Manual Técnico: GuardaTusVuelos

Índice

1. Introducción
2. Análisis del problema
3. Diseño de la solución
4. Documentación de la solución
5. Enlaces de interés

1. Introducción

La aplicación de Búsqueda de Vuelos es una solución móvil diseñada para simplificar el proceso de búsqueda y encontrar los mejores vuelos.

Ofrece una interfaz intuitiva y funcionalidades que permiten a los usuarios encontrar las mejores ofertas de vuelos de manera eficiente.

La importancia de este software radica en su capacidad para ahorrar tiempo y dinero a los viajeros, proporcionando una experiencia de usuario fluida.

2. Análisis del problema

2.1 Problemática

- Dificultad para comparar precios de vuelos entre diferentes aerolíneas.
- Necesidad de una interfaz móvil intuitiva para la búsqueda de vuelos.
- Falta de herramientas para guardar y gestionar vuelos favoritos.
- Necesidad de personalización en la búsqueda de vuelos (idioma, moneda).

2.2 Clientes potenciales

- Viajeros frecuentes
- Turistas ocasionales
- Agencias de viajes pequeñas

2.3 Análisis DAFO

Fortalezas:

- Interfaz intuitiva y fácil de usar
- Funcionalidad de guardado de vuelos favoritos
- Personalización de moneda
- Respuesta de búsqueda automática de los mejores vuelos.

Debilidades:

- Dependencia de APIs de terceros para la información de vuelos
- Necesidad de actualizaciones frecuentes para mantener la compatibilidad.
- Peticiones y funcionalidades limitadas al usar el plan gratuito de la API.

Oportunidades:

- Aumento en el uso de aplicaciones móviles para planificación de viajes
- Si se usa otro plan de pago y otras api a la vez, esta aplicación puede escalar a muchas funcionalidades que otras app del mercado no tengan.

Amenazas:

- Competencia de aplicaciones similares.
- Cambios en las políticas de las APIs de vuelos utilizadas.

2.4 Monetización y beneficios

- Futuras funcionalidades premium.
- Publicidad de servicios relacionados con viajes.

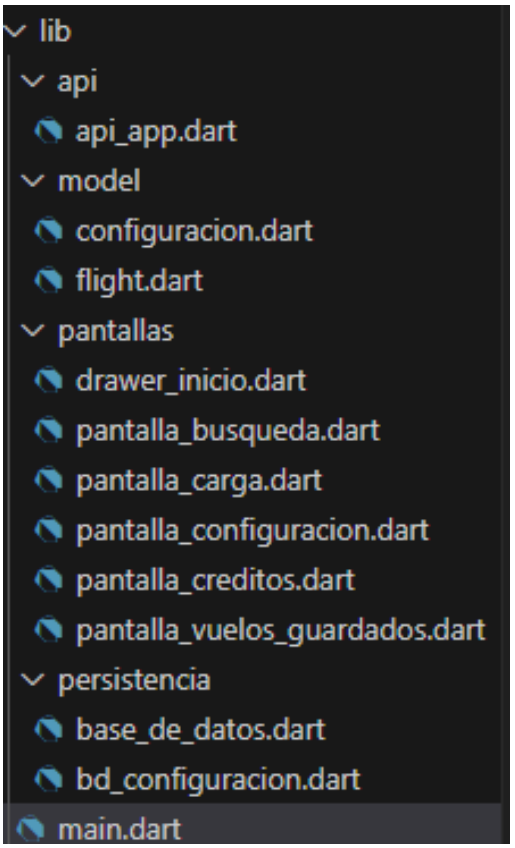
3. Diseño de la solución

3.1 Tecnologías elegidas

- Framework: Flutter
- Lenguaje de programación: Dart

- Base de datos local: SQLite
- API de vuelos: Google Flights API (a través de SerpAPI)

3.2 Arquitectura:



Main.dart:

```
Run | Debug | Profile
void main() {
  sqfliteFfiInit();
  databaseFactory = databaseFactoryFfi;
  BdConfiguracion.initDatabase();

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

  @override
  Widget build(BuildContext context) {

    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'GuardaTusVuelosHD',

      home: PantallaCarga(),
    ); // MaterialApp
  }
}
```

Es la clase que inicializa algunas cosas, el sqflifeffinit es para que funcione sqlite en windows, que hago la mayoría de pruebas ahí porque en casa tengo problemas con emuladores.
Databasefactory configura la base de datos para usar sqlite.
initDatabase, inicializa mi base de datos.
Despues de inicizaliar todo lanza la app, con la ventana pantallaCarga.

