

Guia Prático: Introdução ao Desenvolvimento com Flutter e Dart

Flutter:

1. Introdução ao Flutter:

Flutter é um framework desenvolvido pelo Google que permite a criação de aplicativos nativos multiplataforma. Isso inclui o suporte a sistemas operacionais como Android, iOS, Windows, macOS, Linux e Web, utilizando uma única base de código. Uma das suas principais características é o uso de **widgets**, componentes visuais reutilizáveis que formam a interface de usuário. Cada widget em Flutter é implementado como uma classe em Dart.

2. Criando um Projeto Flutter:

Para iniciar um projeto Flutter, é necessário configurar o ambiente de desenvolvimento. Uma das maneiras mais simples é utilizar o **Visual Studio Code (VS Code)**, um editor leve e altamente configurável. O Flutter e o Dart devem ser instalados como extensões no VS Code.

Exemplo de criação de um projeto no terminal:

```
bash                                                                    Copiar código

flutter create nome_do_projeto
cd nome_do_projeto
code .
```

Após isso, o arquivo principal da aplicação será o `main.dart`, que contém a função `main()`. O ponto de entrada da aplicação é representado por essa função.

3. Estrutura Básica do App:

A estrutura básica de um aplicativo Flutter é iniciada com o `MaterialApp`, que é o widget de nível superior. Ele configura a navegação entre telas e diversos parâmetros visuais do app. Um widget importante é o `Scaffold`, que fornece uma estrutura visual padrão para apps usando **Material Design**.

Exemplo básico de app Flutter:

```
dart Copiar código

import 'package:flutter/material.dart';

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

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

O **hot reload** permite que você veja as mudanças no app quase em tempo real sem perder o estado da aplicação. Já o **hot restart** reinicia toda a aplicação.

4. Exibição de Imagens:

A exibição de imagens dinâmicas em Flutter pode ser feita utilizando requisições HTTP. Para este exemplo, vamos usar a API da Pexels para carregar imagens.

Configuração da dependência http no pubspec.yaml:

```
yaml Copiar código

dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.3
```

Exemplo de requisição para carregar imagens:

```
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:convert';

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

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

class ImageScreen extends StatefulWidget {
  @override
  _ImageScreenState createState() => _ImageScreenState();
}

class _ImageScreenState extends State<ImageScreen> {
  Future<List<String>> fetchImages() async {
    final response = await http.get(Uri.parse('https://api.pexels.com/v1/curated'),
      headers: {'Authorization': 'sua_api_key'});

    if (response.statusCode == 200) {
      List<dynamic> data = json.decode(response.body)['photos'];
      return data.map((photo) => photo['src']['medium'] as String).toList();
    } else {
      throw Exception('Falha ao carregar imagens');
    }
  }
}
```

```

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text('Imagens Dinâmicas')),
    body: FutureBuilder<List<String>>({
      future: fetchImages(),
      builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.waiting) {
          return Center(child: CircularProgressIndicator());
        } else if (snapshot.hasError) {
          return Center(child: Text('Erro: ${snapshot.error}'));
        } else {
          return ListView.builder(
            itemCount: snapshot.data!.length,
            itemBuilder: (context, index) {
              ~~~~~
            },
          );
        }
      },
    ),
  );
}

```

5. Integração com a API:

O pacote `http` facilita a comunicação entre o app e APIs REST. O método `get` é usado para realizar requisições e manipular as respostas JSON.

Exemplo de utilização do pacote `http`:

```

dart
Copiar código

var url = Uri.parse('https://api.pexels.com/v1/curated');
var response = await http.get(url, headers: {'Authorization': 'sua_api_key'});
print('Response status: ${response.statusCode}');
print('Response body: ${response.body}');

```

Para manipular o JSON, utilizamos o método `json.decode()` para converter a resposta em um mapa de dados.

6. Gerenciamento de Estado com `StatefulWidget`:

O estado em Flutter é alterado dinamicamente durante a execução do aplicativo. O uso de `StatefulWidget` é apropriado para gerenciar o estado que precisa mudar, como uma lista de imagens carregadas.

