PROGRAMAÇÃO II

AULA 3

Prof. Elton Masaharu Sato

CONVERSA INICIAL

Começaremos a aprender como utilizar widgets do Flutter para criar um aplicativo. Esta etapa terá vários exercícios que poderão ser feitos na prática. É recomendado que você teste os códigos por si próprio em um compilador de sua preferência.

Para compilar os códigos desta etapa você pode tanto utilizar o editor já instalado ou o dartpad para as aplicações simples, com exceções, já que o dartpad não suporta certas bibliotecas como o dart.io, que possui alguns dos métodos de entrada e saída do programa.

No decorrer desta etapa, veremos os seguintes tópicos:

- stateless e stateful;
- widgets básicos;
- como montar uma interface de usuário com widgets;
- como receber entrada do usuário através de widgets.

TEMA 1 – STATELESS E STATEFUL

Como vimos anteriormente, widgets são alguns dos mais importantes pilares do Flutter. Praticamente tudo em Flutter é um widget. Com esses widgets o Flutter é capaz de montar os mais variados tipos de aplicativos.

Os widgets são divididos em dois grupos principais, os stateful e os stateless, que podem ser traduzidos como *com estado* e *sem estado*, respectivamente. A diferença principal entre os dois é a existência de uma variável de estado que pode alterar um widget, ou seja, um widget stateless é alterado por eventos externos, enquanto um widget stateful possui um mecanismo que altera o widget dependendo de seu estado.

Figura 1 – Analogia da sala de estar

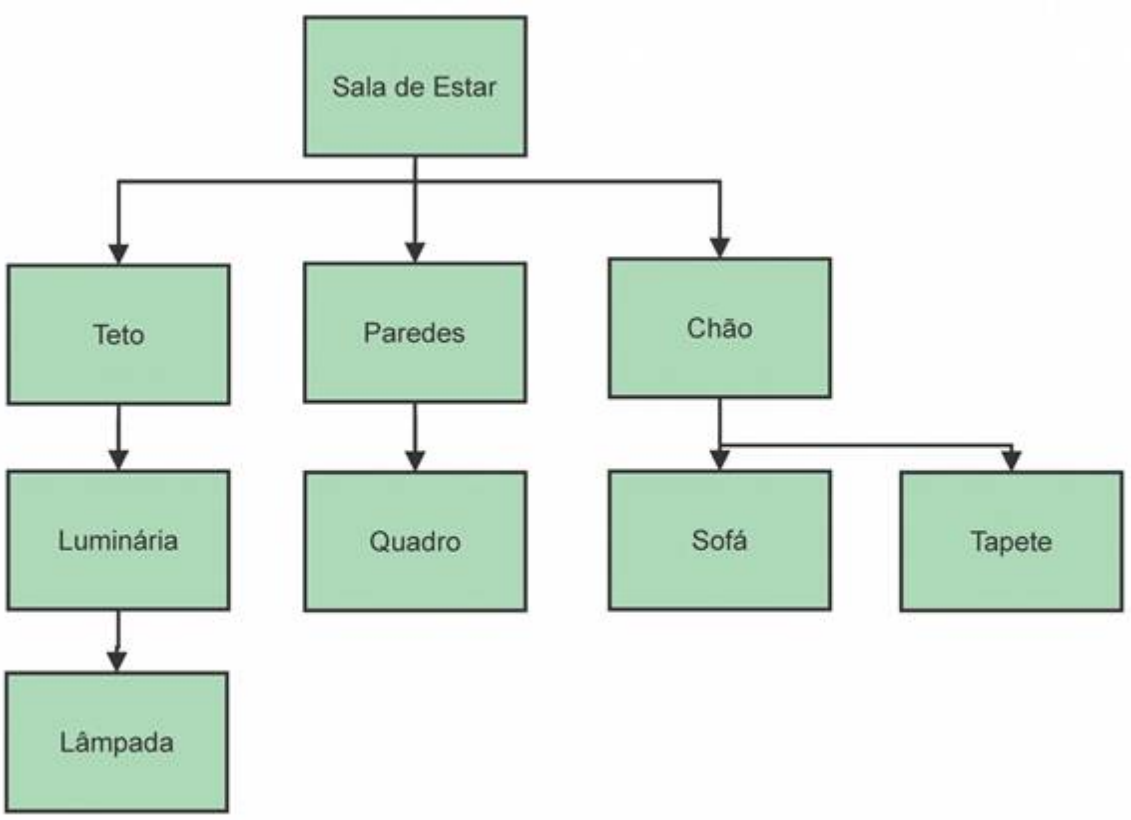
Crédito: Photographee.eu/Shutterstock.

Uma analogia que ajuda a entender é ver um aplicativo como uma sala de estar de uma casa (Figura 1). Widgets são objetos, assim como praticamente tudo na sala pode ser categorizado como um objeto.

Para saber quais objetos serão stateless e quais serão stateful, basta pensar em quais objetos mudam de estado e quais permanecem os mesmos. Por exemplo, o tapete, os objetos de decoração e o sofá podem ser stateless, já que só ficam parados. A televisão, a lareira e as lâmpadas podem ser stateful, já que podem estar nos estados “ligado” ou “desligado” e, no caso da televisão, pode haver diversos estados, considerando que cada canal ou programa pode ser um estado diferente da televisão.

