**Apostila Flutter (Parte 2): Listas, Grades e Feedback ao Usuário**

**4. Layouts para Conteúdo Dinâmico e Rolável**

A maioria dos aplicativos precisa exibir listas de dados que podem ser muito longas para caber em uma única tela. O Flutter oferece widgets otimizados para isso, garantindo uma rolagem suave e bom desempenho.

**ListView**

* **Visual:** Uma lista de itens que pode ser rolada verticalmente ou horizontalmente.
* **Descrição:** O widget mais comum para exibir uma lista de outros widgets. É altamente otimizado e constrói os itens sob demanda, economizando memória.
* **Construtor Principal:**
  + **ListView.builder():** A melhor opção para listas longas ou dinâmicas. Ele constrói apenas os itens que estão visíveis na tela.
* **Principais Propriedades do ListView.builder:**
  + itemCount: O número total de itens na sua lista de dados.
  + itemBuilder: Uma função que é chamada para construir cada item da lista. Ela recebe o context e um index (a posição do item).
  + scrollDirection: Define a direção da rolagem (Axis.vertical ou Axis.horizontal).

**Exemplo de ListView.builder:**

Vamos criar uma lista simples de contatos.

Dart

// Dados de exemplo

final List<String> contatos = [

'Ana', 'Bruno', 'Carla', 'Daniel', 'Eduarda', 'Fábio', 'Gabriela', 'Hugo'

];

// Dentro do seu widget build

ListView.builder(

itemCount: contatos.length, // O tamanho da nossa lista de dados

itemBuilder: (BuildContext context, int index) {

// Para cada item na lista, criamos um ListTile

return Card(

child: ListTile(

leading: Icon(Icons.person),

title: Text(contatos[index]), // Pega o nome do contato pelo índice

subtitle: Text('Clique para ver detalhes'),

trailing: Icon(Icons.arrow\_forward\_ios),

onTap: () {

print('Clicou em ${contatos[index]}');

},

),

);

},

)

**Dica:** O widget ListTile é perfeito para criar linhas em uma lista com um padrão de ícone inicial (leading), título, subtítulo e um ícone final (trailing).

**GridView**

* **Visual:** Uma grade de itens que pode ser rolada.
* **Descrição:** Similar ao ListView, mas organiza os itens em uma grade com múltiplas colunas.
* **Construtor Principal:**
  + **GridView.builder():** Assim como no ListView, é a escolha de melhor performance para grades.
* **Principais Propriedades do GridView.builder:**
  + itemCount: O número total de itens.
  + itemBuilder: A função que constrói cada item da grade.
  + gridDelegate: A propriedade mais importante. Define como os itens são organizados na grade. A mais comum é a **SliverGridDelegateWithFixedCrossAxisCount**.
    - crossAxisCount: O número de colunas na grade.
    - crossAxisSpacing: O espaçamento horizontal entre os itens.
    - mainAxisSpacing: O espaçamento vertical entre os itens.

**Exemplo de GridView.builder:**

Vamos criar uma galeria de fotos simples.

Dart

// Dados de exemplo (poderiam ser URLs de imagens)

final List<IconData> galleryIcons = [

Icons.photo\_camera, Icons.image, Icons.photo\_album, Icons.collections,

Icons.photo\_library, Icons.add\_a\_photo, Icons.camera\_alt, Icons.photo\_filter

];

// Dentro do seu widget build

GridView.builder(

padding: const EdgeInsets.all(10.0),

gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(

crossAxisCount: 2, // 2 colunas

crossAxisSpacing: 10.0, // Espaçamento horizontal

mainAxisSpacing: 10.0, // Espaçamento vertical

),

itemCount: galleryIcons.length,

itemBuilder: (BuildContext context, int index) {

// Para cada item, criamos um Card com um ícone dentro

return Card(

elevation: 4,

color: Colors.blue.shade100,

child: InkWell( // Adiciona o efeito de "splash" ao tocar

onTap: () {

print('Abriu a foto $index');

},

child: Icon(

galleryIcons[index],

size: 60,

color: Colors.blue.shade800,

),

),

);

},

)

**5. Widgets de Feedback ao Usuário**

É crucial informar ao usuário o que está acontecendo no app. Estes widgets são usados para exibir alertas, confirmações e mensagens temporárias.

**AlertDialog**

* **Visual:** Uma caixa de diálogo modal (bloqueia o resto da UI) que aparece sobre o conteúdo da tela.
* **Descrição:** Usado para situações que exigem uma confirmação ou atenção do usuário. Geralmente contém um título, uma mensagem e botões de ação.
* **Como usar:** Você precisa chamar a função showDialog() para exibi-lo.
* **Principais Propriedades:**
  + title: O título do diálogo (um Text).
  + content: O corpo da mensagem (pode ser qualquer widget, mas geralmente um Text).
  + actions: Uma lista de botões (TextButton, ElevatedButton) na parte inferior do diálogo.

