Aplicações Mobile Aula 03

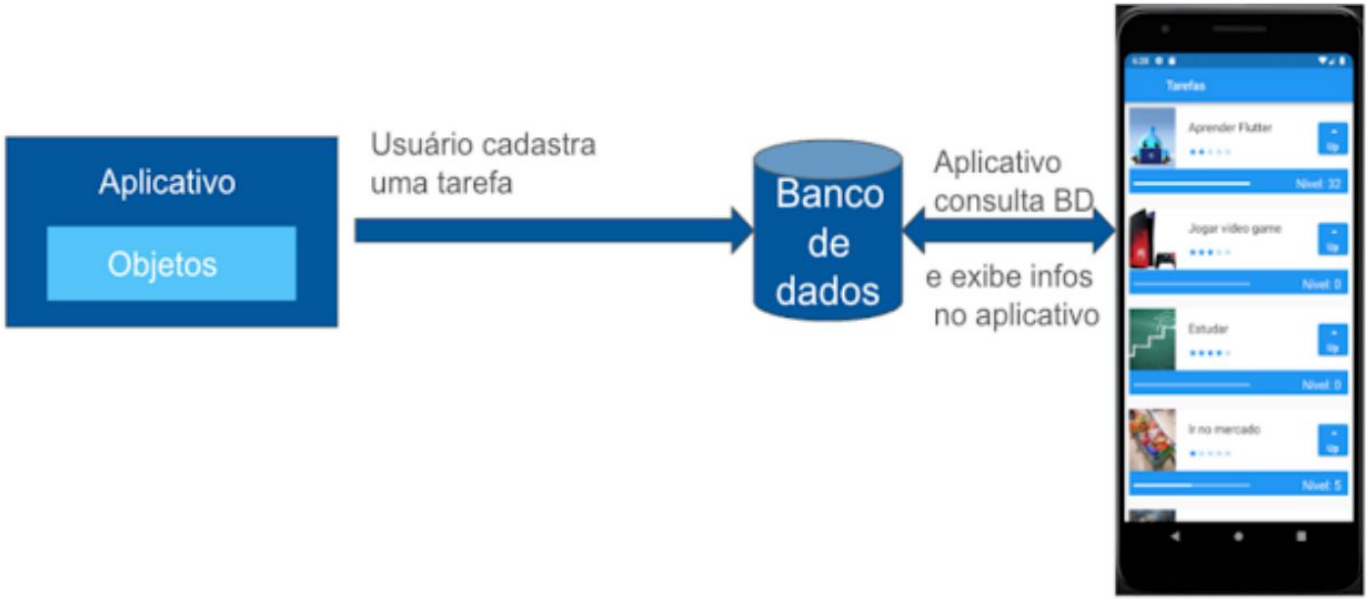
Prof. Me Daniel Vieira

SENAI

Agenda

1 - Persistência de dados
2 - Bibliotecas utilizadas em Flutter para persistência de dados
3 - Shared preferences
4 - Sqlite
5 - Exemplo de aplicativo

Persistência de dados



Principais bibliotecas utilizadas para persistir dados no Flutter

Biblioteca	Descrição	Relacional/ Não relacional	Linguagem base	Vantagem principal	Desvantagem principal	Aplicações
Shared _ preferences	Armazenamento simples de pares chave-valor	Não Relacional	Dart	Simplicidade	Limitação de dados	Dados de usuários como login e senha
Sqlite	Interface SQLite direta	Relacional	SQL	Consultas avançadas	Complexidade	Características de um produto como? Cor, quantidade, preço
Hive	Banco de dados NoSQL rápido	Não Relacional	Dart	Performance	Escalabilidade	Lista de contatos, lista de tarefas

Principais bibliotecas utilizadas para persistir dados no Flutter

Biblioteca	Descrição	Relacional/ Não relacional	Linguagem base	Vantagem principal	Desvantagem principal	Aplicações
Drift	ORM para SQLite com integração Dart	Relacional	Dart/SQL	Integração com Dart	Curva de aprendizado	Gerenciamento de estoque ou finanças, onde precisa realizar consultas SQL complexas com joins, agregações e subconsulta
Firebase	Banco de dados Cloud	Não relacional	JSON (estrutura de dados)	Fácil integração com aplicativos móveis, sincronização em tempo real e escalabilidade automática.	Não é ideal para dados altamente relacionados e complexos devido à falta de suporte a joins.	Aplicativos que precisam de dados em tempo real, sincronização rápida entre dispositivos ou integração com serviços na nuvem, como autenticação e armazenamento.

