }
}
class ItemTile extends StatelessWidget {
  final int itemNo;
  ItemTile(this.itemNo,);
  @override
  Widget build(BuildContext context) {
    final Color color = Colors.primaries[itemNo % Colors.primaries.length];
    return Card(
        color: color,
          elevation: 4,
          child: Container(
            alignment: Alignment.center,
            child: Text(
              '$itemNo',
              textScaleFactor: 1.5,
              textAlign: TextAlign.center,
            ),
          ),
    );
  }
}
```



What is BuildContext in Flutter?

- Everything in Flutter is a **widget**. Whether it is a container, text, etc., everything is virtually a widget, whether it is displayed in a UI or not.
- The UI has stacks of widgets popularly called a **widget tree**.
- A widget that renders another widget is the **parent widget**, child of parent can also render another widget.
- In right side diagram, Scaffold is the root widget, and it is the parent of all the widgets in the tree. The Scaffold widget renders the AppBar & Container widget, and the Container widget renders the Column widget. The Column widget renders both Text and Row widgets.



What is BuildContext in Flutter?

- So, we can say that Scaffold is the parent of Container, and that the Container is the parent of Column and the child of Scaffold. The Column is the child of Container and parent of Text and Row . Text and FlatButton are the children of Column.
- Now that we can see the relationship, let's see how BuildContext comes in.
- **BuildContext** is a locator that is used to track each widget in a tree and locate them and their position in the tree. The BuildContext of each widget is passed to their build method. Remember that the build method returns the widget tree a widget renders.
- The BuildContext in the Scaffold widget is different from the BuildContext in the Container widget and all widgets in its tree. So, the BuildContext of the Scaffold is the BuildContext parent of the BuildContexts of its child widgets.
- Therefore all the widget has the unique buildContext in widget tree.
- **Very important point is from a child BuildContext, it is easily possible to find an ancestor (= parent) Widget.**

MediaQuery

- MediaQuery widget carries information about a piece of media, media means window, with its size or orientation.
- MediaQuery also plays an important role in getting or controlling the size of the current window of the device. it could be a browser, in case of Flutter web application. Or it could be a size of the mobile.
- MediaQuery has a properties such as size & orientation. Size contains the width and the height of the current window. to get size we use MediaQuery.of(context) function.
- if no MediaQuery is in scope, then the MediaQuery.of(context) function will throw an exception. However, we can avoid that with another function, MediaQuery.maybeOf(context).