**Exemplo de AlertDialog:**

Vamos criar um botão que, ao ser pressionado, pergunta ao usuário se ele deseja sair.

Dart

// Esta função pode ser chamada no onPressed de um botão

void \_mostrarDialogoDeSaida(BuildContext context) {

showDialog(

context: context,

builder: (BuildContext context) {

return AlertDialog(

title: Text("Confirmar Saída"),

content: Text("Você tem certeza que deseja sair do aplicativo?"),

actions: <Widget>[

// Botão de Cancelar

TextButton(

child: Text("Cancelar"),

onPressed: () {

Navigator.of(context).pop(); // Fecha o diálogo

},

),

// Botão de Sair

ElevatedButton(

child: Text("Sair"),

onPressed: () {

// Coloque aqui a lógica para sair do app

Navigator.of(context).pop(); // Fecha o diálogo

},

style: ElevatedButton.styleFrom(backgroundColor: Colors.red),

),

],

);

},

);

}

// Em algum lugar na sua UI, chame a função:

ElevatedButton(

onPressed: () => \_mostrarDialogoDeSaida(context),

child: Text('Sair do App'),

)

**SnackBar**

* **Visual:** Uma mensagem temporária que aparece na parte inferior da tela.
* **Descrição:** Usada para fornecer feedback breve sobre uma operação. Por exemplo, "Item salvo" ou "Email enviado". Não interrompe o fluxo do usuário.
* **Como usar:** Você precisa usar o ScaffoldMessenger para exibir uma SnackBar.
* **Principais Propriedades:**
  + content: O conteúdo principal (geralmente um Text).
  + duration: Por quanto tempo a mensagem ficará visível (Duration(seconds: 3)).
  + action: Uma ação opcional que o usuário pode tomar (um SnackBarAction).

**Exemplo de SnackBar:**

Um botão que, ao ser pressionado, simula o salvamento de um item e exibe uma confirmação.

Dart

// Botão que dispara a SnackBar

ElevatedButton(

child: Text('Salvar Item'),

onPressed: () {

// Cria a SnackBar

final snackBar = SnackBar(

content: Text('Item salvo com sucesso!'),

duration: Duration(seconds: 3),

action: SnackBarAction(

label: 'Desfazer',

onPressed: () {

// Lógica para desfazer a ação aqui

},

),

);

// Mostra a SnackBar na tela

ScaffoldMessenger.of(context).showSnackBar(snackBar);

},

)

**Importante:** Para que o ScaffoldMessenger.of(context) funcione, o widget que o chama precisa estar dentro de um Scaffold.

**Próximos Passos**

Agora que você sabe como exibir listas de dados e dar feedback ao usuário, os próximos tópicos a explorar seriam:

* **Navegação entre Telas:** Aprender a usar Navigator.push() e Navigator.pop() para criar um fluxo de múltiplas telas.
* **Gerenciamento de Estado:** Entender como gerenciar os dados do seu aplicativo de forma mais eficiente do que apenas com setState.
* **Consumo de APIs:** Aprender a buscar dados da internet para popular suas listas e grades.

Excelente! Vamos avançar para os próximos tópicos cruciais no desenvolvimento Flutter: como fazer suas telas conversarem entre si (Navegação) e uma introdução a como gerenciar os dados (Estado) do seu aplicativo de forma inteligente.

**Apostila Flutter (Parte 3): Navegação e Gerenciamento de Estado Básico**

Até agora, construímos componentes e telas isoladas. O poder de um aplicativo, no entanto, está em seu fluxo: mover-se entre telas, passar informações e garantir que os dados estejam consistentes em todos os lugares.

**6. Navegação entre Telas**

No Flutter, as telas (ou rotas) são gerenciadas como uma "pilha" (stack). Pense em uma pilha de cartas:

* **Navigator.push**: Coloca uma nova tela (carta) no topo da pilha.
* **Navigator.pop**: Remove a tela do topo, revelando a anterior.

**Navegando para uma Nova Tela (push)**

Para ir de uma Tela A para uma Tela B, usamos o Navigator.push. Ele precisa de duas coisas: o context (o endereço da tela atual na árvore de widgets) e uma Route (o caminho para a nova tela, geralmente um MaterialPageRoute).

**Exemplo:**

Temos duas telas: PrimeiraTela e SegundaTela.

Dart

// tela\_um.dart

class PrimeiraTela extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text('Primeira Tela')),

body: Center(

child: ElevatedButton(

child: Text('Ir para a Segunda Tela'),

onPressed: () {

Navigator.push(

context,

MaterialPageRoute(builder: (context) => SegundaTela()),

);

},

),

),

);

}

}

// tela\_dois.dart

class SegundaTela extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text('Segunda Tela')),

body: Center(

child: ElevatedButton(

child: Text('Voltar para a Primeira Tela'),

onPressed: () {

Navigator.pop(context); // Ação de voltar

},

),

),

);

}

}

**Nota:** A AppBar do Flutter já adiciona um botão de "voltar" automaticamente quando navegamos para uma nova tela, que executa Navigator.pop() por padrão.

