Marzo, 22 2025.

**MANUAL TECNICO PADILLAROUTE.**

v 1.0.0

TABLA DE CONTENIDO

[**INTRODUCCION** 4](#_Toc193796873)

[**CAPITULO 1: ARQUITECTURA.** 5](#_Toc193796874)

[**A) PATRON ARQUITECTONICO.** 5](#_Toc193796875)

[**CAPITULO 2: BASE DE DATOS.** 7](#_Toc193796876)

[**B) REALTIME DATABASE.** 7](#_Toc193796877)

[**C) OBJECTBOX STORE.** 7](#_Toc193796878)

[**CAPITULO 3: BACKEND.** 7](#_Toc193796879)

[**A) APIS.** 7](#_Toc193796880)

[**i.** **MAPS.** 7](#_Toc193796881)

[**ii.** **GEOLOCATOR.** 7](#_Toc193796882)

[**B) MODELOS.** 7](#_Toc193796883)

[**i.** **USUARIOS.** 7](#_Toc193796884)

[**ii.** **VEHICULOS.** 7](#_Toc193796885)

[**iii.** **PARADAS.** 7](#_Toc193796886)

[**iv.** **RUTAS.** 7](#_Toc193796887)

[**v.** **VIAJES.** 7](#_Toc193796888)

[**vi.** **INCIDENTES.** 7](#_Toc193796889)

[**vii.** **LOGS.** 7](#_Toc193796890)

[**C) SERVICIOS.** 7](#_Toc193796891)

[**i.** **CONECTOR REALTIME DATABASE.** 7](#_Toc193796892)

[**ii.** **CONECTOR OBJECTBOX STORE.** 7](#_Toc193796893)

[**iii.** **AUTENTICACIÓN.** 7](#_Toc193796894)

[**iv.** **NOTFICACIÓNES.** 7](#_Toc193796895)

[**v.** **GESTION DE USUARIOS.** 7](#_Toc193796896)

[**vi.** **GESTION DE VEHICULOS.** 7](#_Toc193796897)

[**vii.** **GESTION DE PARADAS.** 7](#_Toc193796898)

[**viii.** **GESTION DE RUTAS.** 7](#_Toc193796899)

[**ix.** **GESTION DE VIAJES.** 8](#_Toc193796900)

[**x.** **GESTION DE INCIDENTES.** 8](#_Toc193796901)

[**xi.** **LOGS DEL SISTEMA.** 8](#_Toc193796902)

[**CAPITULO 4: INTERFAZ (FRONTEND).** 8](#_Toc193796903)

[**A) PANTALLAS.** 8](#_Toc193796904)

[**B) WIDGETS PERSONALIZADOS.** 8](#_Toc193796905)

[**CAPITULO 5: SEGURIDAD.** 8](#_Toc193796906)

[**A) AUTENTICACIÓN DE IDENTIDAD DE USUARIOS.** 8](#_Toc193796907)

[**B) BUENAS PRACTICAS.** 8](#_Toc193796908)

[**i.** **ENCRIPTACION DE CONTRACSEÑAS.** 8](#_Toc193796909)

[**C) PERMISOS**. 8](#_Toc193796910)

# **INTRODUCCION**

Este manual técnico describe el diseño, arquitectura y funcionamiento de [nombre de la aplicación], una aplicación desarrollada en **Flutter** con almacenamiento local en **ObjectBox** y sincronización con Firebase Realtime **Database**.

El objetivo de esta aplicación es [breve descripción del propósito de la app]. Para lograrlo, se ha diseñado con una arquitectura modular y escalable, permitiendo una fácil expansión y mantenimiento.

A lo largo de este documento, se detallarán los componentes clave**, la estructura de la base de datos, las interacciones con Firebase y las mejores prácticas utilizadas en su desarrollo**. Este manual está dirigido a desarrolladores, ingenieros de software y cualquier persona interesada en comprender el funcionamiento interno de la aplicación.

***Nota:******se presupone que quien esté leyendo esto tiene, de antemano, conocimientos en el stack mencionado anteriormente y usado para este proyecto. Por lo cual, no nos centraremos en explicar cosas como la estructura de un proyecto de Flutter, qué es Firebase, patrones de diseño, etc.***

# **ARQUITECTURA.**

## **PATRON ARQUITECTONICO.**

Este proyecto utiliza una versión personalizada del patrón de diseño arquitectónico **Modelo-Vista-Controlador**; dentro de lo poco que se personalizó se encuentran los nombres de los componentes: **Modelo-Pantalla-Servicio**.

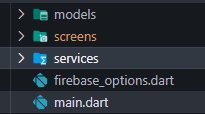


Imagen 1.1 Patrón arquitectónico de la aplicación.

* **Models** contiene los modelos de datos necesarios para trabajar con las bases de datos local y remota (Realtime Database y ObjectBox Store respectivamente).
* **Screens**, justo como lo están pensado, contiene las pantallas y widgets utilizados para construir la interfaz gráfica de la aplicación.
* **Services** guarda toda la lógica de negocios; conectores a bases de datos, servicios para realizar operaciones CRUD, servicios de autenticación de identidad, generadores de archivos, entre otros. Mas adelante se explorará a profundidad.

## **FLUJO DE DATOS.**

Revisemos cómo funciona el sistema. Se pueden distinguir tres actores distintos: un encargado de la administración de recursos como usuarios, vehículos, reportes, rutas, etc; otro cuya tarea es llevar el registro de los viajes en tiempo real; finalmente la base de datos remota, que es el principal centro de almacenamiento de datos.

