SI700 – Programação para Dispositivos Móveis

Aula 3 – Conceitos Básicos de Flutter

Prof. Ulisses Martins Dias

2022

Faculdade de Tecnologia – Unicamp

Roteiro

Widgets

Tipos de Widgets

Top-Level Widgets

Scaffold

PageController, PageView

SizedBox, FittedBox e ConstrainedBox

Center, Row e Column

Container

Text

Widgets

Widgets

- **Widgets** são classes convencionais de Dart que definirão a interface gráfica de um aplicativo. Dessa forma, em Flutter, tanto o comportamento dos aplicativos quanto a interface gráfica são escritos em Dart.
- Widgets são organizados hierarquicamente. Uma hierarquia de widgets é também considerada um widget.
- Em geral, elementos que você a priori não imaginaria como widgets, às vezes são widgets, como paddings e decorators, por exemplo, aplicados a uma imagem ou texto.

Widgets

- No seu código, a interface gráfica será uma grande árvore de widgets, também chamada de widget tree. Será importante conhecer os elementos no topo da árvore e os parâmetros dos construtores deles.
- Alguns widgets na árvore terão apenas um filho, enquanto que outros terão vários filhos.
- Os filhos não são iguais, podendo alguns serem adicionados em parâmetros diferentes no construtor do widget pai.

- Para interagir com a árvore de widgets, objetos da classe **BuildContext** são associados a cada widget e permitem acessar informações dos pais (e dos ancestrais, em cadeia) na árvore.
- Um **BuildContext** permite acessar dados que ajudam a definir o widget sendo criado. Por exemplo, os dados armazenados em **ThemeData** podem ser usados para que um widget tenha uma aparência consistente com a interface gráfica.
- Em última instância, você deve imaginar a árvore de widget de uma maneira diferente das árvores aprendidas quando estudando estruturas de dados convencionais. Aqui, um nó da árvore aponta para o nó pai, enquanto que lá os nós da árvore apontavam para os filhos.

Tipos de Widgets

Tipos de Widgets

- A maioria dos widgets são **StatelessWidgets** ou **StatefulWidgets**.
- Um **StatelessWidget** é imutável uma vez que é adicionado na tela. Para renderizar novamente um **StatelessWidget** na tela, é preciso criar uma nova instância dele.
- Um **StatefulWidget** é acompanhado de um objeto da classe **State**, que representa o estado atual do widget. O State pode ser mudado dinamicamente.

Tipos de Widgets

- Um **StatefulWidget** pode ser desenhado múltiplas vezes na interface gráfica durante o ciclo de vida do widget para se adequar a mudanças de estados.
- A renderização de um widget na interface gráfica ocorre invocando o método **build**, que todo os widgets possuem.
- O método **build** retornará uma instância da classe widget, sendo essa instância aquilo que será apresentado na tela quando este widget for adicionado.
- O método **build** de um **StatelessWidget** é invocado apenas uma vez, enquanto que o método **build** de um **StatefulWidget** pode ser invocado várias vezes.

Tipos de Widgets

Exemplo Minimalista de **StatelessWidget**

```
class MyApp extends StatelessWidget {
  Widget build (BuildContext context){
    return OneOrMoreWidgets;
  }
}
```

Tipos de Widgets

- Ao programar um **StatefulWidget**, você irá implementar duas classes, uma delas será a classe que herda de **StatefulWidget** e a outra será a classe que herda de **State**. Você deverá imaginar essas duas classes como um bloco.
- Coloque no **StatefulWidget** os atributos e propriedados constantes e no **State** os atributos e propriedades variáveis.
- No **StatefulWidget**, o método **build** ficará na classe **State**, dado que depende do conhecimento do estado atual para renderizar na tela.
- Flutter irá invocar o método build sempre que julgar necessário, sem perder o estado atual associado ao widget.
- O **BuildContext** associado ao elemento na árvore de widgets irá armazenar uma referência tanto ao widget quanto ao **State**.

