

Velozes e Flutterosos

Desafio do Código



Felipe M. Scalco

Fundamentos do Flutter

00

O que é Flutter?

Flutter é um framework de código aberto desenvolvido pelo Google, lançado inicialmente em maio de 2017. Criado para facilitar o desenvolvimento de aplicativos nativos com uma única base de código, o Flutter utiliza a linguagem Dart e um motor gráfico próprio chamado Skia. Este framework surgiu como uma solução para os desafios enfrentados no desenvolvimento multiplataforma, permitindo que desenvolvedores escrevam uma única vez e implantem em Android, iOS, Web e desktop. As vantagens do Flutter incluem o hot reload, que agiliza o processo de desenvolvimento ao permitir que mudanças no código sejam visualizadas instantaneamente; a extensa coleção de widgets personalizáveis, que facilita a criação de interfaces de usuário sofisticadas e responsivas; e a redução de tempo e custo com a manutenção de um único código para diversas plataformas. No entanto, existem desvantagens, como a necessidade de aprender Dart, o que pode ser um obstáculo para iniciantes, e a dependência de bibliotecas de terceiros para algumas funcionalidades, o que pode resultar em desafios de compatibilidade e manutenção.



Instalação e Configuração

01

Instalação do Flutter

Para adentrar o mundo da programação com Flutter, a primeira etapa é garantir que o ambiente de desenvolvimento esteja adequadamente configurado. Felizmente, o processo de instalação do Flutter é tão direto quanto os resultados que você poderá alcançar com ele.

Verificação de Pré-requisitos

Antes de começar, verifique se o seu sistema atende aos requisitos mínimos do Flutter. Certifique-se de ter o Git instalado para acessar os repositórios necessários.

Instalação do Flutter

Obtenção do Flutter SDK

O Flutter SDK é facilmente obtido diretamente do GitHub. Utilizando o Git, simplesmente clone o repositório do Flutter e acrescente o caminho para o binário ao seu PATH.

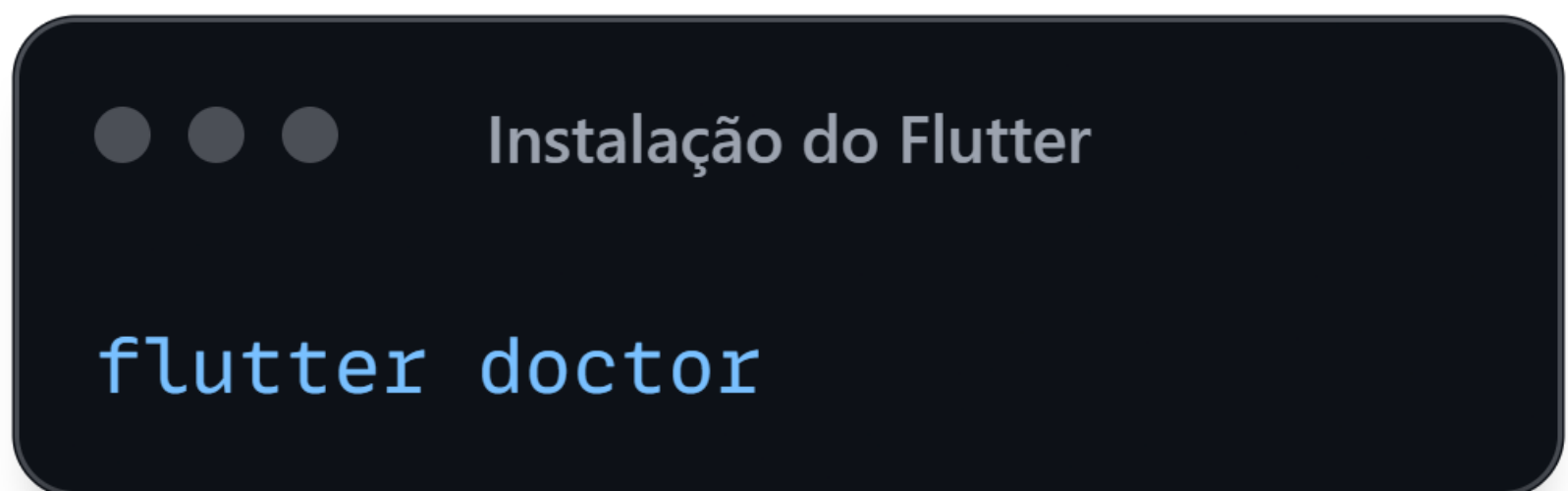
Instalação do Flutter

```
git clone https://github.com/flutter/flutter.git  
export PATH="$PATH:`pwd`/flutter/bin"
```