```
dart Copiar código

class ImageScreen extends StatefulWidget {
  @override
  _ImageScreenState createState() => _ImageScreenState();
}

class _ImageScreenState extends State<ImageScreen> {
  List<String> images = [];

  void _addImage(String imageUrl) {
    setState(() {
      images.add(imageUrl);
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Imagens')),
      body: ListView.builder(
        itemCount: images.length,
        itemBuilder: (context, index) {
          return Image.network(images[index]);
        },
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          _addImage('nova_url_imagem');
        },
        child: Icon(Icons.add),
      ),
    );
  }
}
```

7. Introdução ao Gerenciamento de Estado em Flutter:

No Flutter, **estado** refere-se à coleção de variáveis que podem ser alteradas durante a vida útil de um widget. Gerenciar o estado de forma eficaz é fundamental, especialmente quando se trabalha com interfaces de usuário dinâmicas. Um dos métodos mais básicos de gerenciamento de estado é o uso de **StatefulWidgets**, que permitem que a interface do usuário responda a mudanças no estado da aplicação.

Exemplo de **StatefulWidgets** básico:

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

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

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

class CounterApp extends StatefulWidget {
  @override
  _CounterAppState createState() => _CounterAppState();
}

class _CounterAppState extends State<CounterApp> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Gerenciamento de Estado'),
      ),
      body: Center(
        child: Text('Contagem: $_counter'),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Embora **StatefulWidgets** sejam úteis em aplicações simples, eles apresentam limitações em aplicativos mais complexos, pois o código tende a se tornar difícil de

manter à medida que cresce.

8. Problemas com o Gerenciamento de Estado Local:

O **StatefulWidget** se torna um desafio quando o estado precisa ser compartilhado entre várias partes da aplicação. O conceito de **lifting state up** é uma técnica para mover o estado para widgets pais, facilitando o compartilhamento entre widgets filhos.

Exemplo de Lifting State Up:

```
dart Copiar código

class ParentWidget extends StatefulWidget {
  @override
  _ParentWidgetState createState() => _ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Contador: $_counter'),
        ChildWidget(onPressed: _incrementCounter),
      ],
    );
  }
}

class ChildWidget extends StatelessWidget {
  final VoidCallback onPressed;

  ChildWidget({required this.onPressed});
```

```
@override
Widget build(BuildContext context) {
  return ElevatedButton(
    onPressed: onPressed,
    child: Text('Adicionar'),
  );
}
```


Neste exemplo, o estado (`_counter`) é gerenciado pelo widget pai e compartilhado com o widget filho por meio de uma função de callback.

9. Introdução ao Provider:

Provider é uma solução para gerenciar estado global em Flutter, permitindo que o estado seja acessado em qualquer parte da aplicação. O **Provider** facilita o gerenciamento de estado reativo em conjunto com a classe **ChangeNotifier**.

Configuração do Provider no pubspec.yaml:

yaml

 Copiar código

```
dependencies:
  provider: ^6.0.0
```

Exemplo de utilização do Provide


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

void main() => runApp(
  ChangeNotifierProvider(
    create: (context) => Counter(),
    child: MyApp(),
  ),
);

class Counter with ChangeNotifier {
  int _count = 0;

  int get count => _count;

  void increment() {
    _count++;
    notifyListeners();
  }
}

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

class CounterScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Provider Exemplo'),
      ),
      body: Center(
        child: Text('Contagem: ${context.watch<Counter>().count}'),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => context.read<Counter>().increment(),
        child: Icon(Icons.add),
      ),
    );
  }
}
```

DART:

1. Introdução ao Dart:

Dart é uma linguagem de programação criada pelo Google, usada principalmente no desenvolvimento de aplicativos móveis e web, particularmente no framework Flutter. A linguagem é **estaticamente tipada** e oferece suporte à **orientação a objetos**.

2. Sintaxe Básica:

Em Dart, erros de compilação ocorrem quando o código está incorretamente formatado ou quando não segue as regras da linguagem, enquanto erros de execução ocorrem durante a execução do programa.

2.1 Exemplo de erro de compilação:

```
dart Copiar código  
  
int x = "hello"; // Erro de compilação: Tipo incorreto
```

Exemplo de erro de execução:

2.2 Dart oferece três Tipos de Variáveis:

As variáveis em programação podem ser categorizadas com base em seu escopo, tempo de vida e uso. Aqui estão os principais tipos de variáveis:

2.2.1. Variável Global:

- **Definição:** Uma variável global é acessível por todas as funções ou métodos em um programa. Em Dart, todas as variáveis declaradas fora das funções ou classes são consideradas globais, mas não há modificadores de acesso explícitos como em algumas linguagens, como `public` ou `static`.
- **Escopo:** A variável é acessível em todo o arquivo ou programa.

Exemplo:

```
dart Copiar código

int contadorGlobal = 0;

void incrementar() {
    contadorGlobal++;
}

void main() {
    incrementar();
    print(contadorGlobal); // Saída: 1
}
```

2.2.2. Variável Estática:

- **Definição:** Uma variável estática pertence à classe em vez de instâncias da classe. Ela é inicializada uma vez e mantém seu valor entre diferentes instâncias.
- **Escopo:** Associada à classe, compartilhada por todas as instâncias da classe.

