PROGRAMAÇÃO II

AULA 4

Prof. Elton Masaharu Sato

CONVERSA INICIAL

Nesta etapa, descobriremos como fazer uma navegação de telas de forma a adequar a experiência do nosso usuário. Os temas abordados serão:

- Navegador: como navegar entre telas;
- Rotas Nomeadas: como navegar de forma organizada;
- Transições de Tela: como customizar a transição de tela;
- Widgets de Herói: um widget que sobrevive às transições de rota, voando de uma rota para outra;
- Temas: como customizar o tema de uma tela.

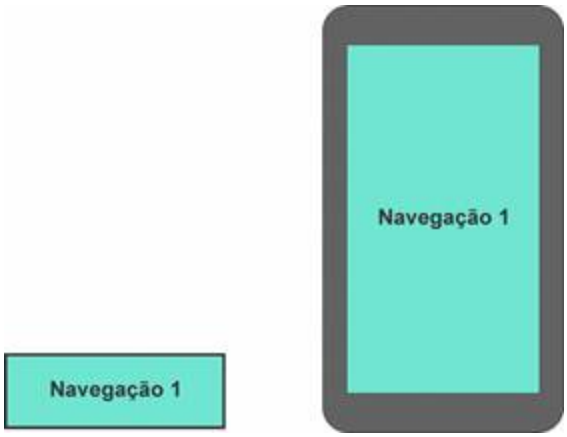
TEMA 1 – NAVEGADOR

Vimos em conteúdos anteriores como estruturar uma tela de um aplicativo Flutter, mas normalmente, um aplicativo é composto por múltiplas telas. Um navegador permite que nos movamos entre telas e é um widget importante, responsável por gerenciar as alterações.

A classe “Navigator” do Flutter permite que façamos a transição de widgets de tela em nosso aplicativo. Para tal, ele realiza o processo de roteamento, utilizando o que chamamos de rotas, terminologia Flutter para páginas e telas do aplicativo. Um navegador de aplicativos funciona em forma de pilha, e quando se navega para uma nova tela, essa nova tela é colocada no topo da pilha. Toda vez que o usuário clica para voltar, é chamada uma função que remove a tela que estiver no topo da pilha.

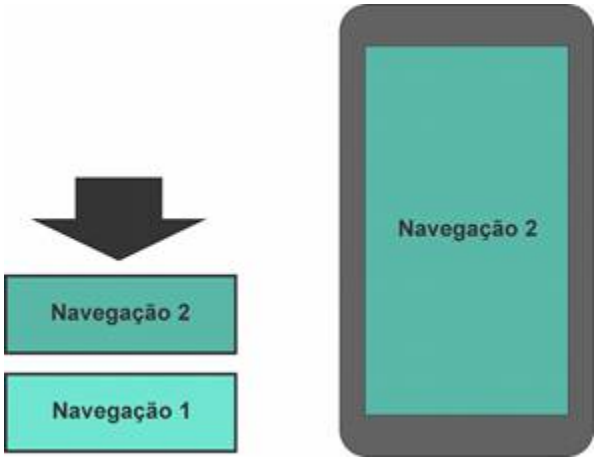
Para realizar essas chamadas de novas telas e remoção de telas, são programadas duas funções, a função “push”, que coloca uma nova rota na pilha, e a função “pop”, que remove a rota do topo da pilha.

Figura 1 – Exemplo de Navegador 1



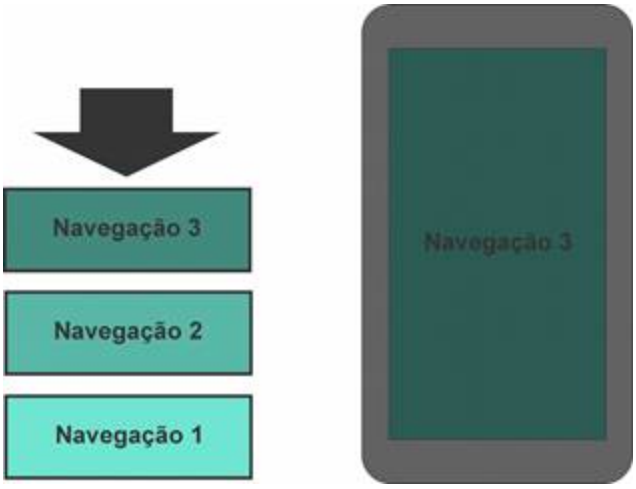
Crédito: Elton Sato.

