

O *SQLite* é uma tecnologia de banco de dados que não faz uso de um servidor, sendo que os dados são armazenados diretamente no dispositivo. Dessa forma, não há a necessidade de conexão com a internet e não há tráfego de dados para fora da aplicação.

Biblioteca sqflite

A criação de um banco de dados local utilizando o SQLite é bastante simples, mas antes de podermos criá-lo precisamos baixar um pacote do flutter para que tenhamos acesso a métodos específicos para tal operação. Este pacote é a ***sqflite***, e está disponível no site oficial de bibliotecas do Flutter e do Dart, o pub.dev. Podemos fazer uma busca pelo nome do pacote ou podemos simplesmente acessar a documentação do pacote.

Para instalarmos o pacote em nosso projeto, podemos fazê-lo de duas formas:

- Declarando manualmente o pacote como uma dependência em nosso arquivo *pubspec.yaml*;
- Rodar o comando de instalação diretamente no terminal. Isso fará com que o próprio gerenciador de pacotes do Dart/Flutter faça isso para nós.

Para nosso exemplo prático iremos abrir o terminal na pasta do projeto e iremos rodar o comando: ***flutter pub add sqflite***. Isso fará a declaração da dependência e também irá baixar os arquivos que necessitamos para a manipulação do banco de dados SQLite, assim como as informações que ele irá armazenar.

Criando um banco de dados

A criação de um banco de dados local utilizando o SQLite é bastante simples. Em nosso exemplo criaremos dois arquivos .dart, um chamado *IRepository*, o qual será uma classe abstrata, e outro chamado *usuario_bd_repository*, o qual utilizaremos para gerenciar nossa base de dados. Para tal utilizaremos o padrão de projeto *Repository*. Isso centralizará e facilitará a manutenção e operação do banco de dados. Ambos arquivos serão armazenados em uma pasta chamada ***repositories***, a qual ficará dentro da pasta *lib* como mostra a figura 30.

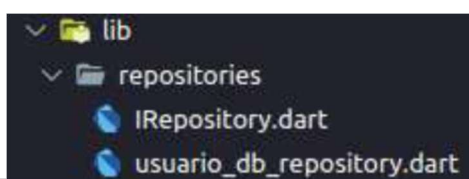


Figura - Estrutura da pasta repositories

Agora que temos nosso arquivo de manipulação, vamos criar os demais arquivos. Para tal, vamos dividi-los em duas pastas (models e repositories). Vamos começar pelos arquivos da pasta **model**.

Primeiramente devemos criar um arquivo chamada “irepository.dart”, esse arquivo servirá como interface para a implementação do padrão *Repository*.

Essa classe abstrata deverá conter todos os métodos comuns aos repositórios que desejamos criar. Para nosso exemplo utilizaremos as 4 operações básicas do CRUD, ou seja:

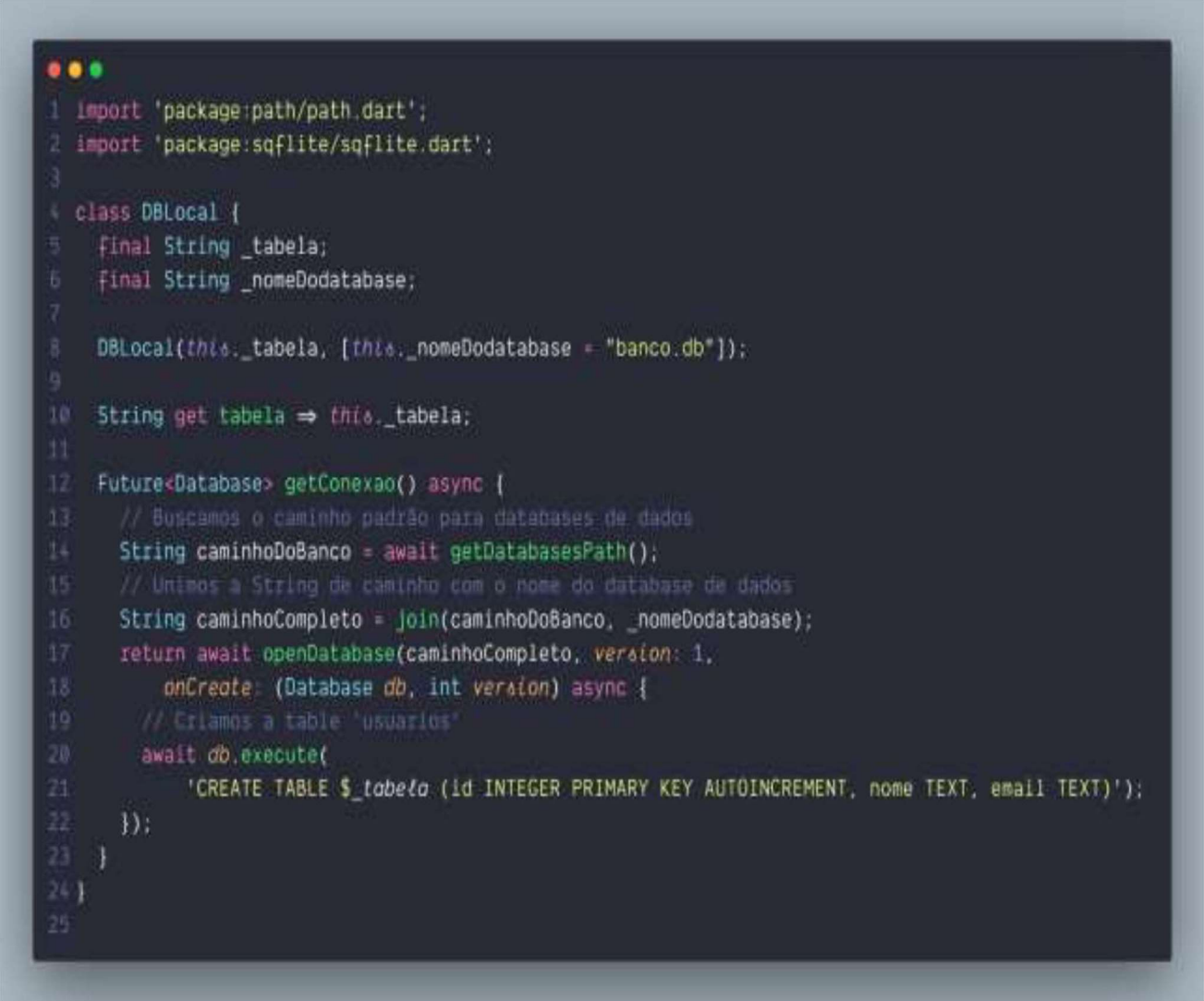
- Inserção de novos registros;
- Atualização de um registro;
- Busca de registros;
- Listagem de registros;
- Remoção de um registro;

Na figura abaixo podemos ver a estrutura desta classe abstrata.



```
1 abstract class IRepository<T> {
2   Future<List<T>> listar();
3
4   Future<T> buscar(int id);
5
6   Future<int?> inserir(T entidade);
7
8   Future<int?> atualizar({
9     required T entidade,
10    required String condicao,
11    required List valoresCondicao,
12  });
13
14   Future<int?> remover({
15     required String condicao,
16     required List valoresCondicao,
17  });
18 }
19
```

Agora que já temos nossa interface para nossos repositórios, vamos para a declaração da conexão com um banco de dados local, essa classe se chamará **DBLocal** e ficará no arquivo **dblocal.dart**. Na figura abaixo podemos ver a estrutura desta classe.



```
1 import 'package:path/path.dart';
2 import 'package:sqflite/sqflite.dart';
3
4 class DBLocal {
5   final String _tabela;
6   final String _nomeDoDatabase;
7
8   DBLocal(this._tabela, [this._nomeDoDatabase = "banco.db"]);
9
10  String get tabela => this._tabela;
11
12  Future<Database> getConexao() async {
13    // Buscamos o caminho padrão para databases de dados
14    String caminhoDoBanco = await getDatabasesPath();
15    // Unimos a String de caminho com o nome do database de dados
16    String caminhoCompleto = join(caminhoDoBanco, _nomeDoDatabase);
17    return await openDatabase(caminhoCompleto, version: 1,
18      onCreate: (Database db, int version) async {
19        // Criamos a table 'usuarios'
20        await db.execute(
21          'CREATE TABLE $_tabela (id INTEGER PRIMARY KEY AUTOINCREMENT, nome TEXT, email TEXT)');
22      });
23  }
24 }
25
```

