DESENVOLVIMENTO *MOBILE*

SOLUÇÕES EDUCACIONAIS INTEGRADAS

Componentes especializados de interface de usuário em Flutter

Carlos Wagner de Queiroz

OBJETIVOS DE APRENDIZAGEM

- Descrever o funcionamento de *widgets* especializados.
- Implementar caixas de diálogo.
- Construir telas para manipulação de coleções.

Introdução

As empresas estão apostando cada vez mais em aplicativos *mobile* para facilitar a comunicação e a fidelização de clientes. As vantagens são inúmeras e ganham força por conta do constante crescimento no uso de dispositivos móveis. Com a cultura de entrega rápida, motivada pelo *mindset* digital (mentalidade digital), alguns programadores usam *frameworks* para agilizar seus trabalhos.

O Flutter é um *kit* de desenvolvimento de *software* (SDK) elaborado pela Google, de código aberto, muito usado para produção de aplicações *mobile*, tanto para Android como iOS. O seu ciclo de desenvolvimento é orientado ao *design* e usa os *widgets* como blocos básicos para a customização da interface do usuário do aplicativo.

Neste capítulo, vamos nos aprofundar mais nos conhecimentos dos *widgets* especializados do Flutter, melhorando a sua habilidade no desenvolvimento das interfaces *mobile*, mostrando como implementar componentes, caixa de diálogo e como construir telas a partir da manipulação de coleções. O Flutter é um facilitador que trará muitas vantagens para o seu trabalho como programador.

Widgets especializados

O Flutter é projetado para trabalhar em camadas, e cada camada é isolada de forma independente com bibliotecas integradas (Figura 1). A camada *widgets* é uma abstração de composição. *Widgets* são os blocos de construção da interface do usuário de um aplicativo Flutter, e cada *widget* é uma declaração imutável de parte da interface do usuário (FLUTTER..., 2020).

Com base na composição, os *widgets* trabalham em uma hierarquia em que cada *widget* herda propriedades do seu `parent` e vai até o *widget* raiz, que é o contêiner que hospeda o aplicativo Flutter, normalmente MaterialApp ou CupertinoApp (Figura 1).

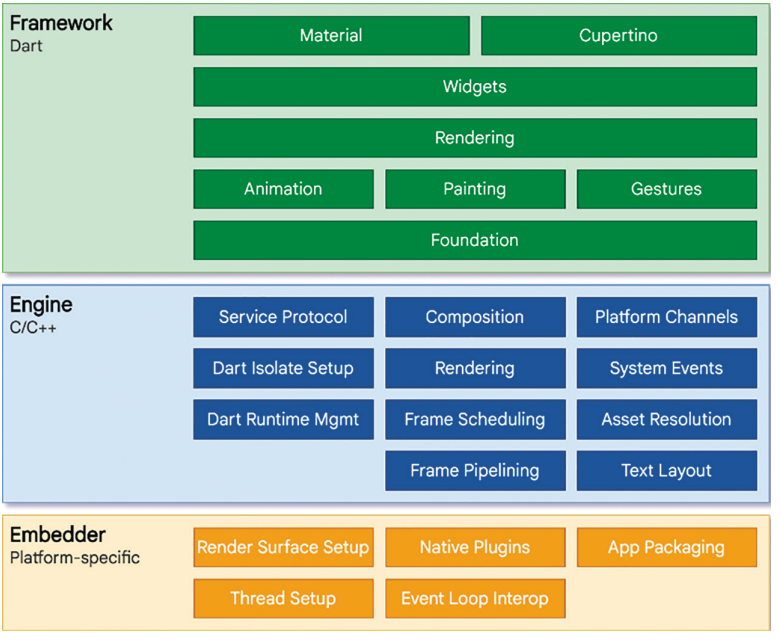

Figura 1. Diagrama de arquitetura do Flutter.
Fonte: Flutter... ([2021], documento *on-line*).

Outra característica interessante do Flutter é o uso de uma máquina virtual (VM) e o recurso de *hot reloads* (recarga automática), ou seja, quando é feita a edição do código, todos as instâncias dos componentes são modificadas e é possível visualizar as alterações sem a necessidade de uma recompilação completa.