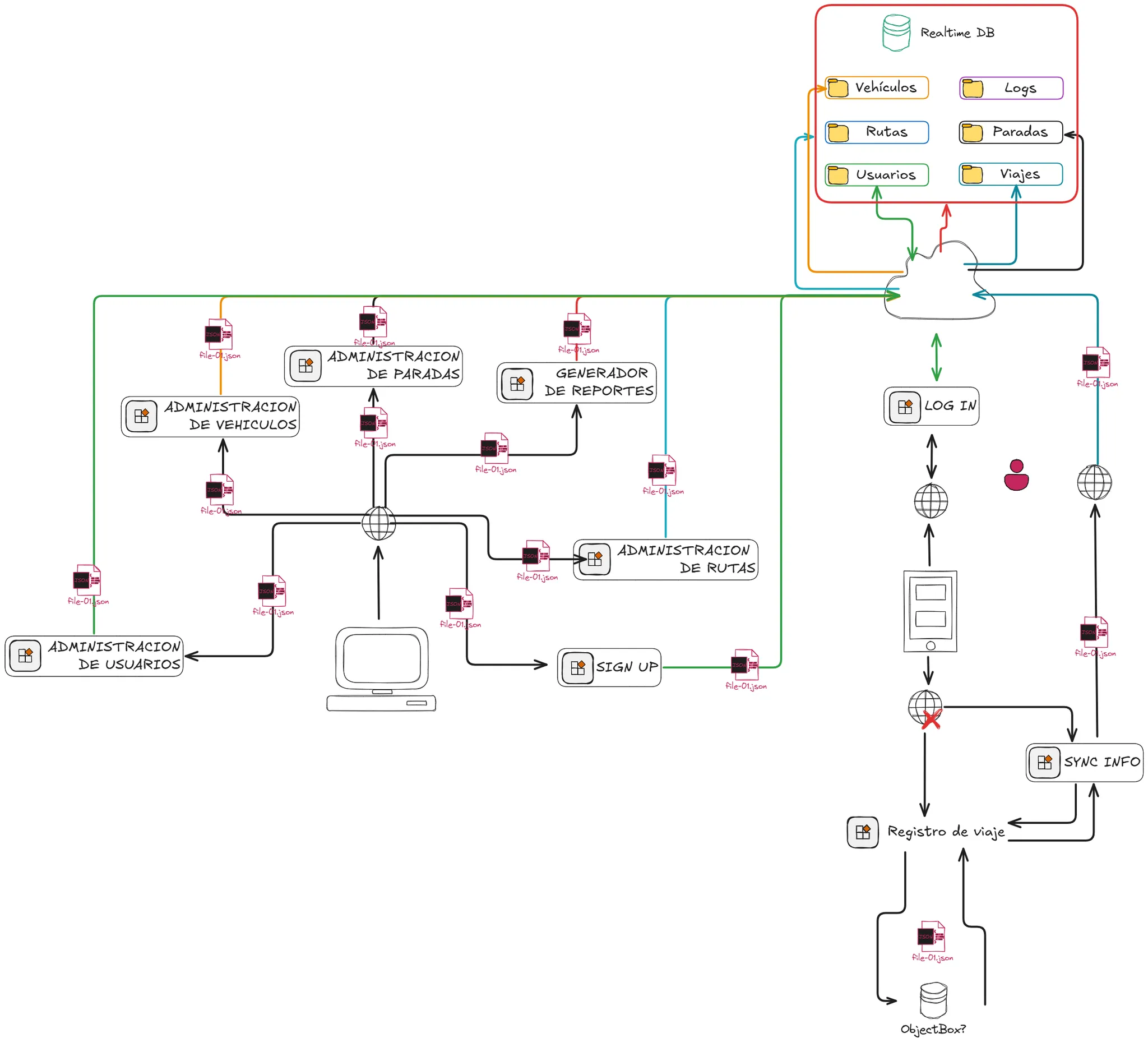


Imagen 1.2 Flujo de datos del sistema.

# **BASE DE DATOS.**

El proyecto utiliza dos bases de datos cuyos fines son distintos; por un lado, Realtime Database, cómo principal medio de almacenamiento; por el otro lado, Objectbox Storage, para almacenamiento local temporal.

## **REALTIME DATABASE.**

Esta es el principal medio de almacenamiento de la aplicación. La base de datos contiene 7 colecciones:

1. Usuarios.
2. Vehículos.
3. Paradas.
4. Rutas.
5. Registro de viajes.
6. Registro de incidentes.
7. Logs.

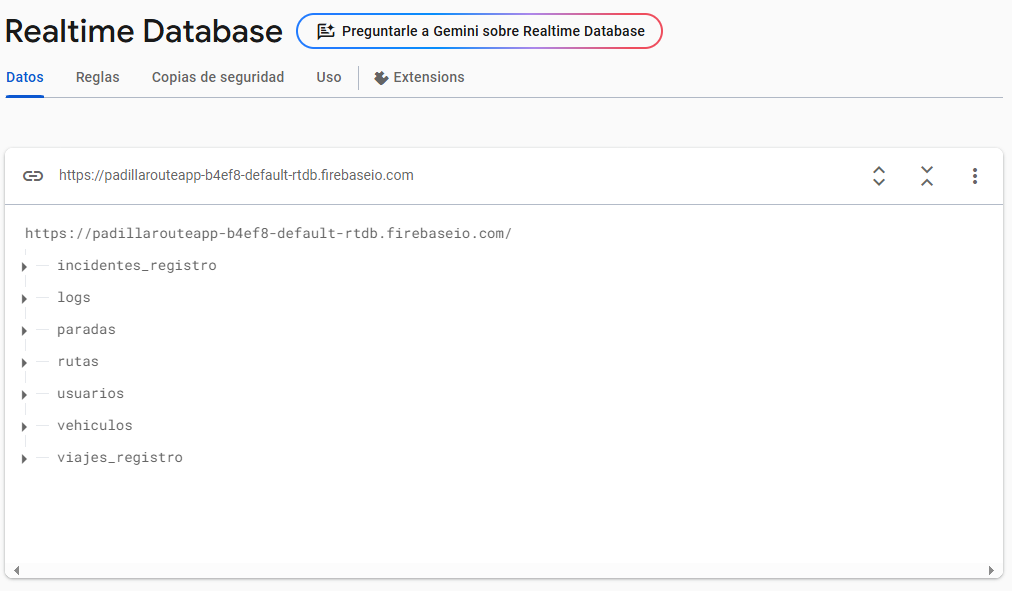


Imagen 2.1 Base de datos de la aplicación.

A continuación, se detallan los modelos de datos de cada colección:

### **MODELOS**.

### **USUARIOS.**

Representa a los empleados de la empresa y los clasifica en tres tipos: choferes, administrativos y gerentes mediante el uso de un enum llamado Rol.

enum Rol {

  @JsonValue('chofer')

  chofer,

  @JsonValue('administrativo')

  administrativo,

  @JsonValue('gerente')

  gerente

}

También se definen los campos necesarios; Id único del usuario, nombre, apellidos, numero de teléfono, correo electrónico, contraseña, rol, estatus, id del vehículo usado, token personal para mensajeria.

final int idUsuario;

final String nombre;

final String apellidos;

final int? Teléfono;

final String correo;

final String contrasena;

final Rol rol;

final bool activo;

final int? idVehiculo;

final String? fcmToken;

### **VEHICULOS.**

Son las unidades que los choferes usan para realizar su labor: transporta personal a sus diferentes destinos. Se define el identificador único, matricula, marca, modelo, capacidad de pasajeros, número de serie, estatus.

  final int idVehiculo;

  final String placa;

  final String marca;

  final String modelo;

  final int capacidad;

  final int numeroSerie;

  final Estatus estatus;

El estatus se refiere a si la unidad esta en uso activo o en mantenimiento.

enum Estatus {

  @JsonValue('activo')

  activo,

  @JsonValue('inactivo')

  inactivo,

  @JsonValue('mantenimiento')

  mantenimiento,

}

### **PARADAS.**

Las paradas son los puntos en los que los pasajeros abordan los vehículos de la empresa. Se define el identificador único, nombre de la parada, hora en la que se debe llegar, hora en la que se debe partir y las coordenadas en las que se localiza.

  final int idParada;

  final String nombre;

  final String horaLlegada;

  final String horaSalida;

  final String coordenadas;

### **RUTAS.**

Son colecciones de paradas que los choferes deben recorrer para llevar a sus pasajeros a sus destinos. Define el identificador único, identificador único del chofer encargado, identificador único del vehículo usado, nombre de la ruta, parada de origen, parada de destino, colección de todas las paradas que la conforman.

  final int idRuta;

  final int idChofer;

  final int idVehiculo;

  final String nombre;

  final String origen;

  final String destino;

  final List<String> paradas;

### **VIAJES.**

Son los registros de cada viaje realizado a manera de bitácora; cada registro tiene definido un identificador único, identificador de la ruta recorrida, identificador del chofer responsable, identificador del vehículo usado, lista de paradas con el numero de pasajeros recogidos, hora de inicio y termino del viaje, duración total del viaje, cantidad de pasajeros transportados, distancia recorrida, velocidad promedio del vehículo.

  final int idRegistro;

  final int idRuta;

  final int idVehiculo;

  final int idUsuario;

  final Map<String, dynamic> paradasRegistro;

  final String horaInicio;

  final String horaFinal;

  final int tiempoTotal;

  final int totalPasajeros;

  final int distanciaRecorrida;

  final int velocidadPromedio;

### **INCIDENTES.**

Los incidentes son situaciones excepcionales, previstas o no, que obstaculizan el cumplimiento de la labor de los choferes. Pueden ser retrasos o desvíos debido a tráfico, averías, etc. No son estrictamente accidentes con víctimas mortales.

Define un identificador único, identificador del usuario autor, identificador del vehículo usado, descripción del incidente y fecha en que se registró.

  final int idRegistro;

  final int idUsuario;

  final int idVehiculo;

  final String descripcion;

  final String fecha;

## **OBJECTBOX STORAGE.**

Objectbox es la base de datos local utilizada únicamente para almacenar los registros de los viajes y sincronizarlos con Realtime Database cuando es necesario o posible hacerlo.

Define el identificador único del registro, de la ruta seguida, del chofer y el vehículo; la lista de paradas y cantidad de pasajeros que abordaron en cada una; hora de inicio y termino, duración total, total de pasajeros, distancia recorrida y velocidad promedio del vehículo.

  int id = 0;

  int idRuta;

  int idVehiculo;

  int idChofer;

  String paradasRegistro; // Store paradasRegistro as JSON

  String horaInicio;

  String horaFinal;

  int tiempoTotal;

  int totalPasajeros;

  double distanciaRecorrida;

  double velocidadPromedio;

# **BACKEND.**

El backend de la aplicación PadillaRoute está compuesto por un conjunto de servicios y APIs que permiten gestionar la lógica de negocio, la persistencia de datos, la autenticación, las notificaciones, el seguimiento en tiempo real y el control de usuarios, vehículos, rutas, viajes e incidentes. Este capítulo describe las tecnologías utilizadas y los componentes desarrollados para garantizar un funcionamiento robusto, modular y escalable.