PantallaCarga.dart:

```dart
class PantallaCarga extends StatelessWidget {
  const PantallaCarga({super.key});

  @override
  Widget build(BuildContext context) {
    // Simula una carga de 8 segundos
    Future.delayed(const Duration(seconds: 8), () {
      Navigator.pushReplacement(
        // ignore: use_build_context_synchronously
        context,
        MaterialPageRoute(builder: (context) => const Drawerinicio()),
      );
    }); // Future.delayed

    return Scaffold(
      body: Container(
        decoration: BoxDecoration(
          gradient: LinearGradient(
            begin: Alignment.topLeft,
            end: Alignment.bottomRight,
            colors: [
              Colors.blue[50]!,
              Colors.white,
            ],
          ), // LinearGradient
        ), // BoxDecoration
        child: Center(
          child: Column(
```

Uso un future para posteriormente crear una acción, en este caso es un delayed que dura 8 segundos simulando una pantalla de carga para posteriormente inicializar la pantalla del drawer.

DrawerInicio:

```dart
class DrawerinicioState extends State<Drawerinicio> {
  int _selectedIndex = 0;

  final GlobalKey<PantallabusquedaState> _busquedaKey = GlobalKey<PantallabusquedaState>();

  late List<Widget> _screens;

  @override
  void initState() {
    super.initState();
    _initializeScreens();
  }
  // Inicializa las pantallas, ademas de mantener los vuelos guardados
  void _initializeScreens() {
    _screens = <Widget>[
      Pantallabusqueda(key: _busquedaKey),
      Pantallavuelosguardados(onFavoritesChanged: _updateFavorites),
      const PantallaConfiguracion(),
      const PantallaCreditos(),
    ]; // <Widget>[]
  }
```

En drawerInicio, creo las variables index que luego asignará a cada pantalla, lista para almacenar las pantallas.

InitializeScreens, inicializa las pantallas y mantiene los vuelos guardados.

```
// Actualiza los favoritos de la pantalla de búsqueda
  void _updateFavorites() {
    if (_busquedaKey.currentState != null) {
      _busquedaKey.currentState!.updateFavoriteStatus();
    }
  }

  // Método para cambiar de pantalla
  void _onItemTapped(int index) {
    setState(() {
      _selectedIndex = index;
    });
  }
```

UpdateFavorites, actualizo los favoritos en la pantalla busqueda (si sale el icono rojo o no de los vuelos)

onTimeTapped, método para cambiar de pantalla, usando el indice de estas.

Pantalla Configuración:

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

  @override
  PantallaConfiguracionState createState() => PantallaConfiguracionState();
}
BdConfiguracion bd = BdConfiguracion();

Future<void> _botonPref(String moneda, String idioma) async {

 await bd.savePreferences(moneda,idioma);

}

class PantallaConfiguracionState extends State<PantallaConfiguracion> {
  static String moneda = 'USD';
  static String idioma = 'Español';

  @override
  void initState() {
    super.initState();
```

Creo instancia de la base de datos conf, creo un método asíncrono que guarda las preferencias.

Creo las variables moneda y idioma.

El resto es el diseño, dropdown button para elegir la moneda y el botón de guardado, que asigna lo elegido las variables y las variables se pasa por parámetros a la base de datos inicializada dicha previamente.

PantallaVuelosGuardados:

```
class PantallavuelosguardadosState extends State<Pantallavuelosguardados> {
  late Future<List<Flight>> _favoritesFuture;

  @override
  void initState() {
    super.initState();
    _favoritesFuture =
        DatabaseHelper.instance.getFavorites();
  }
```

Creo una lista de vuelos futuros.
Inicializo la base de datos y llamo al método getFavorites, para que cargue los vuelos favoritos.