O Flutter tem uma camada de código em C/C++, mas sua implementação é basicamente feita em Dart, uma linguagem de programação desenvolvida pela Google, inicialmente, para substituir o JavaScript em desenvolvimento de *scripts web*.

Saiba mais

Para começar os seus trabalhos de desenvolvimento de aplicativos com o Flutter é preciso instalar um *software* de desenvolvimento integrado para as plataformas de dispositivos móveis, como o Visual Studio Code ou o Android Studio (para nossos exemplos neste capítulo, usaremos o Android Studio); outra alternativa *web* para treino é o Dartpad.

É necessário fazer a instalação do Flutter e, então, fazer a extração para uma pasta que não requeira privilégios de administrador, como a pasta c:\Program Files\. A dica é criar uma pasta diretamente no diretório C:\. Após fazer a extração, no Explorador de Arquivos Explorer, clique com o botão direito em Este Computador (Meu Computador) e escolha opção Propriedades > Configurações avançadas do sistema > Avançado > Variáveis de ambiente, clique em Path > Editar > Novo, entre na pasta Bin (subpasta do diretório Flutter que você extraiu), copie o caminho do diretório e cole como novo Path. Para finalizar, é preciso configurar o Flutter no Android Studio. Clique em File > Settings > Plugins e instale os *plugins* para o Dart e para o Flutter.

Ao abrir o Android Studio novamente, caso a janela de boas-vindas não apareça, clique em File > Settings > Appearance & Behavior > System Settings e desmarque "Reopen last project on startup". Reinicie o Android Studio e perceba que a tela de boas-vindas voltou a aparecer; inclusive, apareceu também a opção Create New Flutter Project. Clique em Configure > AVD Manager e instale um emulador para os seus projetos.

Embora o Flutter direcione o desenvolvimento de suas aplicações para a linguagem Dart, é possível criar funcionalidades usando Java/Kotlin ou Objective-C/Swift no aplicativo *host*, porque o Flutter usa componentes padrões específicos do Android ou do iOS, trabalhando com as APIs específicas de cada plataforma.

Por sua vez, esses componentes têm a função de determinar e organizar o posicionamento de outros *widgets* (*widgets* de *layout*) ou criar componentes que serão visualizados pelos usuários no aplicativo (*widgets* de interface).

Dentre os principais *widgets* de *layout*, temos:

- Scaffold: é um *widget* que gera um *layout* padrão para aplicativos, contendo uma appBar e o conteúdo da tela;
- Stack: faz a função de "empilhamento" dos *widgets* na tela, do topo ao final da interface;
- Container: utilizado para comportar *widgets* em uma estrutura separada por bordas.

Os *widgets* de interface são os elementos gráficos que serão incorporados no *layout* para serem organizados na tela. Componentes visuais são: botões, textos, ícones, camadas, enfim, qualquer elemento visual que compõe o *layout*.

Dois conjuntos de *widgets* de interface são amplamente utilizados pelo Flutter, o Material e o Cupertino. O primeiro "desenha" os *widgets* de interface com os padrões do Material Design da Google, e o segundo com base no iOS (PINHEIRO, 2020).

Nesta seção, vamos abordar *widgets* fundamentais no desenvolvimento de qualquer aplicativo, explorando duas abordagens de navegação entre seções, uma incorporada à barra de ferramentas appBar em forma de botão (Drawer), e outra, o BottonNavigationBar, alocada na parte inferior do *layout* do aplicativo. Os *widgets* de navegação ajudam o usuário a se locomover dentre as seções do aplicativo; em aplicativos com várias seções, pode-se usar uma barra inferior de navegação (BottonNavigationBar) e um botão como Drawer. É uma abordagem muito usada para garantir a navegabilidade entre todas as seções sem "poluir" o *design* do aplicativo.

AppBar

O appBar é um widget básico para a criação de qualquer aplicativo. É uma barra de aplicativos de Material Design, ou seja, é um elemento que potencializa outros *widgets*. Esse tipo de barra geralmente é usado na propriedade Scaffold.appBar.

Para ilustrarmos sua importância na prática, vamos implementar um appBar simples. Depois incorporaremos um *widget* de navegação e acrescentaremos alguns itens (*menus*) como exemplo.

