

MANUAL DEL PROGRAMADOR

SPORTRESERVE MOBILE



MANUAL REALIZADO POR

JUAN CAMILO NEUTA SUNCE

FUNDACIÓN ESCUELA TECNOLOGICA DE NEIVA “JESÚS OVIEDO PEREZ” PROYECTO DE GRADO – CICLO “TECNOLOGÍA EN DESARROLLO DE SISTEMAS DE INFORMACIÓN Y REDES”

RIVERA - HUILA

Tabla de contenido

[Introducción 6](#_Toc213341311)

[2.1 Diagrama General del Sistema 7](#_Toc213341312)

[2.2 Descripción de la Arquitectura Cliente-Servidor 8](#_Toc213341313)

[2.3 Tecnologías Empleadas 9](#_Toc213341314)

[3. Frontend – Aplicación Móvil (Flutter) 10](#_Toc213341315)

[3.1 Módulo de Autenticación (auth/) 12](#_Toc213341316)

[3.1.1 Widget AuthTextField 12](#_Toc213341317)

[3.1.2. Página/screen LoginPage 13](#_Toc213341318)

[3.1.3 Página/screen RegisterPage 14](#_Toc213341319)

[3.2 Módulo de Canchas (canchas/) 15](#_Toc213341320)

[3.2.1 Página/screen CanchaDetailPage 15](#_Toc213341321)

[3.2.2 Página/Screen CanchasPage 18](#_Toc213341322)

[3.2.3 Página/Screen MapaCanchasPage 21](#_Toc213341323)

[3.2.4 Página/Screen RegistrarCanchaPage 25](#_Toc213341324)

[3.2.5 Página/Screen OnBoardingPage 29](#_Toc213341325)

[3.2.6. Página/Screen UserProfilePage 34](#_Toc213341326)

[3.2.7. Página/Screen Mis\_Reservas\_Page 38](#_Toc213341327)

[3.3 Mappers 42](#_Toc213341328)

[3.4 Archivo app\_database.dart 44](#_Toc213341329)

[3.4.1. Descripción general 45](#_Toc213341330)

[3.4.2. Dependencias principales 45](#_Toc213341331)

[3.4.3. Estructura del código: 45](#_Toc213341332)

[3.4.4. Tablas principales 46](#_Toc213341333)

[3.4.5. Clase principal *AppDatabase* 47](#_Toc213341334)

[3.4.6. Métodos principales 47](#_Toc213341335)

[*3.4.6.1.* *cachearCanchas(List<Map<String, dynamic>> data)* 47](#_Toc213341336)

[3.4.6.2. ObtenerCanchas() 48](#_Toc213341337)

[3.4.6.3. limpiarCache() 49](#_Toc213341338)

[3.4.7. función \_openConnection() 49](#_Toc213341339)

[3.5. Modelo cancha\_meta.dart: 51](#_Toc213341340)

[3.5.1. Descripción general 51](#_Toc213341341)

[3.5.2. Dependencias principales 51](#_Toc213341342)

[3.5.3. Estructura del archivo 52](#_Toc213341343)

[3.5.3.1. Importaciones 52](#_Toc213341344)

[3.5.3.2. Definición de la clase CanchaMeta 52](#_Toc213341345)

[3.5.3.3. Constructor por fábrica *buildFromModel* 54](#_Toc213341346)

[3.5.3.4. Alias estático fromCancha (Alias que mejora la legibilidad cuando se usa en vistas o servicios.) 55](#_Toc213341347)

[3.5.3.5. Métodos auxiliares visuales 55](#_Toc213341348)

[3.6. Modelo cancha.dart 56](#_Toc213341349)

[3.6.1. Descripción general: 56](#_Toc213341350)

[3.6.2. Estructura general 56](#_Toc213341351)

[3.6.2.1. Campos del modelo cancha.dart: 57](#_Toc213341352)

[3.6.3. Constructor principal 58](#_Toc213341353)

[3.6.3.1. Características: 58](#_Toc213341354)

[3.6.4. Conversiones 58](#_Toc213341355)

[3.6.4.1. Conversión desde JSON (fromJson) 58](#_Toc213341356)

[3.6.4.2. Conversión hacia JSON (toJson) 60](#_Toc213341357)

[3.7. Modelo reservation\_slot.dart 62](#_Toc213341358)

[3.7.1. Descripción general 62](#_Toc213341359)

[3.7.2. Estructura general del modelo reservation\_slot.dart 62](#_Toc213341360)

[3.7.3. Constructor principal 63](#_Toc213341361)

[3.8. Modelo reservation.dart 64](#_Toc213341362)

[3.8.1. Descripción general 64](#_Toc213341363)

[3.8.2. Estructura de la clase 64](#_Toc213341364)

[3.8.3. Propiedad calculada Duration 65](#_Toc213341365)

[3.8.4. Validaciones internas 65](#_Toc213341366)

[3.8.5. Integración con otros componentes 65](#_Toc213341367)

[3.9. Modelo *user\_profile\_laravel.dart*: 66](#_Toc213341368)

[3.9.1. Descripción general: 66](#_Toc213341369)

[3.9.2. Estructura de la clase 66](#_Toc213341370)

[3.9.3. Constructor principal 67](#_Toc213341371)

[3.9.4. Integración con otros módulos 67](#_Toc213341372)

[3.10. Reserva\_repository.dart 69](#_Toc213341373)

[3.10.1. Descripción general 69](#_Toc213341374)

[3.10.2. Código base: 69](#_Toc213341375)

[3.10.3. Estructura de la clase 70](#_Toc213341376)

[3.10.4. Posible expansión futura: 70](#_Toc213341377)

[3.11. Servicio app\_initializer.dart 71](#_Toc213341378)

[3.11.1. Descripción general 71](#_Toc213341379)

[3.11.2. Dependencias principales 71](#_Toc213341380)

[3.11.3. Estructura general 72](#_Toc213341381)

[3.11.4. Método principal initialize() 73](#_Toc213341382)

[3.12. Servicio auth\_service.dart 74](#_Toc213341383)

[3.12.1. Descripción general 74](#_Toc213341384)

[3.12.2. Dependencias principales 74](#_Toc213341385)

[3.12.3. Arquitectura general: 74](#_Toc213341386)

[3.12.4. Atributos y constantes 75](#_Toc213341387)

[3.12.5. Propiedades públicas 75](#_Toc213341388)

[3.12.6. Métodos principales 75](#_Toc213341389)

[3.12.6.1. 75](#_Toc213341390)

[3.12.6.2. \_ensureSessionLoaded() 76](#_Toc213341391)

[3.12.6.3. \_persistToken() Y \_persistUser() 76](#_Toc213341392)

[3.12.6.4. clearSession() 76](#_Toc213341393)

[3.12.7. Autenticación con el backend 76](#_Toc213341394)

[3.12.7.1. register() 76](#_Toc213341395)

[3.12.7.2. Login() 76](#_Toc213341396)

[3.12.7.3. Logout() 77](#_Toc213341397)

[3.12.8. **Manejo del perfil** 77](#_Toc213341398)

[3.12.8.1. getProfile({bool forceRefresh = false}) 77](#_Toc213341399)

[3.12.8.2. updateProfile() 77](#_Toc213341400)

[3.12.8.3. changePassword() 77](#_Toc213341401)

[3.12.8.4. uploadProfilePhoto(String filePath) 78](#_Toc213341402)

[3.12.8.5. updateLocalProfile() 78](#_Toc213341403)

[3.12.9. **Comunicación HTTP segura** 78](#_Toc213341404)

[3.12.10. Persistencia y sincronización local 78](#_Toc213341405)

[3.12.11. Ciclo de autenticación 79](#_Toc213341406)

[3.12.12. Diagrama simplificado 79](#_Toc213341407)

[**3.13.** **Servicio cancha\_service.dart** 80](#_Toc213341408)

[3.13.1. Descripción general 80](#_Toc213341409)

[3.13.2. Dependencias principales 80](#_Toc213341410)

[3.13.3. Estructura General 81](#_Toc213341411)

[3.13.3.1. Método obtenerCanchas() 81](#_Toc213341412)

[3.13.3.2. Método getCanchaById(int id) 82](#_Toc213341413)

[3.13.4. Integración con la base local (Drift) 82](#_Toc213341414)

[3.13.5. Manejo de errores y depuración 83](#_Toc213341415)

[**3.14.** **Servicio notification\_service.dart** 84](#_Toc213341416)

[3.14.1. Descripción general 84](#_Toc213341417)

[3.14.2. Dependencias principales 84](#_Toc213341418)

[3.14.3. Arquitectura de la clase 84](#_Toc213341419)

[3.14.3.1. Inicialización del servicio 85](#_Toc213341420)

[3.14.3.2. Configuración de zona horaria 86](#_Toc213341421)

[3.14.3.3. Configuración del sistema de notificaciones 86](#_Toc213341422)

[3.14.3.4. Notificaciones inmediatas 87](#_Toc213341423)

[3.14.3.5. Programación de recordatorios 88](#_Toc213341424)

[3.14.3.6. Interacción del usuario 89](#_Toc213341425)

[3.14.3.7. Preferencias del usuario 89](#_Toc213341426)

[3.15. Servicio reservation\_service.dart 91](#_Toc213341427)

[3.15.1. Descripción general 91](#_Toc213341428)

[3.15.2. Dependencias principales 91](#_Toc213341429)

[3.15.3. Estructura de la clase 91](#_Toc213341430)

[3.15.4. Funcionalidadesprincipales 92](#_Toc213341431)

[3.15.4.1. Crear una nueva reserva 92](#_Toc213341432)

[3.15.4.2. Flujo de ejecución 92](#_Toc213341433)

[3.15.4.3. Consultar disponibilidad de una cancha 93](#_Toc213341434)

[3.15.4.4. Obtener reservas del usuario autenticado 94](#_Toc213341435)

[3.15.4.5. Cancelar una reserva existente 95](#_Toc213341436)

[3.15.4.6. Obtener horarios disponibles por día 95](#_Toc213341437)

# Introducción

El presente documento constituye el Manual del Desarrollador del sistema SportReserve Mobile, una aplicación híbrida que integra un frontend desarrollado en Flutter (Dart) y un backend implementado en Laravel (PHP), respaldado por una base de datos PostgreSQL.  
El objetivo de este manual es servir como guía técnica integral para los desarrolladores responsables del mantenimiento, la evolución y la ampliación de las funcionalidades del sistema. Se ofrece una descripción estructurada de la arquitectura general, los componentes, las dependencias, las rutas, los modelos y los servicios que conforman tanto la capa cliente como la capa servidor.

El documento también detalla los procedimientos de instalación, configuración, despliegue y conexión entre los entornos, asegurando una sincronización confiable entre la aplicación móvil y el backend mediante APIs REST seguras, autenticadas con el mecanismo de Laravel Sanctum.

Adicionalmente, este manual busca establecer y unificar las buenas prácticas de desarrollo aplicables al proyecto SportReserve, promoviendo la implementación de un código modular, reutilizable y mantenible, en concordancia con los principios de arquitectura limpia, desacoplamiento funcional y escalabilidad tecnológica.2. Arquitectura general del sistema

## 2.1 Diagrama General del Sistema

El sistema SportReserve Mobile adopta una arquitectura cliente-servidor distribuida, donde la aplicación móvil (frontend) actúa como cliente, interactuando con un servidor web (backend) implementando en Laravel, que a su vez se comunica con una base de datos PostgreSQL para el almacenamiento persistente de la información.

A nivel general, el flujo de comunicación se represente así:



Esta arquitectura facilita la escalabilidad, mantenimiento modular y la integración con futuros servicios, como notificaciones push, almacenamiento en la nube o dashboards administrativos web

## 2.2 Descripción de la Arquitectura Cliente-Servidor

El sistema está estructurado bajo una arquitectura multicapa, con responsabilidades claramente separadas:

* Capa de Presentación (Frontend Flutter):  
  Se encarga de la interfaz de usuario, experiencia visual y navegación. Implementa comunicación asíncrona con el backend mediante peticiones HTTP, consumo de APIs REST y manejo de estados locales.
* Capa Lógica y Negocio (Backend Laravel):  
  Implementa las reglas de negocio, validaciones de datos, control de acceso y flujo de información entre el cliente y la base de datos. Está protegida por un sistema de autenticación basado en Laravel Sanctum, que utiliza tokens personales para asegurar la comunicación.
* Capa de Datos (PostgreSQL):  
  Contiene las tablas y relaciones del sistema, garantizando integridad referencial y consistencia transaccional. Incluye entidades como usuarios, canchas, reservas, pagos, y roles administrativos.

La interacción entre estas capas se realiza a través de endpoints REST definidos en el backend, que retornan respuestas JSON estructuradas para su interpretación en la aplicación móvil.

## 2.3 Tecnologías Empleadas

|  |  |  |
| --- | --- | --- |
| Componente | Tecnología | Función principal |
| Frontend móvil | Flutter (Dart) | Desarrollo de la interfaz de usuario y experiencia móvil nativa para Android/iOS. |
| Backend API | Laravel (PHP 10.x) | Framework backend para la lógica de negocio, autenticación y exposición de endpoints REST. |
| Base de datos | PostgreSQL | Almacenamiento estructurado y relacional de todos los datos del sistema. |
| Autenticación | Laravel Sanctum | Sistema de autenticación mediante tokens para solicitudes seguras desde el cliente móvil. |
| Mapa interactivo | Flutter Map + OpenStreetMap | Visualización de ubicaciones y detalles de canchas deportivas. |
| Almacenamiento local | Shared Preferences | Gestión de sesiones y configuraciones persistentes en el dispositivo. |
| Gestión de imágenes | Image Picker | Carga y actualización de fotos de perfil y comprobantes de pago. |
| Control de versiones | Git / GitHub | Control de cambios, ramas y despliegue colaborativo. |

# 3. Frontend – Aplicación Móvil (Flutter)

El frontend del sistema está construido utilizando Flutter bajo el patrón de arquitectura modular. El árbol de directorios se encuentra organizado por funcionalidades (features), servicios, modelos y recursos. A continuación, se describe la estructura general del proyecto con sus principales componentes:

* **Lib/**

Directorio principal del código fuente de la aplicación

* **|---- features/**

Contiene los módulos funcionales de la app, organizados por características (auth, reservas, canchas, perfil, etc.).

* **| |---- auth**/

Incluye las pantallas y controladores relacionados con el inicio de sesión, registro y autenticación de usuarios

* **| |---- reservas/**

Contiene la lógica y vistas para la gestión de reservas, historial y comprobantes

* **| |---- canchas/**

Maneja el módulo de canchas, incluyendo mapa, detalles e información adicional de cada cancha.

* **| |---- profile/**

Sección destinada a la gestión del perfil de usuario y opciones personales.

* **| |---- models/**

Define las clases de modelo utilizadas para representar entidades como Usuario, Reserva y Cancha.

* **| |---- services/**

Agrupa los servicios responsables de la comunicación con el backend Laravel mediante peticiones HTTP.

* **| |---- widgets/**

Componentes reutilizables de interfaz como botones, campos de texto y tarjetas.

* **| |---- router/**

Define la configuración de rutas y navegación de la aplicación utilizando GoRouter.

* **| |---- main.dart/**

Punto de entrada principal de la aplicación. Inicializa dependencias, temas y configuración de rutas.

* **| |---- assets/**

Directorio que almacena recursos gráficos, íconos, fuentes y archivos de configuración utilizados por la app.

3.1 Módulo de Autenticación (auth/)

3.1.1 Widget AuthTextField

Ubicación *lib/features/auth/widgets/auth\_text\_field.dart*

Descripción general:

El widget AuthTextField define un componente de entrada de texto reutilizable para los formularios del módulo de autenticación. Su objetivo es unificar el estilo visual de los campos de texto (correo, nombre, contraseña, etc.) bajo los lineamientos de diseño del sistema SportReserve Mobile.  
Admite modo **oscuro o claro**, mantiene coherencia con la tipografía oficial (Google Fonts – *Poppins*) y presenta bordes redondeados con retroalimentación visual al recibir foco.

Características técnicas:

* Tipo: StatelessWidget (es un tipo de widget inmutable, su estado interno no cambia durante la ejecución, su función principal es construir una interfaz estática o dependiente únicamente de parámetros externos)
* Personalizable mediante parámetros (label, icon, obscureText, dark).
* Utiliza OutlineInputBorder con efecto dinámico en modo oscuro/claro.
* Integra colorimetría verde institucional de SportReserve.
* Altamente reutilizable en múltiples pantallas (login, registro, recuperación de contraseña).