Figura 2 – Exemplo de navegador 2



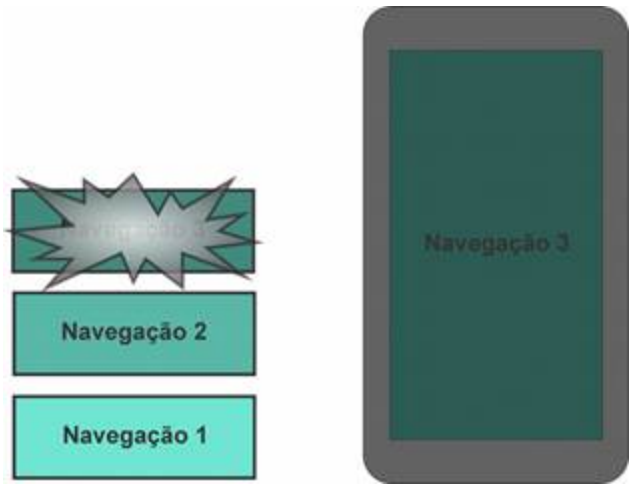
Crédito: Elton Sato.

Figura 3 – Exemplo de Navegador 3



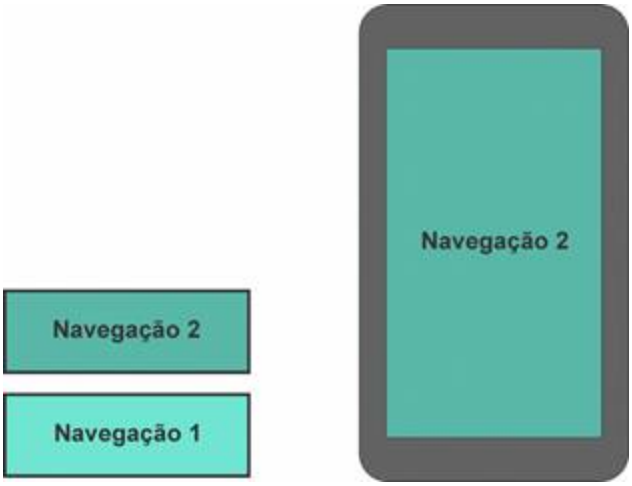
Crédito: Elton Sato.

Figura 4 – Exemplo de Navegador 4



Crédito: Elton Sato.

Figura 5 – Exemplo de Navegador 5



Crédito: Elton Sato.

Nas figuras 1-5, podemos observar a progressão de um aplicativo em andamento.

- Na primeira figura, a tela inicial do aplicativo é a navegação 1;
- Na figura 2, o usuário navegou para uma segunda rota, a navegação 2;
- Em termos de código, o programa efetuou um comando de “push” da navegação 2;
- Na figura 3, acontece algo de forma similar, e o aplicativo se move para a terceira rota – a navegação 3 – por meio de mais um comando de “push”;
- Na figura 4, observamos então o “pop”, um comando que remove a rota que estiver no topo da pilha;
- E, finalmente, na figura 5, podemos observar o resultado do “pop”, no qual voltamos então para a navegação 2, já que agora essa rota é a que está no topo da pilha.

1.1 CÓDIGO DE UM NAVEGADOR

Para utilizar o navegador, basta realizar a chamada do Navigator com as seguintes funções:

- Navigator.push;
- Navigator.pop.

Exemplo de código:

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

void main() {

runApp(const MaterialApp( title: 'Navigation Basics', home: PrimeiraRota(),

));

