este código configura la estructura básica de una aplicación Flutter usando Material Design

```
// main.dart

import 'package:flutter/material.dart';
import 'home_screen.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Gestor de Gastos',
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.indigo),
        useMaterial3: true,
      ),
      home: const HomeScreen(),
      debugShowCheckedModeBanner: false,
    );
  }
}
```

Explicación Parte por Parte:

1. import 'package:flutter/material.dart';
    - ¿Qué hace? Importa la biblioteca principal de Flutter que implementa el diseño de Material Design (el sistema de diseño de Google). Esta biblioteca proporciona la gran mayoría de los widgets que usarás para construir la interfaz de usuario de tu aplicación (botones, texto, barras de aplicación, contenedores, etc.). Sin esta importación, no podrías usar widgets como MaterialApp, ThemeData, Colors, etc.

2. import 'home_screen.dart';
    - ¿Qué hace? Importa un archivo local llamado home_screen.dart. Esto significa que el código en este archivo actual (main.dart) necesita usar algo que está definido en home_screen.dart, específicamente la clase HomeScreen. Esta es la forma en que separas tu código en diferentes archivos para organizarlo mejor.

3. void main() { ... }
    - ¿Qué hace? Esta es la función principal de cualquier programa en Dart (el lenguaje que usa Flutter). Es el punto de entrada de tu aplicación. Cuando ejecutas la aplicación, el código comienza a ejecutarse desde esta función.

4. runApp(const MyApp());
    - ¿Qué hace? Esta es una función esencial de Flutter. Toma un widget (MyApp en este caso) y lo establece como la raíz de tu árbol de widgets. Flutter se encarga de "pintar" este widget y todos sus hijos en la pantalla del dispositivo. La palabra clave const es una optimización para crear instancias constantes de widgets si es posible, lo que puede mejorar el rendimiento.

5. class MyApp extends StatelessWidget { ... }
    - ¿Qué hace? Define una clase llamada MyApp.
    - extends StatelessWidget: Indica que MyApp es un widget y, específicamente, un StatelessWidget. Un StatelessWidget es un widget cuya configuración no cambia con el tiempo. Su apariencia depende únicamente de los argumentos que se le pasan cuando se crea y del BuildContext. No tiene estado interno que pueda modificarse después de ser construido. Esta clase MyApp actuará como el contenedor principal de tu aplicación.

6. const MyApp({super.key});
    - ¿Qué hace? Este es el constructor de la clase MyApp. Permite crear instancias de MyApp.
    - {super.key}: Es un patrón común en los constructores de widgets. key es un parámetro opcional que Flutter usa internamente para identificar widgets de manera eficiente cuando el árbol de widgets se reconstruye. super.key pasa cualquier key proporcionada al constructor de la clase padre (StatelessWidget).

7. @override
    - ¿Qué hace? Esta es una anotación (metadato). Indica que el método que sigue (build) está sobrescribiendo un método de la clase padre (StatelessWidget). La clase StatelessWidget requiere que sus hijos implementen un método build.

8. Widget build(BuildContext context) { ... }
    - ¿Qué hace? Este es el método más importante de cualquier StatelessWidget (y StatefulWidget). Flutter llama a este método cada vez que necesita dibujar o actualizar el widget en la pantalla.
    - Debe retornar un Widget, que es la descripción de la parte de la interfaz de usuario que este widget representa.
    - BuildContext context: Este objeto te da información sobre la posición del widget en el árbol de widgets. Es útil para acceder a temas, navegar entre pantallas, etc.

9. return MaterialApp(...)
    - ¿Qué hace? Retorna un widget MaterialApp. Este es un widget de conveniencia que envuelve una serie de widgets necesarios para crear una aplicación con Material Design. Proporciona configuraciones como el título de la aplicación, el tema, la navegación y la pantalla de inicio. Es el widget raíz típico para una aplicación Material Design.

10. title: 'Gestor de Gastos'
    - ¿Qué hace? Establece el título de la aplicación. Este título no se muestra en la pantalla de la aplicación en sí, sino que es utilizado por el sistema operativo (por ejemplo, en la barra de tareas en Android cuando se ve la lista de aplicaciones recientes, o como el título de la ventana en escritorio).

11. theme: ThemeData(...)
    - ¿Qué hace? Configura el tema visual general de la aplicación (colores, tipografía, formas de los widgets, etc.).
    - ColorScheme.fromSeed(seedColor: Colors.indigo): Crea un esquema de colores para el tema basado en un color "semilla" (Colors.indigo, que es un azul oscuro). Flutter genera automáticamente una paleta completa de colores primarios, secundarios, de error, etc., derivados de este color semilla.
    - useMaterial3: true: Habilita el uso de las características y el diseño de Material Design 3, la versión más reciente del sistema de diseño de Google.

12. home: const HomeScreen(),
    - ¿Qué hace? Especifica cuál es la pantalla inicial (la pantalla principal) de la aplicación. En este caso, establece que el widget HomeScreen (importado al principio) debe ser la primera pantalla que se muestre cuando la aplicación se inicie. La palabra clave const aquí se aplica a la instancia del widget HomeScreen.

13. debugShowCheckedModeBanner: false
    - ¿Qué hace? Controla si se muestra o no una pequeña cinta roja con la palabra "DEBUG" en la esquina superior derecha de la aplicación cuando se está ejecutando en modo de depuración. Establecerlo a false lo oculta para tener una apariencia más limpia, incluso durante el desarrollo.

En resumen, este código configura la estructura básica de una aplicación Flutter usando Material Design. Define el punto de entrada (main), crea un widget raíz (MyApp) que actúa como contenedor de la aplicación, configura el tema y especifica que la pantalla de inicio será el widget definido en home_screen.dart.

Define la estructura de un gasto y la pantalla principal (HomeScreen) de tu aplicación de gestor de gastos.