Instalação do Flutter

Configuração do Flutter Doctor

O Flutter Doctor é uma ferramenta vital que verifica se o ambiente de desenvolvimento está pronto para a ação. Execute “flutter doctor” no terminal e siga as instruções para resolver quaisquer problemas detectados.



Configuração do Ambiente de Desenvolvimento

Com o Flutter instalado, é hora de integrá-lo ao seu ambiente de desenvolvimento preferido. Seja você um fã de Visual Studio Code, Android Studio ou qualquer outro ambiente, o Flutter se adapta a você.

Instalação de Plugins e Extensões

Para uma experiência otimizada, instale as extensões pertinentes ao Flutter em seu IDE de escolha. Elas oferecem suporte para autocompletar, depuração e outras funcionalidades essenciais.

Configuração do Emulador ou Dispositivo Físico

Para testar suas aplicações, é necessário um emulador ou dispositivo físico. Configurar um emulador é simples e pode ser feito através do Android Studio ou do AVD Manager.

Exemplo Prático: Criando um Projeto Flutter

Vamos dar vida ao aprendizado com um exemplo prático. Criaremos um aplicativo Flutter básico que exibe uma mensagem de boas-vindas.

Criação de um Novo Projeto

Utilize o comando “flutter create” seguido pelo nome do seu projeto para criar a estrutura inicial de um aplicativo Flutter.



Exemplo Prático: Criando um Projeto Flutter

Edição do Código

Abra o projeto em seu IDE e navegue até o arquivo “lib/main.dart”. Substitua o código existente pelo seguinte:

```

Hello Flutter

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

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

Exemplo Prático: Criando um Projeto Flutter

Execução do código

Utilize o comando “flutter run” no terminal, estando na raiz do seu projeto, para compilar e executar o aplicativo no emulador ou dispositivo conectado.



Fundamentos do Dart

02

Introdução ao Dart

O Dart é uma linguagem de programação desenvolvida pela Google, projetada para criar aplicativos robustos e de alto desempenho. Com uma sintaxe limpa e uma estrutura orientada a objetos, o Dart é uma escolha popular para o desenvolvimento de aplicativos web, mobile e até mesmo para servidores. E além de tudo é a linguagem padrão do Flutter.



Sintaxe Básica

Para começar, vamos dar uma olhada em um simples "Olá, Mundo!" em Dart:


A dark-themed code editor window titled "Dart" with three window control buttons (minimize, maximize, close) in the top-left corner. The code inside is a simple Dart program that prints "Olá, Mundo!".

```
void main() {  
    print('Olá, Mundo!');  
}
```

Neste exemplo, `void main()` é o ponto de entrada do programa, e `print()` é uma função que exibe texto no console. Simples, não é?

Variáveis e Tipos de Dados

Em Dart, você pode declarar variáveis usando `var`, `dynamic` ou especificando o tipo. Por exemplo:


A code editor window with a dark background and rounded corners. It has three small circular window control buttons (red, yellow, green) in the top-left corner and the title 'Dart' in the top-right corner. The code is written in a light blue/cyan monospace font. It defines a `main` function containing several variable declarations with their types: `String` for `sobrenome`, `int` for `idade`, `double` for `altura`, and `bool` for `isProgramador`.

