Banco de Dados SQLite

SQLite

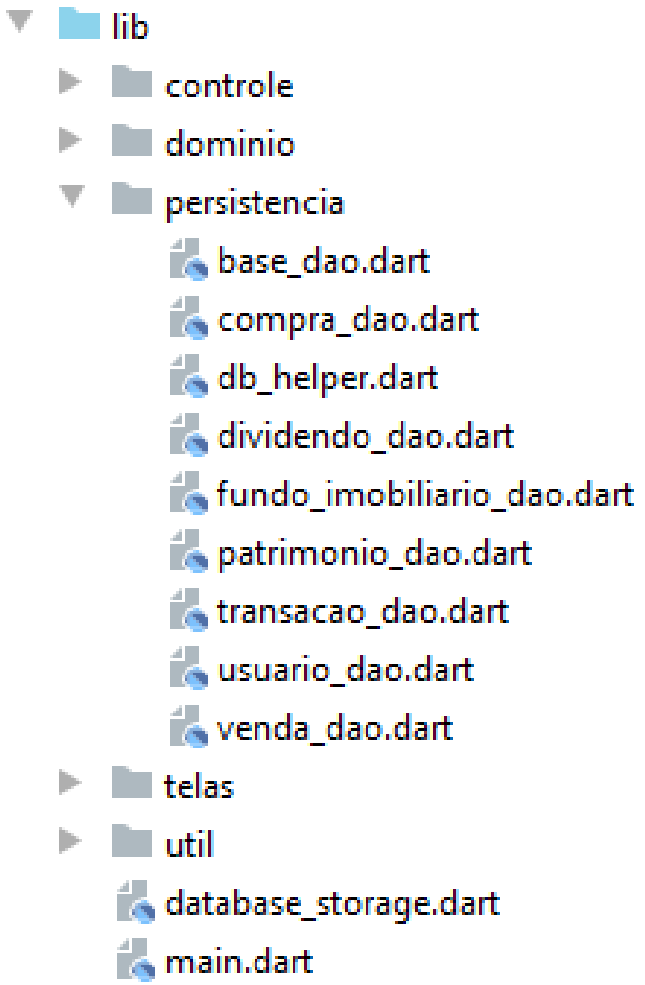
- É uma biblioteca em linguagem C que implementa um espécie de mini banco de dados.
- Todo o banco de dados é armazenado num arquivo único.
- Não é necessário instalação, configuração ou administração de banco de dados.
- Suporte a transações (COMMIT/ROLLBACK).
- Recomendável para aplicações desktop, embarcadas (dispositivos móveis) e aplicações web de pequeno porte.
- Não deve ser utilizado em aplicações de alta concorrência e sistemas ou aplicações web de grande porte.

Plugin sqflite

- Para trabalhar com o SQLite precisaremos utilizar o plugin **sqflite**. Esse plugin pode ser obtido no **pub.dev**.
- A instalação e configuração são similares as de outros plugins e pode ser vista em: https://pub.dev/packages/sqflite#-installing-tab-.

