PRIMEIRO PROGRAMA USANDO FLUTTER

Introdução - Flutter é um framework de código aberto desenvolvido pelo Google para a criação de aplicativos móveis nativos para Android e iOS a partir de um único código-base. Ele utiliza a linguagem de programação Dart, que é fácil de aprender e possui um rico conjunto de bibliotecas.

Com o Flutter, é possível criar interfaces de usuário incríveis, altamente customizáveis e responsivas. Ele oferece uma grande variedade de widgets, que são os blocos de construção fundamentais para construir a interface do usuário.

1- **Crie um novo projeto** Flutter com o seguinte comando no terminal:

flutter create nome_do_programa

2- Abra o Visual Code ou outro editor que seu preferência e selecione a pasta onde foi criado.

Widgets: Em Flutter, tudo é um widget. Widgets são objetos responsáveis por construir a interface do usuário e definir como ela deve se comportar. Existem dois tipos de widgets: Stateless e Stateful.

- **Widgets Stateless**: São widgets que não mudam de estado (imutáveis) durante a execução do aplicativo. Eles são usados para exibir conteúdo que não precisa ser atualizado ou alterado, como textos, imagens ou ícones.
- **Widgets Stateful:** São widgets que podem mudar de estado (mutáveis) durante a execução do aplicativo. Eles são usados para criar componentes interativos, como botões ou formulários, que podem ser atualizados com base em interações do usuário.

Usando widgets para criar diferentes tipos de interfaces de usuário: Flutter possui uma vasta coleção de widgets pré-construídos para criar interfaces de usuário comuns, como botões, listas, barras de navegação, caixas de diálogo, entre outros. Além disso, você pode criar seus próprios widgets personalizados para atender às suas necessidades específicas.

Ao combinar diferentes widgets, você pode criar interfaces de usuário complexas e interativas. Por exemplo, usando widgets de layout como Row, Column, Container e Expanded, você pode criar layouts flexíveis e responsivos.

Estilizando suas interfaces de usuário: Flutter oferece suporte a personalização de interfaces de usuário por meio de estilos e temas. Você pode estilizar widgets usando propriedades como cores, tamanhos de fonte,

espaçamento e bordas. Além disso, você pode criar temas globais para aplicar estilos consistentes em todo o aplicativo.

Através de recursos como gradientes, sombras, bordas arredondadas e animações, pode-se adicionar a aparência de um aplicativo, tornando-o visualmente atraente e profissional.

Neste projeto, vamos criar um aplicativo móvel simples usando o framework Flutter. O aplicativo será chamado de "HelloWord" e terá como objetivo exibir a mensagem "Hello, Flutter em sua tela inicial.

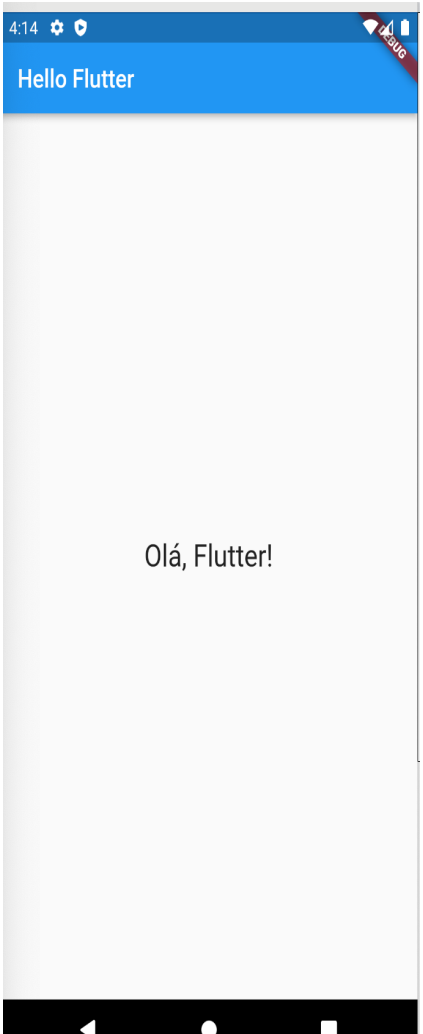
```dart
import 'package:flutter/material.dart';

void main() {
  runApp(HelloWordApp());
}

class HelloWordApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Hello Flutter'),
        ),
        body: Center(
          child: Text(
            'Olá, Flutter!',
            style: TextStyle(fontSize: 24),
          ),
        ),
      ),
    );
  }
}
```

Abaixo segue o código comentado:

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

// A função main() é o ponto de entrada do aplicativo Flutter.
// Ela chama a função runApp(), passando a instância do HelloWordApp como argumento,
// para iniciar a execução do aplicativo.

void main() {
  runApp(HelloWordApp());
}

// A classe MelloWordApp é um widget que herda de StatelessWidget.
// StatelessWidget é usado quando o conteúdo do widget não muda durante a execução do aplicativo.
class HelloWordApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // O método build() é obrigatório em um widget StatelessWidget e retorna
    // a interface gráfica do widget.
```

```
    // MaterialApp é um widget que configura a estrutura básica do aplicativo.
    // Ele define temas, roteamento, localização, entre outras configurações.
    return MaterialApp(
      // O widget MaterialApp possui uma propriedade chamada 'home',
      // que define o widget que será exibido como tela inicial do aplicativo.
      home: Scaffold(
        // Scaffold é um widget que fornece uma estrutura básica de layout de
        // Material Design, como AppBar, FloatingActionButton, etc.

        // AppBar é uma barra de aplicativo que normalmente fica na parte superior da tela.
        appBar: AppBar(
          // O título da AppBar é definido como 'HelloWord'.
          title: Text('MelloWord'),
        ),

        // O corpo do Scaffold é definido através da propriedade 'body'.
        body: Center(
          // O widget Center centraliza seu único filho na tela.
          child: Text(
            // Text é um widget usado para exibir texto na interface do usuário.
            // O texto exibido é definido como 'Hello, Hello!'.
            'Hello, Hello!',
            // O estilo do texto é definido através da propriedade 'style'.
            // Aqui, estamos definindo o tamanho da fonte como 24.
            style: TextStyle(fontSize: 24),
          ),
        ),
      ),
    );
  }
}
```

O Seguinte código apresentara o resultado conforme a figura abaixo :



Atividade -

Criando uma Lista de Saudações

Objetivo: Praticar a criação de interfaces de usuário simples usando o Flutter.

Instruções:

-Crie um novo projeto Flutter ou use o projeto hello flutter
-Modifique o aplicativo existente para exibir uma lista de saudações em vez de apenas uma saudação.
-Crie uma lista de saudações, como por exemplo: "Olá, Flutter!", "Bem-vindo ao Flutter!", "Saudações de Flutter!", etc.
-Use um widget ListView ou Column para exibir as saudações na tela.
-Estilize as saudações para que sejam exibidas de forma agradável, como aumentar o tamanho da fonte, alterar as cores, adicionar margens, etc.
(Desafio) Adicione um ícone ao lado de cada saudação na lista, como um ícone de cumprimento ou um ícone do Flutter.

Exemplo dessa atividade

```
import 'package:flutter/material.dart';
void main() {
 runApp(HelloWordApp());
}
class HelloWordApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     home: Scaffold(
       appBar: AppBar(
         title: Text('Hello Flutter'),
       ),
       body: Center(

         // Aqui estamos usando a classe GreetingsList para exibir nossa lista de saudações.

         child: GreetingsList(),
       ),
     ),
   );
 }
}

class GreetingsList extends StatelessWidget {

 // Lista de saudações que serão exibidas.

 final List<String> greetings = [
  'Olá, Flutter!',
```

```
  'Bem-vindo ao Flutter!',
  'Flutter é incrível!'
  'Estou aprendendo Flutter!',
 'Muito bom!!',
];

@override
Widget build(BuildContext context) {
  return ListView.builder(
 // Definimos o número de itens na lista como o tamanho da lista de saudações.
    itemCount: greetings.length,
    // O itemBuilder é uma função que constrói cada item na lista.
    itemBuilder: (context, index) {
      return ListTile(
        // Um ícone de estrela à esquerda de cada saudação.
        leading: Icon(Icons.star),
        // O texto da saudação.
        title: Text(greetings[index]),
      );
    },
  );
 }
}
```

Esses são os principais tópicos que será mostrado "**Criando interfaces de usuário com Flutter**". Dominar a criação de interfaces de usuário é fundamental para o desenvolvimento de aplicativos eficientes e atrativos.

Outro exemplo de código para criar uma mensagem e através de um botão chamar outra tela.

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

void main() {
  runApp(MeuApp());
}

class MeuApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {34
    return MaterialApp(
      home: PaginaInicial(),
    );
  }
}

class PaginaInicial extends StatefulWidget {
  @override
  _PaginaInicialState createState() => _PaginaInicialState();
}

class _PaginaInicialState extends State<PaginaInicial> {
  String textoExibido = 'Olá, mundo!';
```

