Anexo 7 – Interface:

Principais Widgets

Vamos explorar alguns dos principais widgets do Flutter, detalhando suas características e fornecendo exemplos de uso.

Scaffold

O Scaffold é um dos widgets fundamentais no Flutter e desempenha um papel central na construção de interfaces de usuário. Ele fornece a estrutura básica de uma página visual, incluindo elementos como barra de aplicativo (AppBar), corpo da página, gaveta lateral (Drawer), barra de navegação inferior (BottomNavigationBar), entre outros. Aqui estão alguns dos componentes mais comuns dentro de um Scaffold:

AppBar:

O AppBar é uma barra de aplicativo localizada na parte superior da tela.

Pode conter um título, ações (botões) e outros elementos, como abas.

Exemplo:

```
appBar: AppBar(
 title: Text('Minha Página'),
 actions: [
  IconButton(
   icon: Icon(Icons.search),
   onPressed: () {
    // Ação ao pressionar o ícone de pesquisa
   },
  ),
 ],
),
```

Corpo da Página (Body):

O corpo da página é o local onde você coloca o conteúdo principal da tela.

Pode conter uma variedade de widgets, como Column, ListView, Container, etc.

Exemplo:

```
body: Column(
 children: [
  Text('Conteúdo da Página'),
  // Outros widgets aqui...
 ],
),
```

Gaveta Lateral (Drawer):

O Drawer é um menu lateral que pode ser acessado deslizando da borda esquerda da tela ou tocando em um ícone de menu.

Geralmente usado para fornecer navegação entre diferentes seções do aplicativo.

Exemplo:

```
drawer: Drawer(
 child: ListView(
  children: [
   ListTile(
    title: Text('Página Principal'),
    onTap: () {
     // Ação ao tocar na opção do Drawer
    },
   ),
   // Outras opções de navegação...
  ],
 ),
),
```

Barra de Navegação Inferior (BottomNavigationBar):

A BottomNavigationBar é uma barra na parte inferior da tela que permite a navegação entre diferentes telas ou seções do aplicativo.

Cada item na barra geralmente representa uma página diferente.

Exemplo:

```
bottomNavigationBar: BottomNavigationBar(
 items: [
  BottomNavigationBarItem(
   icon: Icon(Icons.home),
   label: 'Home',
  ),
  BottomNavigationBarItem(
   icon: Icon(Icons.settings),
   label: 'Configurações',
  ),
 ],
 // Lógica para lidar com a navegação entre as páginas
),
```

O Scaffold simplifica a criação de interfaces de usuário padrão, fornecendo uma estrutura predefinida e consistente para as páginas do seu aplicativo. No entanto, é altamente personalizável, permitindo que você ajuste cada parte conforme necessário para atender aos requisitos específicos da sua aplicação.

Container:

O Container é um widget versátil que pode conter outros widgets e aplicar decorações, margens, preenchimentos, entre outros.

Exemplo de Uso:

```
Container(
 width: 200,
 height: 100,
 color: Colors.blue,
```

```
  child: Text('Conteúdo do Container'),
)
```

Row e Column:

Os widgets Row e Column são usados para organizar outros widgets em uma linha ou coluna, respectivamente.

Exemplo de Uso:

```
Row(
  children: [
    Icon(Icons.star),
    Text('Avaliação: 4.5'),
  ],
)
```

```
Column(
  children: [
    Text('Item 1'),
    Text('Item 2'),
    Text('Item 3'),
  ],
)
```

ListView:

O ListView é usado para criar uma lista rolável de widgets, seja na horizontal ou na vertical.

Exemplo de Uso:

```
ListView(
  children: [
    ListTile(title: Text('Item 1')),
    ListTile(title: Text('Item 2')),
```

```
  ListTile(title: Text('Item 3')),
 ],
)
```

Stack:

O Stack permite sobrepor widgets, permitindo um posicionamento preciso na tela.

Exemplo de Uso:

```
Stack(
 children: [
  Positioned(
   left: 10,
   top: 10,
   child: Text('Texto na posição (10, 10)'),
  ),
  Positioned(
   right: 10,
   bottom: 10,
   child: Text('Texto na posição (direita, abaixo)'),
  ),
 ],
)
```

TextField:

O TextField é usado para obter entrada de texto do usuário.

Exemplo de Uso:

```
TextField(
 decoration: InputDecoration(
  labelText: 'Digite aqui',
  hintText: 'Texto de ajuda',
 ),
```

)

ElevatedButton e TextButton:

Esses widgets criam botões interativos.

Exemplo de Uso:

ElevatedButton(onPressed: () {}, child: Text('Botão Elevado'))

TextButton(onPressed: () {}, child: Text('Botão de Texto'))

Image:

O Image é usado para exibir imagens.