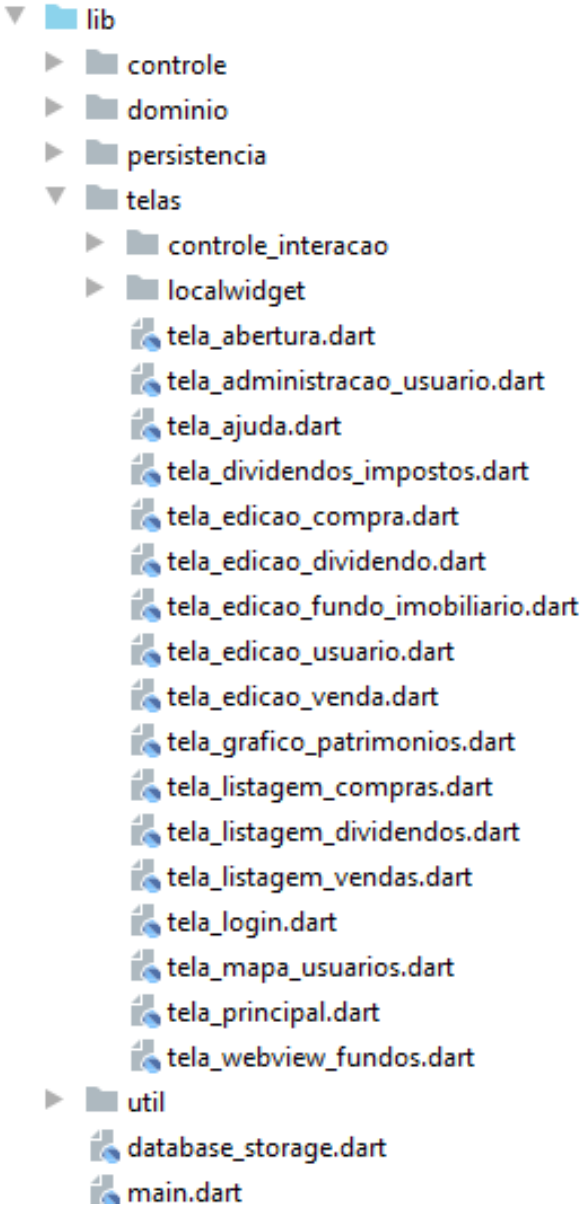
Pacote de Persistência

- No app FIIs Plan utilizamos o modelo de divisão em camadas. Nessa aula em específico trataremos basicamente de classes do pacote **persistencia**.
- As classes do pacote de persistencia terão a finalidade de trabalhar com o banco de dados SQLite e portanto terão a responsabilidade de persistir os dados do app na memória persistente do dispositivo.



Demais pacotes

- Temos também o pacote de **controle** que faz o papel de tratar questões específicas do fluxo do domínio do problema. São classes que conversam com as classes de **persistencia** e **dominio**.
- O pacote de **telas** contem as telas da aplicação e possui como sub pacotes: o pacote de **controle_interacao** (são as controladoras ligadas aos mecanismos de interação das telas) e **localwidget** (são widgets criados especificamente para esse app, como por exemplo o menu lateral). É importante notar que nas telas busca-se colocar o desenho em si da tela, enquanto no **controle_interacao** os comportamentos de interação das telas (normalmente existe um controlador de interação por tela).
- Em **util** estão widgets gerais, classes utilitárias e funções gerais que podem ser reaproveitados em outros projetos.



DatabaseHelper

- A classe **DatabaseHelper** é a classe responsável pela criação e atualização do banco de dados.
- No app FIIs Plan foi criada para ser um Singleton responsável pelo gerenciamento do banco de dados SQLite.

```dart
import 'dart:async';

import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';

class DatabaseHelper {
  // Para garantir apenas uma instância (Singleton) de DatabaseHelper
  static final DatabaseHelper _instance = DatabaseHelper.getInstance();
  // Esse é um named contructor (que chama o construtor padrão alocando o objeto)
  DatabaseHelper.getInstance();
  // Se o usuário usar DatabaseHelper() é a mesma coisa de fazer DatabaseHelper.getInstance()
  factory DatabaseHelper() => _instance;
```

https://dart.dev/guides/language/language-tour#factory-constructors

DatabaseHelper

```dart
static Database _db;

Future<Database> get db async {
  if (_db != null) {
    return _db;
  }
  _db = await _initDb();
  return _db;
}

Future _initDb() async {
  String databasesPath = await getDatabasesPath();
  String path = join(databasesPath, 'fundos.db');
  print("Database path ==> $path");

  var db = await openDatabase(path, version: 1, onCreate: _onCreate, onUpgrade: _onUpgrade);
  return db;
}
```

- Aqui temos a inicialização do banco de dados.
- O parâmetro **version** do método **openDataBase** indica a versão do banco. Quando o app for instalado pela primeira vez no dispositivo, o método colocado no parâmetro **onCreate** será chamado. Quando **version** for aumentado, ou seja, mudar por exemplo, de 1 para 2, então o método colocado em **onUpgrade** será chamado. Veremos esses métodos nos próximos slides.
- Temos o uso de um método **get** para facilitar o acesso ao banco de dados **_db**.
- E temos também a definição do nosso banco para o app. Nesse caso '**fundos.db**'.