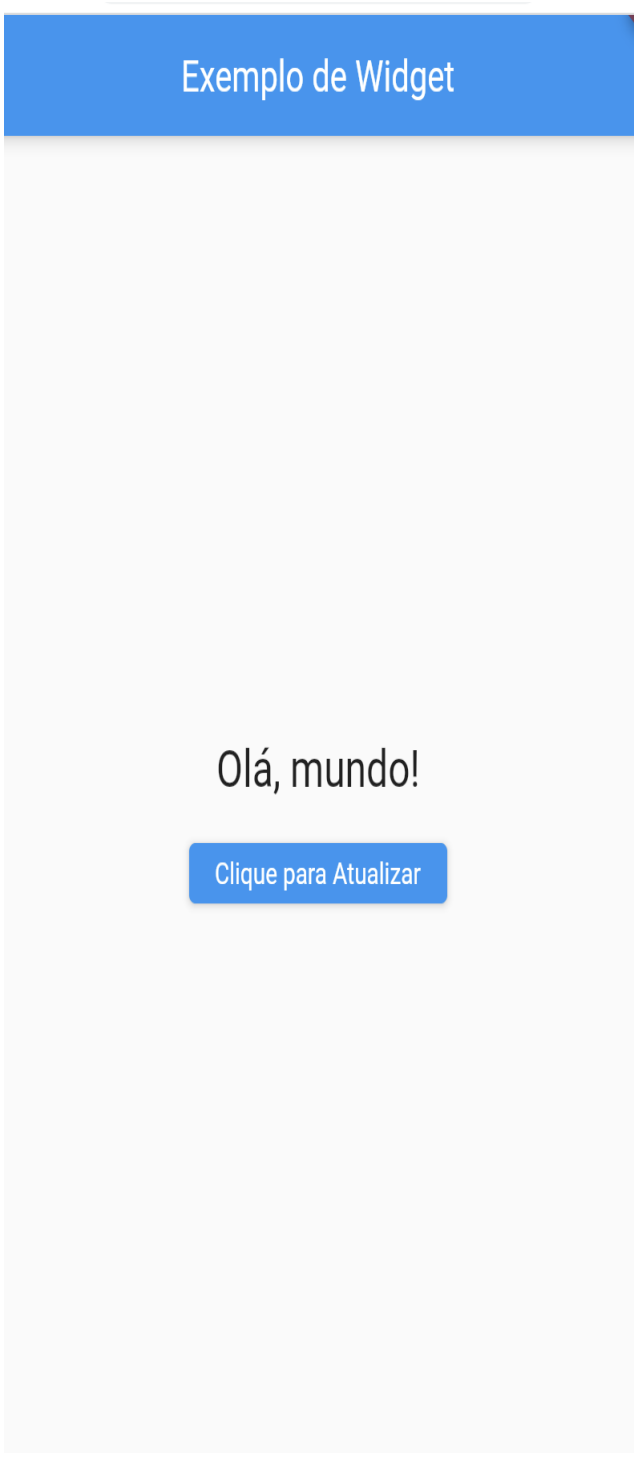
```
  void _atualizarTexto() {
    setState(() {
      textoExibido = 'Texto atualizado!';
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Exemplo de Widget'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(
              textoExibido,
              style: TextStyle(fontSize: 24),
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: _atualizarTexto,
              child: Text('Clique para Atualizar'),
            ),
          ],
        ),
      ),
    );
  }
}
```

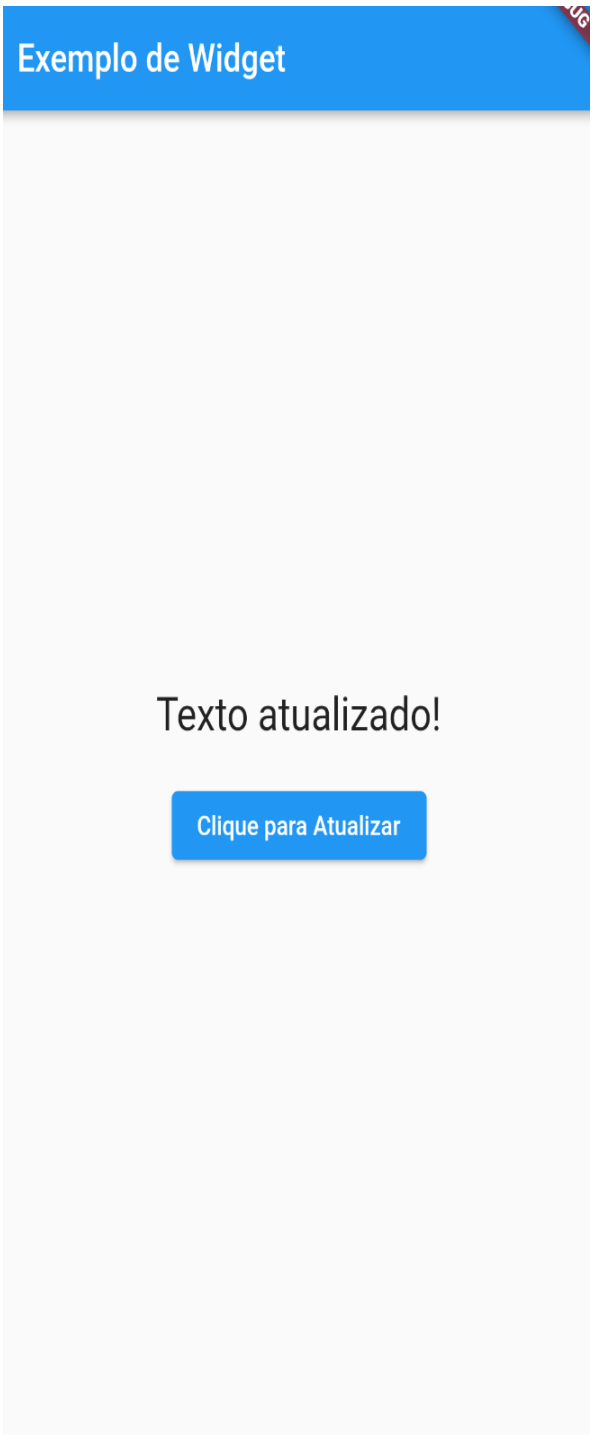
Explicação do código:

1. Primeiro, foi importado o pacote flutter/material.dart, que contém os widgets e recursos básicos do Flutter.
2. Em seguida, criado um novo widget chamado MeuApp, que é um StatelessWidget. Ele será o ponto de partida do aplicativo e retornará uma MaterialApp que define a interface do aplicativo.
3. O MaterialApp possui uma PaginaInicial como página inicial, definida pela propriedade home.
4. A classe PaginaInicial é um StatefulWidget porque precisaremos atualizar o texto exibido na tela. O estado desse widget é mantido em _PaginaInicialState.
5. O estado _PaginaInicialState possui uma variável textoExibido para armazenar o texto a ser exibido na tela. Inicialmente, o texto é definido como "Olá, mundo!".
6. O método _atualizarTexto é chamado quando o botão é pressionado e atualiza o texto para "Texto atualizado!".

7. No método build, construímos a interface do usuário. Foi usado um Scaffold como estrutura principal, que possui um AppBar e o body.
8. O body consiste em um Center que contém uma Column. A Column contém um Text para exibir o texto atualizado e um ElevatedButton que, quando pressionado, chama _atualizarTexto.

Este exemplo é bastante simples, mas ilustra como você pode usar widgets para criar interfaces de usuário e atualizá-las com base em eventos. Conforme você avança, poderá explorar mais widgets e recursos do Flutter para criar interfaces de usuário mais complexas e atraentes.