3.1.2. Página/screen LoginPage

Ubicación *lib/features/auth/login\_page.dart*

Descripción general:

La clase LoginPage implementa la pantalla de **inicio de sesión** de los usuarios.  
Está desarrollada como un StatefulWidget para gestionar estados locales (campos, carga asíncrona, errores, etc.).  
Su diseño combina **estética moderna**, **gradientes oscuros**, **efectos de iluminación verde** y un fondo con imagen deportiva, reforzando la identidad visual de SportReserve.

**Principales responsabilidades:**

* Capturar credenciales del usuario (correo y contraseña).
* Invocar el método AuthService.instance.login() para autenticación mediante API REST.
* Mostrar retroalimentación visual (carga, error, éxito) con SnackBar.
* Redirigir automáticamente según contexto:
  + Si hay una cancha pendiente, regresa a la pantalla correspondiente.
  + En caso contrario, navega al perfil del usuario (/profile).

**Elementos destacados:**

* Campos personalizados mediante \_buildTextField() con bordes brillantes y fondo translúcido.
* Botón principal con animación de gradiente verde y CircularProgressIndicator en carga.
* Texto dinámico para redirección al registro mediante GoRouter.

**Diseño visual:**

* Fondo: imagen assets/images/fondo\_campo.jpg con degradado negro-verde.
* Paleta: tonos verdes (#00E676, #43A047) y blancos.
* Iconografía: Icons.sports\_soccer como logotipo principal

3.1.3 Página/screen RegisterPage

Ubicación *lib/features/auth/register\_page.dart*

Descripción general:

La clase RegisterPage representa la pantalla de registro de nuevos usuarios.  
Está diseñada con un enfoque centrado en la usabilidad y la claridad visual, utilizando un contenedor translúcido sobre una imagen de fondo deportiva y efectos de iluminación verde.

Funciones principales:

* Captura de nombre, correo y contraseña del usuario.
* Validación de datos y envío al backend mediante AuthService.instance.register().
* Presentación de mensajes de confirmación o error con SnackBar.
* Redirección automática al login tras registro exitoso (context.go('/login')).

**Componentes clave:**

* Reutiliza AuthTextField para mantener coherencia visual en los campos de texto.
* Botón principal con diseño moderno (ElevatedButton) y animación de carga.
* Contenedor con borde semitransparente y sombra verde para enfoque del usuario.

**Elementos técnicos:**

* Tipografía: GoogleFonts.poppins para todos los textos.
* Estructura adaptable con SingleChildScrollView para pantallas pequeñas.
* Gestión del estado de carga con variable isLoading.

3.2 Módulo de Canchas (canchas/)

3.2.1 Página/screen CanchaDetailPage

Ubicación: lib/features/canchas/cancha\_detail\_page.dart

Descripción general

La clase CanchaDetailPage implementa la pantalla de detalle y reserva de una cancha deportiva seleccionada desde el mapa o listado.

Su objetivo principal es permitir al usuario autenticado visualizar la información completa de la cancha (nombre, ubicación, descripción, servicios, precio) y realizar una reserva programada mediante la selección de fecha, hora y cantidad de horas.

Esta página combina la presentación de datos con la interacción directa hacia el backend a través del ReservationService, garantizando la conexión entre la vista móvil y las operaciones en el servidor Laravel.

Tipo de widget - StatefulWidget

Este componente mantiene estado interno, ya que necesita actualizar dinámicamente los valores seleccionados (fecha, hora, duración de reserva, estado de carga).

La clase está compuesta por:

* CanchaDetailPage: la clase principal (definición del widget).
* \_CanchaDetailPageState: la clase interna que administra el estado, eventos y renderización del contenido.

Dependencias utilizadas:

|  |  |
| --- | --- |
| Paquete | Uso principal |
| intl | Formateo de fechas (DateFormat) para las reservas. |
| go\_router | Navegación hacia otras pantallas (/login, /register). |
| auth\_service.dart | Validación y obtención del usuario autenticado. |
| reservation\_service.dart | Creación de reservas mediante API REST. |
| notification\_service.dart | Envío de notificaciones locales tras una reserva exitosa. |
| cancha\_meta.dart | Representación visual de metadatos de la cancha (tipo, superficie, disponibilidad). |

Lógica principal:

1. Reserva de cancha (\_handleReserve)

* Verifica que el usuario esté autenticado; en caso contrario, muestra un modal de inicio de sesión o registro.
* Valida que se haya seleccionado una hora válida.
* Crea una reserva con:
  + ID de cancha.
  + Fecha (yyyy-MM-dd).
  + Hora (HH:mm).
  + Duración (\_cantidadHoras).
  + Precio unitario (precioPorCancha).
* Muestra notificaciones locales (NotificationService) al confirmar la reserva.

1. Modal de autenticación ( \_showLoginRegisterModal )

* Se ejecuta cuando el usuario intenta reservar sin haber iniciado sesión.
* Implementa un BottomSheet personalizable con opciones para iniciar sesión o crear cuenta.
* El modal se adapta según si la sesión ha expirado o si el usuario es nuevo.

1. Selección de fecha y hora

* Usa showDatePicker y showTimePicker para permitir al usuario elegir fecha y hora de reserva dentro de un rango de 30 días.
* Los valores seleccionados se almacenan en \_selectedDate y \_selectedTime.

1. Construcción de interfaz (build)

La interfaz muestra dos secciones principales:

4.1 Área de reserva (solo si el usuario está autenticado):

* Campos para fecha, hora y duración.
* Botón “Reservar” con animación de carga (CircularProgressIndicator).
* Mensajes dinámicos mediante SnackBar.

4.2 Información detallada de la cancha:

* Descripción general.
* Dirección, precio por hora y lista de servicios.
* Chips visuales que indican disponibilidad, tipo de superficie y estado actual.

Componentes visuales auxiliares

La página define varios widgets internos reutilizables para mantener el código modular y limpio:

|  |  |
| --- | --- |
| Widget | Propósito |
| \_ServiceChip | Representa visualmente los servicios disponibles (por ejemplo: “Parqueadero”, “Iluminación”). |
| \_DetailTile | Muestra información estructurada con ícono, título y subtítulo (ubicación, precio, etc.). |
| \_InfoChip | Chip con ícono e información complementaria (tipo de cancha, superficie). |
| \_AvailabilityChip | Indica disponibilidad actual utilizando los métodos estáticos de CanchaMeta (color, etiqueta, fondo). |

Cada uno de estos widgets es un StatelessWidget, ideal para mostrar información sin requerir estado mutable.

Comportamiento visual y experiencia de usuario

* Diseño moderno con márgenes amplios, bordes redondeados y tipografía limpia.
* Adaptado para desplazamiento vertical mediante SingleChildScrollView.
* Colores dominantes: verde institucional, blanco y gris claro.
* Accesibilidad optimizada con textos alternativos y estructura semántica.

### 3.2.2 Página/Screen CanchasPage

Ubicación: *lib/features/canchas/canchas\_page.dart*

Descripción general

La clase CanchasPage representa la pantalla principal de exploración y reserva de canchas deportivas dentro del sistema.  
Su propósito es mostrar de forma dinámica todas las canchas registradas en el backend y permitir que el usuario consulte sus detalles o acceda al mapa interactivo.

Esta vista es el punto de partida del flujo de reserva e integra tanto la visualización de datos remotos como la navegación hacia otras pantallas (CanchaDetailPage, MapaCanchasPage y UserProfilePage).

Tipo de widget - StatefulWidget

Este componente utiliza estado interno para gestionar la carga de datos asincrónica desde el backend.  
En particular, emplea un *Future<List<Cancha>>* que se inicializa en el método *initState().*

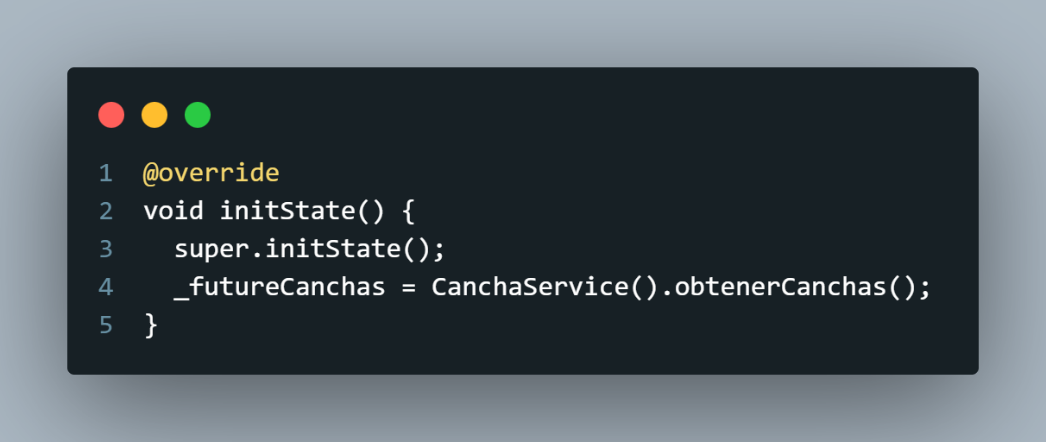
Dependencias Utilizadas

|  |  |
| --- | --- |
| Paquete / Archivo | Función |
| cancha\_service.dart | Obtiene la lista de canchas desde el backend mediante peticiones HTTP. |
| cancha\_detail\_page.dart | Navegación al detalle de una cancha seleccionada. |
| mapa\_canchas\_page.dart | Acceso al mapa con ubicación de las canchas. |
| user\_profile\_page.dart | Redirección al perfil del usuario autenticado. |
| review\_service.dart | Obtiene y monitorea calificaciones y reseñas en tiempo real. |
| cancha\_meta.dart | Calcula metadatos (color, tipo, superficie, íconos, precio) para personalizar la UI. |
| review\_summary.dart | Modelo que resume promedio y cantidad de reseñas. |

Lógica principal:

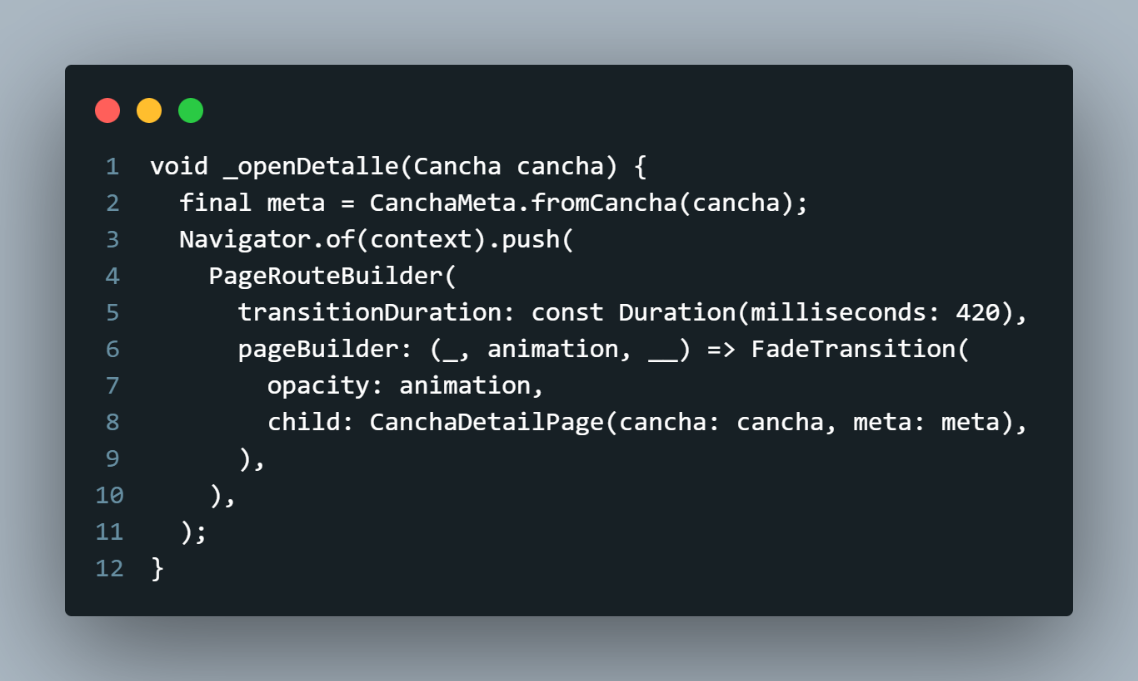
* **Inicialización de datos**

En el método initState(), se instancia \_futureCanchas como un Future que ejecuta CanchaService().obtenerCanchas(). De esta manera, los datos de las canchas se consultan automáticamente al cargar la vista.



* Construcción de interfaz (build)
  + La estructura base de la página está compuesta por:
  + AppBar:
    - Título: *“Reserva tu cancha”*.
    - Iconos de acceso rápido:
      * Perfil del usuario (UserProfilePage)
      * Mapa de canchas (MapaCanchasPage).
  + FutureBuilder:  
    Administra los tres posibles estados del Future:
    - Carga → muestra CircularProgressIndicator.
    - Error → muestra mensaje con descripción del error.
    - Éxito → renderiza la lista de canchas.
  + ListView.separated:  
    Muestra cada cancha en forma de tarjeta personalizada (\_CanchaCard).
* Navegación

El método \_openDetalle(Cancha cancha) permite abrir la página de detalle mediante una animación de **transición suave (FadeTransition)** usando PageRouteBuilder.  
Esto mejora la experiencia de usuario al navegar entre pantallas.



* Componente \_CanchaCard

Cada cancha se muestra dentro de una tarjeta estilizada que contiene información relevante y botones de acción.

Tipo: StatelessWidget

Al no tener estado interno, su única función es presentar los datos recibidos desde CanchasPage.

Estructura de contenido:

* Imagen principal / gradiente de color: representa visualmente el tipo de cancha (fútbol, tenis, baloncesto, etc.) mediante el color definido en CanchaMeta.
* Chips de estado:
  + \_AvailabilityChip → indica disponibilidad.
  + Contenedor con tipo de cancha (meta.type).
* Datos principales:
  + Nombre (cancha.nombre).
  + Ubicación (cancha.ubicacion).
  + Precio por hora (meta.pricePerHour).
  + Calificación promedio (ReviewSummary).
* Acciones:
  + Botón principal “Reservar”.
  + Botón secundario “Info” (navegación al detalle).

### 3.2.3 Página/Screen MapaCanchasPage

Ubicación: *lib/features/canchas/mapa\_canchas\_page.dart*

-Descripción general

MapaCanchasPage es la pantalla dedicada a la visualización geográfica interactiva de las canchas deportivas disponibles en el sistema SportReserve.  
Su objetivo principal es permitir al usuario localizar, explorar y acceder a la información de cada cancha mediante un mapa dinámico implementado con la librería flutter\_map (basada en OpenStreetMap).

Desde esta vista, el usuario puede:

* Consultar la posición exacta de las canchas registradas.
* Identificar disponibilidad mediante colores e íconos personalizados.
* Visualizar una hoja informativa flotante (*bottom sheet*) con los detalles principales de la cancha.
* Acceder directamente al detalle completo para reservar.

Dependencias utilizadas:

|  |  |
| --- | --- |
| Paquete / Archivo | Propósito |
| flutter\_map | Renderización del mapa y administración de capas (tiles, marcadores). |
| latlong2 | Representación de coordenadas geográficas (LatLng). |
| cancha\_service.dart | Obtención de la lista de canchas desde el backend Laravel. |
| cancha\_meta.dart | Generación de metadatos visuales (icono, color, tipo, disponibilidad). |
| cancha\_detail\_page.dart | Navegación hacia el detalle completo de una cancha seleccionada. |
| auth\_service.dart | Validación del perfil del usuario autenticado. |
| user\_profile\_laravel.dart | Modelo del perfil de usuario. |
| go\_router.dart | Navegación declarativa entre rutas del sistema. |

**Lógica funcional principal**

**1. Carga inicial de datos**

En el método initState(), se ejecutan dos procesos paralelos:

* \_loadCanchas():  
  Obtiene la lista de canchas desde el backend mediante CanchaService().obtenerCanchas() y actualiza la interfaz una vez completada la carga.
* \_loadUserProfile():  
  Recupera el perfil del usuario autenticado utilizando AuthService.instance.getProfile().



* Visualización del mapa

El mapa se renderiza mediante el widget FlutterMap, configurado con:

* Centro inicial en Neiva, Huila (LatLng(2.9360, -75.2895)).
* Zoom predeterminado: 14.5.
* Capa base (TileLayer) con estilo claro de CartoDB.
* Capa de marcadores (MarkerLayer) que muestra todas las canchas disponibles.