home_screen.dart

```
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import 'db_helper.dart';
import 'add_expense_screen.dart';

class Gasto {
  int? id;
  final String descripcion;
  final String categoria;
  final double monto;
  final DateTime fecha;

  Gasto({
    this.id,
    required this.descripcion,
    required this.categoria,
    required this.monto,
    required this.fecha,
  });

  Map<String, dynamic> toMap() {
    return {
      'id': id,
      'descripcion': descripcion,
      'categoria': categoria,
      'monto': monto,
      'fecha': fecha.toIso8601String(),
    };
  }

  factory Gasto.fromMap(Map<String, dynamic> map) {
    return Gasto(
      id: map['id'],
      descripcion: map['descripcion'],
      categoria: map['categoria'],
      monto: map['monto'],
      fecha: DateTime.parse(map['fecha']),
    );
  }
```

```
}

class HomeScreen extends StatefulWidget {
  const HomeScreen({super.key});

  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
  List<Gasto> _gastos = [];

  @override
  void initState() {
    super.initState();
    _cargarGastos();
  }

  Future<void> _cargarGastos() async {
    final gastos = await DBHelper.obtenerGastos();
    setState(() {
      _gastos = gastos;
    });
  }

  double get _totalGastos => _gastos.fold(0.0, (sum, g) => sum + g.monto);

  Future<void> _eliminarGasto(int id) async {
    final confirm = await showDialog<bool>(
      context: context,
      builder:
          (context) => AlertDialog(
            title: const Text('Confirmar eliminación'),
            content: const Text('¿Deseas eliminar este gasto?'),
            actions: [
              TextButton(
                onPressed: () => Navigator.pop(context, false),
                child: const Text('Cancelar'),
              ),
              TextButton(
                onPressed: () => Navigator.pop(context, true),
                child: const Text('Eliminar'),
              ),
            ],
```

```
        ),
    );

    if (confirm == true) {
      await DBHelper.eliminarGasto(id);
      _cargarGastos();
    }
  }

  void _agregarOEditarGasto({Gasto? gastoExistente}) async {
    final resultado = await Navigator.push(
      context,
      MaterialPageRoute(
        builder: (_) => AddExpenseScreen(gasto: gastoExistente),
      ),
    );
    if (resultado == true) {
      _cargarGastos();
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Gestor de Gastos')),
      body: Column(
        children: [
          Padding(
            padding: const EdgeInsets.all(16),
            child: Text(
              'Total Gastado: \$${_totalGastos.toStringAsFixed(2)}',
              style: const TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
            ),
          ),
          Expanded(
            child:
                _gastos.isEmpty
                    ? const Center(child: Text('No hay gastos registrados.'))
                    : ListView.builder(
                      itemCount: _gastos.length,
                      itemBuilder: (context, index) {
                        final gasto = _gastos[index];
                        return ListTile(
                          title: Text(gasto.descripcion),
```

```
                    subtitle: Text(
                      '${gasto.categoria} - ${DateFormat('dd/MM/yyyy').format(gasto.fecha)}',
                    ),
                    trailing: Text('\$${gasto.monto.toStringAsFixed(2)}'),
                    onTap:
                        () => _agregarOEditarGasto(gastoExistente: gasto),
                    onLongPress: () => _eliminarGasto(gasto.id!),
                  );
                },
              ),
          ),
        ],
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => _agregarOEditarGasto(),
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Explicación Parte por Parte:

1. `import 'package:flutter/material.dart';`
   - **¿Qué hace?** Importa la biblioteca principal de Flutter para construir interfaces de usuario con Material Design. Es necesaria para usar la mayoría de los widgets visuales.

2. `import 'package:intl/intl.dart';`
   - **¿Qué hace?** Importa la biblioteca `intl` (internationalization). Se utiliza aquí específicamente para formatear fechas (`DateFormat`). Necesitarás agregar esta dependencia a tu archivo `pubspec.yaml` si aún no lo has hecho (`dependencies: flutter: sdk: flutter intl: ^x.y.z`, donde `x.y.z` es la versión).

3. `import 'db_helper.dart';`
    - **¿Qué hace?** Importa un archivo local llamado `db_helper.dart`. Este archivo probablemente contiene la lógica para interactuar con la base de datos (guardar, obtener, eliminar gastos). La clase `DBHelper` se utiliza en este código para realizar operaciones de base de datos.

4. `import 'add_expense_screen.dart';`
    - **¿Qué hace?** Importa un archivo local llamado `add_expense_screen.dart`. Este archivo probablemente define la pantalla donde el usuario puede agregar o editar un gasto. Se utiliza aquí para navegar a esa pantalla.

5. `class Gasto { ... }`
    - **¿Qué hace?** Define una clase modelo llamada `Gasto`. Esta clase representa la estructura de un objeto gasto en tu aplicación. Contiene los campos que describen un gasto individual.
    - `int? id;`: Un campo opcional para el ID único del gasto, probablemente usado para la base de datos. Es opcional (`?`) porque un gasto nuevo aún no tendrá un ID hasta que se guarde.
    - `final String descripcion;`: La descripción del gasto (por ejemplo, "Café de la mañana"). Es `final` porque una vez que se crea un objeto `Gasto`, su descripción no debería cambiar.
    - `final String categoria;`: La categoría del gasto (por ejemplo, "Comida", "Transporte"). También es `final`.
    - `final double monto;`: El monto del gasto. Es `final`.
    - `final DateTime fecha;`: La fecha en que ocurrió el gasto. Es `final`.

6. `Gasto({ ... });`
    - **¿Qué hace?** Este es el constructor de la clase `Gasto`. Permite crear nuevas instancias de `Gasto`. Los campos marcados con `required` deben ser proporcionados al crear un `Gasto`, mientras que `id` es opcional.

7. `Map<String, dynamic> toMap() { ... }`
    - **¿Qué hace?** Es un método dentro de la clase `Gasto` que convierte un objeto `Gasto` en un `Map`. Un `Map` es una estructura de datos clave-valor, similar a un diccionario en Python o un objeto JSON. Este método es útil para preparar los datos de un `Gasto` para ser guardados en una base de datos (como SQLite), ya que las bases de datos a menudo trabajan con este formato.
    - `fecha.toIso8601String()`: Convierte el objeto `DateTime` de la fecha a una cadena de texto en formato ISO 8601, que es un formato estándar para representar fechas y horas y es adecuado para almacenar en bases de datos.

8. `factory Gasto.fromMap(Map<String, dynamic> map) { ... }`
    - **¿Qué hace?** Este es un *constructor factory*. Los constructores factory se utilizan para crear instancias de una clase de maneras más flexibles que un constructor normal. En este caso, `fromMap` se utiliza para crear un objeto `Gasto` a partir de un `Map` (por ejemplo, un `Map` obtenido de la base de datos).
    - `DateTime.parse(map['fecha'])`: Convierte la cadena de texto de la fecha (que se guardó en formato ISO 8601) de nuevo a un objeto `DateTime`.

9. `class HomeScreen extends StatefulWidget { ... }`
    - **¿Qué hace?** Define la clase del widget `HomeScreen`.
    - `extends StatefulWidget`: Indica que `HomeScreen` es un `StatefulWidget`. Un `StatefulWidget` es un widget que puede cambiar su apariencia en respuesta a interacciones del usuario o cambios en los datos. A diferencia de un `StatelessWidget`, tiene un estado mutable asociado que puede ser modificado.

10. `const HomeScreen({super.key});`
    - **¿Qué hace?** El constructor de `HomeScreen`, similar al de `MyApp`, pasando la clave al padre.

11. `@override State<HomeScreen> createState() => _HomeScreenState();`
    - **¿Qué hace?** Este método es requerido por `StatefulWidget`. Crea y retorna el objeto `State` asociado a este widget. El estado (`_HomeScreenState`) es donde se manejan los datos que pueden cambiar y donde se define el método `build` que describe la interfaz de usuario.

12. `class _HomeScreenState extends State<HomeScreen> { ... }`
    - **¿Qué hace?** Define la clase de estado para `HomeScreen`. El guion bajo (`_`) al principio del nombre (`_HomeScreenState`) indica que esta clase es privada y solo es accesible dentro de este archivo (`home_screen.dart`).
    - `List<Gasto> _gastos = [];`: Declara una lista privada llamada `_gastos`. Esta lista almacenará los objetos `Gasto` que se mostrarán en la pantalla. Es mutable porque el contenido de la lista cambiará a medida que se carguen, agreguen o eliminen gastos.

13. `@override void initState() { ... }`
    - **¿Qué hace?** Este es un método del ciclo de vida del `State` que se llama *una vez* cuando el objeto `State` se crea por primera vez. Es el lugar ideal para realizar configuraciones iniciales, como cargar datos de una base de datos o configurar listeners.
    - `super.initState();`: Siempre debes llamar a `super.initState()` al principio de tu implementación de `initState`.
    - `_cargarGastos();`: Llama a la función privada `_cargarGastos` para cargar los gastos existentes cuando la pantalla se inicia.

14. `Future<void> _cargarGastos() async { ... }`
    - **¿Qué hace?** Una función asíncrona (`async`) para cargar los gastos de la base de datos.
    - `final gastos = await DBHelper.obtenerGastos();`: Llama a un método `obtenerGastos` de la clase `DBHelper` (que interactúa con la base de datos). `await` espera a que la operación de la base de datos se complete antes de continuar.
    - `setState(() { _gastos = gastos; });`: Esta es la clave para actualizar la interfaz de usuario en un `StatefulWidget`. Cuando se llama a `setState`, le dices a Flutter que el estado de este widget ha cambiado y que necesita reconstruir (llamar al método `build` nuevamente) para reflejar los nuevos datos. Aquí, actualiza la lista `_gastos` con los gastos obtenidos de la base de datos.

15. `double get _totalGastos => _gastos.fold(0.0, (sum, g) => sum + g.monto);`
    - **¿Qué hace?** Define un *getter* llamado `_totalGastos`. Un getter es una forma de calcular un valor basado en el estado actual sin tener que llamarlo como una función. Calcula la suma total de los montos de todos los gastos en la lista `_gastos` usando el método `fold`.

16. `Future<void> _eliminarGasto(int id) async { ... }`
    - **¿Qué hace?** Una función asíncrona para eliminar un gasto por su ID.
    - `final confirm = await showDialog<bool>( ... );`: Muestra un diálogo de confirmación al usuario antes de eliminar un gasto. `showDialog` es una función de Flutter para mostrar diálogos. El `await` espera a que el usuario interactúe con el diálogo y retorne un valor (`true` si confirma, `false` si cancela).
    - `AlertDialog(...)`: El widget que define la apariencia del diálogo, con un título, contenido y acciones (botones).
    - `Navigator.pop(context, false)` **/** `Navigator.pop(context, true)`: Cuando se presiona un botón en el diálogo, `Navigator.pop` cierra el diálogo y retorna el valor especificado (`false` o `true`).
    - `if (confirm == true) { ... }`: Si el usuario confirmó la eliminación...
    - `await DBHelper.eliminarGasto(id);`: Llama al método `eliminarGasto` de `DBHelper` para eliminar el gasto de la base de datos.
    - `_cargarGastos();`: Vuelve a cargar la lista de gastos desde la base de datos para actualizar la interfaz de usuario después de la eliminación.

17. `void _agregarOEditarGasto({Gasto? gastoExistente}) async { ... }`
    - **¿Qué hace?** Una función asíncrona para navegar a la pantalla de agregar/editar gasto.
    - `{Gasto? gastoExistente}`: Define un parámetro opcional y con nombre `gastoExistente`. Si se proporciona un objeto `Gasto`, significa que se está editando un gasto existente; si es `null`, se está agregando uno nuevo.
    - `final resultado = await Navigator.push(...)`: Navega a una nueva pantalla (`AddExpenseScreen`). `Navigator.push` agrega una nueva ruta a la pila de navegación y `await` espera a que la nueva pantalla se cierre y retorne un resultado.
    - `MaterialPageRoute(...)`: Define la transición a la nueva pantalla con una animación estándar de Material Design.
    - `builder: (_) => AddExpenseScreen(gasto: gastoExistente)`: Crea la instancia de `AddExpenseScreen`, pasando el `gastoExistente` si se proporcionó.
    - `if (resultado == true) { _cargarGastos(); }`: Si la pantalla `AddExpenseScreen` retorna `true` (indicando que se guardó un cambio), se recargan los gastos para actualizar la lista.

18. `@override Widget build(BuildContext context) { ... }`
    - **¿Qué hace?** Este método describe la interfaz de usuario de la pantalla `HomeScreen` basándose en el estado actual (`_gastos`). Flutter lo llama cada vez que el estado cambia (después de `setState`).
    - `return Scaffold(...)`: `Scaffold` es un widget que proporciona una estructura básica para una pantalla de Material Design (barra de aplicación, cuerpo, botón de acción flotante, etc.).
19. `appBar: AppBar(title: const Text('Gestor de Gastos'))`
    - **¿Qué hace?** Define la barra de aplicación en la parte superior de la pantalla con el título "Gestor de Gastos".
20. `body: Column(...)`
    - **¿Qué hace?** El cuerpo principal de la pantalla. `Column` organiza sus hijos verticalmente.
21. `Padding(...)`
    - **¿Qué hace?** Agrega espacio (padding) alrededor de su hijo. Aquí, agrega 16 píxeles de espacio en todos los lados alrededor del texto del total gastado.
22. `Text('Total Gastado: \$${_totalGastos.toStringAsFixed(2)}', ...)`
    - **¿Qué hace?** Muestra el total de gastos. `_totalGastos` accede al valor calculado por el getter. `toStringAsFixed(2)` formatea el número para mostrar siempre dos decimales.
23. `Expanded(...)`
    - **¿Qué hace?** Un widget que expande a su hijo para llenar el espacio disponible en la dirección principal de un `Row`, `Column` o `Flex`. Aquí, hace que la lista de gastos o el mensaje "No hay gastos" ocupe todo el espacio vertical restante después del total gastado.
24. `_gastos.isEmpty ? const Center(...) : ListView.builder(...)`
    - **¿Qué hace?** Un operador ternario que muestra un mensaje si la lista de gastos está vacía (`_gastos.isEmpty`) o muestra la lista de gastos si no está vacía.
    - `Center(child: Text('No hay gastos registrados.'))`: Centra el texto que indica que no hay gastos.
    - `ListView.builder(...)`: Un widget eficiente para construir listas largas. Construye los elementos de la lista a medida que son necesarios, lo que ahorra recursos.
    - `itemCount: _gastos.length`: Especifica cuántos elementos hay en la lista (el número de gastos).
    - `itemBuilder: (context, index) { ... }`: Una función que se llama para construir cada elemento de la lista. `index` es la posición del elemento actual.

25. `final gasto = _gastos[index];`
    - **¿Qué hace?** Obtiene el objeto `Gasto` correspondiente al índice actual de la lista.

26. `return ListTile(...)`
    - **¿Qué hace?** Un widget de Material Design que es ideal para mostrar elementos en una lista. Tiene un título, subtítulo y un widget al final (`trailing`).
    - `title: Text(gasto.descripcion)`: Muestra la descripción del gasto como título del elemento de la lista.
    - `subtitle: Text(...)`: Muestra la categoría y la fecha formateada como subtítulo.
    - `DateFormat('dd/MM/yyyy').format(gasto.fecha)`: Formatea la fecha del gasto al formato "día/mes/año".
    - `trailing: Text('\$${gasto.monto.toStringAsFixed(2)}')`: Muestra el monto del gasto formateado con el símbolo de dólar y dos decimales al final del elemento de la lista.
    - `onTap: () => _agregarOEditarGasto(gastoExistente: gasto)`: Define la acción que ocurre cuando se toca el elemento de la lista. Llama a `_agregarOEditarGasto`, pasando el gasto actual para editarlo.
    - `onLongPress: () => _eliminarGasto(gasto.id!)`: Define la acción que ocurre cuando se mantiene presionado el elemento de la lista. Llama a `_eliminarGasto` con el ID del gasto (el `!` indica que estamos seguros de que el ID no es nulo en este punto).

27. `floatingActionButton: FloatingActionButton(...)`
    - **¿Qué hace?** Define el botón de acción flotante en la esquina inferior derecha de la pantalla.
    - `onPressed: () => _agregarOEditarGasto()`: Define la acción que ocurre cuando se presiona el botón. Llama a `_agregarOEditarGasto` sin pasar un gasto, lo que indica que se va a agregar un nuevo gasto.
    - `child: const Icon(Icons.add)`: Muestra un icono de suma dentro del botón.

En resumen, este código define la estructura de datos para un gasto y crea la pantalla principal de la aplicación que muestra una lista de gastos, el total gastado, y permite agregar, editar o eliminar gastos navegando a otras pantallas y interactuando con una base de datos a través de `DBHelper`.

gasto.dart

La clase Gasto define la estructura de un gasto, incluyendo sus propiedades