DatabaseHelper - onCreate

```
void _onCreate(Database db, int newVersion) async {
  await db.execute(
      'CREATE TABLE USUARIO (id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE, nome TEXT, tipo  TEXT,'
      'login TEXT, senha TEXT, endereco TEXT, urlFoto TEXT)');
  await db.execute(
      'CREATE TABLE FUNDO_IMOBILIARIO (id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE, sigla  TEXT, nome  TEXT, segmento  TEXT)');
  await db.execute(
      'CREATE TABLE PATRIMONIO (id  INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE, valor_medio  REAL, qt_cotas  INTEGER, '
      ' id_fundo INTEGER,'
      ' id_usuario INTEGER,'
      'FOREIGN KEY(id_fundo) REFERENCES FUNDO_IMOBILIARIO(id),'
      'FOREIGN KEY(id_usuario) REFERENCES USUARIO(id))');
  await db.execute(
      'CREATE TABLE COMPRA (id  INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE, data_transacao TEXT, valor_cota REAL, '
          'quantidade INTEGER, taxa REAL, '
          'id_patrimonio INTEGER,'
          'FOREIGN KEY(id_patrimonio) REFERENCES PATRIMONIO(id))');
  await db.execute(
      'CREATE TABLE VENDA (id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE, data_transacao TEXT, valor_cota REAL, '
          'quantidade INTEGER, taxa REAL, '
          'id_patrimonio INTEGER,'
          'FOREIGN KEY(id_patrimonio) REFERENCES PATRIMONIO(id))');
  await db.execute(
      'CREATE TABLE DIVIDENDO (id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE, data TEXT, valor REAL, '
          'id_patrimonio INTEGER,'
          'FOREIGN KEY(id_patrimonio) REFERENCES PATRIMONIO(id))');

  await db.transaction((txn) async {
    int id2 = await txn.rawInsert(
        'INSERT INTO USUARIO(nome, tipo, login, senha, urlFoto, endereco) '
            'VALUES("Teste", "Padrão", "teste@gmail.com", "123", null, "Rua Teste")');
    print('inserted1: $id2');

    int id3 = await txn.rawInsert(
        'INSERT INTO USUARIO(nome, tipo, login, senha, urlFoto, endereco) '
            'VALUES("Administrador", "Administrador", "admin", "admin", null, "Rua Pedro Epichin, 351, Colatina Velha, Colatina, ES")');
    print('inserted1: $id3');
  });
}
```

- No método **_onCreate** colocamos os scripts de criação de tabelas e os dados iniciais.

DatabaseHelper – onUpgrade e close

- No app **FIls Plan** não utilizamos o método **_onUpgrade** pois não houve atualização no banco de dados.
- Também acabamos por não utilizar o método **close()** pois mantivemos o banco de dados “aberto” durante toda a execução do aplicativo e não houve qualquer problema.

```
Future<FutureOr<void>> _onUpgrade(Database db, int oldVersion, int newVersion) async {
  print("_onUpgrade: oldVersion: $oldVersion > newVersion: $newVersion");

  if(oldVersion == 1 && newVersion == 2) {
    await db.execute("alter table USUARIO add column NOVA TEXT");
  }
}

Future close() async {
  var dbClient = await db;
  return dbClient.close();
}
```


BaseDAO

```dart
import 'package:sqflite/sqflite.dart';
import 'db_helper.dart';

abstract class BaseDAO<T>{
  // Vai criar um get abstrato, ou seja, irá forçar os descendentes a atribuí-lo
  String get nomeTabela;

  // Obriga a definir um fromMap para o objeto específico
  T fromMap(Map<String, dynamic> map);

  Future<Database> get db => DatabaseHelper.getInstance().db;
```

- Classe genérica criada para reaproveitar códigos gerais de todas as classes DAO (de persistência).
- Possui 2 métodos abstratos para obrigar os descendentes dessa classe a implementarem comportamentos necessários a seu funcionamento. Qualquer classe DAO deverá implementar o **get nomeTabela** informando a qual tabela se refere. Terá de implementar também o método **fromMap** que converte um Map num objeto específico.
- O uso desses 2 métodos abstratos ficará mais claro nas classes descendentes.