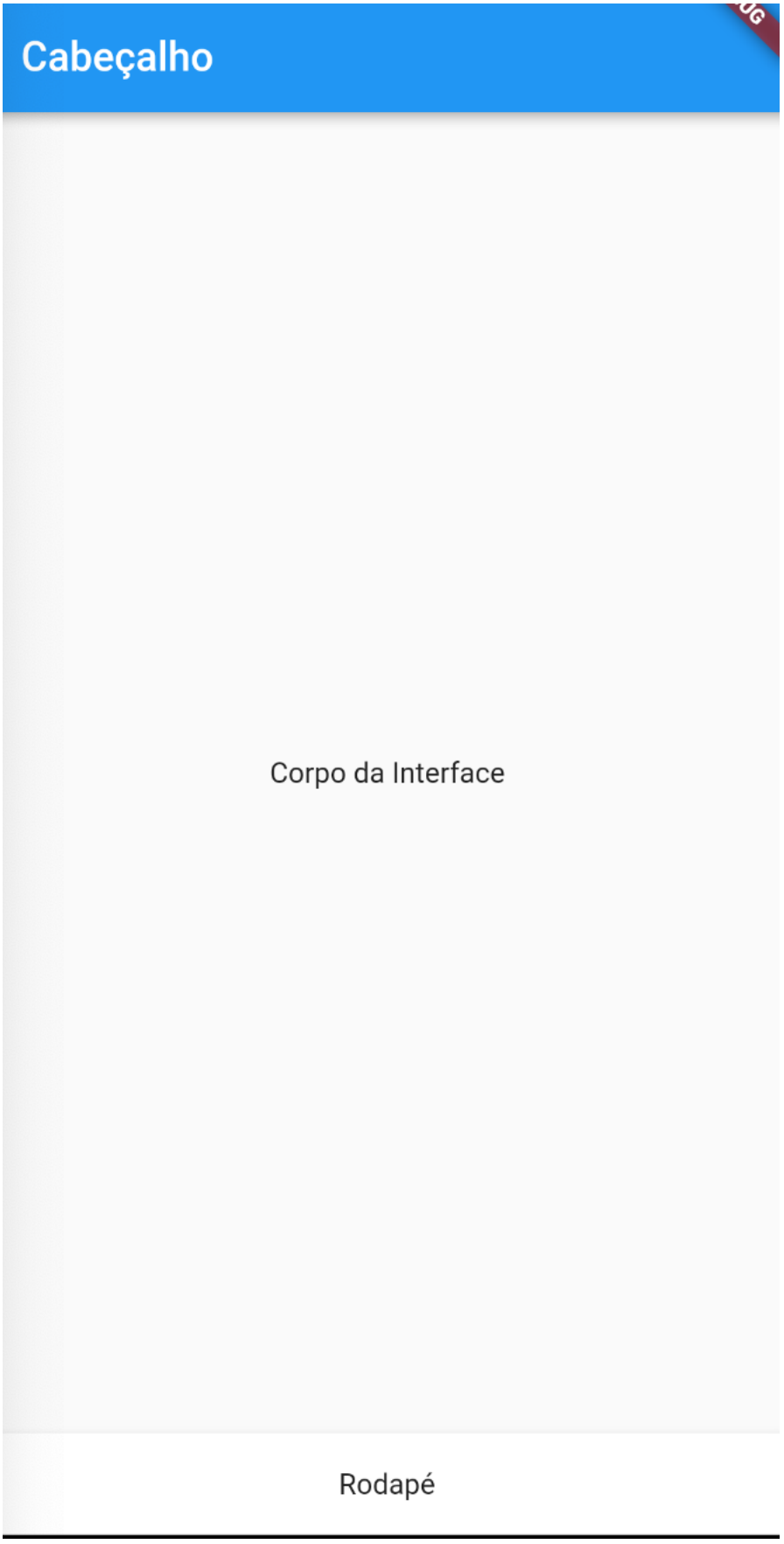




Exercícios

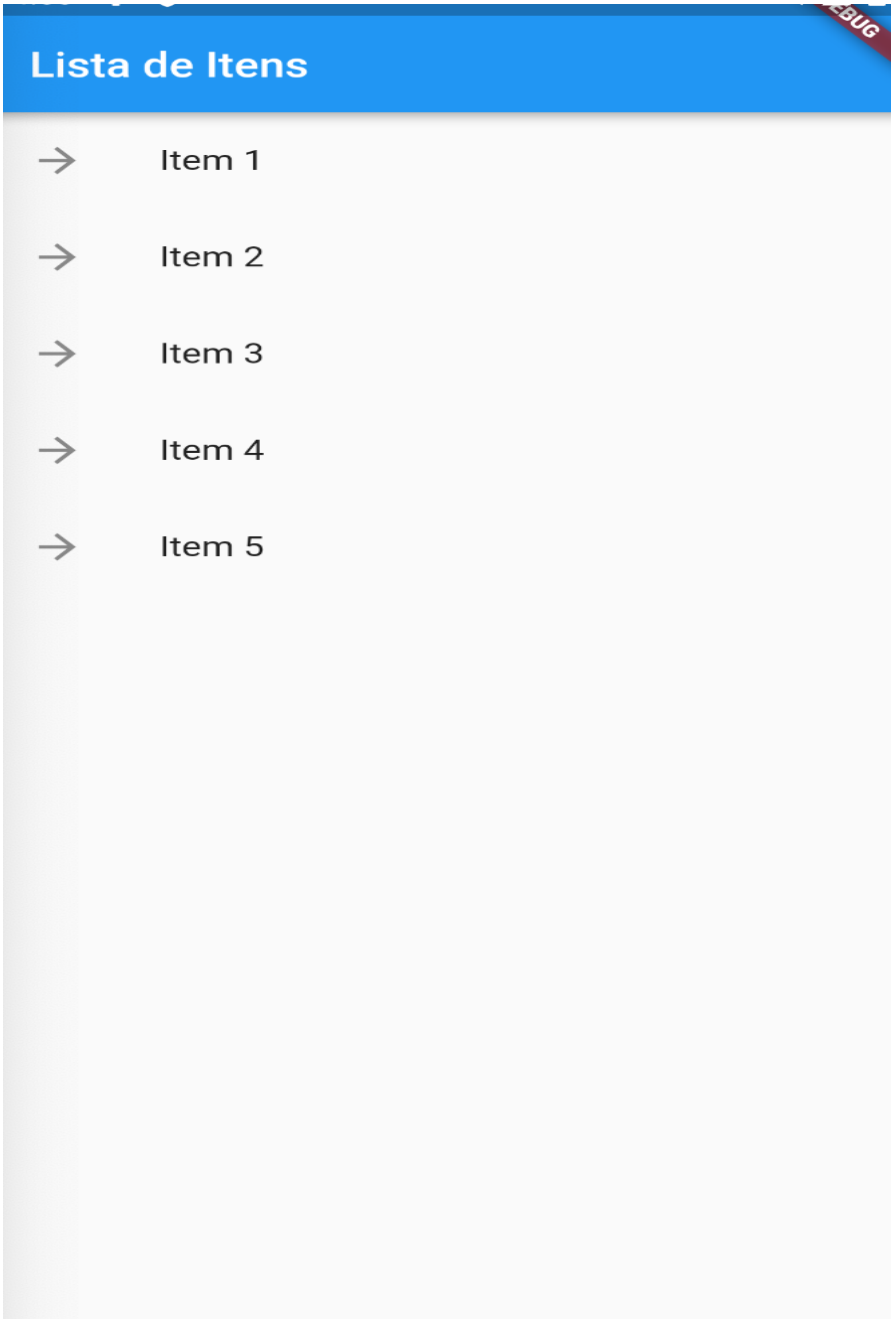
Exercício 1 - Interface de usuário com cabeçalho, corpo e rodapé: Neste exercício, será uma interface de usuário simples com um cabeçalho, um corpo e um rodapé. Vamos usar o Scaffold como estrutura principal para organizar esses elementos.

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

void main() {
  runApp(MeuApp());
}

class MeuApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: PaginaInicial(),
    );
  }
}