Principais bibliotecas utilizadas para persistir dados no Flutter

Biblioteca	Descrição	Relacional/ Não relacional	Linguagem base	Vantagem principal	Desvantagem principal	Aplicações
Drift	ORM para SQLite com integração Dart	Relacional	Dart/SQL	Integração com Dart	Curva de aprendizado	Gerenciamento de estoque ou finanças, onde precisa realizar consultas SQL complexas com joins, agregações e subconsulta
Firebase	Banco de dados Cloud	Não relacional	JSON (estrutura de dados)	Fácil integração com aplicativos móveis, sincronização em tempo real e escalabilidade automática.	Não é ideal para dados altamente relacionados e complexos devido à falta de suporte a joins.	Aplicativos que precisam de dados em tempo real, sincronização rápida entre dispositivos ou integração com serviços na nuvem, como autenticação e armazenamento.

Shared preferences

O SharedPreferences é uma biblioteca do Flutter amplamente utilizada para armazenar dados simples e persistentes diretamente no dispositivo do usuário. Por sua simplicidade e abordagem intuitiva, essa ferramenta é, muitas vezes, o primeiro contato dos desenvolvedores com a persistência de dados no Flutter.

Os dados são armazenados em um formato de chave-valor, que funciona de maneira semelhante aos Maps no Dart, ao Local Storage no JavaScript ou ao UserDefaults no iOS. Essa estrutura organiza informações de forma clara e acessível, permitindo que cada dado seja identificado por uma chave única.

Shared preferences

- Maps no Dart: Os Maps são coleções associativas que armazenam pares de chave e valor. Por exemplo, um Map pode ser usado para associar o nome de um usuário (chave) ao seu endereço de e-mail (valor).
- Local Storage no JavaScript: O Local Storage permite que dados sejam salvos localmente no navegador do usuário, também no formato chave-valor, tornando as informações acessíveis mesmo após recarregar ou fechar a página.
- UserDefaults no iOS: O UserDefaults é uma API utilizada para armazenar pequenas quantidades de dados, como preferências ou configurações, no dispositivo iOS. Ele utiliza a mesma lógica de chave-valor para recuperar dados de forma rápida.

Shared preferences

O SharedPreferences é ideal para salvar pequenas quantidades de dados que precisam ser acessadas rapidamente pelo aplicativo, como preferências do usuário, configurações, tokens de autenticação ou estados simples da interface. Por não exigir uma estrutura complexa, como um banco de dados, ele é uma solução prática e eficiente para armazenamentos leves e temporários.

Vantagens do SharedPreferences

- Simplicidade: O SharedPreferences é fácil de usar e não requer configuração complicada, sendo ideal para desenvolvedores iniciantes.
- Desempenho: Oferece acesso rápido aos dados, já que são armazenados localmente no dispositivo.
- Persistência: Os dados permanecem disponíveis mesmo após o aplicativo ser fechado e reaberto, garantindo uma boa experiência do usuário.
- Pequeno armazenamento: É ideal para salvar configurações, preferências do usuário, tokens de autenticação e pequenos pedaços de informação, mas com a limitação de não suportar grandes volumes de dados.

Shared preferences

Desvantagens do SharedPreferences

- Limitação de dados: Não é adequado para armazenar grandes volumes de dados ou informações complexas. É ideal apenas para dados simples e de pequeno porte.
- Falta de segurança: Os dados armazenados no SharedPreferences não são criptografados por padrão, tornando-os vulneráveis ao acesso indevido, especialmente para informações sensíveis, como senhas ou dados pessoais.
- Sem suporte para dados estruturados: Apenas permite armazenar tipos de dados primitivos (strings, números, booleanos) e listas simples. Não suporta armazenamento de objetos complexos ou relações entre dados.

Shared preferences

- Quando usar shared_preferences?

- O SharedPreferences é ideal para situações em que você precisa armazenar informações básicas que devem persistir entre sessões do aplicativo. Ele é amplamente utilizado em cenários como:
-
- Configurações do usuário: temas do aplicativo (modo claro/escuro), preferências de idioma ou outras opções personalizáveis.
- Tokens de autenticação ou informações simples de sessão que não são altamente sensíveis.
- Estados simples do aplicativo: como a última aba acessada, o progresso em um tutorial ou opções escolhidas pelo usuário.
- O SharedPreferences é uma ferramenta prática e eficiente para gerenciar dados simples e persistentes no Flutter, oferecendo uma solução rápida e fácil de implementar para necessidades comuns de armazenamento local.

- No entanto, é importante lembrar que o SharedPreferences não é a opção mais segura para dados sensíveis, pois eles são armazenados sem criptografia. Para cenários que exigem um nível maior de segurança, existe o flutter_secure_storage, considerado o "irmão mais seguro" do SharedPreferences. Ele oferece as mesmas funcionalidades, mas com criptografia integrada, sendo ideal para armazenar dados confidenciais, como tokens de acesso e informações sensíveis do usuário.