```
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import 'db_helper.dart';
import 'add_expense_screen.dart';

class Gasto {
  int? id;
  final String descripcion;
  final String categoria;
  final double monto;
  final DateTime fecha;

  Gasto({
    this.id,
    required this.descripcion,
    required this.categoria,
    required this.monto,
    required this.fecha,
  });

  Map<String, dynamic> toMap() {
    return {
      'id': id,
      'descripcion': descripcion,
      'categoria': categoria,
      'monto': monto,
      'fecha': fecha.toIso8601String(),
    };
  }
```

```
  factory Gasto.fromMap(Map<String, dynamic> map) {
    return Gasto(
      id: map['id'],
      descripcion: map['descripcion'],
      categoria: map['categoria'],
      monto: map['monto'],
      fecha: DateTime.parse(map['fecha']),
    );
  }
}

class HomeScreen extends StatefulWidget {
  const HomeScreen({super.key});

  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
  List<Gasto> _gastos = [];

  @override
  void initState() {
    super.initState();
    _cargarGastos();
  }

  Future<void> _cargarGastos() async {
    final gastos = await DBHelper.obtenerGastos();
    setState(() {
      _gastos = gastos;
    });
  }

  double get _totalGastos => _gastos.fold(0.0, (sum, g) => sum + g.monto);
```

```
Future<void> _eliminarGasto(int id) async {
  final confirm = await showDialog<bool>(
    context: context,
    builder:
        (context) => AlertDialog(
          title: const Text('Confirmar eliminación'),
          content: const Text('¿Deseas eliminar este gasto?'),
          actions: [
            TextButton(
              onPressed: () => Navigator.pop(context, false),
              child: const Text('Cancelar'),
            ),
            TextButton(
              onPressed: () => Navigator.pop(context, true),
              child: const Text('Eliminar'),
            ),
          ],
        ),
  );

  if (confirm == true) {
    await DBHelper.eliminarGasto(id);
    _cargarGastos();
  }
}

void _agregarOEditarGasto({Gasto? gastoExistente}) async {
  final resultado = await Navigator.push(
    context,
    MaterialPageRoute(
      builder: (_) => AddExpenseScreen(gasto: gastoExistente),
    ),
  );
  if (resultado == true) {
    _cargarGastos();
  }
}
```

```
 @override
 Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Gestor de Gastos')),
    body: Column(
     children: [
      Padding(
       padding: const EdgeInsets.all(16),
       child: Text(
        'Total Gastado: \$${_totalGastos.toStringAsFixed(2)}',
        style: const TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
       ),
      ),
      Expanded(
       child:
          _gastos.isEmpty
             ? const Center(child: Text('No hay gastos registrados.'))
             : ListView.builder(
              itemCount: _gastos.length,
              itemBuilder: (context, index) {
               final gasto = _gastos[index];
               return ListTile(
                title: Text(gasto.descripcion),
                subtitle: Text(
                 '${gasto.categoria} -
${DateFormat('dd/MM/yyyy').format(gasto.fecha)}',
                ),
                trailing: Text('\$${gasto.monto.toStringAsFixed(2)}'),
                onTap:
                   () => _agregarOEditarGasto(gastoExistente: gasto),
                onLongPress: () => _eliminarGasto(gasto.id!),
               );
              },
             ),
      ),
     ],
```

```
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => _agregarOEditarGasto(),
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Explicación Parte por Parte:

1. **import 'package:flutter/material.dart';**
   - **¿Qué hace?** Importa la biblioteca principal de Flutter para construir interfaces de usuario con Material Design. Es necesaria para usar la mayoría de los widgets visuales.
2. **import 'package:intl/intl.dart';**
   - **¿Qué hace?** Importa la biblioteca intl (internationalization). Se utiliza aquí específicamente para formatear fechas (DateFormat). Necesitarás agregar esta dependencia a tu archivo pubspec.yaml si aún no lo has hecho (dependencies: flutter: sdk: flutter intl: ^x.y.z, donde x.y.z es la versión).
3. **import 'db_helper.dart';**
   - **¿Qué hace?** Importa un archivo local llamado db_helper.dart. Este archivo probablemente contiene la lógica para interactuar con la base de datos (guardar, obtener, eliminar gastos). La clase DBHelper se utiliza en este código para realizar operaciones de base de datos.
4. **import 'add_expense_screen.dart';**
   - **¿Qué hace?** Importa un archivo local llamado add_expense_screen.dart. Este archivo probablemente define la pantalla donde el usuario puede agregar o editar un gasto. Se utiliza aquí para navegar a esa pantalla.
5. **class Gasto { ... }**
   - **¿Qué hace?** Define una clase modelo llamada Gasto. Esta clase representa la estructura de un objeto gasto en tu aplicación. Contiene los campos que describen un gasto individual.
   - **int? id;**: Un campo opcional para el ID único del gasto, probablemente usado para la base de datos. Es opcional (?) porque un gasto nuevo aún no tendrá un ID hasta que se guarde.
   - **final String descripcion;**: La descripción del gasto (por ejemplo, "Café de la