}

class PrimeiraRota extends StatelessWidget {

const PrimeiraRota({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) {

return Scaffold( appBar: AppBar(

title: const Text('Primeira Rota'),

),

body: Center(

child: ElevatedButton(

child: const Text('Abrir Segunda Rota'), onPressed: () {

Navigator.push( context,

MaterialPageRoute(builder: (context) => const SegundaRota()),
```

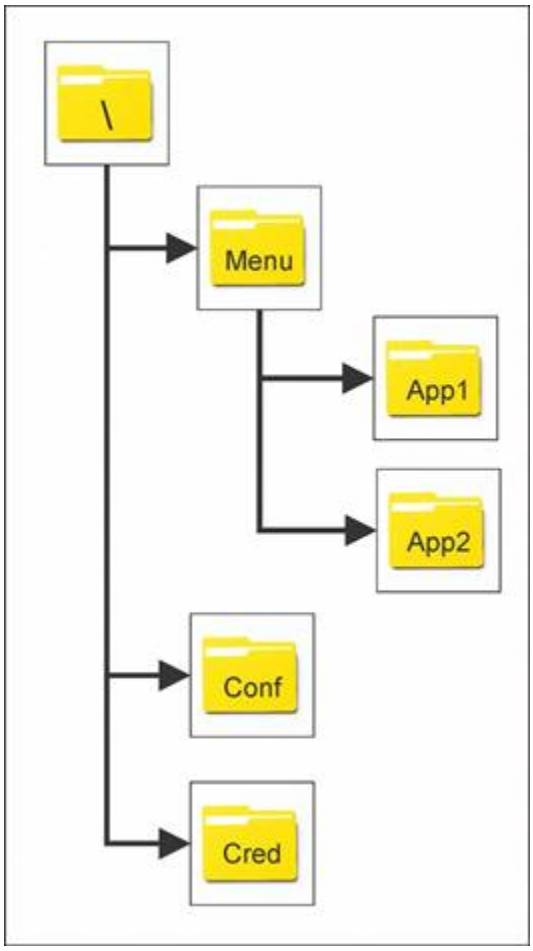
```
);
},
),
),
);
}
}
class SegundaRota extends StatelessWidget {
const SegundaRota({Key? key}) : super(key: key);
@override
Widget build(BuildContext context) { return Scaffold(
appBar: AppBar(
title: const Text('Segunda Rota'),
),
body: Center(
child: ElevatedButton( onPressed: () { Navigator.pop(context);
},
child: const Text('Voltar'),
),
),
);
}
```

```
}

class SecondRoute extends StatelessWidget {

const SecondRoute({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: const Text('Second Route'),

),

body: Center(

child: ElevatedButton( onPressed: () { Navigator.pop(context);

},

child: const Text('Go back!'),

),

),

);

}

}
```

TEMA 2 – ROTAS NOMEADAS

Rotas Nomeadas são uma forma mais organizada de estruturar as suas rotas. Em vez de se rotear manualmente cada tela ou página em cada tela, cria-se um arquivo que trata somente do roteamento.

Figura 6 – Rotas



A figura 6 exemplifica uma possível estrutura de rota nomeada. Esse tipo de estrutura facilita o entendimento de onde cada tela de um aplicativo deve estar e de como ela deve ser acessada. Apesar da aparência em árvore, ela funciona de forma similar a uma árvore de pastas de um computador: embora o caminho mais orgânico seja mover-se da pasta principal para cada subpasta, é possível pular de uma pasta para outra de qualquer lugar da árvore, e, ao clicar em retornar, o explorer volta para a última pasta que foi navegada.

Esse tipo de roteamento é útil quando se precisa navegar para uma mesma tela em diversas partes de seu aplicativo e se deseja evitar a escrita de um código duplicado em várias partes do aplicativo.

Para se trabalhar com rotas nomeadas, utiliza-se a função “Navigator.pushNamed()” para entrar em uma rota nomeada, e o já conhecido “Navigator.pop()” para retornar para uma rota anterior. Exemplificando uma rota nomeada, nós precisaremos de duas etapas importantes antes de realizarmos o “pushNamed” e o seu respectivo “pop”.

Uma delas é termos as telas pelas quais queremos navegar criadas, e a outra é criar um arquivo que defina as rotas e as nomeie. Primeiramente, então, vamos criar duas telas para podemos navegar entre elas:

```
class PrimeiraTela extends StatelessWidget {

const PrimeiraTela({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: const Text('Primeira Tela'),

),

body: Center(

child: ElevatedButton(

// Dentro do widget Primeira Rota onPressed: () {

//TODO: Navega para a segunda rota usando uma rota nomeada

},

child: const Text('Ir para Segunda Rota'),

),

),

);

}

}

class SegundaRota extends StatelessWidget {
```

```
const SegundaRota({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: const Text('Segunda Rota'),