BaseDAO

```
Future<int> obterQuantidadeBase({List<String> nomes_filtros = null, List valores = null}) async{
  final dbClient = await db;

  String sql;
  if (nomes_filtros == null){
    sql = 'SELECT count(*) FROM $nomeTabela';
  } else {
    sql = 'SELECT count(*) FROM $nomeTabela WHERE ';
    int qt_filtros = nomes_filtros.length;
    for (int i = 0; i < (qt_filtros - 1); i++) {
      sql += ' ${nomes_filtros[i]} = ? and ';
    }
    sql += ' ${nomes_filtros[qt_filtros - 1]} = ?';
  }
  final list = await dbClient.rawQuery(sql, valores);
  return Sqflite.firstIntValue(list);
}
```

- **obterQuantidadeBase** é um método que, dados os filtros e valores para esses filtros, retorna a quantidade de registros que atende aos requisitos de filtro informados.
- Como podemos ver é montado um **SELECT** com uma cláusula **WHERE** usando os filtros e seus valores.
- Notar o uso de **nomeTabela** (aquele get abstrato do slide anterior). Esse método será implementado por um descendente de **BaseDAO** e portanto ele terá de ter definido o nome da tabela, daí porque podemos utilizá-lo.

BaseDAO

```dart
void atualizarBase({List<String> colunas, List<String> nomes_filtros, List valores}) async{
  var dbClient = await db;
  String sql = 'UPDATE $nomeTabela SET ';
  int qt_colunas = colunas.length;
  for(int i=0; i<(qt_colunas-1); i++){
    sql += ' ${colunas[i]} = ?, ';
  }
  sql += ' ${colunas[qt_colunas-1]} = ?';
  sql += ' WHERE ';
  int qt_filtros = nomes_filtros.length;
  for (int i = 0; i < (qt_filtros - 1); i++) {
    sql += ' ${nomes_filtros[i]} = ? and ';
  }
  sql += ' ${nomes_filtros[qt_filtros - 1]} = ?';
  print('==> Update: $sql');
  var id = await dbClient.rawUpdate(sql, valores);
}
```

- Aqui temos a montagem de cláusulas **UPDATE** para a atualização de registros.
- Os parâmetros informados devem ser as **colunas** a serem atualizadas, os **nomes_filtros** utilizados e em **valores** colocamos os valores tanto das colunas quanto dos filtros.
- Novamente temos o uso de **nomeTabela** que, conforme descrito nos slides anteriores, será definido em classes descendentes.

BaseDAO

- O método **obterListaBase** informa uma lista de filtros e valores para obter os itens que atendem a esses critérios de filtragem.
- Esse método chama um mais geral (**obterListaQueryBase**), que dado um **sql** e os valores a serem informados retorna a lista de objetos desejada.

```
Future<List<T>>  obterListaBase({List<String> nomes_filtros = null, List valores = null}) async {
  String sql;
  if (nomes_filtros == null){
    sql = 'SELECT * FROM $nomeTabela';
  } else {
    sql = 'SELECT * FROM $nomeTabela WHERE ';
    int qt_filtros = nomes_filtros.length;
    for (int i = 0; i < (qt_filtros - 1); i++) {
      sql += ' ${nomes_filtros[i]} = ? and ';
    }
    sql += ' ${nomes_filtros[qt_filtros - 1]} = ?';
  }
  return obterListaQueryBase(sql, valores);
}
```


BaseDAO

```
Future<List<T>>  obterListaQueryBase(String sql, [List<dynamic> arguments] ) async{
  var dbClient = await db;
  final list = await dbClient.rawQuery(sql, arguments);
  final List<T> list_entity = list.map<T>((map) => fromMap(map)).toList();
  return list_entity;
}
```

- **obterListaQueryBase** recebe o **sql** e os **arguments** a serem passados para a consulta.
- Ele executa a query (com os argumentos informados) e retorna uma lista de Maps.
- O método **map** (da lista) então chama **fromMap** (que será definido nos descendentes) para cada **Map** de **list**. Ao final temos uma lista de objetos do tipo T, tipo esse que também será definido numa classe descendente de BaseDAO.