A continuación se describirá a detalle el funcionamiento de los servicios y APIs utilizados por la aplicación PadillaRoute:

## **APIS.**

Las APIs utilizadas en esta aplicación permiten interactuar con funcionalidades clave del dispositivo y servicios externos como Google Maps y los sensores de geolocalización. A continuación, se detallan las principales:

### **MAPS.**

La API de Google Maps se integra a través del paquete google\_maps\_flutter, permitiendo mostrar mapas interactivos en la aplicación, así como renderizar rutas, marcadores personalizados y ubicación del usuario en tiempo real. Esta funcionalidad es crucial para la visualización de paradas, rutas y seguimiento de recorridos en tiempo real.

**Principales funcionalidades implementadas:**

1. Visualización de mapa centrado en la ubicación actual del dispositivo.
2. Marcado de paradas sobre el mapa (iconos personalizados).
3. Trazado de rutas entre paradas.
4. Seguimiento en tiempo real de los vehículos durante un viaje.
5. Detección del inicio o fin de una ruta mediante geolocalización y actualización en Firebase.

Para utilizar este paquete, se agregó la dependencia correspondiente en el archivo pubspec.yaml:



Ilustración 3.1 Dependencia usada para el uso de Google Maps.

Además, se configuró la clave de API de Google Maps en el archivo AndroidManifest.xml y en el archivo AppDelegate.swift para iOS.

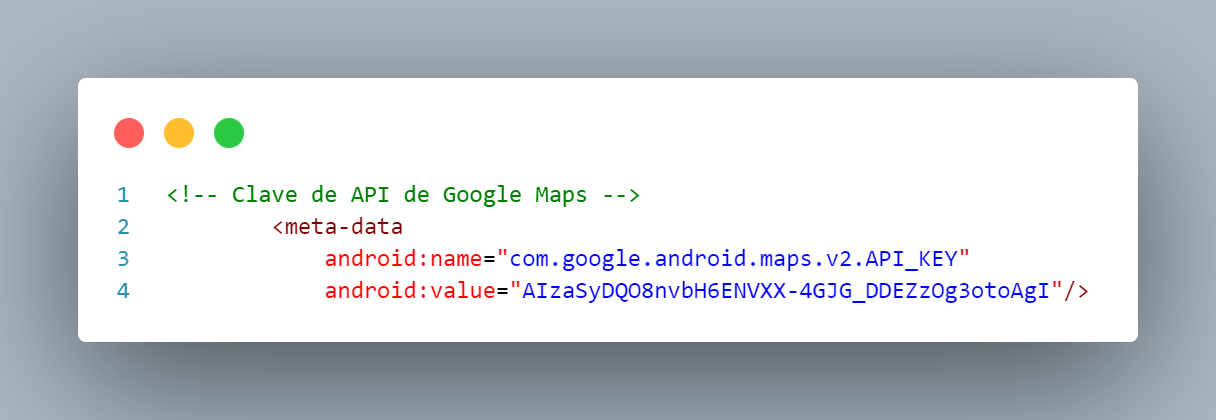


Ilustración 3.2 Configuración de API para el SO Android.

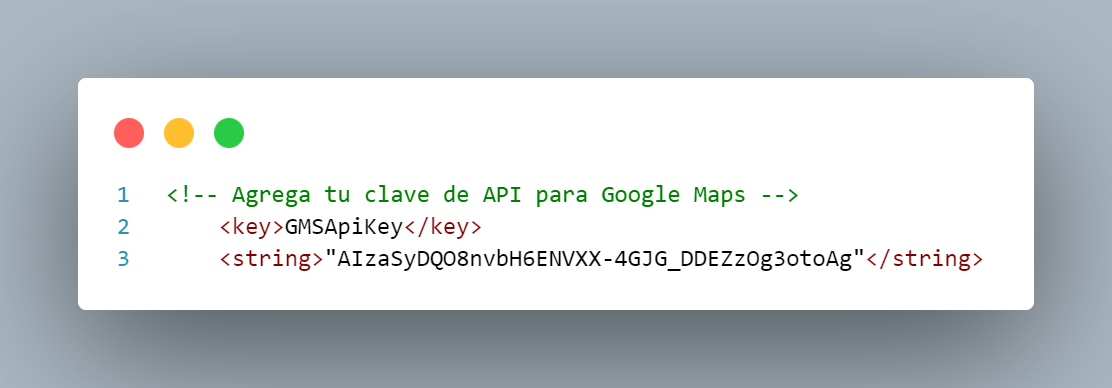


Ilustración 3.3 Configuración de API para el SO iOS.

### **GEOLOCATOR.**

El paquete geolocator proporciona acceso a los servicios de localización del dispositivo. Esta API es fundamental para obtener la posición actual del chofer, calcular distancias entre puntos, registrar la ruta recorrida y activar acciones derivadas de ello (por ejemplo, registrar una parada cuando el vehículo llega a un punto geográfico determinado).

**Principales funcionalidades implementadas:**

1. Obtención de coordenadas GPS del dispositivo.
2. Cálculo de distancia entre dos ubicaciones.
3. Detección de cambios en la ubicación mediante stream.
4. Validación de permisos de ubicación (en primer y segundo plano).
5. Actualización periódica de la ubicación en Firebase.

Dependencia usada:



Ilustración 3.4 Dependencia usada para el uso de Geolocator.