mañana"). Es final porque una vez que se crea un objeto Gasto, su descripción no debería cambiar.
    - **final String categoria;**: La categoría del gasto (por ejemplo, "Comida", "Transporte"). También es final.
    - **final double monto;**: El monto del gasto. Es final.
    - **final DateTime fecha;**: La fecha en que ocurrió el gasto. Es final.
6. **Gasto({ ... });**
    - **¿Qué hace?** Este es el constructor de la clase Gasto. Permite crear nuevas instancias de Gasto. Los campos marcados con required deben ser proporcionados al crear un Gasto, mientras que id es opcional.
7. **Map<String, dynamic> toMap() { ... }**
    - **¿Qué hace?** Es un método dentro de la clase Gasto que convierte un objeto Gasto en un Map. Un Map es una estructura de datos clave-valor, similar a un diccionario en Python o un objeto JSON. Este método es útil para preparar los datos de un Gasto para ser guardados en una base de datos (como SQLite), ya que las bases de datos a menudo trabajan con este formato.
    - **fecha.toIso8601String()**: Convierte el objeto DateTime de la fecha a una cadena de texto en formato ISO 8601, que es un formato estándar para representar fechas y horas y es adecuado para almacenar en bases de datos.
8. **factory Gasto.fromMap(Map<String, dynamic> map) { ... }**
    - **¿Qué hace?** Este es un *constructor factory*. Los constructores factory se utilizan para crear instancias de una clase de maneras más flexibles que un constructor normal. En este caso, fromMap se utiliza para crear un objeto Gasto a partir de un Map (por ejemplo, un Map obtenido de la base de datos).
    - **DateTime.parse(map['fecha'])**: Convierte la cadena de texto de la fecha (que se guardó en formato ISO 8601) de nuevo a un objeto DateTime.
9. **class HomeScreen extends StatefulWidget { ... }**
    - **¿Qué hace?** Define la clase del widget HomeScreen.
    - **extends StatefulWidget**: Indica que HomeScreen es un StatefulWidget. Un StatefulWidget es un widget que puede cambiar su apariencia en respuesta a interacciones del usuario o cambios en los datos. A diferencia de un StatelessWidget, tiene un estado mutable asociado que puede ser modificado.
10. **const HomeScreen({super.key});**
    - **¿Qué hace?** El constructor de HomeScreen, similar al de MyApp, pasando la clave al padre.
11. **@override State<HomeScreen> createState() => _HomeScreenState();**

- **¿Qué hace?** Este método es requerido por StatefulWidget. Crea y retorna el objeto State asociado a este widget. El estado (_HomeScreenState) es donde se manejan los datos que pueden cambiar y donde se define el método build que describe la interfaz de usuario.

12. **class _HomeScreenState extends State<HomeScreen> { ... }**
    - **¿Qué hace?** Define la clase de estado para HomeScreen. El guion bajo (_) al principio del nombre (_HomeScreenState) indica que esta clase es privada y solo es accesible dentro de este archivo (home_screen.dart).
    - **List<Gasto> _gastos = [];**: Declara una lista privada llamada _gastos. Esta lista almacenará los objetos Gasto que se mostrarán en la pantalla. Es mutable porque el contenido de la lista cambiará a medida que se carguen, agreguen o eliminen gastos.
13. **@override void initState() { ... }**
    - **¿Qué hace?** Este es un método del ciclo de vida del State que se llama *una vez* cuando el objeto State se crea por primera vez. Es el lugar ideal para realizar configuraciones iniciales, como cargar datos de una base de datos o configurar listeners.
    - **super.initState();**: Siempre debes llamar a super.initState() al principio de tu implementación de initState.
    - **_cargarGastos();**: Llama a la función privada _cargarGastos para cargar los gastos existentes cuando la pantalla se inicia.
14. **Future<void> _cargarGastos() async { ... }**
    - **¿Qué hace?** Una función asíncrona (async) para cargar los gastos de la base de datos.
    - **final gastos = await DBHelper.obtenerGastos();**: Llama a un método obtenerGastos de la clase DBHelper (que interactúa con la base de datos). await espera a que la operación de la base de datos se complete antes de continuar.
    - **setState(() { _gastos = gastos; });**: Esta es la clave para actualizar la interfaz de usuario en un StatefulWidget. Cuando se llama a setState, le dices a Flutter que el estado de este widget ha cambiado y que necesita reconstruir (llamar al método build nuevamente) para reflejar los nuevos datos. Aquí, actualiza la lista _gastos con los gastos obtenidos de la base de datos.
15. **double get _totalGastos => _gastos.fold(0.0, (sum, g) => sum + g.monto);**
    - **¿Qué hace?** Define un *getter* llamado _totalGastos. Un getter es una forma de calcular un valor basado en el estado actual sin tener que llamarlo como una función. Calcula la suma total de los montos de todos los gastos en la lista

_gastos usando el método fold.

16. **Future<void> _eliminarGasto(int id) async { ... }**
    - **¿Qué hace?** Una función asíncrona para eliminar un gasto por su ID.
    - **final confirm = await showDialog<bool>(...);**: Muestra un diálogo de confirmación al usuario antes de eliminar un gasto. showDialog es una función de Flutter para mostrar diálogos. El await espera a que el usuario interactúe con el diálogo y retorne un valor (true si confirma, false si cancela).
    - **AlertDialog(...)**: El widget que define la apariencia del diálogo, con un título, contenido y acciones (botones).
    - **Navigator.pop(context, false) / Navigator.pop(context, true)**: Cuando se presiona un botón en el diálogo, Navigator.pop cierra el diálogo y retorna el valor especificado (false o true).
    - **if (confirm == true) { ... }**: Si el usuario confirmó la eliminación...
    - **await DBHelper.eliminarGasto(id);**: Llama al método eliminarGasto de DBHelper para eliminar el gasto de la base de datos.
    - **_cargarGastos();**: Vuelve a cargar la lista de gastos desde la base de datos para actualizar la interfaz de usuario después de la eliminación.
17. **void _agregarOEditarGasto({Gasto? gastoExistente}) async { ... }**
    - **¿Qué hace?** Una función asíncrona para navegar a la pantalla de agregar/editar gasto.
    - **{Gasto? gastoExistente}**: Define un parámetro opcional y con nombre gastoExistente. Si se proporciona un objeto Gasto, significa que se está editando un gasto existente; si es null, se está agregando uno nuevo.
    - **final resultado = await Navigator.push(...)**: Navega a una nueva pantalla (AddExpenseScreen). Navigator.push agrega una nueva ruta a la pila de navegación y await espera a que la nueva pantalla se cierre y retorne un resultado.
    - **MaterialPageRoute(...)**: Define la transición a la nueva pantalla con una animación estándar de Material Design.
    - **builder: (_) => AddExpenseScreen(gasto: gastoExistente)**: Crea la instancia de AddExpenseScreen, pasando el gastoExistente si se proporcionó.
    - **if (resultado == true) { _cargarGastos(); }**: Si la pantalla AddExpenseScreen retorna true (indicando que se guardó un cambio), se recargan los gastos para actualizar la lista.
18. **@override Widget build(BuildContext context) { ... }**
    - **¿Qué hace?** Este método describe la interfaz de usuario de la pantalla HomeScreen basándose en el estado actual (_gastos). Flutter lo llama cada

vez que el estado cambia (después de setState).
    - **return Scaffold(...)**: Scaffold es un widget que proporciona una estructura básica para una pantalla de Material Design (barra de aplicación, cuerpo, botón de acción flotante, etc.).
19. **appBar: AppBar(title: const Text('Gestor de Gastos'))**
    - **¿Qué hace?** Define la barra de aplicación en la parte superior de la pantalla con el título "Gestor de Gastos".
20. **body: Column(...)**
    - **¿Qué hace?** El cuerpo principal de la pantalla. Column organiza sus hijos verticalmente.
21. **Padding(...)**
    - **¿Qué hace?** Agrega espacio (padding) alrededor de su hijo. Aquí, agrega 16 píxeles de espacio en todos los lados alrededor del texto del total gastado.
22. **Text('Total Gastado: \$${_totalGastos.toStringAsFixed(2)}', ...)**
    - **¿Qué hace?** Muestra el total de gastos. _totalGastos accede al valor calculado por el getter. toStringAsFixed(2) formatea el número para mostrar siempre dos decimales.
23. **Expanded(...)**
    - **¿Qué hace?** Un widget que expande a su hijo para llenar el espacio disponible en la dirección principal de un Row, Column o Flex. Aquí, hace que la lista de gastos o el mensaje "No hay gastos" ocupe todo el espacio vertical restante después del total gastado.
24. **_gastos.isEmpty ? const Center(...) : ListView.builder(...)**
    - **¿Qué hace?** Un operador ternario que muestra un mensaje si la lista de gastos está vacía (_gastos.isEmpty) o muestra la lista de gastos si no está vacía.
    - **Center(child: Text('No hay gastos registrados.'))**: Centra el texto que indica que no hay gastos.
    - **ListView.builder(...)**: Un widget eficiente para construir listas largas. Construye los elementos de la lista a medida que son necesarios, lo que ahorra recursos.
    - **itemCount: _gastos.length**: Especifica cuántos elementos hay en la lista (el número de gastos).
    - **itemBuilder: (context, index) { ... }**: Una función que se llama para construir cada elemento de la lista. index es la posición del elemento actual.
25. **final gasto = _gastos[index];**
    - **¿Qué hace?** Obtiene el objeto Gasto correspondiente al índice actual de la

lista.