class PaginaInicial extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Cabeçalho'),
      ),
      body: Center(
        child: Text('Corpo da Interface'),
      ),
      bottomNavigationBar: BottomAppBar(
        child: Container(
          height: 50,
          child: Center(
            child: Text('Rodapé'),
          ),
        ),
      ),
    );
  }
}
```


Cabeçalho
Corpo da Interface
Rodapé

Exercício 2 - Interface de usuário com uma lista de itens: Neste exercício, criaremos uma interface de usuário com uma lista de itens. Usaremos o widget ListView.builder para criar a lista dinamicamente com base nos dados fornecidos.

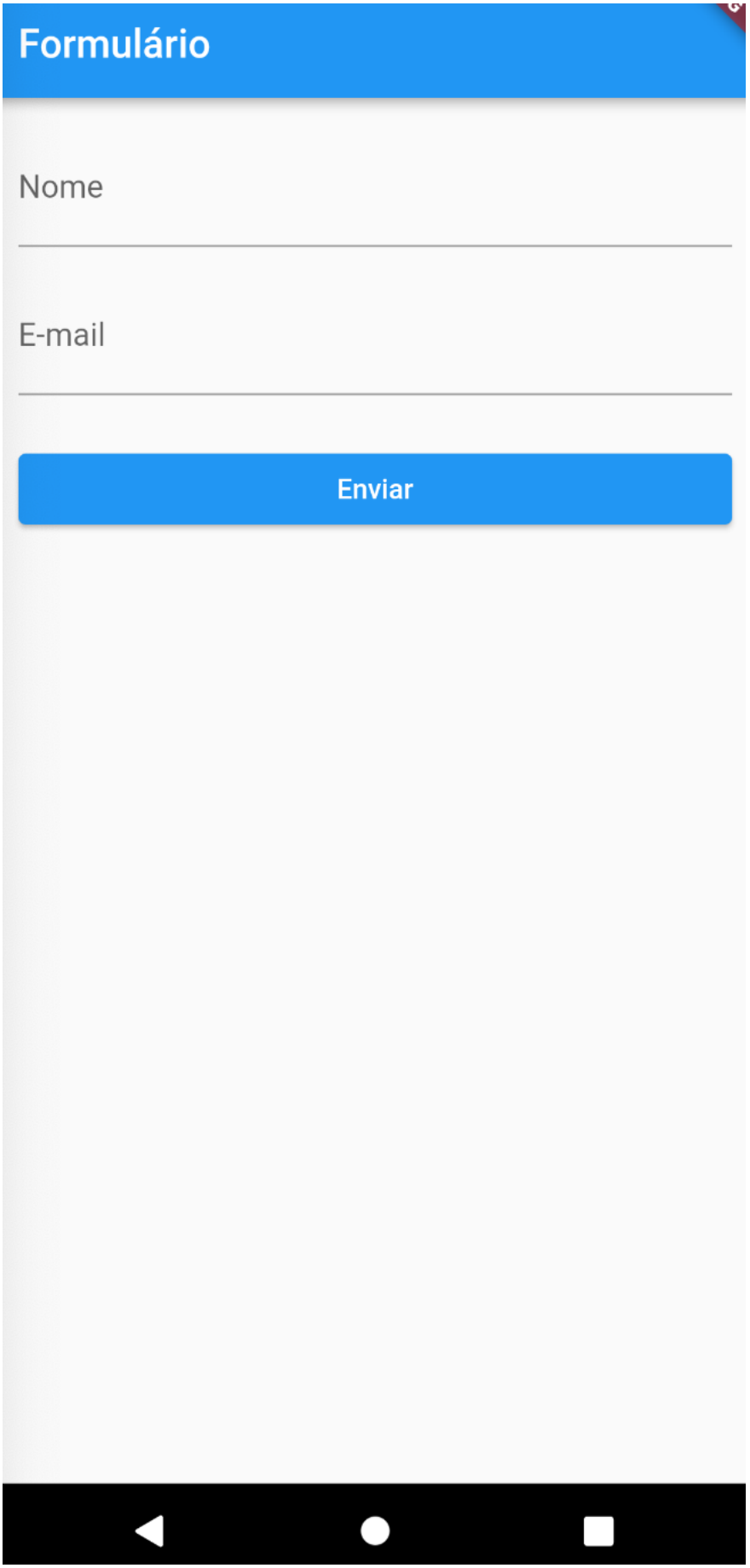
```
import 'package:flutter/material.dart';

void main() {
  runApp(MeuApp());
}

class MeuApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: PaginaInicial(),
    );
  }
}

class PaginaInicial extends StatelessWidget {
  final List<String> itens = [
    'Item 1',
    'Item 2',
    'Item 3',
    'Item 4',
    'Item 5',
  ];

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Lista de Itens'),
      ),
      body: ListView.builder(
        itemCount: itens.length,
        itemBuilder: (context, index) {
          return ListTile(
            title: Text(itens[index]),
            leading: Icon(Icons.arrow_forward),
          );
        },
      ),
    );
  }
}
```



Exercício 3 - Interface de usuário com um formulário: Neste exercício, criaremos uma interface de usuário com um formulário que permite ao usuário inserir texto em campos de entrada.

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

void main() {
  runApp(MeuApp());
}

class MeuApp extends StatelessWidget {
```

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

class PaginaInicial extends StatelessWidget {
  final TextEditingController _nomeController = TextEditingController();
  final TextEditingController _emailController = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Formulário'),
      ),
      body: Padding(
        padding: EdgeInsets.all(16.0),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.stretch,
          children: [
            TextFormField(
              controller: _nomeController,
              decoration: InputDecoration(labelText: 'Nome'),
            ),
            SizedBox(height: 16),
            TextFormField(
              controller: _emailController,
              decoration: InputDecoration(labelText: 'E-mail'),
            ),
            SizedBox(height: 24),
            ElevatedButton(
              onPressed: () {
                // Lógica para processar o formulário
                String nome = _nomeController.text;
                String email = _emailController.text;
                print('Nome: $nome, E-mail: $email');
              },
              child: Text('Enviar'),
            ),
          ],
        ),
      ),
    );
  }
}
```


Formulário
Nome
E-mail
Enviar

Exercício 4

Criar uma calculadora sempre com operações básica Multiplicação, Divisão, Soma e Subtração.

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

void main() {
  runApp(CalculadoraApp());
}

class CalculadoraApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Calculadora(),
    );
  }
}

class Calculadora extends StatefulWidget {
  @override
  _CalculadoraState createState() => _CalculadoraState();
}

class _CalculadoraState extends State<Calculadora> {
  String _output = "0"; // Inicializa o valor de saída da calculadora.

  // Função para atualizar o valor de saída da calculadora.
  void _updateOutput(String value) {
    setState(() {
      if (_output == "0") {
        _output = value;
      } else {
        _output += value;
      }
    });
  }

  // Função para realizar o cálculo final e exibir o resultado.
  void _calculateResult() {
    Parser p = Parser();
    Expression exp = p.parse(_output);
    ContextModel cm = ContextModel();
    double result = exp.evaluate(EvaluationType.REAL, cm);
```

```
  setState(() {
    _output = result.toString();
  });
}

// Função para limpar o valor de saída e reiniciar os valores.
void _clearOutput() {
  setState(() {
    _output = "0";
  });
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Calculadora Simples'),
    ),
    body: Column(
      crossAxisAlignment: CrossAxisAlignment.stretch,
      children: [
        Expanded(
          child: Container(
            padding: EdgeInsets.all(16.0),
            color: Colors.grey[200],
            child: Align(
              alignment: Alignment.bottomRight,
              child: Text(
                _output,
                style: TextStyle(fontSize: 40.0, fontWeight: FontWeight.bold),
              ),
            ),
          ),
        ),
        buildButtonRow(["7", "8", "9", "/"]),
        buildButtonRow(["4", "5", "6", "*"]),
        buildButtonRow(["1", "2", "3", "-"]),
        buildButtonRow(["0", ".", "+"]),
        buildButtonRow(["C", "="]),
      ],
    ),
  );
}

// Função auxiliar para criar uma linha de botões com os valores fornecidos.
Widget buildButtonRow(List<String> values) {
  return Expanded(
    child: Row(
      crossAxisAlignment: CrossAxisAlignment.stretch,
```

```
        children: values.map((value) {
          return Expanded(
            child: InkWell(
              onTap: () {
                if (value == "=") {
                  _calculateResult();
                } else if (value == "C") {
                  _clearOutput();
                } else {
                  _updateOutput(value);
                }
              },
              child: Container(
                color: Colors.grey[300],
                child: Center(
                  child: Text(
                    value,
                    style: TextStyle(fontSize: 32.0),
                  ),
                ),
              ),
            ),
          );
        }).toList(),
      ),
    );
  }
}
```

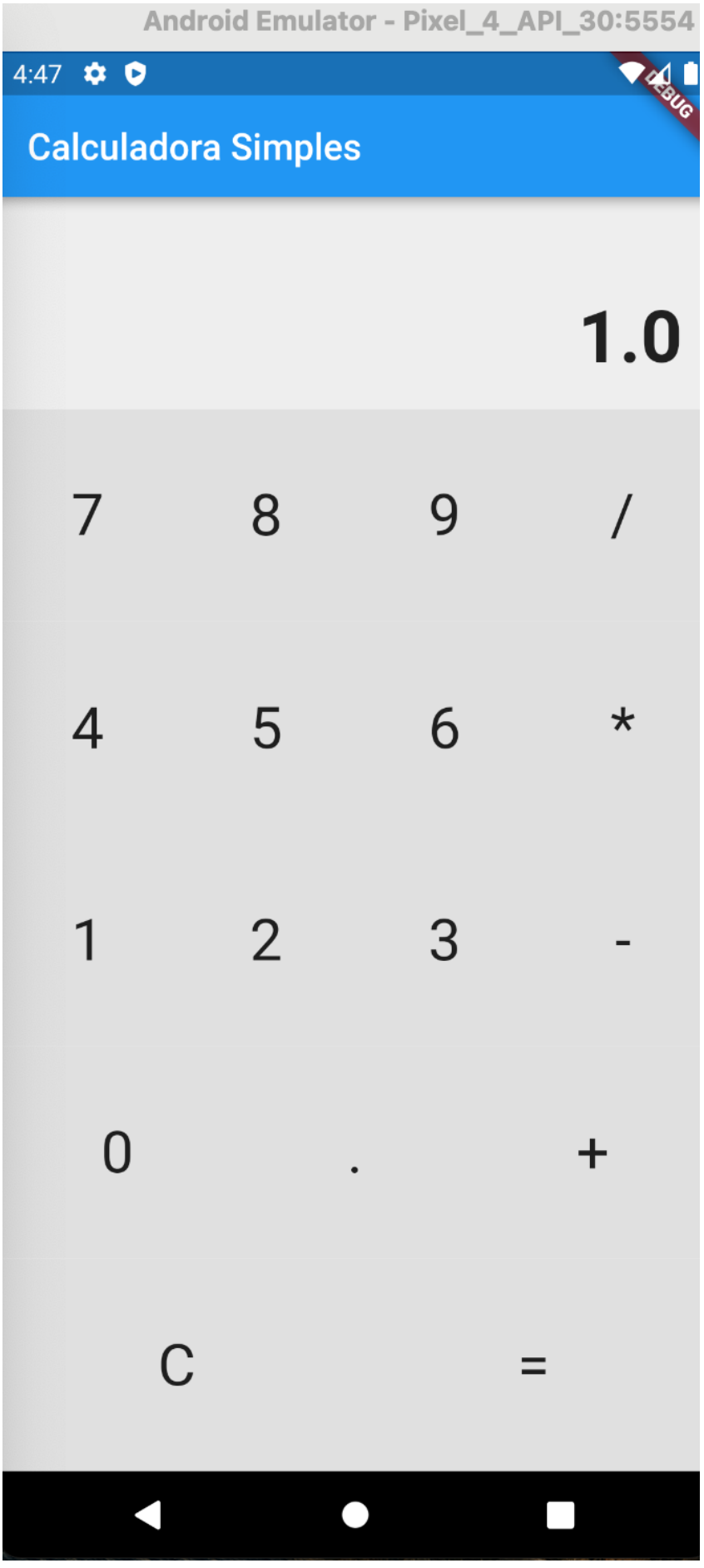
Passo 1: antes de executar o código abra o arquivo pubspec.yaml do seu projeto Flutter.