```
void main() {  
  var nome = 'João';  
  String sobrenome = 'Silva';  
  int idade = 30;  
  double altura = 1.75;  
  bool isProgramador = true;  
}
```

Aqui, temos variáveis para armazenar o nome, sobrenome, idade, altura e se a pessoa é um programador. Dart é uma linguagem tipada, o que significa que você pode especificar o tipo das variáveis, mas também possui inferência de tipos.

Estruturas de Controle

As estruturas de controle em Dart são semelhantes a outras linguagens de programação. Aqui está um exemplo de um loop for:

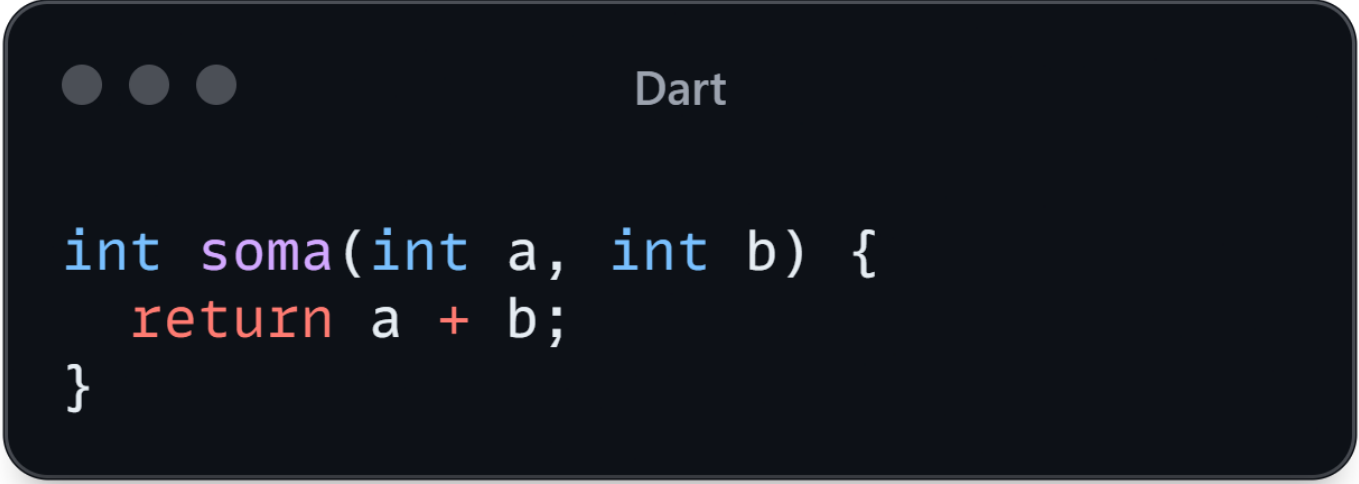
A code editor window with a dark background and rounded corners. At the top, there are three small gray circles on the left and the word 'Dart' in the center. The code is written in a color-coded font: 'void' is purple, 'main()' is blue, '{' is white, 'for' is red, 'var' is red, 'i' is blue, '=' is red, '0' is blue, ';' is red, 'i' is blue, '<' is red, '5' is blue, ';' is red, 'i++' is red, '{' is white, 'print' is purple, 'Número:' is blue, '\$i' is blue, ';' is red, '}' is white, and the final '}' is white.