Para a implementação do appBar, digite:

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "CursosOnline",
      home: Scaffold(
        appBar: AppBar(
          title: Text('CursosOnline'),
        ),
      ),
    );
  }
}
```

Vamos implementar recursos ao nosso appBar acrescentando uma opção de navegação. Em aplicativos que usam Material Design, existem duas opções principais de navegação: `tabs` e `drawers`. Quando não há espaço suficiente para apoiar as `tabs`, `drawer` é uma alternativa prática.

Drawer

`Drawer` (gaveta), como o nome sugere, é um elemento visual como um botão "hambúrguer", aquele botão de *menus* presente em quase todos os aplicativos móveis. Esse botão é uma propriedade do *widget* `Scaffold` que, por padrão, fica alocado no appBar.

Portanto, acrescente o comando ao código logo após `Scaffold`:

```
home: Scaffold(
  drawer: Drawer(
      ),
  appBar: AppBar(
```

Pronto, agora basta configurarmos o nosso `drawer`. Vamos colocar as informações do usuário, como se ele estivesse logado, e os itens do *menu*. Para isso, acrescente o código para o `Drawer` como mostra seguir:

```
drawer: Drawer(
  child: ListView(
    padding: EdgeInsets.zero,
    children:<Widget>[
      UserAccountsDrawerHeader(
          accountName: Text("Fulano de Tal"),
          accountEmail: Text("fulano@mail.com"),
      currentAccountPicture: CircleAvatar(
        child: Text("FULANO")
      ),
      ), //UserAccoutsDrawerHeader
    ]
  )
),
appBar: AppBar(
```

O `Drawer` deve ficar como mostra a Figura 2, mas não acaba por aqui; agora vamos adicionar as listas dos *menus* do nosso botão.

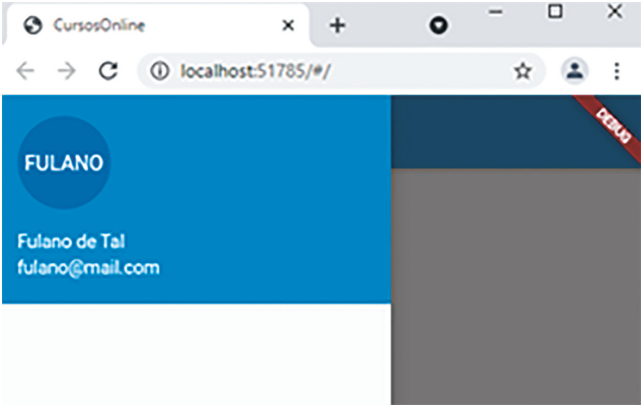

Figura 2. Primeiras inserções ao `Drawer`.

Agora vamos adicionar uma lista de itens (*menus*) ao `Drawer`. Nessa lista, vamos inserir os *menus* que reportarão futuras páginas do aplicativo. Usaremos ícones da biblioteca Material Icons para ilustrar cada opção.

Acrescente os códigos após o fechamento da `UserAccountsDrawerHeader`, como mostrado a seguir:

```
  child: Text("FULANO")
),
),//UserAccoutsDrawerHeader

ListTile(
  leading: Icon(Icons.person),
  title: Text("Minha Conta"),
  onTap:(){
    Navigator.pop(context);
  }
),
ListTile(
  leading: Icon(Icons.shopping _ basket),
  title: Text("Meus pedidos"),
  onTap:(){
    Navigator.pop(context);
  }
),
ListTile(
    leading: Icon(Icons.favorite),
    title: Text("Favoritos"),
    onTap:(){
      Navigator.pop(context);
    }
),
]
          ),
        ),
```