Passo 2: Adicione a dependência do pacote http abaixo de dependencies no arquivo pubspec.yaml:

yamlCopy code
dependencies: flutter: sdk: math_expressions: ^2.0.2

Passo 3: Salve o arquivo pubspec.yaml.

Passo 4: Após salvar o arquivo, execute o comando flutter pub get no terminal para que o Flutter baixe e instale as dependências do pacote Math.


Android Emulator - Pixel_4_API_30:5554
4:47
DEBUG
Calculadora Simples
1.0
7
8
9
/
4
5
6
*
1
2
3
-
0
.
+
C
=

TEMA "Trabalhando com dados e estado" em Flutter. Vamos aprender como gerenciar o estado dos widgets e como lidar com dados em um aplicativo Flutter.

Conteúdo da aula:

1. Introdução ao estado em Flutter
2. Stateful vs. Stateless Widgets
3. Gerenciando o estado com StatefulWidget
4. Atualizando o estado com setState()
5. Trabalhando com dados em um aplicativo Flutter
6. Exemplo prático: Lista de Tarefas

1. Introdução ao estado em Flutter: O estado em um aplicativo Flutter refere-se a qualquer informação que pode mudar durante a execução do aplicativo. Por exemplo, quando você tem um contador que aumenta ou diminui, um campo de entrada de texto que pode ser alterado ou uma lista de itens que pode ser atualizada, você está lidando com o estado.

2. Stateful vs. Stateless Widgets: Em Flutter, existem dois tipos principais de widgets: Stateless e Stateful.

- Stateless Widgets: São widgets imutáveis, ou seja, uma vez criados, não podem ser alterados. Eles são usados para exibir informações que não mudam durante a execução do aplicativo.
- Stateful Widgets: São widgets mutáveis, ou seja, podem ser alterados durante a execução do aplicativo. Eles são usados para criar componentes interativos, onde o estado pode ser atualizado em resposta a interações do usuário ou outras alterações.

3. Gerenciando o estado com StatefulWidget: Para gerenciar o estado em um aplicativo Flutter, usamos o widget StatefulWidget. Um StatefulWidget é composto por duas classes: a classe StatefulWidget que é imutável e a classe State que é mutável e armazena o estado do widget.

4. Atualizando o estado com setState(): Quando o estado de um StatefulWidget precisa ser atualizado, usamos o método setState(). Esse método notifica o framework Flutter de que o estado foi alterado e solicita uma reconstrução do widget.

5. Trabalhando com dados em um aplicativo Flutter: Para trabalhar com dados em um aplicativo Flutter, podemos usar diferentes fontes de dados, como APIs, bancos de dados locais ou dados estáticos. Os dados podem ser exibidos em widgets usando a interpolação de strings ou como parte de listas, grades e outros widgets.

6. Exemplo prático: Lista de Tarefas: Vamos criar um exemplo prático de um aplicativo simples de Lista de Tarefas. Nesse exemplo, usaremos um

StatefulWidget para gerenciar o estado da lista de tarefas. O usuário poderá adicionar tarefas, marcar como concluídas e excluir tarefas.

Nesse exemplo, aprenderemos a usar o estado, atualizar o estado com setState() e como trabalhar com uma lista de dados em Flutter.

Recursos adicionais:

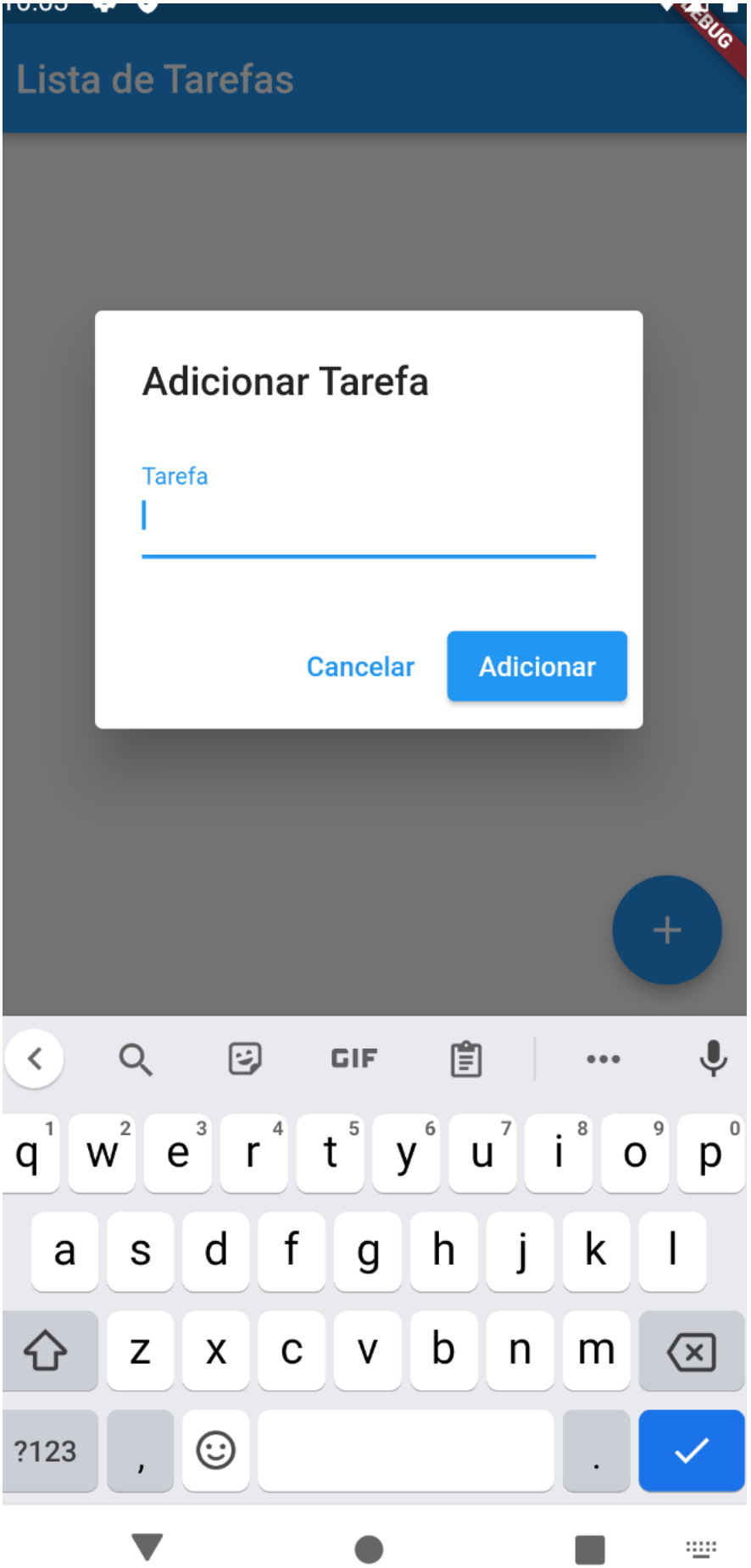
- Consumindo APIs para obter dados externos.
- Armazenando dados localmente com o SQLite ou o SharedPreferences.
- Utilizando gerenciadores de estado externos como o Provider, MobX ou Redux.