Porém um objeto ser stateless não significa que ele não pode nunca ser alterado. O tapete, por exemplo, será colocado onde o programador o criar, mas é possível movê-lo manualmente depois. No entanto, se a intenção é que o usuário mude o tapete de lugar com frequência, talvez seja melhor programar um tapete mágico que se mova por conta própria com um widget stateful.

Figura 2 – Árvore de objetos da sala de estar



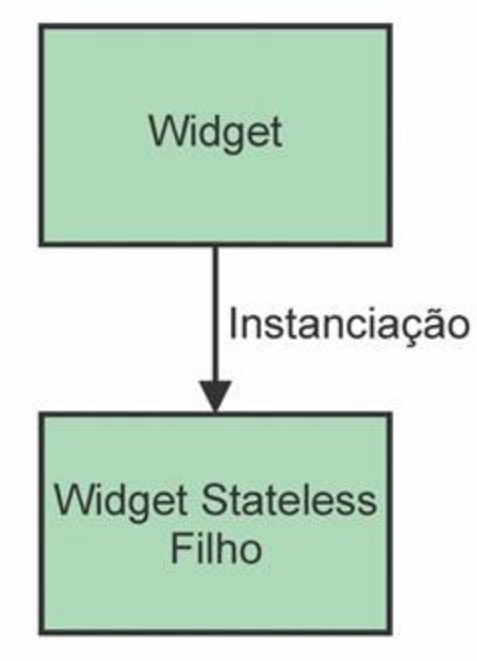
Fonte: Elton Masaharu Sato.

Na Figura 2 temos um diagrama que exemplifica uma árvore de objetos do exemplo da sala de estar. Dentro da sala de estar há o teto, as paredes e o chão e, “dentro” de cada um desses lugares, há seus respectivos objetos. Similarmente a uma árvore genealógica, um programa Flutter pode ser entendido como uma árvore genealógica de objetos.

1.1 STATELESS

Um aplicativo é tipicamente composto por muitos widgets e alguns deles continuam os mesmos depois que são criados; praticamente não se alteram no decorrer da execução do aplicativo. Esses widgets não se alteram por um comportamento próprio e devem ser manualmente alterados por um outro widget se quisermos que sejam alterados.

Figura 3 – Widget stateless



Fonte: Elton Masaharu Sato.

A Figura 3 apresenta um diagrama de como um widget stateless é criado. Durante sua instanciação, o widget stateless recebe seus valores iniciais e se mantém assim.

Vamos criar nosso primeiro código que escreve o texto “Olá Mundo!” na tela. Pode parecer complicado à primeira vista, mas no decorrer de nossa caminhada vamos aprender o que cada parte do código significa.

Figura 4 – Exemplo de código de “Olá Mundo!”

```
1 import 'package:flutter/material.dart';
2
3 void main() => runApp(MeuPrimeiroAplicativo());
4
5 class MeuPrimeiroAplicativo extends StatelessWidget {
6   Widget build(BuildContext context) {
7     return MaterialApp( home: Material( child: Text("Olá Mundo!") ) );
8   }
9 }
10
```

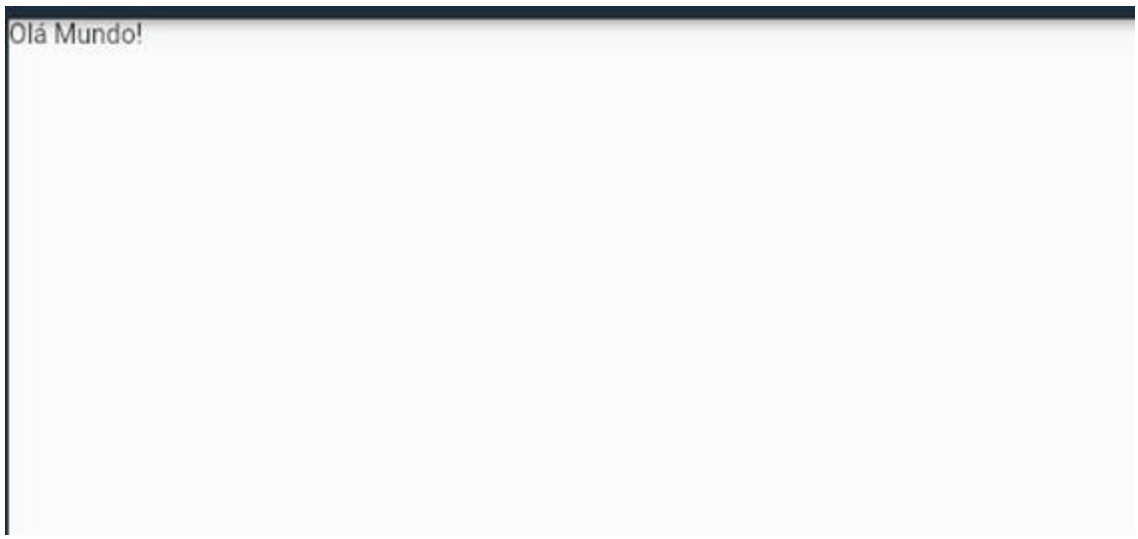
Fonte: Dartpad, 2022.

Vamos agora analisar linha a linha nosso código da Figura 4. Não é preciso se preocupar em tentar entender tudo de uma vez. O código só está aqui para termos um exemplo prático. Por enquanto só precisamos de uma ideia geral do que é um widget stateless.

- A linha 1 importa as funcionalidades do Flutter para o código.