Exemplo:

```
dart Copiar código

class Contador {
    static int valor = 0;

    void incrementar() {
        valor++;
    }
}

void main() {
    Contador c1 = Contador();
    Contador c2 = Contador();

    c1.incrementar();
    print(Contador.valor); // Saída: 1

    c2.incrementar();
    print(Contador.valor); // Saída: 2
}
```

2.2.3. Variável Local:

- **Definição:** Uma variável local é declarada dentro de um método ou função e só pode ser usada dentro desse escopo.
- **Escopo:** Limitado ao bloco onde foi declarada.

Exemplo:

```
dart Copiar código

void main() {
  int contadorLocal = 5; // Variável local dentro da função main
  print(contadorLocal);
}
```

2.3 Dart oferece três formas principais de declarar variáveis:

- **var:** Para variáveis cujo tipo é inferido.
- **final:** Variáveis imutáveis que são definidas uma vez.
- **const:** Como **final**, mas com valor determinado em tempo de compilação.

Exemplo de variáveis:

```
dart Copiar código

var nome = "Ana";
final idade = 25;
const pi = 3.14159;
```

2.4 Dart oferece três formas principais de declarar variáveis:

Aqui estão os tipos de dados primitivos e compostos comumente usados em Dart e outras linguagens de programação.

2.4.1. Boolean (Booleana):

- **Definição:** Representa valores de verdadeiro (`true`) ou falso (`false`).
- **Tamanho:** 1 bit (ou conforme a implementação da linguagem).

Exemplo:

```
dart Copiar código

bool ativo = true;
```

2.4.2. char (caracter):

- **Definição:** Representa um único caractere. Em Dart, não há um tipo `char` dedicado, mas os caracteres podem ser representados como `String` de comprimento 1.
- **Tamanho:** 1 byte (geralmente).

Exemplo:

```
dart Copiar código  
  
String letra = 'A'; // Não há tipo char em Dart
```

2.4.3. byte:

- **Definição:** Um tipo de dado que pode armazenar um valor inteiro de 0 a 255. Em Dart, você pode trabalhar com bytes através de arrays de inteiros.

Exemplo:

```
dart Copiar código  
  
List<int> bytes = [65, 66, 67]; // Representa os caracteres A, B, C
```

2.4.4. short:

- **Definição:** Tipo inteiro de 16 bits, disponível em algumas linguagens. Não existe nativamente em Dart, mas pode-se usar `int` para representar valores dentro do intervalo -32.768 a 32.767 .

Exemplo:

```
dart Copiar código  
  
int numeroCurto = 32767; // Valor máximo de um short
```

2.4.5. int:

- **Definição:** Tipo de dado para armazenar números inteiros.
- **Tamanho:** 32 ou 64 bits, dependendo da plataforma.

Exemplo:


```
dart Copiar código  
  
int numeroInteiro = 100;
```

2.4.6. long:

- **Definição:** Tipo de dado para armazenar números inteiros grandes. Em Dart, `int` já suporta grandes valores.
- **Tamanho:** 64 bits.

Exemplo:

dart

 Copiar código


```
int numeroGrande = 9223372036854775807; // Suportado por int em Dart
```

2.4.7. float:

- **Definição:** Tipo de dado para armazenar números de ponto flutuante com precisão simples. Em Dart, você pode usar `double`, que trata valores de ponto flutuante.
- **Tamanho:** 32 bits.

Exemplo:

dart

 Copiar código


```
double numeroFlutuante = 3.14;
```

2.4.8. double:

- **Definição:** Tipo de dado para armazenar números de ponto flutuante com precisão dupla.
- **Tamanho:** 64 bits.

Exemplo:

dart

 Copiar código


```
double valorPreciso = 3.141592653589793;
```

2.4.9. String:

- **Definição:** Tipo de dado usado para representar sequências de caracteres.

Exemplo:

dart

 Copiar código


```
String texto = 'Olá, mundo!';
```

2.4.10. Array:

- **Definição:** Coleção ordenada de elementos do mesmo tipo. Em Dart, as arrays são representadas por `List`.

Exemplo:

dart

 Copiar código


```
List<int> numeros = [1, 2, 3, 4, 5];
```

2.4.11. dynamic:

- **Definição:** Permite que uma variável altere seu tipo de valor em tempo de execução. É útil quando o tipo de dado não é conhecido durante a compilação.

Exemplo:

dart

 Copiar código


```
List<int> numeros = [1, 2, 3, 4, 5];
```

2.4.12. const:

- **Definição:** Declara uma constante em tempo de compilação.

Exemplo:

dart

 Copiar código


```
const pi = 3.14159;
```

2.4.13. final:

- **Definição:** Declara uma constante que pode ser inicializada em tempo de execução.

Exemplo:

dart

 Copiar código

```
final dataAtual = DateTime.now();
```

3. Funções e Métodos:

Dart permite a criação de funções com ou sem parâmetros e oferece suporte a funções anônimas (lambdas). Funções podem retornar valores ou serem do tipo `void` (sem retorno).

Exemplo de função com retorno:

```
dart Copiar código

int soma(int a, int b) {
  return a + b;
}
```

Exemplo de função anônima:

```
dart Copiar código

var multiplicar = (int x, int y) => x * y;
```

4. Classes e Objetos:

Dart suporta a criação de classes com atributos e métodos, oferecendo suporte a herança e encapsulamento.

Exemplo de classe em Dart:

```
dart Copiar código

class Pessoa {
  String nome;
  int idade;

  Pessoa(this.nome, this.idade);

  void exibirInfo() {
    print("Nome: $nome, Idade: $idade");
  }
}

void main() {
  var p1 = Pessoa("Ana", 25);
  p1.exibirInfo();
}
```

5. Controle de Fluxo:

5.1 Estruturas de Controle:

No Dart, como em outras linguagens de programação, as estruturas de controle permitem que o fluxo de execução do código seja alterado de acordo com condições ou repetições. As principais estruturas de controle são:

- **Condicionais:**

- `if/else`: Utilizado para verificar se uma condição seja verdadeira ou falsa.
- `switch`: Útil para verificar múltiplas condições, especialmente quando os valores são discretos.

Exemplo:

```
dart Copiar código

void main() {
  int numero = 5;

  // Estrutura if-else
  if (numero > 0) {
    print('Número positivo');
  } else {
    print('Número negativo ou zero');
  }

  // Estrutura switch
  switch (numero) {
    case 1:
      print('Um');
      break;
    case 5:
      print('Cinco');
      break;
    default:
      print('Outro número');
  }
}
```

5.2 Laços de Repetição:

Dart suporta os loops `for`, `while` e `do-while`:

- `for`: Loop com um contador definido.
- `while`: Executa o bloco de código enquanto uma condição for verdadeira.
- `do-while`: Similar ao `while`, mas garante que o código seja executado pelo menos uma vez.

Exemplo:

```
dart Copiar código

void main() {
  // Loop for
  for (int i = 0; i < 5; i++) {
    print('Contador: $i');
  }

  // Loop while
  int contador = 0;
  while (contador < 5) {
    print('Contador: $contador');
    contador++;
  }

  // Loop do-while
  int contador2 = 0;
  do {
    print('Contador do-while: $contador2');
    contador2++;
  } while (contador2 < 5);
}
```

5.3 Operador Ternário:

O operador ternário é uma forma concisa de escrever uma condição if-else em uma única linha.

Exemplo:

```
dart Copiar código

void main() {
  int idade = 18;
  String resultado = idade >= 18 ? 'Maior de idade' : 'Menor de idade';
  print(resultado);
}
```

6. Listas, Mapas e Sets:

Dart possui diferentes coleções para armazenar dados, como Listas, Sets e Mapas.

6.1 Listas:

Uma lista é uma coleção ordenada de elementos.

Exemplo:

```
dart Copiar código

void main() {
  List<String> frutas = ['Maçã', 'Banana', 'Laranja'];
  frutas.add('Morango'); // Adiciona um elemento
  print(frutas);
}
```

6.2 Sets:

Sets são coleções de elementos únicos.

Exemplo:

```
dart Copiar código

void main() {
  Set<int> numeros = {1, 2, 3, 4, 4}; // O valor duplicado será ignorado
  numeros.add(5);
  print(numeros);
}
```

6.3 Mapas:

Mapas são coleções de pares chave-valor.

Exemplo:

```
dart Copiar código

void main() {
  Map<String, String> capitais = {
    'Brasil': 'Brasília',
    'França': 'Paris',
  };
  capitais['Espanha'] = 'Madrid'; // Adiciona uma nova chave-valor
  print(capitais);
}
```

7. Funções de Ordem Superior:

Funções de ordem superior são funções que podem receber ou retornar outras funções.

7.1 Funções `map()`, `reduce()` e `fold()`:

Essas funções são amplamente usadas para manipulação de coleções.

- `map()`: Transforma cada elemento de uma lista.
- `reduce()`: Reduz uma coleção a um único valor com base em uma função acumulativa.
- `fold()`: Similar ao `reduce()`, mas permite definir um valor inicial.