Shared preferences

Para utilizar o pacote shared_preferences é preciso adicionar como dependência do projeto

```
1. Dependencies:
2. Shared_preferences: ^2.0.8
```

Para utilizar o shared_preferences é preciso adicionar a biblioteca conforme

```
Import  package: shared_preferences/shared_preferences
```

Principais comandos da biblioteca shared preferences

Principais comandos da biblioteca shared_preferences

- Salvar: Usa setString, setInt, setBool etc., dependendo do tipo de dado.
- Recuperar: Usa getString, getInt, getBool para ler o valor associado a uma chave.
- Remover: Usa remove para apagar uma informação específica.
- Verificar: Usa containsKey para checar se uma chave existe.
- Essas operações tornam o SharedPreferences uma ferramenta poderosa para armazenar informações simples e persistentes no Flutter.

Exemplo de código com Shared Preferences

```
/*
O fluxo de dados com o SharedPreferences:

1. Cria a instancia no Shared preferences para realizar as operações com os dados

2. Recupera a informação

3. Remove uma informação

*/
// Código para salvar uma informação
// Obtém a instância do SharedPreferences, que é responsável por gerenciar o armazenamento local
final prefs = await SharedPreferences.getInstance();
// Salva o valor 'John' associado à chave 'username' no armazenamento local
prefs.setString('username', 'John');

```

Exemplo de código com Shared Preferences

```
 // Código para recuperar uma informação
18. // Obtém novamente a instância do SharedPreferences
19. final prefs = await SharedPreferences.getInstance();
20. // Recupera o valor associado à chave 'username'. Se a chave não existir, retorna uma string vazia ('')
21. final username = prefs.getString('username') ?? '';
22. // Exibe o nome de usuário recuperado no console
23. print('Username: $username');
24. 
25. // Código para remover uma informação
26. // Obtém a instância do SharedPreferences
27. final prefs = await SharedPreferences.getInstance();
28. // Remove o dado associado à chave 'username' do armazenamento local
29. prefs.remove('username');
30. 
31. // Código para verificar se uma informação existe
32. // Obtém a instância do SharedPreferences
33. final prefs = await SharedPreferences.getInstance();
34. // Verifica se a chave 'username' existe no armazenamento local
35. if (prefs.containsKey('username')) {
36.   // Caso a chave exista, imprime que o nome de usuário foi salvo anteriormente
37.   print('O nome de usuário foi salvo anteriormente');
38. } else {
39.   // Caso a chave não exista, imprime que o nome de usuário não foi salvo anteriormente
40.   print('O nome de usuário não foi salvo anteriormente');
41. }
```

Sqflite

O sqflite é uma biblioteca Flutter que fornece uma interface para o SQLite, um sistema de banco de dados relacional leve embutido no dispositivo.

Com o sqflite, é possível realizar operações CRUD (criar, ler, atualizar e excluir) em tabelas, oferecendo um armazenamento local robusto e estruturado em comparação com soluções mais simples, como o SharedPreferences.

Por utilizar o SQL (linguagem popular de gerenciamento de banco de dados) e ser relativamente fácil de implementar, o sqflite se torna uma das primeiras opções de estudo para quem deseja aprender a trabalhar com persistência de dados no Flutter.

Sqflite

Vantagens da sqflite

- Estrutura de Dados Complexa: Suporta a criação de bancos de dados relacionais com múltiplas tabelas e relações complexas entre dados. Para aplicativos que precisam armazenar dados estruturados em múltiplas tabelas, como um sistema de gerenciamento de estoque, finanças ou um aplicativo de agenda com tarefas, categorias e usuários, o Sqflite permite criar essas relações complexas entre dados. Por exemplo, um aplicativo de vendas pode usar tabelas para produtos, pedidos e clientes, vinculando-as através de chaves primárias e estrangeiras.
- Grande Volume de Dados: Adequado para armazenar e gerenciar grandes quantidades de dados de forma eficiente. Aplicativos que precisam lidar com grandes volumes de dados, como bibliotecas digitais, catálogos de produtos ou aplicativos de streaming offline, se beneficiam do Sqflite**,** que gerencia e organiza eficientemente grandes quantidades de registros.