- A linha 3 cria a nossa classe principal (main) e diz para ela rodar o widget do nosso aplicativo da classe “MeuPrimeiroAplicativo”.
- A linha 5 cria a classe “MeuPrimeiroAplicativo” e estende um widget stateless.
- A linha 6, dentro da nossa classe “MeuPrimeiroAplicativo”, chama um método que retorna um widget e lhe passará seu contexto. De forma simples, o contexto de um widget é onde ele se encontra na árvore genealógica de objetos.
- A linha 7 é o que o método acima retorna, neste caso, um widget do tipo MaterialApp. Esse MaterialApp possui um Material do tipo Texto com a string “Olá Mundo!”.

Figura 5 – Saída do código de “Olá Mundo!”



Fonte: Dartpad, 2022.

A Figura 5 mostra a saída do código da Figura 4. Caso o programa seja testado em um aparelho Android ou em um simulador, preste atenção ao texto que aparecerá no canto superior esquerdo da tela.

Aviso

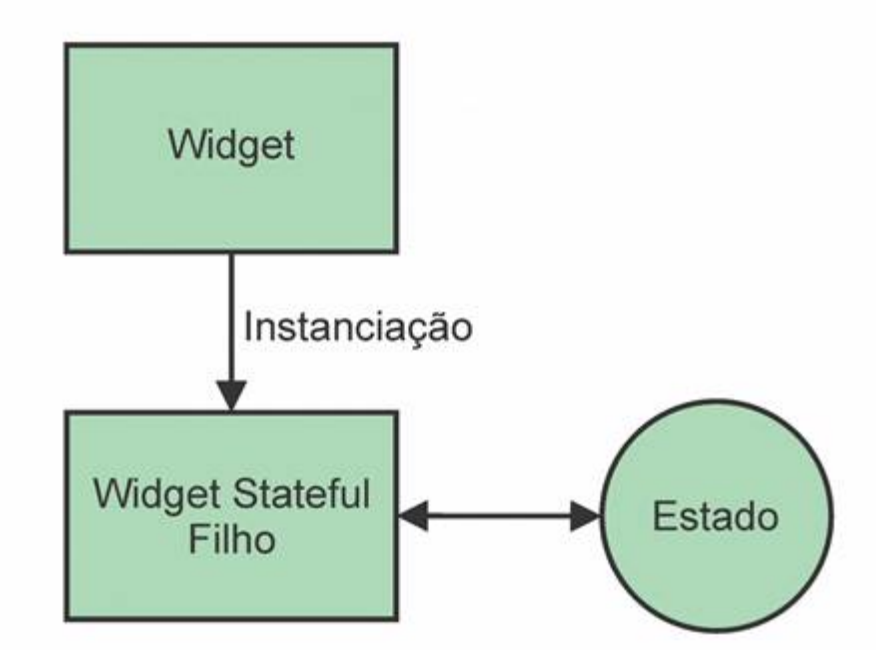
O Flutter é SenSíVel a LeTrAs MAIÚSCULAS E minúsculas, portanto, quando for escrever o código de exemplo, preste atenção nesse detalhe. Caso esteja utilizando o VS Code ou o dartpad, o editor irá mudar as cores dos textos para refletir o que está sendo escrito. Quando terminar de escrever e testar o código normalmente, mude algumas letras de minúsculas para maiúsculas e vice-versa para ver como o código se altera.

Exemplo: altere o primeiro comando “import” para “Import” e o código deverá parar de funcionar.

1.2 STATEFUL

Diferentemente dos widgets stateless, que são instanciados e permanecem estáticos até o final de sua execução, os widgets stateful possuem um estado que representa o estado atual do widget. A Figura 6 a seguir apresenta um diagrama de como um widget stateful é montado. Um widget stateful está ligado ao seu estado e, quando este for alterado, o widget será alterado.

Figura 6 – Widget stateful



Fonte: Elton Masaharu Sato.

A Figura 7 apresenta um código de um widget stateful que tem a seguinte função: um texto “...” e um botão logo abaixo. Quando clicarmos no botão deverá aparecer o texto “Olá Mundo!”. Vamos ver o código e analisar suas linhas.

Figura 7 – Exemplo de código de “Olá Mundo!” com widget stateful

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

Run | Debug | Profile
void main() {
  runApp(MeuPrimeiroAplicativo());
}

class MeuPrimeiroAplicativo extends StatefulWidget {

  @override
  State<MeuPrimeiroAplicativo> createState() => _MPAState();
}

class _MPAState extends State<MeuPrimeiroAplicativo> {

  String txt = '...';

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Column(
          children: <Widget>[
            Text(txt),
            FloatingActionButton(
              onPressed: (){
                setState(() {
                  txt =  "Olá Mundo!";
                });
              }
            ) // FloatingActionButton
            ], // <Widget>[]