26. **return ListTile(...)**
    - **¿Qué hace?** Un widget de Material Design que es ideal para mostrar elementos en una lista. Tiene un título, subtítulo y un widget al final (trailing).
    - **title: Text(gasto.descripcion)**: Muestra la descripción del gasto como título del elemento de la lista.
    - **subtitle: Text(...)**: Muestra la categoría y la fecha formateada como subtítulo.
    - **DateFormat('dd/MM/yyyy').format(gasto.fecha)**: Formatea la fecha del gasto al formato "día/mes/año".
    - **trailing: Text('\$${gasto.monto.toStringAsFixed(2)}')**: Muestra el monto del gasto formateado con el símbolo de dólar y dos decimales al final del elemento de la lista.
    - **onTap: () => _agregarOEditarGasto(gastoExistente: gasto)**: Define la acción que ocurre cuando se toca el elemento de la lista. Llama a _agregarOEditarGasto, pasando el gasto actual para editarlo.
    - **onLongPress: () => _eliminarGasto(gasto.id!)**: Define la acción que ocurre cuando se mantiene presionado el elemento de la lista. Llama a _eliminarGasto con el ID del gasto (el ! indica que estamos seguros de que el ID no es nulo en este punto).
27. **floatingActionButton: FloatingActionButton(...)**
    - **¿Qué hace?** Define el botón de acción flotante en la esquina inferior derecha de la pantalla.
    - **onPressed: () => _agregarOEditarGasto()**: Define la acción que ocurre cuando se presiona el botón. Llama a _agregarOEditarGasto sin pasar un gasto, lo que indica que se va a agregar un nuevo gasto.
    - **child: const Icon(Icons.add)**: Muestra un icono de suma dentro del botón.
    -

En resumen, este código define la estructura de datos para un gasto y crea la pantalla principal de la aplicación que muestra una lista de gastos, el total gastado, y permite agregar, editar o eliminar gastos navegando a otras pantallas y interactuando con una base de datos a través de DBHelper.

DBHelper:

encargado de la interacción con la base de datos.

```
import 'dart:async';
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';
import 'package:path_provider/path_provider.dart';
import 'dart:io';

import 'home_screen.dart'; // Usamos la clase Gasto

class DBHelper {
  static Database? _database;

  static Future<Database> getDatabase() async {
    if (_database != null) return _database!;
    return await _initDB();
  }

  static Future<Database> _initDB() async {
    Directory documentsDirectory = await getApplicationDocumentsDirectory();
    final path = join(documentsDirectory.path, 'gastos.db');

    return openDatabase(
      path,
      version: 1,
      onCreate: (db, version) async {
        await db.execute('''
          CREATE TABLE gastos(
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            descripcion TEXT,
            monto REAL,
            categoria TEXT,
            fecha TEXT
          )
        ''');
      },
    );
  }
```

```
 static Future<int> insertarGasto(Gasto gasto) async {
  final db = await getDatabase();
  return await db.insert('gastos', gasto.toMap());
 }

 static Future<List<Gasto>> obtenerGastos() async {
  final db = await getDatabase();
  final List<Map<String, dynamic>> maps = await db.query('gastos');
  return List.generate(maps.length, (i) => Gasto.fromMap(maps[i]));
 }

 static Future<int> actualizarGasto(Gasto gasto) async {
  final db = await getDatabase();
  return await db.update(
    'gastos',
    gasto.toMap(),
    where: 'id = ?',
    whereArgs: [gasto.id],
  );
 }

 static Future<int> eliminarGasto(int id) async {
  final db = await getDatabase();
  return await db.delete('gastos', where: 'id = ?', whereArgs: [id]);
 }
}
```

Explicación Parte por Parte:

1. **import 'dart:async';**
    - **¿Qué hace?** Importa la biblioteca dart:async, que es fundamental para trabajar con operaciones asíncronas en Dart, como Future y async/await, que son esenciales para las operaciones de base de datos.
2. **import 'package:sqflite/sqflite.dart';**
    - **¿Qué hace?** Importa la biblioteca sqflite, que es un plugin de Flutter para trabajar con bases de datos SQLite en dispositivos móviles. Proporciona las funciones para abrir la base de datos, ejecutar comandos SQL e interactuar con los datos. Necesitarás agregar esta dependencia a tu archivo pubspec.yaml.
3. **import 'package:path/path.dart';**
    - **¿Qué hace?** Importa la biblioteca path, que proporciona utilidades para manipular rutas de archivos y directorios de manera independiente del

sistema operativo. Se utiliza aquí para unir la ruta del directorio de documentos con el nombre del archivo de la base de datos. Necesitarás agregar esta dependencia a tu archivo pubspec.yaml.

4. **import 'package:path_provider/path_provider.dart';**
    - **¿Qué hace?** Importa la biblioteca path_provider, que proporciona acceso a rutas comunes del sistema de archivos en el dispositivo (como el directorio de documentos de la aplicación). Se utiliza para encontrar dónde guardar el archivo de la base de datos. Necesitarás agregar esta dependencia a tu archivo pubspec.yaml.
5. **import 'dart:io';**
    - **¿Qué hace?** Importa la biblioteca dart:io, que proporciona acceso a operaciones de entrada/salida (I/O), incluyendo la manipulación de directorios (Directory).
6. **import 'home_screen.dart'; // Usamos la clase Gasto**
    - **¿Qué hace?** Importa el archivo home_screen.dart. Aunque este archivo define la pantalla principal, lo importante aquí es que probablemente es donde se define la clase Gasto que se utiliza en este archivo db_helper.dart para representar los datos que se guardan y recuperan de la base de datos.
7. **class DBHelper { ... }**
    - **¿Qué hace?** Define una clase llamada DBHelper. Esta clase actúa como una clase de utilidad estática (no necesitas crear una instancia de ella) que encapsula toda la lógica relacionada con la base de datos.
8. **static Database? _database;**
    - **¿Qué hace?** Declara una variable estática y privada (_) llamada _database de tipo Database?.
    - static: Significa que esta variable pertenece a la clase DBHelper en sí misma, no a una instancia específica de la clase. Habrá una única instancia de _database compartida por toda la aplicación.
    - Database?: Indica que puede contener un objeto Database (la conexión a la base de datos) o ser null inicialmente.
9. **static Future<Database> getDatabase() async { ... }**
    - **¿Qué hace?** Este es un método estático y asíncrono que proporciona acceso a la instancia de la base de datos.
    - **if (_database != null) return _database!;**: Comprueba si la base de datos ya ha sido inicializada (_database no es null). Si es así, retorna la instancia existente (_database!, el ! es el operador "bang" que afirma que el valor no es nulo). Esto asegura que solo se abra una conexión a la base de datos en toda

la aplicación (patrón Singleton).
    - **return await _initDB();**: Si la base de datos aún no está inicializada, llama al método privado _initDB() para inicializarla y espera (await) a que se complete antes de retornar la base de datos recién inicializada.
10. **static Future<Database> _initDB() async { ... }**
    - **¿Qué hace?** Este es un método estático y asíncrono privado que se encarga de abrir y, si es necesario, crear la base de datos.
    - **Directory documentsDirectory = await getApplicationDocumentsDirectory();**: Obtiene el directorio donde la aplicación puede almacenar documentos.
    - **final path = join(documentsDirectory.path, 'gastos.db');**: Construye la ruta completa al archivo de la base de datos. join es una función de la biblioteca path que une partes de una ruta de manera segura para el sistema operativo. El archivo de la base de datos se llamará gastos.db.
    - **return openDatabase(...)**: Llama a la función openDatabase de sqflite para abrir la base de datos.
        - **path**: La ruta al archivo de la base de datos.
        - **version: 1**: La versión de la base de datos. Esto es importante para las migraciones (actualizaciones de la estructura de la base de datos) en el futuro.
        - **onCreate: (db, version) async { ... }**: Una función de callback que se ejecuta solo la primera vez que se crea la base de datos (cuando no existe el archivo gastos.db).
            - **await db.execute(''' CREATE TABLE gastos(...) ''');**: Ejecuta una sentencia SQL para crear la tabla gastos.
                - CREATE TABLE gastos: Crea una tabla llamada gastos.
                - id INTEGER PRIMARY KEY AUTOINCREMENT: Define una columna id de tipo entero que es la clave primaria y se incrementa automáticamente con cada nueva fila.
                - descripcion TEXT, monto REAL, categoria TEXT, fecha TEXT: Define las otras columnas para almacenar la descripción, monto (números de punto flotante), categoría y fecha (almacenada como texto en formato ISO 8601, como vimos en la clase Gasto).
11. **static Future<int> insertarGasto(Gasto gasto) async { ... }**
    - **¿Qué hace?** Un método estático y asíncrono para insertar un nuevo gasto en la base de datos.
    - **final db = await getDatabase();**: Obtiene la instancia de la base de datos (la

inicializa si es necesario).
    - **return await db.insert('gastos', gasto.toMap());**: Llama al método insert de la base de datos.
        - 'gastos': El nombre de la tabla donde insertar.
        - gasto.toMap(): Convierte el objeto Gasto a un Map utilizando el método toMap() que definimos en la clase Gasto. El método insert espera un Map.
    - Retorna un Future<int> que se completa con el ID de la fila recién insertada.
12. **static Future<List<Gasto>> obtenerGastos() async { ... }**
    - **¿Qué hace?** Un método estático y asíncrono para obtener todos los gastos de la base de datos.
    - **final db = await getDatabase();**: Obtiene la instancia de la base de datos.
    - **final List<Map<String, dynamic>> maps = await db.query('gastos');**: Ejecuta una consulta (query) en la tabla gastos para obtener todas las filas. El resultado es una lista de Maps, donde cada Map representa una fila de la tabla.
    - **return List.generate(maps.length, (i) => Gasto.fromMap(maps[i]));**: Convierte la lista de Maps a una lista de objetos Gasto.
        - List.generate(maps.length, ...): Crea una nueva lista con el mismo número de elementos que la lista de Maps.
        - (i) => Gasto.fromMap(maps[i]): Para cada Map en la lista (en el índice i), llama al constructor factory Gasto.fromMap() para crear un objeto Gasto a partir de ese Map.
13. **static Future<int> actualizarGasto(Gasto gasto) async { ... }**
    - **¿Qué hace?** Un método estático y asíncrono para actualizar un gasto existente en la base de datos.
    - **final db = await getDatabase();**: Obtiene la instancia de la base de datos.
    - **return await db.update(...)**: Llama al método update de la base de datos.
        - 'gastos': El nombre de la tabla.
        - gasto.toMap(): Los nuevos valores para la fila, proporcionados como un Map.
        - where: 'id = ?': La cláusula WHERE para especificar qué filas actualizar. ? es un marcador de posición para evitar inyección SQL.
        - whereArgs: [gasto.id]: Proporciona los argumentos para los marcadores de posición en la cláusula WHERE. Aquí, el ID del gasto a actualizar.
    - Retorna un Future<int> que se completa con el número de filas afectadas por la actualización.