BaseDAO

```dart
Future<int> excluirBase({List<String> nomes_filtros, List valores}) async {
  var dbClient = await db;
  String sql = 'DELETE FROM $nomeTabela WHERE ';
  int qt_filtros = nomes_filtros.length;
  for (int i = 0; i < (qt_filtros - 1); i++) {
    sql += ' ${nomes_filtros[i]} = ? and ';
  }
  sql += ' ${nomes_filtros[qt_filtros - 1]} = ?';
  return await dbClient.rawDelete(sql, valores);
}

Future<int> excluirTodosBase() async {
  var dbClient = await db;
  return await dbClient.rawDelete('DELETE FROM $nomeTabela');
}
```

- Aqui temos o método **excluirBase** que tem como objetivo excluir os registros que atendam as condições de filtro (**nome_filtros** com seus respectivos **valores**).
- Também faz-se uso de **nomeTabela** e da montagem de cláusula **WHERE**.
- Por fim, o método **excluirTodosBase** apaga todos os registros da tabela **nomeTabela**.

DividendoDAO

```dart
import 'package:fundosimobiliarios/dominio/dividendo.dart';
import 'package:fundosimobiliarios/dominio/patrimonio.dart';
import 'package:fundosimobiliarios/persistencia/base_dao.dart';
import 'package:fundosimobiliarios/util/formatacao.dart';

class DividendoDAO extends BaseDAO<Dividendo> {

  @override
  // TODO: implement tableName
  String get nomeTabela => "DIVIDENDO";


  @override
  Dividendo fromMap(Map<String, dynamic> map) {
    // TODO: implement fromJson
    return Dividendo.fromMap(map);
  }
```

- **DividendoDAO** herda de **BaseDAO** e informa que o tipo **T** usado em **BaseDAO** será **Dividendo**.
- O primeiro passo é definir o **nomeTabela** para **DIVIDENDO** (que é a tabela em que essa DAO fará a persistência).
- Na sequência temos a definição de fromMap que por usa vez chama o construtor nomeado fromMap da classe Dividendo. Esse construtor cria um objeto do tipo Dividendo a partir de um Map. Abaixo podemos ver o método fromMap da classe Dividendo:

```dart
Dividendo.fromMap(Map<String, dynamic> map) : super.fromMap(map){
  data = gerarDateTime(map["data"]);
  valor = map["valor"];
}
```


Classe Dividendo

```dart
import 'package:fundosimobiliarios/dominio/objeto.dart';
import 'package:fundosimobiliarios/dominio/patrimonio.dart';
import 'package:fundosimobiliarios/util/formatacao.dart';

class Dividendo extends Objeto{
  DateTime data;
  double valor;
  Patrimonio patrimonio;


  Dividendo({this.data, this.valor, this.patrimonio});

  Dividendo.fromMap(Map<String, dynamic> map) : super.fromMap(map){
    data = gerarDateTime(map["data"]);
    valor = map["valor"];
  }

  String obterDataTransacao(){
    return formatarDateTime(data);
  }

}
```

```dart
abstract class Objeto{
  int id;

  Objeto();

  Objeto.fromMap(Map<String, dynamic> map){
    id = map["id"];
  }

}
```