Cada marcador corresponde a una cancha y se genera dinámicamente con la clase \_FacilityMarker.

* Interacción con marcadores

Cuando el usuario toca un marcador:

* Se almacena la cancha seleccionada en \_selectedCancha.
* El mapa se reposiciona ligeramente hacia arriba para que el marcador no quede cubierto por el panel inferior.



* Panel inferior (Bottom Sheet)

Al seleccionar una cancha, aparece un panel flotante \_SelectedCanchaSheet con:

* Nombre de la cancha e ícono representativo.
* Dirección exacta.
* Tarifa por hora.
* Botón para “Ver detalles”, que redirige a CanchaDetailPage.

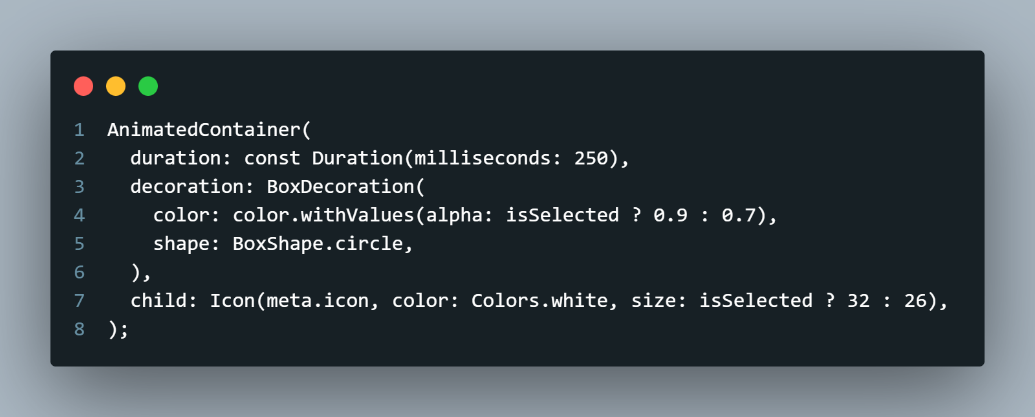
Este panel usa animaciones suaves (AnimatedPositioned) y diseño translúcido con bordes redondeados para una experiencia moderna.

* Control de zoom

En la esquina inferior derecha se ubican dos botones (FloatingActionButton) que permiten aumentar o disminuir el nivel de zoom de manera gradual.  
El mapa se actualiza instantáneamente con \_mapController.move().

Componentes visuales auxiliares

|  |  |  |
| --- | --- | --- |
| Componente | Tipo | Función |
| \_FacilityMarker | StatelessWidget | Representa el marcador visual de cada cancha. Cambia de tamaño y opacidad si está seleccionada. |
| \_SelectedCanchaSheet | StatelessWidget | Panel inferior que muestra los datos resumidos de la cancha seleccionada con botón de navegación. |



### 3.2.4 Página/Screen RegistrarCanchaPage

Ubicación: lib/features/canchas/registrar\_cancha\_page.dart

Descripción general

La clase RegistrarCanchaPage implementa la pantalla de registro de canchas deportivas dentro del módulo de administración de SportReserve Mobile.  
Permite crear nuevos registros enviando los datos de una cancha (nombre, tipo, ubicación, coordenadas geográficas, precio y disponibilidad) hacia la API del backend Laravel mediante solicitudes HTTP.

Este formulario fue diseñado para ser intuitivo, validado y completamente funcional, integrando una experiencia visual consistente con el tema de la aplicación.

Tipo de widget - StatefulWidget

Se emplea un widget con estado mutable debido a la necesidad de controlar el estado del formulario, el progreso de la operación (variable \_loading) y la interacción con los campos controlados mediante TextEditingController.

Dependencias utilizadas

|  |  |
| --- | --- |
| Paquete / Librería | Propósito |
| flutter/material.dart | Construcción de la interfaz visual. |
| http | Envío de solicitudes HTTP tipo POST al backend Laravel. |
| dart:convert | Codificación de los datos en formato JSON antes del envío. |

Lógica funcional principal

* Controladores del formulario

Se declaran seis controladores de texto (TextEditingController) que capturan los datos ingresados por el usuario:

|  |  |  |
| --- | --- | --- |
| Controlador | Campo asociado | Descripción |
| \_nombreController | Nombre | Nombre comercial de la cancha. |
| \_tipoController | Tipo | Tipo de cancha (fútbol 5, tenis, baloncesto, etc.). |
| \_ubicacionController | Ubicación | Dirección física o descripción geográfica. |
| \_latController | Latitud | Coordenada geográfica (norte-sur). |
| \_lngController | Longitud | Coordenada geográfica (este-oeste). |
| \_precioController | Precio por hora | Valor cobrado por hora de uso de la cancha. |

* Envío de datos (\_registrarCancha)

El método privado \_registrarCancha() realiza todo el proceso de validación y comunicación con el backend.

Flujo de ejecución:

1. Verifica que todos los campos sean válidos mediante \_formKey.currentState!.validate().
2. Construye el cuerpo del mensaje (body) en formato JSON.
3. Envía una solicitud POST al endpoint: <http://192.168.20.82:8000/api/canchas>
4. Si la respuesta del servidor tiene código **201 (Created)** o **200 (OK)**:

* Muestra un SnackBar de éxito.
* Limpia todos los campos del formulario.
* Espera un segundo y retorna a la vista anterior (Navigator.pop(context, true)).
* En caso de error, muestra mensajes detallados en consola y en la interfaz.

Fragmento Principal:



* Validaciones

Cada campo del formulario tiene una validación básica implementada directamente en su propiedad validator:

*validator: (v) => v!.isEmpty ? 'Campo obligatorio' : null,*

Esto asegura que no se envíen solicitudes incompletas y que todos los datos requeridos estén presentes antes de la comunicación con el backend.

* **Interfaz de usuario**

La estructura de la pantalla está compuesta por:

* AppBar con el título *“Registrar Cancha”*.
* Form con *GlobalKey<FormState>* para control de validaciones.
* ListView con campos organizados en orden lógico.
* Campos numéricos *(keyboardType: TextInputType.number)* para latitud, longitud y precio.
* Botón principal (*ElevatedButton.icon*) con:
  + Ícono Icons.save\_rounded.
  + Indicador de carga (CircularProgressIndicator) cuando \_loading == true.
  + Validación de estado para evitar múltiples envíos simultáneos.

Diseño visual:

* Márgenes uniformes (padding: EdgeInsets.all(20)).
* Espaciado entre filas (SizedBox(height: 20)).
* Colores coherentes con el tema (scheme.primary).

Consideraciones técnicas

* La dirección IP local (192.168.20.82:8000) debe reemplazarse por el dominio o dirección pública del backend en entornos de producción.
* La variable \_loading previene dobles envíos y permite feedback visual al usuario.
* El uso de jsonEncode garantiza compatibilidad con la API REST desarrollada en Laravel.
* El formulario está completamente encapsulado dentro del widget, asegurando un manejo de estado seguro y desacoplado.

### 3.2.5 Página/Screen OnBoardingPage

Ubicación: lib/features/onboarding/onboarding\_page.dart

Descripción general

La clase OnboardingPage implementa la pantalla de introducción y bienvenida (onboarding) del sistema SportReserve Mobile.  
Este módulo tiene como propósito guiar al usuario nuevo a través de una serie de pantallas ilustrativas, que explican brevemente la funcionalidad principal de la aplicación antes de ingresar al mapa o al flujo de autenticación.

El sistema utiliza un controlador de páginas (PageController) y almacenamiento persistente con SharedPreferences para asegurarse de que la experiencia de onboarding se muestre solo una vez (la primera vez que el usuario abre la aplicación).

Tipo de widget - StatefulWidget

Se utiliza un widget con estado mutable porque la vista debe reaccionar dinámicamente a los cambios de página, el índice actual (currentIndex) y la finalización del proceso de onboarding.

La clase se divide en:

* OnboardingPage: definición del widget.
* \_OnboardingPageState: lógica del comportamiento y renderizado.

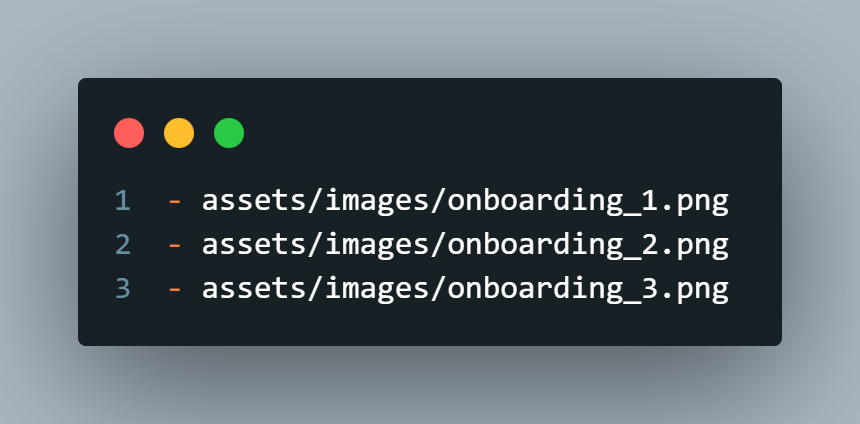
Dependencias utilizadas:

|  |  |
| --- | --- |
| Paquete / Librería | Propósito |
| flutter/material.dart | Construcción de la interfaz visual. |
| flutter/services.dart | Configuración de estilo del sistema (barras de estado y navegación). |
| shared\_preferences | Almacenamiento persistente local (para registrar si el usuario ya vio el onboarding). |
| go\_router | Navegación declarativa hacia la siguiente ruta tras finalizar la introducción. |

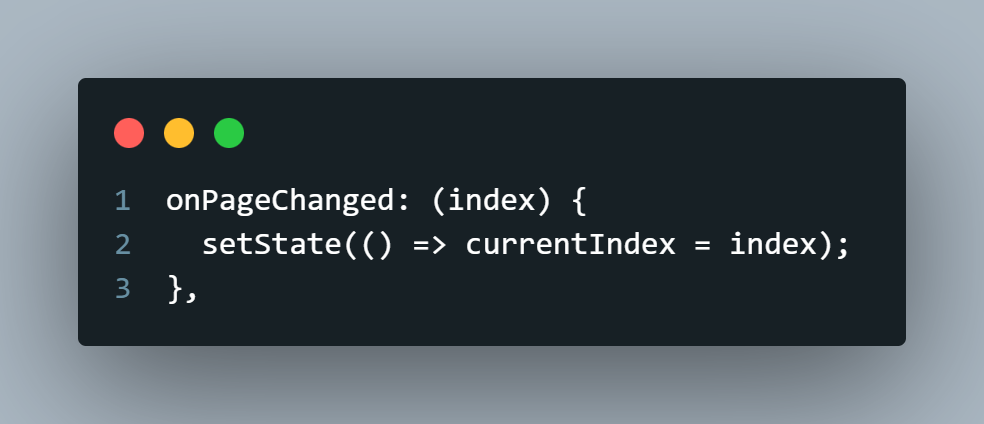
Lógica principal:

* Control del flujo de páginas

El widget utiliza un PageView.builder controlado por un PageController para generar tres páginas de presentación, cada una con una imagen ilustrativa ubicada en:



El índice de página actual se guarda en la variable currentIndex, que se actualiza en tiempo real en el método:



* Finalización del Onboarding

El método \_finishOnboarding() realiza dos acciones principales:

* + Guarda el valor hasSeenOnboarding = true en SharedPreferences, indicando que el usuario ya completó el proceso.
  + Redirige al usuario hacia la siguiente pantalla (por defecto /mapa) utilizando GoRouter:



Este comportamiento garantiza que, en ejecuciones futuras, la aplicación pueda omitir el onboarding y llevar al usuario directamente a la vista principal (mapa o login).

* Navegación entre páginas

En la parte inferior se encuentran dos botones de control (IconButton):

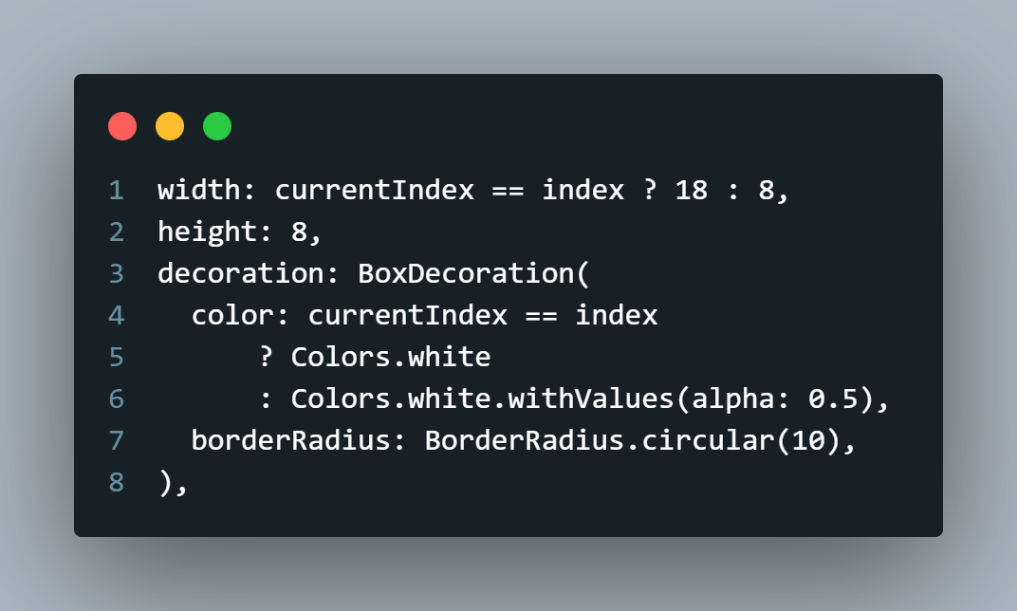
* Retroceder:

\_controller.previousPage(...)

Desplaza una página hacia atrás si currentIndex > 0.

* Avanzar o finalizar:
  + Si el usuario está en la última página (currentIndex == onboardingImages.length - 1), se ejecuta \_finishOnboarding().
  + Si no, avanza a la siguiente con \_controller.nextPage(...).
* Indicadores de progreso

Se muestra un conjunto de puntos (*dots indicators*) que reflejan el progreso del usuario en el recorrido.  
Cada punto se genera dinámicamente con List.generate() y se anima mediante AnimatedContainer:



El indicador activo se agranda y cambia de opacidad para una sensación de movimiento fluida.

* Configuración visual y de sistema

Antes de construir la interfaz, se configura el estilo de la barra de estado y navegación mediante SystemChrome.setSystemUIOverlayStyle:



Esto asegura coherencia visual con el color principal de SportReserve (verde #3F815D) y una interfaz moderna tipo “pantalla completa”.

* Diseño e interacción

Componentes visuales principales:

* PageView: muestra las imágenes de introducción.
* Indicadores inferiores: muestran el avance entre páginas.
* Botones de control: permiten moverse hacia adelante o atrás.
* Colores dominantes:
  + Fondo: verde institucional (#3F815D).
  + Superposiciones: negro translúcido (alpha 0.3).
  + Íconos: blancos (Colors.white).

Interacción destacada:

* Animaciones suaves (Duration(milliseconds: 300–400)).
* Diseño centrado en imágenes a pantalla completa.
* Controles accesibles desde la parte inferior de la pantalla.

Consideraciones técnicas

* La ruta /mapa debe existir en el archivo router.dart del proyecto.  
  En caso contrario, se debe ajustar a la ruta de inicio del sistema (por ejemplo /login o /home).
* Si se desea reiniciar la experiencia de onboarding, basta con borrar la clave hasSeenOnboarding desde SharedPreferences.
* El diseño es responsivo y se adapta automáticamente a distintas resoluciones gracias a los cálculos de MediaQuery.

### 3.2.6. Página/Screen UserProfilePage

Ubicación**:** lib/features/profile/user\_profile\_page.dart

* Descripción general

UserProfilePage representa la pantalla de gestión del perfil del usuario dentro de la aplicación móvil.  
Su objetivo principal es mostrar la información personal del usuario autenticado, permitir la actualización de su foto de perfil, el cambio de contraseña, la consulta de sus reservas y la posibilidad de cerrar sesión de forma segura.

La vista se adapta automáticamente al estado de sesión:

* Si el usuario está autenticado, muestra su perfil completo.
* Si no ha iniciado sesión, presenta una vista informativa con opciones para acceder o registrarse.