Exemplo de Uso:

Image.network('https://exemplo.com/imagem.jpg')

Image.asset('assets/imagem_local.png')

ListView.builder:

Este é um construtor especial do ListView que permite criar itens de maneira eficiente sob demanda.

Exemplo de Uso:

```
ListView.builder(
  itemCount: 100,
  itemBuilder: (context, index) {
    return ListTile(title: Text('Item $index'));
  },
)
```

FutureBuilder:

O FutureBuilder é usado para construir widgets com base no resultado de uma operação assíncrona (como uma chamada de API).

Exemplo de Uso:

```
FutureBuilder<String>(
 future: fetchUserData(),
 builder: (context, snapshot) {
  if (snapshot.connectionState == ConnectionState.done) {
   return Text('Dados: ${snapshot.data}');
  } else if (snapshot.hasError) {
   return Text('Erro: ${snapshot.error}');
  } else {
   return CircularProgressIndicator();
  }
 },
)
```

Estes são apenas alguns dos muitos widgets disponíveis no Flutter. A combinação e personalização destes widgets permitem a construção de interfaces ricas e interativas.

Posicionamento e Alinhamento:

Vamos explorar mais detalhadamente os conceitos de Posicionamento e Alinhamento no Flutter, destacando os widgets Align, Padding e Margin, bem como o uso de Positioned.

Align:

O widget Align é utilizado para alinhar um widget filho dentro de um pai, especificando a posição relativa desse widget em relação ao tamanho do pai. A propriedade alignment é crucial para definir como o widget filho deve ser alinhado.

Exemplo de Uso:

```
Align(
```

alignment: Alignment.topRight,

child: Text('Conteúdo alinhado no canto superior direito'),

)

Neste exemplo, o texto é alinhado no canto superior direito do pai.

Padding e Margin:

Ambos Padding e Margin são usados para ajustar o espaço ao redor de um widget. A diferença principal é que Padding afeta o espaço interno do widget, enquanto Margin afeta o espaço externo em relação a outros widgets.

Exemplo de Uso:

Padding(

padding: EdgeInsets.all(16.0),

child: Container(

color: Colors.blue,

child: Text('Conteúdo com espaçamento interno de 16 pixels'),

),

)

Container(

margin: EdgeInsets.all(16.0),

color: Colors.blue,

child: Text('Conteúdo com margem externa de 16 pixels'),

)

Positioned:

O widget Positioned é comumente usado dentro de um Stack e é utilizado para posicionar um widget filho de forma absoluta em relação à borda do Stack.

Exemplo de Uso:

Stack(

children: [

```
  Positioned(
    top: 10.0,
    left: 20.0,
    child: Text('Posicionado em (20, 10)'),
  ),
  Positioned(
    bottom: 0.0,
    right: 0.0,
    child: Text('Posicionado no canto inferior direito'),
  ),
 ],
)
```

No exemplo acima, o primeiro texto é posicionado em (20, 10) pixels do canto superior esquerdo do Stack, e o segundo texto é posicionado no canto inferior direito.

Considerações Importantes:

Os valores para padding e margin podem ser definidos de várias maneiras, incluindo EdgeInsets.all, EdgeInsets.symmetric e EdgeInsets.only.

As propriedades de alinhamento, como Alignment, EdgeInsets, e Positioned, aceitam valores proporcionais, pontos percentuais e valores fixos.

O uso desses widgets permite criar layouts flexíveis e responsivos, adaptando-se a diferentes tamanhos de tela e dispositivos.

A prática e a experimentação são fundamentais para compreender plenamente como esses widgets afetam o layout do seu aplicativo.

Esses conceitos são essenciais para a construção de interfaces ricas e bem organizadas no Flutter, proporcionando controle preciso sobre a posição e o espaço dos elementos na tela.

Unidades de Medida

Vamos explorar mais a fundo os conceitos de Unidades de Medida no contexto do Flutter:

Pixels:

O pixel é uma unidade de medida fundamental em design gráfico e desenvolvimento de interfaces. Em Flutter, as dimensões, como largura (width) e altura (height), podem ser definidas em pixels. Quando você especifica dimensões em pixels, está indicando a quantidade exata de pixels que um elemento deve ocupar na tela.

Exemplo de Uso:

```
Container(
  width: 200.0,
  height: 100.0,
  color: Colors.blue,
  child: Text('Este container tem 200 pixels de largura e 100 pixels de altura.'),
)
```

Porcentagem:

Ao criar layouts responsivos, é frequentemente útil usar porcentagens em relação ao tamanho da tela. Embora o Flutter não tenha uma unidade de porcentagem diretamente, você pode calcular dinamicamente o tamanho em relação à largura ou altura da tela usando o widget MediaQuery.