),

body: Center(

child: ElevatedButton(

// Dentro do widget Segunda Rota onPressed: () {

//TODO: Navega para a Primeira Rota usando a função pop da pilha de rotas

},

child: const Text('Voltar'),

),

),

);

}

}
```

A seguir, vamos codificar e nomear as nossas rotas dentro da main:

```
void main() { runApp( MaterialApp(

title: 'Rotas Nomeadas',

// O aplicativo começa na rota nomeada "/". Neste caso, é a primeira rota initialRoute: '/',

routes: {
```

// Quando navega-se para a rota nomeada "/", constrói-se o widget Primeira Tela. '/': (context) => const PrimeiraTela(),

// Quando navega-se para a rota nomeada "/segunda", constrói-se o widget Segunda Tela. '/segunda': (context) => const SegundaRota(),

},

),

);

}

E, finalmente, colocaremos as funções de “pushNamed” dentro da Primeira Rota e “pop” na Segunda Rota:

onPressed: () {

// Navega para a segunda rota usando uma rota nomeada Navigator.pushNamed(context, '/segunda');

},

onPressed: () {

// Navega para a Primeira Rota usando a função pop da pilha de rotas Navigator.pop(context);

},

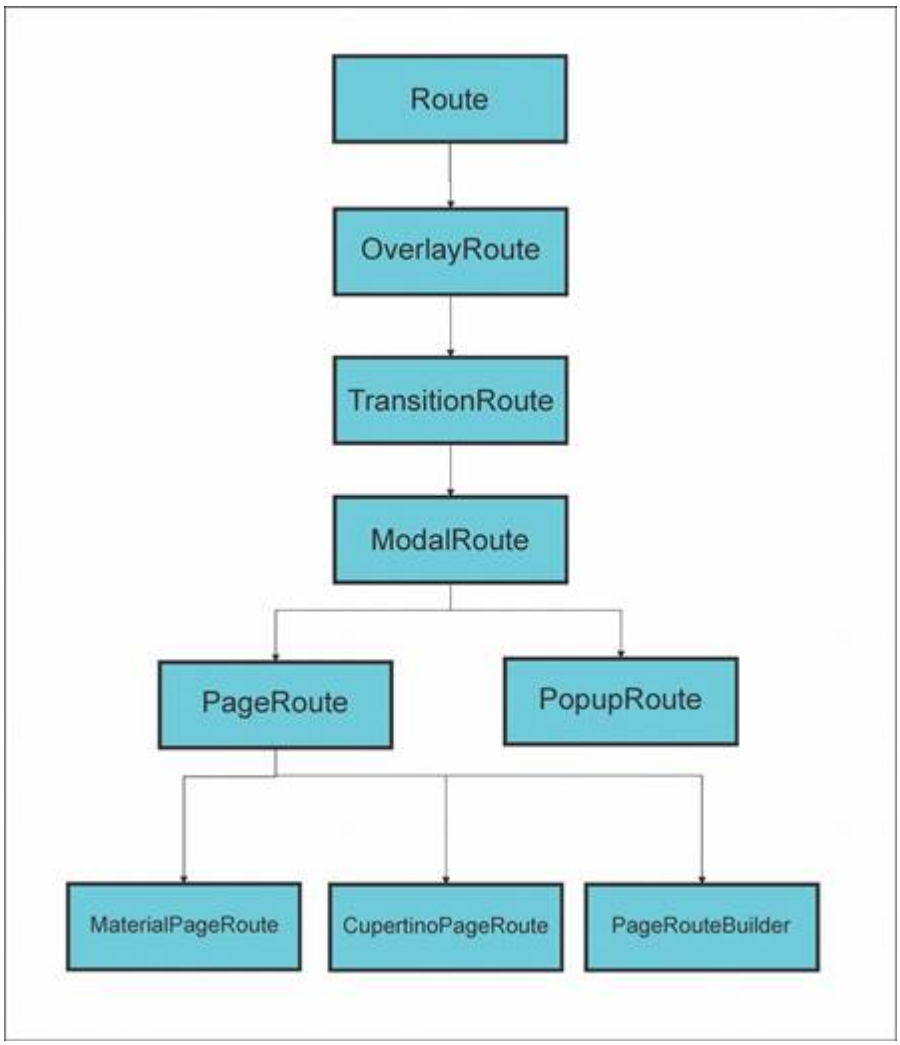
TEMA 3 – TRANSIÇÕES DE TELA

O Navigator controla a navegação entre telas por meio do uso de rotas, dessa forma, podemos utilizar as rotas nomeadas para estruturar e organizar o roteamento de nosso aplicativo.

Para a animação dessa transição entre telas, o Flutter utiliza uma animação padrão que varia se o usuário estiver rodando o aplicativo em Android ou em iOS.

Em Android, a animação de transição de telas é a tela nova vindo de baixo para cima, com sua opacidade aumentando gradualmente até que a nova tela ocupe toda a tela, e com opacidade em 100% para sair de uma tela usando a função “pop”, o contrário acontece. Já em iOS, uma nova tela desliza da direita para a esquerda, e, na saída, ele faz o contrário.

Figura 7 – Hierarquia de Classes de PageRouteBuilder



Crédito: Elton Sato.

A figura 7 mostra a hierarquia de classes para a geração de uma classe do tipo PageRouteBuilder. No parágrafo anterior, vimos como o Material Page Route (Android) e o Cupertino Page Route (iOS) operam, mas veremos agora como podemos criar a nossa própria transição de rota utilizando o Page Route Builder.