Tipo de widget - StatefulWidget

Se utiliza para mantener y actualizar dinámicamente la información del perfil, ya que algunas operaciones (como actualizar foto, cambiar contraseña o cerrar sesión) modifican el estado en tiempo real.

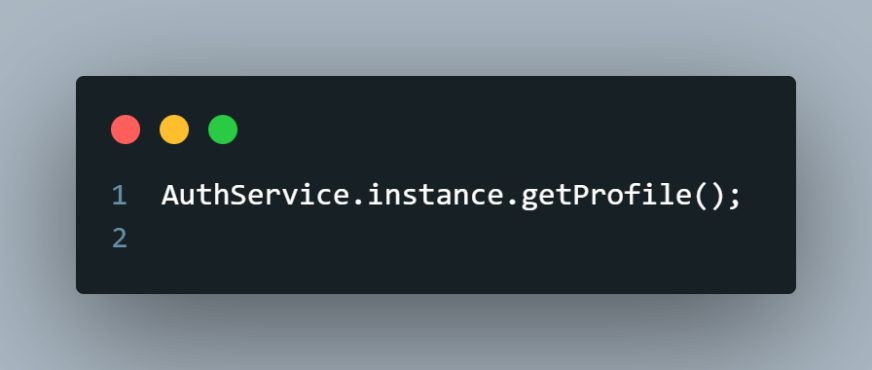
* Dependencias utilizadas

|  |  |
| --- | --- |
| Paquete / Librería | Propósito |
| flutter/material.dart | Construcción de la interfaz visual. |
| image\_picker | Selección de imágenes desde la galería del dispositivo. |
| go\_router | Navegación entre rutas (por ejemplo: /mapa, /login, /register, /mis-reservas). |
| auth\_service.dart | Gestión de autenticación, obtención y actualización del perfil. |
| user\_profile\_laravel.dart | Modelo de datos del perfil de usuario. |

Lógica Principal

* Carga inicial del perfil

En el método initState(), se inicializa la variable \_profileFuture con la llamada a:



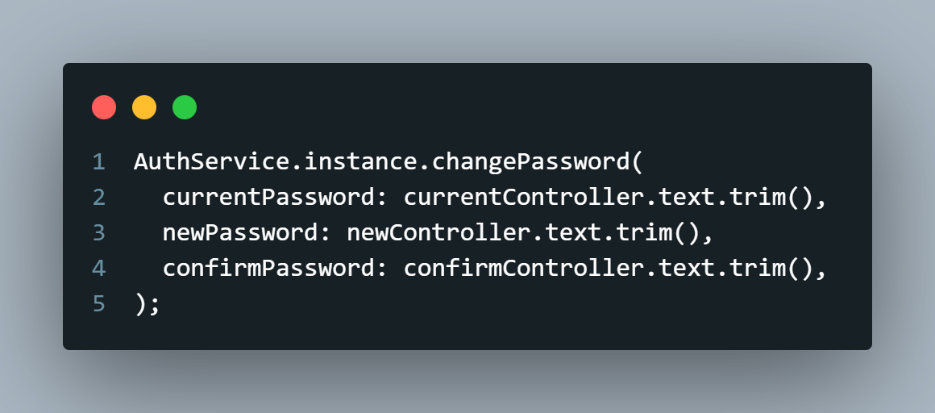
Esto obtiene los datos del usuario autenticado (nombre, correo, rol y foto) desde el backend Laravel.

* Cambio de contraseña

El método \_showChangePasswordDialog() despliega un cuadro de diálogo modal que contiene tres campos:

* Contraseña actual
* Nueva contraseña
* Confirmación de nueva contraseña

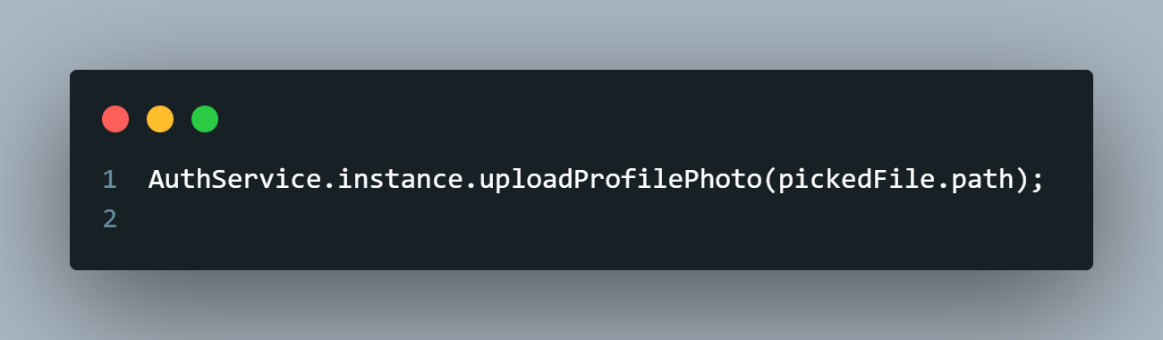
Luego invoca:



Si la operación es exitosa, muestra un SnackBar verde con confirmación; si falla, notifica el error en rojo.

* Cambio de foto de perfil

El método \_pickAndUploadPhoto() permite seleccionar una nueva imagen de perfil desde la galería mediante ImagePicker.  
Posteriormente, sube la imagen al servidor con:



Al completarse, actualiza la instancia local del usuario (updateLocalProfile) y refresca la vista con setState().

* Cierre de sesión

La función \_logout() despliega un modal de confirmación con botones “Cancelar” y “Sí, cerrar”.  
Si el usuario confirma:

1. Se ejecuta AuthService.instance.logout().
2. Se redirige al mapa con: context.go('/mapa');

Esto garantiza que las rutas protegidas no sean accesibles tras cerrar sesión.

* Estructura visual

AppBar

* Título: “Mi perfil”.
* Botón de retroceso (←) hacia /mapa.

Cuerpo principal (body)

Dividido en dos escenarios:

A. Usuario autenticado

Se muestra:

* Tarjeta principal con datos del usuario:
  + Foto de perfil (CircleAvatar con imagen de red o ícono predeterminado).
  + Nombre y correo electrónico.
  + Chip verde de “Administrador” si el rol lo indica.
* Opciones funcionales (tarjetas interactivas):
  + “Cambiar contraseña” → abre modal.
  + “Cambiar foto de perfil” → abre galería.
* Botones inferiores:
  + “Ver mis reservas” (context.push('/mis-reservas')).
  + “Volver al mapa”.
  + “Cerrar sesión” (rojo).

B. Usuario invitado

Se presenta un mensaje informativo con botones:

* “Iniciar sesión” → /login.
* “Registrarse” → /register.

Widgets auxiliares

|  |  |
| --- | --- |
| Método | Función |
| \_buildOptionTile() | Genera tarjetas de acción con ícono, título, subtítulo y callback. |
| \_buildGuestView() | Muestra la vista alternativa para usuarios no autenticados. |

* Diseño y experiencia de usuario
  + Fondo claro (#F5FAF5) con tarjetas blancas y sombras suaves.
  + Colores institucionales: verde para acciones principales, azul grisáceo para botones secundarios y rojo para logout.
  + Efectos visuales: bordes redondeados, sombras (BoxShadow) y SnackBar para retroalimentación.
  + Experiencia totalmente adaptativa y responsiva.

### 3.2.7. Página/Screen Mis\_Reservas\_Page

Ubicación: lib/features/reservas/mis\_reservas\_page.dart

* Descripción General:

Esta pantalla muestra al usuario autenticado la lista de sus reservas realizadas dentro del sistema SportReserve Mobile.  
Consume el servicio ReservationService para obtener los datos desde el backend Laravel y los renderiza en forma de tarjetas, mostrando información relevante como nombre de la cancha, fecha, hora, precio y estado de la reserva.

El widget utiliza un FutureBuilder para manejar el estado de la carga de datos (esperando, error, éxito) y actualizar dinámicamente la interfaz según la respuesta recibida.

* Funcionamiento General
  + Cuando el usuario abre la página, se ejecuta el método \_fetchReservas() desde el initState().
  + Este método invoca ReservationService().obtenerMisReservas() que se conecta al backend.
  + Se evalúa la estructura de la respuesta (puede llegar como lista o mapa JSON).
  + El resultado se convierte en una lista dinámica de reservas (List<dynamic>).
  + Finalmente, el FutureBuilder construye la interfaz según los estados:
    - **Cargando:** muestra un CircularProgressIndicator.
    - **Error:** muestra el mensaje del error en texto rojo.
    - **Éxito:** lista todas las reservas del usuario.
* **Estructura del código**

**Declaración del widget: class MisReservasPage extends StatefulWidget { ... }**

Se usa StatefulWidget porque la vista depende de datos que se cargan de manera asíncrona.

Inicialización



Se inicializa el Future que cargará los datos de reservas cuando la vista sea creada.

* Obtención de reservas: Future<List<dynamic>> \_fetchReservas() async { ... }

Este método maneja todos los posibles formatos que el backend puede devolver:

* Si recibe una lista directa, la retorna tal cual.
* Si recibe un mapa JSON, busca los campos reservas, data o reserva y los convierte a lista.
* Si el backend responde con error (success == false), lanza una excepción.
* Si no se encuentra estructura válida, devuelve una lista vacía.

En caso de excepción, lanza un error manejado con mensaje claro:

*throw Exception('Error al cargar las reservas: $e');*

* Construcción de la interfaz

El método build() define la estructura visual:

* **AppBar:** título “Mis Reservas”.
* **FutureBuilder:** controla la carga de datos.
* **ListView.builder:** genera dinámicamente las tarjetas de cada reserva.

Dentro de cada Card se muestran:

* Nombre de la cancha
* Fecha formateada con intl (dd/MM/yyyy)
* Hora de inicio y fin
* Precio en COP
* Estado (confirmada, cancelada, pendiente), visualizado con un Chip de color distinto según el caso:
  + Verde → confirmada
  + Rojo → cancelada
  + Naranja → pendiente



* Consideraciones técnicas
* Se usa FutureBuilder<List<dynamic>> para simplificar la carga asíncrona.
* El método \_fetchReservas() contempla todos los formatos de respuesta posibles desde Laravel.
* El uso de Intl mejora la presentación de fechas al formato local (dd/MM/yyyy).
* El diseño usa Card con bordes redondeados y Chip de color variable para un look moderno y consistente con la app.
* Si no existen reservas, se muestra un mensaje amigable:

*const Center(child: Text('No tienes reservas aún 🕓'));*

* Dependencias utilizadas:
* package:intl/intl.dart → formateo de fechas.
* package:sportreserve\_mobile\_frontend/services/reservation\_service.dart → conexión con el backend Laravel.
* package:flutter/material.dart → widgets base del framework Flutter
* Errores manejados
* Si la respuesta del servidor es incorrecta o vacía → se muestra mensaje “Error al cargar las reservas”.
* Si el usuario no tiene reservas → mensaje “No tienes reservas aún 🕓”.
* Los errores de conexión o parsing se atrapan con try/catch y se muestran visualmente al usuario

## 3.3 Mappers

Ubicación: lib/mappers/cancha\_mapper.dart

* Descripción general

El archivo cancha\_mapper.dart cumple la función de traducir los datos JSON provenientes del backend Laravel hacia el modelo de dominio Cancha utilizado dentro de la aplicación Flutter.  
Su objetivo principal es estandarizar y asegurar que los datos que llegan del servidor se transformen correctamente en un objeto Cancha del frontend, independientemente de cómo vengan estructurados los nombres o tipos de campos en la respuesta.

* Funcionamiento general

Este archivo define la clase CanchaMapper, que contiene un método estático principal:

*static model.Cancha fromJson(Map<String, dynamic> json)*

Este método recibe un Map<String, dynamic> (objeto JSON recibido desde la API) y devuelve una instancia de model.Cancha completamente inicializada.

El mapeo contempla diferentes formatos posibles, validando tipos de datos y valores nulos, para evitar errores en la conversión y garantizar que el modelo contenga información consistente.

* Estructura del código
* Importación del modelo

*import 'package:sportreserve\_mobile\_frontend/models/cancha.dart' as model;*

Se importa el modelo Cancha con un alias model para mantener claridad en el código, especialmente al instanciar el objeto dentro del mapper.

* Clase principal *CanchaMapper 🡪 class CanchaMapper { ... }*

Esta clase no necesita ser instanciada, por lo que su método fromJson es estático.  
El objetivo es permitir el uso directo como CanchaMapper.fromJson(json) desde cualquier parte del proyecto.

* Método estático fromJson()

El método analiza campo por campo el JSON recibido y crea el objeto Cancha.  
Su implementación completa:



* Lógica interna campo por campo

|  |  |  |  |
| --- | --- | --- | --- |
| Campo | Fuente JSON | Transformación | Valor por defecto |
| id | json['id'] | Convierte de String a int si es necesario | 0 |
| nombre | json['nombre'] | Texto directo | 'Cancha sin nombre' |
| tipo | json['tipo'] | Texto directo | 'Desconocido' |
| ubicacion | json['ubicacion'] | Texto directo | 'Ubicación no registrada' |
| capacidad | json['capacidad'] | Entero directo | 0 |
| precioPorCancha | Busca entre varios posibles nombres: precio\_por\_hora, precio, precioPorCancha | Convertido a double | 0.0 |
| latitud | json['latitud'] | Convierte de String a double si es necesario | 0.0 |
| longitud | json['longitud'] | Convierte de String a double si es necesario | 0.0 |
| disponibilidad | json['disponibilidad'] | Evalúa si el valor es true, 1 o '1' | false |
| imagen | json['imagen'] | Texto directo | '' |
| descripción | json['descripcion'] | Texto directo | '' |
| servicios | json['servicios'] | Texto directo | '' |

* Funcionalidad clave

El mapper se asegura de que, sin importar cómo el backend devuelva los datos, el modelo Flutter mantenga una estructura uniforme y segura.  
Por ejemplo:

* Si el backend devuelve los precios como string ("20000"), el mapper los convierte a double.
* Si el campo disponibilidad llega como '1' o 1, se interpreta correctamente como true.
* Si algún campo no existe, se define un valor por defecto legible para evitar errores de renderización
* Dependencias
* models/cancha.dart: modelo de dominio que define la clase Cancha con sus atributos.
* No utiliza paquetes externos; toda la conversión se maneja con funciones nativas de Dart (tryParse, toDouble, validaciones de tipo).

## 3.4 Archivo app\_database.dart

Ubicación: lib/db/app\_database.dart

Propósito de app\_database.dart:

Implementar la base de datos local de la aplicación móvil SportReserve utilizando Drift (ORM para SQLite).  
Esta base sirve como caché offline para los datos de las canchas deportivas obtenidos desde el backend Laravel.

* + 1. Descripción general

Este archivo define la estructura local de datos mediante el ORM drift, que permite a la app almacenar, consultar y sincronizar información incluso sin conexión a internet.  
El archivo contiene tres partes principales:

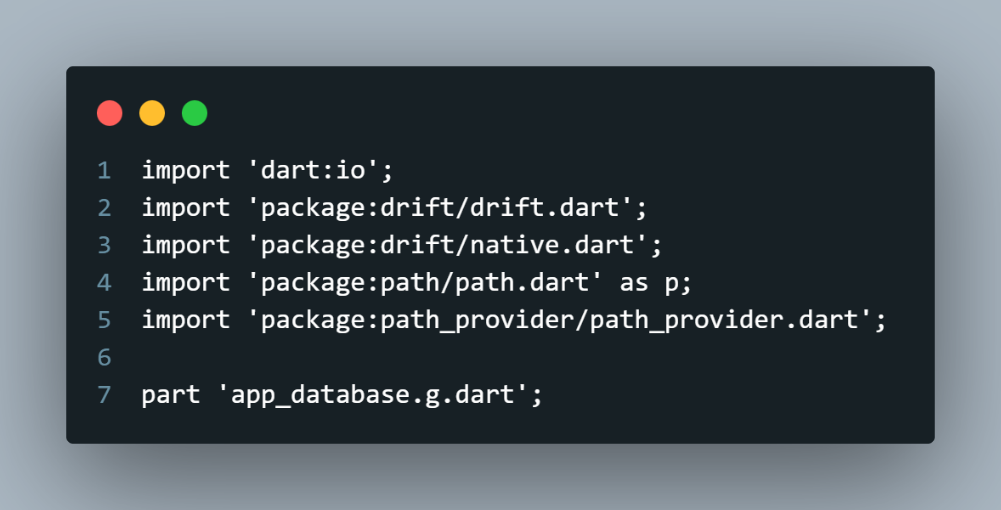
1. Definición de la tabla Canchas
2. Clase AppDatabase (gestor principal de la base)
3. Función \_openConnection() para inicializar la base local

El archivo generado automáticamente app\_database.g.dart (por build\_runner) complementa este archivo con el código necesario para que Drift funcione correctamente.