```
        appBar: AppBar(
          title: Text('CursosOnline'),
        ),
      ),
    );
  }
}
```

O `Drawer` deve ficar como mostra a Figura 3.

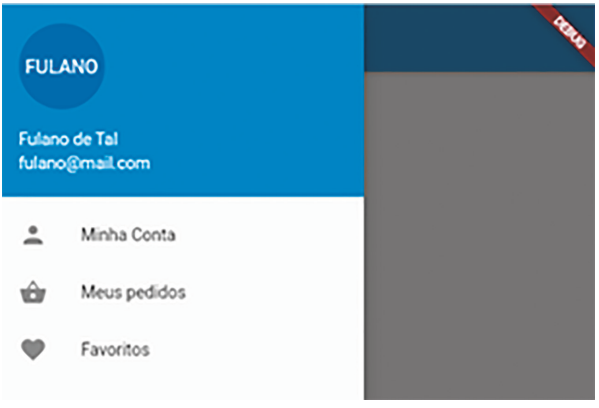

Figura 3. Opções do `Drawer`.

Fique atento

O TabBar é um *widget* de Design Material que também pode ser usado como navegador. É geralmente usado no appBar em conjunto com um TabBarView. Esse *widget* gera uma linha horizontal de guias. Você pode fazer as guias mostrarem textos, ícones ou especificar um *widget* filho. Depois, é só criar conteúdo para cada guia.

BottonNavigationBar

Alguns *designers* preferem o uso do BottonNavigationBar em vez do `Drawer`, outros preferem os dois. A tomada de decisão sobre qual elemento compositivo usar deve considerar o tipo de aplicativo, a quantidade de seções e o perfil do público-alvo (leve em consideração a usabilidade e acessibilidade).

O BottonNavigationBar é uma barra de navegação que fica na parte inferior do *layout* do aplicativo, também conhecida como barra de guias no iOS. Permite que o usuário alterne entre as diferentes seções do aplicativo rapidamente. Esse componente de navegação, assim como o `Drawer`, compõe a Widget Scaffold (KHAN, 2019).

Esse tipo de navegação fará alterações no estado da tela da aplicação, por isso é do tipo StatefulWidget. Crie um novo projeto para essa aplicação, para isso digite o código a seguir:

```
import 'package:flutter/material.dart';
void main() {
  runApp(MaterialApp())
    home: HomeScreen(),
  ),
  );
}
class HomeScreen extends StatefulWidget {
  @override
  _HomeScreenState createState() => _HomeScreenState();
}
class _HomeScreenState extends State<HomeScreen> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
    );
  }
}
```

Agora que a estrutura da página está concluída, vamos acrescentar o BottonNavigatorBar ao *widget* Scaffold com os mesmos itens que usamos para o `Drawer` no exemplo anterior (Minha conta, Meus pedidos e Favoritos). Acescente ao código a *widget* Scaffold, conforme o código seguinte:

```
return Scaffold(
      appBar: AppBar(),
      bottomNavigationBar: BottomNavigationBar(
         currentIndex:0,
items: [
          BottomNavigationBarItem(
            icon: Icon(Icons.person),
            label: "Minha Conta"
              ),
          BottomNavigationBarItem(
              icon: Icon(Icons.shopping _ basket),
            label: "Meus Pedidos"
          ),
          BottomNavigationBarItem(
              icon: Icon(Icons.favorite),
            label: "Favoritos"
          ), ], ), ); }}
```

Para a navegação, basta criar a lista de páginas que serão linkadas com os botões do BottonNavigationBar e definir o conteúdo do corpo de cada página do aplicativo que irá modificar de acordo com a opção clicada no navegador, mantendo o BottomNavigationBar fixo.

Caixas de diálogo

As caixas de diálogo são próprias para informar algo que não pode passar despercebido pelo usuário, pode ser uma informação que mereça destaque, uma pergunta, um alerta de erro, um anúncio publicitário, dentre outros. Elas são exibidas em *pop-up* e são compostas por títulos (`title`), conteúdo (`content`), geralmente de texto, e botões de ação (`actions`) (FERNANDO, 2021).

O `AlertDialog` é um dos componentes mais usados para criação de caixas de diálogo. O `AlertDialog` usa os recursos do Material Design para aplicações para Android, e o `CupertinoAlertDialog` usa parâmetros parecidos para renderizar para aplicações iOS. São pareados com os métodos auxiliares showDialog e showCupertinoDialog, respectivamente para mostrar os diálogos.

AlertDialog

Começaremos com o AlertDialog. Primeiro vamos criar a estrutura básica do nosso aplicativo:

```
import 'package:flutter/material.dart';
void main(){
  runApp(new MyApp());
class MyApp extends StatelessWidget {
   @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Flutter AlertDialog',
      home: Scaffold(
        appBar: AppBar(
          title: Text("Exemplo de AlertDialog"),
        ),
        body: Home(),
      ),
      theme: ThemeData(primaryColor: Colors.orange,
      ) );  }}
```

Agora vamos criar a classe que receberá os comandos do AlertDialog, os comandos gerarão um botão no `body` que, ao ser pressionado, disparará o *pop-up* da caixa de diálogo.