Sqflite

- Consultas Poderosas: Permite a execução de consultas SQL avançadas para filtrar, ordenar e agrupar dados conforme necessário. Em aplicativos que requerem filtros complexos, como um app de busca avançada de hotéis com critérios como localização, preço e comodidades, as consultas SQL do Sqflite tornam possível implementar essas funcionalidades.
- Persistência e Performance: Oferece um armazenamento persistente e de alto desempenho para dados estruturados. Aplicativos que necessitam de acesso rápido a dados persistentes, como um aplicativo de notas ou diário pessoal, podem confiar no desempenho do Sqflite para carregar e salvar informações estruturadas sem comprometer a experiência do usuário.

Sqflite

Desvantagens da sqflite

- Complexidade: Requer um entendimento sólido de bancos de dados relacionais e SQL, o que pode aumentar a complexidade do desenvolvimento. Para aplicativos simples, como um contador de cliques ou um gerenciador de tarefas básico, usar o Sqflite pode ser excessivamente complexo. Isso ocorre porque o desenvolvedor precisa lidar com SQL e gerenciar a estrutura do banco, o que pode não ser necessário para projetos menores.
- Configuração: A configuração inicial e a manutenção de um banco de dados relacional podem ser mais trabalhosas comparadas a soluções mais simples. Em projetos pequenos ou com cronogramas apertados, a configuração inicial de tabelas e relações no Sqflite pode atrasar o desenvolvimento. Por exemplo, para um aplicativo que apenas salva preferências ou tokens, soluções como o SharedPreferences ou Hive seriam mais práticas.
- Sobrecarga: Pode ser excessivo para armazenamento de dados simples ou pequenas quantidades de informações. Se o aplicativo armazena apenas pequenos pedaços de informação, como configurações ou estado de temas, usar Sqflite pode ser uma solução exagerada. Nesse caso, a simplicidade de alternativas mais leves, como o SharedPreferences**,** seria suficiente.

Sqflite

O sqflite é ideal para aplicações que exigem:

- Armazenamento de dados estruturados: Quando os dados são complexos e requerem uma organização em tabelas com relações entre elas, como em um aplicativo de gestão de clientes ou inventário.
- Consultas e Filtragem de Dados: Quando há necessidade de realizar consultas avançadas ou filtragem de dados baseadas em múltiplos critérios.
- Grandes Volumes de Dados: Para aplicações que precisam armazenar e acessar uma grande quantidade de informações de maneira eficiente e rápida.
- Persistência Longa e Segura: Quando é necessário garantir que os dados serão persistidos de forma segura e disponível por longos períodos, mesmo após várias reinicializações do aplicativo.

Sqflite

O Sqflite é uma excelente ferramenta para desenvolvedores Flutter que precisam de um banco de dados local robusto. Ele é ideal para aplicativos que lidam com dados estruturados, grandes volumes de informações ou precisam de consultas SQL avançadas. Seus benefícios incluem a capacidade de criar e gerenciar tabelas de forma eficiente, atender a requisitos de armazenamento complexos e oferecer alta performance na manipulação de dados.

Sqflite

O Sqflite é uma excelente ferramenta para desenvolvedores Flutter que precisam de um banco de dados local robusto. Ele é ideal para aplicativos que lidam com dados estruturados, grandes volumes de informações ou precisam de consultas SQL avançadas. Seus benefícios incluem a capacidade de criar e gerenciar tabelas de forma eficiente, atender a requisitos de armazenamento complexos e oferecer alta performance na manipulação de dados.

Persistência de dados

Para utilizar banco de dados com o aplicativo será necessário adicionar uma dependência ao projeto
sqflite: ^2.0.3

```
35    # The following adds the Cupertino Icons font to your application.
36    # Use with the CupertinoIcons class for iOS style icons.
37    cupertino_icons: ^1.0.6
38    sqflite: ^2.0.3
39    path: ^1.8.9
40
41  dev_dependencies:
42    flutter_test:
43      sdk: flutter
44
```

Persistência de dados

CRUD - Create, read, update, delete
Criar, ler, atualizar e deletar banco de dados
Link Github SQL
https://github.com/tekartik/sqflite/blob/master/sqflite/doc/sql.md