* + 1. Dependencias principales

|  |  |
| --- | --- |
| Paquete | Función |
| drift | ORM para definir tablas y realizar operaciones SQL en Dart. |
| drift/native.dart | Permite crear una base SQLite nativa en Flutter. |
| path | Construye rutas de archivos en el sistema operativo. |
| path\_provider | Permite obtener directorios del dispositivo (documentos, cache, etc.). |

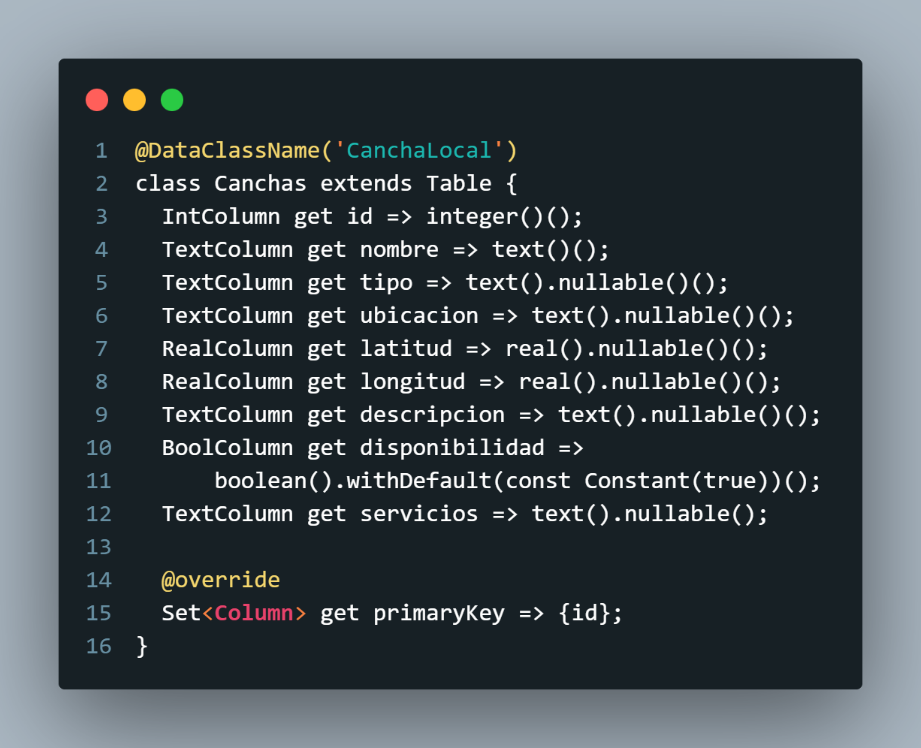
* + 1. Estructura del código:
* Importaciones y configuración base:



* Se importan los paquetes necesarios para crear y manejar la base local.
* La línea part 'app\_database.g.dart'; indica que este archivo será complementado automáticamente por Drift al ejecutar:

*flutter pub run build\_runner build --delete-conflicting-outputs*

* + 1. Tablas principales
  + Tabla canchas

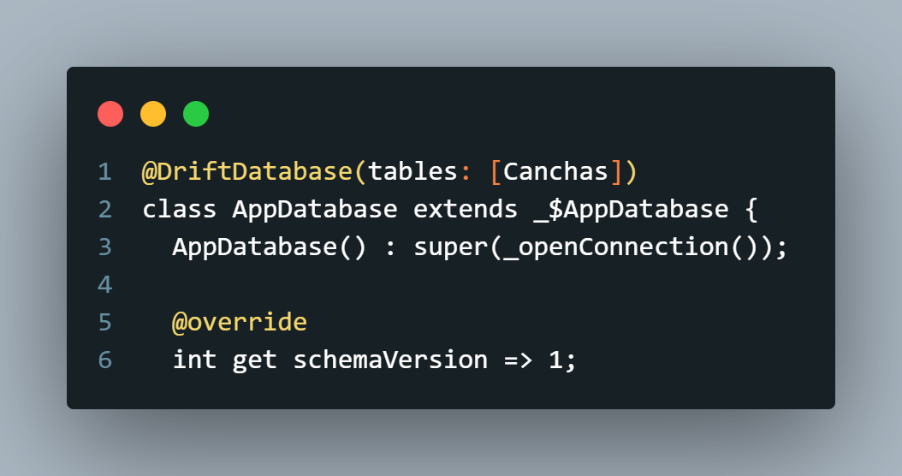


* Descripción:  
  Esta clase representa la tabla Canchas en SQLite. Cada campo equivale a una columna de la tabla:

|  |  |  |
| --- | --- | --- |
| Columna | Tipo | Descripción |
| id | int | Identificador único de la cancha (clave primaria). |
| nombre | text | Nombre de la cancha. |
| tipo | text? | Tipo de superficie o deporte asociado. |
| ubicacion | text? | Dirección o ubicación textual. |
| latitud / longitud | real? | Coordenadas geográficas para mostrar en el mapa. |
| descripcion | text? | Texto descriptivo de la cancha. |
| disponibilidad | bool | Estado de disponibilidad (true = activa). |
| servicios | text? | Lista o descripción de servicios adicionales. |

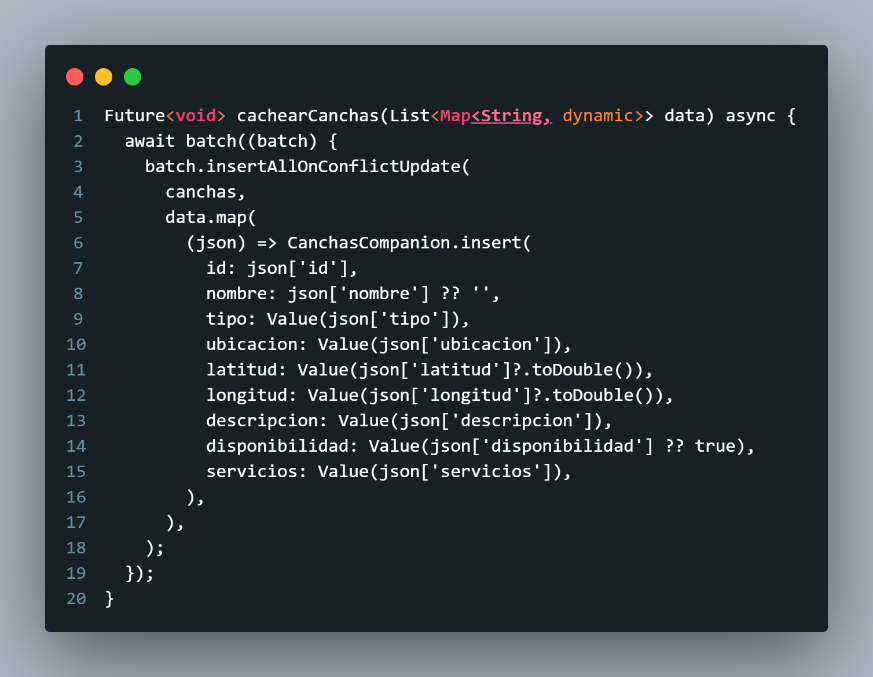
El decorador @DataClassName('CanchaLocal') genera una clase modelo llamada CanchaLocal para interactuar con esta tabla desde Dart.

* + 1. Clase principal *AppDatabase*



* @DriftDatabase(tables: [Canchas]) indica qué tablas maneja la base.
* La clase hereda de \_$AppDatabase (generada automáticamente).
* schemaVersion permite manejar migraciones de la base (por ahora 1).
  + 1. Métodos principales
       1. *cachearCanchas(List<Map<String, dynamic>> data)*

Guarda o actualiza una lista de canchas obtenidas del backend



* Qué hace:
  + Inserta todos los registros en la tabla canchas.
  + Si un registro ya existe (por mismo id), lo actualiza automáticamente (insertAllOnConflictUpdate).
  + Se usa batch() para procesar múltiples inserciones en una sola transacción.
    - 1. ObtenerCanchas()

Devuelve todas las canchas almacenadas localmente.

*Future<List<CanchaLocal>> obtenerCanchas() => select(canchas).get();*

* **Qué hace:**
  + Ejecuta una consulta SELECT \* FROM canchas.
  + Retorna una lista de objetos CanchaLocal.
    - 1. limpiarCache()

Elimina todas las canchas almacenadas localmente.

*Future<void> limpiarCache() => delete(canchas).go();*

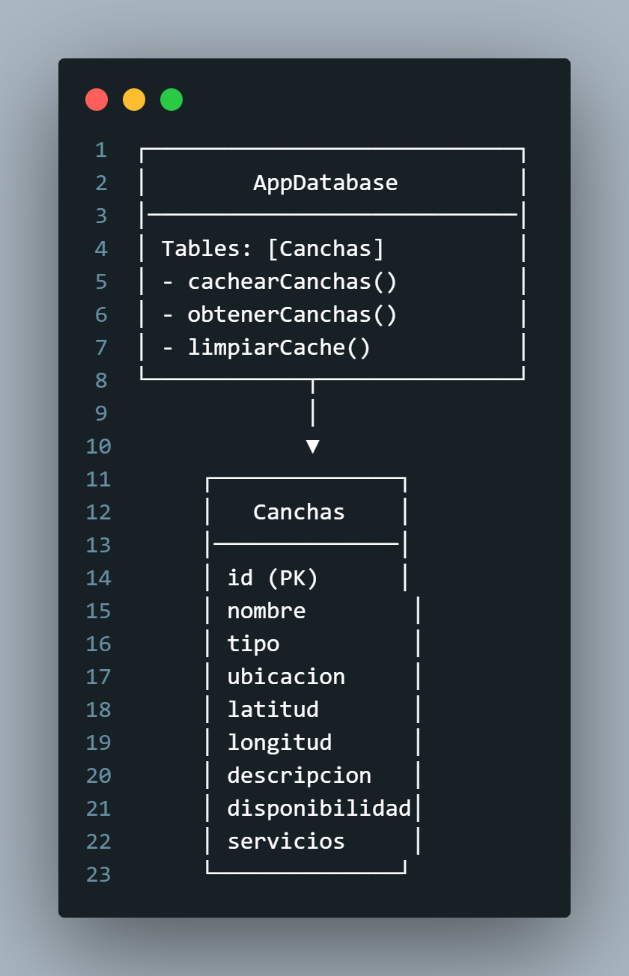
* Qué hace:
  + Ejecuta un DELETE FROM canchas.
  + Se usa normalmente al cerrar sesión o al refrescar la base por completo.
    1. función \_openConnection()



* **Qué hace:**
  + Obtiene el directorio de documentos de la app.
  + Crea o abre el archivo app\_cache.db dentro de esa carpeta.
  + Retorna una conexión NativeDatabase, usada por Drift.

Se usa LazyDatabase para que la base se cree **solo cuando realmente se necesite**, optimizando recursos.

Esquema general simplificado



* 1. Modelo cancha\_meta.dart:

Ubicación: lib/models/cancha\_meta.dart

Propósito: Definir la información visual y de contexto asociada a cada cancha, incluyendo íconos, colores, tipo de superficie y estado de disponibilidad.  
Esta clase complementa el modelo de datos principal (Cancha) proporcionando metadatos necesarios para la presentación en la interfaz de usuario.

* + 1. Descripción general

El archivo cancha\_meta.dart contiene la clase CanchaMeta, la cual no representa datos del backend, sino información auxiliar para renderizar canchas en la interfaz de Flutter de forma más atractiva y semántica.

Se usa principalmente en las vistas:

* cancha\_detail\_page.dart
* mapa\_canchas\_page.dart
* canchas\_page.dart

y permite, por ejemplo:

* Mostrar el ícono correcto (⚽ 🏀 🏐 🎾).
* Cambiar colores según el deporte.
* Indicar si la cancha está disponible.
* Mostrar etiquetas descriptivas como “Césped sintético”, “Parqué”, etc.
  + 1. Dependencias principales

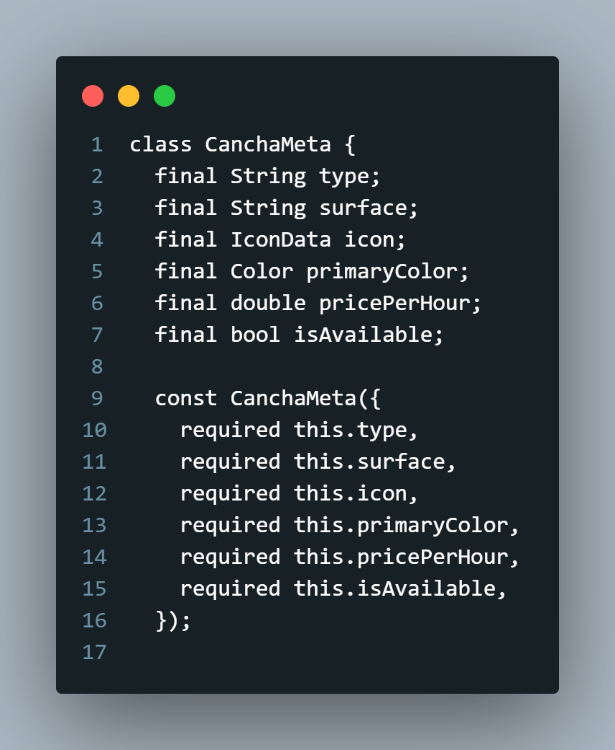
|  |  |
| --- | --- |
| Paquete | Uso |
| flutter/material.dart | Define íconos, colores y widgets visuales. |
| models/cancha.dart | Modelo base de la cancha con sus atributos (nombre, tipo, ubicación, etc.). |

* + 1. Estructura del archivo
       1. Importaciones



Permiten usar los íconos (Icons.sports\_soccer, Icons.sports\_tennis, etc.) y el modelo de datos Cancha que proviene del backend o del cache local.

* + - 1. Definición de la clase CanchaMeta

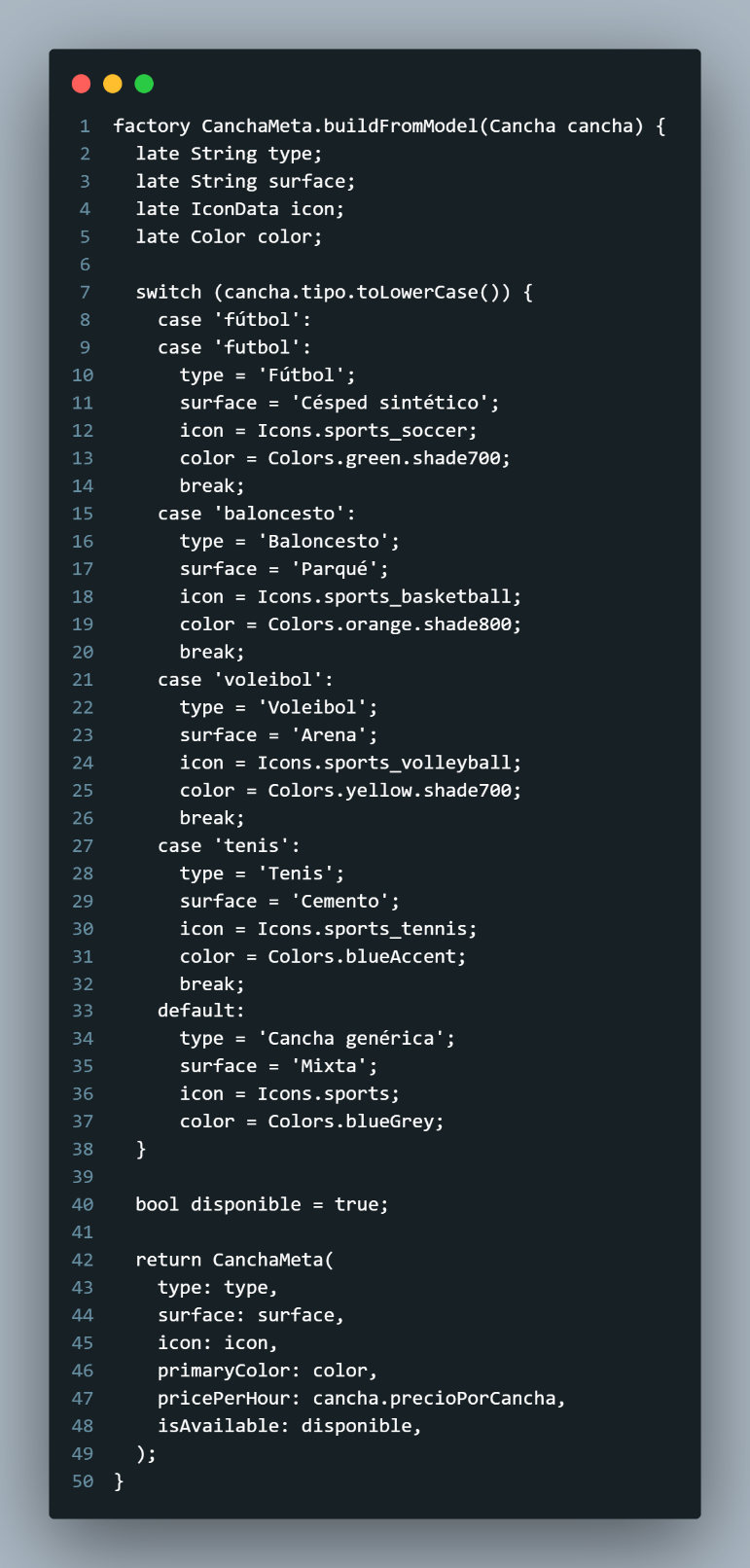


Descripción:

