import 'package:path/path.dart'; // Importa 'path' para usar join y getDatabasesPath

import 'package:sqflite/sqflite.dart';

// Clase para representar una transacción (Modelo de Datos)

// Esta clase define la estructura de los datos que guardarás en la base de datos.

class Transaction {

final int? id; // El ID será autogenerado por la base de datos (nullable)

final String tipo; // Por ejemplo: 'Ingreso' o 'Gasto'

final String nombre; // Descripción de la transacción

final double cantidad; // Monto de la transacción

final DateTime fecha; // Fecha y hora de la transacción

// Constructor de la clase Transaction

Transaction({

this.id, // El ID es opcional al crear una nueva transacción (la DB lo asignará)

required this.tipo,

required this.nombre,

required this.cantidad,

required this.fecha,

});

// Método para convertir un objeto Transaction a un Map.

// Esto es necesario para insertar datos en la base de datos SQLite,

// ya que sqflite trabaja con Maps.

Map<String, dynamic> toMap() {

return {

'id':

id, // Si 'id' es null, SQLite lo autogenerará (PRIMARY KEY AUTOINCREMENT)

'tipo': tipo,

'nombre': nombre,

'cantidad': cantidad,

'fecha':

fecha

.toIso8601String(), // Guarda la fecha como String en formato estándar ISO 8601

};

}

// Método factory para crear un objeto Transaction desde un Map.

// Esto es necesario para leer datos de la base de datos SQLite,

// ya que sqflite devuelve los resultados como una lista de Maps.

factory Transaction.fromMap(Map<String, dynamic> map) {

return Transaction(

id: map['id'], // Lee el ID del Map

tipo: map['tipo'], // Lee el tipo del Map

nombre: map['nombre'], // Lee el nombre del Map

cantidad: map['cantidad'], // Lee la cantidad del Map

fecha: DateTime.parse(

map['fecha'],

), // Convierte el String ISO de vuelta a un objeto DateTime

);

}

}

// Clase para gestionar la base de datos (Tu DatabaseHelper)

// Implementa el patrón Singleton para asegurar que solo haya una instancia del helper.

class DatabaseHelper {

// 1. Declaración de la única instancia de la clase (Singleton).

// Se inicializa usando el constructor privado \_internal().

static final DatabaseHelper \_instance = DatabaseHelper.\_internal();

// 2. El constructor factory. Este es el constructor público que se usa

// para obtener la instancia de DatabaseHelper. Siempre devuelve la \_instance.

factory DatabaseHelper() {

return \_instance;

}

// 3. El constructor privado con nombre (\_internal).

// Este constructor es privado (por el guion bajo inicial) y se usa solo

// dentro de la clase para crear la única instancia (\_instance).

// Su declaración es necesaria para que \_instance pueda inicializarse con él.

DatabaseHelper.\_internal(); // <--- ¡Esta es la definición correcta de tu constructor privado!

// Variable estática para almacenar la instancia de la base de datos.

// Es nullable porque la base de datos puede no estar abierta inicialmente.

static Database? \_database;

// Getter asíncrono para obtener la instancia de la base de datos.

// Si la base de datos ya está abierta (\_database no es null), la devuelve directamente.

// Si no, llama a \_initDatabase() para abrirla/crearla y luego la devuelve.

Future<Database> get database async {

if (\_database != null) return \_database!;

\_database = await \_initDatabase();

return \_database!;

}

// Inicializa (abre o crea) la base de datos.

// Esta función es asíncrona porque abrir/crear una base de datos es una operación que puede tomar tiempo.

Future<Database> \_initDatabase() async {

// Obtiene la ruta estándar donde se guardan las bases de datos en la aplicación

// (esto varía según la plataforma: Android, iOS, etc.)

String documentsPath = await getDatabasesPath();

// Combina la ruta obtenida con el nombre que quieres darle a tu archivo de base de datos.

String databasePath = join(documentsPath, 'finances.db');

// Abre la base de datos.

// openDatabase() es una función de sqflite.

// - path: La ruta completa al archivo de la base de datos.

// - version: La versión del esquema de tu base de datos. Útil para migraciones futuras.

// - onCreate: Una función que se llama solo la primera vez que se crea la base de datos.

return await openDatabase(

databasePath,

version: 1, // Versión inicial de la base de datos

onCreate: \_onCreate, // Llama a \_onCreate para crear las tablas

);

}

// Función que se ejecuta solo la primera vez que se crea la base de datos.

// Aquí defines la estructura de tus tablas.

Future<void> \_onCreate(Database db, int version) async {

// Ejecuta el comando SQL para crear la tabla 'transactions'.

// Define las columnas: id (clave primaria autoincremental), tipo, nombre, cantidad, fecha.

await db.execute(

'CREATE TABLE transactions(id INTEGER PRIMARY KEY AUTOINCREMENT, tipo TEXT, nombre TEXT, cantidad REAL, fecha TEXT)',

);

}

// --- Métodos CRUD (Crear, Leer, Actualizar, Eliminar) ---

// Inserta una nueva transacción en la base de datos.

Future<int> insertTransaction(Transaction transaction) async {

final db = await database; // Obtiene la instancia de la base de datos

return await db.insert(

'transactions', // Nombre de la tabla donde insertar

transaction

.toMap(), // Convierte el objeto Transaction a Map para la inserción

conflictAlgorithm:

ConflictAlgorithm

.replace, // Si hay un conflicto de ID, reemplaza la fila existente

);

}

// Obtiene todas las transacciones de la base de datos.

Future<List<Transaction>> getTransactions() async {

final db = await database; // Obtiene la instancia de la base de datos

// Consulta todas las filas de la tabla 'transactions'.

// orderBy: 'fecha DESC' ordena los resultados por fecha de forma descendente (más recientes primero).

final List<Map<String, dynamic>> maps = await db.query(

'transactions', // Nombre de la tabla a consultar

orderBy: 'fecha DESC', // Opcional: ordena por fecha descendente

);

// Convierte la lista de Maps (obtenida de la DB) a una lista de objetos Transaction.

return List.generate(maps.length, (i) {

return Transaction.fromMap(maps[i]); // Usa el factory constructor fromMap

});

}

// Elimina una transacción específica por su ID.

Future<int> deleteTransaction(int id) async {

final db = await database; // Obtiene la instancia de la base de datos

return await db.delete(

'transactions', // Nombre de la tabla

where:

'id = ?', // Cláusula WHERE para especificar qué fila eliminar (el ?)

whereArgs: [

id,

], // Argumento para la cláusula WHERE (el valor que reemplaza al ?)

);

}

// Elimina transacciones por tipo (usado para borrar ingresos o gastos).

Future<int> deleteTransactionsByType(String tipo) async {

final db = await database; // Obtiene la instancia de la base de datos

return await db.delete(

'transactions', // Nombre de la tabla

where:

'tipo = ?', // Cláusula WHERE para especificar qué filas eliminar por tipo

whereArgs: [

tipo,

], // Argumento para la cláusula WHERE ('Ingreso' o 'Gasto')

);

}

// Cierra la base de datos (opcional, el sistema la gestiona en general).

// Útil en pruebas o si necesitas liberar recursos explícitamente.

Future<void> close() async {

final db = await database; // Obtiene la instancia de la base de datos

await db.close();

\_database =

null; // Resetea la instancia para futuras aperturas si es necesario

}

}