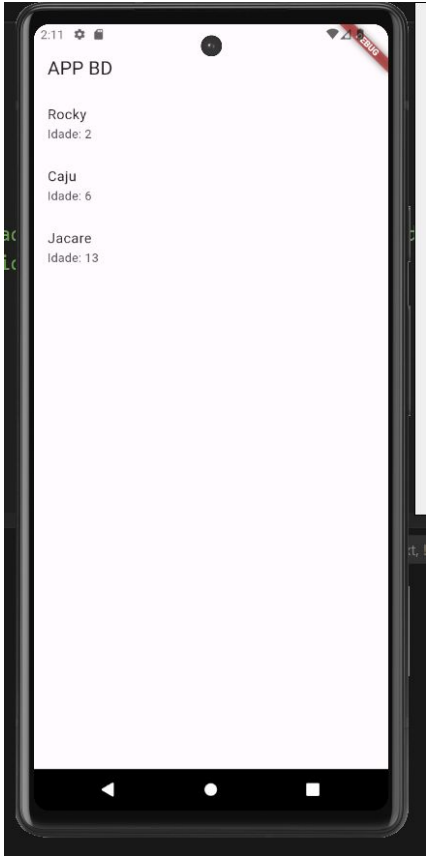
Persistência de dados

Insert - Método para inserir informações no banco de dados
Insert(nome da tabela, Map values)
As informações a serem inseridas devem estar no formato de mapa
Update - Método para mudar, atualizar, reformular alguma informação já existente no banco de dados

Delete - Deleta uma informação existente no banco de dados
Query - (nomedatabela) retorna todas as tarefas
Query (nomedatabela, where: informação a ser procurada, whereArgs[local onde procurar])

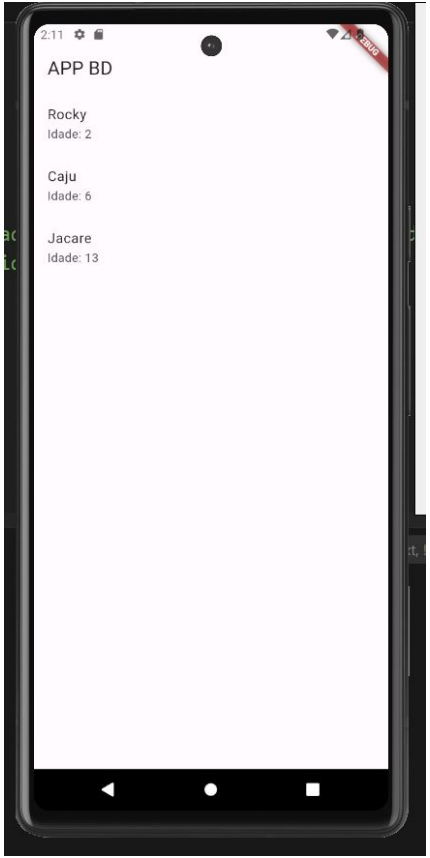
Persistência de dados

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

Run | Debug | Profile
void main() async {
  WidgetsFlutterBinding.ensureInitialized(); // Para garantir que o Flutter esteja inicializado antes de acessar o
  // banco de dados
  await _insertInitialDog(); // Espera a inserção do cachorro inicial antes de construir o widget
  runApp(MaterialApp(
    home: Home(),
  )); // MaterialApp
}

```



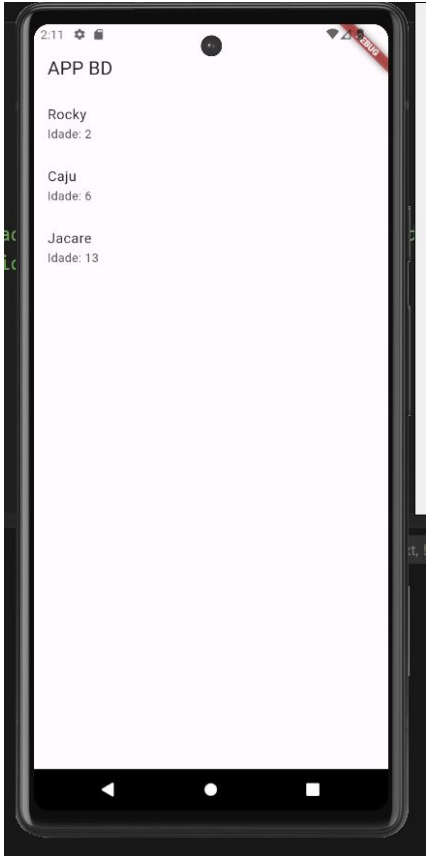
Persistência de dados

```dart
Future<void> _insertInitialDog() async {
  var database = await _initializeDatabase();
  var Rocky = Dog(id: 5, nome: "Rocky", idade: 2);
  var Caju = Dog(id: 6, nome: "Caju", idade: 6);
  var Jacare = Dog(id: 7, nome: "Jacare", idade: 13);
  //await _insertDog(database, Caju);
  await _insertDog(database, Jacare);
}

```



Persistência de dados

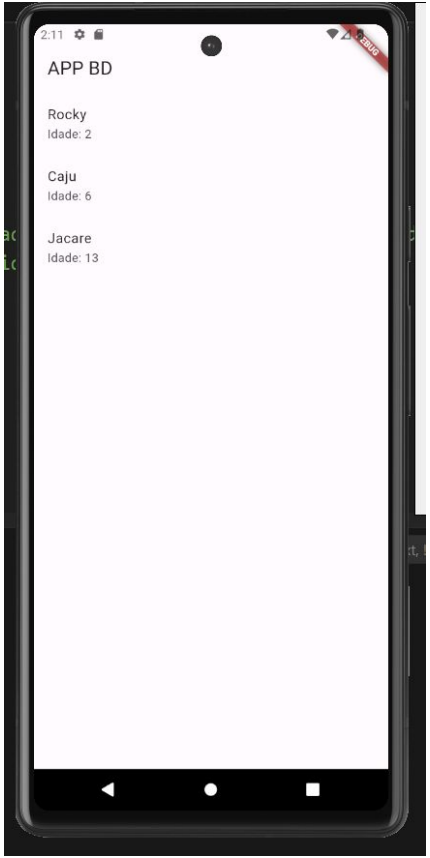
```dart
Future<Database> _initializeDatabase() async {
  return openDatabase(
    join(await getDatabasesPath(), 'dogs.db'),
    onCreate: (db, version) {
      db.execute(
        'CREATE TABLE dogs(id INTEGER PRIMARY KEY, nome TEXT, idade INTEGER)',
      );
    },
    version: 1,
  );
}
```



Persistência de dados

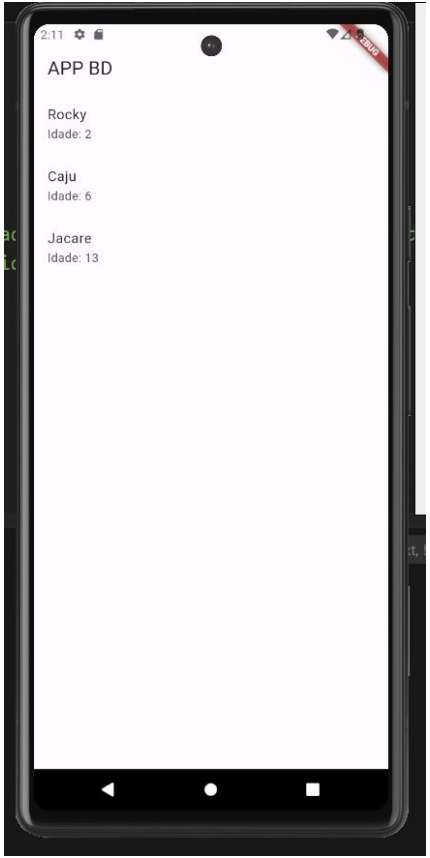
```dart
Future<void> _insertDog(Database database, Dog dog) async {
  await database.insert('dogs', dog.toMap(),
      conflictAlgorithm: ConflictAlgorithm.replace);
}
```

```dart
class Home extends StatefulWidget {
  const Home({Key? key}) : super(key: key);

  @override
  State<Home> createState() => _HomeState();
}
```



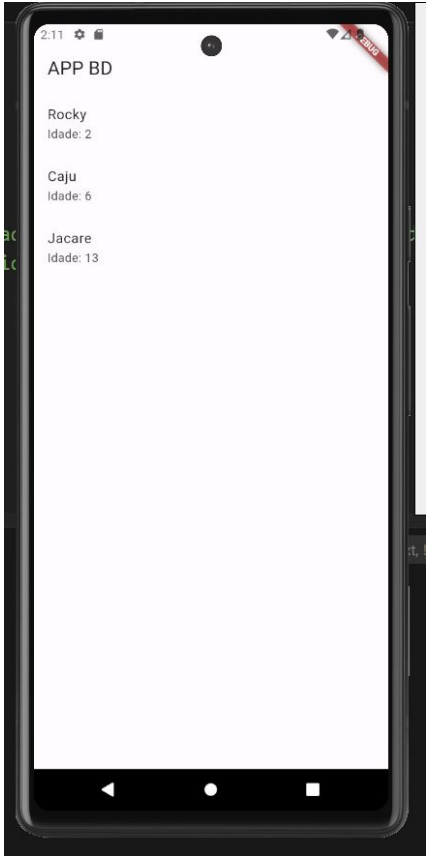
Persistência de dados

```dart
class _HomeState extends State<Home> {
  late Future<List<Dog>> _dogs;