```
void main() {  
  for (var i = 0; i < 5; i++) {  
    print('Número: $i');  
  }  
}
```

Este loop imprimirá os números de 0 a 4 no console.

Funções

As funções em Dart são declaradas usando a palavra-chave `void` para funções que não retornam valor ou especificando o tipo de retorno para funções que retornam algo. Veja um exemplo:

A code editor window with a dark background and rounded corners. It has three small circular window control buttons (red, yellow, green) in the top-left corner and the title 'Dart' in the top-right corner. The code inside is written in a light blue monospace font. It defines a function named 'soma' that takes two integer parameters, 'a' and 'b', and returns their sum. The return type 'int' is highlighted in purple, the parameter types 'int' are in blue, the function name 'soma' is in light blue, and the 'return' keyword is in orange.

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

Esta função soma aceita dois parâmetros inteiros e retorna sua soma.

Classes e Objetos

Dart é uma linguagem orientada a objetos, o que significa que você pode criar classes e instanciar objetos a partir delas. Aqui está um exemplo:

Dart

```
class Pessoa {  
  String nome;  
  int idade;  
  
  Pessoa(this.nome, this.idade);  
  
  void saudacao() {  
    print('Olá, meu nome é $nome e tenho $idade anos.');  }  
}  
  
void main() {  
  var pessoa = Pessoa('Maria', 25);  
  pessoa.saudacao();  
}
```

Neste exemplo, criamos uma classe Pessoa com dois atributos e um método saudacao para cumprimentar a pessoa.

Widgets e Layouts Básicos

03

Widgets: A Essência da Interface do Usuário

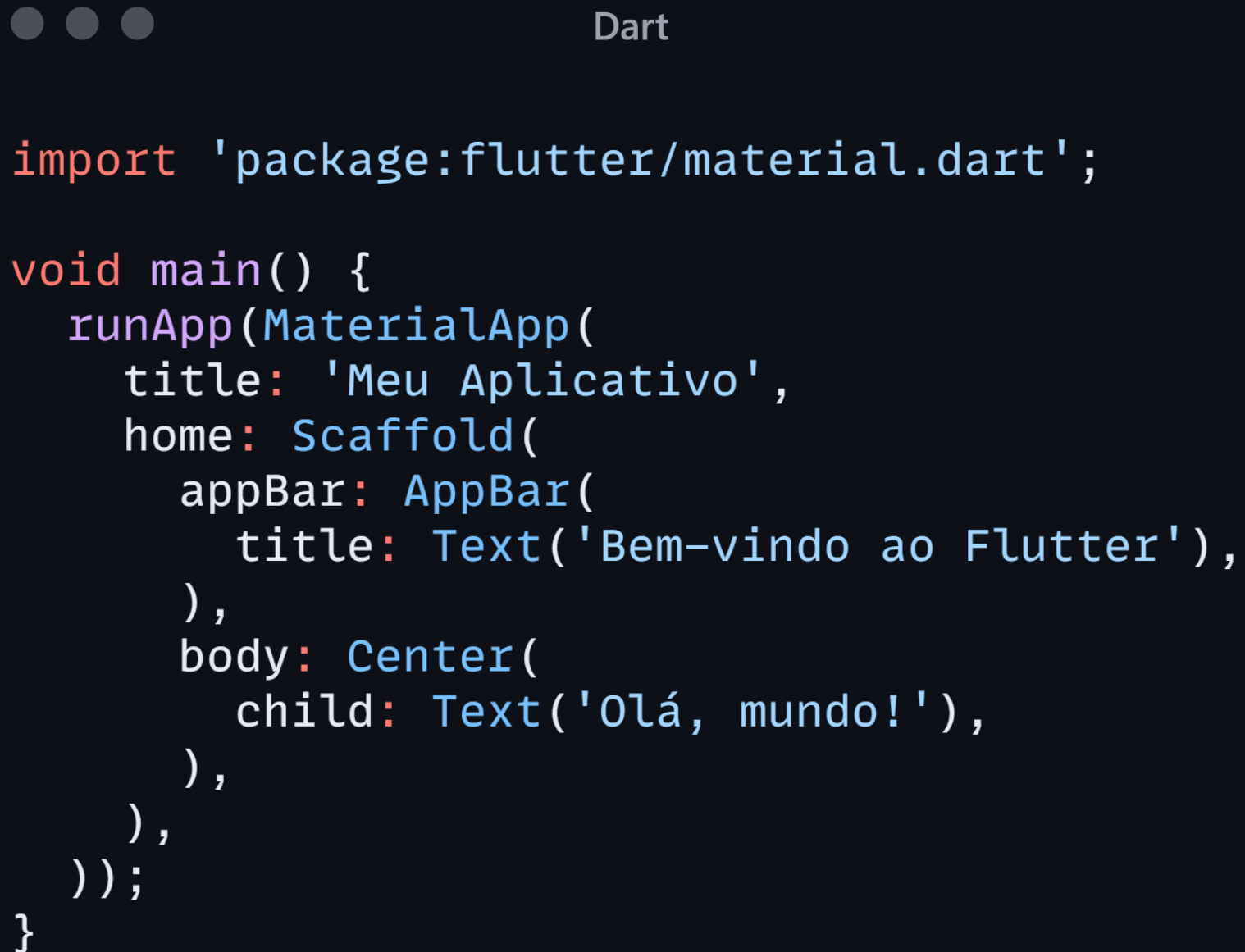
Os widgets são os componentes básicos de qualquer aplicativo Flutter, sendo responsáveis por tudo o que é exibido na tela. Eles variam desde botões simples até elementos mais complexos, como listas roláveis e gráficos interativos. Aqui estão alguns dos widgets mais comuns que você encontrará:

1. `MaterialApp`: O Contêiner Global
2. `Text`: Exibindo Texto na Tela
3. `RaisedButton`: Botões Interativos

Widgets

MaterialApp: O Contêiner Global

O MaterialApp é o widget que define o esqueleto básico de um aplicativo Flutter. Ele configura aspectos importantes, como a barra de título, a navegação e o tema global do aplicativo. Veja um exemplo básico:



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

void main() {
  runApp(MaterialApp(
    title: 'Meu Aplicativo',
    home: Scaffold(
      appBar: AppBar(
        title: Text('Bem-vindo ao Flutter'),
      ),
      body: Center(
        child: Text('Olá, mundo!'),
      ),
    ),
  ));
}
```

Widgets

Text: Exibindo Texto na Tela

O widget Text é usado para exibir texto na tela. Ele pode ser personalizado com diferentes estilos de fonte, tamanhos e cores. Aqui está um exemplo simples:

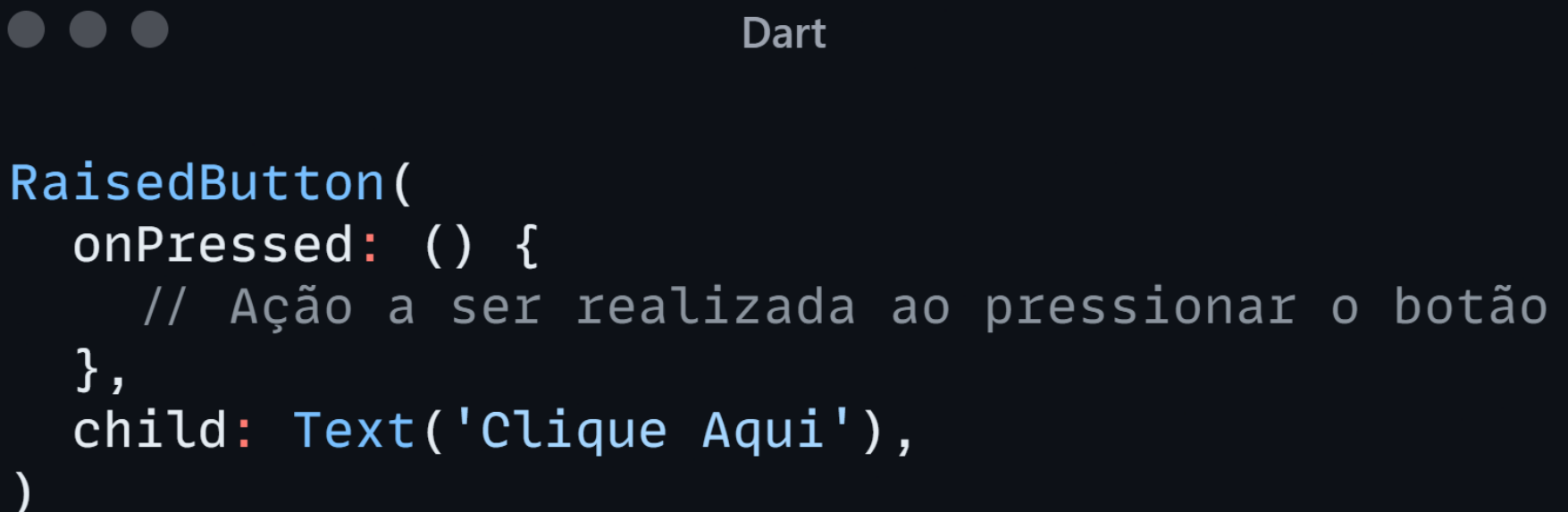
Dart

```
Text(  
  'Este é um texto de exemplo',  
  style: TextStyle(fontSize: 20, fontWeight: FontWeight.bold),  
)
```