**Passando Dados para a Nova Tela**

É muito comum precisar enviar informações para a tela que está sendo aberta. Fazemos isso adicionando um parâmetro ao construtor da tela de destino.

**Exemplo:** A PrimeiraTela enviará uma mensagem para a SegundaTela.

Dart

// tela\_dois.dart (Modificada)

class SegundaTela extends StatelessWidget {

// 1. Crie uma variável para receber o dado

final String mensagem;

// 2. Adicione-a ao construtor

SegundaTela({required this.mensagem});

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text('Segunda Tela')),

body: Center(

// 3. Use o dado recebido

child: Text(

mensagem,

style: TextStyle(fontSize: 24),

),

),

);

}

}

// tela\_um.dart (Modificando a chamada)

// ...

onPressed: () {

Navigator.push(

context,

MaterialPageRoute(

builder: (context) => SegundaTela(mensagem: 'Olá da Primeira Tela!'),

),

);

},

// ...

**Retornando Dados da Tela Anterior**

E se a SegundaTela precisar devolver uma informação para a PrimeiraTela (por exemplo, após o usuário fazer uma escolha)?

1. O Navigator.pop() pode carregar um valor de retorno.
2. O Navigator.push() retorna um Future, que será completado com o valor de retorno quando a tela for "popada".

**Exemplo:** Uma tela de seleção que retorna a opção escolhida.

Dart

// tela\_de\_selecao.dart

class TelaDeSelecao extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text('Escolha uma opção')),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

ElevatedButton(

child: Text('Opção A'),

onPressed: () {

// 1. Retorne um valor ao fechar a tela

Navigator.pop(context, 'Você escolheu a Opção A');

},

),

ElevatedButton(

child: Text('Opção B'),

onPressed: () {

Navigator.pop(context, 'Você escolheu a Opção B');

},

),

],

),

),

);

}

}

// Na tela principal, para chamar a tela de seleção

// Esta função precisa ser async para usar o await

void \_navegarEEsperarResultado(BuildContext context) async {

// 2. Use 'await' para esperar o resultado do Navigator.push

final resultado = await Navigator.push(

context,

MaterialPageRoute(builder: (context) => TelaDeSelecao()),

);

// 3. Use o resultado! (Geralmente mostrando em uma SnackBar)

if (resultado != null) {

ScaffoldMessenger.of(context)

..removeCurrentSnackBar()

..showSnackBar(SnackBar(content: Text('$resultado')));

}

}

**7. Introdução ao Gerenciamento de Estado**

Conforme seu aplicativo cresce, surge um problema: como compartilhar e modificar dados entre telas e widgets distantes?

**Qual é o Problema?**

Usar setState() é ótimo para gerenciar o estado *interno* de um único widget (como marcar um Checkbox). Mas imagine um carrinho de compras: vários widgets (a lista de produtos, o ícone do carrinho na AppBar, a tela de checkout) precisam saber quantos itens estão no carrinho e precisam ser atualizados quando um item é adicionado. Passar esses dados por construtores se torna um pesadelo.

**A Solução: Centralizar o Estado**

A ideia do gerenciamento de estado é remover a "lógica de negócio" e os "dados" de dentro dos widgets da UI e colocá-los em uma classe separada e centralizada. Os widgets, então, apenas "escutam" as mudanças nesse estado central e se reconstroem quando necessário.

A abordagem mais simples e recomendada para iniciantes é usar o pacote **provider**.

**Conceitos do provider**

1. **ChangeNotifier**: Uma classe do próprio Flutter que facilita a criação de um "modelo de estado". Quando os dados mudam, você chama um método chamado notifyListeners().
2. **ChangeNotifierProvider**: Um widget do pacote provider que você coloca no topo da sua árvore de widgets. Ele "fornece" sua classe de estado para todos os widgets filhos.
3. **Consumer / context.watch**: Widgets que "escutam" as mudanças no seu estado. Quando notifyListeners() é chamado, eles se reconstroem automaticamente com os novos dados.
4. **context.read**: Usado para acessar o estado sem "escutar" as mudanças. É ideal para chamar funções (como adicionarItem()).

**Exemplo Prático: Um Contador Compartilhado**

Vamos refazer um contador simples, mas com o estado separado, para que múltiplos widgets possam acessá-lo.

**1. Adicione o provider ao seu pubspec.yaml:**

YAML

dependencies:

flutter:

sdk: flutter

provider: ^6.0.0 # Use a versão mais recente

Rode flutter pub get no terminal.