```
class Home extends StatelessWidget{
  @override
  Widget build(BuildContext context){
    return Center(
      child: ElevatedButton(
        onPressed: (){
          showDialog(
              context: context,
              builder: (BuildContext context){
                return AlertDialog(
                  title: new Text("Parabéns"),
                  content: new Text("Você criou a sua primeira
caixa de diálogo usando o AlertDialog."),
                  actions: <Widget>[
                    new TextButton(
                        child: new Text("Fechar"),
                      onPressed: (){
                          Navigator.of(context).pop();
                      },
                    ),
                  ],
                );
              },
          );
        },
        child: Text("Clique Aqui"),
     ),);}}}
```

No resultado, devemos ter um botão “Clique Aqui” que acionará uma janela *pop-up* de alerta com o método `onPressed()` e posteriormente o showDialog, como mostra a Figura 4:

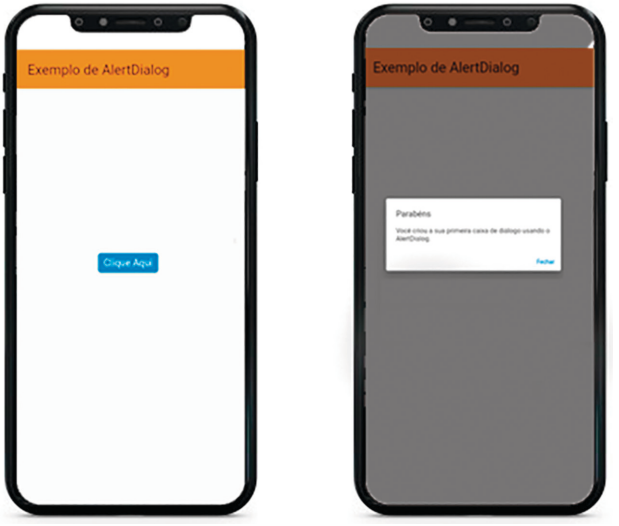

Figura 4. Exemplo de AlertDialog.

ShowDialog

O ShowDialog usa um *context* e um *builder* para retornar o diálogo específico. Basicamente, ele é usado para alterar a tela atual do nosso aplicativo para mostrar o *pop-up* da caixa de diálogo. O usuário sai da tela atual e vê uma nova tela. Usamos essa caixa de diálogo quando queremos mostrar uma guia que irá abrir qualquer tipo de caixa de diálogo, ou criamos uma guia frontal para mostrar o processo em segundo plano (SHARMA, 2021).

Para testarmos o `showDialog`, vamos criar uma caixa de diálogo usando uma `string` de inserção, ou seja, na caixa de diálogo será pedido para inserir um texto que, ao ser confirmado, aparecerá acima do botão. Para isso, digite o código:

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

Nesse exemplo, vamos definir um final, ou seja, uma atribuição a uma variável que não poderá ser alterada. Digite:

```
final Color darkBlue = Color.fromARGB(255, 18, 32, 47);
```

E continue com a estrutura básica:

```
void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData.dark().copyWith(scaffoldBackgroundCo
lor: darkBlue),
      debugShowCheckedModeBanner: false,
      home: Home(),
    );
  }
}
```

Crie agora a classe `Home` com uma *widget* do `Stateful`, pois é aqui que acontecerão as modificações:

```
class Home extends StatefulWidget {
 class Home extends StatefulWidget {
  @override
  _HomeState createState() => _HomeState();
}
class _HomeState extends State<Home> {
  TextEditingController _controller = TextEditingController();
  String inputString = "Definir Usuário";
```

Criamos uma *string* de inserção e vamos entrar com esse valor (“Definir usuário”). Poderíamos deixar vazio, mas sugiro colocar esse valor somente por didatismo; esse texto será modificado quando for inserido o texto requerido na caixa de diálogo. Vamos digitar o *widget* Scaffold, que é onde declararemos o nosso ShowDialog:

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(inputString),
            ElevatedButton(
              child: Text("Clique Aqui"),
              onPressed: () {
                showDialog(
                  context: context,
                  builder: (BuildContext context) {
                    return AlertDialog(
                      title: Text("Digite o nome do usuário"),
                      content: TextFormField(
                        controller: _controller,
                      ),
                      actions: <Widget>[
                        TextButton(
                          child: Text("OK"),
                          onPressed: () {
                            Navigator.pop(context, _controller.
text);
                          },
                        )
                      ],
                    );
                  },
                ).then((val) {
                  setState(() {
                    inputString = val;
                  });
```

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

Veja, na Figura 5, como ficou a nossa implementação.

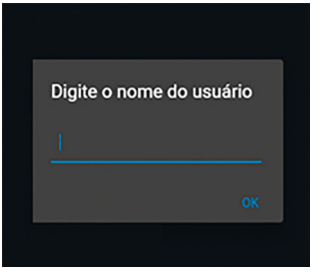

Figura 5. Exemplo de ShowDialog no DartPad.

Fique atento

Sempre que aparecer um erro `Deprecated` em seu código, é porque algum comando foi descontinuado em uma nova versão de alguma linguagem de programação. O `RaiseButton` e o `FlatButton`, por exemplo, são componentes descontinuados do Dart nas novas versões com o Flutter; para esses comandos, a solução agora é `ElevatedButton` e `FlateButton`, respectivamente.

Widgets de *layouts* para coleções

Nesta seção, exploraremos alguns tipos de *widgets* para o posicionamento de componentes na tela de IU que possuem vários filhos. Por exemplo, o *widget* Row permite a disposição de seus filhos na direção horizontal, enquanto o *widget* Column permite a disposição de seus filhos na direção vertical. Ao compor Row e Column, é possível criar um *widget* com qualquer nível de complexidade. No entanto, existem *widgets* que permitem organizar filhos como uma lista e outros por meio de galerias (grades).

Esse tipo de *widget* para organização dos componentes na tela permite aplicativos com *layouts* avançados; para ilustrarmos esse comportamento, vamos conhecer o *widget* ListView e o *widget* GridView.

ListView

O ListView é um *widget* básico de listas roláveis. São usados construtores para a sua criação, o padrão é o List<Widget>, mas também é possível usar o ListView.builder, ListView.separated e o ListView.custom.

Para começar, vamos implementar uma pequena aplicação que gerará uma lista simples usando o construtor ListView<Widget>. Esse tipo de construtor funciona bem para listas simples, porém, em uma lista com um número maior de filhos, é aconselhável que se use o ListView.builder.

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Lista Básica';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: Text(title),
        ),
        body: ListView(
          children: <Widget>[
            ListTile(
              leading: Icon(Icons.map),
              title: Text('Mapa'),
            ),
            ListTile(
              leading: Icon(Icons.photo_album),
              title: Text('Álbum'),
            ),
```

```
            ListTile(
              leading: Icon(Icons.phone),
              title: Text('Fone'),
            ),
          ],
        ),
      ),
    );
  }
}
```

Como é uma implementação relativamente simples, sugiro que a faça usando o Dartpad. Veja o resultado dessa aplicação na Figura 6.

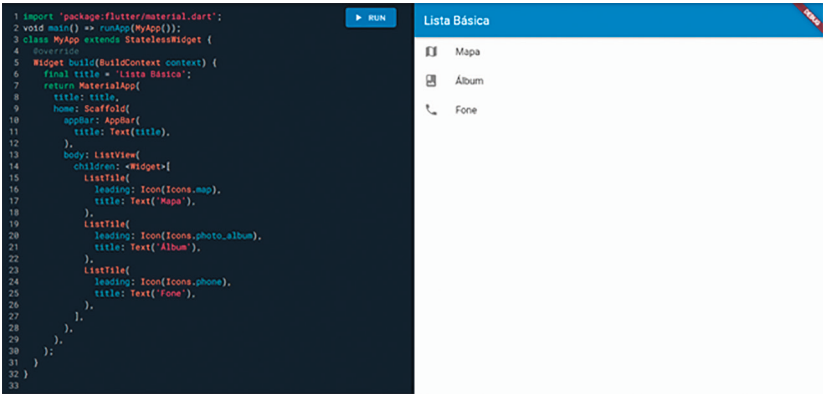

Figura 6. Exemplo de ListView.

GridView

Caso você necessite criar um *layout* complexo com várias linhas e colunas, a melhor maneira de fazer isso é usando o *widget* GridView. Para isso, é usado um construtor de contagem que possibilita ao *designer* definir quantos itens quer na grade com a propriedade `crossAxisCount`. Com a `crossAxisSpacing`, define-se os espaços entre as colunas, e com `mainAxisSpacing`, os espaços entre as linhas.

Vejamos agora um exemplo de como alterar a grade de *layout* da sua aplicação usando o *widget* GridView. Vamos começar com a estrutura e depois vamos acrescentar ícones da biblioteca do Flutter. Para isso, digite o código a seguir no arquivo main.dart do seu projeto:

```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Gerando uma grade de itens';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: Text(title),
        ),
```

Vamos agora compor o *body* com o *widget* GridView. Vamos defini-lo com três colunas e a possibilidade de até 10 *widgets* para inserção de conteúdo, ou seja, para exibição do `index`. Continue:

```
        body: GridView.count(
          crossAxisCount: 3,
          children: List.generate(10, (index) {
            return Center(
              child: Text(
                'Item $index',
                style: Theme.of(context).textTheme.headline5,
              ); }), ),), ); }}
```

Veja, na Figura 7, como deve ficar o seu *layout* com grade.

Figura 7. Exemplo de GridView simples.

Vamos fazer inserção de ícones no índice da sua grade, para isso usaremos ícones da biblioteca do Flutter. Altere o código do `body` da estrutura anterior para:

```
body: GridView.count(
          // Cria um grid com duas colunas
            crossAxisCount: 3,
           children: List.generate(opcoes.length, (index) {
             return Center(
               child: OpcaoCard(opcao: opcoes[index]),
             );} ) ) ) );}}
```

Nessa nova estrutura, o `index` retornará uma lista que será alimentada por uma classe que criaremos com o nome `OpcaoCard`. Primeiro vamos criar uma classe para definirmos os elementos da lista: títulos e imagens, que terão valores `final`. Para isso, digite:

```
class Opcao {
  const Opcao({required this.titulo, required this.icon});
  final String titulo;
  final IconData icon;
}
```

```
const List<Opcao> opcoes = const <Opcao>[
  const Opcao(titulo: 'Carro', icon: Icons.directions_car),
  const Opcao(titulo: 'Bike', icon: Icons.directions_bike),
  const Opcao(titulo: 'Barco', icon: Icons.directions_boat),
  const Opcao(titulo: 'Ônibux', icon: Icons.directions_bus),
  const Opcao(titulo: 'Trem', icon: Icons.directions_railway),
  const Opcao(titulo: 'Andar', icon: Icons.directions_walk),
  const Opcao(titulo: 'Carro', icon: Icons.directions_car),
  const Opcao(titulo: 'Bike', icon: Icons.drafts),
  const Opcao(titulo: 'Barco', icon: Icons.dvr),
  const Opcao(titulo: 'Copy', icon: Icons.copyright),
  const Opcao(titulo: 'Train', icon: Icons.cloud_off),
  const Opcao(titulo: 'Car', icon: Icons.directions_car),
  const Opcao(titulo: 'Bike', icon: Icons.directions_bike),
  const Opcao(titulo: 'Barco', icon: Icons.directions_boat),
  const Opcao(titulo: 'Ônibus', icon: Icons.directions_bus),
  const Opcao(titulo: 'Trem', icon: Icons.directions_railway),
  const Opcao(titulo: 'Andar', icon: Icons.directions_walk),
  const Opcao(titulo: 'Carro', icon: Icons.directions_car),
  const Opcao(titulo: 'Bike', icon: Icons.drafts),
  const Opcao(titulo: 'Barco', icon: Icons.dvr),
];
```

Agora sim geraremos a classe `OpcaoCard`:

```
class OpcaoCard extends StatelessWidget {
  const OpcaoCard({Key? key, required this.opcao}) :
super(key: key);
  final Opcao opcao;
  @override
  Widget build(BuildContext context) {
    final TextStyle? textStyle = Theme.of(context)
        .textTheme.headline4;
```

```
    return Card(
        color: Colors.white,
        child: Center(
          child: Column(
              mainAxisSize: MainAxisSize.min,
              crossAxisAlignment: CrossAxisAlignment.center,
              children: <Widget>[
                Icon(opcao.icon, size:80.0),
                Text(opcao.titulo, style: textStyle),
              ]
          ),
        )
    );
  }
}
```

E temos a nossa GridView, como mostra a Figura 8.

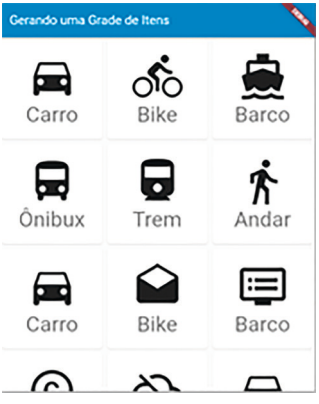

Figura 8. Exemplo de GridView com ícones.

Exemplo

Recursos como o *widget* GridView são muito usados em aplicativos de imagens, como aqueles que precisam de uma estrutura de grades para organizar uma lista de imagens — uma galeria de fotos, por exemplo. O *widget* GridView, portanto, é um *array* bidimensional de *widgets* roláveis.

Neste capítulo, fomos introduzidos ao universo de desenvolvimento Mobile pelo *framework* Flutter. Vimos que todo o desenvolvimento com o Flutter é baseado em um conjunto de *widgets* que determinam os elementos visuais da interface da aplicação. Dentre esses componentes, vimos o AppBar, o Drawer, o BootonNavigationBar e também os *widgets* para implementação de caixas de diálogo (AlertDialog e SimpleDialog). Para finalizar, usamos alguns componentes para manipular o ListView e o GridView. Abordamos com foco no desenvolvimento prático, a fim de que você consiga criar as suas primeiras aplicações Mobile e fique motivado a se aprofundar cada vez mais nesse *framework* que ganha cada vez mais força no mercado de desenvolvimento de *softwares*.

Referências

FERNANDO, I. Criação de caixas de diálogo no Flutter. *BR Atsit*, Bucareste, 14 jun. 2021. Disponível em: https://br.atsit.in/archives/57444. Acesso em: 6 jul. 2021.

FLUTTER architectural overview. *Flutter*, [*S. l.*], [2021]. Disponível em: https://flutter.dev/docs/resources/architectural-overview. Acesso em: Acesso em: 6 jul. 2021.

FLUTTER documentation. *Flutter*, [*S. l.*], 2020. Disponível em: https://flutter.dev/docs. Acesso em: 6 jul. 2021.

KHAN, S. Flutter Bottom Navigation Bar Android e IOS. 2019. *Medium*, [*S. l.*], 6 June 2019. Disponível em: https://medium.com/@sarimk80/flutter-bottom-navigation-bar-android-and-ios-8971ab7d92a3. Acesso em: 6 jul. 2021.

PINHEIRO, F. Flutter: O que são widgets e qual sua importância. *TreinaWeb*, São Paulo, 2020. Disponível em: https://www.treinaweb.com.br/blog/flutter-o-que-sao-widgets-e-qual-sua-importancia. Acesso em: 6 jul. 2021.

SHARMA, N. Flutter – Dialog. *GeeksforGeeks*, Noida, 15 Feb. 2021. Disponível em: https://www.geeksforgeeks.org/flutter-dialogs/. Acesso em: Acesso em: 6 jul. 2021

Leituras recomendadas

BRACHA, G. *The Dart programming language*. Boston: Addison-Wesley Professional, 2016. 201 p.

WINDMILL, E. *Flutter in action*. Shelter Island: Manning, 2020. 368 p.

ZAMMETTI, F. *Flutter na prática*: melhore seu desenvolvimento mobile com o SDK open source mais recente do Google. São Paulo: Novatec, 2020. 368 p.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS