Flutter: Banco de Dados Local

Prof. Alberto Sales

SQLite
Pacote SQFLite

O que é SQFlite?

- **SQFlite é um plugin popular do Flutter que fornece acesso ao banco de dados SQLite.**
  - O **SQLite** é um mecanismo de banco de dados relacional leve, sem servidor e independente.
  - É perfeito para armazenamento local e persistente de dados em seus aplicativos móveis do **Flutter**.
  - Permite criar aplicativos "offline-first" que podem funcionar sem conexão com a internet.

Disponibilizando SQLite

- **Adicionar pacote sqflite no pubspec.yaml**
- **Criar classe de dados para manipulação e transferência de dados entre a view e regras de negócios**
  - Uma classe auxiliar para acesso aos dados locais

Disponibilizando SQLite

- **A Classe Auxiliar de Banco de Dados (DatabaseHelper)**
  - Um padrão comum é criar uma classe **Auxiliar de Banco de Dados (serviços)**, geralmente como um **Singleton**, para gerenciar a instância única do banco de dados.
  - Isso evita múltiplas conexões e garante que as operações do seu banco de dados sejam centralizadas.

Classe de Serviços - DatabaseHelper

- **Definindo os objetos de instância e método construtor interno**

```dart
//define um construtor interno para ser usado somente dentro
//da classe ou nos seus conteúdos dependentes (contendos bibliotecas, por exemplo)
static final DatabaseHelper _instance = DatabaseHelper._internal();
factory DatabaseHelper() => _instance;
static Database? _database;

DatabaseHelper._internal();
```

Classe de Serviços - DatabaseHelper

- **Em Flutter (Dart), a palavra-chave factory define um construtor especial que nem sempre cria uma nova instância da sua classe.**
  - Ela é usada para retornar uma instância existente (singleton), uma nova instância ou um subtipo, proporcionando flexibilidade na criação de objetos, como:
    - analisar JSON,
    - implementar o padrão Singleton ou
    - retornar instâncias em cache.

Método construtor do tipo _internal()

- **Em Dart, _internal() normalmente se refere a um construtor privado dentro de uma classe.**
  - O sublinhado inicial (_) em _internal significa que este construtor é privado para sua própria biblioteca.
  - Isso significa que ele só pode ser chamado de dentro do mesmo arquivo de biblioteca onde a classe está definida, impedindo a instanciação direta da classe de fora dessa biblioteca.

Método construtor do tipo _internal()

- **Construtor Privado:**
  - A parte **_internal** é apenas um nome convencional para um construtor privado.
  - Qualquer nome de construtor prefixado com um sublinhado, como NomeDaClasse._qualquerNome(), forneceria a mesma privacidade (restrição de acesso).

Método construtor do tipo _internal()

- **Encapsulamento e Controle:**
  - Ao tornar um construtor privado, você obtém controle sobre como as instâncias da sua classe são criadas.
  - Isso é particularmente útil em padrões de projeto como o padrão **Singleton**, onde você deseja garantir que exista apenas uma instância de uma classe.

Método construtor do tipo _internal()

- **Construtores de Fábrica (Factory):**
  - Construtores privados são frequentemente usados em conjunto com construtores do tipo fábrica (**factory**).
  - Um construtor de fábrica pode então gerenciar a criação e o retorno de instâncias, potencialmente retornando instâncias existentes (para armazenamento em cache ou singletons)
  - ou criando novas com base em lógica específica, enquanto ainda usa o construtor privado **_internal** para a criação real do objeto.

Definindo métodos para manipulação dos dados

- **Métodos para obter a instância do serviço e inicialização da instância.**

```
Future<Database> get database async {
  if (_database != null) return _database!;
  _database = await _initDatabase();
  return _database!;
}

Future<Database> _initDatabase() async {
  Directory documentsDirectory = await getApplicationDocumentsDirectory();
  String path = join(documentsDirectory.path, 'person_database.db');
  return await openDatabase(path, version: 1, onCreate: _onCreate);
}
```

Métodos para Manipulação de Dados

- **Métodos para criar a tabela e método para inserir dados**

```
Future<void> _onCreate(Database db, int version) async {
  await db.execute('''
        CREATE TABLE persons(
          id INTEGER PRIMARY KEY AUTOINCREMENT,
          firstName TEXT,
          lastName TEXT
        )
      ''');
}

Future<int> insertPerson(Person person) async {
  Database db = await database;
  return await db.insert('persons', person.toMap());
}
```

Métodos para Manipulação de Dados

- **Método para obter dados do banco de dados**

```
Future<List<Person>> getPersons() async {
  Database db = await database;
  List<Map<String, dynamic>> maps = await db.query('persons');
  return List.generate(maps.length, (i) {
    return Person.fromMap(maps[i]);
  });
}
```

Métodos para Manipulação de Dados

- **Método para atualizar e deletar dados no banco de dados**

```
Future<int> updatePerson(Person person) async {
  Database db = await database;
  return await db.update(
    'persons',
    person.toMap(),
    where: 'id = ?',
    whereArgs: [person.id],
  );
}

Future<int> deletePerson(int id) async {
  Database db = await database;
  return await db.delete('persons', where: 'id = ?', whereArgs: [id]);
}
```

Notificando Objetos com ChangeNotifier

Mantendo estados (ChangeNotifier)

- **ChangeNotifier é uma classe no framework Flutter que fornece uma maneira simples de notificar widgets quando uma alteração ocorre em algum objeto.**
  - É uma parte fundamental de muitas soluções de gerenciamento de estado, especialmente quando usada com o pacote **Provider**.
    - Essencialmente, uma classe que estende ou mescla **ChangeNotifier** pode ser "ouvida" por outros widgets.

Mantendo estados (ChangeNotifier)

- **Método notifyListeners**
  - Quando você deseja atualizar a interface do usuário após uma alteração, basta chamar o método **notifyListeners**() dentro da sua classe **ChangeNotifier**.
    - Este método informa a todos os widgets que estão utilizando “ouvindo” o objeto que os dados foram alterados e que eles devem reconstruí-lo

Mantendo estados (ChangeNotifier)

- **A maneira mais comum de usar ChangeNotifier é com o widget ChangeNotifierProvider do pacote do provedor.**
  - Isso torna uma instância de **ChangeNotifier** disponível para todos os seus descendentes na árvore de widgets.
  - Os **widgets** que precisam dos dados podem acessá-los usando **Provider.of<MyModel>(context)** ou um widget **Consumer**.
    - Essa abordagem ajuda a separar a lógica de negócios do código da interface do usuário, resultando em aplicativos mais limpos e fáceis de manter.

A Classe ViewModel

- **É um padrão de arquitetura projetado para separar a interface de usuário da lógica de negócio.**

```
class PersonViewModel extends ChangeNotifier {
  List<Person> _persons = [];
  final DatabaseHelper _databaseHelper = DatabaseHelper();

  List<Person> get persons => _persons;

  PersonViewModel() {
    _loadPersons();
  }

  Future<void> _loadPersons() async {
    _persons = await _databaseHelper.getPersons();
    notifyListeners();
  }
```

A Classe ViewModel

- **Métodos para adicionar, atualizar e deletar dados**

```dart
Future<void> addPerson(Person person) async {
  await _databaseHelper.insertPerson(person);
  await _loadPersons(); // Reload data after adding
}

Future<void> updatePerson(Person person) async {
  await _databaseHelper.updatePerson(person);
  await _loadPersons(); // Reload data after updating
}

Future<void> deletePerson(int id) async {
  await _databaseHelper.deletePerson(id);
  await _loadPersons(); // Reload data after deleting
}
```