* + Representa los metadatos visuales y de contexto de una cancha.  
    No se guarda en base de datos ni se recibe desde el backend, sino que se construye dinámicamente según los atributos del modelo Cancha.

|  |  |  |
| --- | --- | --- |
| Propiedad | Tipo | Descripción |
| type | String | Nombre del deporte asociado a la cancha. |
| surface | String | Tipo de superficie (césped, parqué, cemento, arena, etc.). |
| icon | IconData | Ícono representativo del deporte. |
| primaryColor | Color | Color principal para usar en UI (fichas, tarjetas, íconos). |
| pricePerHour | double | Precio por hora, tomado del modelo Cancha. |
| isAvailable | bool | Estado de disponibilidad (true = disponible). |

* + - 1. Constructor por fábrica *buildFromModel*



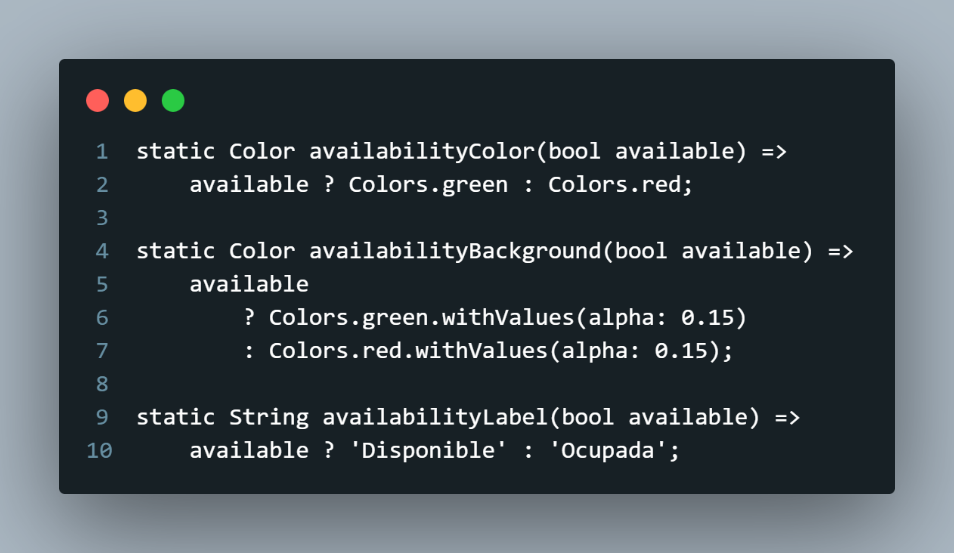
* Qué hace:
* Analiza el campo tipo del modelo Cancha.
* Asigna íconos y colores distintos según el deporte.
* Define texto de superficie según el tipo.
* Si el backend no define disponibilidad (disponible o status), la marca como **disponible por defecto**.
* Retorna una nueva instancia de CanchaMeta.
  + - 1. Alias estático fromCancha (Alias que mejora la legibilidad cuando se usa en vistas o servicios.)

*static CanchaMeta fromCancha(Cancha cancha) => CanchaMeta.buildFromModel(cancha);*

ejemplo:

*final meta = CanchaMeta.fromCancha(cancha);*

* + - 1. Métodos auxiliares visuales



Proveen estilos visuales estandarizados según la disponibilidad de la cancha

|  |  |  |
| --- | --- | --- |
| Método | Descripción | Ejemplo |
| availabilityColor | Retorna el color de texto según disponibilidad. | Verde / Rojo |
| availabilityBackground | Retorna el color de fondo con transparencia. | Verde claro / Rojo claro |
| availabilityLabel | Texto “Disponible” o “Ocupada”. | Se usa en chips o tarjetas. |

* 1. Modelo cancha.dart

Ubicación: lib/models/cancha.dart

Propósito: Definir el modelo de datos principal de una cancha deportiva, que se utiliza tanto para la comunicación con el backend Laravel (API REST) como para la persistencia local mediante Drift (en app\_database.dart).

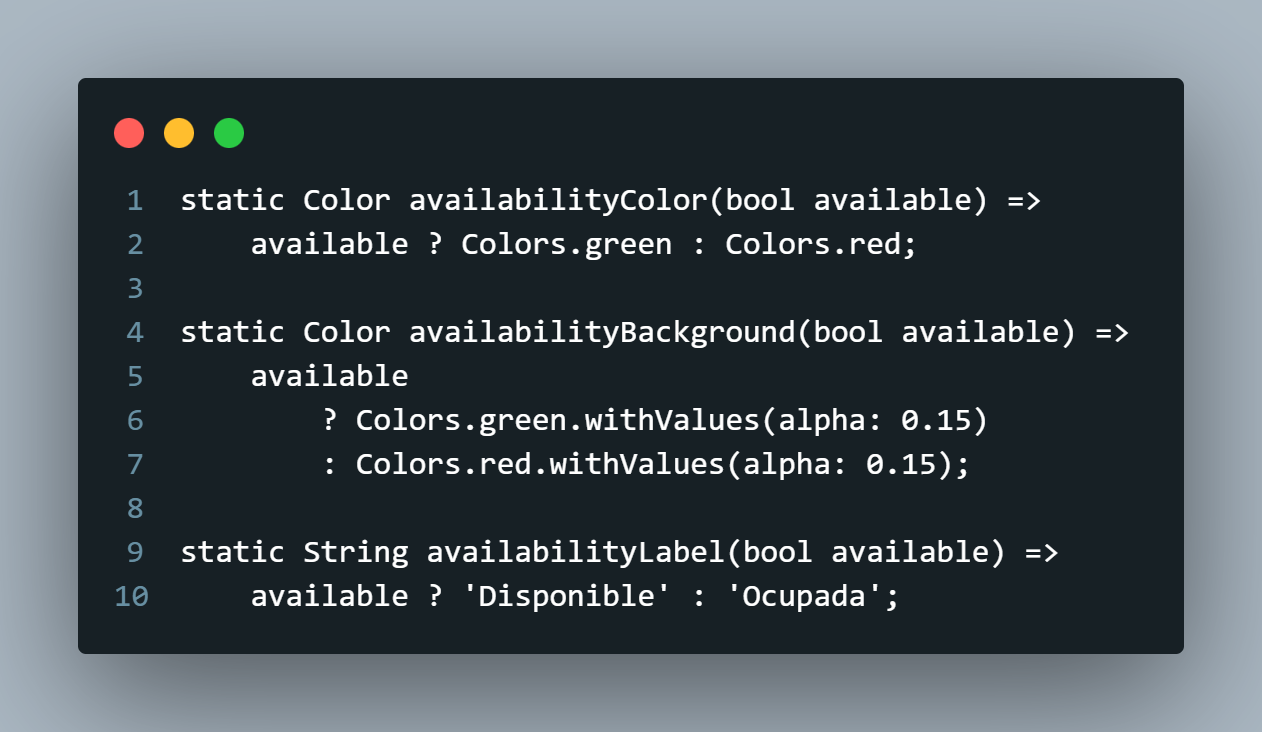
* + 1. Descripción general:

El modelo Cancha representa toda la información estructurada de una cancha dentro del sistema SportReserve.  
Incluye:

* Identificación (ID y nombre)
* Información geográfica (latitud, longitud, ubicación)
* Características (tipo, servicios, descripción, capacidad)
* Estado (disponibilidad)
* Precio y recursos gráficos (imagen)

Este modelo es esencial para sincronizar los datos entre el backend Laravel y la base local (Drift), además de ser la fuente de datos de las vistas del mapa, listado y detalle de canchas.

* + 1. Estructura general



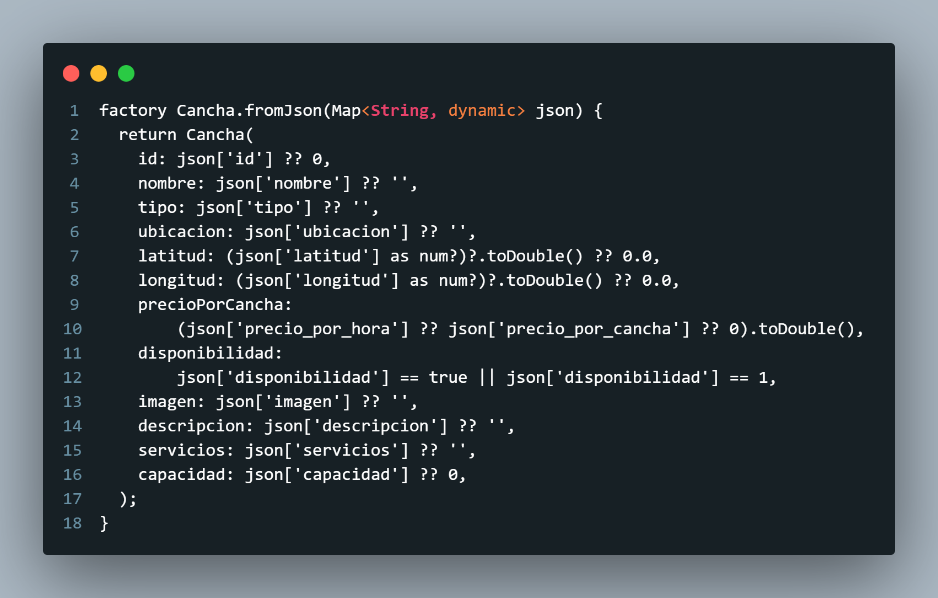
* **Descripción:**  
  Representa un registro completo de una cancha.  
  Los atributos son **inmutables** (final) y deben ser inicializados mediante el constructor.
  + - 1. Campos del modelo cancha.dart:

|  |  |  |
| --- | --- | --- |
| Campo | Tipo | Descripción |
| id | int | Identificador único de la cancha. |
| nombre | String | Nombre comercial o descriptivo de la cancha. |
| tipo | String | Tipo de deporte asociado (Fútbol, Baloncesto, etc.). |
| ubicacion | String | Dirección o texto descriptivo de la ubicación. |
| latitud / longitud | double | Coordenadas geográficas exactas para mostrar en el mapa. |
| precioPorCancha | double | Precio de reserva por hora o por turno. |
| disponibilidad | bool | Estado actual (true = disponible). |
| imagen | String | Ruta o URL de la imagen asociada. |
| descripcion | String | Texto explicativo o promocional. |
| servicios | String | Lista o descripción de servicios adicionales (baños, iluminación, etc.). |
| capacidad | int | Número máximo de jugadores o personas permitidas. |

* + 1. Constructor principal



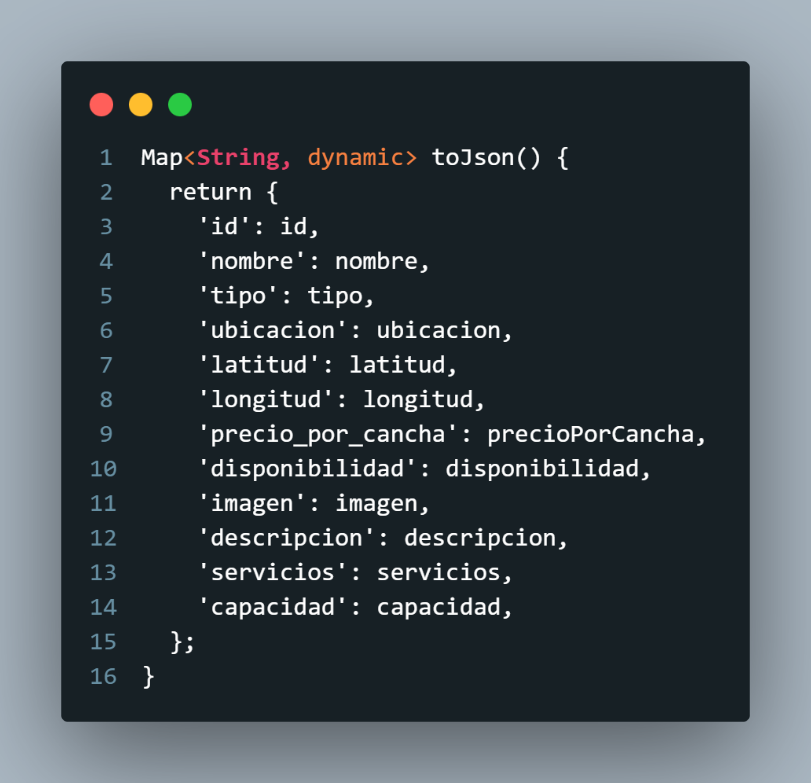
* + - 1. Características:
* Todos los campos son obligatorios (required).
* Esto garantiza consistencia de datos y evita valores nulos dentro del sistema.
  + 1. Conversiones
       1. Conversión desde JSON (fromJson)



* **Qué hace la conversión desde JSON:**
  + Permite construir un objeto Cancha a partir de una respuesta JSON del backend.
  + Soporta diferentes nombres de campos del backend (por ejemplo, precio\_por\_hora o precio\_por\_cancha).
  + Convierte automáticamente los valores numéricos a double.
  + Interpreta disponibilidad desde valores true, 1 o false.
* Ejemplo de conversión desde JSON:



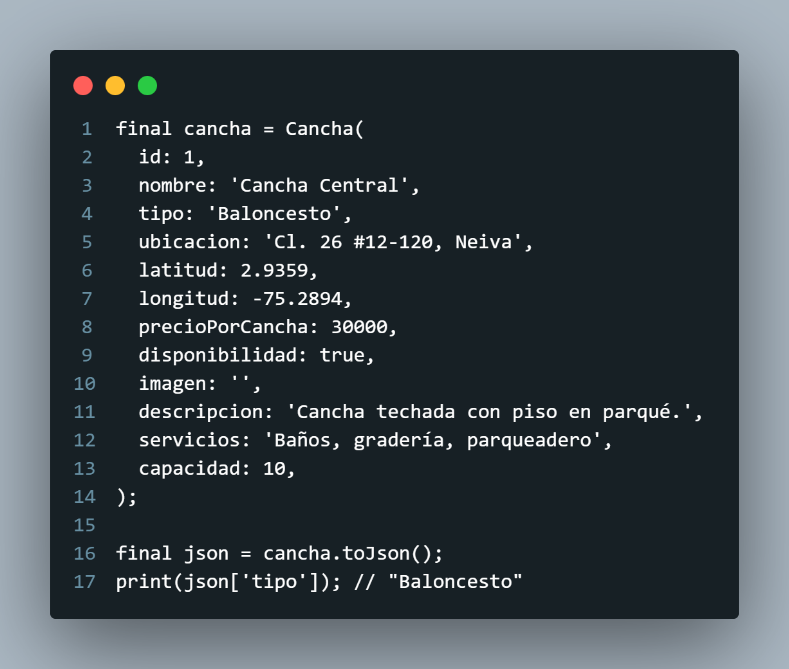
* + - 1. Conversión hacia JSON (toJson)



Qué hace la conversión hacia JSON:

* Convierte un objeto Cancha a un mapa JSON.
* Se utiliza para enviar datos al backend (por ejemplo, al registrar o actualizar una cancha)

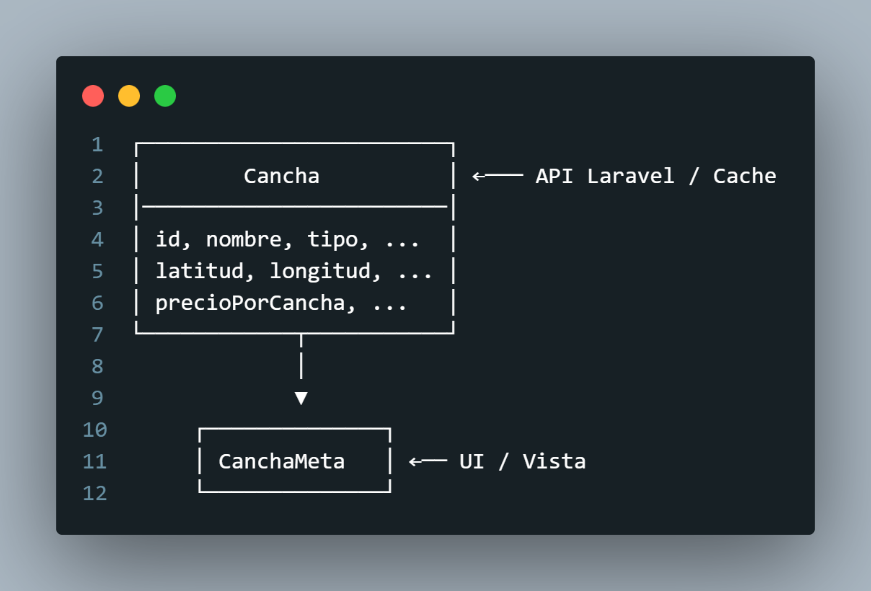
Ejemplo de la conversión hacia JSON:



Integración con otros módulos:

|  |  |
| --- | --- |
| Módulo | Uso |
| CanchaMapper | Convierte entre Map<String, dynamic> y Cancha. |
| CanchaMeta | Genera los atributos visuales e íconos según el tipo de cancha. |
| CanchaService | Descarga datos desde el backend y los sincroniza con el cache local. |
| AppDatabase | Guarda una versión simplificada (CanchaLocal) para acceso offline. |

Relación entre modelos:



* 1. Modelo reservation\_slot.dart

Ubicación: *lib/models/reservation\_slot.dart*

Proposito: Definir la estructura de un **bloque horario (slot)** disponible o reservado dentro del sistema de reservas de SportReserve Mobile.