```
// Método para eliminar un vuelo de favoritos
void _eliminarFavorito(Flight vuelo) async {
  if (vuelo.id != null) {
    await DatabaseHelper.instance.deleteFavorite(vuelo.id!);
    setState(() {
      _favoritesFuture = DatabaseHelper.instance.getFavorites();
    });
    widget.onFavoritesChanged(); // Notificar que los favoritos han cambiado
  }
}
```

Este método elimina un vuelo de favoritos llamando a la función de la base de datos y cambia el estado.

Pantalla Busqueda:

```dart
class PantallabusquedaState extends State<Pantallabusqueda> {
  // variables
  static Future<List<Flight>> resultados = Future.value([]);
  Map<int, bool> favoriteStatus = {};
  Map<String, String> airportData = {};
  DateTime fechaSalida = DateTime.now();
  DateTime fechaVuelta = DateTime.now();
  int numeroPersonas = 1;
  bool soloIda = false;
  TextEditingController controladorOrigen = TextEditingController();
  TextEditingController controladorDestino = TextEditingController();
  final _formKey = GlobalKey<FormState>();

  // inicizalizo variable resultados
  @override
  void initState() {
    super.initState();
    updateFavoriteStatus();
  }
```

Creo todas estas variables necesarias, los controladores de origen destino, fechas, resultados en un future que obviamente es una lista de vuelos, numero de personas, formkey para validar campos del formulario..

Inicializo actualizando los vuelos favs.

```dart
bool validacion() {
  if (!(_formKey.currentState?.validate() ?? false)) {
    return true;
  }
  if(fechaVuelta.isBefore(fechaSalida)){
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('La fecha de vuelta no puede ser anterior a la fecha de salida')),
    );
    return true;
  }
  return false;

}

void _updateOrigenText(String text) {
  try {
    final codigo = ApiApp.obtenerCodigoA(text) ?? text.toUpperCase();
    controladorOrigen.value = TextEditingValue(
      text: codigo,
      selection: TextSelection.collapsed(offset: codigo.length),
    ); // TextEditingValue
  } catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Error al obtener código de aeropuerto: $e')),
    );
  }
}
```

Metodo validación, compruebo que los controladores de texto origen y destino estén bien escritos los parámetros y también compruebo que la fecha vuelta no sea previa a la de salida.

Método updatOrigenText y destinoText, estos métodos llaman al método obtenerCodigoA, que lo que hace es transformar en tiempo real cuando el usuario escribe una ciudad acortarla al código IATA y ponerlo en mayúsculas.

```dart
// seleccionar fecha
Future<void> _seleccionarFecha(
    BuildContext context, bool esFechaSalida) async {
  final DateTime? fechaSeleccionada = await showDatePicker(
    context: context,
    initialDate: esFechaSalida ? fechaSalida : DateTime.now(),
    firstDate: DateTime(2000),
    lastDate: DateTime(2101),
  );
  // metodo que actualiza la fecha
  if (fechaSeleccionada != null) {
    setState(() {
      if (esFechaSalida) {
        fechaSalida = fechaSeleccionada;
      } else {
        fechaVuelta = fechaSeleccionada;
      }
    });
  }
}
void updateFavoriteStatus() async {
// Obtiene la lista de vuelos almacenada en la variable 'resultados'.
final flights = await resultados;

for (int i = 0; i < flights.length; i++) {
  // Verifica si el vuelo actual es favorito y almacena el resultado en el
  favoriteStatus[i] = await DatabaseHelper.instance.isFavorite(flights[i])
}

// Si el widget aún está montado (es decir, no ha sido eliminado del árbol
// actualiza el estado del widget para reflejar los cambios en la interfaz
if (mounted) {
  setState(() {});
}
```

El método selector de fechas, hace que cree un showdatepicker para que el usuario pueda elegir por dias en un calendario a las variables de fecha, pongo rango minimo y maximo, pongo por defecto la fecha de hoy en la variable y hago un setState que actualiza la fecha con un if.

El metodo updateFavoriteStatus comprueba de la lista de vuelos cual es el favorito y lo actualiza llamando a la base de datos.