14. **static Future<int> eliminarGasto(int id) async { ... }**
    - **¿Qué hace?** Un método estático y asíncrono para eliminar un gasto de la base de datos por su ID.
    - **final db = await getDatabase();**: Obtiene la instancia de la base de datos.
    - **return await db.delete('gastos', where: 'id = ?', whereArgs: [id]);**: Llama al método delete de la base de datos, especificando la tabla y la cláusula WHERE para identificar la fila a eliminar.
    - Retorna un Future<int> que se completa con el número de filas eliminadas.

En resumen, la clase DBHelper proporciona una capa de abstracción para interactuar con la base de datos SQLite de tu aplicación. Se encarga de inicializar la base de datos (creando la tabla si es la primera vez), y ofrece métodos convenientes para insertar, obtener, actualizar y eliminar objetos Gasto de la base de datos, utilizando la clase Gasto y sus métodos toMap y fromMap para la conversión de datos.

código Flutter para la clase DatabaseHelper:

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';
import 'home_screen.dart'; // Asegúrate de que Gasto esté definido aquí

class DatabaseHelper {
  static final DatabaseHelper _instance = DatabaseHelper._internal();
  factory DatabaseHelper() => _instance;
  DatabaseHelper._internal();

  static Database? _database;

  Future<Database> get database async {
    _database ??= await _initDB('gastos.db');
    return _database!;
  }

  Future<Database> _initDB(String fileName) async {
```

```
  final path = join(await getDatabasesPath(), fileName);
  return await openDatabase(path, version: 1, onCreate: _onCreate);
}

Future<void> _onCreate(Database db, int version) async {
  await db.execute('''
    CREATE TABLE gastos(
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      descripcion TEXT,
      categoria TEXT,
      monto REAL,
      fecha TEXT
    )
  ''');
}

Future<int> insertGasto(Gasto gasto) async {
  final db = await database;
  return await db.insert('gastos', gasto.toMap());
}

Future<List<Gasto>> getGastos() async {
  final db = await database;
  final result = await db.query('gastos', orderBy: 'fecha DESC');
  return result.map((e) => Gasto.fromMap(e)).toList();
}

Future<int> updateGasto(Gasto gasto) async {
  final db = await database;
  return await db.update(
    'gastos',
    gasto.toMap(),
    where: 'id = ?',
    whereArgs: [gasto.id],
  );
}
```

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

Explicación Parte por Parte:

1. **import 'package:sqflite/sqflite.dart';**
   - **¿Qué hace?** Importa la biblioteca sqflite, el plugin esencial para trabajar con bases de datos SQLite en aplicaciones Flutter. Proporciona las funcionalidades necesarias para interactuar con la base de datos.
2. **import 'package:path/path.dart';**
   - **¿Qué hace?** Importa la biblioteca path, utilizada para construir rutas de archivos y directorios de manera segura y compatible con diferentes sistemas operativos. Aquí se usa para determinar la ubicación del archivo de la base de datos.
3. **import 'home_screen.dart'; // Asegúrate de que Gasto esté definido aquí**
   - **¿Qué hace?** Importa el archivo home_screen.dart. Esto es necesario porque la clase DatabaseHelper interactúa con objetos de tipo Gasto, que se asume están definidos en home_screen.dart (o en un archivo importado por él).
4. **class DatabaseHelper { ... }**
   - **¿Qué hace?** Define la clase DatabaseHelper. Esta clase contiene la lógica para gestionar la base de datos de gastos.
5. **static final DatabaseHelper _instance = DatabaseHelper._internal();**
   - **¿Qué hace?** Esta línea es parte de la implementación del patrón de diseño **Singleton**. Crea una única instancia privada (_instance) de DatabaseHelper cuando la clase se carga por primera vez. final asegura que esta instancia no pueda ser reasignada.
6. **factory DatabaseHelper() => _instance;**
   - **¿Qué hace?** Define un constructor factory. Cuando se llama a DatabaseHelper(), este constructor factory *siempre* devuelve la misma instancia privada _instance creada anteriormente. Esto garantiza que solo exista una única instancia de DatabaseHelper en toda la aplicación.
7. **DatabaseHelper._internal();**
   - **¿Qué hace?** Define un constructor privado con nombre (_internal). Al ser privado (por el guion bajo _), evita que se puedan crear nuevas instancias de

DatabaseHelper directamente desde fuera de la clase, forzando el uso del constructor factory para obtener la única instancia.

8. **static Database? _database;**
    - **¿Qué hace?** Declara una variable estática y privada (_) de tipo Database?. Esta variable almacenará la referencia a la base de datos abierta. Es nullable (?) porque inicialmente no hay ninguna base de datos abierta.
9. **Future<Database> get database async { ... }**
    - **¿Qué hace?** Define un *getter* asíncrono llamado database. Este getter es la forma recomendada de obtener la instancia de la base de datos en esta implementación Singleton.
    - **_database ??= await _initDB('gastos.db');**: Utiliza el operador ??= (asignación si es nulo). Si _database es null, llama al método _initDB para inicializar la base de datos, espera a que termine (await) y asigna el resultado a _database. Si _database ya tiene un valor (no es nulo), esta línea no hace nada. Esto asegura que la base de datos se inicialice solo la primera vez que se accede a ella.
    - **return _database!;**: Retorna la instancia de la base de datos. El operador ! (bang) se utiliza aquí para afirmar que _database no es null en este punto, ya que la línea anterior garantizó su inicialización si era necesario.
10. **Future<Database> _initDB(String fileName) async { ... }**
    - **¿Qué hace?** Un método privado y asíncrono que se encarga de abrir o crear la base de datos.
    - **final path = join(await getDatabasesPath(), fileName);**: Obtiene la ruta estándar donde sqflite almacena las bases de datos en el dispositivo (getDatabasesPath()) y la combina con el nombre del archivo de la base de datos proporcionado (fileName, que será 'gastos.db').
    - **return await openDatabase(path, version: 1, onCreate: _onCreate);**: Llama a la función openDatabase para abrir la base de datos en la ruta especificada.
        - path: La ruta completa al archivo de la base de datos.
        - version: 1: La versión de la base de datos. Útil para futuras migraciones.
        - onCreate: _onCreate: Especifica la función (_onCreate) que se ejecutará la primera vez que se cree el archivo de la base de datos.
11. **Future<void> _onCreate(Database db, int version) async { ... }**
    - **¿Qué hace?** Un método privado y asíncrono que se ejecuta cuando la base de datos se crea por primera vez.
    - **await db.execute(''' CREATE TABLE gastos(...) ''');**: Ejecuta una sentencia

SQL para crear la tabla llamada gastos.
    - CREATE TABLE gastos: Crea la tabla.
    - id INTEGER PRIMARY KEY AUTOINCREMENT: Define la columna id como entero, clave primaria y que se auto-incrementa.
    - descripcion TEXT, categoria TEXT, monto REAL, fecha TEXT: Define las otras columnas para almacenar la descripción, categoría, monto (como número real) y fecha (como texto, probablemente en formato ISO 8601).
12. **Future<int> insertGasto(Gasto gasto) async { ... }**
    - **¿Qué hace?** Un método asíncrono para insertar un nuevo gasto en la base de datos.
    - **final db = await database;**: Obtiene la instancia de la base de datos utilizando el getter database.
    - **return await db.insert('gastos', gasto.toMap());**: Llama al método insert de la base de datos. Inserta el Map que representa el objeto Gasto (obtenido con gasto.toMap()) en la tabla gastos. Retorna un Future<int> que se completa con el ID de la fila recién insertada.
13. **Future<List<Gasto>> getGastos() async { ... }**
    - **¿Qué hace?** Un método asíncrono para obtener todos los gastos de la base de datos.
    - **final db = await database;**: Obtiene la instancia de la base de datos.
    - **final result = await db.query('gastos', orderBy: 'fecha DESC');**: Ejecuta una consulta (query) en la tabla gastos. orderBy: 'fecha DESC' especifica que los resultados deben ordenarse por la columna fecha en orden descendente (los gastos más recientes primero). El resultado es una lista de Maps.
    - **return result.map((e) => Gasto.fromMap(e)).toList();**: Procesa la lista de Maps obtenida.
        - result.map((e) => Gasto.fromMap(e)): Itera sobre cada Map (e) en la lista result y utiliza el constructor factory Gasto.fromMap para convertir cada Map en un objeto Gasto.
        - .toList(): Convierte el resultado del mapeo (un iterable) de nuevo a una lista de objetos Gasto.
14. **Future<int> updateGasto(Gasto gasto) async { ... }**
    - **¿Qué hace?** Un método asíncrono para actualizar un gasto existente en la base de datos.
    - **final db = await database;**: Obtiene la instancia de la base de datos.
    - **return await db.update(...)**: Llama al método update de la base de datos.
        - 'gastos': La tabla a actualizar.

        - gasto.toMap(): Los nuevos valores para la fila, proporcionados como un Map.
        - where: 'id = ?': La cláusula WHERE para especificar qué fila actualizar, usando un marcador de posición ?.
        - whereArgs: [gasto.id]: Proporciona el valor para el marcador de posición (?), que es el ID del gasto a actualizar.
    - Retorna un Future<int> que se completa con el número de filas afectadas por la actualización.
15. **Future<int> deleteGasto(int id) async { ... }**
    - **¿Qué hace?** Un método asíncrono para eliminar un gasto de la base de datos por su ID.
    - **final db = await database;**: Obtiene la instancia de la base de datos.
    - **return await db.delete('gastos', where: 'id = ?', whereArgs: [id]);**: Llama al método delete de la base de datos, especificando la tabla y la cláusula WHERE con el ID del gasto a eliminar.
    - Retorna un Future<int> que se completa con el número de filas eliminadas.

Esta clase DatabaseHelper proporciona una forma robusta y bien estructurada para manejar las operaciones de la base de datos en tu aplicación Flutter, asegurando una única conexión a la base de datos a través del patrón Singleton y ofreciendo métodos claros para las operaciones CRUD.

Add_expense_screen.dart

AddExpenseScreen:

```
// ignore_for_file: use_build_context_synchronously

import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import 'home_screen.dart'; // Necesario para la clase Gasto
import 'db_helper.dart'; // O DatabaseHelper, dependiendo de tu implementación

class AddExpenseScreen extends StatefulWidget {
  final Gasto? gasto;

  const AddExpenseScreen({super.key, this.gasto});

  @override
  State<AddExpenseScreen> createState() => _AddExpenseScreenState();
}

class _AddExpenseScreenState extends State<AddExpenseScreen> {
  final _formKey = GlobalKey<FormState>();
  late TextEditingController _descripcionController;
  late TextEditingController _montoController;
  String? _categoriaSeleccionada;
  DateTime? _fechaSeleccionada;

  final List<String> _categorias = [
    'Comida',
    'Transporte',
    'Entretenimiento',
    'Otros',
  ];
```

```
@override
void initState() {
  super.initState();
  final gasto = widget.gasto;
  _descripcionController = TextEditingController(
    text: gasto?.descripcion ?? '',
  );
  _montoController = TextEditingController(
    text: gasto != null ? gasto.monto.toString() : '',
  );
  _categoriaSeleccionada = gasto?.categoria;
  _fechaSeleccionada = gasto?.fecha;
}

void _seleccionarFecha() async {
  final DateTime? picked = await showDatePicker(
    context: context,
    initialDate: _fechaSeleccionada ?? DateTime.now(),
    firstDate: DateTime(2020),
    lastDate: DateTime(2100),
  );
  if (picked != null) {
    setState(() {
      _fechaSeleccionada = picked;
    });
  }
}

void _guardarGasto() async {
  if (_formKey.currentState!.validate() && _fechaSeleccionada != null) {
    final nuevoGasto = Gasto(
      id: widget.gasto?.id,
      descripcion: _descripcionController.text,
      monto: double.parse(_montoController.text),
      categoria: _categoriaSeleccionada!,
      fecha: _fechaSeleccionada!,
    );
```

```
    // Usar DBHelper o DatabaseHelper según tu implementación
    if (widget.gasto == null) {
      await DBHelper.insertarGasto(nuevoGasto); // O
DatabaseHelper().insertGasto(nuevoGasto);
    } else {
      await DBHelper.actualizarGasto(nuevoGasto); // O
DatabaseHelper().updateGasto(nuevoGasto);
    }

    Navigator.pop(context, true);
  }
 }

 @override
 void dispose() {
  _descripcionController.dispose();
  _montoController.dispose();
  super.dispose();
 }

 @override
 Widget build(BuildContext context) {
  final esEdicion = widget.gasto != null;

  return Scaffold(
   appBar: AppBar(title: Text(esEdicion ? 'Editar Gasto' : 'Agregar Gasto')),
   body: Padding(
    padding: const EdgeInsets.all(16.0),
    child: Form(
     key: _formKey,
     child: ListView(
      children: [
       TextFormField(
        controller: _descripcionController,
        decoration: const InputDecoration(labelText: 'Descripción'),
        validator:
```

```
      (value) =>
          value == null || value.isEmpty
              ? 'Campo requerido'
              : null,
),
const SizedBox(height: 16),
DropdownButtonFormField<String>(
  value: _categoriaSeleccionada,
  items:
      _categorias
          .map(
            (cat) =>
                DropdownMenuItem(value: cat, child: Text(cat)),
          )
          .toList(),
  onChanged:
      (value) => setState(() => _categoriaSeleccionada = value),
  decoration: const InputDecoration(labelText: 'Categoría'),
  validator:
      (value) =>
          value == null ? 'Seleccione una categoría' : null,
),
const SizedBox(height: 16),
TextFormField(
  controller: _montoController,
  decoration: const InputDecoration(labelText: 'Monto'),
  keyboardType: TextInputType.number,
  validator: (value) {
    if (value == null || value.isEmpty) return 'Campo requerido';
    final num? parsed = num.tryParse(value);
    return parsed == null ? 'Ingrese un número válido' : null;
  },
),
const SizedBox(height: 16),
Row(
  children: [
    Expanded(
```