Para criar uma animação customizada, certifique-se de fazer o seguinte:

- Criar e definir um PageRouteBuilder;
- Criar um Tween por meio de um TransitionsBuilder;
- Use um AnimatedWidget (Widget animado).

3.1 CRIAR E DEFINIR UM PAGEROUTEBUILDER

Primeiro, nós precisaremos de uma Page Route Builder, e duas funções são importantes, uma é o pageBuilder, que constrói a próxima página à qual se deseja transitar, e a outra é o transitionsBuilder, que construirá a animação de transição.

Route _criarRota() {

return PageRouteBuilder(

pageBuilder: (context, animation, secondaryAnimation) => const Tela2(), transitionsBuilder: (context, animation, secondaryAnimation, child) {

return child

},

);

}

3.2 CRIAR UM TWEEN POR MEIO DE UM TRANSITIONSBUILDER

Depois, vamos criar um Tween com a ajuda de um TransitionsBuilder, o qual definirá alguns parâmetros da animação, como os pontos de início e fim da animação. O Offset inicial determina em que ponto começa a animação, sendo a primeira variável o deslocamento horizontal e a segunda variável o deslocamento vertical.

transitionsBuilder: (context, animation, secondaryAnimation, child) { const begin = Offset(0.0, 1.0);

const end = Offset.zero; const curve = Curves.ease;

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

return child

);

}

3.3 USE UM ANIMATEDWIDGET

Dentro do nosso TransitionBuilder, devemos escolher o tipo de animação que será utilizada. Nesse exemplo, utilizaremos um SlideTransition, que será o retorno da função de cima, substituindo a linha “return child”.

return SlideTransition(

position: animation.drive(tween), child: child,

);

3.4 EXEMPLO FINAL

Juntando os códigos, teremos o exemplo a seguir:

Route _criarRota() { return PageRouteBuilder(

pageBuilder: (context, animation, secondaryAnimation) => const Tela2(), transitionsBuilder: (context, animation, secondaryAnimation, child) { const begin = Offset(0.0, 1.0);

const end = Offset.zero; const curve = Curves.ease;

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

return SlideTransition(

position: animation.drive(tween),

child: child,

);