Voltando a DividendoDAO

- O método **obterLista** retorna os dividendos do **patrimonio** informado como parâmetro.
- Para tanto utiliza o **id_patrimonio** e seu respectivo valor como condição de filtragem para **obterListaBase**.
- Os dividendos serão obtidos e em cada um deles será setado o atributo de patrimonio com o valor informado como parâmetro.

```
Future<List<Dividendo>> obterLista(Patrimonio patrimonio) async{
  List<Dividendo> dividendos = await this.obterListaBase(
      nomes_filtros : ["id_patrimonio"],
      valores : [patrimonio.id]);
  for(Dividendo dividendo in dividendos){
    dividendo.patrimonio = patrimonio;
  }
  return dividendos;
}
```


```
Future<int> excluir(Dividendo dividendo) async {
  return this.excluirBase(
    nomes_filtros: ["id"],
    valores: [dividendo.id],
  );
}

Future<int> inserir(Dividendo dividendo) async{
  return this.inserirBase(
      colunas : ["data", "valor", "id_patrimonio"],
      valores: [formatarDateTime(dividendo.data), dividendo.valor, dividendo.patrimonio.id]);
}
```

DividendoDAO

- Temos um método para excluir um dividendo usando seu **id** e outro para inserir um dividendo no banco de dados.
- Para inserir o dividendo no banco será necessário converter a data (dividendo.data) de **DateTime** para **String**. Isso será feito através da função **formatarDateTime**. O banco de dados SQLlite não possui o tipo DateTime para armazenamento desse tipo de dado, daí essa necessidade.
- Note que o uso da classe **BaseDAO** facilitou nossa programação e deixou nossas classes DAO como menos código e mais simples.

PatrimonioDAO

```dart
import 'package:fundosimobiliarios/dominio/patrimonio.dart';
import 'package:fundosimobiliarios/dominio/usuario.dart';
import 'package:fundosimobiliarios/persistencia/base_dao.dart';
import 'package:sqflite/sqflite.dart';

class PatrimonioDAO extends BaseDAO<Patrimonio>{
  @override
  // TODO: implement tableName
  String get nomeTabela => "PATRIMONIO";

  @override
  Patrimonio fromMap(Map<String, dynamic> map) {
    // TODO: implement fromJson
    return Patrimonio.fromMap(map);
  }
```

- O uso dos métodos de **BaseDAO** facilita nossa programação mas não conseguiremos usá-los sempre. Existem situações que temos que fazer um controle de transação mais complexo e dessa forma codificar alguns SQLs de forma mais direta.
- De qualquer forma, a implementação do **get nomeTabela** e do método **fromMap** continuarão sendo úteis.

```dart
Future<List<Patrimonio>> obterLista(Usuario usuario) async{
  final dbClient = await db;

  final list = await dbClient.rawQuery('select * from $nomeTabela join FUNDO_IMOBILIARIO '
      ' on PATRIMONIO.id_fundo = FUNDO_IMOBILIARIO.id'
      ' where PATRIMONIO.id_usuario = ? ',[usuario.id]);

  final List<Patrimonio> patrimonios = list.map<Patrimonio>((json) => fromMap(json)).toList();

  for(Patrimonio patrimonio in patrimonios){
    patrimonio.usuario = usuario;
  }

  return patrimonios;
}
```

PatrimonioDAO

- Aqui precisamos fazer um **join** da tabela de **PATRIMONIO** com a tabela de **FUNDO_IMOBILIARIO**, nesse contexto não foi possível utilizar o método **obterListaBase** como fizemos em **DividendoDAO**.
- Mas conseguimos aproveitar o método **fromMap** (que chama o fromMap de Patrimônio) para converter cada **Map** num **Patrimonio**.
- Notar também que a condição de filtragem é por **id_usuario**, ou seja, estão sendo obtidos os patrimônios de um dado usuário.

Classe Patrimonio

```dart
import 'package:fundosimobiliarios/dominio/fundo_imobiliario.dart';
import 'package:fundosimobiliarios/dominio/objeto.dart';
import 'package:fundosimobiliarios/dominio/usuario.dart';

class Patrimonio extends Objeto{
  FundoImobiliario fundo;
  Usuario usuario;
  double valor_medio;
  int qt_cotas;

  Patrimonio({this.fundo, this.usuario, this.valor_medio,
      this.qt_cotas});

  @override
  String toString() {
    return 'Patrimonio{id: $id, fundo: $fundo, usuario: $usuario, '
        'valor_medio: $valor_medio, qt_cotas: $qt_cotas}';
  }

  Patrimonio.fromMap(Map<String, dynamic> map) : super.fromMap(map){
    valor_medio = map["valor_medio"];
    qt_cotas = map["qt_cotas"];
    fundo = FundoImobiliario.fromMap(map);
  }
}
```

```dart
class FundoImobiliario extends Objeto{
  String sigla;
  String nome;
  String segmento;

  FundoImobiliario({this.sigla, this.nome, this.segmento});

  @override
  String toString() {
    return 'FundoImobiliario{sigla: $sigla, nome: $nome, segmento: $segmento}';
  }

  FundoImobiliario.fromMap(Map<String, dynamic> map) : super.fromMap(map){
    sigla = map["sigla"];
    nome = map["nome"];
    segmento = map["segmento"];
  }
}
```


Voltando a PatrimonioDAO

```
void incluir(Patrimonio patrimonio) async {
  var dbClient = await db;
  await dbClient.transaction((txn) async {
    String sigla = patrimonio.fundo.sigla;
    String nome = patrimonio.fundo.nome;
    String segmento = patrimonio.fundo.segmento;
    int id_fundo = await txn.rawInsert(
        'INSERT INTO FUNDO_IMOBILIARIO(sigla, nome, segmento) VALUES(?, ?, ?)',
        [sigla, nome, segmento]);
    patrimonio.fundo.id = id_fundo;
    print('inserted1: $id_fundo');

    int id2 = await txn.rawInsert(
        'INSERT INTO $nomeTabela(valor_medio, qt_cotas, id_fundo, id_usuario) '
            'VALUES(?, ?, ?, ?)',
        [ patrimonio.valor_medio, patrimonio.qt_cotas, id_fundo, patrimonio.usuario.id]);
    print('inserted2: $id2');
  });
}
```

- No **FIIs Plan** toda vez que um Patrimônio é inserido também é inserido o Fundo Imobiliário relacionado a ele.
- Para tanto faz-se necessário o uso de transações (para garantir que ou tudo ocorra, ou nada ocorra).
- O método **transaction** cria uma transação e as operações que acontecerem dentro dele ocorrerão dentro da transação. Nesse caso específico foram feitas as duas inserções, uma na tabela de **FUNDO_IMOBILIARIO** e outra na tabela de **PATRIMONIO**.

PatrimonioDAO

- O método remover apaga os registros nas tabelas de **PATRIMONIO** e **FUNDO_IMOBILIARIO**. Obviamente na ordem contrária a da inserção e também garantindo o controle transacional.

```
Future<int> remover(Patrimonio patrimonio) async {
  print(patrimonio.toString());
  var dbClient = await db;
  await dbClient.transaction((txn) async {
    await txn.rawDelete('delete from $nomeTabela where id = ?', [patrimonio.id]);
    return await txn.rawDelete('delete from FUNDO_IMOBILIARIO where id = ?', [patrimonio.fundo.id]);
  });
}
```


TransacaoDAO

- As classes de **CompraDAO** e **VendaDAO** são muito similares pois uma **Compra** e uma **Venda** são muito similares (ver FIIs Plan).
- Para evitar de ficar propagando o mesmo código nessas duas classes foi criada uma classe Abstrata TransacaoDAO (bem como a classe Transacao). Essa classe generaliza o que há de comum entre **CompraDAO** e **VendaDAO** deixando-as bem simples. Note que **TransacaoDAO** não tem relação direta com nenhuma tabela e por isso não implementa **get nomeTabela** (ela só pode fazer isso porque é abstrata).

```dart
class CompraDAO extends TransacaoDAO<Compra>{

  @override
  // TODO: implement tableName
  String get nomeTabela => "COMPRA";

  @override
  Compra fromMap(Map<String, dynamic> map) {
    // TODO: implement fromJson
    return Compra.fromMap(map);
  }
}
```

```dart
class VendaDAO extends TransacaoDAO<Venda>{
  @override
  // TODO: implement tableName
  String get nomeTabela => "VENDA";

  @override
  Venda fromMap(Map<String, dynamic> map) {
  // TODO: implement fromJson
  return Venda.fromMap(map);
  }
}
```