Widgets

RaisedButton: Botões Interativos

O RaisedButton é um widget que cria um botão com uma superfície elevada quando pressionado. É frequentemente usado para interações do usuário, como enviar formulários ou acionar ações. Veja como implementar um:



```
RaisedButton(  
  onPressed: () {  
    // Ação a ser realizada ao pressionar o botão  
  },  
  child: Text('Clique Aqui'),  
)
```


Layouts: Organizando Seu Espaço de Trabalho


Os layouts em Flutter determinam como os widgets são organizados e exibidos na tela. Eles permitem criar interfaces de usuário flexíveis e responsivas, adaptáveis a diferentes tamanhos de tela e orientações. Aqui estão alguns layouts básicos:

1. Column: Organização Vertical
2. Row: Organização Horizontal
3. Container: Controle Total

Layouts

Column: Organização Vertical

O layout Column organiza os widgets de forma vertical, um abaixo do outro. É útil para listas verticais e interfaces de usuário que precisam de uma disposição vertical. Veja um exemplo:



```
Column(  
  children: <Widget>[  
    Text('Widget 1'),  
    Text('Widget 2'),  
    Text('Widget 3'),  
  ],  
)
```

Layouts

Row: Organização Horizontal

O layout Row organiza os widgets horizontalmente, um ao lado do outro. É ideal para menus, barras de ferramentas e outras disposições horizontais. Veja como é utilizado:

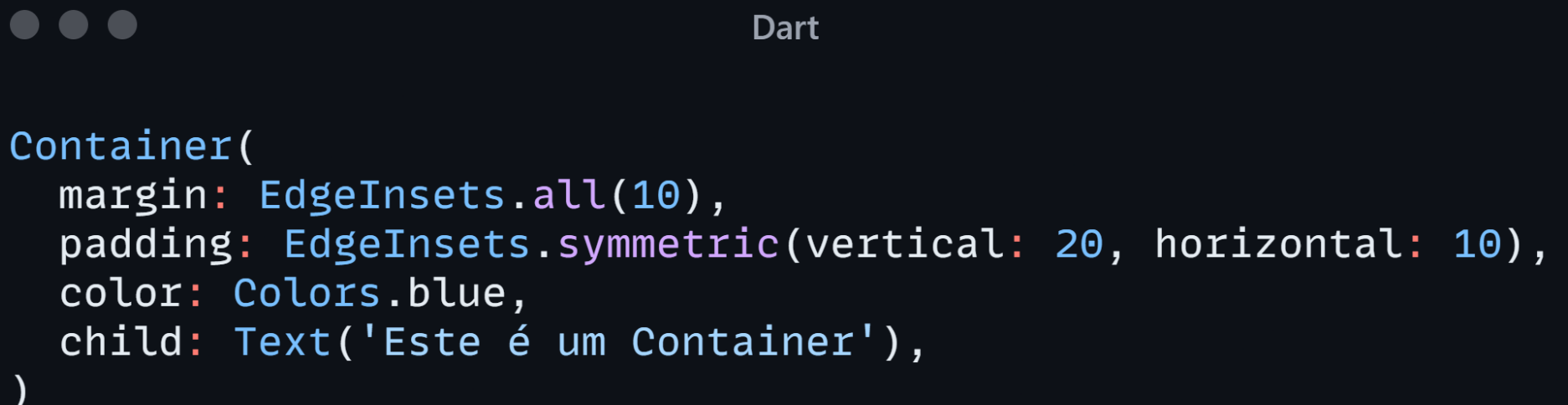


```
Row(  
  children: <Widget>[  
    Icon(Icons.home),  
    Icon(Icons.search),  
    Icon(Icons.notifications),  
  ],  
)
```