Exemplo Minimalista de **StatefulWidget**

```
class MyApp extends StatefulWidget {
  State<MyApp> createState(){
    return MyAppState();
  }
}

class MyAppState extends State<MyApp>{
  Widget build (BuildContext context){
    return OneOrMoreWidgets;
  }
}
```

Tipos de Widgets

- Para notificar o **StatefulWidget** de que alguma mudança no estado deve ser refletida na tela, usamos o método **setState**.
- A simples invocação do **setState** já força o framework a analisar quais propriedades foram modificadas no estado para realizar a mudança.
- Você também pode passar para **setState** uma função que realiza uma série de mudanças no estado para que, ao final dessas mudanças, a tela seja renderizada novamente.
- No exemplo a seguir, criamos uma função anônima que incrementa um atributo **_count**.

```
FloatingActionButton(
  onPressed: () => setState(() {
      _count++;
  })
```

Top-Level Widgets

Top-Level Widgets

- Existem centenas de widgets em Flutter, sendo impossível e desnecessário conhecer todos. Vamos conhecer os mais usados em algumas categorias.
- Chamaremos de Top-level widget o primeiro widget que você colocará na árvore. Este widget é instanciado invocando a função runApp na função main do Dart.
- A escolha do top-level impacta na aparência do aplicativo e nos padrões a serem seguidos.
- De um modo geral, o widget **MaterialApp** é o mais usado para usar o padrão MaterialDesign do google. Ele funcionará em android e iOS.
- Uma segunda opção seria o **CupertinoApp**, normalmente usado em aplicativos focados na aparência do iOS, criando padrões de fonte e estilo de rolagem comuns a esses sistemas.

Top-Level Widgets

- **MaterialApp**: possui alguns atributos (propriedades) comumente configurados.
  - **title**: define o nome do aplicativo.
  - **theme**: usa um objeto **ThemeData** para especificar cores e outras propriedades relacionadas ao *look and feel* do aplicativo. Em geral, widgets filhos irão procurar por essa informação na árvore para desenhar a própria interface de maneira adequada.
  - **home**: tela que o usuário visualizará por primeiro e ocupará inicialmente o centro do aplicativo. Em geral, coloque um **Scaffold** nessa posição.

```
MaterialApp(
  title: 'Flutter Demo',
  theme: ThemeData(
    primarySwatch: Colors.blue,
    visualDensity: VisualDensity.adaptivePlatformDensity,
  ),
  home: MyHomePage(title: 'Page'),
);
```

Scaffold

Scaffold

- Implementa a estrutura básica da tela com o layout do MaterialDesign, e gerencia elementos como: barra de navegação, *drawers*, barra de botões na parte inferior, botão flutuante, ...
- A versão cupertino do Scaffold é a **CupertinoPageScaffold**, que possui propriedades similares.

Scaffold

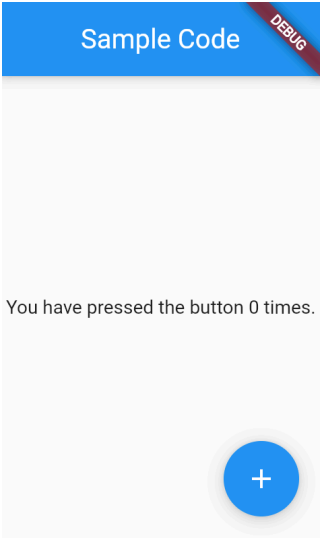
O **Scaffold** possui apenas um construtor que você instanciará configurando uma série de parâmetros. Por exemplo:

- **appBar**: barra horizontal mostrada no topo de um aplicativo.
- **body**: widget que será mostrado na parte central do scaffold.
- **floatingActionButton**: botão flutuante que será adicionado na tela, tipicamente no canto inferior direito.
- **floatingActionButtonLocation**: permite configurar a posição do botão flutuante.
- **bottomNavigationBar**: sequência de botões organizados na horizontal no rodapé de um aplicativo.
- **bottomSheet**: faixa mostrada no rodapé do aplicativo, podendo ser persistente ou modal.
- **backgroundColor**: cor de fundo do aplicativo.