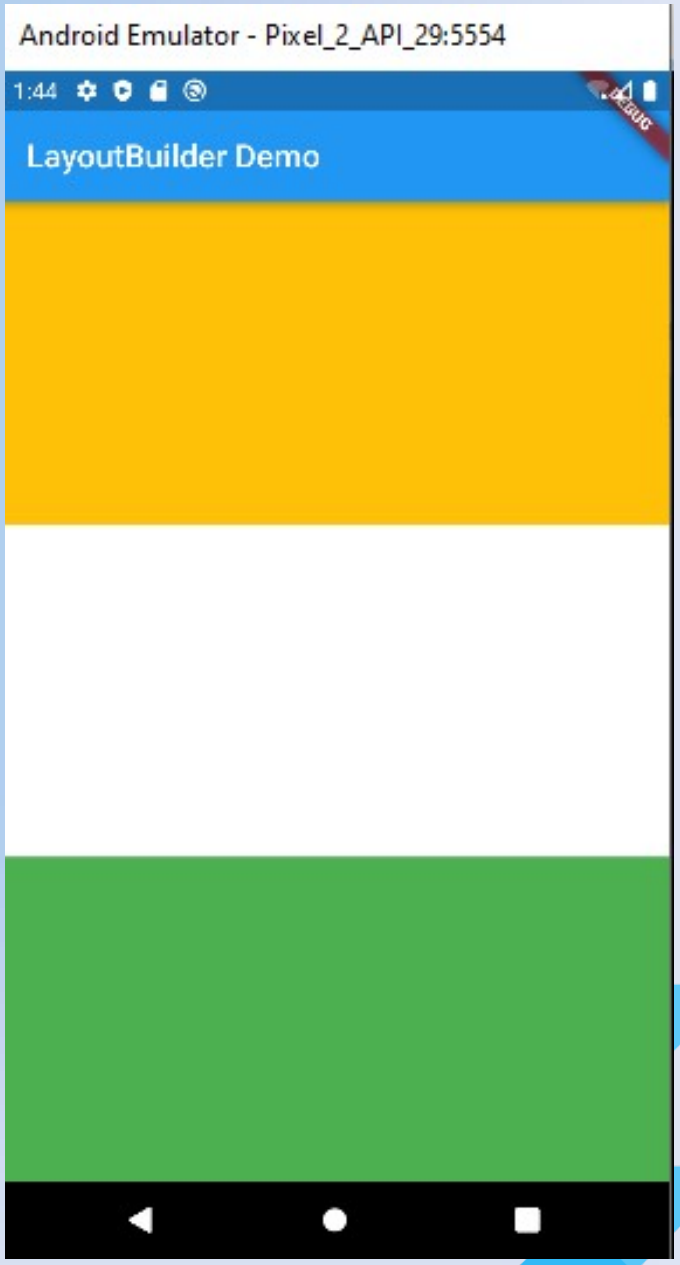
Example

```
class MediaQueryExample extends StatelessWidget {
    var size,height,width;
    @override
    Widget build(BuildContext context) {
          size = MediaQuery.of(context).size;
          height = size.height;
          width = size.width;
          return Scaffold(
                    appBar: AppBar(
                              title: Text("Example of MediaQuery"),
                              backgroundColor: Colors.red,
                    ),
                    body: Container(
                              color: Colors.yellow,
                              height: height/2,
                              width: width/2,
                    ),
          );
    }
}
```

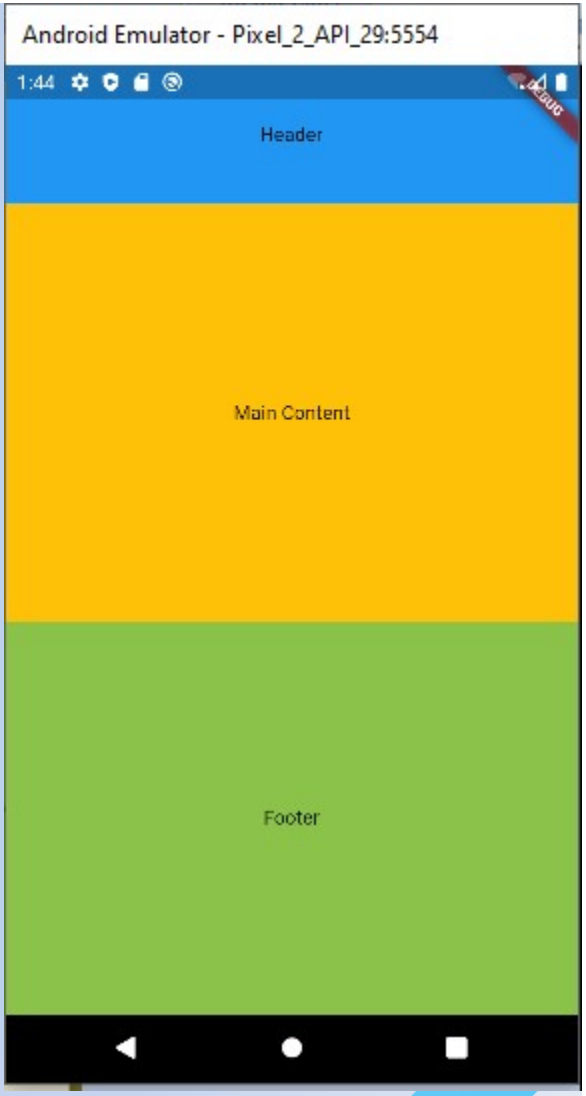
LayoutBuilder Widget In Flutter

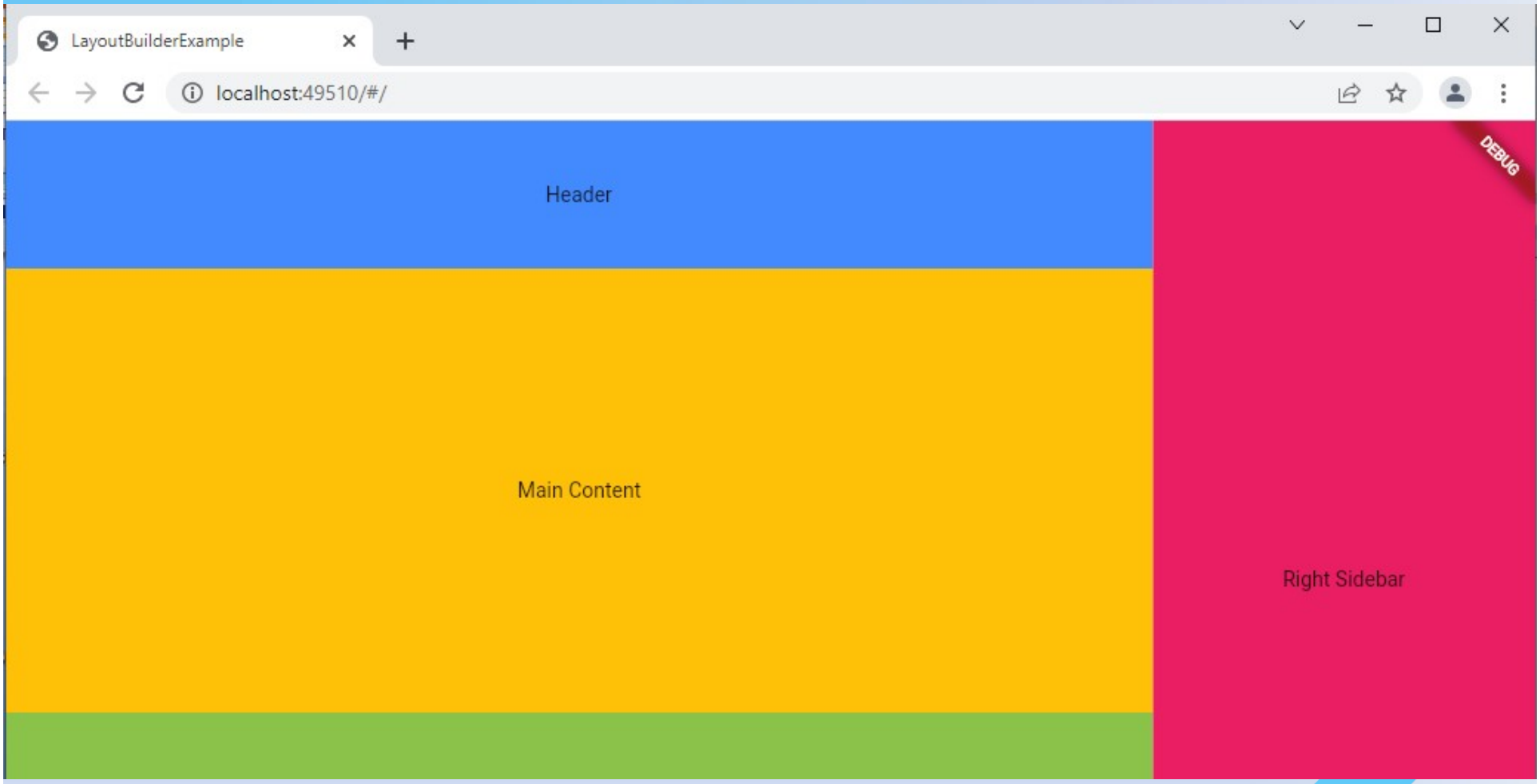
- The LayoutBuilder widget in Flutter helps you build a widget tree that can depend on the parent widget's size (a minimum and maximum width, and a minimum and maximum height).
- LayoutBuilder create responsive layout in the applications which is used on different screen size.
- For an example you can display two items per line on wider device and one item per line on narrow device using layout builder.
- Layoutbuilder has two parameters. build context and Boxconstrant. BuildContext refers to a widget.