```dart
// metodo que busca vuelos con los parametros introducidos
void _buscarVuelos() async {
  if (validacion()) {
    return;
  }
  try {
    setState(() {
      resultados = ApiApp.fetchFlights(
          controladorOrigen.text.toUpperCase(),
          controladorDestino.text.toUpperCase(),
          fechaSalida,
          fechaVuelta,
          soloIda ? 2 : 1,
          numeroPersonas,
          PantallaConfiguracionState.moneda);
    });
  } catch (e) {
    ScaffoldMessenger.of(context)
        .showSnackBar(SnackBar(content: Text('Error al buscar vuelos: $e')));
  }
}
```

El metodo buscarVuelos, primero comprueba que la validacion de los campos es correcta, si lo es hace un setState y asigna a la variable resultados todos los parametros pasandoselo al metodo fetchFlights de la clase ApiApp.

```dart
FutureBuilder<List<Flight>>(
  future: resultados,
  builder: (context, snapshot) {
    if (snapshot.connectionState == ConnectionState.waiting) {
      return const Center(child: CircularProgressIndicator());
    } else if (snapshot.hasError) {
      return Center(child: Text('Error: ${snapshot.error}'));
    } else if (snapshot.hasData) {
      return ListView.builder(
        shrinkWrap: true,
        itemCount: snapshot.data!.length,
        itemBuilder: (context, index) {
          final flight = snapshot.data![index];
          bool isFavorite = favoriteStatus[index] ?? false;

          return Card(
            elevation: 5,
            margin: const EdgeInsets.symmetric(vertical: 8),
            shape: RoundedRectangleBorder(
              borderRadius: BorderRadius.circular(20),
            ), // RoundedRectangleBorder
            child: ListTile(
              contentPadding: const EdgeInsets.all(16),
              leading: Icon(Icons.flight_takeoff,
                  color: Colors.blue, size: 40), // Icon
              title: Text(
                '${flight.aeropuertoOrigen} → ${flight.aeropuertoDestino}',
                style: TextStyle(
                    fontWeight: FontWeight.bold, fontSize: 16), // TextStyle
              ), // Text
```

FutureBuilder, uso un futureBuilder para enseñar los resultados, ya que no se sabe siquiera si va a haber resultado o no ya que la peticion

puede ser o no correcta, y creo un card para hacer un diseño atractivo a los vuelos.

Flight:

```dart
class Flight {
  final int? id;
  final String? aeropuertoOrigen;
  final String? aeropuertoDestino;
  final String? horaSalida;
  final String? horaLlegada;
  final int? precio;
  final String? aerolinea;
  final String? moneda;
  final String? claseVuelo;
  final int? maxDuracion;

  const Flight({
    this.id,
    this.aeropuertoOrigen,
    this.aeropuertoDestino,
    this.horaSalida,
    this.horaLlegada,
    this.precio,
    this.aerolinea,
    this.moneda,
    this.claseVuelo,
    this.maxDuracion
  });

  // Convertir un Flight a un Map para la base de datos
  Map<String, dynamic> toMap() {
    return {
      'id': id,
      'aeropuertoOrigen': aeropuertoOrigen,
      'aeropuertoDestino': aeropuertoDestino,
      'horaSalida': horaSalida,
      'horaLlegada': horaLlegada,
      'precio': precio,
      'aerolinea': aerolinea,
      'moneda' : moneda,
      'claseVuelo' : claseVuelo,
      'maxDuracion' : maxDuracion
    };
  }
```

Atributos del vuelo, los convierto en un map para que los lea la base de datos.