Este modelo se utiliza para representar la disponibilidad de una cancha en fechas y horas específicas, tanto en la vista de selección de horario como en la comunicación con el backend Laravel.

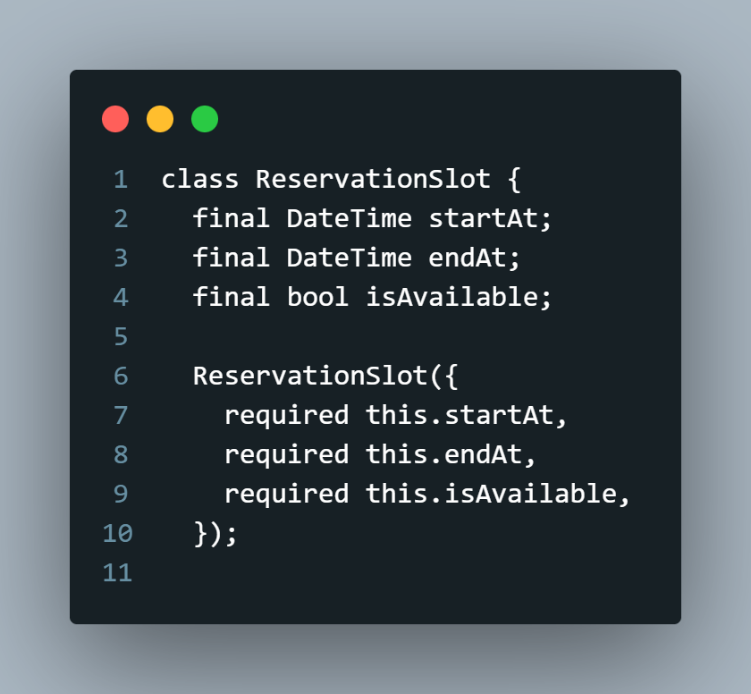
* + 1. Descripción general

La clase ReservationSlot representa un **intervalo de tiempo** dentro del calendario de reservas.  
Cada objeto indica:

* Cuando inicia la franja (startAt)
* Cuando termina (endAt)
* Si está disponible o no (isAvailable)

Se utiliza en módulos como:

* Reserva de canchas
* Calendario de disponibilidad
* Confirmación de reservas
  + 1. Estructura general del modelo reservation\_slot.dart



|  |  |  |
| --- | --- | --- |
| Campo | Tipo | Descripción |
| startAt | DateTime | Fecha y hora de inicio del bloque horario. |
| endAt | DateTime | Fecha y hora de finalización del bloque. |
| isAvailable | bool | Indica si el horario está disponible (true) o ya reservado (false). |

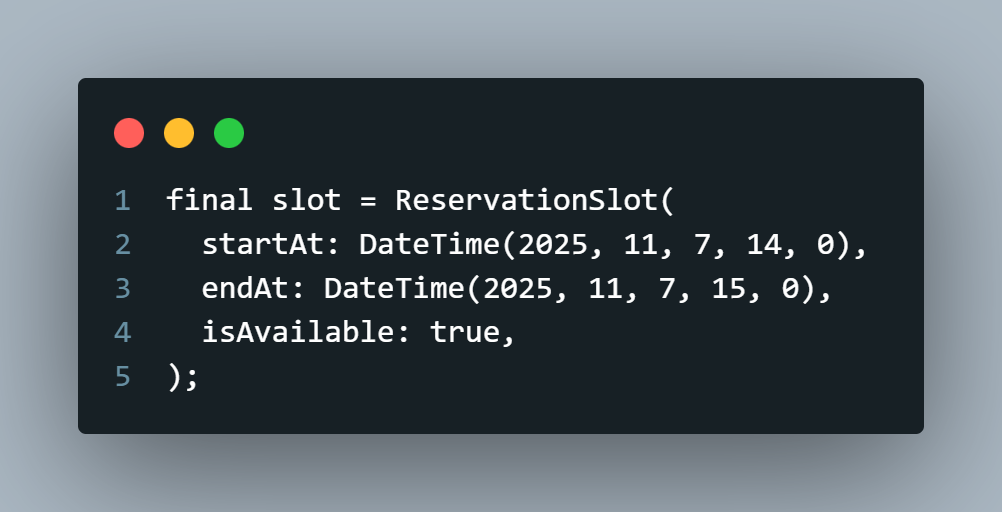
* + 1. Constructor principal



Qué haceel constructor principal del modelo reservation\_slot.dart**:**

* Crea una instancia inmutable (final) con los tres campos requeridos.
* Se usa cuando los datos son generados manualmente o desde la aplicación.

Ejemplo ReservationSlot:



* 1. Modelo reservation.dart

Ubicación: lib/models/reservation.dart

Propósito: Representar una reserva deportiva proveniente del backend Laravel, incluyendo su información temporal, estado, usuario y cancha asociada.  
El modelo se usa tanto para mostrar reservas activas o pasadas en la interfaz como para enviar datos al backend al crear una nueva reserva.

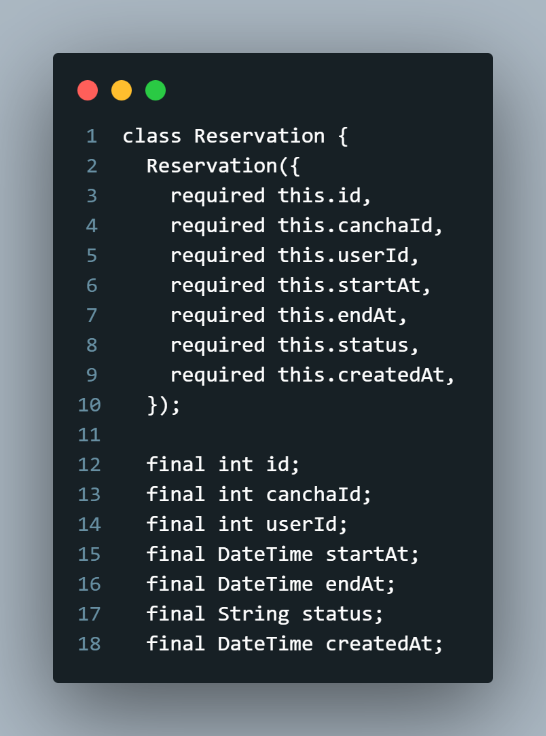
* + 1. Descripción general

La clase Reservation encapsula los datos de una reserva registrada en el sistema.  
Incluye:

* Identificadores (id, canchaId, userId)
* Fechas (startAt, endAt, createdAt)
* Estado (status)
* Propiedades derivadas (como la duración total)

Este modelo es bidireccional:

* Entrada: convierte respuestas JSON del backend a objetos Dart (fromJson)
* Salida: convierte objetos Dart a mapas JSON (toJson) para enviarlos a Laravel
  + 1. Estructura de la clase



|  |  |  |
| --- | --- | --- |
| Campo | Tipo | Descripción |
| id | int | Identificador único de la reserva. |
| canchaId | int | Identificador de la cancha reservada. |
| userId | int | Identificador del usuario que realizó la reserva. |
| startAt | DateTime | Fecha y hora de inicio de la reserva. |
| endAt | DateTime | Fecha y hora de finalización de la reserva. |
| status | String | Estado actual de la reserva (ej. pending, confirmed, cancelled). |
| createdAt | DateTime | Fecha y hora en que se creó el registro en el sistema. |

* + 1. Propiedad calculada Duration

*Duration get duration => endAt.difference(startAt);*

Que hace la propiedad Duration:

* Calcula automáticamente la duración total de la reserva. Ideal para validaciones o visualización en la interfaz.

Ejemplo de la propiedad Duration:

*print(reserva.duration.inHours); // 2 (si la reserva dura 2 horas)*

* + 1. Validaciones internas

El modelo está diseñado para detectar errores de formato temprano, mediante excepciones controladas (FormatException).  
Esto evita que datos inválidos lleguen a la capa de presentación o persistencia.

* + 1. Integración con otros componentes

|  |  |
| --- | --- |
| Componente | Uso |
| Cancha | Identifica la cancha asociada a la reserva (canchaId). |
| AuthService | Asocia la reserva al usuario autenticado (userId). |
| ReservationService / ReservationController | Crea, actualiza o consulta las reservas en el backend. |
| MisReservasPage | Muestra las reservas activas o pasadas del usuario. |

* 1. Modelo *user\_profile\_laravel.dart*:

Ubicación: lib/models/user\_profile\_laravel.dart

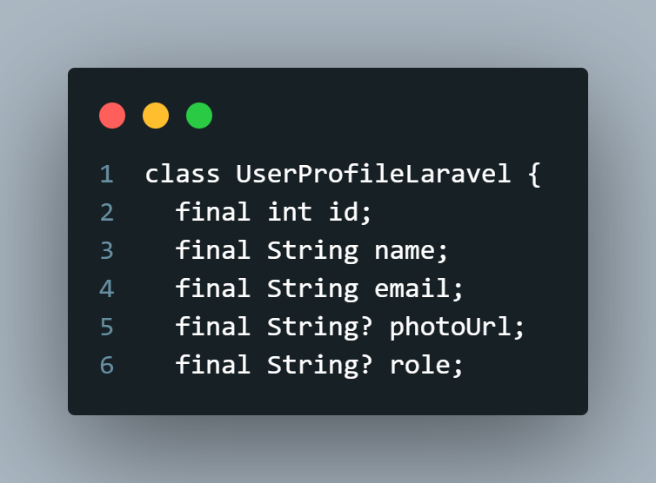
Propósito: Representar el perfil del usuario autenticado proveniente del backend Laravel, incluyendo su identificación, nombre, correo, rol y foto de perfil.  
Este modelo es utilizado principalmente por el servicio de autenticación (AuthService) y las vistas de perfil de usuario.

* + 1. Descripción general:

UserProfileLaravel encapsula los datos del usuario autenticado y mantiene compatibilidad directa con las respuestas JSON devueltas por los endpoints de Laravel (/api/login, /api/profile, /api/register, etc.).

Además, permite:

* Almacenar información del rol del usuario (admin, monitor, user, etc.).
* Mantener una foto de perfil (photoUrl).
* Crear copias modificadas del usuario mediante copyWith.
  + 1. Estructura de la clase



|  |  |  |
| --- | --- | --- |
| Campo | Tipo | Descripción |
| id | int | Identificador único del usuario en Laravel. |
| name | String | Nombre completo del usuario. |
| email | String | Correo electrónico asociado a la cuenta. |
| photoUrl | String? | URL o ruta local de la foto de perfil (opcional). |
| role | String? | Rol asignado dentro del sistema (por ejemplo, Administrador, Monitor, Usuario). |

* + 1. Constructor principal

*UserProfileLaravel({*

*required this.id,*

*required this.name,*

*required this.email,*

*this.photoUrl,*

*this.role,*

*});*

Características:

* Crea un objeto UserProfileLaravel con valores obligatorios (id, name, email).
* Los campos opcionales (photoUrl, role) pueden ser nulos si el backend no los envía.
* Diseñado para ser inmutable y seguro frente a datos incompleto
  + 1. Integración con otros módulos

|  |  |
| --- | --- |
| Módulo | Función |
| AuthService | Utiliza UserProfileLaravel para gestionar sesión, perfil y logout. |
| user\_profile\_page.dart | Muestra los datos del usuario autenticado. |
| app\_router.dart | Redirige según el rol (/monitores, /admin, /mapa). |
| Laravel Backend | Retorna el perfil mediante /api/profile o /api/user. |

* 1. Reserva\_repository.dart

Ubicación: lib/repositories/reserva\_repository.dart

Propósito: Centralizar el acceso a los datos relacionados con reservas desde la base de datos local (Drift) y —en el futuro— desde el backend Laravel.  
Este repositorio sirve como intermediario entre la capa de datos (AppDatabase) y la capa de servicios (ReservationService, CanchaService).

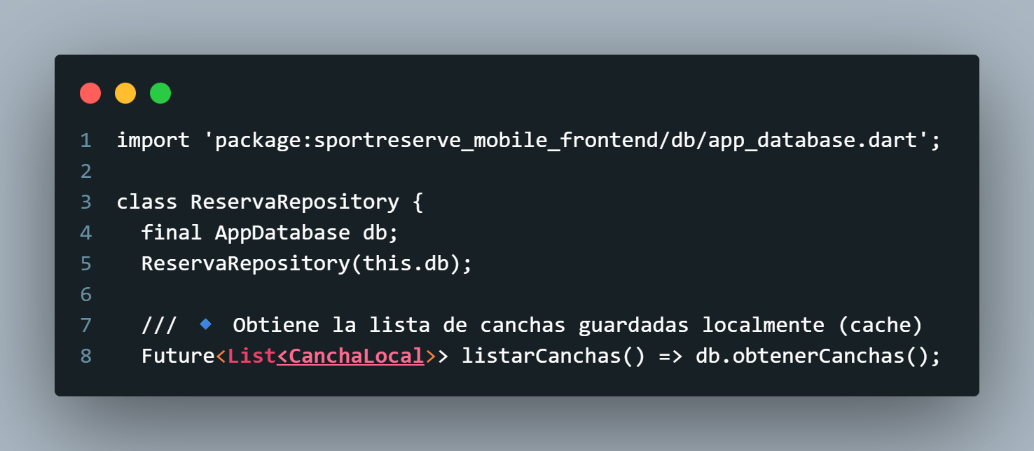
* + 1. Descripción general

El ReservaRepository tiene como objetivo aislar la lógica de obtención de datos locales, de forma que:

* La app pueda acceder a los datos de las canchas o reservas sin preocuparse por la fuente (local o remota).
* En el futuro, sea posible combinar datos offline (Drift) y online (Laravel API) fácilmente.

Por ahora, su función principal es consultar las canchas almacenadas en la base local (app\_cache.db) como cache.

* + 1. Código base:



* + 1. Estructura de la clase

|  |  |
| --- | --- |
| Elemento | Descripción |
| final AppDatabase db | Instancia de la base de datos local Drift (app\_cache.db). |
| ReservaRepository(this.db) | Constructor que recibe la base local desde donde se accederá a los datos. |
| listarCanchas() | Método que retorna todas las canchas almacenadas en la base local. |

* + 1. Posible expansión futura:

Aunque ahora solo maneja canchas, este repositorio puede evolucionar para incluir más funcionalidades:

|  |  |
| --- | --- |
| Método futuro | Descripción |
| Future<List<Reservation>> listarReservas() | Obtener todas las reservas almacenadas localmente o desde el backend. |
| Future<void> guardarReserva(Reservation reserva) | Guardar una reserva confirmada en cache. |
| Future<void> limpiarReservas() | Borrar todas las reservas locales. |
| Future<CanchaLocal?> buscarCanchaPorId(int id) | Buscar una cancha específica en la base local. |

* 1. Servicio app\_initializer.dart

Ubicación: lib/services/app\_initializer.dart

Propósito: Gestionar la inicialización global de los servicios base de la aplicación móvil SportReserve, garantizando que todos los módulos críticos (autenticación y notificaciones) estén configurados correctamente antes de que el usuario interactúe con la interfaz.

