# Flutter

Flutter

Usando Widgets Comuns

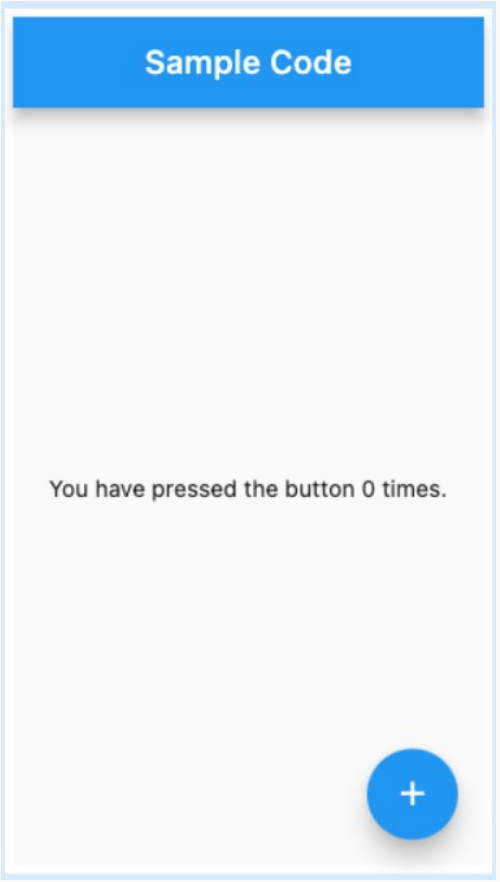
# Basic Widgets

Flutter

#1

# Scaffold

O widget **Scaffold** implementa a estrutura para um layout visual básico do **Material Design**, permitindo que se adicione facilmente vários widgets, como **AppBar**, **Body**, **BottomAppBar**, **FloatingActionButton**, **Drawer**, **Snack Bar**, **BottomSheet** e muito mais.