```dart
// Convertir un JSON a un Flight
factory Flight.fromJson(Map<String, dynamic> json, String moneda) {
  return Flight(
    aeropuertoOrigen: json['flights'][0]['departure_airport']['name'],
    aeropuertoDestino: json['flights'][0]['arrival_airport']['name'],
    horaSalida: json['flights'][0]['departure_airport']['time'],
    horaLlegada: json['flights'][0]['arrival_airport']['time'],
    precio: json['price'],
    aerolinea: json['flights'][0]['airline'],
    moneda:moneda,
    claseVuelo: json['flights'][0]['travel_class'],
    maxDuracion: json['flights'][0]['duration']
  );
}

// Convertir un Map de la base de datos a un Flight
factory Flight.fromMap(Map<String, dynamic> map) {
  return Flight(
    id: map['id'],
    aeropuertoOrigen: map['aeropuertoOrigen'],
    aeropuertoDestino: map['aeropuertoDestino'],
    horaSalida: map['horaSalida'],
    horaLlegada: map['horaLlegada'],
    precio: map['precio'],
    aerolinea: map['aerolinea'],
    moneda: map['moneda'],
    claseVuelo: map['claseVuelo'],
    maxDuracion: map['maxDuracion']

  );
}
```

Flight.fromJson, esto es lo que convierte el JSon al flight, es decir cojo mis atributos y veo en la pagina de la api como se llama realmente los parametros que quiero buscar y los igualo por ejemplo, aerolinea, esta dentro del json del array flights, llamado airline.

from map, es lo mismo pero convierto los datos ya mapeados a un objeto que flight para que lo lea la base de datos.

```dart
class Configuracion {
  static  String? moneda;
  static String? idioma;

}
```

Simplemente las variables estáticas de configuración.

ApiApp:

```dart
class ApiApp {

// Metodo para obtener los vuelos
static Future<List<Flight>> fetchFlights(
    String ciudadOrigen, String ciudadDestino, DateTime fechaSalida, DateTime fechaVuelta, int tipo, int nPasajeros, String moneda) async {
 String url ;
 // Si el tipo es 1, es un vuelo de ida y vuelta
  if(tipo==1){
    url = "https://serpapi.com/search.json?engine=google_flights&departure_id=$ciudadOrigen&arrival_id=$ciudadDestino&outbound_date=${fechaSalida.toString().substring(0,10)}&return_dat
  }else{
    url = "https://serpapi.com/search.json?engine=google_flights&departure_id=$ciudadOrigen&arrival_id=$ciudadDestino&outbound_date=${fechaSalida.toString().substring(0,10)}&type=$tipo
  }
  //print(url);
  // Realiza la petición HTTP
    final response = await http.get(Uri.parse(url));
    //print(response.body);
    //print(response.statusCode);
    // Comprueba si la petición ha sido correcta
    if (response.statusCode == 200) {
      var jsonBody = json.decode(response.body);
      // Parsear el JSON
      List data;
      try {
        data = jsonBody['best_flights']as List;
      if(data.isEmpty){
        data = jsonBody['other_flights']as List;
      }
      } catch (e) {
        var variable = json.decode(response.body)['error'];
      throw Exception('Error al obtener los vuelos. $variable');
      }

      String moneda = jsonBody['search_parameters']['currency'];
      // Mapear los vuelos
      List<Flight> vuelos =
          data.map((json) => Flight.fromJson(json, moneda)).toList();

      // Filtrar vuelos según las fechas
```

```
return_date=${fechaVuelta.toString().substring(0,10)}&type=$tipo&currency=$moneda&hl=en&adults=$nPasajeros&api_key=22444223f4ca84ed0407a56f4ff1c3f114f084d21a77640369c4b6020000da38";

type=$tipo&hl=en&currency=$moneda&adults=$nPasajeros&api_key=22444223f4ca84ed0407a56f4ff1c3f114f084d21a77640369c4b6020000da38";
```

El metodo que obtiene los vuelos, tipo future, que es una lista de vuelos, con todos los parametros de la clase flight.
El enlace que es lo mas importante, coje la peticion de la Api googleFlights y después poniendo las variables igualandolo a los parametros de la api reales a como se llamen para que las lea, por ejemplo ciudadOrigen es departure_id., luego al final pongo mi apikey.

el metodo sigue realizando una peticion HTTP, si es correcta cojo con una variable data y meto el json dentro, para luego buscar por bestFlights o other flights que son los arrays “raices” donde estan todos los datos de la api y busco a partir de ahi aerolinea, destino fecha ..