**2. Crie a classe de Estado (ChangeNotifier):**

Dart

// counter\_model.dart

import 'package:flutter/foundation.dart';

class CounterModel extends ChangeNotifier {

int \_count = 0;

int get count => \_count; // Um "getter" para acessar o valor de forma segura

void increment() {

\_count++;

notifyListeners(); // Avisa a todos os "ouvintes" que o estado mudou!

}

}

**3. "Forneça" o estado para o seu aplicativo:**

No seu main.dart, envolva seu MaterialApp com o ChangeNotifierProvider.

Dart

// main.dart

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import 'counter\_model.dart'; // Importe seu modelo

import 'my\_home\_page.dart'; // Importe sua tela

void main() {

runApp(

ChangeNotifierProvider(

create: (context) => CounterModel(), // Cria uma instância do seu estado

child: MyApp(),

),

);

}

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

title: 'Exemplo de Estado',

home: MyHomePage(),

);

}

}

**4. Use o estado na sua UI:**

Dart

// my\_home\_page.dart

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import 'counter\_model.dart';

class MyHomePage extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text('Estado com Provider'),

),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

Text(

'O botão foi pressionado esta quantidade de vezes:',

),

// Este Text widget vai se reconstruir toda vez que o contador mudar

Text(

'${context.watch<CounterModel>().count}', // "Assiste" a mudanças no contador

style: Theme.of(context).textTheme.headlineMedium,

),

],

),

),

floatingActionButton: FloatingActionButton(

onPressed: () {

// Apenas chama a função, não precisa reconstruir o botão

context.read<CounterModel>().increment(); // "Lê" o estado para chamar um método

},

tooltip: 'Incrementar',

child: Icon(Icons.add),

),

);

}

}

Com essa estrutura, a sua UI (MyHomePage) está "burra". Ela não sabe como a contagem funciona, apenas exibe o valor e chama uma função. Toda a lógica está no CounterModel. Isso torna seu código muito mais limpo, organizado e fácil de testar.

**Próximos Passos**

* **Consumo de APIs:** Com o gerenciamento de estado e navegação dominados, o próximo passo é buscar dados de uma fonte externa (internet) usando pacotes como http ou dio.
* **Persistência de Dados:** Aprender a salvar dados localmente no dispositivo usando shared\_preferences (para dados simples) ou sqflite (para bancos de dados).

**Apostila Flutter (Parte 4): Consumindo APIs e Programação Assíncrona**

Seu aplicativo raramente viverá isolado. Ele precisará buscar notícias, carregar perfis de usuário, exibir produtos de uma loja, etc. Essas informações vêm de **APIs (Application Programming Interfaces)**, que são, basicamente, "pontes" que permitem que seu app solicite dados de um servidor na internet.

**8. Entendendo APIs e JSON**

* **O que é uma API?** Imagine que você está em um restaurante. Você (o app) não vai até a cozinha (o servidor) pegar sua comida. Você faz um pedido ao garçom (a API), que vai até a cozinha, pega o que você pediu e traz para você. A API define as "regras" de como fazer esses pedidos.
* **O que é JSON?** Quando o garçom traz sua comida, ela vem em um prato (um formato). Na web, o formato mais comum para troca de dados é o **JSON (JavaScript Object Notation)**. É um formato de texto leve e legível por humanos que o Dart consegue entender e transformar em objetos.

**Exemplo de JSON:**

JSON

{

"userId": 1,

"id": 1,

"title": "delectus aut autem",

"completed": false

}

**9. Programação Assíncrona: Future, async e await**

Buscar dados da internet leva tempo. Se o seu aplicativo travasse e esperasse a resposta, a experiência do usuário seria terrível. Para resolver isso, o Dart usa programação **assíncrona**.

* **Future**: É como uma "promessa". Quando você faz uma requisição à rede, o Dart não te dá a resposta imediatamente. Ele te dá um Future, que promete que "no futuro, eu terei um valor (os dados) ou um erro".
* **async e await**: São palavras-chave que tornam o trabalho com Futures muito mais fácil e legível.
  + async: Você marca uma função com async para dizer que ela fará operações assíncronas.
  + await: Dentro de uma função async, você usa await para "pausar" a execução daquela função até que um Future seja completado. Enquanto isso, o resto do seu app continua rodando normalmente.

**Conceito:**

Dart

// Função que busca dados (note o 'async' e o 'Future<String>')

Future<String> buscarDadosDoServidor() async {

print('Iniciando a busca...');

// Simula uma espera de 2 segundos pela rede

await Future.delayed(Duration(seconds: 2));

print('Dados recebidos!');

return 'Estes são os dados do servidor!';

}

// Como chamar a função

void minhaFuncaoPrincipal() async {

print('Vou chamar a função que busca dados.');

String resultado = await buscarDadosDoServidor(); // Espera o resultado

print(resultado);

print('Processo finalizado.');

}

**10. Fazendo Requisições HTTP**

Para fazer as chamadas de rede (API), usaremos o pacote oficial http.