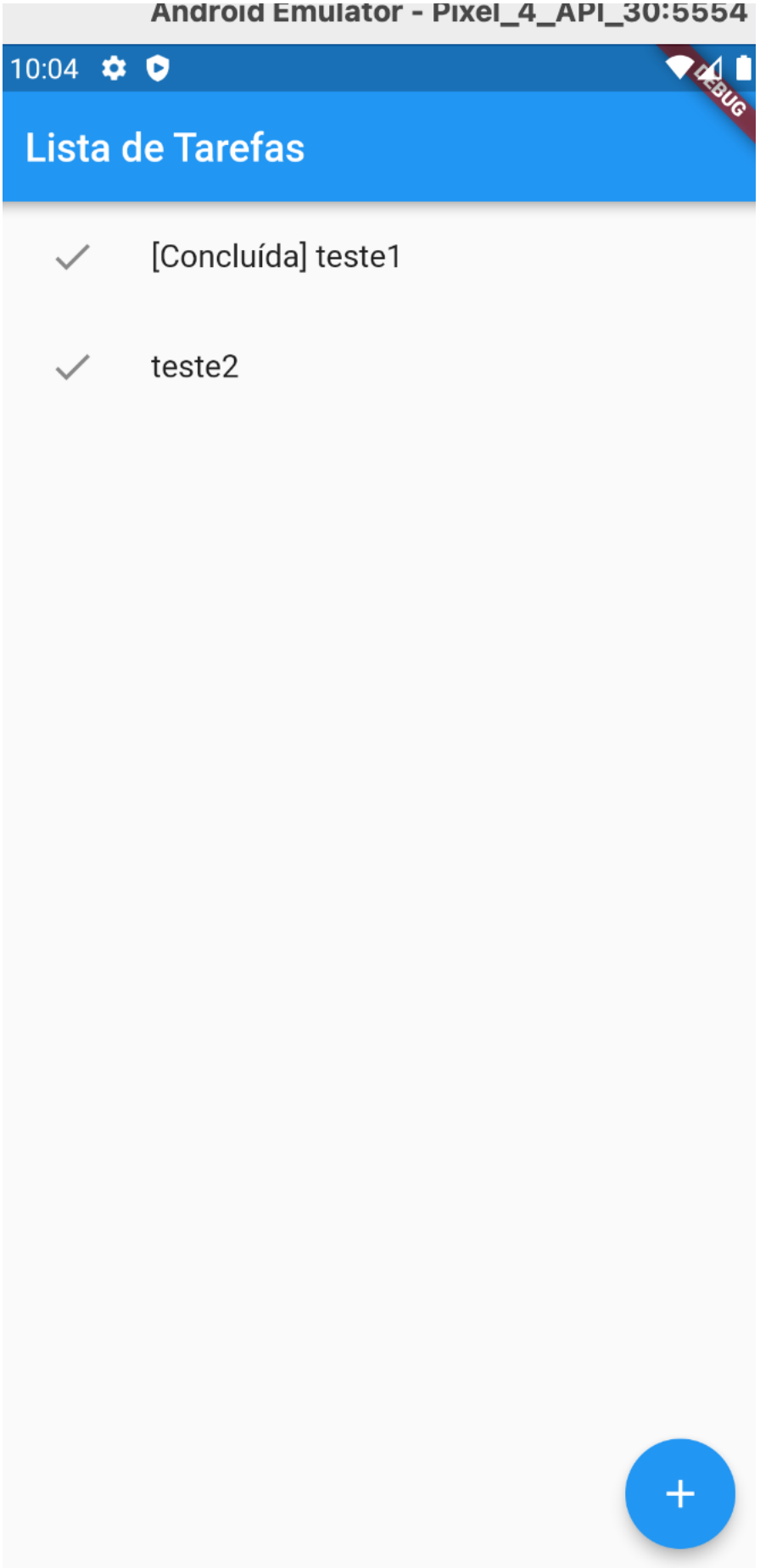
```
import 'package:flutter/material.dart';

void main() {
  runApp(MeuApp());
}

class MeuApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: PaginaInicial(),
    );
  }
}

class PaginaInicial extends StatefulWidget {
  @override
  _PaginaInicialState createState() => _PaginaInicialState();
}

class _PaginaInicialState extends State<PaginaInicial> {
  List<String> tarefas = [];
  TextEditingController _tarefaController = TextEditingController();

  void _adicionarTarefa(String tarefa) {
    setState(() {
      tarefas.add(tarefa);
      _tarefaController.clear();
    });
  }

  void _marcarComoConcluida(int index) {
    setState(() {
      tarefas[index] = '[Concluída] ' + tarefas[index];
    });
  }
```

```
void _excluirTarefa(int index) {
  setState(() {
    tarefas.removeAt(index);
  });
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Lista de Tarefas'),
    ),
    body: ListView.builder(
      itemCount: tarefas.length,
      itemBuilder: (context, index) {
        return Dismissible(
          key: Key(tarefas[index]),
          onDismissed: (direction) {
            _excluirTarefa(index);
          },
          child: ListTile(
            title: Text(tarefas[index]),
            leading: IconButton(
              icon: Icon(Icons.check),
              onPressed: () {
                _marcarComoConcluida(index);
              },
            ),
          ),
        );
      },
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: () {
        showDialog(
          context: context,
          builder: (BuildContext context) {
            return AlertDialog(
              title: Text('Adicionar Tarefa'),
              content: TextField(
                controller: _tarefaController,
                decoration: InputDecoration(labelText: 'Tarefa'),
              ),
              actions: [
                TextButton(
                  onPressed: () {
                    Navigator.of(context).pop();
                  },
                  child: Text('Cancelar'),
```

```
            ),
            ElevatedButton(
              onPressed: () {
                _adicionarTarefa(_tarefaController.text);
                Navigator.of(context).pop();
              },
              child: Text('Adicionar'),
            ),
          ],
        );
      },
    );
  },
  child: Icon(Icons.add),
 ),
);
```

1. Criamos um StatefulWidget chamado PaginaInicial, que é responsável por gerenciar o estado do aplicativo.
2. A lista tarefas armazenará as tarefas adicionadas pelo usuário.
3. O TextEditingController _tarefaController é usado para controlar o campo de texto onde o usuário pode inserir novas tarefas.
4. No método _adicionarTarefa, a tarefa é adicionada à lista e o campo de texto é limpo.
5. No método _marcarComoConcluida, a tarefa é marcada como concluída adicionando [Concluída] ao texto.
6. No método _excluirTarefa, a tarefa é removida da lista quando o usuário desliza o item para excluir.
7. No método build, construímos a interface do usuário com um ListView.builder para exibir a lista de tarefas.
8. Cada tarefa é exibida como um ListTile, e podemos marcá-la como concluída usando o ícone de check.
9. O FloatingActionButton permite ao usuário adicionar novas tarefas.
10. Quando o usuário toca no botão de adicionar, uma caixa de diálogo é exibida para inserir a tarefa. A tarefa é adicionada à lista quando o usuário toca em "Adicionar".


Lista de Tarefas
DEBUG
Adicionar Tarefa
Tarefa
Cancelar
Adicionar
GIF


Android Emulator - Pixel_4_API_30:5554
10:04
DEBUG
Lista de Tarefas
[Concluída] teste1
teste2

Conteúdo da aula:

1. Introdução às Animações em Flutter
2. Tipos de Animações
3. Usando a Classe Animation
4. Criando Animações com AnimatedContainer
5. Animações com AnimatedOpacity
6. Animações com AnimatedBuilder
7. Trabalhando com Curvas de Animação
8. Exemplo Prático: Animação de Transição entre Telas

1. Introdução às Animações em Flutter: As animações em Flutter permitem dar vida à sua interface do usuário, tornando-a mais dinâmica e interativa. Flutter fornece uma poderosa API de animação que facilita a criação de animações suaves e atraentes.

2. Tipos de Animações: Em Flutter, podemos criar diferentes tipos de animações, como translação (mover), rotação, escala, opacidade e muito mais. Cada tipo de animação pode ser aplicado a vários widgets para criar efeitos visuais interessantes.

3. Usando a Classe Animation: Flutter oferece a classe Animation, que representa uma animação em andamento. É possível utilizar essa classe para definir os valores iniciais e finais da animação, bem como determinar a duração e a curva de animação.

4. Criando Animações com AnimatedContainer: O AnimatedContainer é um widget especial que anima automaticamente quando há mudanças em suas propriedades, como altura, largura, cor e muito mais. Isso torna a criação de animações básicas uma tarefa fácil.

5. Animações com AnimatedOpacity: O AnimatedOpacity é usado para animar a opacidade de um widget. Ele pode ser útil para criar transições suaves, tornando os widgets gradualmente visíveis ou invisíveis.