Exemplo:

```
dart Copiar código

void main() {
  List<int> numeros = [1, 2, 3, 4];

  // map() transforma cada elemento
  List<int> dobrados = numeros.map((n) => n * 2).toList();
  print(dobrados); // [2, 4, 6, 8]

  // reduce() soma todos os elementos
  int soma = numeros.reduce((valor, elemento) => valor + elemento);
  print(soma); // 10

  // fold() começa com um valor inicial
  int produto = numeros.fold(1, (valor, elemento) => valor * elemento);
  print(produto); // 24
}
```

8. Exceções:

Dart permite o tratamento de exceções com os blocos `try`, `catch`, e `finally`.

8.1 Exemplo de Tratamento de Exceções:

Exemplo:

```
dart Copiar código

void main() {
  try {
    int resultado = 10 ~/ 0; // Tentativa de dividir por zero
  } catch (e) {
    print('Ocorreu um erro: $e');
  } finally {
    print('Bloco finally sempre é executado.');
```

8.2 Exceções Personalizadas:

É possível lançar exceções personalizadas com `throw`.

Exemplo:

```
dart Copiar código

void verificarIdade(int idade) {
  if (idade < 18) {
    throw Exception('Menor de idade');
  } else {
    print('Maior de idade');
  }
}

void main() {
  try {
    verificarIdade(17);
  } catch (e) {
    print(e);
  }
}
```

9. Programação Assíncrona:

Dart suporta programação assíncrona através de `Future` e `async/await`.

9.1 `Future` e `async/await`:

Um `Future` representa um valor ou erro que estará disponível em algum momento no futuro.

Exemplo:

```
dart Copiar código

Future<String> buscarDados() async {
  await Future.delayed(Duration(seconds: 2)); // Simula uma operação demorada
  return 'Dados carregados';
}

void main() async {
  print('Iniciando...');
  String resultado = await buscarDados();
  print(resultado);
}
```

9.2 Streams:

Streams permitem manipular fluxos contínuos de dados.

Exemplo:

```
dart Copiar código

Stream<int> contarAteTres() async* {
  for (int i = 1; i <= 3; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i;
  }
}

void main() async {
  await for (int valor in contarAteTres()) {
    print(valor);
  }
}
```


10. Classes Avançadas:

10.1 Construtores Nomeados e Factory:

Dart permite a criação de construtores nomeados e `factory`.

Exemplo:

dart

 Copiar código

```
class Pessoa {
  String nome;
  int idade;

  // Construtor normal
  Pessoa(this.nome, this.idade);

  // Construtor nomeado
  Pessoa.comIdade(this.nome) {
    idade = 18; // Idade padrão
  }

  // Construtor factory
  factory Pessoa.jovem(String nome) {
    return Pessoa(nome, 15);
  }
}

void main() {
  Pessoa p1 = Pessoa('Ana', 20);
  Pessoa p2 = Pessoa.comIdade('Carlos');
  Pessoa p3 = Pessoa.jovem('Lucas');


  print(p1.nome); // Ana
  print(p2.idade); // 18
  print(p3.idade); // 15
}
```

10.2 final e const:

- **final**: Variável que só pode ser atribuída uma vez.
- **const**: Valor imutável em tempo de compilação.

Exemplo:

dart

 Copiar código


```
void main() {  
  final idade = 30;  
  const pi = 3.14;  
  
  // idade = 35; // Erro, não pode ser alterada  
  // pi = 3.1415; // Erro, valor constante  
}
```

11. Módulos e Importações:

Dart permite dividir o código em diferentes arquivos para modularização.

Exemplo:

dart

 Copiar código

```
// arquivo principal.dart  
import 'utils.dart';  
  
void main() {  
  saudar('Mundo');  
}  
  
// arquivo utils.dart  
void saudar(String nome) {  
  print('Olá, $nome!');  
}
```


12. Manipulação de JSON:

Dart oferece a biblioteca `dart:convert` para manipular dados JSON.

12.1. Converter JSON para Objeto Dart:

Exemplo:

dart


 Copiar código

```
import 'dart:convert';  
  
void main() {  
  String jsonString = '{"nome": "Ana", "idade": 25}';  
  Map<String, dynamic> dados = json.decode(jsonString);  
  print(dados['nome']); // Ana  
}
```


12.2 Modelo de Classe para JSON

Exemplo:

dart

 Copiar código

```
class Pessoa {  
  String nome;  
  int idade;  
  
  Pessoa(this.nome, this.idade);  
  
  // Converter JSON para objeto
```