También se añadieron permisos específicos en los archivos de configuración para Android (AndroidManifest.xml) e iOS (Info.plist), asegurando que la aplicación pueda usar la ubicación aún en segundo plano si es necesario para el seguimiento de viajes.



Ilustración 3.5 Configuración de permisos de ubicación para el SO Android.

## **SERVICIOS.**

Esta sección agrupa los componentes encargados de manejar la lógica para interactuar directamente con los datos de la aplicación, ya sea a través de almacenamiento local o en la nube. Los servicios están organizados de forma modular para facilitar su reutilización, mantenimiento y escalabilidad. A continuación, se describen los servicios que componen la lógica de la aplicación PadillaRoute:

### **CONECTOR REALTIME DATABASE.**

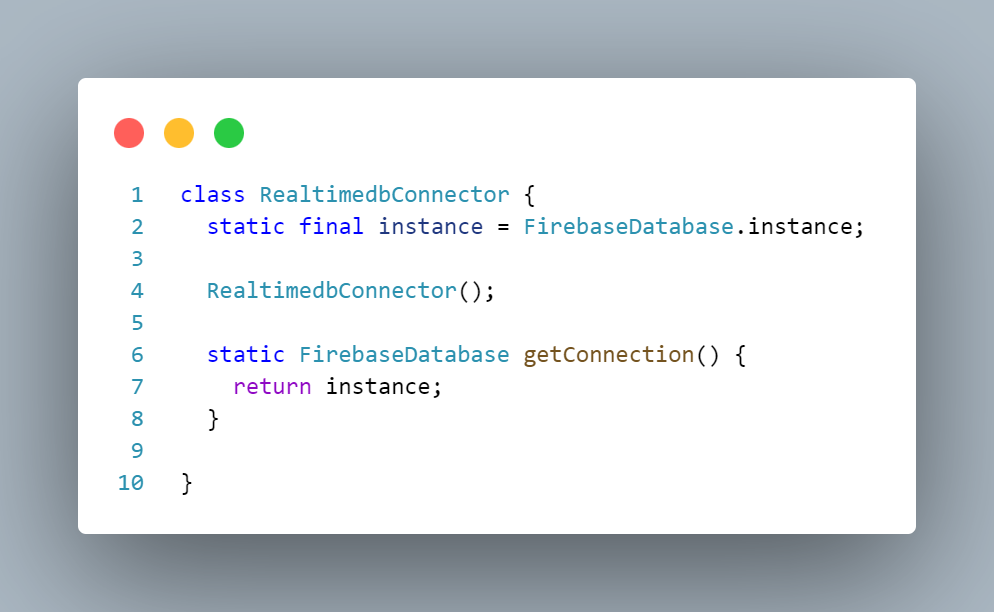
El conector RealtimedbConnector encapsula la instancia principal de Firebase Realtime Database utilizada a lo largo de la aplicación para leer y escribir datos remotos.

Este conector actúa como un singleton, asegurando que siempre se utilice la misma instancia de base de datos a través del método estático getConnection(). Esto facilita el desacoplamiento entre la lógica de negocio y la infraestructura de almacenamiento remoto.

**Características:**

1. Inicialización centralizada de FirebaseDatabase.
2. Uso de FirebaseDatabase.instance como punto único de conexión.
3. Facilita el mantenimiento y la reutilización del acceso a Realtime Database.

**Fragmento representativo:**



### **CONECTOR OBJECTBOX STORE.**

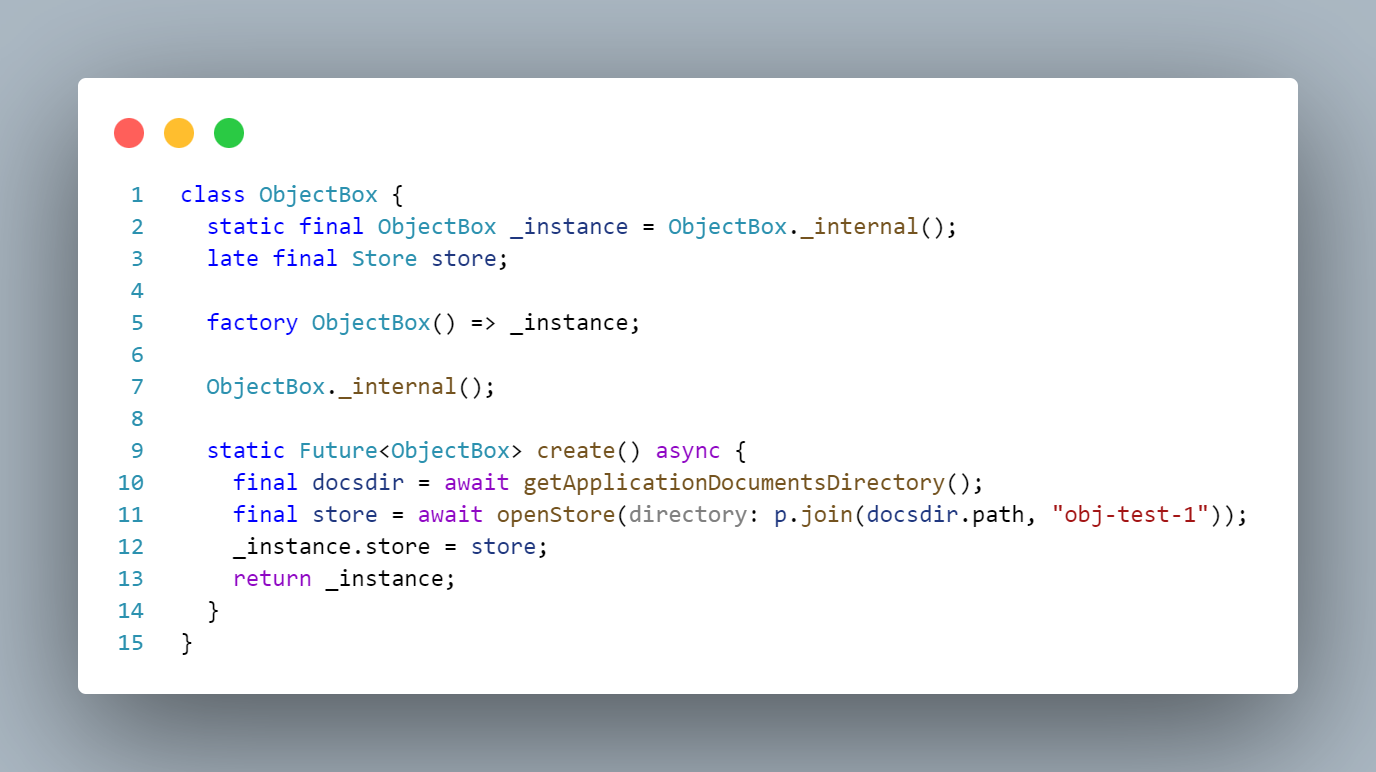
El conector ObjectBox es responsable de inicializar y proveer acceso a la base de datos local basada en ObjectBox. Implementado como un singleton, permite mantener una única instancia del Store que gestiona las operaciones locales, como el almacenamiento temporal de viajes o sincronización offline.