**1. Adicione o http ao seu pubspec.yaml:**

YAML

dependencies:

flutter:

sdk: flutter

http: ^1.1.0 # Use a versão mais recente

Rode flutter pub get no terminal para instalar o pacote.

**2. O processo de buscar e mostrar dados:**

1. **Chamar a API:** Usar o pacote http para fazer uma requisição (geralmente GET).
2. **Verificar a Resposta:** Checar se a requisição foi bem-sucedida (código de status 200).
3. **Decodificar o JSON:** Transformar o texto JSON recebido em um objeto ou mapa Dart.
4. **Converter em Objetos:** Mapear os dados decodificados para uma classe de modelo do seu app (boas práticas).
5. **Exibir na Tela:** Usar um widget especial, o FutureBuilder, para lidar com os diferentes estados da requisição (carregando, concluído, erro).

**11. FutureBuilder: O Widget para Operações Assíncronas**

O FutureBuilder é um widget incrível que se reconstrói com base no estado de um Future. Ele simplifica a lógica de "mostrar um loading, depois mostrar os dados ou um erro".

* **Principais Propriedades:**
  + future: O Future que ele irá "observar" (por exemplo, sua função que chama a API).
  + builder: Uma função que é chamada toda vez que o estado do future muda. Ela recebe o context e um snapshot. O snapshot contém o status da conexão e os dados (ou erro).
* **Estados do snapshot.connectionState:**
  + ConnectionState.none: O future é nulo.
  + ConnectionState.waiting: O future ainda está em execução (ótimo para mostrar um CircularProgressIndicator).
  + ConnectionState.done: O future foi concluído. Agora você pode checar se há erro (snapshot.hasError) ou dados (snapshot.hasData).

**Exemplo Prático: Listando "Posts" de uma API Falsa**

Vamos usar a API pública [JSONPlaceholder](https://jsonplaceholder.typicode.com/) para buscar uma lista de posts e exibi-los.

**1. Crie a classe de modelo (Post):**

Dart

// post\_model.dart

import 'dart:convert';

// Função para ajudar a decodificar uma lista de JSONs

List<Post> postFromJson(String str) => List<Post>.from(json.decode(str).map((x) => Post.fromJson(x)));

class Post {

final int userId;

final int id;

final String title;

final String body;

Post({

required this.userId,

required this.id,

required this.title,

required this.body,

});

factory Post.fromJson(Map<String, dynamic> json) => Post(

userId: json["userId"],

id: json["id"],

title: json["title"],

body: json["body"],

);

}

**2. Crie a função que busca os dados:**

Dart

// api\_service.dart

import 'package:http/http.dart' as http;

import 'post\_model.dart';

class ApiService {

Future<List<Post>> fetchPosts() async {

final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));

if (response.statusCode == 200) {

// Se a chamada à API deu certo, decodificamos o JSON.

return postFromJson(response.body);

} else {

// Se algo deu errado, lançamos uma exceção.

throw Exception('Falha ao carregar os posts');

}

}

}

**3. Crie a tela para exibir os dados com FutureBuilder:**

Dart

// posts\_screen.dart

import 'package:flutter/material.dart';

import 'api\_service.dart';

import 'post\_model.dart';

class PostsScreen extends StatefulWidget {

@override

\_PostsScreenState createState() => \_PostsScreenState();

}

class \_PostsScreenState extends State<PostsScreen> {

late Future<List<Post>> futurePosts;

final ApiService apiService = ApiService();

@override

void initState() {

super.initState();

// Iniciamos a chamada da API aqui

futurePosts = apiService.fetchPosts();

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text('Posts da API'),

),

body: Center(

child: FutureBuilder<List<Post>>(

future: futurePosts,

builder: (context, snapshot) {

if (snapshot.connectionState == ConnectionState.waiting) {

// 1. Enquanto os dados não chegam, mostre um loading.

return CircularProgressIndicator();

} else if (snapshot.hasError) {

// 2. Se deu algum erro, mostre o erro.

return Text('Erro: ${snapshot.error}');

} else if (snapshot.hasData) {

// 3. Se os dados chegaram, construa a lista.

List<Post> posts = snapshot.data!;

return ListView.builder(

itemCount: posts.length,

itemBuilder: (context, index) {

Post post = posts[index];

return ListTile(

title: Text(post.title, style: TextStyle(fontWeight: FontWeight.bold)),

subtitle: Text(post.body),

leading: CircleAvatar(

child: Text(post.id.toString()),

),

);

},

);

} else {

// Caso padrão (pouco comum de acontecer)

return Text('Nenhum dado para exibir.');

}

},

),

),

);

}

}

Com esta estrutura, seu aplicativo agora é capaz de buscar dados dinâmicos da internet e exibi-los de forma robusta e profissional, tratando os estados de carregamento e erro.