```dart
import 'package:flutter/material.dart';
class LayoutBuilderExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "LayoutBuilderExample",
      home: Scaffold(
        appBar: AppBar(
          title: Text('LayoutBuilder Demo'),
        ),
        body: LayoutBuilder(
          builder: (context, constraints) {
            return Column(children: [
              Container(
                  color: Colors.amber,
                  width: constraints.maxWidth,
                  height: constraints.maxHeight * 0.33),
              Container(
                  color: Colors.white,
                  width: constraints.maxWidth,
                  height: constraints.maxHeight * 0.34),
              Container(
                  color: Colors.green,
                  width: constraints.maxWidth,
                  height: constraints.maxHeight * 0.33),
            ]);
          },
        ),
      ),
    );
  }
}
```



```dart
class LayoutBuilderExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "LayoutBuilderExample",
      home: Scaffold(body: LayoutBuilder(builder: (context, constraints) {
      // Large screens (tablet on landscape mode, desktop, TV)
      if (constraints.maxWidth > 600) {
        return Row(
          children: [
            Column(
              children: [
                Container(
                  height: 100,
                  width: constraints.maxWidth * 0.75,
                  color: Colors.blueAccent,
                  child: Center(
                    child: Text('Header'),
                  ),
                ),
                Container(
                  height: 300,
                  width: constraints.maxWidth * 0.75,
                  color: Colors.amber,
                  child: Center(
                    child: Text('Main Content'),
                  ),
                ),
                Container(
                  height: constraints.maxHeight - 400,
                  width: constraints.maxWidth * 0.75,
                  color: Colors.lightGreen,
                  child: Center(
                    child: Text('Footer'),
                  ),
                ),
              ],
            ),
            Container(
              width: constraints.maxWidth * 0.25, height: constraints.maxHeight, color: Colors.pink,

              child: Center(child: Text('Right Sidebar'),),
            ),
          ],
        );
      }
      // Samll screens
      return Column(
        children: [
          Container(
            height: 100, color: Colors.blue, child: Center(child: Text('Header'),),
          ),
          Container(
            height: 300, color: Colors.amber, child: Center(child: Text('Main Content'),),
          ),
          Container(
            height: constraints.maxHeight - 400, color: Colors.lightGreen,
            child: Center(child: Text('Footer'),),
          )
        ],
      );
    })),
    );
  }
}
```



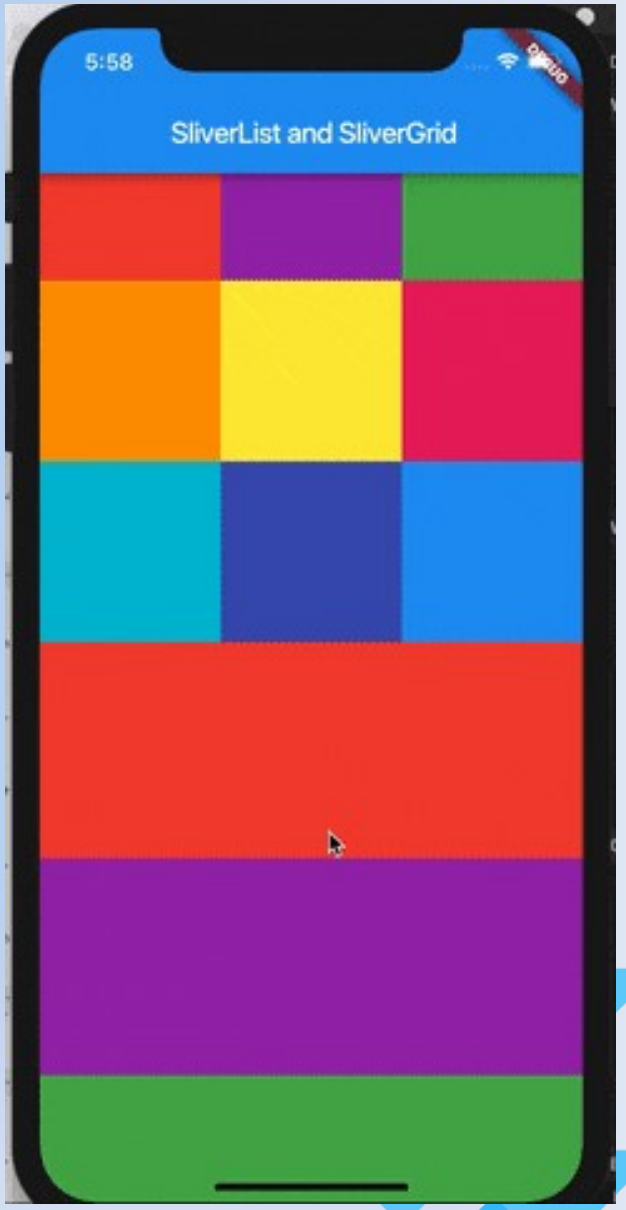

LayoutBuilderExample
localhost:49510/#/
Header
Main Content
Right Sidebar
DEBUG

What is difference between MediaQuery and LayoutBuilder?