TransacaoDAO

```dart
abstract class TransacaoDAO<T extends Transacao> extends BaseDAO<T>{

  Future<List<T>> obterLista(Patrimonio patrimonio) async{
    List<T> transacoes = await this.obterListaBase(
        nomes_filtros : ["id_patrimonio"],
        valores : [patrimonio.id]);
    for(T transacao in transacoes){
      transacao.patrimonio = patrimonio;
    }
    return transacoes;
  }
```

```dart
class CompraDAO extends TransacaoDAO<Compra>{

  @override
  // TODO: implement tableName
  String get nomeTabela => "COMPRA";

  @override
  Compra fromMap(Map<String, dynamic> map) {
    // TODO: implement fromJson
    return Compra.fromMap(map);
  }
}
```

- Aqui vemos o uso significativo de **Generics**.
- **T** deve ser uma classe que descenda de **Transacao** (apenas **Compra** e **Venda** herdam de **Transacao**).
- O Tipo **T** não é definido aqui, apenas nos descendentes (CompraDAO e VendaDAO).
- Conseguimos buscar uma lista de **transacoes**, mas nesse nível não sabemos se serão de **compra** ou **venda**. Isso só será definido nos descendentes. Entretanto todo o restante do comportamento é implementado aqui.

TransacaoDAO

```
void excluir(T transacao) async{
  print(transacao.toString());
  var dbClient = await db;
  await dbClient.transaction((txn) async {
    Patrimonio patrimonio = transacao.patrimonio;
    int id1 = await txn.rawUpdate('UPDATE PATRIMONIO SET '
        ' valor_medio = ?, qt_cotas = ? WHERE id = ?',
        [ patrimonio.valor_medio, patrimonio.qt_cotas, patrimonio.id]);
    print('updated1: $id1');

    int id2 = await txn.rawDelete(
        'delete from $nomeTabela where id = ?', [transacao.id]);
    print('deleted2: $id2');
  });
}
```

- Quando ocorre a exclusão de uma **Compra** ou uma **Venda** é necessário atualizar a tabela de **Patrimonio** (pois essa mantém o controle sobre a quantidade de cotas (qt_cotas) e o valor médio (valor_medio) dessas cotas.
- Para essa operação será necessário, mais uma vez, fazer o controle transacional. Assim se houver algum problema todas as operações poderão ser desfeitas (ou faz tudo ou não faz nada).

TransacaoDAO

- Da mesma forma que na exclusão, na inclusão de uma Compra ou Venda é necessário atualizar a Tabela de **PATRIMONIO**.

```
void inserir(T transacao) async {
  var dbClient = await db;
  await dbClient.transaction((txn) async {
    Patrimonio patrimonio = transacao.patrimonio;
    int id1 = await txn.rawUpdate('UPDATE PATRIMONIO SET '
        ' valor_medio = ?, qt_cotas = ? WHERE id = ?',
        [ patrimonio.valor_medio, patrimonio.qt_cotas, patrimonio.id]);
    print('updated1: $id1');

    int id2 = await txn.rawInsert(
        'INSERT INTO $nomeTabela (data_transacao, valor_cota, quantidade, taxa, id_patrimonio) '
            'VALUES(?, ?, ?, ?, ? )',
        [formatarDateTime(transacao.data_transacao), transacao.valor_cota, transacao.quantidade, transacao.taxa,
          transacao.patrimonio.id]);
    print('inserted2: $id2');
  });
}
```


TransacaoDAO

- Aqui temos a obtenção da quantidade de Compras ou Vendas de um dado **patrimonio**.
- Notar o uso do método **obterQuantidadeBase** de **BaseDAO**.

```
Future<int> obterQuantidade(Patrimonio patrimonio) async{
  return await obterQuantidadeBase(
      nomes_filtros : ["id_patrimonio"],
      valores : [patrimonio.id]
  );
}
```


Dúvidas?