      ), // Column
    ); // MaterialApp
  }
}
```

Fonte: Dartpad, 2022.

- A linha 1 importa as funcionalidades do Flutter para o código.
- As linhas 3, 4 e 5 criam nossa classe principal (main) e dizem a ela para rodar o widget do nosso aplicativo da classe “MeuPrimeiroAplicativo”. Note que essa é uma forma diferente de escrever a mesma função da linha 3 da Figura 4.
- A linha 7 cria a classe “MeuPrimeiroAplicativo” e estende um widget stateful.
- A linha 10 cria um estado para o widget.
- A linha 13 é a classe do nosso estado. Ela é uma extensão de um estado do widget “MeuPrimeiroAplicativo”; por convenção, colocamos um underline na frente de seu nome.
- Na linha 15 temos uma variável de texto que se inicia com “...”.

- Na linha 18 temos a construção do nosso widget de estado.
- A linha 20 descreve uma coluna de widgets.
- A linha 22 declara o primeiro widget da nossa coluna, um texto com a string da variável “txt”.
- A linha 23 declara um botão com uma única função; quando pressionado, ele chama o método setState.
- A linha 25 tem o setState, uma função da classe estado; quando chamado, reconstrói o widget stateful com as alterações feitas, nesse caso, a alteração do texto txt.

Figura 8 – Saída do código do stateful (antes de clicar no botão)

Fonte: Elton Masaharu Sato.

Figura 9 – Saída do código do stateful (depois de clicar no botão)

Fonte: Elton Masaharu Sato.

As figuras 8 e 9 apresentam as saídas do nosso exemplo de widget stateful. Quando o botão é pressionado, o texto muda de “...” para “Olá Mundo!”.

TEMA 2 – SCAFFOLD

Figura 10 – Exemplo de um scaffold

Fonte: Flutter, 2022.

Um scaffold, em tradução literal *andaime*, oferece suporte e armação para se montar uma apresentação visual clássica de um aplicativo, como mostra a Figura 10. Um scaffold pode ter os seguintes elementos:

- uma Appbar nas partes superior e inferior;
- um corpo central;
- barra de navegação inferior;
- um drawer na lateral com mais opções;
- botões;
- e outros.

Esses são alguns dos elementos mais comuns para um scaffold, mas é possível encontrar uma lista completa e atualizada a seguir. Disponível em: <docs.flutter.dev/flutter/material/Scaffold-class.ht

ml>. Acesso em: 8 jul. 2022.

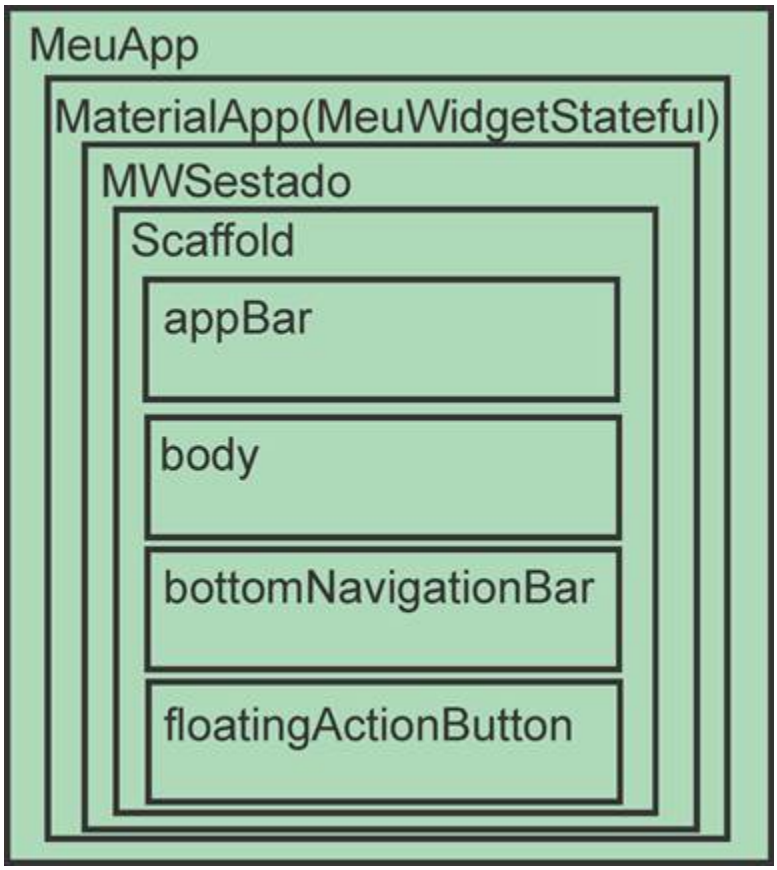
Figura 11 – Exemplo de scaffold

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

void main() => runApp(MeuApp());

class MeuApp extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MeuWidgetStateful(),
    );
  }
}

class MeuWidgetStateful extends StatefulWidget {

  @override
  State<MeuWidgetStateful> createState() => _MWSestado();
}

class _MWSestado extends State<MeuWidgetStateful> {
  bool pressed = false;
  String txt = "...";

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Aplicativo de Olá Mundo!'),
      ),
      body: Center(
        child: Text(txt),
      ),
      bottomNavigationBar: BottomAppBar(
        shape: const CircularNotchedRectangle(),
        child: Container(height: 50.0),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => setState(() {
          if(pressed){
            txt = 'Olá Mundo!';
            pressed = false;
          } else {
            txt = '...';
            pressed = true;
          }
        }),
        child: const Icon(Icons.favorite),
      ),
      floatingActionButtonLocation: FloatingActionButtonLocation.centerDocked,
    );
  }
}

```

Fonte: Dartpad, 2022.

O código da Figura 11 pode ser explicado utilizando um diagrama de Venn como o da Figura 12.

Figura 12 – Diagrama do código do scaffold



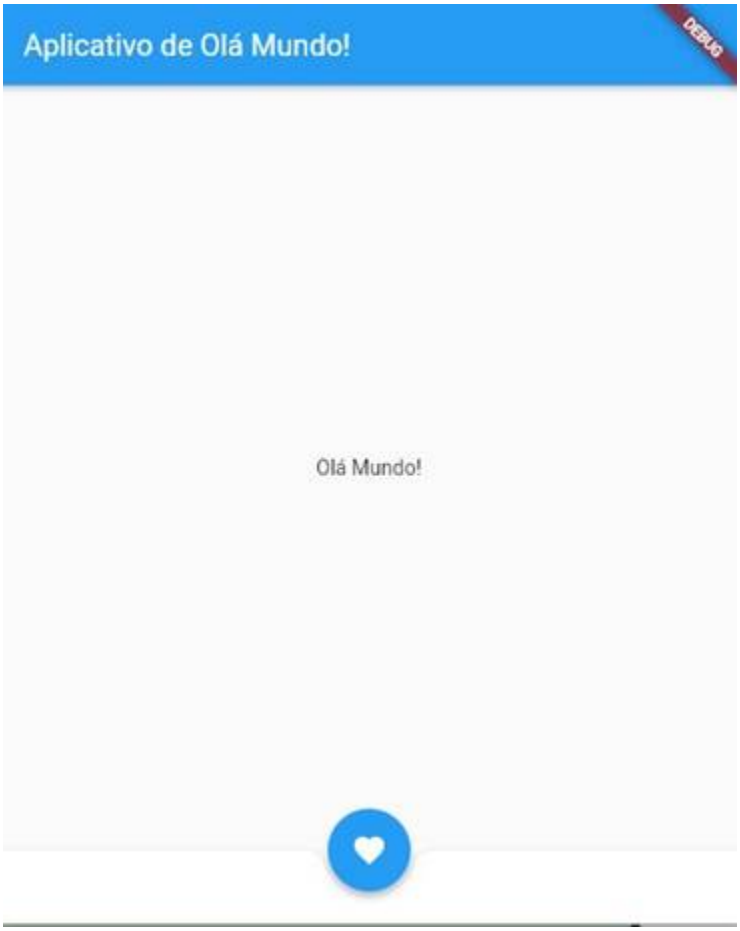
Fonte: Elton Masaharu Sato.

Observando o código do scaffold e o diagrama de Venn, podemos perceber como o aplicativo é montado.

- Começamos pela main que cria o “MeuApp”.
- Nosso “MeuApp” cria o “MeuWidgetStateful”.
- “MeuWidgetStateful” cria o estado “_MWSestado”.
- “_MWSestado” cria um scaffold.
- O scaffold possui um appBar, um body, um bottomNavigationBar e um floatingActionButton.

A Figura 13 apresenta a saída do nosso aplicativo. Note que nosso floatingActionButton tem um pequeno trecho de código que altera o texto de “...” para “Olá Mundo!” e vice-versa.

Figura 13 – Saída do código do scaffold



Fonte: Dartpad, 2022.

TEMA 3 – WIDGETS DE TEXTO E IMAGEM

Como o Flutter possui um grande foco na interface do usuário, alguns dos widgets mais importantes que você irá aprender são os blocos básicos de montagem do nosso aplicativo.

3.1 WIDGET DE TEXTO

Um widget de texto é escrito da seguinte forma:

- Text(String texto).

Porém, dentro dele é possível passar vários outros parâmetros com o texto, como se segue.

- locale: seleciona uma fonte diferente dependendo da localização passada.
- MaxLines: determina uma quantidade máxima de linhas.
- overflow: determina o que fazer quando o texto é muito longo para o espaço do texto.
- style: estilo do texto; inclui informações como fonte, tamanho da fonte e se o texto está em negrito, itálico ou outras opções de estilo e cor.

- TextAlign: alinhamento do texto horizontalmente.
- TextDirection: é possível alterar a direcionalidade do texto para textos que devem ser invertidos.

Exemplo:

```
Text(
        'Exemplo de Texto',
        textAlign: TextAlign.center,
        style: TextStyle(
                fontWeight: FontWeight.bold,
                fontSize 18.0
        )
)
```

Saída do exemplo:

Exemplo de Texto

Caso deseje escrever um texto mais complexo e rico, utilizando diversas alterações de estilos, recomenda-se o uso deste construtor:

- Text.rich(InlineSpan textSpan).

Esse construtor também aceita diversos parâmetros como o anterior, porém utiliza um outro tipo de texto, o textSpan, que recebe um conjunto de textos.

Exemplo:

```
Text.rich(
        TextSpan(
                text: ' Um exemplo ',
                children: <TextSpan>[
                        TextSpan(text: ' de um ', style: TextStyle(fontStyle: FontStyle.italic)),
                        TextSpan(text: ' trecho de texto. ', style : TextStyle(fontStyle: FontStyle.bold))
                ]
        )
)
```

Saída do exemplo:

Um exemplo *de um* **trecho de texto.**

3.2 WIDGET DE IMAGEM

Uma interface deve transmitir mensagens não apenas textuais, mas também visuais para seu usuário. Para isso, devemos ser capazes de anexar imagens às nossas interfaces. Existem diversas formas de declarar uma imagem:

- Image.asset. (Carrega uma imagem do aplicativo.)