**Próximos Passos**

* **Persistência de Dados:** Aprender a salvar dados no próprio dispositivo para que o app funcione offline ou para guardar preferências do usuário (shared\_preferences).
* **Layouts Responsivos:** Como fazer seu app ter uma boa aparência em diferentes tamanhos de tela (celulares, tablets).
* **Animações:** Adicionar "vida" à sua interface com animações simples.

**Apostila Flutter (Parte 5): Persistência de Dados Local**

Até agora, todos os dados do nosso aplicativo são "voláteis", ou seja, desaparecem quando o app é fechado. A persistência de dados resolve isso, salvando as informações permanentemente no armazenamento do dispositivo.

Existem várias maneiras de fazer isso no Flutter, cada uma adequada para um tipo diferente de dado. Vamos começar com a mais simples e comum.

**12. shared\_preferences: Guardando Dados Simples**

* **O que é?** É a solução ideal para salvar dados simples de chave-valor, como configurações do usuário (ex: modo escuro ativado), o nome de usuário logado, um token de autenticação ou a pontuação mais alta de um jogo.
* **Como funciona?** Pense nisso como um pequeno cofre no seu app. Você guarda um valor (como true ou "Carlos") e dá a ele um nome único (uma "chave", como darkMode ou username). Depois, você pode usar essa chave para recuperar o valor a qualquer momento.
* **Tipos de dados suportados:** String, int, double, bool e List<String>.

**1. Adicione o shared\_preferences ao seu pubspec.yaml:**

YAML

dependencies:

flutter:

sdk: flutter

shared\_preferences: ^2.2.0 # Use a versão mais recente

Rode flutter pub get no terminal.

**2. Como Usar:** O processo é sempre:

1. Obter uma instância do SharedPreferences.
2. Usar os métodos set (ex: setBool, setString) para salvar.
3. Usar os métodos get (ex: getBool, getString) para ler.

**Importante:** Todas as operações são assíncronas, então usaremos async/await.

**Exemplo Prático: Um Interruptor de Tema (Claro/Escuro)**

Vamos criar uma tela com um Switch que permite ao usuário escolher o tema. Essa escolha será salva e, da próxima vez que o app for aberto, ele já iniciará com o tema escolhido.

**1. Crie uma classe para gerenciar as preferências:**

É uma boa prática centralizar a lógica de SharedPreferences em um único lugar.

Dart

// preferences\_service.dart

import 'package:shared\_preferences/shared\_preferences.dart';

class PreferencesService {

// Salva a configuração do modo escuro

Future<void> saveThemePreference(bool isDarkMode) async {

final prefs = await SharedPreferences.getInstance();

await prefs.setBool('isDarkMode', isDarkMode);

}

// Lê a configuração do modo escuro

Future<bool> getThemePreference() async {

final prefs = await SharedPreferences.getInstance();

// Retorna o valor salvo. Se não houver nada, retorna 'false' como padrão.

return prefs.getBool('isDarkMode') ?? false;

}

}

**2. Crie a tela de configurações:**

Esta tela terá um Switch para alterar e salvar o tema.

Dart

// settings\_screen.dart

import 'package:flutter/material.dart';

import 'preferences\_service.dart';

class SettingsScreen extends StatefulWidget {

@override

\_SettingsScreenState createState() => \_SettingsScreenState();

}

class \_SettingsScreenState extends State<SettingsScreen> {

final \_preferencesService = PreferencesService();

bool \_isDarkMode = false;

@override

void initState() {

super.initState();

// Ao iniciar a tela, carregamos a preferência salva

\_loadThemePreference();

}

void \_loadThemePreference() async {

bool isDarkMode = await \_preferencesService.getThemePreference();

setState(() {

\_isDarkMode = isDarkMode;

});

}

void \_toggleTheme(bool value) {

setState(() {

\_isDarkMode = value;

});

// Salva a nova preferência no dispositivo

\_preferencesService.saveThemePreference(value);

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text('Configurações')),

body: SwitchListTile(

title: Text('Modo Escuro'),

value: \_isDarkMode,

onChanged: \_toggleTheme,

secondary: Icon(Icons.brightness\_6),

),

);

}

}

**Como usar:** Agora você pode chamar a SettingsScreen a partir de qualquer botão no seu app. A escolha do usuário será salva. Para que o tema do aplicativo *realmente mude*, você precisaria integrar essa lógica com o gerenciamento de estado (como o provider da Parte 3) no topo do seu aplicativo (MaterialApp).

**13. E para Dados Complexos? Bancos de Dados SQL**

E se você precisar salvar uma lista de objetos complexos, como tarefas, despesas ou contatos, com relacionamentos entre eles? Para isso, shared\_preferences não é o ideal. A solução para dados estruturados é um banco de dados SQL.