La función create() se encarga de configurar el directorio donde se almacenarán los archivos de la base de datos, utilizando el paquete path\_provider para obtener la ruta adecuada en el dispositivo.

**Características:**

1. Inicialización diferida (lazy initialization) del Store.
2. Gestión del ciclo de vida del Store de ObjectBox.
3. Persistencia local eficiente y sincronizable con Realtime Database.

**Fragmento representativo:**



### **AUTENTICACIÓN.**

El servicio de autenticación es gestionado a través de Firebase Authentication, utilizando el helper FirebaseAuthHelper. Este módulo encapsula todas las operaciones relacionadas con la creación, inicio y cierre de sesión de usuarios en la aplicación.

**Características principales del servicio de autenticación:**

1. Registro de nuevos usuarios mediante correo electrónico y contraseña.
2. Validación de credenciales para el inicio de sesión.
3. Cierre de sesión de usuarios activos.
4. Manejo de errores con mensajes personalizados según el tipo de fallo (por ejemplo: contraseña incorrecta, usuario no encontrado, cuenta deshabilitada, etc).

El helper utiliza internamente la instancia de FirebaseAuth como punto de entrada para autenticar con Firebase:



Este servicio es esencial para controlar el acceso a las funcionalidades sensibles de la aplicación, ya que garantiza que solo los usuarios registrados y autenticados puedan interactuar con los datos en Realtime Database y acceder a sus roles correspondientes (chofer, administrativo, gerente).

### **NOTFICACIÓNES.**

El sistema de notificaciones en la aplicación está compuesto por dos capas: Firebase Cloud Messaging (FCM) para el envío y recepción de mensajes, y Flutter Local Notifications para mostrar notificaciones visuales en el dispositivo del usuario.

**Componentes clave:**

* **fcm\_service.dart:** se encarga de construir y enviar mensajes push tanto a topics generales como a usuarios individuales mediante tokens FCM. Utiliza peticiones HTTP firmadas con un JWT y autenticadas vía OAuth2 para comunicarse directamente con la API REST de FCM.
* **usuarios\_helper.dart:** gestiona la lectura y actualización del token FCM de cada usuario, almacenado en Realtime Database.
* **main.dart:** inicializa FCM, solicita permisos y configura el comportamiento de los mensajes entrantes tanto en primer como segundo plano, mostrando notificaciones locales cuando se recibe un mensaje.

**Flujo general:**

1. El dispositivo cliente obtiene un token FCM único y lo guarda en la base de datos.
2. Cuando un administrador o evento requiere enviar una notificación, se construye el mensaje en fcm\_service.dart.
3. El servidor FCM entrega el mensaje al token correspondiente.
4. Si la app está en primer plano, la notificación se muestra usando FlutterLocalNotificationsPlugin.

Este sistema garantiza la entrega de mensajes críticos (como inicio de ruta, reportes de incidentes o asignación de viajes) a los usuarios adecuados, fortaleciendo la comunicación interna de la operación.

### **GESTION DE USUARIOS.**

La gestión de usuarios está centralizada en la clase UsuariosHelper, la cual interactúa directamente con la colección usuarios de la base de datos remota Firebase Realtime Database, utilizando el servicio RealtimeDbHelper como intermediario. Este módulo encapsula las operaciones básicas y avanzadas necesarias para administrar los datos relacionados con los empleados de la empresa.

**Funciones principales:**

1. **Creación de usuarios (setNew):** Registra un nuevo usuario en la base de datos remota, enviando un objeto Usuario convertido a formato JSON.
2. **Actualización (update):** Permite modificar los datos de un usuario existente, identificándolo mediante su campo idUsuario. La clave interna generada por Firebase es recuperada antes de ejecutar la actualización.
3. **Eliminación (delete):** Elimina un usuario de la base de datos utilizando su ID único, previa obtención de la clave de referencia.
4. **Consulta individual (get):** Recupera los datos de un usuario por su ID y los convierte en un objeto Usuario.
5. **Consulta por correo (getByEmail):** Permite recuperar un usuario a partir de su campo correo, útil para autenticación o procesos de validación.
6. **Listado completo (getAll):** Obtiene todos los registros de la colección de usuarios, convirtiéndolos en objetos Usuario mediante mapeo.
7. **Gestión del token FCM (getUserFCMToken, updateFCMToken):** Permite obtener y actualizar el token de mensajería FCM de un usuario específico, asegurando la correcta recepción de notificaciones push.