```dart
// Método para obtener los códigos de aeropuertos
static String? obtenerCodigoA(String ciudad) {
  // Normalizar la entrada: quitar espacios en blanco y convertir a formato título
  ciudad = ciudad.trim().split(' ').map((word) => word.capitalize()).join(' ');

  Map<String, String> aeropuertos = {
    "Madrid": "MAD",
    "Barcelona": "BCN",
    "Valencia": "VLC",
    "Sevilla": "SVQ",
    "Bilbao": "BIO",
    "Málaga": "AGP",
    "Palma De Mallorca": "PMI",
    "Tenerife": "TFN",
    "Santiago De Compostela": "SCQ",
    "Ibiza": "IBZ",
```

Este método ya usado previamente, transforma la ciudad a codigo iata de las ciudades que he escrito.

DatabaseHelper:

```dart
// Definir las clases para la base de datos
class DatabaseHelper {
  static final DatabaseHelper instance = DatabaseHelper._init();
  static Database? _database;

  DatabaseHelper._init();

  Future<Database> get database async {
    if (_database != null) return _database!;
    _database = await _initDB('./lib/persistencia/guardaVuelosDB.db');

      return _database!;
  }

  Future<Database> _initDB(String path) async {

    return await openDatabase(path, version: 1, onCreate: _createDB);
  }

 void _createDB(Database db, int version) async {
  await db.execute('''
    CREATE TABLE favoritos(
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      aeropuertoOrigen TEXT,
      aeropuertoDestino TEXT,
      horaSalida TEXT,
      horaLlegada TEXT,
      precio INTEGER,
      aerolinea TEXT,
      moneda TEXT,
      claseVuelo TEXT,
      maxDuracion INTEGER
    )
  ''');
}
```

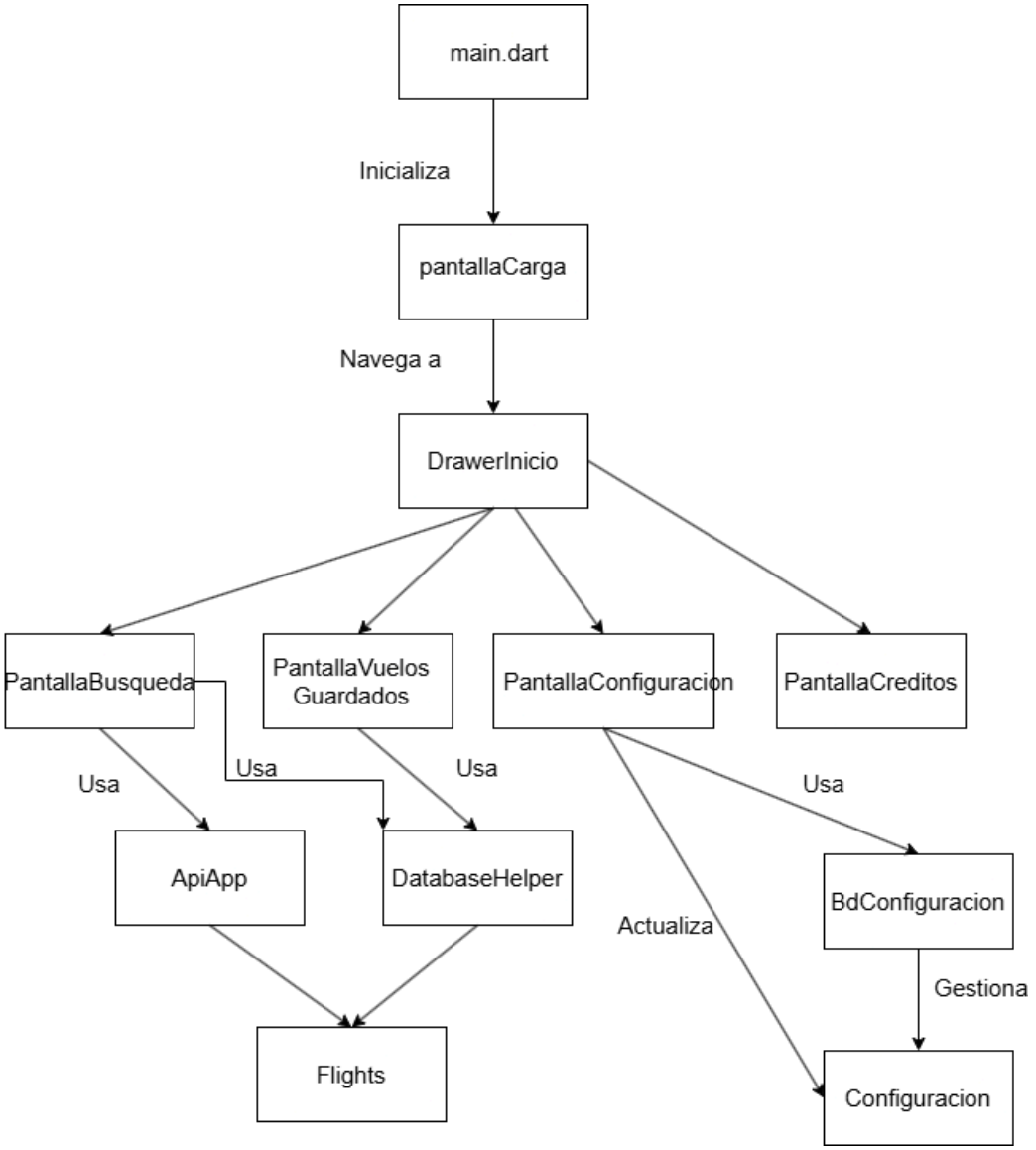
Inicializo estancia de mi base de datos y pongo la ruta donde quiero guardarlo, metodo init hago un await de openDatabase.

El método createDb crea la tabla favoritos con sus campos.

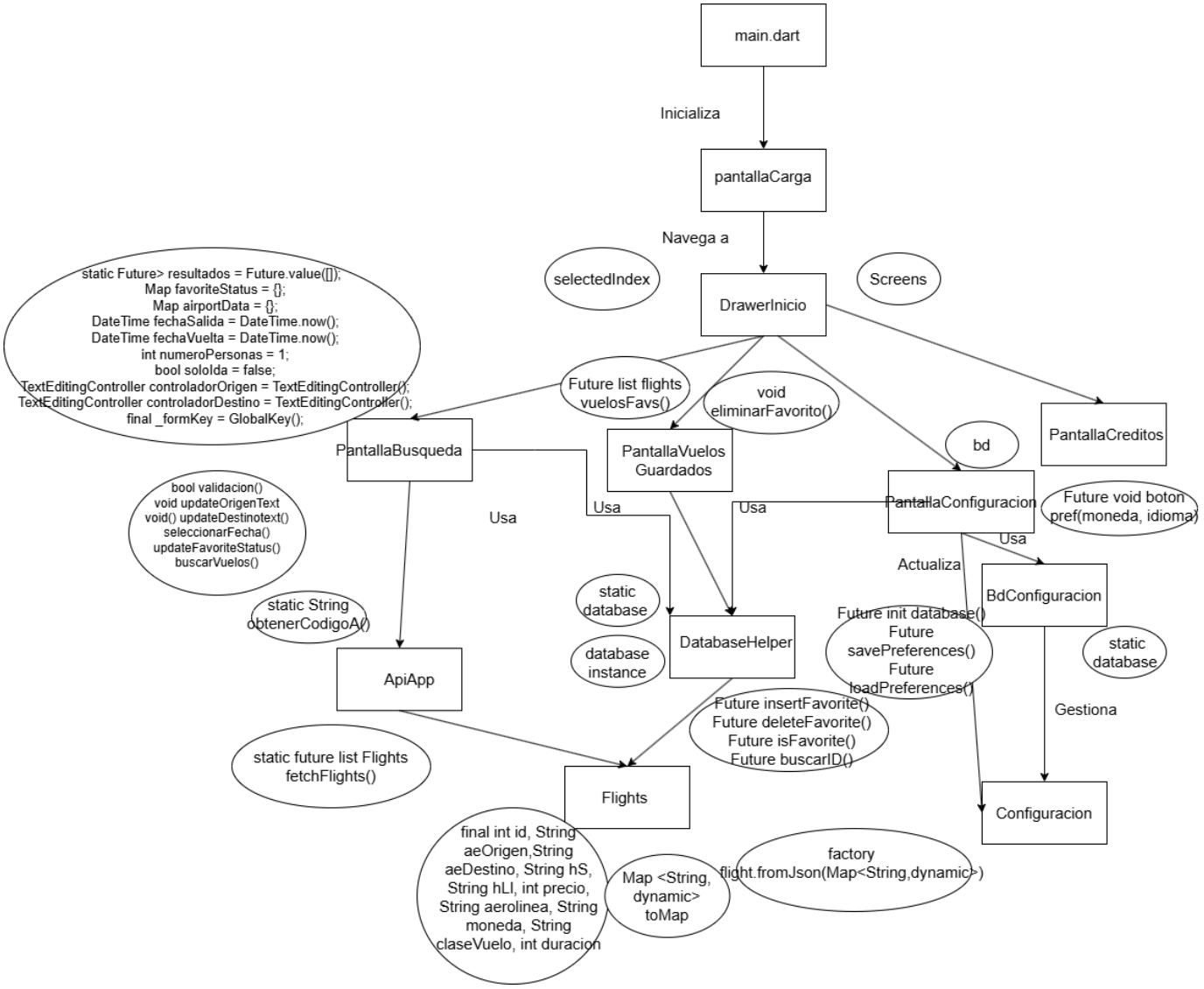
```dart
// Insertar un vuelo en la base de datos
Future<int> insertFavorite(Map<String, dynamic> flightData) async {
  final db = await instance.database;
  return await db.insert('favoritos', flightData); // Acepta un Map<String, dynamic>
}