```
},
);
}
```

Para utilizar o código, basta utilizar o Navigator.push da seguinte forma:

```
Navigator.of(context).push(_criarRota());
```

TEMA 4 – WIDGET DE HERÓI

Um widget de Herói refere-se a um Widget que “voa” entre telas. É possível criar uma animação de herói utilizando um widget do Flutter chamado de “Herói”.

Um dos pontos mais interessantes de um widget de herói é a sua capacidade de alterar a forma do widget, de um círculo para um quadrado, por exemplo. Esse tipo de animação é bastante comum em aplicativos, e as chances são de que vários de seus aplicativos favoritos implementem uma animação dessas de alguma forma, pois ele é ideal para uma transição suave de qualquer elemento na tela para ficar em tela cheia.

Outra utilidade que permite uma transição suave é a sua capacidade de mover um elemento de uma tela para outra, como quando se clica em um produto em uma tela, a qual move o produto para uma nova tela com uma descrição mais detalhada do produto e um botão de compra.

Um widget de herói pode também ser chamado de widget de herói radial quando o foco está na alteração de forma do widget. Tanto a sua forma padrão quanto a radial utilizam a mesma estrutura básica. Animações de herói são implementadas utilizando dois widgets de herói, um descrevendo o widget na rota de origem e outro na rota de destino do herói. A estrutura da animação de herói segue os seguintes passos:

1. Defina o widget de herói inicial. Ele será chamado no código de “source hero” ou “herói fonte”, em português. O herói especifica sua representação gráfica na tela e um tag de identificação;
2. Defina o widget de herói final. Ele será chamado no código de “destination hero” ou “herói destino”, em português. Assim como o widget de herói anterior, esse também especifica sua

representação gráfica na tela e um tag de identificação. É importante notar que ambos os widgets devem possuir a mesma tag de identificação;

3. Crie uma rota que contenha o widget de herói destino;
4. Inicie a animação fazendo um Navigator.push da rota de destino. Um Navigator.pop fará o papel de voltar um widget de herói.

A figura 8 apresenta um exemplo de Animação de Herói, mostrando como um widget pode se mover de uma rota para outra e alterar o seu formato durante o processo.

Figura 8 – Widget de Herói

Crédito: Elton Sato.

4.1 CÓDIGO DE EXEMPLO DE UM WIDGET DE HERÓI

O código a seguir executa um aplicativo igual ao mostrado na figura 8:

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

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

}
```

```
class MyApp extends StatelessWidget {

const MyApp({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) { return const MaterialApp( debugShowCheckedModeBanner: false, home: TextScreen(),

);

}

}

class TextScreen extends StatelessWidget {

const TextScreen({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: const Text('Animação de Herói'),

),

body: Container(

alignment: Alignment.bottomCenter, child: GestureDetector(

onTap: () { Navigator.of(context).push(

MaterialPageRoute(builder: (context) => const SecondPage()));

},

child: const Hero( tag: "hero1",

child: CircleAvatar( radius: 70,
```

```
backgroundImage: NetworkImage(
'https://upload.wikimedia.org/wikipedia/commons/thumb/d/df/The_%E2%80%9CRover
%E2%80%9D_Safety_Bicycle.svg/640px-The_%E2%80%9CRover
%E2%80%9D_Safety_Bicycle.svg.png')),
)),
),
);
}
}
class SecondPage extends StatefulWidget {
const SecondPage({Key? key}) : super(key: key);
@override
_SecondPageState createState() => _SecondPageState();
}
class _SecondPageState extends State<SecondPage> { @override
Widget build(BuildContext context) { return Scaffold(
appBar: AppBar(
title: const Text("Segunda Tela"),
),
body: Hero( tag: "hero1",
child: Container(
alignment: Alignment.topCenter, width: double.infinity,
```

```
height: 300,

decoration: const BoxDecoration( image: DecorationImage( image: NetworkImage(

'https://upload.wikimedia.org/wikipedia/commons/thumb/d/df/The_%E2%80%9CRover

%E2%80%9D_Safety_Bicycle.svg/640px-The_%E2%80%9CRover

%E2%80%9D_Safety_Bicycle.svg.png'),

)),

),

));

}

}
```

TEMA 5 – TEMAS

Desenvolver um aplicativo é mais que só funcionalidades, pois a experiência do usuário é bastante importante. É por esse motivo que o Flutter se desassocia dos detalhes das especificações da interface e implementa múltiplas plataformas com um único código, como o Material Design e o Cupertino iOS. Além disso, o Flutter possui widgets exclusivos para trabalhar com a experiência do usuário, e um dos mais importantes para customizar a experiência de seu aplicativo é o widget de tema.

Utilizando um widget de tema, podemos compartilhar esquemas de cores e estilos de fontes entre diferentes partes de nosso aplicativo, ou em nosso aplicativo como um todo, o que dependerá somente do ponto da nossa árvore de widgets em que será implementado o tema.