Agora que já criamos nossa classe de acesso a base de dados local, vamos criar nossa classe de modelo para o objeto que desejamos armazenar nesta base de dados. Na figura abaixo podemos ver a estrutura desta classe.

```

1 class Usuario {
2     final int? id;
3     final String nome;
4     final String email;
5
6     Usuario({
7         this.id,
8         required this.nome,
9         required this.email,
10    });
11
12    // Converte o objeto Usuario em um Map<String, dynamic>
13    Map<String, dynamic> toMap() {
14        return {
15            'id': id,
16            'nome': nome,
17            'email': email,
18        };
19    }
20
21    // Converte um Map<String, Object> em um objeto Usuario
22    factory Usuario.fromMap(Map<String, dynamic> mapa) {
23        return Usuario(
24            id: mapa['id'],
25            nome: mapa['nome'],
26            email: mapa['email'],
27        );
28    }
29 }
30

```

Agora que já finalizamos mais essa classe, vamos nos voltar para nossas classes de repositório, as quais manteremos na pasta “*repositories*”.

Primeiramente criaremos uma nova classe abstrata, a qual irá representar o padrão *Repository* para nossa classe de modelo *Usuario*. Chamaremos essa classe de ***UsuarioRepository***, sendo que ela deverá implementar a classe ***IRepository***. Na figura abaixo podemos ver a estrutura desta classe.

```

1 import 'package:exemplo_flutter/models/DBLocal.dart';
2 import 'package:exemplo_flutter/models/usuario.dart';
3 import 'package:exemplo_flutter/models/irepository.dart';
4
5 abstract class UsuarioRepository implements IRepository<Usuario> {
6   late DBLocal dbLocal;
7 }
8

```

Agora que concluímos nossa classe **UsuarioRepository**, vamos a nossa última classe, onde implementamos nossa interface. Essa classe será chamada de **UsuarioDBRepository**, e será responsável por implementar os métodos abstratos vindos da classe **UsuarioRepository**.

Primeiramente devemos sobrescrever o objeto **dbLocal**, o qual deverá ser inicializado pelo construtor da classe. Vejamos como ficará nosso código.

```

1 import 'package:exemplo_flutter/models/DBLocal.dart';
2 import 'package:exemplo_flutter/models/usuario.dart';
3 import 'package:exemplo_flutter/repositories/usuario_repository.dart';
4 import 'package:sqflite/sqlite_api.dart';
5
6 class UsuarioDBRepository implements UsuarioRepository {
7   @override
8   late DBLocal dbLocal;
9
10  UsuarioDBRepository() {
11    dbLocal = DBLocal("usuarios");
12  }
13 }
14

```

Agora que já iniciamos a construção da nossa classe, vamos implementar os métodos abstratos recebidos da classe abstrata **UsuarioRepository**.

Inserindo registros

Para inserirmos um registro em uma tabela de nosso banco de dados utilizaremos a função `insert`, essa função recebe dois parâmetros:

- ***String***: Nome da tabela na qual o registro deve ser inserido;
- ***Map<String, dynamic>***: Objeto Map contendo os dados a serem armazenados.

Para nosso exemplo implementamos o método **inserir**, o qual recebe como parâmetro um objeto **Usuario**. Vejamos como ficou nosso código final.

```
1 @override
2 Future<int?> inserir(Usuario entidade) async {
3     Database banco = await dbLocal.getConexao();
4     return await banco.insert(dbLocal.tabela, entidade.toMap());
5 }
```