- The main difference between MediaQuery and LayoutBuilder is that MediaQuery uses the complete context of the screen rather than just the size of your particular widget, whereas LayoutBuilder can determine the maximum width and height of a particular widget.

Sliver

- A sliver is a portion of a scrollable area that you can define to behave in a special way.
- You can use slivers to achieve custom scrolling effects, such as elastic scrolling.
- Sliver are useful in following case
  - You need an app bar with nonstandard behavior (disappearing as you scroll, changing size or color as you scroll, etc).
  - Need to scroll a ScollorView & GridView together as one unit.
  - Do something different like a collapsing list with headers.



sliver

- All of these sliver components must be inside a CustomScrollView.
- The rest is up to you for how to combine your list of slivers to make your custom scrollable area.
- CustomScrollView is the Widget that allows different types of scrollable widgets and lists. And these scrollable lists and widgets are called slivers.
- The CustomScrollView has many properties and one of them is slivers. Which takes list slivers as an input.
- There are several types of slivers available. For example - SliverAppBar, SliverGridList, and SliverList, etc.
- CustomScrollView only takes Slivers Widgets as a child. Normal widgets like Container, ListTile, etc will not work.

non functional example

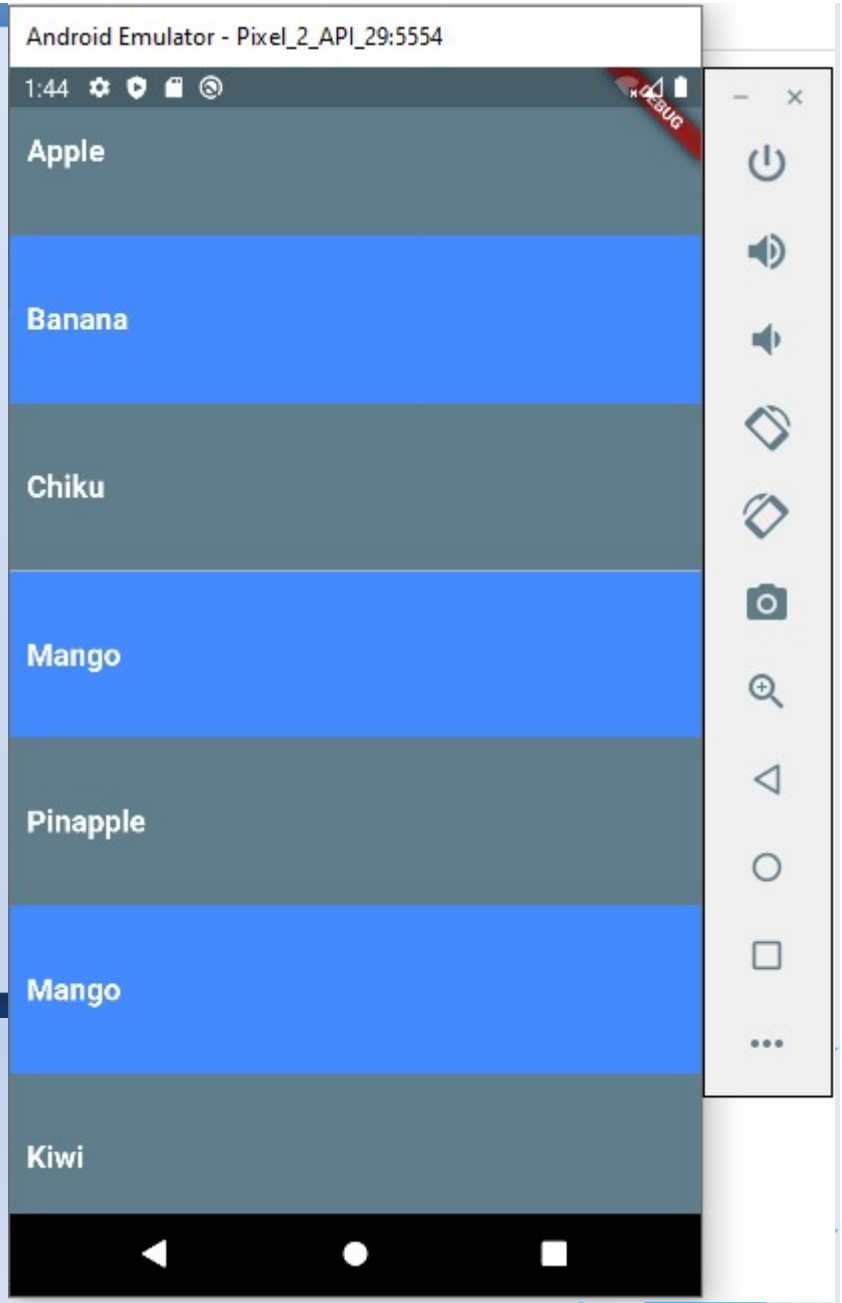
```
Scaffold(
body: CustomScrollView(
  slivers: [
      SliverAppBar(/**/),
      SliverGridList(/**/),
      SliverList(/**/),
      //......