6. Animações com AnimatedBuilder: O AnimatedBuilder é uma ferramenta poderosa que permite criar animações personalizadas combinando Animation com a facilidade de criação de widgets no Flutter. Com o AnimatedBuilder, é possível criar animações mais complexas e personalizadas.

7. Trabalhando com Curvas de Animação: As curvas de animação controlam a taxa de mudança da animação ao longo do tempo. Flutter oferece várias curvas de animação predefinidas, como Curves.linear, Curves.easeIn, Curves.easeOut, Curves.easeInOut e também é possível criar curvas personalizadas.

8. Exemplo Prático: Animação de Transição entre Telas: Neste exemplo, criaremos um aplicativo com duas telas e implementaremos uma animação de transição suave ao alternar entre elas. A animação proporcionará uma experiência mais fluida para o usuário ao navegar entre as telas.

Recursos adicionais:

- Animações com a biblioteca de terceiros "flutter_animation_set".
- Criando animações com "ImplicitAnimations".
- Animação de troca de tema com "Hero Animation".

Codigo

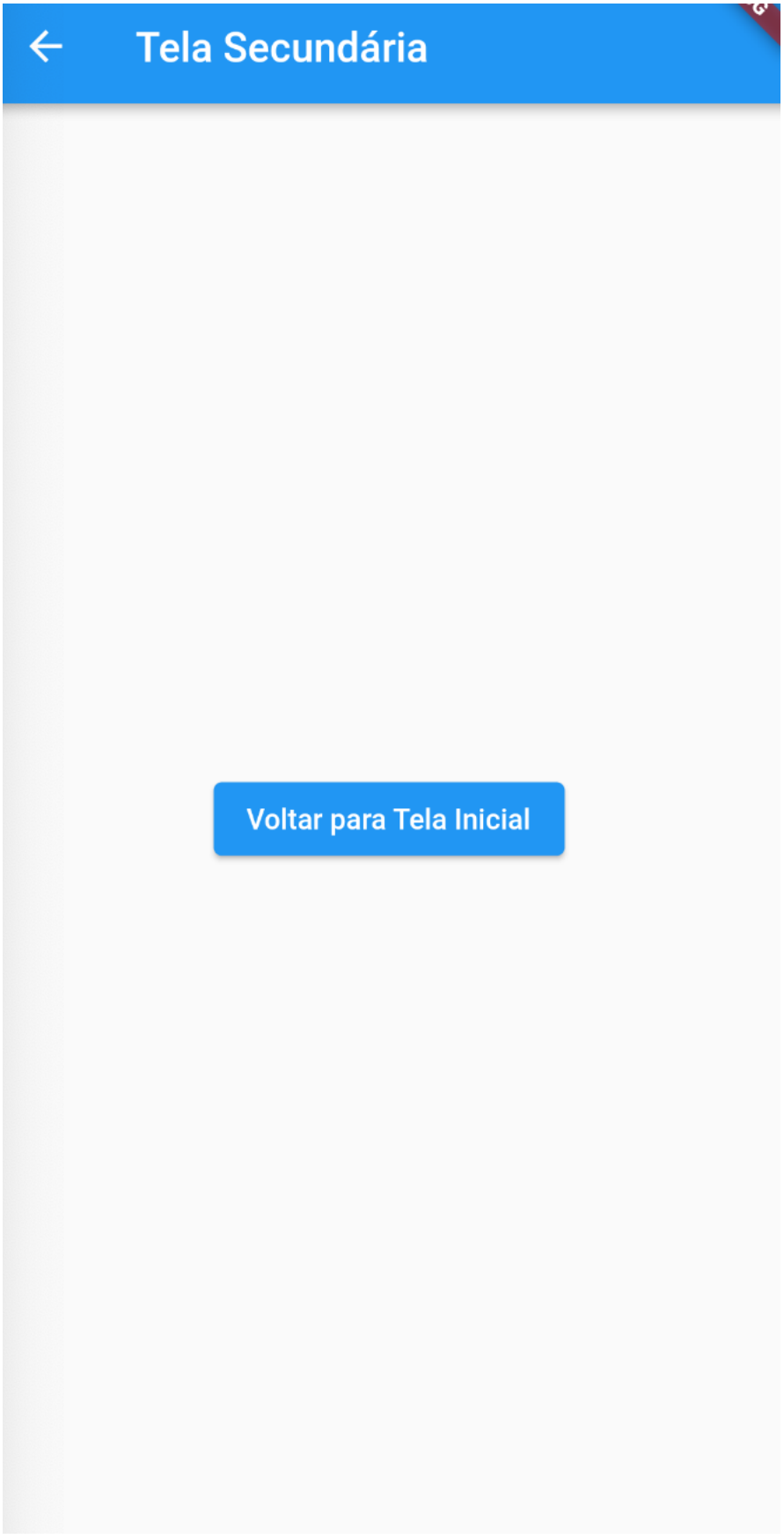
```
import 'package:flutter/material.dart';

void main() {
  runApp(MeuApp());
}

class MeuApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: TelaInicial(),
    );
  }
}

class TelaInicial extends StatefulWidget {
  @override
  _TelaInicialState createState() => _TelaInicialState();
}

class _TelaInicialState extends State<TelaInicial> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Tela Inicial'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.push(
              context,
              _createRoute(),
            );
          },
          child: Text('Ir para Tela Secundária'),
```

```
        ),
      ),
    );
  }

  // Função para criar a animação de transição
  Route _createRoute() {
    return PageRouteBuilder(
                    pageBuilder: (context, animation, secondaryAnimation) =>
TelaSecundaria(),
      transitionsBuilder: (context, animation, secondaryAnimation, child) {
        var begin = Offset(1.0, 0.0);
        var end = Offset.zero;
        var curve = Curves.easeInOut;

        var tween =
            Tween(begin: begin, end: end).chain(CurveTween(curve: curve));

        return SlideTransition(
          position: animation.drive(tween),
          child: child,
        );
      },
    );
  }
}

class TelaSecundaria extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Tela Secundária'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: Text('Voltar para Tela Inicial'),
        ),
      ),
    );
  }
}
```

1. riamos uma classe TelaInicial que é um StatefulWidget com um botão para navegar para a TelaSecundaria.

2. Quando o botão é pressionado, usamos a função _createRoute() para criar uma animação de transição suave usando PageRouteBuilder.
3. A função _createRoute() retorna um PageRouteBuilder que define a animação de transição da tela inicial para a tela secundária.
4. No transitionsBuilder, utilizamos um SlideTransition para criar uma animação de deslizamento horizontal ao navegar para a tela secundária.
5. A classe TelaSecundaria exibe a tela secundária com um botão para retornar à tela inicial.

Ao executar o aplicativo, você verá que ao tocar no botão "Ir para Tela Secundária", a tela transitará suavemente com uma animação de deslizamento para a direita. Quando você tocar no botão "Voltar para Tela Inicial", a transição ocorrerá da tela secundária para a tela inicial.


Tela Secundária
Voltar para Tela Inicial

Aqui ainda não consegui

1. Introdução à Manipulação de Dados em Flutter: Apresentaremos a importância da manipulação de dados em aplicativos Flutter e a necessidade de armazenar informações localmente ou na nuvem.

2. Armazenamento Local com Shared Preferences: Aprenderemos como usar o pacote shared_preferences para armazenar dados de forma simples e leve em SharedPreferences (chave-valor) no dispositivo do usuário.

3. Armazenamento Local com SQLite: Veremos como usar o pacote sqflite para armazenar dados em um banco de dados SQLite local no dispositivo do usuário, permitindo operações mais complexas de consulta e gerenciamento de dados.

4. Armazenamento em Nuvem com Firebase Firestore: Introduziremos o Firebase Firestore, um banco de dados NoSQL em tempo real oferecido pelo Firebase, que permite armazenar e sincronizar dados na nuvem.

5. Utilizando o Firebase Firestore em Flutter: Explicaremos como configurar o Firebase Firestore em um projeto Flutter e como realizar operações básicas de leitura e escrita de dados no Firestore.

6. CRUD (Create, Read, Update, Delete) com Firebase Firestore: Aprenderemos como realizar operações de CRUD (criar, ler, atualizar e excluir) no Firebase Firestore, permitindo manipular dados na nuvem de forma eficiente.

7. Mecanismos de Persistência em Nuvem Alternativos: Apresentaremos outros serviços de armazenamento em nuvem além do Firebase Firestore, como o Firebase Realtime Database, AWS Amplify, entre outros.

8. Gerenciamento de Estado e Dados: Discutiremos estratégias para o gerenciamento de estado e dados em aplicativos Flutter, garantindo a consistência dos dados locais e em nuvem.