Exemplo:

```
Image.asset(
        'assets/images/exemplo.jpg'
)
```

- Image.network. Carrega uma imagem da internet. O usuário precisa de conexão com a internet para conseguir carregar a imagem e o carregamento pode ser lento dependendo da velocidade da conexão e do tamanho da imagem.

Exemplo:

```
Image.network(
        'https://upload.wikimedia.org/wikipedia/commons/3/31/Pizza_mezzo_a_mezzo.jpg'
)
```

- Image.file. Carrega uma imagem de um arquivo do dispositivo do usuário.

Exemplo:

```
Image.file(
        '/caminho/do/arquivo.png'
)
```

- Image.memory. Mais raramente utilizado, pode carregar uma imagem que está na RAM do dispositivo.

Assim como os widgets de texto, os widgets de imagem também podem ser alterados com o uso de parâmetros.

- alignment: altera o alinhamento da posição da imagem.
- colorBlendMode: aplica um filtro de cor na imagem.
- color: determina a cor que será utilizada para aplicar o filtro de cor.
- fit: determina como ajustar a imagem para encaixar em um espaço.
- height: determina a altura da imagem.
- width: determina a largura da imagem.
- loadingBuilder: cria uma barra de carregamento para a imagem.
- opacity: determina o quão opaca/transparente é a imagem.
- repeat: se houver espaço sobrando, repete a imagem para cobrir o espaço vazio.
- semanticLabel: uma descrição da imagem que é lida para usuários de TalkBack ou VoiceOver.

Exemplo:

```
Image.network(
        'https://upload.wikimedia.org/wikipedia/commons/3/31/Pizza_mezzo_a_mezzo.jpg',
        loadingBuilder: (context, child, progress){
                return progress == null ? child : LinearProgressIndicator();
        },
        height:400,
        width:400

)
```

TEMA 4 – WIDGETS DE BOTÃO E ORGANIZAÇÃO

4.1 WIDGET DE BOTÃO

Flutter não possui um widget de botão genérico, mas possui vários tipos diferentes de botões que seguem as convenções de Material Design da Google e do iOS Cupertino. A seguir, alguns dos tipos comuns de botões a serem utilizados.

- ElevatedButton. Um botão retangular simples com animação ao ser pressionado. Se a função "onPressed" for "null", o botão fica cinza com uma aparência de desabilitado.

Figura 14 – Exemplo de botão elevado

Fonte: Elton Masaharu Sato

- FloatingActionButton. Um botão arredondado que flutua sobre a interface.

Figura 15 – Exemplo de botão flutuante

Fonte: Elton Masaharu Sato

- TextButton. Um botão em forma de texto. Note que os outros tipos de botões também podem conter texto.

Figura 16 – Exemplo de botão de texto

Fonte: Elton Masaharu Sato

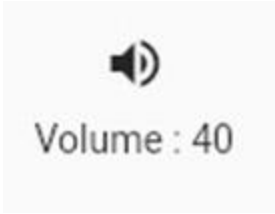
- OutlinedButton. Similar a um TextButton, porém com uma linha de contorno ao redor do botão.

Figura 17 – Exemplo de botão com contorno

Fonte: Elton Masaharu Sato

- IconButton. Similar a um TextButton, porém utiliza um ícone em vez de um texto.

Figura 18 – Exemplo de botão com ícone



Fonte: Elton Masaharu Sato

Dependendo do tipo de botão, ele terá diferentes tipos de propriedades além da "onPressed", que verifica quando um botão é clicado.

- Color: cor do botão, quando aplicável.
- Icon: ícone do botão, quando aplicável.
- Label: texto do botão, quando aplicável.

4.2 LINHA E COLUNA (ROW E COLUMN)

Um único widget na tela não é uma ótima forma de criar uma interface amigável para o usuário. Normalmente é utilizada uma coleção de widgets que são dispostos com a ajuda de widgets de organização como *containers*. Assim como o scaffold cria uma interface, podemos utilizar rows (linhas) e columns (colunas) para organizar os nossos widgets na tela ou dentro de um scaffold.

Exemplo de row:

```
Row(
        children: <Widget>[
                Text('Widget 1'),
                Icon(Icons.thumb_up),
                Text('Widget 3')
        ]
)
```

Figura 19 – Exemplo de widgets em linha

Exemplo de column:

```
Column(
        children: <Widget>[
                Text('Widget 1'),
                Icon(Icons.thumb_up),
                Text('Widget 3')
        ]
)
```

Figura 20 – Exemplo de widgets em coluna

As figuras 19 e 20 exemplificam a saída dos nossos códigos. Perceba que os códigos são escritos de formas similares; declara-se row ou column e, então, declara-se uma lista de children (filhos).

TEMA 5 – GESTOS DO USUÁRIO

Interação com o usuário é uma das características mais importantes de um aplicativo para que o usuário tenha uma experiência personalizada. Uma das mais importantes ferramentas de interação com um usuário são os gestos que ele realiza na tela.

5.1 WIDGET DE DETECÇÃO DE GESTOS

Para criar um detector de gestos, cria-se um widget assim como os outros widgets vistos.

Vejamos um exemplo de código.

```
GestureDetector(
        função: (){
                //código que será executado
        }
)
```

O detector de gestos requer uma função que é a condição de ativação do gesto, ou seja, no lugar onde aparece “função”, o desenvolvedor deverá escrever uma das possíveis condições de ativação. As mais comuns são as que se seguem.