Exemplo de Uso:

```
Container(
  width: MediaQuery.of(context).size.width * 0.5,
  height: MediaQuery.of(context).size.height * 0.3,
  color: Colors.green,
  child: Text('Este container ocupa 50% da largura e 30% da altura da tela.'),
)
```

Device-Independent Layout:

Flutter utiliza unidades de medida independentes de dispositivo para garantir uma experiência consistente em diferentes densidades de pixels e tamanhos de tela. Em vez de depender diretamente de pixels físicos, o Flutter usa logical pixels. Essas unidades são proporcionais à densidade de pixels do dispositivo, proporcionando uma aparência consistente em vários dispositivos.

Exemplo de Uso:

```
Container(
  width: 100.0,
  height: 50.0,
  color: Colors.red,
  child: Text('Este container tem 100 unidades lógicas de largura e 50 unidades lógicas de altura.'),
)
```

Ao usar unidades independentes de dispositivo, o Flutter ajustará automaticamente o tamanho dos elementos para garantir uma aparência consistente em diferentes dispositivos, independentemente da densidade de pixels.

Observações:

O uso de pixels pode ser adequado para elementos que precisam de um tamanho específico e não devem variar com o tamanho da tela.

As porcentagens são frequentemente utilizadas para criar layouts responsivos, adaptando-se dinamicamente ao tamanho da tela.

As unidades independentes de dispositivo são recomendadas para garantir que seu aplicativo tenha uma aparência consistente em uma variedade de dispositivos.

Ao combinar essas unidades de medida, você terá um controle flexível sobre o layout do seu aplicativo Flutter em diferentes cenários e dispositivos.

MediaQuery

O widget MediaQuery no Flutter é uma ferramenta poderosa que permite ao desenvolvedor obter informações sobre o ambiente de exibição do dispositivo, como tamanho da tela, orientação e densidade de pixels. Essas informações são cruciais para criar layouts responsivos que se ajustam a diferentes dispositivos. Vamos explorar o conceito e realizar demonstrações práticas do uso do MediaQuery.

Conceito:

O MediaQuery é um widget que fornece informações sobre o ambiente de exibição. Ele é geralmente utilizado para acessar a MediaQueryData, que contém informações como:

- size: Tamanho da tela (largura e altura).

- devicePixelRatio: Densidade de pixels do dispositivo.
- orientation: Orientação da tela (horizontal ou vertical).

E muitas outras propriedades úteis.

Demonstração Prática:

Vamos criar uma demonstração prática que utiliza MediaQuery para tornar um layout responsivo. Neste exemplo, ajustaremos o tamanho do texto com base na largura da tela.

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

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

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  // Obtém informações sobre o ambiente de exibição
  MediaQueryData mediaQueryData = MediaQuery.of(context);

  // Determina a largura da tela
  double screenWidth = mediaQueryData.size.width;

  // Calcula o tamanho do texto com base na largura da tela
  double textSize = screenWidth * 0.05;

  return MaterialApp(
   home: Scaffold(
    appBar: AppBar(
     title: Text('Demo MediaQuery'),
    ),
    body: Center(
```

```
        child: Container(
          width: screenWidth * 0.8,
          padding: EdgeInsets.all(16.0),
          color: Colors.lightBlue,
          child: Text(
            'Texto Responsivo',
            style: TextStyle(fontSize: textSize),
            textAlign: TextAlign.center,
          ),
        ),
      ),
    ),
  );
 }
}
```

Neste exemplo, estamos utilizando MediaQuery para obter informações sobre a tela. Em seguida, calculamos dinamicamente o tamanho do texto com base na largura da tela. Isso torna o texto responsivo, ajustando-se automaticamente ao tamanho da tela.

Algumas Aplicações Práticas:

- Layouts Responsivos: Ajustar dinamicamente o layout com base no tamanho da tela.
- Tipografia Responsiva: Modificar tamanhos de fonte para garantir uma boa legibilidade em diferentes dispositivos.
- Imagens Responsivas: Carregar diferentes resoluções de imagens com base na densidade de pixels do dispositivo.
- Personalização Dinâmica: Adaptar a aparência do aplicativo com base nas características do dispositivo.

Exemplo de Uso:

```
Container(
 margin: EdgeInsets.all(16.0),
 padding: EdgeInsets.symmetric(horizontal: 20.0, vertical: 10.0),
 width: MediaQuery.of(context).size.width * 0.8,
```