**Observaciones técnicas:**

La estructura del usuario incluye campos como nombre, apellidos, correo, contraseña, rol (chofer, administrativo, gerente), estado activo, ID de vehículo y token FCM.

Se implementa control de errores para validar la existencia de los usuarios antes de ejecutar operaciones críticas.

Es un servicio esencial para el control de accesos, asignación de vehículos y envío de notificaciones personalizadas.

### **GESTION DE VEHICULOS.**

La clase VehiculosHelper es responsable de administrar las operaciones CRUD sobre la colección vehiculos en la base de datos remota Firebase Realtime Database. Este helper se conecta mediante el servicio RealtimeDbHelper y gestiona los datos de las unidades utilizadas en el transporte de personal.

**Funciones principales:**

1. **Creación de vehículos (setNew):** Agrega un nuevo registro de vehículo a la base de datos utilizando su representación en formato JSON.
2. **Actualización (update):** Modifica los datos de un vehículo específico, identificado por su idVehiculo. Primero se obtiene la clave interna del registro en Firebase antes de aplicar los cambios.
3. **Eliminación (delete):** Elimina un vehículo de la base de datos. Se requiere la clave generada por Firebase para realizar esta operación de forma segura.
4. **Consulta individual (get):** Recupera un vehículo específico mediante su ID, devolviendo un objeto Vehiculo mapeado desde los datos en JSON.
5. **Listado completo (getAll):** Obtiene y transforma todos los registros de la colección vehiculos en objetos Vehiculo.

**Observaciones técnicas:**

1. Los vehículos son definidos por campos como matrícula, marca, modelo, capacidad, número de serie y estatus (activo, inactivo, mantenimiento).
2. Las operaciones están diseñadas para garantizar integridad de datos y escalabilidad.
3. El helper es utilizado en módulos de asignación de rutas y monitoreo, siendo clave para asociar vehículos con choferes y viajes activos.

### **GESTION DE PARADAS.**

El módulo de Gestión de Paradas se encarga del manejo de la colección de “paradas” en Firebase Realtime Database. La clase ParadasHelper encapsula todas las operaciones CRUD (crear, leer, actualizar y eliminar) aplicables a los registros de paradas, las cuales representan los puntos de abordaje o bajada de pasajeros.

**Funcionalidades principales:**

1. **Creación de paradas (setNew):**
   1. Registra una nueva parada en la base de datos.
   2. Recibe un objeto Parada, lo convierte a formato JSON y lo envía mediante el método setNewEntry del servicio RealtimeDbHelper.
2. **Actualización de paradas (update):**
3. Permite modificar los datos de una parada existente identificada por su campo idParada.
4. Obtiene la clave única (key) generada por Firebase antes de aplicar la actualización con el método updateEntry.
5. Lanza una excepción si no se encuentra la parada.
6. **Eliminación de paradas (delete):**
7. Elimina una parada de la base de datos a partir de su ID.
8. Similar al proceso de actualización, se obtiene previamente la clave única para remover el registro con deleteEntry.
9. **Consulta individual (get):**
10. Recupera los datos de una parada específica utilizando su ID.
11. Convierte la respuesta JSON en un objeto Parada a través del método Parada.fromJson.
12. **Listado completo (getAll):**
13. Obtiene todas las paradas almacenadas, mapeándolas a objetos Parada para su utilización en la lógica de la aplicación.
14. **Obtención de la clave única (getKey):**
15. Facilita la identificación interna de los registros en Firebase a partir de un campo específico (por ejemplo, "idParada").

Esta implementación permite que el sistema administre de manera eficaz todos los puntos geográficos donde los usuarios interactúan con el servicio, asegurando la integridad y sincronización de la información almacenada en tiempo real.

### **GESTION DE RUTAS.**

El módulo de Gestión de Rutas está a cargo de la administración de la colección “rutas”, la cual agrupa una serie de paradas conformando el recorrido que los choferes deben seguir durante un viaje. La clase RutasHelper gestiona de forma integral todas las operaciones sobre los registros de rutas en la base de datos remota.

**Funcionalidades principales:**

1. **Creación de rutas (setNew):**
   1. Permite agregar una nueva ruta a la base de datos.
   2. Recibe un objeto Ruta, lo transforma a JSON y utiliza el método setNewEntry del RealtimeDbHelper para insertar el registro.
2. **Actualización de rutas (update):**
3. Posibilita modificar los detalles de una ruta existente, identificándola a través del campo idRuta.
4. Para ello, primero se extrae la clave única correspondiente y luego se actualiza la entrada con updateEntry.
5. Si no se encuentra la ruta, se lanza una excepción.
6. **Eliminación de rutas (delete):**
7. Elimina una ruta específica de la colección.
8. Similar a la actualización, primero se obtiene la clave interna con getKey y posteriormente se invoca el método deleteEntry.
9. **Consulta individual (get):**
10. Recupera los datos de una ruta por su ID único.
11. Convierte el registro JSON obtenido en un objeto Ruta utilizando el método Ruta.fromJson.
12. **Listado completo (getAll):**
13. Obtiene todos los registros de rutas almacenados.
14. Los datos se convierten a objetos Ruta para facilitar la manipulación y visualización en la aplicación.
15. **Obtención de la clave única (getKey):**
16. Proporciona la clave interna necesaria para identificar y operar sobre un registro específico de la colección “rutas”, utilizando un campo identificador, como “idRuta”.