Layouts

Container: Controle Total

O widget Container oferece controle total sobre a aparência e o layout de seu filho. Ele pode ser usado para adicionar margens, preenchimento, cor de fundo e muito mais. Aqui está um exemplo simples:



```
Container(  
  margin: EdgeInsets.all(10),  
  padding: EdgeInsets.symmetric(vertical: 20, horizontal: 10),  
  color: Colors.blue,  
  child: Text('Este é um Container'),  
)
```

Gerenciamento de Estado e Navegação

04

Gerenciamento de Estado: A Base do Flutter

No desenvolvimento de aplicativos Flutter, entender e implementar o gerenciamento de estado é crucial para criar interfaces responsivas e dinâmicas. O Flutter oferece diversas opções para gerenciar o estado da sua aplicação, desde simples `StatefulWidget` até soluções mais avançadas como `Provider` ou `Bloc`.

No exemplo, utilizamos `StatefulWidget` para gerenciar o estado do contador. A cada clique no botão flutuante, o estado é atualizado e a interface reflete essa mudança:

```
Dart

import 'package:flutter/material.dart';

class CounterWidget extends StatefulWidget {
  @override
  _CounterWidgetState createState() => _CounterWidgetState();
}
```

```
class _CounterWidgetState extends State<CounterWidget> {
  int _counter = 0;

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Contador'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'Contagem:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Navegação: Conduzindo o Usuário pelo Aplicativo

Navegar entre telas é uma parte essencial da experiência do usuário em qualquer aplicativo móvel. No Flutter, a navegação é facilitada através do uso de rotas e do widget Navigator. Com o Flutter, podemos criar transições fluidas entre telas de forma simples e eficiente.

No exemplo, definimos duas telas usando StatelessWidget e as navegamos utilizando o Navigator. Ao pressionar o botão na tela inicial, o usuário é levado para a tela de detalhes. Para retornar, basta pressionar o botão "Voltar".

```
Dart

import 'package:flutter/material.dart';

void main() {
  runApp(MaterialApp(
    initialRoute: '/',
    routes: {
      '/': (context) => HomeScreen(),
      '/details': (context) => DetailsScreen(),
    },
  ));
}
```