  @override
  void initState() {
    super.initState();
    _dogs = _fetchDogs();
  }

```



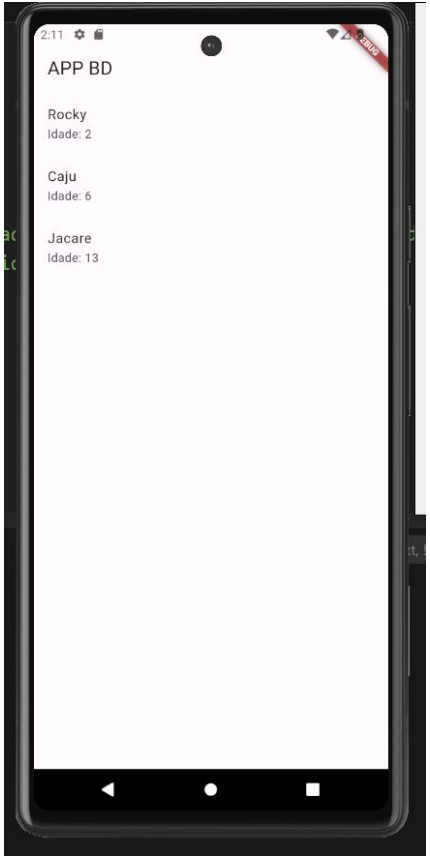
Persistência de dados

```dart
Future<List<Dog>> _fetchDogs() async {
  var database = await _initializeDatabase();
  final List<Map<String, dynamic>> maps = await database.query('dogs');

  return List.generate(maps.length, (i) {
    return Dog(
      id: maps[i]['id'],
      nome: maps[i]['nome'],
      idade: maps[i]['idade'],
    ); // Dog
  }); // List.generate
}
```



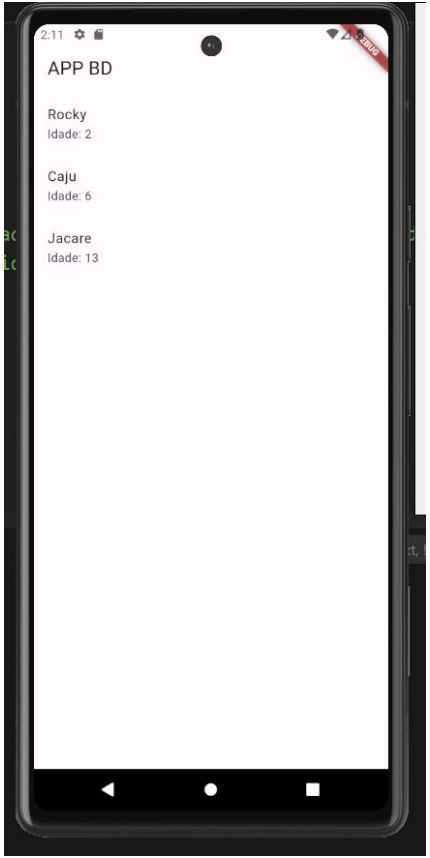
Persistência de dados

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("APP BD"),
    ), // AppBar
    body: FutureBuilder<List<Dog>>(
      future: _dogs,
      builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.waiting) {
          return Center(child: CircularProgressIndicator());
        } else if (snapshot.hasError) {
          return Center(child: Text('Error: ${snapshot.error}'));
        } else {
          final dogs = snapshot.data!;
```



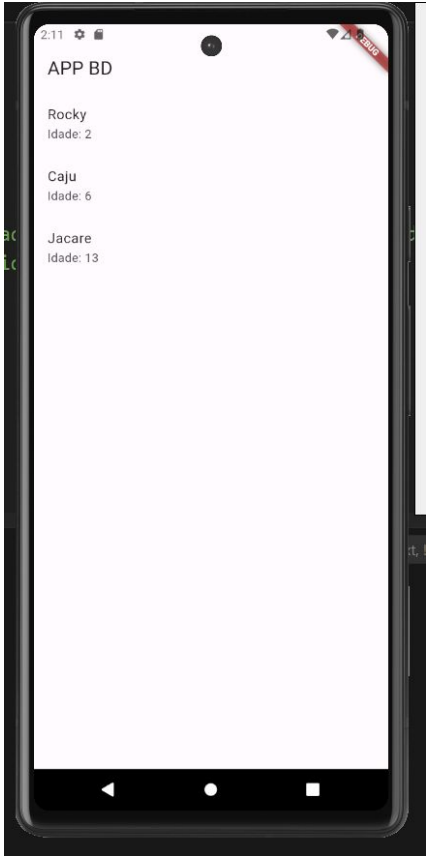
Persistência de dados

```
84                return ListView.builder(
85                  itemCount: dogs.length,
86                  itemBuilder: (context, index) {
87                    final dog = dogs[index];
88                    return ListTile(
89                      title: Text(dog.nome),
90                      subtitle: Text('Idade: ${dog.idade}'),
91                    ); // ListTile
92                  },
93                ); // ListView.builder
94              }
95            },
96          ), // FutureBuilder
97        ); // Scaffold
98      }
99    }
```