Este servicio es fundamental para organizar y administrar los itinerarios que los choferes deben seguir, garantizando que cada ruta se gestione de manera centralizada y se pueda utilizar para optimizar el seguimiento de viajes y la asignación de recursos en la aplicación.

### **GESTION DE VIAJES.**

La gestión de viajes en la aplicación PadillaRoute utiliza una arquitectura de sincronización dual: los registros se almacenan inicialmente de forma local utilizando ObjectBox, y posteriormente se sincronizan con Firebase Realtime Database para mantener la persistencia remota. Esta lógica está distribuida en dos clases principales: ViajesRegistroHelper (local) y ViajesHelper (remota).

1. **Registro local con ObjectBox – ViajesRegistroHelper:**
2. Utiliza una instancia del Store de ObjectBox para almacenar objetos ViajeRegistro en el dispositivo.
3. Proporciona operaciones básicas:

* **saveRegistro:** guarda un nuevo viaje localmente.
* **getAllRegistros:** recupera todos los viajes almacenados.
* **getRegistro:** obtiene un viaje por su ID.
* **deleteRegistro:** elimina todos los registros (usado tras la sincronización).

1. **Sincronización con Firebase – DataSync:**
2. El servicio DataSync revisa periódicamente si hay viajes pendientes por sincronizar con Realtime Database.
3. Al detectar uno o más viajes, toma el primero, lo convierte al modelo compatible (viajeRegistroRealTime.ViajeRegistro) y lo guarda remotamente usando ViajesHelper.
4. Una vez sincronizado, el viaje es eliminado de la base de datos local.
5. **Gestión remota con Realtime Database – ViajesHelper:**
6. Permite realizar operaciones CRUD completas:

* **setNew:** guarda un viaje en Firebase.
* **update:** actualiza un registro existente identificado por idViaje.
* **delete:** elimina un viaje específico.
* **get:** recupera un viaje individual.
* **getAll:** obtiene la lista completa de registros de viaje.

Este sistema híbrido garantiza que los viajes puedan registrarse y visualizarse incluso sin conexión a internet, sincronizándose automáticamente cuando la red esté disponible.

### **GESTION DE INCIDENTES.**

La gestión de incidentes se realiza completamente en Firebase Realtime Database a través del servicio IncidentesHelper, el cual administra el ciclo de vida de los registros de situaciones atípicas que afectan el normal desarrollo de los viajes (ej. retrasos, desvíos, fallas mecánicas).

**Características clave de IncidentesHelper:**

1. **setNew:** crea un nuevo incidente en la base de datos.
2. **update:** actualiza un incidente existente usando su idRegistro.
3. **delete:** elimina un registro de incidente.
4. **get:** consulta individual por ID.
5. **getAll:** devuelve todos los incidentes almacenados.

**Consultas por criterios temporales:**

1. **fetchByMonth:** obtiene todos los incidentes registrados en un mes específico.
2. **fetchByDate:** filtra los incidentes por día.
3. **fetchByDateRange:** recupera los incidentes dentro de un intervalo de fechas definido.

Todos los métodos utilizan el conector RealtimeDbHelper para abstraer la lógica de acceso a la base de datos. Esta herramienta facilita el análisis de la operación mediante informes mensuales o diarios, detectando patrones y áreas críticas que requieren intervención.

### **LOGS DEL SISTEMA.**

El sistema de logs registra acciones importantes realizadas por los usuarios dentro de la aplicación, como inicios de sesión, creación o modificación de rutas, incidentes reportados, obtención de ubicaciones, entre otros. Este mecanismo permite llevar un control exhaustivo de los eventos para auditoría, trazabilidad y mejora continua.

**Gestión de logs mediante LogsHelper:**

1. Se conecta a la colección logs en Firebase.
2. **setNew:** registra un nuevo evento, almacenando datos como usuario, tipo de operación, descripción y fecha/hora.
3. **update:** permite modificar un registro de log si es necesario.
4. **delete:** elimina un log específico.
5. **get:** consulta individual.
6. **getAll:** devuelve todos los logs registrados.

Los logs pueden ser consultados por los administradores para evaluar el comportamiento de los usuarios, auditar cambios realizados en el sistema o detectar posibles errores operativos.

Cada entrada de log contiene al menos:

1. ID del log.
2. Correo del usuario que ejecutó la acción.
3. Tipo de acción (alta, baja, modificación).
4. Descripción del evento.
5. Fecha y hora exacta del suceso.

Este sistema aporta transparencia y seguridad, permitiendo rastrear el historial de acciones en tiempo real.

# **INTERFAZ (FRONTEND).**

## **PANTALLAS.**

## **WIDGETS PERSONALIZADOS.**

# **SEGURIDAD.**

## **AUTENTICACIÓN DE IDENTIDAD DE USUARIOS.**

## **BUENAS PRACTICAS.**

### **ENCRIPTACION DE CONTRACSEÑAS.**

## **PERMISOS**.

TABLA DE IMAGENES.

[Imagen 1.1 Patrón arquitectónico de la aplicación. 5](#_Toc195005956)

[Imagen 1.2 Flujo de datos del sistema. 6](#_Toc195005957)

[Imagen 2.1 Base de datos de la aplicación. 7](#_Toc195005958)