O código básico de um Tema é escrito da seguinte forma:

```
theme: ThemeData(

propriedade: valor propriedade: valor propriedade: valor(
```

subpropriedade: valor subpropriedade: valor

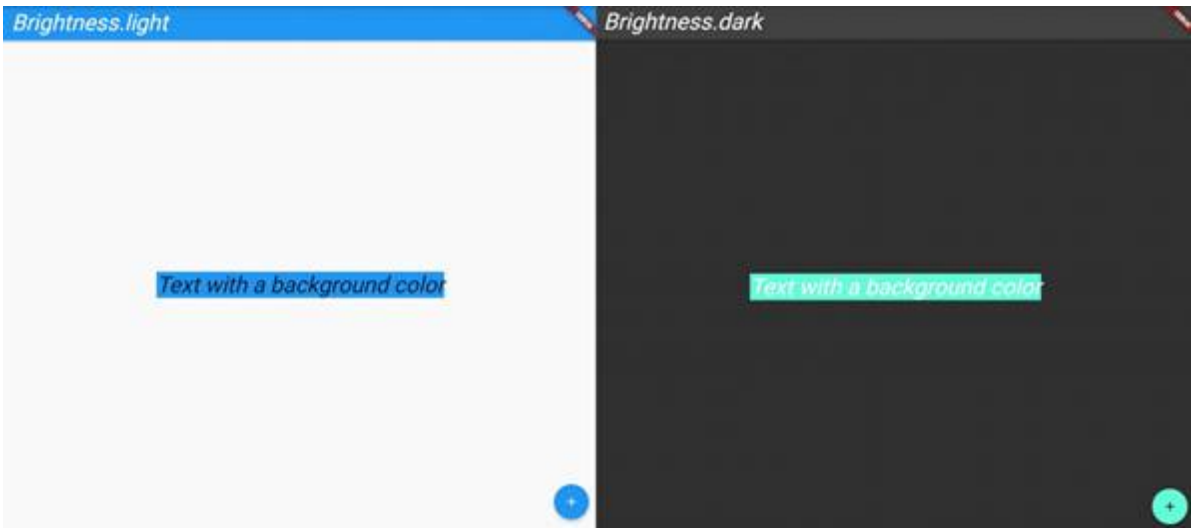
)

)

Existe uma enorme quantidade de propriedades possíveis para serem alteradas. Nesta etapa, abordamos apenas as mais utilizadas. Para acessar uma lista completa, visite: <https://api.flutter.dev/flutter/material/ThemeData-class.html>. Acesso em: 27 jun. 2022.

5.1 BRIGHTNESS

Figura 9 – Modo de Brilho



Utilizando a propriedade *brightness*, é possível alterar o modo de brilho da tela utilizando os valores “brightness.light” ou “brightness.dark”, assim como demonstrado pela figura 9.

5.2 CORES

É possível alterar as cores do theme por meio do uso da propriedade de esquema de cores, assim como demonstrado pelo código a seguir:

colorScheme: ColorScheme.fromSwatch(primarySwatch: Colors.blue,

).copyWith(

secondary: Colors.green,

),

Uma lista das cores padrões disponíveis é encontrada no site: <https://api.flutter.dev/flutter/material/Colors-class.html>. Acesso em: 27 jun. 2022.

5.3 TEMAS DE TEXTO

Figura 10 – Temas de texto

Crédito: Elton Sato.

Dentro do tema, o desenvolvedor pode deixar registrada uma variedade de temas diferentes de texto para serem utilizados pelos widgets. Ele funciona de forma similar aos estilos de texto do Microsoft Word.

Código:

textTheme: const TextTheme(

nomeDoEstilo: TextStyle(fontSize: 10.0, FontWeight: FontWeight.bold), nomeDoEstilo2: TextStyle(fontSize: 48.0, fontFamily: 'Hind'),

)

FINALIZANDO

Esta etapa focou nos assuntos referentes à navegação do aplicativo e à experiência do usuário. Vimos como fazer uma navegação entre telas de forma padrão, customizar ambas as nossas rotas quanto à transição de rotas, além de criar rotas nomeadas e widgets que podem se mover de uma rota para outra.

REFERÊNCIAS

COLORS class. Flutter. Disponível em: <https://api.flutter.dev/flutter/material/Colors-class.html>. Acesso em: 26 jun. 2022.

THEME data class. Flutter. Disponível em: <https://api.flutter.dev/flutter/material/ThemeData-class.html>. Acesso em: 26 jun. 2022.