Persistência de dados

```dart
class Dog {
  final int id;
  final String nome;
  final int idade;

  Dog({required this.id, required this.nome, required this.idade});

  Map<String, dynamic> toMap() {
    return {'id': id, 'nome': nome, 'idade': idade};
  }
}
```



Programa 2

```dart
import 'package:flutter/material.dart';
import 'package:sqflite/sqflite.dart'; // pacote utilizado para criar a tabela do banco de dados
import 'package:path/path.dart'; // pacote que pega o diretorio do banco de dados

Run | Debug | Profile
void main() async {
  runApp(MaterialApp(
    home: Home(),
  )); // MaterialApp

 // getDatabasePath pega o caminho do banco de dados
  final database = openDatabase(join(await getDatabasesPath(),'dogs.db'),
  onCreate: (db, version) {
    // Cria o banco de dados
    db.execute('CREATE TABLE dogs(id INTEGER PRIMARY KEY, nome TEXT, idade INTEGER)');
  },version: 1);
```

Programa 2

```dart
// getDatabasePath pega o caminho do banco de dados
 final database = openDatabase(join(await getDatabasesPath(),'dogs.db'),
 onCreate: (db, version) {
   // Cria o banco de dados
   db.execute('CREATE TABLE dogs(id INTEGER PRIMARY KEY, nome TEXT, idade INTEGER)');
 },version: 1);

 // salva informação
 Future<void> insertDog(Dog dog)async{
   final db = await database; // armazena em db o banco de dados
   // conflictAlgorithm caso de erro tenta sobrescrever
   db.insert('dogs', dog.toMap(),conflictAlgorithm: ConflictAlgorithm.replace); // salva a informação no banco de dados

 }
```

Programa 2

```
24      // listar as informações salvas
25      Future<List<Dog>> dogs() async {
26      // Get a reference to the database.
27      final db = await database;
28
29      // Query the table for all the dogs.
30      final List<Map<String, Object?>> dogMaps = await db.query('dogs');
31
32      // Convert the list of each dog's fields into a list of `Dog` objects.
33      return [
34        for (final {
35              'id': id as int,
36              'nome': nome as String,
```

Programa 2

```dart
  final List<Map<String, Object?>> dogMaps = await db.query('dogs');

  // Convert the list of each dog's fields into a list of `Dog` objects.
  return [
    for (final {
          'id': id as int,
          'nome': nome as String,
          'idade': idade as int,
        } in dogMaps)
      Dog(id: id, nome: nome, idade: idade),
  ];
}

```

Programa 2

```
43    // Função para alterar dados já salvos no banco de dados
44    Future<void> updateDog(Dog dog)async{
45     final db = await database;
46     await db.update('dogs',dog.toMap(), where: 'id = ?',whereArgs: [dog.id]);
47
48    }
49
50    // Funçao delete
51    Future<void> deleteDog(int id)async{
52     final db = await database;
53     await db.delete('dogs',where:  'id = ?',whereArgs: [id]);
54     print("Deletando dado");
55
56    }
```

Programa 2

```
57  // insere dados
58  var Rocky = Dog(
59    id: 1,
60    nome: 'Rocky',
61    idade: 2,
62  );
63  var Marley = Dog(
64    id: 3,
65    nome: 'Marley',
66    idade: 5,
67  );
68  var Jacare = Dog(
69    id: 2,
70    nome: 'Jacare',
71    idade: 13.
```

Programa 2

```
57    // insere dados
58    var Rocky = Dog(
59      id: 1,
60      nome: 'Rocky',
61      idade: 2,
62    );
63    var Marley = Dog(
64      id: 3,
65      nome: 'Marley',
66      idade: 5,
67    );
68    var Jacare = Dog(
69      id: 2,
70      nome: 'Jacare',
71      idade: 13.
```

```
73    insertDog(Marley);
74    Rocky = Dog(
75      id: Rocky.id,
76      nome: Rocky.nome,
77      idade: Rocky.idade + 7,
78    );
79    await updateDog(Rocky);
80
81  // deletando um elemento
82    deleteDog(1);
83    print(await dogs());
84
```

Programa 2

```dart
// Classe Dog
class Dog{
  final int id;
  final String nome;
  final int idade;
  Dog({required this.id, required this.nome, required this.idade});
  // Função que converte os dados para formato Map
  Map<String,Object?> toMap(){
    return{
      'id':id,
      'nome':nome,
      'idade':idade
    };
  }

```

Obrigado!

Prof. Me Daniel Vieira

Email: danielvieira2006@gmail.com

Linkedin: Daniel Vieira

Instagram: Prof daniel.vieira95