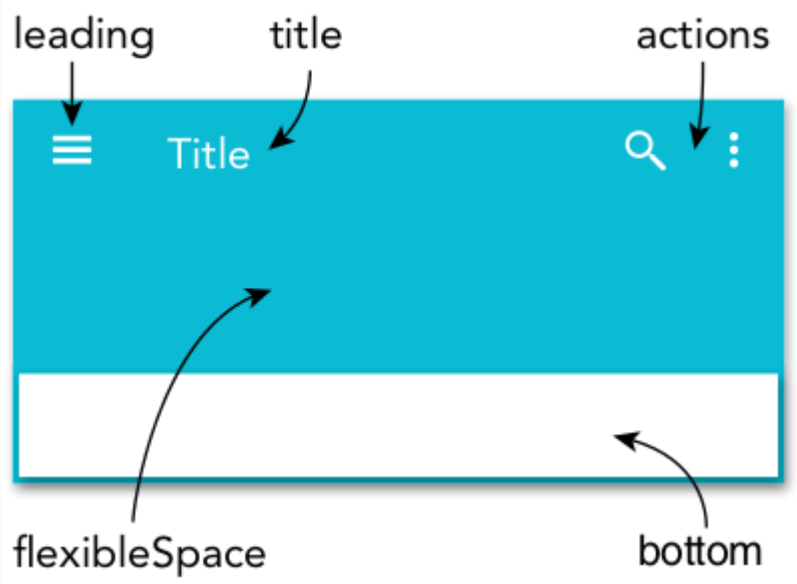


Flutter

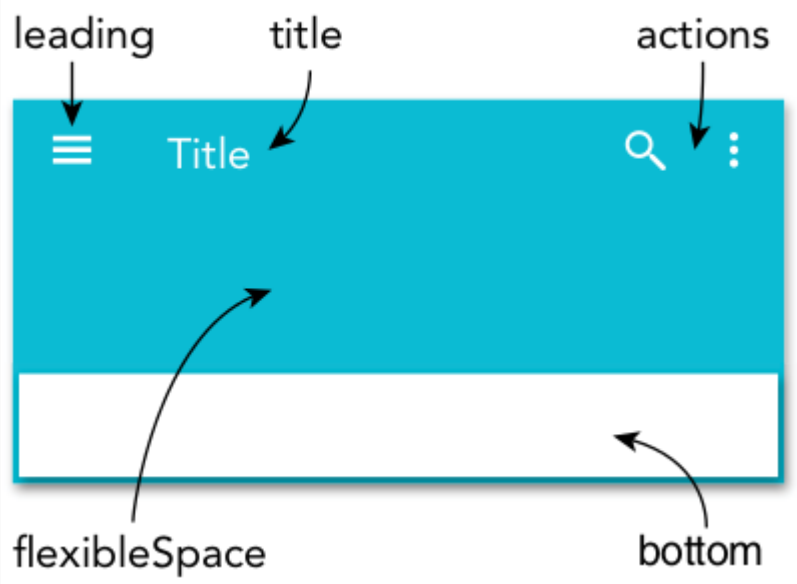
#2

# AppBar

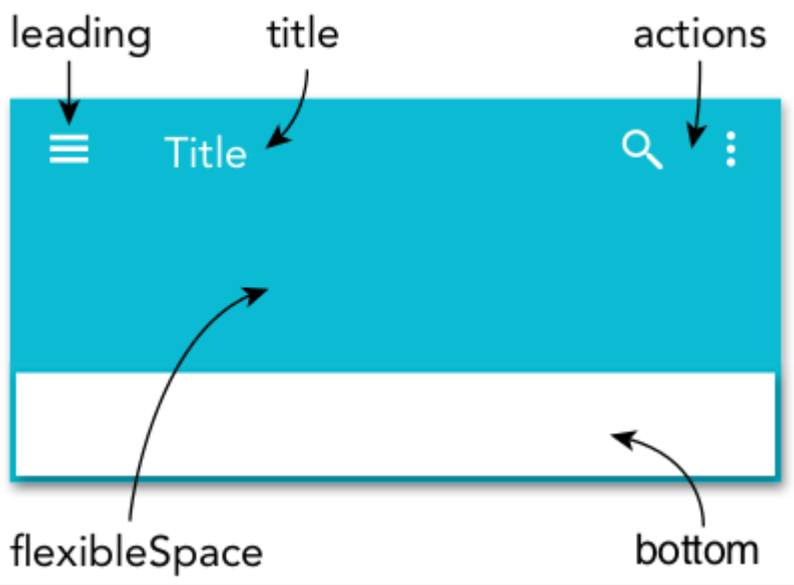
# Adicionando Widget AppBar no Scaffold



```
class _HomeState extends State<Home> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Home'),
      ),
      body: Container(),
    );
  }
}
```



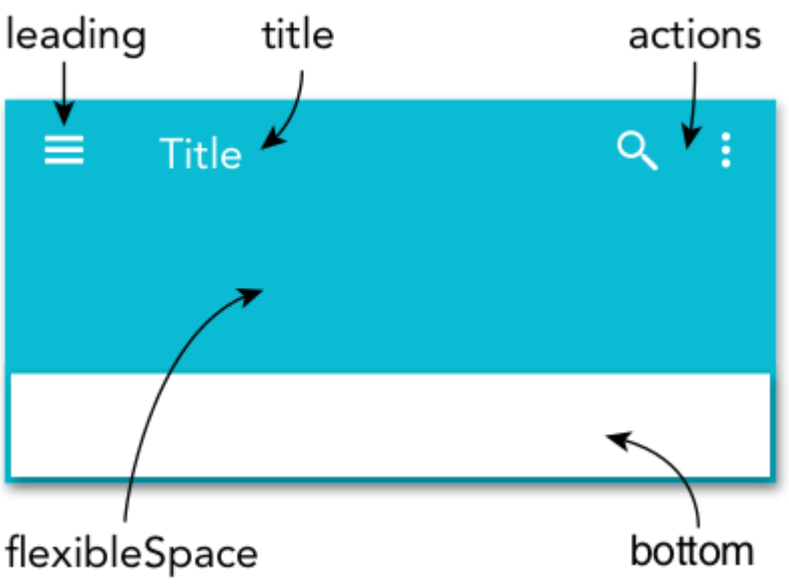
Adicione ao AppBar um IconButton principal. Se você substituir a propriedade **leading** (principal), geralmente é um IconButton ou BackButton.




IconButton or BackButton.

```
leading: IconButton(
  icon: Icon(Icons.menu),
  onPressed: () { },
),
```

A propriedade actions recebe uma lista de widgets; adicionaremos dois widgets **IconButton**.




```
actions: <Widget>[
  IconButton(
    icon: Icon(Icons.search),
    onPressed: () {},
  ),
  IconButton(
    icon: Icon(Icons.more_vert),
    onPressed: () {},
  ),
],
```

Flutter

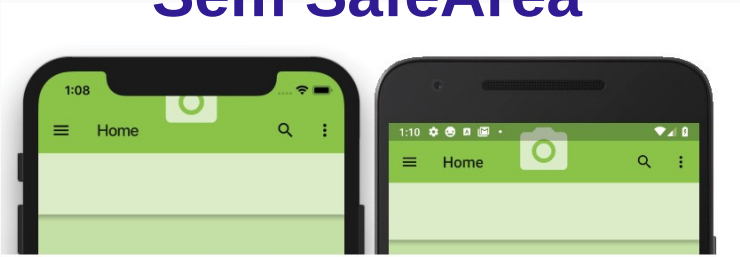
#3

# SafeArea

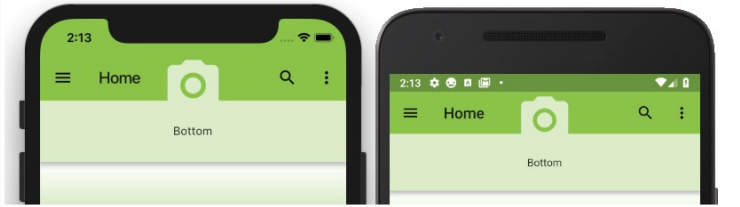
iPhone X ou dispositivos Android tem um recorte parcial que obscurece a tela geralmente localizado na parte superior do dispositivo.