// Eliminar un vuelo por su ID
Future<int> deleteFavorite(int id) async {
  final db = await instance.database;
  return await db.delete(
    'favoritos',
    where: 'id = ?',
    whereArgs: [id],
  );
}
 Future<bool> isFavorite(Flight flight) async {
  final db = await instance.database;
  final result = await db.query(
    'favoritos',
    where: 'aeropuertoOrigen = ? AND aeropuertoDestino = ? AND horaSalida = ? AND horaLlegada = ?',
    whereArgs: [flight.aeropuertoOrigen, flight.aeropuertoDestino, flight.horaSalida, flight.horaLlegada],
  );
  return result.isNotEmpty;
}

Future <Flight> buscarID(int id) async {
  final db = await instance.database;
  return await db.query('favoritos', where: 'id = ?',
  whereArgs: [id]).then((value) =>  Flight.fromMap(value.first));
}
// Obtener todos los vuelos favoritos
Future<List<Flight>> getFavorites() async {
  final db = await instance.database;
  final List<Map<String, dynamic>> maps = await db.query('favoritos');
  return List.generate(maps.length, (i) {
    return Flight.fromMap(maps[i]);
  }); // List.generate
}
```

Estos métodos son los que inserta los vuelos a la tabla, elimina, comprueba que sea favorito y buscan todos mediante el id.

3.3 Diagrama de clases



3.4 Diagrama E/R



3.5 Consideraciones técnicas

- Uso de SQLite para almacenamiento local de favoritos y configuraciones
- Manejo de errores y excepciones en llamadas a API
- Optimización de rendimiento en la carga de resultados de búsqueda

4. Documentación de la solución

El código fuente de la aplicación se encuentra en GitHub y es publico.
https://github.com/AlbertoEP123/flutterEjer

5. Enlace de interés

https://serpapi.com/google-flights-api

https://es.stackoverflow.com/questions/418737/sqlite-en-flutter

https://kikesan.medium.com/obtener-datos-de-un-api-rest-con-flutter-507a01382577

```
https://serpapi.com/search.json?engine=google_flights&departure_id=CDG,ORY&arrival_id=LAX&outbound_date=2025-02-08&return_date=2025-02-14&currency=USD&hl=en
```