9. Exemplo Prático: Criando um App com Armazenamento Local e em Nuvem: Criaremos um exemplo prático de um aplicativo Flutter que permite ao usuário criar, ler, atualizar e excluir dados, salvando-os localmente com SharedPreferences e também sincronizando-os na nuvem com o Firebase Firestore.

xemplo prático de um aplicativo Flutter que utiliza tanto armazenamento local com shared_preferences quanto armazenamento em nuvem com o Firebase Firestore. Neste exemplo, criaremos um aplicativo de lista de tarefas, onde o

usuário poderá adicionar tarefas, que serão salvas localmente no dispositivo e também sincronizadas na nuvem.

Antes de prosseguir, certifique-se de ter configurado corretamente o ambiente de desenvolvimento Flutter em seu computador e adicionado os pacotes shared_preferences e cloud_firestore ao seu projeto (conforme mencionado em aulas anteriores).

1. Configuração do Firebase Firestore: Antes de começarmos, você precisa configurar o Firebase Firestore em seu projeto Flutter. Siga os passos abaixo:

- Acesse o Console do Firebase (https://console.firebase.google.com/) e crie um novo projeto.
- No painel do projeto, clique em "Adicionar app" e selecione a opção Flutter.
- Siga as instruções para adicionar o arquivo de configuração google-services.json ao seu projeto Flutter.

Aula: Manipulando Recursos do Dispositivo em Flutter

Objetivo da Aula:
Nesta aula, você aprenderá como manipular recursos do dispositivo em Flutter, focando em imagens. Abordaremos como acessar a galeria e a câmera do dispositivo para carregar e capturar imagens. Além disso, você aprenderá a exibir as imagens selecionadas em seu aplicativo Flutter.

Conteúdo da Aula:

1. Introdução
   - Visão geral da aula
   - Explicação sobre a manipulação de recursos do dispositivo

2. Acesso à Galeria
   - Importando dependências
   - Criando o botão para acessar a galeria
   - Lidando com permissões do usuário
   - Carregando imagens da galeria

3. Captura de Imagem pela Câmera
   - Importando dependências
   - Criando o botão para acessar a câmera
   - Lidando com permissões do usuário
   - Capturando uma imagem da câmera

4. Exibindo Imagens Selecionadas
   - Criando uma lista para armazenar as imagens selecionadas
   - Exibindo as imagens na tela

- Implementando a funcionalidade de remover imagens da lista

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

void main() {
  runApp(MeuApp());
}

class MeuApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: RecursosDoDispositivo(),
    );
  }
}

class RecursosDoDispositivo extends StatefulWidget {
  @override
  _RecursosDoDispositivoState createState() => _RecursosDoDispositivoState();
}

class _RecursosDoDispositivoState extends State<RecursosDoDispositivo> {
  List<File> _imagensSelecionadas = [];

  Future<void> _acessarGaleria() async {
    final picker = ImagePicker();
    final pickedFile = await picker.getImage(source: ImageSource.gallery);

    if (pickedFile != null) {
      setState(() {
        _imagensSelecionadas.add(File(pickedFile.path));
      });
    }
  }

  Future<void> _acessarCamera() async {
    final picker = ImagePicker();
    final pickedFile = await picker.getImage(source: ImageSource.camera);

    if (pickedFile != null) {
      setState(() {
        _imagensSelecionadas.add(File(pickedFile.path));
      });
    }
  }

  void _removerImagem(int index) {
    setState(() {
      _imagensSelecionadas.removeAt(index);
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Manipulando Recursos do Dispositivo'),
      ),
      body: Column(
```

```
        children: [
          SizedBox(height: 20),
          ElevatedButton(
            onPressed: _acessarGaleria,
            child: Text('Acessar Galeria'),
          ),
          ElevatedButton(
            onPressed: _acessarCamera,
            child: Text('Acessar Câmera'),
          ),
          SizedBox(height: 20),
          Expanded(
            child: ListView.builder(
              itemCount: _imagensSelecionadas.length,
              itemBuilder: (context, index) {
                return ListTile(
                  leading: Image.file(_imagensSelecionadas[index]),
                  trailing: IconButton(
                    icon: Icon(Icons.delete),
                    onPressed: () => _removerImagem(index),
                  ),
                );
              },
            ),
          ),
        ],
      ),
    );
  }
}
```

Explicação do Código:

1. O código inicia com a importação das dependências necessárias: `material.dart` para a interface do aplicativo e `image_picker.dart` para acessar as funcionalidades de galeria e câmera.

2. A classe `RecursosDoDispositivo` é um `StatefulWidget` que armazena as imagens selecionadas em uma lista.

3. O método `_acessarGaleria` é chamado ao pressionar o botão "Acessar Galeria". Ele usa o `ImagePicker` para abrir a galeria do dispositivo e, quando uma imagem é selecionada, adiciona-a à lista `_imagensSelecionadas`.

4. O método `_acessarCamera` é chamado ao pressionar o botão "Acessar Câmera". Ele usa o `ImagePicker` para abrir a câmera do dispositivo e, quando uma imagem é capturada, adiciona-a à lista `_imagensSelecionadas`.

5. O método `_removerImagem` é usado para remover imagens da lista `_imagensSelecionadas` ao pressionar o ícone de exclusão.

6. Na interface do aplicativo, há dois botões: "Acessar Galeria" e "Acessar Câmera". Abaixo dos botões, há um `ListView.builder` que exibe as imagens selecionadas na lista `_imagensSelecionadas`, juntamente com um ícone de exclusão para cada imagem.

Com este código de exemplo, você aprendeu a manipular recursos do dispositivo, como a galeria e a câmera, para carregar e capturar imagens em um aplicativo Flutter. Agora, você pode expandir esse exemplo para criar aplicativos mais complexos que lidam com recursos do dispositivo em suas funcionalidades.

Aqui está como você pode adicionar a dependência no **pubspec.yaml**:

1. Abra o arquivo **pubspec.yaml** no diretório raiz do seu projeto.
2. Encontre a seção **dependencies** no arquivo.
3. Adicione a dependência do **image_picker** na seção **dependencies**, como mostrado abaixo:

```
dependencies:
  flutter:
    sdk: flutter
  image_picker: ^0.8.4+4
```

4. Salve o arquivo **pubspec.yaml**.
5. Em seguida, execute o comando **flutter pub get** no terminal do seu IDE ou no terminal do seu sistema operacional para baixar e instalar o pacote **image_picker**.

Programa executado –

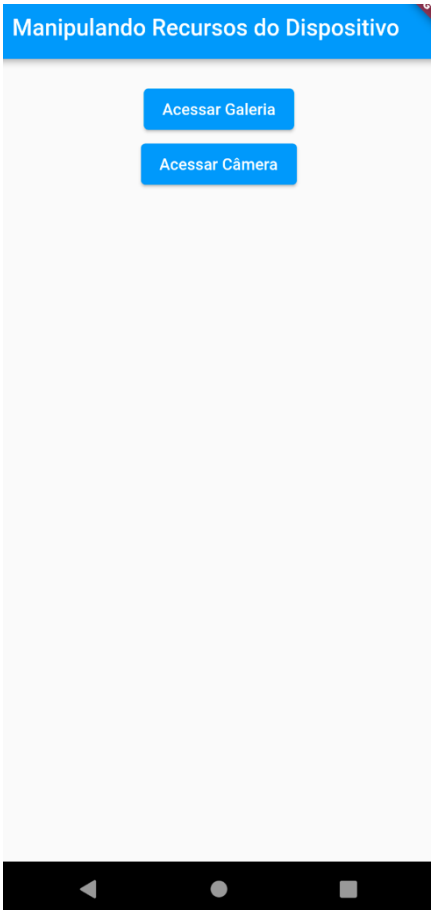


Figura tela principal

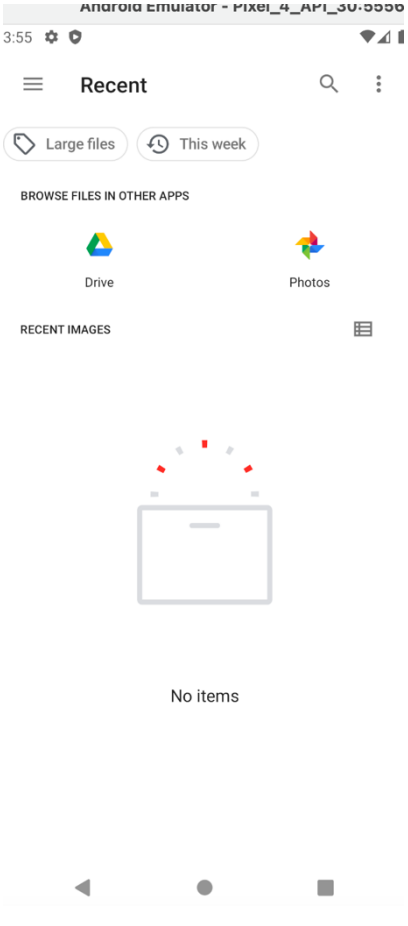


Figura tela arquivos

Figura Foto