O widget **SafeArea** adiciona automaticamente preenchimento suficiente ao widget filho para evitar invasões do sistema operacional.

## Sem SafeArea




## Com SafeArea




```
body: Padding(
  padding: EdgeInsets.all(16.0),
  child: SafeArea(
    child: SingleChildScrollView(
      child: Column(
        children: <Widget>[

        ],
      ),
    ),
  ),
),
```

Flutter

#4

# Container

O widget **Container** ajudará-lo a compor, decorar e posicionar.

Se enveloparmos um widget em um **Container,** podermos personalizá-lo com cores, tamanho, margens, formas, dentre outros...

# Exemplo

```
Container(
  child: Text('IFMT'),
  decoration: BoxDecoration(
    shape: BoxShape.circle,
    color: Colors.orange,
  ),
  margin: EdgeInsets.all(25.0),
  padding: EdgeInsets.all(40.0),
  alignment: Alignment.center,
  width: 200,
  height: 100,
);
```

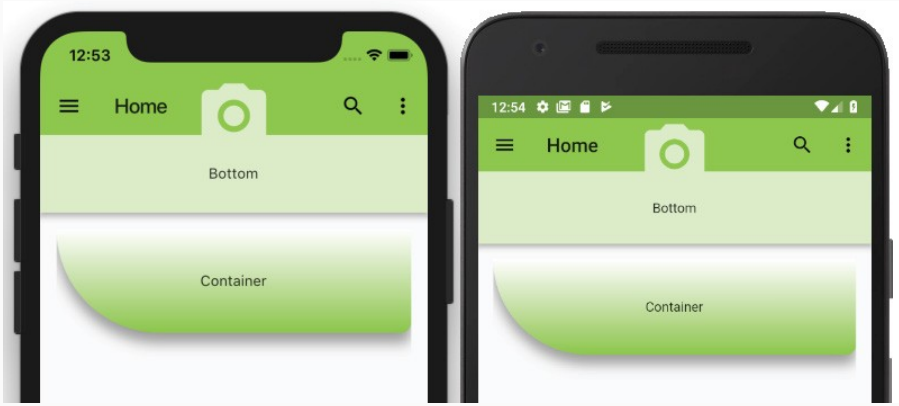
O widget **Container** tem uma propriedade de widget filho opcional e pode ser usado como um widget decorado com uma borda personalizada, cor, restrição, alinhamento, transformação (como girar o widget) e muito mais.

Esse widget pode ser utilizado como um espaço reservado vazio (invisível) e, se um filho for omitido, ele será dimensionado para o tamanho total da tela disponível.

```
@override
Widget build(BuildContext context) {
  return Column(
    children: <Widget>[
      Container(
        height: 100.0,
        decoration: BoxDecoration(
          borderRadius: BorderRadius.only(
            bottomLeft: Radius.circular(100.0),
            bottomRight: Radius.circular(10.0),
          ),
          gradient: LinearGradient(
            begin: Alignment.topCenter,
            end: Alignment.bottomCenter,
            colors: [
              Colors.white,
              Colors.lightGreen.shade500,
            ],
          ),
          boxShadow: [
            BoxShadow(
              color: Colors.grey,
              blurRadius: 10.0,
              offset: Offset(0.0, 10.0),
            ),
          ],
        ),
        child: Center(
          child: RichText(
            text: Text('Container'),          ),
      ),
    ],
  );
}
```

Como um Container funciona:

Os contêineres podem ser widgets poderosos cheios de personalização. Usando decoradores, gradientes e sombras, você pode criar belas interfaces de usuário. Eu gosto de pensar em contêineres como um aprimoramento de um aplicativo da mesma forma que um quadro de ótima aparência adiciona a uma pintura.



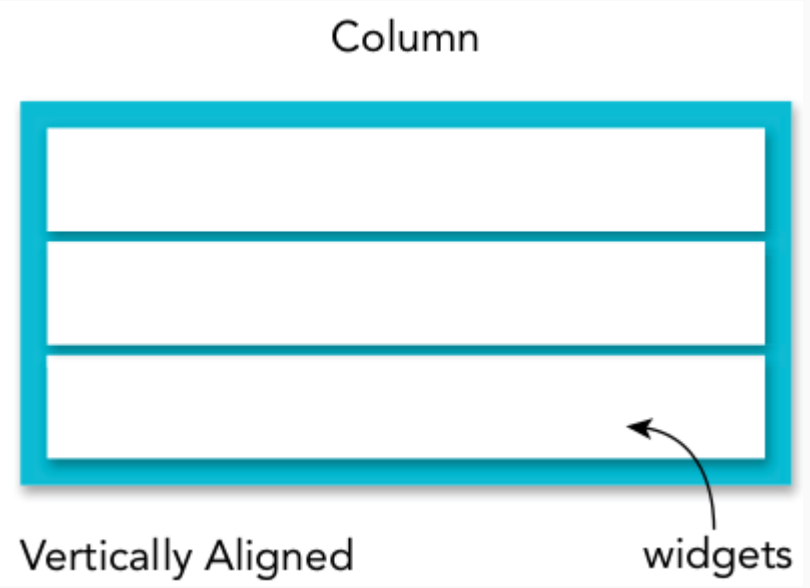
Flutter

#5

# Column

Um widget **Column** exibe seus filhos verticalmente. Contém uma propriedade **children** contendo um array de List<Widget>. **Children** se alinham verticalmente sem ocupar toda a altura da tela.

Cada widget filho pode ser incorporado em um widget **Expanded** para preencher o espaço disponível. Você pode usar **CrossAxisAlignment**, **MainAxisAlignment** e **MainAxisSize** para alinhar e dimensionar quanto espaço é ocupado no eixo principal.



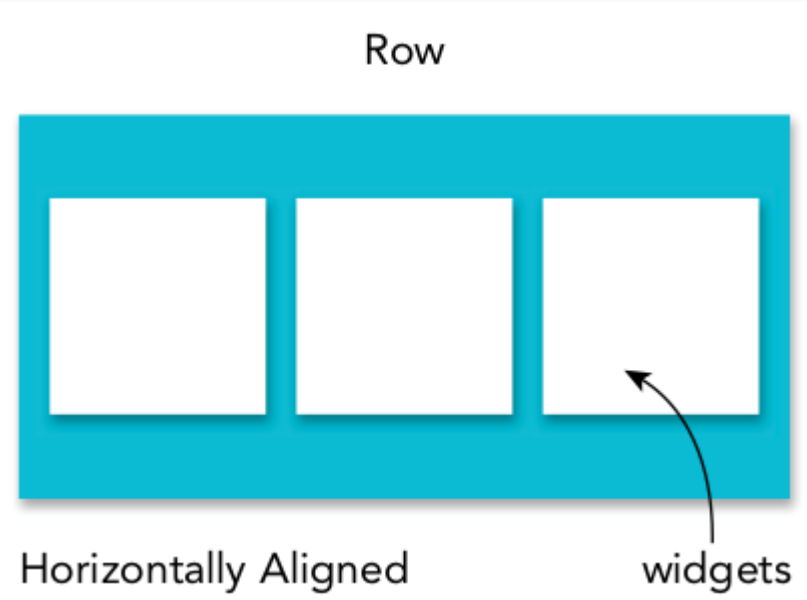

```
Column(
  crossAxisAlignment: CrossAxisAlignment.center,
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  mainAxisSize: MainAxisSize.max,
  children: <Widget>[
    Text('Column 1'),
    Divider(),
    Text('Column 2'),
    Divider(),
    Text('Column 3'),
  ],
),
```

Flutter

#6

# Row

Um widget **Row** exibe seus filhos horizontalmente. Contém, também, uma propriedade children contendo um array de **List<Widget>**. As mesmas propriedades que a Coluna contém são aplicadas ao widget Linha, exceto que o alinhamento é horizontal, não vertical.




```
Row(
  crossAxisAlignment: CrossAxisAlignment.start,
  mainAxisAlignment: MainAxisAlignment.
spaceEvenly,
  mainAxisSize: MainAxisSize.max,
  children: <Widget>[
    Row(
      children: <Widget>[
        Text('Row 1'),
        Padding(padding: EdgeInsets.all(16.0),),
        Text('Row 2'),
        Padding(padding: EdgeInsets.all(16.0),),
        Text('Row 3'),
      ],
    ),
  ],
),
```

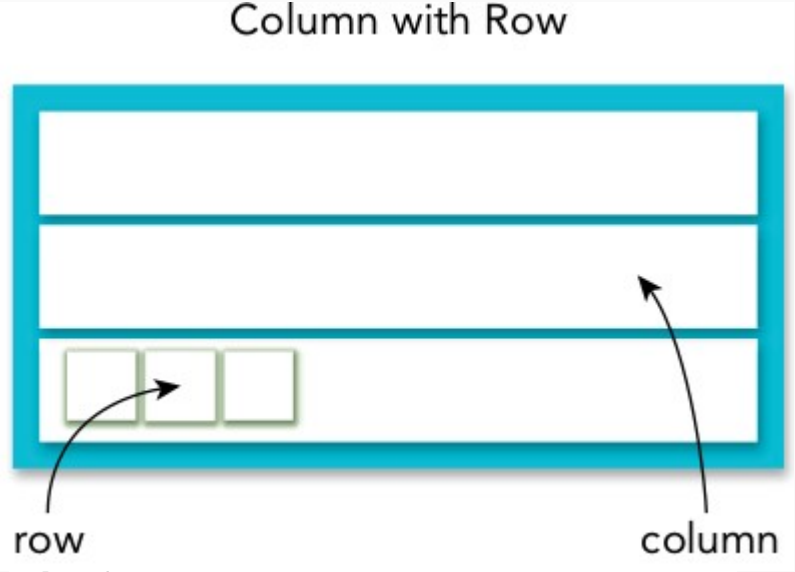
Flutter

# #6.1

## Aninhamento de linhas e colunas

Uma ótima maneira de criar layouts exclusivos é combinar widgets de Column e Row para necessidades particulares.

Imagine ter uma página de diário com texto em uma coluna com uma linha aninhada contendo uma lista de imagens




```
Column(
  crossAxisAlignment: CrossAxisAlignment.start,
  mainAxisAlignment: MainAxisAlignment.
spaceEvenly,
  mainAxisSize: MainAxisSize.max,
  children: <Widget>[
    Text('Columns and Row Nesting 1',),
    Text('Columns and Row Nesting 2',),
    Text('Columns and Row Nesting 3',),
    Padding(padding: EdgeInsets.all(16.0),),
    Row(
      mainAxisAlignment: MainAxisAlignment.spaceEve
      children: <Widget>[
        Text('Row Nesting 1'),
        Text('Row Nesting 2'),
        Text('Row Nesting 3'),
      ],
    ),
  ],
),
```

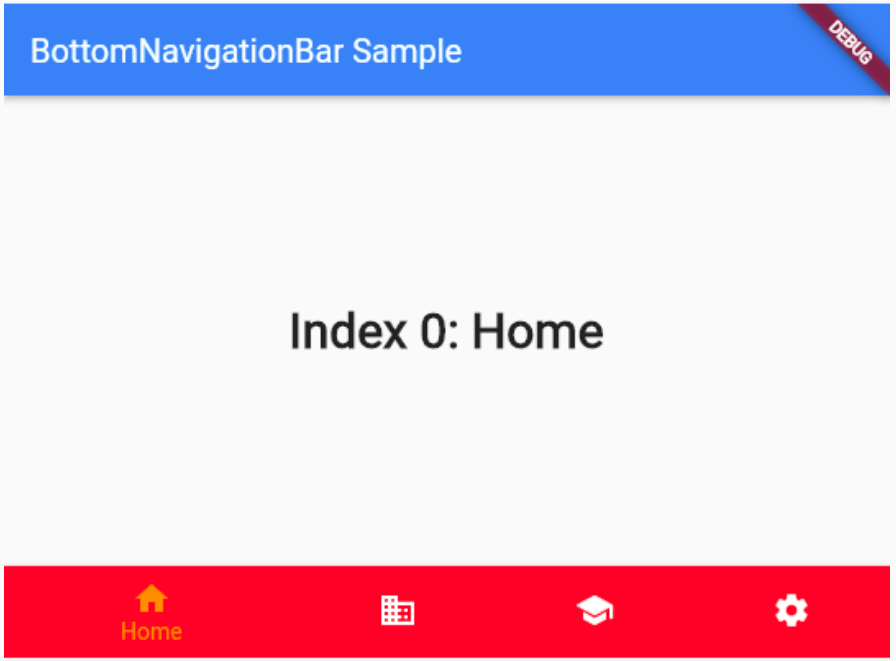
Flutter

#7

# BottomNavigationBar

**BottomNavigationBar** é um widget do Material Design que exibe uma lista de **BottomNavigation-BarItems** que contém um ícone e um título na parte inferior da página.

Quando o **BottomNavigationBarItem** é selecionado, a página apropriada é criada.



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

class Home extends StatefulWidget {
  @override
  _HomeState createState() => _HomeState();
}

class _HomeState extends State<Home> {
  int _currentIndex = 0;
  final List _listPages = [];
  late Widget _currentPage;
  @override
  void initState() {
    super.initState();
    _listPages
      ..add(Categories())
      ..add(Shopping())
      ..add(Util());
    _currentPage = Shopping();
  }

  void _changePage(int selectedIndex) {
    setState(() {
      _currentIndex = selectedIndex;
      _currentPage = _listPages[selectedIndex];
    });
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('BottomNavigationBar'),
      ),
      body: SafeArea(
        child: Padding(
          padding: const EdgeInsets.all(16.0),
          child: _currentPage,
        ),
      ),
      bottomNavigationBar: BottomNavigationBar(
        currentIndex: _currentIndex,
        items: const [
          BottomNavigationBarItem(
            icon: Icon(Icons.cake),
            label: 'Categories',
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.sentiment_satisfied),
            label: 'Shopping',
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.access_alarm),
            label: 'Utils',
          ),
        ],
        onTap: (selectedIndex) => _changePage(selectedIndex),
      ),
    );
  }
}
```

```dart
class Categories extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return const Scaffold(
      body: Center(
        child: Icon(
          Icons.category,
          size: 120.0,
          color: Colors.orange,
        ),
      ),
    );
  }
}
```

```dart
class Shopping extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return const Scaffold(
      body: Center(
        child: Icon(
          Icons.shop_2_rounded,
          size: 120.0,
          color: Colors.lightGreen,
        ),
      ),
    );
  }
}

class Util extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return const Scaffold(
      body: Center(
        child: Icon(
          Icons.umbrella,
          size: 120.0,
          color: Colors.purple,
        ),
      ),
    );
  }
}
```

Flutter

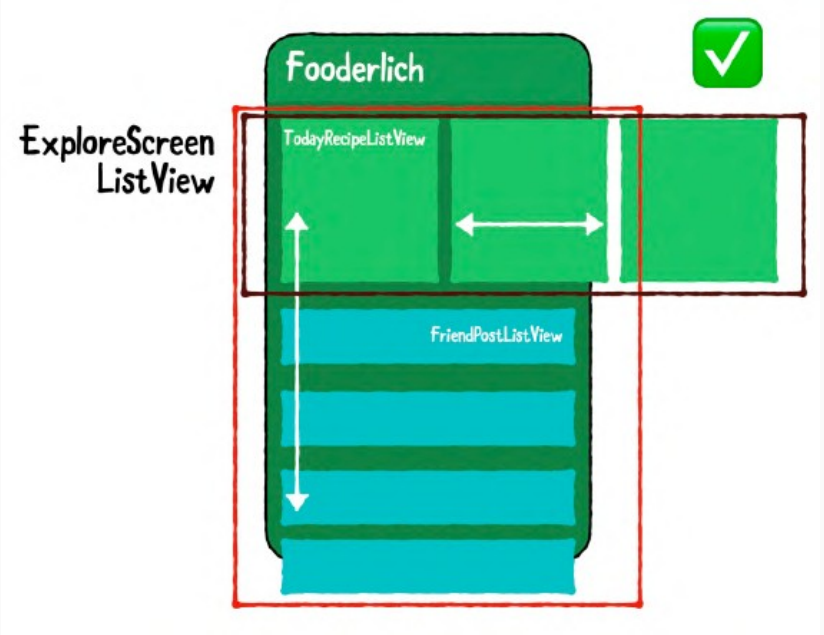
#8

# ListView

O widget **ListView** exibe as informações em forma de listagem - de cima para baixo ou de um lado para o outro.

Além disso, o widget ListView tem seu próprio recurso de rolagem.

Você pode colocar 100 itens em um **ListView** mesmo que apenas 20 itens caibam na tela. Quando o usuário rola a tela, os itens saem da tela enquanto outros itens se movem.




```
33        Expanded(
34          child: ListView.builder(
35            physics: const BouncingScrollPhysics(),
36            itemCount: favors.length,
37            itemBuilder: (BuildContext context, int index) {
38              final favor = favors[index];
39              return Card(
40                key: ValueKey(favor.uuid),
41                margin: const EdgeInsets.symmetric(vertical: 10.0, horizontal: 25.0),
42                child: Padding(
43                  child: Column(
44                    children: <Widget>[
45                      _itemHeader(favor),
46                      Text(favor.description),
47                      _itemFooter(favor)
48                    ],
49                  ),
50                  padding: EdgeInsets.all(8.0),
51                ),
52              );
53            },
54          ),
```

Flutter

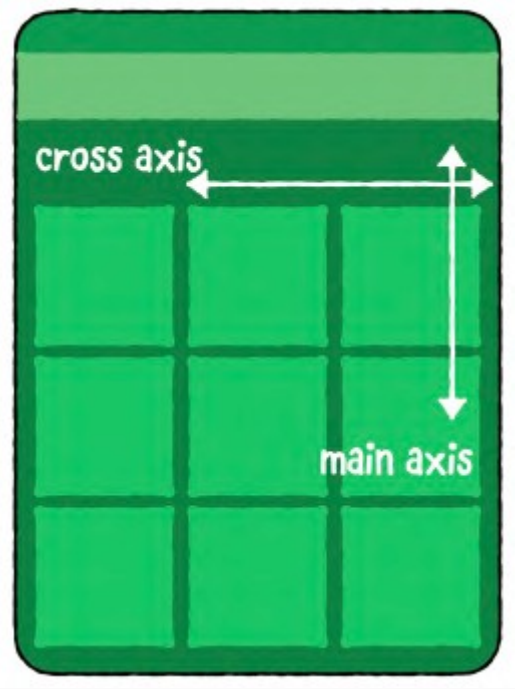
#9

# GridView

O widget **GridView** exibe as informações em forma de grade.

**GridView** é uma matriz **2D** de widgets roláveis. Ele arranja os filhos em uma grade e suporta rolagem horizontal e vertical.

É similar ao **Listview**.



Flutter

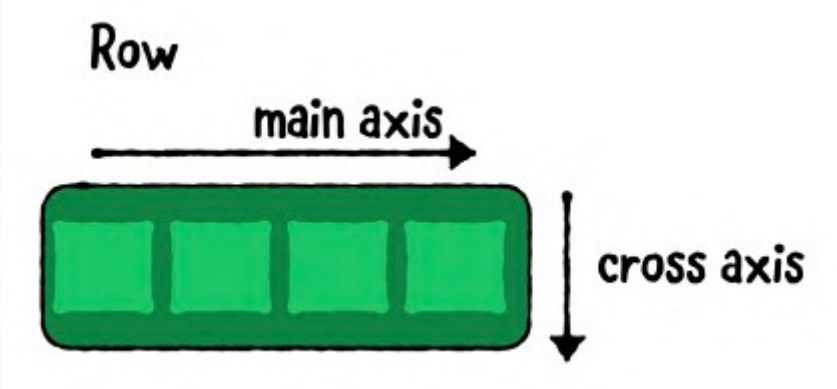
# #9.1

## Compreendendo o eixo transversal e o principal

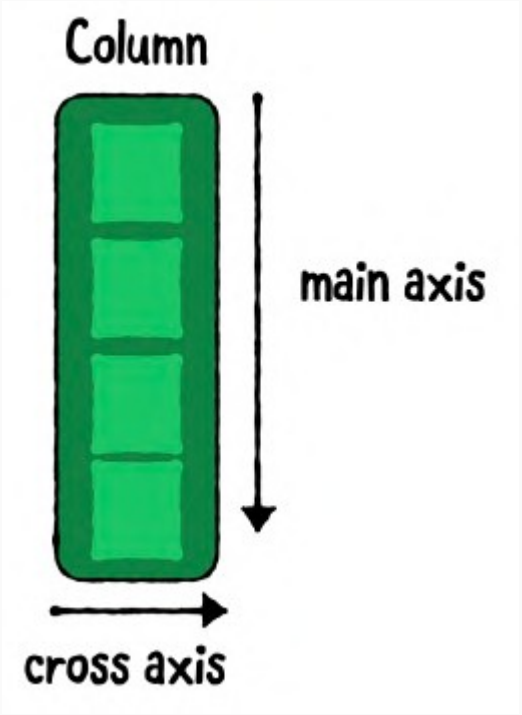
Qual é a diferença entre o eixo principal e o eixo transversal (cruzado)? Lembre-se de que Colunas e Linhas são como **ListViews**, mas sem uma visualização de rolagem.

O eixo principal sempre corresponde à direção de rolagem!

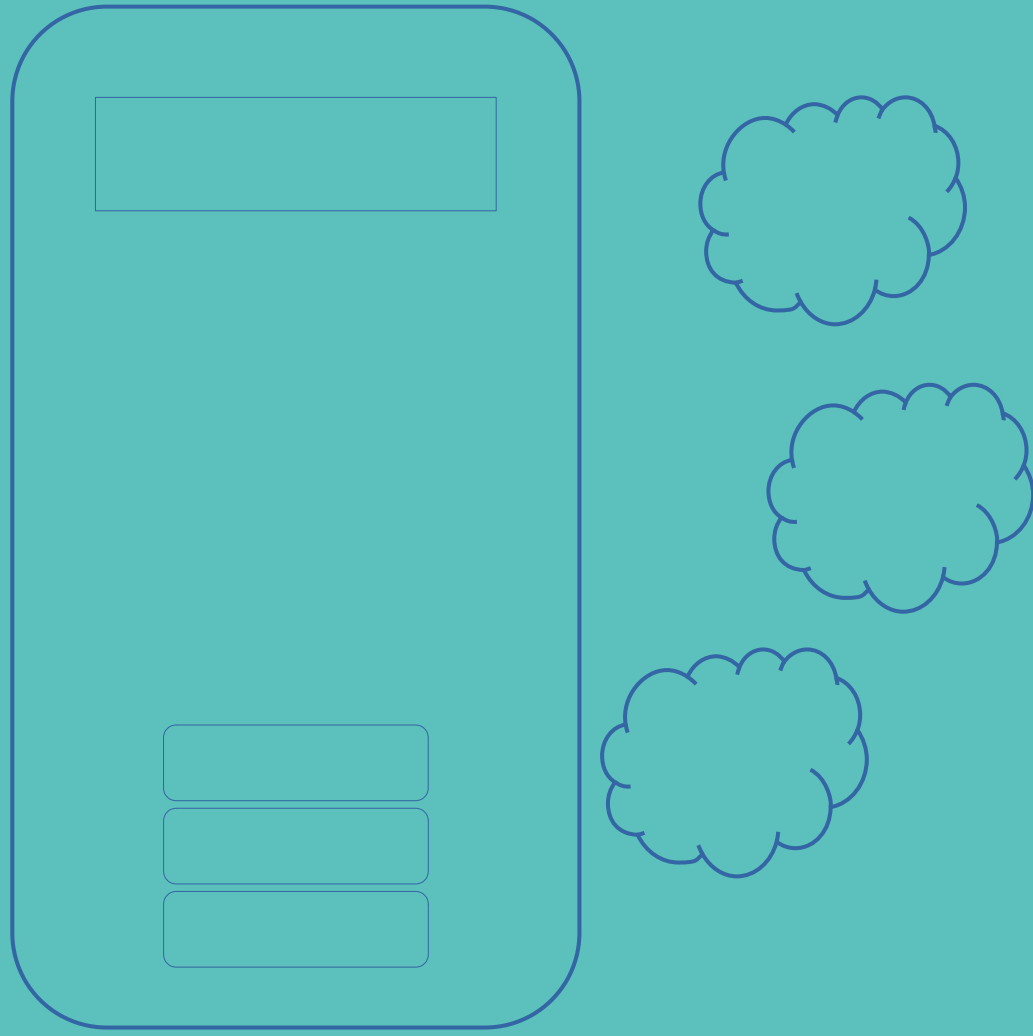
Se sua direção de rolagem for horizontal, você pode pensar nisso como uma linha. O eixo principal representa a direção horizontal, conforme mostrado na imagem acima:



Se sua direção de rolagem for vertical, você pode pensar nela como uma Coluna. O eixo principal representa a direção vertical, conforme mostrado ao lado:



# Async faz IO simples



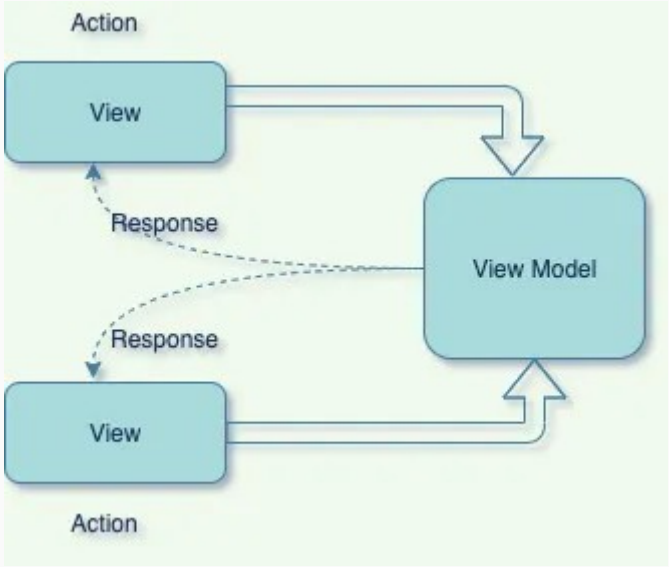
Flutter

# #10

# MVP

O MVVM é útil para mover a lógica de negócios de exibição para ViewModel e Model.

- ViewModel é o mediador entre View e Model que carrega todos os eventos do usuário e retorna o resultado.
-



Flutter

# #10.1

# Compreendendo a estrutura do MVVM

Flutter

#11

# Responsividade

# Responsividade

**Adicionando Responsividade aos Widgets**

- A interface do usuário responsiva altera a interface do usuário da tela/widget do aplicativo de acordo com o tamanho das diferentes telas de celulares.
- A interface do usuário responsiva é muito útil quando o mesmo aplicativo é feito para celular, web, desktop, relógios (wear apps).
- A interface do usuário responsiva reorganiza a interface do usuário de acordo com a orientação e o tamanho do dispositivo.

Flutter

# #11.1

# Utilizando Responsividade

# Responsividade: Fatores que contribuiem para UI’s responsivas

**LayoutBuilder**

- O LayoutBuilder altera a interface do usuário de acordo com as restrições de tamanho da tela do dispositivo.
- Este widget usa um construtor que retorna um widget e altera a exibição de acordo com a condição especificada pelo usuário.

```
LayoutBuilder(
        builder: (context, constraints) {
          if (constraints.maxWidth >= 601) {
            return HorizontalView();
          } else {
            return VerticalView();
          }},),
```

# Responsividade: Fatores que contribuiem para UI’s responsivas

**MediaQuery**

- é a classe/widget que nos permite consultar o tamanho atual da mídia.
  - Podemos usar MediaQueryData.size para obter o tamanho da tela.
  - Isso será reconstruído automaticamente sempre que o MediaQueryData for alterado.
  -

```
MediaQueryData mediaQueryData(BuildContext context) {
  return MediaQuery.of(context);
}

Size size(BuildContext buildContext) {
  return mediaQueryData(buildContext).size;
}

double width(BuildContext buildContext) {
  return size(buildContext).width;
}

double height(BuildContext buildContext) {
  return size(buildContext).height;
}
```

# Responsividade: Fatores que contribuiem para UI’s responsivas

**MediaQuery**

- **MediaQuery.of(context)** retorna o **MediaQueryData**(ex. tamanho da tela, modo de brilho, etc).
- **MediaQuery.of(context).size.width** nos dá a largura.
- **MediaQuery.of(context).size.height** nos dá a altura da mídia atual.