**sqflite**

* **O que é?** É o pacote mais popular do Flutter para interagir com bancos de dados **SQLite**, que é um sistema de banco de dados SQL completo, leve e embutido no próprio dispositivo Android e iOS.
* **Quando usar?**
  + Quando você tem uma grande quantidade de dados estruturados.
  + Quando precisa fazer consultas complexas (filtrar, ordenar, agrupar dados).
  + Para funcionalidades que precisam funcionar 100% offline.
* **Curva de aprendizado:** É um pouco mais complexo que shared\_preferences, pois exige conhecimento básico de comandos SQL como CREATE TABLE, INSERT, SELECT, UPDATE e DELETE.

**Exemplo de como seria (conceitual):**

1. **Definir o Banco:** Você escreveria o código para criar um banco de dados e uma tabela (ex: CREATE TABLE tarefas (id INTEGER PRIMARY KEY, titulo TEXT, concluido INTEGER)).
2. **Criar Funções:** Você criaria funções no Dart como Future<void> inserirTarefa(Tarefa tarefa), que executaria o comando SQL INSERT INTO tarefas....
3. **Integrar na UI:** Sua interface chamaria essas funções para salvar, ler ou apagar tarefas do banco de dados local.

**Resumo: Qual usar?**

| Característica | shared\_preferences | sqflite (Banco de Dados SQL) |
| --- | --- | --- |
| **Tipo de Dado** | Simples (chave-valor) | Complexo e estruturado (objetos, relacionamentos) |
| **Exemplos de Uso** | Configurações, login, tema, token de API | Lista de tarefas, app de anotações, controle de despesas |
| **Complexidade** | Baixa | Média (requer conhecimento de SQL) |
| **Ideal para** | Guardar pequenas informações de configuração | Armazenar e consultar grandes volumes de dados offline |

Exportar para as Planilhas

Dominar a persistência de dados é um grande passo para criar aplicativos robustos e com uma ótima experiência de usuário.

**Próximos Passos**

* **Layouts Responsivos:** Aprender a usar widgets como MediaQuery, LayoutBuilder e FittedBox para que seu app se adapte lindamente a celulares de diferentes tamanhos, tablets e até mesmo desktops.
* **Animações:** Dar um toque profissional à sua interface com animações implícitas e explícitas.
* **Integração com Firebase:** Explorar a plataforma do Google para funcionalidades como autenticação, banco de dados em tempo real (Firestore) e notificações push.

Perfeito! Vamos para a etapa final da nossa apostila, focando em como garantir que seu aplicativo tenha uma aparência excelente em qualquer tamanho de tela e adicionando um toque de profissionalismo com animações.

**Apostila Flutter (Parte 6): Layouts Responsivos e Animações**

Criar um aplicativo que funciona bem é ótimo, mas criar um que *parece* ótimo em qualquer dispositivo — de um celular pequeno a um tablet grande — é o que diferencia um app amador de um profissional.

**14. Layouts Responsivos: Adaptando-se a Diferentes Telas**

Um layout responsivo não é sobre criar uma versão diferente do seu app para cada aparelho, mas sim sobre criar um layout flexível que se adapta de forma inteligente.

**MediaQuery: Descobrindo Tudo Sobre a Tela**

O MediaQuery é o seu principal aliado. Ele permite que você "pergunte" ao dispositivo sobre suas características, como:

* **Tamanho:** MediaQuery.of(context).size.width e MediaQuery.of(context).size.height.
* **Orientação:** MediaQuery.of(context).orientation (se está em modo retrato ou paisagem).
* **Densidade de pixels:** Para carregar imagens de diferentes qualidades.
* **Áreas seguras:** O espaço da tela que não está obstruído por entalhes (notches) ou barras do sistema.

**Exemplo de uso:** Criar um Container que ocupe 80% da largura da tela.

Dart

Container(

width: MediaQuery.of(context).size.width \* 0.8, // 80% da largura

height: 200,

color: Colors.blue,

child: Center(child: Text('Eu sou responsivo!', style: TextStyle(color: Colors.white))),

)

**OrientationBuilder: Retrato vs. Paisagem**

Este widget é reconstruído sempre que a orientação do dispositivo muda, permitindo que você forneça layouts diferentes para cada modo.

**Exemplo:** Mostrar uma grade de 2 colunas em modo retrato e 4 colunas em modo paisagem.

Dart

OrientationBuilder(

builder: (context, orientation) {

return GridView.count(

// Se a orientação for retrato, 2 colunas, senão, 4 colunas.

crossAxisCount: orientation == Orientation.portrait ? 2 : 4,

children: List.generate(8, (index) {

return Center(

child: Text(

'Item $index',

style: Theme.of(context).textTheme.headlineSmall,

),

);

}),

);

},

)

**LayoutBuilder: Adaptando-se ao Espaço Disponível**

Enquanto o MediaQuery informa sobre a tela inteira, o LayoutBuilder informa sobre o espaço que um widget específico tem para se desenhar. Isso é extremamente útil para criar componentes reutilizáveis.