- onTap: toque simples; o widget chama o código quando o widget for tocado com toque simples.
- onDoubleTap: toque duplo; o widget chama o código quando o widget for tocado duas vezes rapidamente.
- onLongPress: toque e manter pressionado; o widget chama o código quando o widget for pressionado por um período longo de tempo, aproximadamente 1 segundo.

É possível também detectar movimentos na horizontal e na vertical em conjunto com o toque da tela, utilizando propriedades como estas.

- onHorizontalDragDown: detecta quando um gesto na tela pode iniciar um arrastar na horizontal.
- onHorizontalDragStart: detecta o começo de um movimento de arrastar na horizontal.
- onHorizontalDragUpdate: detecta o movimento de arrastar na horizontal.
- onHorizontalDragEnd: detecta o término de um movimento de arrastar na horizontal.
- onHorizontalDragCancel: detecta quando um movimento de arrastar na horizontal é cancelado ou não foi completado.
- onVerticalDragDown: detecta quando um gesto na tela pode iniciar um arrastar na vertical.
- onVerticalDragStart: detecta o começo de um movimento de arrastar na vertical.
- onVerticalDragUpdate: detecta o movimento de arrastar na vertical.
- onVerticalDragEnd: detecta o término de um movimento de arrastar na vertical.
- onVerticalDragCancel: detecta quando um movimento de arrastar na vertical é cancelado ou não foi completado.

Além desses gestos, dependendo do aparelho, é possível detectar a pressão que o usuário está colocando na tela. Para isso, devemos utilizar as propriedades:

- onForcePressStart: detecta quando um usuário colocou pressão o suficiente para começar um evento do tipo “pressionar com força”.
- onForcePressUpdate: detecta quando o usuário estiver mantendo a pressão suficiente na tela.
- onForcePressPeak: detecta quando o usuário atinge o máximo possível de detecção de pressão na tela.
- OnForcePressEnd: detecta quando um toque com pressão não estiver mais em contato com a tela.

Código com um aplicativo de testes compatível com dartpad (note que a função de pressão só funcionará em um aparelho que detecte pressão na tela):

```
import
'package:flutter/material.dart';
void main() {

 runApp(MaterialApp(
   title: "Detector de Gestos",
   debugShowCheckedModeBanner:
   false, home: MyApp(),  ));

}
class MyApp extends StatefulWidget {
 @override  _MyAppState createState() => _MyAppState(); }


class _MyAppState extends State<MyApp>
 { String _title = "Detector de Gestos";
 @override
void initState() {
 super.initState();

 _title = "Detector de Gestos";
}
@override
Widget build(BuildContext context) {
 return Scaffold(

   appBar:
    AppBar(
    centerTitle:
    true,

    title: Text(_title),      ), body: ListView( children: <Widget>[

     GestureDetector(
      onTap: () {

        _changeTitle("onTap");      },
```

5.2 DRAWER

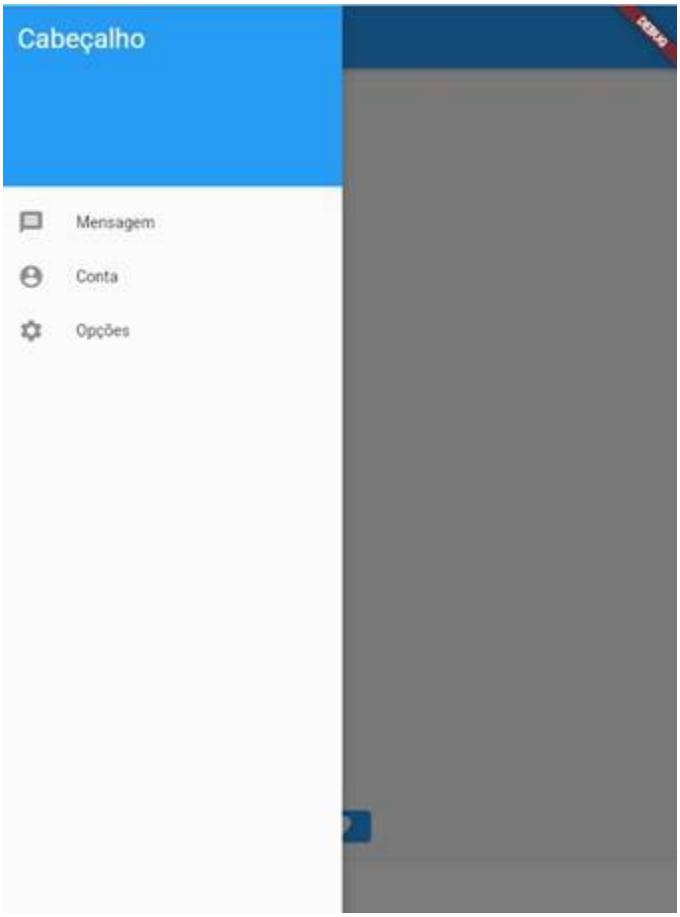
Existe um widget muito comum que já vem com um detector de gestos, que é o drawer, ou *gaveta*, em português. Esse widget é parte de um scaffold, portanto é necessário criar um scaffold caso queira utilizá-lo.

Vejamos um exemplo de código de um widget de drawer.

```
drawer: Drawer(
        //widgets que farão parte do seu drawer
      )
```

Devido à natureza de um drawer, é bastante comum que se coloque uma lista de widgets para ficar na gaveta, como mostra a Figura 21.