```
 child: Row(
  mainAxisAlignment: MainAxisAlignment.spaceBetween,
  children: [
   Icon(Icons.arrow_back),
   Text('Título'),
   Icon(Icons.settings),
  ],
 ),
)
```

O uso inteligente do MediaQuery permite criar aplicativos Flutter que oferecem uma experiência consistente e agradável em uma variedade de dispositivos. Experimente ajustar diferentes propriedades com base nas informações fornecidas por MediaQuery para atender aos requisitos específicos do seu aplicativo.

Lista de Exercícios

Exercícios sobre Montagem de Layout em Dart:

Layout Básico com Container:

Crie um aplicativo Flutter que utilize o widget Container para criar um layout básico.

Adicione pelo menos três widgets filhos dentro do Container com diferentes estilos e cores.

Organização com Row e Column:

Desenvolva uma interface que faça uso dos widgets Row e Column para organizar elementos de forma horizontal e vertical.

Adicione diversos widgets (como Text, Icon e Image) para demonstrar a organização.

Página com ListView:

Construa uma página que contenha um ListView com pelo menos 10 itens.

Cada item deve ser representado por um card simples, exibindo informações fictícias.

Barra de Navegação Personalizada:

Implemente uma barra de navegação personalizada, utilizando o widget AppBar e ícones.

Crie diferentes abas que levem a seções distintas da sua aplicação.

Formulário de Contato:

Crie um formulário de contato com campos para nome, e-mail e mensagem.

Utilize o widget TextField para entrada de dados e adicione um botão para enviar o formulário.

Layout Responsivo com MediaQuery:

Torne o seu layout responsivo utilizando o widget MediaQuery.

Adapte o conteúdo para diferentes tamanhos de tela e orientações.

Menu de Opções (Drawer):

Implemente um menu lateral (Drawer) contendo diversas opções.

Configure o menu para ser acessado através de um botão na barra de aplicativo.

Criação de Cards:

Desenvolva uma tela que utilize o widget Card para exibir informações de produtos fictícios.

Os cards devem conter uma imagem, título e descrição.

Layout com Abas (TabBar):

Crie uma página que faça uso do widget TabBar para organizar conteúdo em diferentes abas.

Adicione conteúdo exclusivo para cada aba.

Interface com Animação:

Integre animações simples em elementos do seu layout.

Por exemplo, faça um widget mover-se ou mudar de cor ao ser clicado.

Estes exercícios fornecerão uma prática sólida na montagem de layouts em Dart, utilizando os principais widgets do Flutter.

Exercícios sobre Organização de Interface Gráfica em Flutter:

Layout Básico:

Crie um aplicativo Flutter com um layout básico contendo um Container, uma Column e uma Row.

Adicione alguns widgets (por exemplo, Text, Icon, Image) dentro desses layouts para criar uma estrutura visual simples.

Listagem Dinâmica:

Desenvolva um aplicativo que utilize o widget ListView.builder para exibir uma lista dinâmica de itens.

Os itens podem ser representados por widgets simples, como Card, contendo informações fictícias.

Barra de Navegação:

Implemente uma barra de navegação usando o widget BottomNavigationBar ou TabBar.

Crie diferentes telas para cada item da barra de navegação, demonstrando a transição entre elas.

Formulário de Cadastro:

Construa um formulário de cadastro com campos como nome, e-mail e senha.

Utilize o widget TextField para entrada de dados e adicione um botão para submeter o formulário.

Layout Responsivo:

Adapte o layout do seu aplicativo para ser responsivo, considerando diferentes tamanhos de tela.

Utilize o widget MediaQuery para ajustar dinamicamente o conteúdo com base nas características do dispositivo.

Menu Deslizante (Drawer):

Implemente um menu deslizante lateral (Drawer) em seu aplicativo.

Adicione itens de menu que direcionam o usuário para diferentes telas ou executam ações específicas.

Uso do Stack:

Explore o uso do widget Stack para sobrepor widgets na tela.

Crie um exemplo com dois ou mais widgets sobrepostos e ajuste suas posições.

Integração de Imagens:

Desenvolva uma aplicação que faça uso do widget Image para exibir imagens.

Experimente carregar imagens de uma URL externa e também de recursos locais.

Personalização de Botões:

Crie um conjunto de botões interativos (ElevatedButton, TextButton, OutlinedButton) com diferentes estilos.

Personalize a aparência dos botões, alterando cores, tamanhos e adicionando ícones.

Barra de Progresso Dinâmica:

Utilize a barra de progresso (LinearProgressIndicator ou CircularProgressIndicator) para indicar visualmente o progresso de uma tarefa.

Atualize dinamicamente o valor da barra de progresso em resposta a eventos no aplicativo.