```
class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Home'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pushNamed(context, '/details');
          },
          child: Text('Ir para Detalhes'),
        ),
      ),
    );
  }
}
```

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

Conectividade e Publicação

05

Conectividade: Mantendo-se Conectado com Seus Usuários

Em um mundo cada vez mais conectado, garantir que seu aplicativo Flutter funcione de forma confiável em diferentes condições de rede é essencial. O Flutter fornece uma série de plugins que permitem verificar e reagir à conectividade da rede de forma simples e eficaz.

Neste exemplo, utilizamos o plugin “connectivity” para verificar o estado da conexão com a internet. Dependendo do resultado, exibimos uma mensagem adequada na interface do aplicativo.

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Conectividade'),
        ),
        body: Center(
          child: FutureBuilder<ConnectivityResult>(
            future: Connectivity().checkConnectivity(),
            builder: (context, snapshot) {
              if (snapshot.connectionState == ConnectionState.waiting) {
                return CircularProgressIndicator();
              }

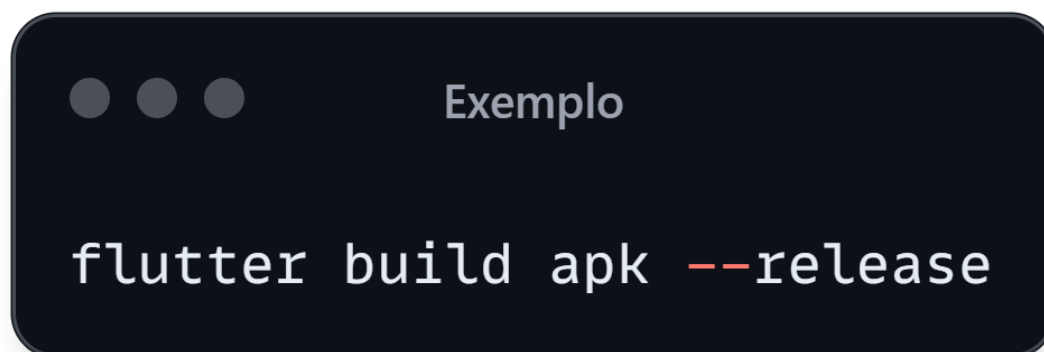
              if (snapshot.hasError) {
                return Text('Erro ao verificar a conexão.');
              }

              if (snapshot.data == ConnectivityResult.none) {
                return Text('Sem conexão com a internet.');
              }

              return Text('Conectado à internet.');
            },
          ),
        ),
      ),
    );
  }
}
```


Publicação: Compartilhando Seu App com o Mundo

Após desenvolver e testar seu aplicativo Flutter, o próximo passo é compartilhá-lo com o mundo. Felizmente, o Flutter simplifica bastante o processo de publicação, permitindo que você compile seu aplicativo para várias plataformas e o envie para as lojas de aplicativos correspondentes.

A dark-themed terminal window with three window control buttons (red, yellow, green) in the top-left corner. The title bar on the right says "Exemplo". The command "flutter build apk --release" is typed in white text.

```
Exemplo  
flutter build apk --release
```

Este comando compila seu aplicativo Flutter para um arquivo APK pronto para distribuição. Após a conclusão, você pode encontrar o arquivo APK na pasta `build/app/outputs/flutter-apk`.

Conclusões

Publicação: Compartilhando Seu App com o Mundo

Parabéns por completar este ebook sobre os fundamentos do Flutter! Esperamos que você tenha adquirido uma compreensão sólida dos conceitos essenciais necessários para iniciar sua jornada como desenvolvedor Flutter.

Lembre-se de que dominar o Flutter é uma jornada contínua. À medida que você se aprofunda no desenvolvimento de aplicativos, você encontrará desafios únicos e novas oportunidades de aprendizado. A prática constante, a exploração de projetos pessoais e a participação na comunidade Flutter são formas excelentes de aprimorar suas habilidades e expandir seu conhecimento.

Como você prossegue em sua jornada de desenvolvimento Flutter, lembre-se de manter-se atualizado com as últimas atualizações e práticas recomendadas. A tecnologia está em constante evolução, e ficar por dentro das novidades garantirá que você esteja sempre no caminho certo para criar aplicativos incríveis.

Desejamos a você muito sucesso em suas futuras empreitadas de desenvolvimento Flutter. Continue explorando, criando e inovando, e estamos ansiosos para ver as incríveis aplicações que você construirá com Flutter!