    ]
  ),
);
```

SliverList

- The SliverList is a widget that takes lists of items as child, same as ListView and GridView.
- SliverList, has one parameter called delegate. There are two types of delegate -
- **SliverChildListDelegate**:
  - Takes a list of widgets that we want to display. Widgets defined in this list will be rendered at once. **There will not be any lazy loading**.
- **SliverChildBuilderDelegate**:
  - Take a list of widgets that will be created lazily. It means as the user scrolls items get rendered.

SliverList Example

```
Scaffold(
          body: CustomScrollView(
            slivers: [
              SliverList(
                delegate: SliverChildBuilderDelegate(
                      (context, index) {
                    return Container(
                      height: 100,
                      alignment: Alignment.centerLeft,
                      padding: EdgeInsets.all(10),
                      color: index.isEven ? Colors.blueGrey : Colors.blueAccent,
                      child: Text(fruits[index],
                        style: TextStyle(
                            color: Colors.white, fontSize: 18,
                            fontWeight: FontWeight.bold
                        ),
                      ),
                    );
                  },
                  childCount: 12,
                ),
              ),
            ],
          )),
```



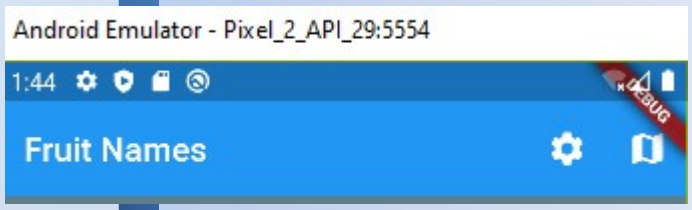
SliverAppBar

- This is the same as the AppBar widget, the difference is that it works with CutomScrollView.
- It means it has all the properties like - title, actions, leading, flexibleSpace etc.
- But it has some additional parameters like pinned, floating, snap, expandedHeight which customizes the behavior of the AppBar.
- Let us see its some properties

Option	Type	Description
title	Widget	The primary widget displayed in the app bar often is a Text widget
leading	Widget	The widget before the title
actions	List<Widget>	A group of widgets after the title
bottom	PreferredSizeWidget	Used to add the bottom section
floating	bool	Determines whether the app bar should become visible as soon as the user scrolls towards the app bar
pinned	bool	Determines whether the app bar should remain visible at the start of the scroll view
snap	bool	If both **snap** and **floating** are true, the floating app bar will snap to the view.

SliverAppBar Example

```
 SliverAppBar(
title: Text('Fruit Names'),
actions: <Widget>[
IconButton(
icon: const Icon(Icons.settings),
onPressed: () {},
),
IconButton(
icon: const Icon(Icons.map),
onPressed: () {},
),
],
),
```



SliverGrid Example

```
SliverGrid(
gridDelegate:
SliverGridDelegateWithFixedCrossAxisCount(
crossAxisCount: 2, mainAxisSpacing: 10,
crossAxisSpacing: 10, childAspectRatio: 2.0,
),
delegate: SliverChildBuilderDelegate(
(context, index) {
return Card(
// generate blues with random shades
color: Colors.blue[Random().nextInt(9) * 100],
child: Container(alignment: Alignment.center, child: Text(fruits[index]),),
);
}, childCount: 12,
),
          ),
```