Scaffold

```
Scaffold(
  appBar: AppBar(
    title: const Text('Sample Code'),
  ),
  body: Center(child: Text('You have pressed the button $_count
  ↪  times.')),
  floatingActionButton: FloatingActionButton(
    onPressed: () => setState(() => _count++),
    tooltip: 'Increment Counter',
    child: const Icon(Icons.add),
  ),
);
```

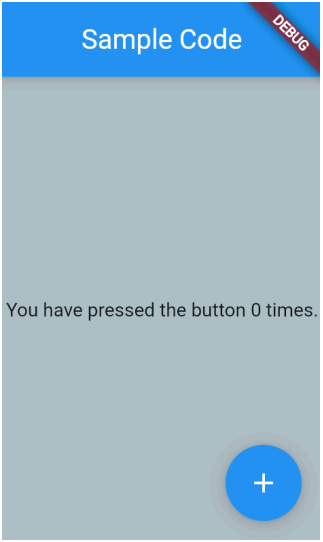
Scaffold



Scaffold

```
Scaffold(
  appBar: AppBar(
    title: const Text('Sample Code'),
  ),
  body: Center(child: Text('You have pressed the button $_count
  ↪  times.')),
  backgroundColor: Colors.blueGrey.shade200,
  floatingActionButton: FloatingActionButton(
    onPressed: () => setState(() => _count++),
    tooltip: 'Increment Counter',
    child: const Icon(Icons.add),
  ),
);
```

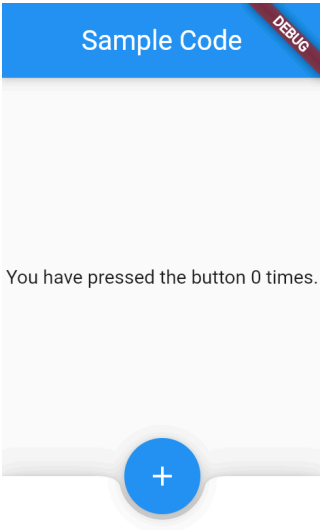
Scaffold



Scaffold

```
Scaffold(
  appBar: ... ,
  body: ... ,
  bottomNavigationBar: BottomAppBar(
    shape: const CircularNotchedRectangle(),
    child: Container(height: 50.0),
  ),
  floatingActionButton: ... ,
  floatingActionButtonLocation:
  ↪ FloatingActionButtonLocation.centerDocked,
);
```

Scaffold



PageController, PageView

PageController, PageView

- Para criar um aplicativo com várias telas e mover de uma tela a outra com **swipe**, podemos usar a classe **PageView**.
- A utilização é bastante simples, basta colocar a **PageView** no parâmetro **body** do **Scaffold** e adicionar as telas no parâmetro **children** da **PageView**.
- Por padrão, o **swipe** gira na horizontal. Entretanto, é possível fazer com que ele gire na vertical atribuindo o valor **Axis.vertical** para o parâmetro **scrollDirection**.
- O parâmetro **onPageChanged** permite indicar qual função deve ser invocada quando o **swipe** ocorrer.

PageController, PageView

- Um objeto da classe **PageController** deve ser alocado caso seja necessário informar qual a primeira tela a ser exibida.
- O objeto **PageController** permite também controlar a movimentação de páginas externamente.

PageController, PageView

```
PageView(
  controller: PageController(
    initialPage: 0, // Configura a tela inicial
  ),
  onPageChanged: (index) {
    localPage = index;
  },
  // O parâmetro "children" recebe todas as páginas
  children: const [
    Tela1(),
    Tela2(),
    Tela3(),
  ],
)
```

SizedBox, FittedBox e ConstrainedBox

SizedBox, FittedBox e ConstrainedBox

- O **SizedBox** força o filho a ter largura e altura específicos. No entanto, ele não implica em mudança de escala de qualquer tipo, apenas reserva o espaço na tela.
- Note no exemplo a seguir que, apesar de na tela termos reservado um espaço grande, a **Text** não aproveita o que foi alocado.

SizedBox, FittedBox e ConstrainedBox

```
Scaffold(
  appBar: AppBar(
    title: const Text('SizedBox'),
  ),
  body: const SizedBox(
    width: 200,
    height: 400,
    child: Text("Child2"),
  ),
)
```

SizedBox, FittedBox e ConstrainedBox

SizedBox, FittedBox e ConstrainedBox

- O **FittedBox** muda a escala do filho e o reposiciona relativamente ao próprio FittedBox. Pode ser usado em conjunto com SizedBox para controlar o tamanho dos objetos.
- Note no exemplo a seguir que **SizedBox** aloca espaço na tela e que este espaço foi usado por **FittedBox** para expandir a **Text**.

SizedBox, FittedBox e ConstrainedBox

```
Scaffold(
  appBar: AppBar(
    title: const Text('SizedBox e FittedBox'),
  ),
  body: const SizedBox(
    width: 200,
    height: 400,
    child: FittedBox(
      child: Text("Child2"),
    ),
  ),
);
```

SizedBox, FittedBox e ConstrainedBox

SizedBox, FittedBox e ConstrainedBox

- Em muitos casos, você achará útil pedir para um determinado widget ocupar todo o espaço disponível ao invés de informar um valor para **SizedBox**. Isso pode ser feito com o construtor nomeado **SizedBox.expand**.

SizedBox, FittedBox e ConstrainedBox

```
Scaffold(
  appBar: AppBar(
    title: const Text('SizedBox e FittedBox'),
  ),
  body: const SizedBox.expand(
    child: FittedBox(
      child: Text("Child2"),
    ),
  ),
)
```

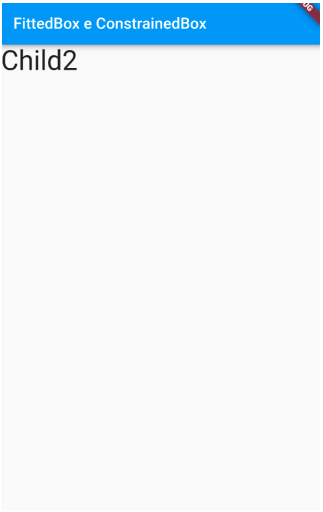
SizedBox, FittedBox e ConstrainedBox

- O **ConstrainedBox**, ao invés de definir um tamanho para o widget, adiciona restrições como tamanho mínimo ou máximo. As opções são **minWidth**, **minHeight**, **maxWidth**, **maxHeight** adicionadas no parâmetro **constraints**.

SizedBox, FittedBox e ConstrainedBox

```
Scaffold(
  appBar: AppBar(
    title: const Text('FittedBox e ConstrainedBox'),
  ),
  body: ConstrainedBox(
    constraints: const BoxConstraints(minWidth: 100),
    child: const FittedBox(
      child: Text("Child2"),
    ),
  ),
)
```

SizedBox, FittedBox e ConstrainedBox



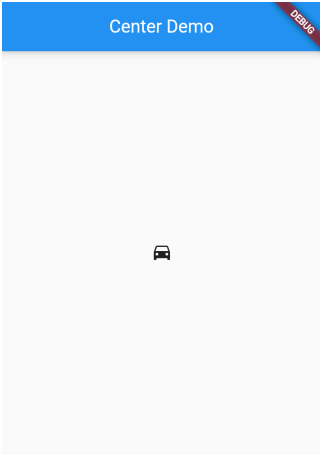
Center, Row e Column

Center, Row e Column

- **Center** possui apenas um filho e centralizará esse filho dentro das dimensões do próprio **Center**.
- Por padrão, o widget **Center** será tão grande quanto possível se houver restrições de tamanho impostas pelo widget pai e se **widthFactor** e **heightFactor** forem nulos.
- No exemplo a seguir, **Scaffold** limita **Center** em todas as dimensões. Nesse caso, **Center** tentará crescer o máximo que conseguir.

```
Scaffold(
  appBar: AppBar(title: const Text('Center Demo')),
  body: const Center(
    child: Icon(Icons.directions_car)
  )
)
```

Scaffold

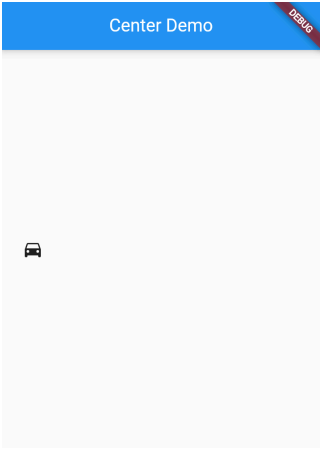


Center, Row e Column

- Os parâmetros **widthFactor** e **heightFactor** configuram o tamanho de **Center** em função do tamanho dos filhos.
- Se **widthFactor** for 2.0, por exemplo, então **Center** não tentará crescer de maneira ilimitada na horizontal, mas terá o dobro do tamanho do seu filho. Note que na vertical **Center** ainda tentará crescer o máximo que conseguir.

```
Scaffold(
  appBar: ...
  body: const Center(
    child: Icon(Icons.directions_car),
    widthFactor:3.0
  )
)
```

Scaffold



- Se **heightFactor** for 2.0, por exemplo, então **Center** não tentará crescer de maneira ilimitada na vertical, mas terá o dobro do tamanho do seu filho. Note que na horizontal **Center** ainda tentará crescer o máximo que conseguir.

```
Scaffold(
  appBar: ...
  body: const Center(
    child: Icon(Icons.directions_car),
    heightFactor:3.0
  )
)
```

Scaffold

Center, Row e Column

- **Row** e **Column** são widgets com propriedades similares. O primeiro posiciona os filhos horizontalmente e o segundo posiciona os filhos verticalmente. Essas seriam as direções (eixos) principais dos widgets.
- Ambos podem ter vários filhos no parâmetro **children**. Esse parâmetro deve receber uma lista de widgets.
- O parâmetro **mainAxisAlignment** permite definir o alinhamento dos filhos na direção (eixo) principal (vertical para **Column** e horizontal para **Row**).
- O parâmetro **crossAxisAlignment** permite definir o alinhamento dos filhos na direção (eixo) secundária (horizontal para **Column** e vertical para **Row**).
- **Column** e **Row** tentam ocupar todo espaço disponível na direção principal, mas ocupam apenas o espaço necessário na direção secundária. Use **SizedBox.expand** se quiser fazer com que todo o espaço disponível seja ocupado nas duas direções.

Center, Row e Column

Em **Column** e **Row**, os principais valores de **mainAxisAlignment** e **crossAxisAlignment** são:

- start: posiciona os filhos no início do eixo.
- end : posiciona os filhos no final do eixo.
- center: posiciona os filhos no centro do eixo.
- spaceBetween: posiciona os filhos alocando espaço igualmente entre eles. O primeiro e o último filho ficarão posicionados no início e no fim do eixo, respectivamente.
- spaceAround: posiciona os filhos alocando espaço igualmente entre eles. Metade do espaço entre os filhos será adicionado também antes do primeiro filho e após o último filho.
- spaceEvenly: posiciona os filhos alocando igualmente espaço entre eles e também antes e depois do primeiro e do último filho, respectivamente.

Center, Row e Column

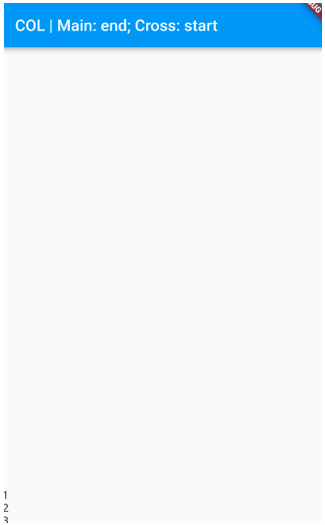
```
Scaffold(
  appBar: AppBar(
    title: const Text('COL | Main: start; Cross: start'),
  ),
  body: SizedBox.expand(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.start,
      crossAxisAlignment: CrossAxisAlignment.start,
      children: const [
        Text("1"),
        Text("2"),
        Text("3"),
      ],    ),  ),);
```

Center, Row e Column

Center, Row e Column

```
Scaffold(
  appBar: AppBar(
    title: const Text('COL | Main: end; Cross: start'),
  ),
  body: SizedBox.expand(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.end,
      crossAxisAlignment: CrossAxisAlignment.start,
      children: const [
        Text("1"),
        Text("2"),
        Text("3"),
      ],    ),  ),);
```

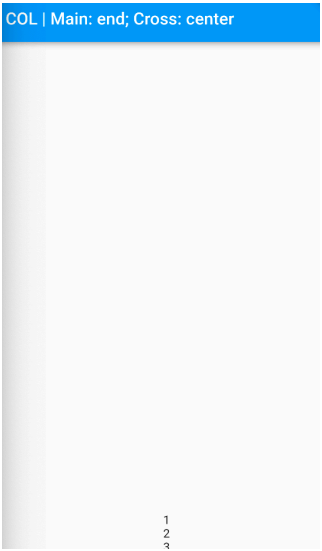
Center, Row e Column



Center, Row e Column

```
Scaffold(
  appBar: AppBar(
    title: const Text('COL | Main: end; Cross: center'),
  ),
  body: SizedBox.expand(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.end,
      crossAxisAlignment: CrossAxisAlignment.center,
      children: const [
        Text("1"),
        Text("2"),
        Text("3"),
      ],    ),  ),);
```

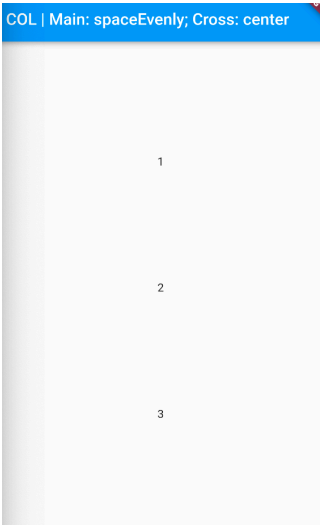
Center, Row e Column



Center, Row e Column

```
Scaffold(
  appBar: AppBar(
    title: const Text('COL | Main: spaceEvenly; Cross: center'),
  ),
  body: SizedBox.expand(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      crossAxisAlignment: CrossAxisAlignment.center,
      children: const [
        Text("1"),
        Text("2"),
        Text("3"),
      ],    ),  ),);
```

Center, Row e Column



Center, Row e Column

```
Scaffold(
  appBar: AppBar(
    title: const Text('COL | Main: spaceEvenly; Cross: center'),
  ),
  body: SizedBox.expand(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      crossAxisAlignment: CrossAxisAlignment.center,
      children: const [
        Text("1"),
        Expanded( // Este filho tomará o espaço excedente.
          child: FittedBox(child: Text("2")),
        ),
        Text("3"),
      ], ), ), );
```


COL | Expanded com FittedBox
1
2
3

Center, Row e Column

- **Row** é muito similar a **Column**, exceto que a adição ocorre na horizontal.
- A combinação de **Row** e **Column** permite a criação de interfaces padronizadas em um formato de tabela.

Container

Container

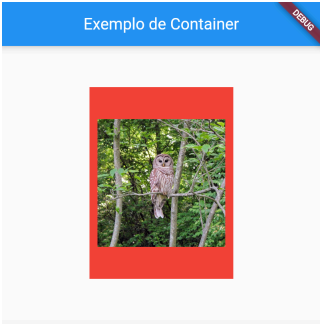
- O **Container** permite compor, decorar e posicionar os elementos filhos. Você deve usá-lo como uma forma conveniente de substituir uma série de outros widgets em um pacote só.
- Caso você insira um **Container** sem nenhum parâmetro além do **child**, então ele não fará diferença alguma na tela. Você precisará configurar os outros parâmetros do construtor:
  - **alignment**, **clipBehavior color**, **constraints**, **decoration**, **margin**, **padding**, **transform**.

Container

Paddings permitem adicionar um espaço entre o **child** e as bordas do **Container**. Não confunda **paddigns** com **margins**, este último adiciona espaço entre as bordas do container e o espaço externo.

```
Center(child: Container(
    padding: const EdgeInsets.all(10.0),
    height: 240,
    width: 180,
    color: Colors.red,
    child:  Image.network(
      'https://flutter.github.io/'
      ↪  'assets-for-api-docs/assets/widgets/owl-2.jpg'),
  ),
);
```

Container

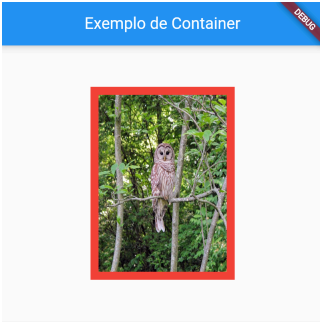


Container

Como a imagem tem aspect ratio diferente do **Container**, um **FittedBox** seria importante.

```
Center(child: Container(
    padding: const EdgeInsets.all(10.0),
    height: 240,
    width: 180,
    color: Colors.red,
    child: FittedBox(
      fit: BoxFit.fill,
      child: Image.network(
        'https://flutter.github.io/'
        ↪ 'assets-for-api-docs/assets/widgets/owl-2.jpg'),
    ),
));
```

Container



Container

O parâmetro **transform** permite fazer algumas operaçõs simples, como uma rotação.

```
Center(child: Container(
    padding: const EdgeInsets.all(10.0),
    height: 240,
    width: 180,
    color: Colors.red,
    child: FittedBox(
      fit: BoxFit.fill,
      child: Image.network(
        'https://flutter.github.io'
        ↪  'assets-for-api-docs/assets/widgets/owl-2.jpg'),
    ),
    transform: Matrix4.rotationZ(0.1),
));
```

Container

Container

O parâmetro **alignment** alinhar o filho no **Container**, útil quando o filho é pequeno. As opções **bottomCenter**, **bottomLeft**, **bottomRight**, **center**, **centerLeft**, **centerRight**, **topCenter**, **topLeft** e **topRight** são as mais comuns.

```
Center(child: Container(
    padding: const EdgeInsets.all(10.0),
    alignment: Alignment.bottomCenter,
    height: 240,
    width: 180,
    color: Colors.red,
    child: const Icon(Icons.alarm),
));
```

Ao configurar **aligment**, o comportamento do **Container** muda para tentar se expandir ao máximo. Esse crescimento está sendo limitado por **width** e **heigh** no exemplo.

Container

Container

Você pode substituir o parâmetro **color** pelo **decoration** para ter mais controle dos efeitos decorativos. Por exemplo, mantendo a cor e mudando o formato do background com **BoxDecoration**:

```
Center(child: Container(
    padding: const EdgeInsets.all(10.0),
    alignment: Alignment.center,
    height: 240,
    width: 180,
    decoration: const BoxDecoration(
      shape: BoxShape.circle,
      color: Colors.red,),
    child: const Icon(Icons.alarm),
));
```

Container

Container

O **BoxDecoration** mereceria uma seção só para ele, mas recomendo que você aprenda com o tempo por conta própria. Apenas olhe algumas opções: **gradient** e **boxShadow**.

```
Container(
  ...
  decoration: const BoxDecoration(
    gradient: LinearGradient(
      colors: [Colors.blue, Colors.red, Colors.yellow,
      ↪  Colors.green]),
    boxShadow: [BoxShadow(color: Colors.grey, blurRadius: 10)],
    shape: BoxShape.circle,
  ),
);
```

Container

Container

O parâmetro **contraints** permite que você limite o crescimento do **Container** ao invés de configurar **width** e **height**.

```
Center(child: Container(
    padding: const EdgeInsets.all(10.0),
    alignment: Alignment.center,
    color: Colors.red,
    constraints: const BoxConstraints(maxWidth: 100),
    child: const Icon(Icons.alarm),
));
```

Container

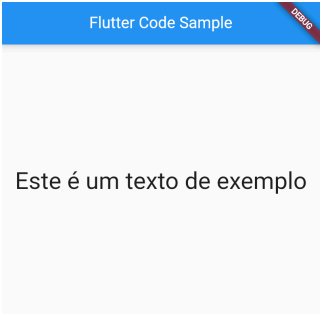
Text

Text

- A classe **Text** é aquela que aprendemos por primeiro, serve para colocar um texto simples na tela.
- Alguns elementos de decoração permitem criar uma interface mais agradável ao usuário. Esses elementos serão adicionados ao parâmetro opcional **style**.
- Formatações comuns são **fontWeight**, **fontStyle**, **color** e **letterSpacing**.

Text

```
Text("Este é um texto de exemplo",
  style: TextStyle(fontSize: 30),
);
```



Text

```
Text("Este é um texto de exemplo",
  style: TextStyle(fontSize: 18,
    fontWeight: FontWeight.bold,
    color: Colors.blueAccent
  ),
);
```

Text

```
Text("Este é um texto de exemplo",
  style: TextStyle(fontSize: 18,
    letterSpacing: 2,
    fontWeight: FontWeight.bold,
    color: Colors.blueAccent),
);
```

Text

```
Text("Este é um texto de exemplo",
  style: TextStyle(
    shadows: [Shadow(blurRadius:10, color:Colors.amber)],
    fontSize: 18,
    letterSpacing: 2,
    fontWeight: FontWeight.bold,color: Colors.blueAccent),
);
```