```
                child: Text(
                  _fechaSeleccionada == null
                      ? 'Seleccione una fecha'
                      : 'Fecha: ${DateFormat('dd/MM/yyyy').format(_fechaSeleccionada!)}',
                ),
              ),
              TextButton(
                onPressed: _seleccionarFecha,
                child: const Text('Elegir Fecha'),
              ),
            ],
          ),
          const SizedBox(height: 24),
          ElevatedButton(
            onPressed: _guardarGasto,
            child: Text(esEdicion ? 'Actualizar Gasto' : 'Guardar Gasto'),
          ),
        ],
      ),
    ),
  ),
 );
 }
}
```

Explicación Parte por Parte:

1. **// ignore_for_file: use_build_context_synchronously**
   - **¿Qué hace?** Esta es una directiva de análisis de Dart. Le dice al analizador que ignore la advertencia use_build_context_synchronously. Esta advertencia aparece a menudo cuando se usa BuildContext después de una operación asíncrona (await) que podría hacer que el widget ya no esté montado en el árbol de widgets. En algunos casos, como después de showDatePicker o Navigator.pop, es seguro usar el context, pero el analizador no puede saberlo con certeza, por lo que se ignora la advertencia.
2. **import 'package:flutter/material.dart';**
   - **¿Qué hace?** Importa la biblioteca principal de Flutter Material Design,

necesaria para construir la interfaz de usuario.

3. **import 'package:intl/intl.dart';**
   - **¿Qué hace?** Importa la biblioteca intl para formatear fechas, utilizada aquí para mostrar la fecha seleccionada en un formato legible.
4. **import 'home_screen.dart'; // Necesario para la clase Gasto**
   - **¿Qué hace?** Importa home_screen.dart, que contiene la definición de la clase Gasto. Esta clase es necesaria para crear y manipular objetos de gasto en esta pantalla.
5. **import 'db_helper.dart'; // O DatabaseHelper, dependiendo de tu implementación**
   - **¿Qué hace?** Importa el archivo que contiene la lógica para interactuar con la base de datos. Dependiendo de la implementación que estés usando, podría ser db_helper.dart o database_helper.dart.
6. **class AddExpenseScreen extends StatefulWidget { ... }**
   - **¿Qué hace?** Define la clase AddExpenseScreen como un StatefulWidget. Esto es necesario porque la pantalla necesita mantener y actualizar el estado de los campos del formulario (texto ingresado, categoría seleccionada, fecha seleccionada).
7. **final Gasto? gasto;**
   - **¿Qué hace?** Declara una variable final y nullable (?) llamada gasto. Este campo se utiliza para pasar un objeto Gasto existente a esta pantalla si se está editando un gasto. Si se está agregando un nuevo gasto, este campo será null.
8. **const AddExpenseScreen({super.key, this.gasto});**
   - **¿Qué hace?** El constructor de AddExpenseScreen. Permite crear una instancia de la pantalla, opcionalmente pasando un gasto existente.
9. **@override State<AddExpenseScreen> createState() => _AddExpenseScreenState();**
   - **¿Qué hace?** Crea y retorna el objeto State asociado a este StatefulWidget.
10. **class _AddExpenseScreenState extends State<AddExpenseScreen> { ... }**
    - **¿Qué hace?** Define la clase de estado para AddExpenseScreen. Aquí es donde se manejan los datos mutables y se define el método build. El guion bajo _ lo hace privado.
11. **final _formKey = GlobalKey<FormState>();**
    - **¿Qué hace?** Crea una GlobalKey para el widget Form. Las GlobalKey son útiles para acceder al estado de un widget desde cualquier parte del árbol de widgets. Aquí, se usa para acceder al estado del formulario y validar sus

campos.

12. **late TextEditingController _descripcionController;**
    - **¿Qué hace?** Declara un TextEditingController para el campo de texto de la descripción. Un TextEditingController permite controlar el texto que se muestra en un TextField o TextFormField y reaccionar a los cambios. late indica que la variable se inicializará más tarde (en initState).
13. **late TextEditingController _montoController;**
    - **¿Qué hace?** Declara un TextEditingController para el campo de texto del monto. También late.
14. **String? _categoriaSeleccionada;**
    - **¿Qué hace?** Declara una variable nullable para almacenar la categoría seleccionada en el DropdownButtonFormField.
15. **DateTime? _fechaSeleccionada;**
    - **¿Qué hace?** Declara una variable nullable para almacenar la fecha seleccionada por el usuario.
16. **final List<String> _categorias = [...];**
    - **¿Qué hace?** Define una lista final de cadenas de texto que representan las categorías disponibles para los gastos.
17. **@override void initState() { ... }**
    - **¿Qué hace?** Este método del ciclo de vida se llama una vez cuando el objeto State se crea. Es el lugar para inicializar variables que dependen del widget asociado (como widget.gasto) o realizar configuraciones iniciales.
    - **final gasto = widget.gasto;**: Obtiene el objeto Gasto pasado al widget (si existe).
    - **_descripcionController = TextEditingController(text: gasto?.descripcion ?? '');**: Inicializa el controlador de descripción. Si gasto no es nulo, establece el texto inicial del controlador a la descripción del gasto; de lo contrario, lo establece a una cadena vacía. El operador ?? (null-aware) proporciona un valor por defecto si la expresión de la izquierda es nula.
    - **_montoController = TextEditingController(text: gasto != null ? gasto.monto.toString() : '');**: Inicializa el controlador de monto. Si gasto no es nulo, establece el texto inicial al monto del gasto convertido a cadena; de lo contrario, a una cadena vacía.
    - **_categoriaSeleccionada = gasto?.categoria;**: Inicializa la categoría seleccionada. Si gasto no es nulo, establece la categoría seleccionada a la categoría del gasto.
    - **_fechaSeleccionada = gasto?.fecha;**: Inicializa la fecha seleccionada. Si

gasto no es nulo, establece la fecha seleccionada a la fecha del gasto.

18. **void _seleccionarFecha() async { ... }**
    - **¿Qué hace?** Una función asíncrona para mostrar un selector de fecha.
    - **final DateTime? picked = await showDatePicker(...)**: Llama a la función showDatePicker de Flutter para mostrar el diálogo del selector de fecha. await espera a que el usuario seleccione una fecha o cancele.
        - context: El contexto del widget.
        - initialDate: La fecha que se muestra inicialmente en el selector (la fecha seleccionada previamente o la fecha actual si no hay ninguna).
        - firstDate y lastDate: Definen el rango de fechas disponibles para seleccionar.
    - **if (picked != null) { ... }**: Si el usuario seleccionó una fecha (picked no es nulo)...
    - **setState(() { _fechaSeleccionada = picked; });**: Actualiza la variable _fechaSeleccionada con la fecha seleccionada y llama a setState para indicarle a Flutter que el estado ha cambiado y la interfaz de usuario debe reconstruirse para mostrar la fecha seleccionada.
19. **void _guardarGasto() async { ... }**
    - **¿Qué hace?** Una función asíncrona para validar el formulario, crear un objeto Gasto y guardarlo o actualizarlo en la base de datos.
    - **if (_formKey.currentState!.validate() && _fechaSeleccionada != null) { ... }**: Valida el formulario. _formKey.currentState!.validate() ejecuta la lógica de validación definida en cada TextFormField y DropdownButtonFormField. También comprueba si se ha seleccionado una fecha.
    - **final nuevoGasto = Gasto(...)**: Crea un nuevo objeto Gasto utilizando los datos ingresados en los campos del formulario y la fecha seleccionada.
        - id: widget.gasto?.id: Si se está editando (widget.gasto no es nulo), usa el ID existente; de lo contrario, el ID será null (lo que indica una nueva inserción).
        - descripcion: _descripcionController.text: Obtiene el texto del controlador de descripción.
        - monto: double.parse(_montoController.text): Obtiene el texto del controlador de monto y lo convierte a un número de punto flotante (double).
        - categoria: _categoriaSeleccionada!: Usa la categoría seleccionada. El ! afirma que no es nulo porque la validación ya lo comprobó.
        - fecha: _fechaSeleccionada!: Usa la fecha seleccionada. El ! afirma que no

es nulo.
    - **if (widget.gasto == null) { ... } else { ... }**: Comprueba si se está agregando un nuevo gasto (widget.gasto es nulo) o editando uno existente.
        - **await DBHelper.insertarGasto(nuevoGasto);**: Si es una nueva inserción, llama al método insertarGasto (o insertGasto si usas DatabaseHelper) de tu clase de ayuda de base de datos.
        - **await DBHelper.actualizarGasto(nuevoGasto);**: Si es una edición, llama al método actualizarGasto (o updateGasto).
    - **Navigator.pop(context, true);**: Cierra la pantalla actual y regresa a la pantalla anterior (probablemente HomeScreen), pasando true como resultado. Este resultado (true) se usa en HomeScreen para saber que se realizó un cambio y se deben recargar los gastos.
20. **@override void dispose() { ... }**
    - **¿Qué hace?** Este método del ciclo de vida se llama cuando el objeto State se elimina permanentemente (cuando la pantalla se cierra). Es crucial para liberar recursos, como los TextEditingController, para evitar fugas de memoria.
    - **_descripcionController.dispose();**: Libera el controlador de descripción.
    - **_montoController.dispose();**: Libera el controlador de monto.
    - **super.dispose();**: Siempre debes llamar a super.dispose() al final de tu implementación de dispose.
21. **@override Widget build(BuildContext context) { ... }**
    - **¿Qué hace?** Este método describe la interfaz de usuario de la pantalla basándose en el estado actual.
    - **final esEdicion = widget.gasto != null;**: Una variable booleana para saber si la pantalla está en modo edición.
    - **return Scaffold(...)**: Proporciona la estructura básica de la pantalla.
    - **appBar: AppBar(title: Text(esEdicion ? 'Editar Gasto' : 'Agregar Gasto'))**: Define la barra de aplicación con un título que cambia según si se está agregando o editando.
    - **body: Padding(...)**: El cuerpo principal de la pantalla, con padding alrededor.
    - **Form(key: _formKey, child: ListView(...))**: Un widget Form que agrupa los campos del formulario y permite la validación. Se le asigna la _formKey para poder acceder a su estado. ListView se usa para permitir el desplazamiento si el contenido supera el tamaño de la pantalla.
    - **TextFormField(...)**: Un campo de texto para la descripción.
        - controller: _descripcionController: Vincula el campo de texto al

controlador.
        - decoration: const InputDecoration(labelText: 'Descripción'): Define la etiqueta del campo.
        - validator: (value) => ...: Define la lógica de validación. Retorna un mensaje de error si el valor es nulo o vacío, de lo contrario retorna null (indicando que es válido).
    - **SizedBox(height: 16)**: Agrega espacio vertical entre los widgets.
    - **DropdownButtonFormField<String>(...)**: Un campo de formulario con un menú desplegable para seleccionar la categoría.
        - value: _categoriaSeleccionada: El valor seleccionado actualmente.
        - items: _categorias.map(...).toList(): Crea los elementos del menú desplegable a partir de la lista _categorias.
        - onChanged: (value) => setState(() => _categoriaSeleccionada = value): Actualiza la categoría seleccionada y reconstruye el widget cuando se selecciona un nuevo valor.
        - decoration: const InputDecoration(labelText: 'Categoría'): Define la etiqueta del campo.
        - validator: (value) => ...: Valida que se haya seleccionado una categoría.
    - **TextFormField(...)**: Un campo de texto para el monto.
        - keyboardType: TextInputType.number: Configura el teclado para mostrar números.
        - validator: (value) { ... }: Valida que el valor no esté vacío y que sea un número válido.
    - **Row(...)**: Organiza los widgets horizontalmente (el texto de la fecha y el botón).
    - **Expanded(...)**: Hace que el widget Text ocupe todo el espacio horizontal disponible en la Row que no sea ocupado por el botón.
    - **Text(...)**: Muestra la fecha seleccionada o un mensaje si no se ha seleccionado ninguna.
    - **TextButton(onPressed: _seleccionarFecha, child: const Text('Elegir Fecha'))**: Un botón para abrir el selector de fecha.
    - **ElevatedButton(onPressed: _guardarGasto, child: Text(...))**: El botón principal para guardar o actualizar el gasto. Llama a la función _guardarGasto cuando se presiona. El texto del botón cambia según si se está editando o agregando.

En resumen, la pantalla AddExpenseScreen es un StatefulWidget que permite al usuario ingresar o editar los detalles de un gasto (descripción, categoría, monto,

fecha) utilizando un formulario. Utiliza controladores para los campos de texto, un selector de fecha y un menú desplegable para la categoría. La lógica para guardar o actualizar el gasto interactúa con tu clase de ayuda de base de datos (DBHelper o DatabaseHelper) y luego regresa a la pantalla anterior.