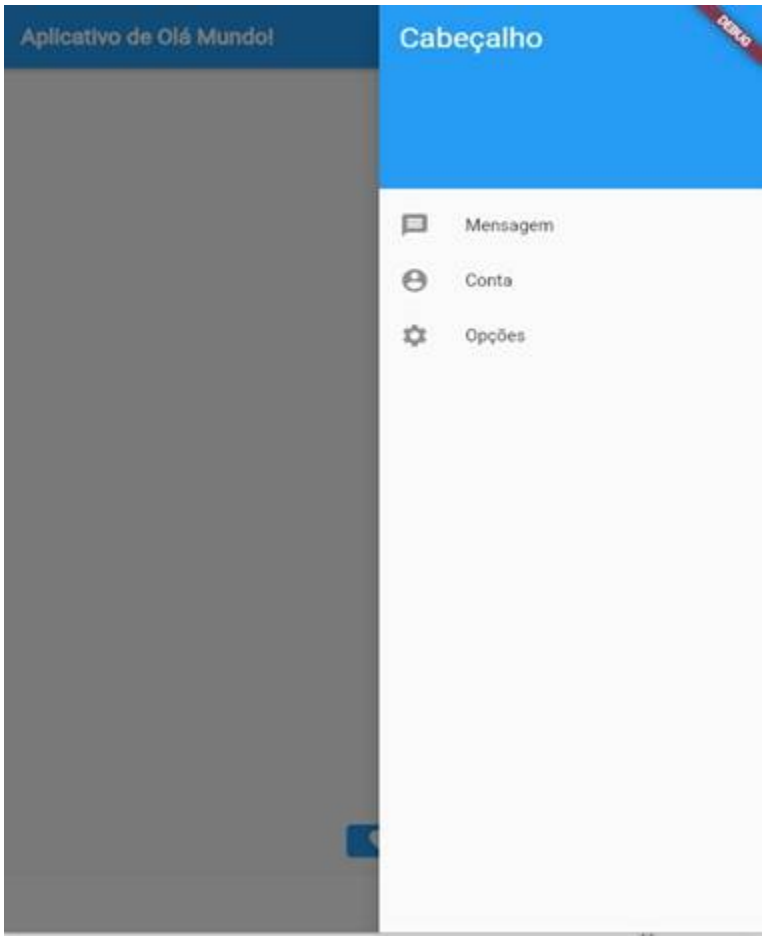
Figura 21 – Exemplo de um drawer



Fonte: Elton Masaharu Sato

O drawer é normalmente aberto com um gesto de movimento que começa no lado esquerdo da tela e se move para o lado direito. Mas é possível também criar um endDrawer, uma gaveta que vem da direita em vez da esquerda. A figura a seguir mostra um exemplo desse drawer.

Figura 22 – Exemplo de endDrawer



Fonte: Elton Masaharu Sato

Os seguintes métodos também são úteis para usar com um drawer.

- openDrawer: abre um drawer de um scaffold.
- openEndDrawer: abre um endDrawer de um scaffold.
- Navigator.pop: um método que veremos com mais detalhes adiante, mas esta função fecha um drawer que estiver aberto.
- DrawerEnableOpenDragGesture: se for falso, desabilita os gestos para abrir um drawer.
- EndDrawerEnableOpenDragGesture: se for falso, desabilita os gestos para abrir um endDrawer.

FINALIZANDO

Nesta etapa, vimos bastante sobre widgets, desde os conceitos de widgets stateful e stateless até os widgets com propriedades mais visíveis como textos, imagens, e onde colocamos esses widgets, como os scaffolds e drawers.

Aprendemos também a detectar interações do nosso usuário, desde simples botões dos mais variados tipos até os gestos que um usuário fizer na tela.

REFERÊNCIAS

DART. **DartDevTools**. Disponível em: <https://dart.dev/tools/dart-devtools>. Acesso em: 22 jun. 2022.

DARTPAD. Disponível em: <https://dartpad.dev/?>. Acesso em: 22 jun. 2022.

FLUTTER. **Scaffold class.** Disponível em: <docs.flutter.dev/flutter/material/Scaffold-class.html>. Acesso em: 22 jun. 2022.

_____. **Image class**. Disponível em: <https://api.flutter.dev/flutter/widgets/Image-class.html>. Acesso em: 22 jun. 2022.

_____. **Text class**. Disponível em: <https://api.flutter.dev/flutter/widgets/Text-class.html>. Acesso em: 22 jun. 2022.

_____. **FloatingActionButton class**. Disponível em: <https://api.flutter.dev/flutter/material/FloatingActionButton-class.html>. Acesso em: 22 jun. 2022.

_____. **ElevatedButton class**. Disponível em: <https://api.flutter.dev/flutter/material/ElevatedButton-class.html>. Acesso em: 22 jun. 2022.

_____. **TextButton class**. Disponível em: <https://api.flutter.dev/flutter/material/TextButton-class.html>. Acesso em: 22 jun. 2022.

_____. **OutlinedButton class**. Disponível em: <https://api.flutter.dev/flutter/material/OutlinedButton-class.html>. Acesso em: 22 jun. 2022.

_____. **Drawer class**. Disponível em: <https://api.flutter.dev/flutter/material/Drawer-class.html>. Acesso em: 22 jun. 2022.

_____. **endDrawer property**. Disponível em: <https://api.flutter.dev/flutter/material/Scaffold/endDrawer.html>. Acesso em: 22 jun. 2022.