Ele fornece constraints (restrições), que dizem a largura e altura máximas (maxWidth, maxHeight) que o widget pode ocupar.

**Exemplo:** Mostrar um Row se houver espaço suficiente, ou um Column se o espaço for estreito.

Dart

LayoutBuilder(

builder: (BuildContext context, BoxConstraints constraints) {

if (constraints.maxWidth > 600) {

// Se a largura for maior que 600 pixels (típico de tablets)

return \_buildWideLayout(); // Layout em linha (Row)

} else {

// Para larguras menores (celulares)

return \_buildNarrowLayout(); // Layout em coluna (Column)

}

},

)

**15. Adicionando Vida com Animações**

Animações sutis podem melhorar drasticamente a experiência do usuário, guiando o olhar e fornecendo feedback visual. O Flutter torna a criação de animações surpreendentemente simples.

**Animações Implícitas com AnimatedContainer**

Este é o jeito mais fácil de começar. O AnimatedContainer é como um Container normal, mas ele anima automaticamente qualquer mudança em suas propriedades (como width, height, color, padding, decoration).

**Como funciona:**

1. Use um AnimatedContainer em vez de um Container.
2. Defina uma duration (quanto tempo a animação levará).
3. Mude o valor de uma de suas propriedades dentro de um setState().

**Exemplo Prático:** Um botão que, ao ser clicado, muda de cor e tamanho suavemente.

Dart

class AnimatedButton extends StatefulWidget {

@override

\_AnimatedButtonState createState() => \_AnimatedButtonState();

}

class \_AnimatedButtonState extends State<AnimatedButton> {

// Variáveis de estado que vamos animar

double \_width = 200.0;

double \_height = 60.0;

Color \_color = Colors.blue;

BorderRadiusGeometry \_borderRadius = BorderRadius.circular(8.0);

void \_animateContainer() {

setState(() {

// Altera as propriedades. O AnimatedContainer fará o resto.

\_width = \_width == 200.0 ? 100.0 : 200.0;

\_color = \_color == Colors.blue ? Colors.red : Colors.blue;

\_borderRadius = \_borderRadius == BorderRadius.circular(8.0)

? BorderRadius.circular(50.0)

: BorderRadius.circular(8.0);

});

}

@override

Widget build(BuildContext context) {

return GestureDetector(

onTap: \_animateContainer,

child: AnimatedContainer(

width: \_width,

height: \_height,

decoration: BoxDecoration(

color: \_color,

borderRadius: \_borderRadius,

),

// Define a duração da animação

duration: Duration(seconds: 1),

// (Opcional) Define a curva da animação para um efeito mais natural

curve: Curves.fastOutSlowIn,

child: Center(

child: Text('Clique em Mim!', style: TextStyle(color: Colors.white)),

),

),

);

}

}

Existem vários outros widgets "Animated" que funcionam da mesma forma, como AnimatedOpacity, AnimatedPositioned e AnimatedPadding.

**Conclusão e Próximos Passos**

**Parabéns!** Você percorreu uma jornada completa pelos fundamentos visuais e funcionais do Flutter. Você aprendeu a:

* **Construir UIs** com widgets essenciais.
* **Organizar layouts** com Row, Column, ListView e GridView.
* **Interagir com o usuário** através de botões, campos e diálogos.
* **Navegar entre telas** e gerenciar o estado com provider.
* **Buscar dados da internet** (APIs) de forma assíncrona.
* **Salvar dados localmente** com shared\_preferences.
* **Criar layouts responsivos** e **animações básicas**.

Com esta base sólida, você está pronto para explorar o ecossistema Flutter mais a fundo.

**Para onde ir agora?**

1. **Firebase:** A plataforma do Google é a companheira perfeita para o Flutter. Explore serviços como:
   * **Authentication:** Para adicionar login com Google, email, etc., de forma fácil.
   * **Cloud Firestore:** Um banco de dados NoSQL em tempo real que sincroniza magicamente entre todos os usuários.
   * **Cloud Storage:** Para salvar arquivos como fotos e vídeos dos usuários.
2. **Gerenciamento de Estado Avançado:** Embora provider seja excelente, para apps muito grandes, explore outras soluções como **BLoC** ou **Riverpod**.
3. **Animações Explícitas:** Para controle total sobre animações complexas, aprenda sobre AnimationController e Tween.
4. **Flutter na Prática:** A melhor maneira de aprender é construindo. Desafie-se a clonar a interface de apps famosos como Instagram, WhatsApp ou Spotify.

A comunidade Flutter é uma das mais ativas e acolhedoras. Participe de fóruns, assista a tutoriais no YouTube e, o mais importante, continue codificando.

**Boa sorte na sua jornada com Flutter!**