* + 1. Descripción general

El archivo app\_initializer.dart define la clase estática AppInitializer, responsable de preparar los servicios esenciales al iniciar la aplicación.  
Su objetivo es realizar tareas iniciales como:

* Validar el estado de autenticación del usuario.
* Inicializar el servicio de notificaciones locales o push.
* Controlar que la inicialización se realice una sola vez durante el ciclo de vida de la app.

Esta clase actúa como un punto de arranque centralizado para configurar el entorno base del sistema.

* + 1. Dependencias principales

|  |  |
| --- | --- |
| Dependencia | Función |
| auth\_service.dart | Gestiona la autenticación del usuario y el manejo del perfil. |
| notification\_service.dart | Inicializa el sistema de notificaciones. |
| flutter/foundation.dart | Proporciona la constante kDebugMode para control de logs y errores en modo desarrollo. |

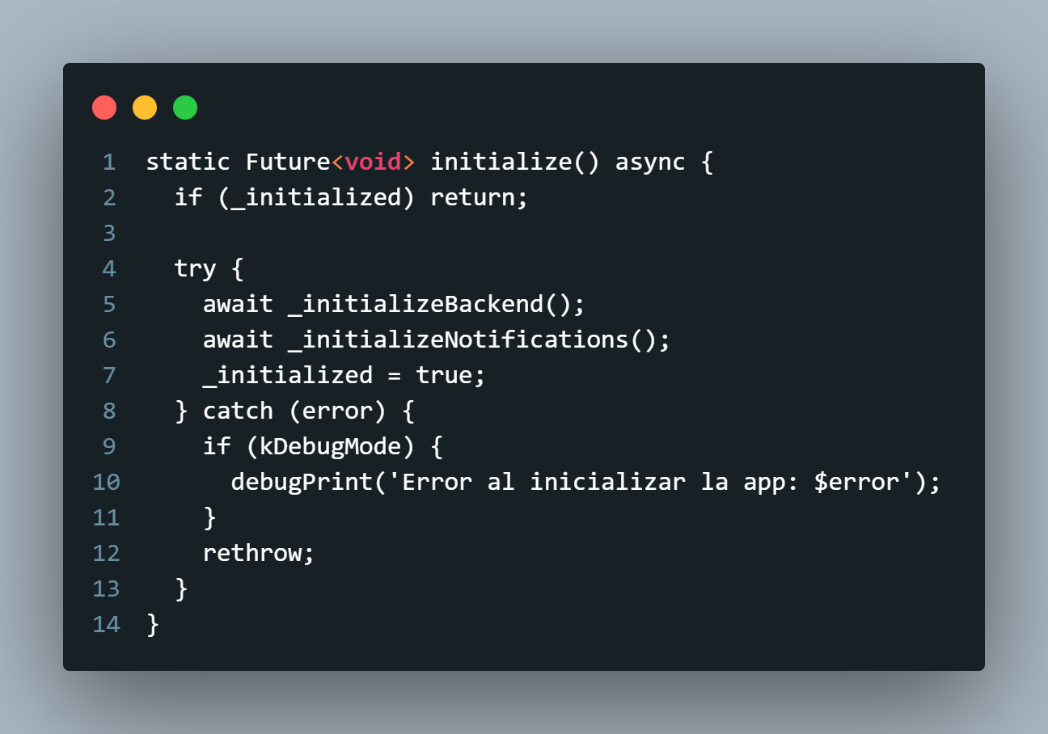
* + 1. Estructura general



* **Clase AppInitializer**
  + La clase AppInitializer utiliza un **constructor privado** (const AppInitializer.\_()) para evitar instanciaciones directas, permitiendo que sus métodos se utilicen únicamente de forma estática.

|  |  |
| --- | --- |
| Elemento | Descripción |
| \_initialized | Variable estática que evita múltiples inicializaciones. |
| initialize() | Método principal que inicializa la app una única vez. |
| \_initializeBackend() | Método auxiliar que carga el estado del usuario autenticado. |
| \_initializeNotifications() | Método auxiliar que configura el servicio de notificaciones. |

* + 1. Método principal initialize()



Función

* Verifica si la aplicación ya ha sido inicializada.
* Si no, ejecuta la configuración del backend y las notificaciones.
* En caso de error, captura la excepción y muestra un log en modo debug.

Comportamiento

* Solo se ejecuta una vez por ciclo de ejecución.
* Mejora la estabilidad de la app al garantizar que los servicios críticos estén listos antes de iniciar.
  1. Servicio auth\_service.dart

Ubicación: lib/services/auth\_service.dart

Propósito: Gestionar el proceso de autenticación de usuarios con el backend Laravel, incluyendo inicio y cierre de sesión, registro, actualización de perfil, persistencia del token de acceso y sincronización del perfil de usuario en almacenamiento local.

* + 1. Descripción general

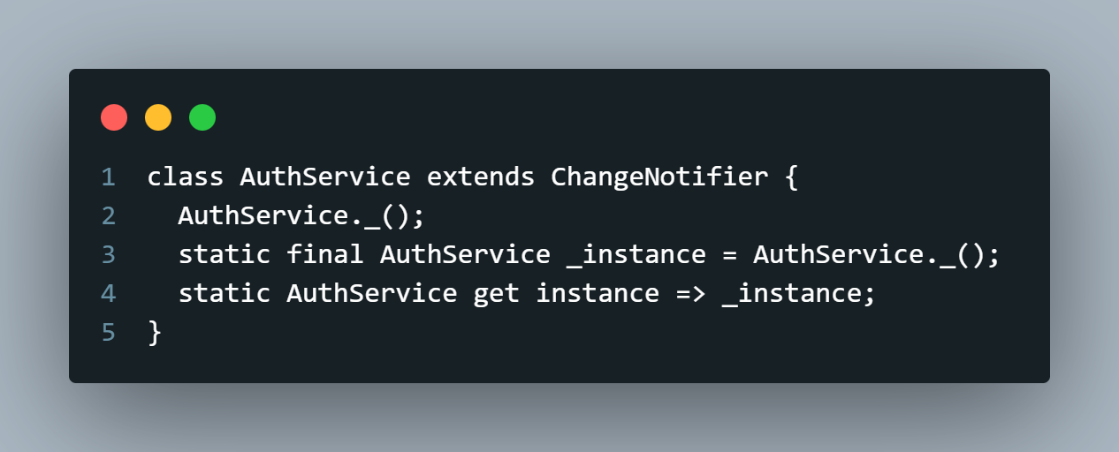
El servicio AuthService constituye el núcleo del sistema de autenticación dentro de SportReserve Mobile.  
Su principal objetivo es administrar de forma centralizada el ciclo de vida de sesión del usuario, garantizando una comunicación segura con la API de Laravel mediante tokens tipo Bearer.

Además, este servicio implementa un mecanismo de persistencia local utilizando SharedPreferences, lo que permite conservar la sesión activa incluso después de cerrar la aplicación.

* + 1. Dependencias principales

|  |  |
| --- | --- |
| Paquete | Uso principal |
| http | Comunicación HTTP con el backend Laravel. |
| shared\_preferences | Almacenamiento local del token y perfil del usuario. |
| flutter/foundation.dart | Depuración y control de logs mediante debugPrint. |
| user\_profile\_laravel.dart | Modelo que representa el perfil autenticado. |

* + 1. Arquitectura general:



La clase AuthService implementa el patrón **Singleton**, lo que garantiza una única instancia accesible desde cualquier parte de la aplicación.

* + 1. Atributos y constantes

|  |  |  |
| --- | --- | --- |
| Campo | Tipo | Descripción |
| \_baseUrl | String | Dirección base del backend Laravel (http://10.0.2.2:8000/api). |
| \_tokenKey | String | Clave de almacenamiento para el token en SharedPreferences. |
| \_userKey | String | Clave de almacenamiento para el perfil del usuario. |
| \_token | String? | Token JWT actual en memoria. |
| \_cachedProfile | UserProfileLaravel? | Perfil de usuario autenticado en memoria. |

* + 1. Propiedades públicas

|  |  |
| --- | --- |
| Propiedad | Descripción |
| token | Devuelve el token actual. |
| currentUser | Retorna el perfil autenticado en memoria. |
| isLoggedIn | Indica si hay una sesión activa (true si existe token). |

* + 1. Métodos principales

\_uri(String path)

Construye la URI completa de la API a partir de una ruta relativa. Evita errores por dobles barras (//) al concatenar la base.

Uri \_uri(String path) {

return Uri.parse('$\_baseUrl${path.startsWith('/') ? path : '/$path'}');

}

* + - 1. \_ensureSessionLoaded()

Carga el token y el perfil del usuario desde SharedPreferences si no están disponibles en memoria. Se ejecuta de forma implícita antes de cualquier llamada autenticada.

* + - 1. \_persistToken() Y \_persistUser()

Guardan en almacenamiento persistente tanto el token de autenticación como el perfil de usuario. Estos métodos garantizan que la sesión se mantenga activa entre reinicios de la aplicación.

* + - 1. clearSession()

Elimina toda la información de sesión local, incluyendo token y datos de usuario.  
Además, notifica a los listeners que la sesión ha cambiado.

* + 1. Autenticación con el backend
       1. register()

Permite registrar un nuevo usuario en el backend.

Future<UserProfileLaravel> register({

required String name,

required String email,

required String password,

})

* Envía los datos a /api/register.
* Si la respuesta es 201, crea y retorna un objeto UserProfileLaravel.
* Maneja errores de validación (422) devolviendo mensajes específicos.
  + - 1. Login()

Inicia sesión con correo y contraseña

Future<bool> login({

required String email,

required String password,

})

* Envía la solicitud a /api/login.
* Si el inicio es exitoso, guarda el token y el perfil en caché.
* Retorna true si el login fue exitoso, false en caso contrario.
  + - 1. Logout()

Cierra la sesión tanto en el backend como en el dispositivo.

* Envía la solicitud a /api/logout.
* Limpia los datos locales incluso si la conexión falla.
* Muestra logs descriptivos en modo debug.
  + 1. Manejo del perfil
       1. getProfile({bool forceRefresh = false})

Obtiene el perfil del usuario autenticado.

* Si forceRefresh es false, usa el caché local.
* Si no hay caché o se requiere actualización, consulta /api/profile.
* En caso de token inválido (401), borra la sesión local.
  + - 1. updateProfile()

Actualiza la información personal del usuario (nombre, correo, contraseña o foto).

* Envía una solicitud PUT a /api/profile.
* Si la actualización es exitosa, actualiza el perfil local y notifica a los listeners.
  + - 1. changePassword()

Permite cambiar la contraseña del usuario autenticado.

* Envía una solicitud POST a /api/change-password.
* Valida los campos: contraseña actual, nueva y confirmación.
  + - 1. uploadProfilePhoto(String filePath)

Sube una nueva foto de perfil al servidor y actualiza el perfil local.

* Usa MultipartRequest para enviar el archivo.
* Actualiza la propiedad photoUrl en \_cachedProfile.
  + - 1. updateLocalProfile()

Actualiza el perfil del usuario en caché y en SharedPreferences sin requerir conexión al servidor. Se utiliza cuando el perfil cambia desde la UI localmente.

* + 1. Comunicación HTTP segura

Todos los endpoints protegidos utilizan el método \_authHeaders() para incluir el encabezado:

Authorization: Bearer <token>

De esta forma, se garantiza que solo usuarios autenticados puedan acceder a los recursos protegidos del backend.

* + 1. Persistencia y sincronización local

AuthService utiliza SharedPreferences para almacenar:

|  |  |  |
| --- | --- | --- |
| Clave | Contenido | Tipo |
| auth\_token | Token JWT devuelto por Laravel. | String |
| auth\_user | Perfil serializado del usuario autenticado. | String (JSON) |

Esto permite que la aplicación recuerde la sesión incluso después de ser cerrada.

* + 1. Ciclo de autenticación

Usuario → Inicia sesión

↓

AuthService.login()

↓

Token y perfil guardados localmente

↓

AuthService.getProfile() → Perfil actualizado desde backend

↓

UI actualizada con datos del usuario

* + 1. Diagrama simplificado



* 1. **Servicio cancha\_service.dart**

Ubicación: lib/services/cancha\_service.dart

Propósito: Gestionar la obtención, almacenamiento y sincronización de la información de las canchas deportivas entre el backend Laravel y la base de datos local del dispositivo (Drift). Este servicio proporciona los métodos necesarios para consultar todas las canchas o una cancha específica, utilizando caché local en caso de pérdida de conexión.

* + 1. Descripción general

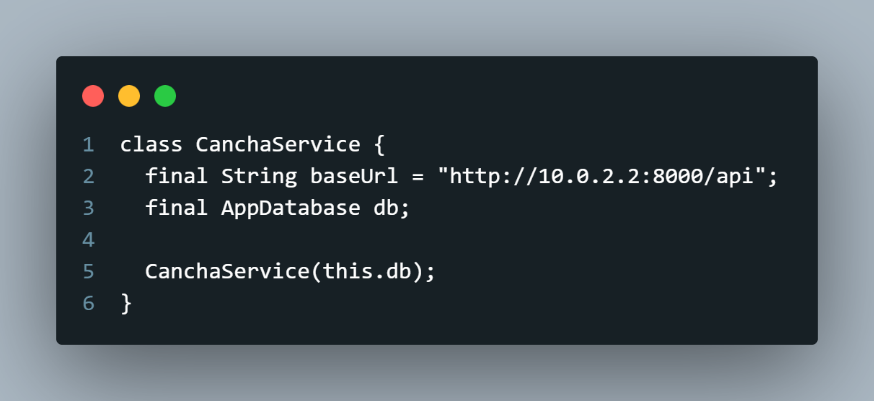
El servicio CanchaService actúa como un puente entre el backend Laravel y la base local (SQLite/Drift), permitiendo que la aplicación funcione tanto en línea como fuera de línea (offline).

Su diseño permite:

* Obtener datos actualizados desde el servidor.
* Almacenar temporalmente la información para uso sin conexión.
* Recuperar datos del caché local cuando no hay conexión disponible.
  + 1. Dependencias principales

|  |  |
| --- | --- |
| Dependencia | Uso principal |
| http | Realiza las solicitudes HTTP hacia la API de Laravel. |
| dart:convert | Convierte datos JSON a objetos Dart (Map, List). |
| flutter/foundation.dart | Permite el uso de debugPrint para mensajes de depuración. |
| cancha.dart | Define el modelo de dominio de una cancha. |
| cancha\_mapper.dart | Convierte los datos JSON en instancias del modelo Cancha. |
| app\_database.dart | Gestiona la persistencia local mediante Drift (SQLite). |

* + 1. Estructura General



|  |  |
| --- | --- |
| Campo | Descripción |
| baseUrl | Dirección base del backend Laravel, adaptada al entorno del emulador Android. |
| db | Instancia de la base de datos local AppDatabase, utilizada para guardar o recuperar canchas en caché. |

* + - 1. Método obtenerCanchas()

Future<List<Cancha>> obtenerCanchas() async { ... }

Descripción

Obtiene la lista completa de canchas desde el backend Laravel.  
En caso de error de conexión o respuesta inválida, se recuperan los datos almacenados localmente en la base app\_cache.db.

Flujo de trabajo

1. Realiza una solicitud GET al endpoint /api/canchas.
2. Interpreta la respuesta JSON proveniente del backend.
3. Convierte la respuesta a una lista de objetos Cancha utilizando CanchaMapper.
4. Guarda la lista de canchas en la base de datos local (cachearCanchas).
5. Si ocurre una excepción (falla de red, servidor inactivo, etc.), carga los datos almacenados en caché local.
   * + 1. Método getCanchaById(int id)

Future<Cancha?> getCanchaById(int id) async { ... }

Descripción

Obtiene la información detallada de una cancha específica según su ID.

Flujo de trabajo

1. Realiza una solicitud GET al endpoint /api/canchas/{id}.
2. Si la respuesta es exitosa (200), decodifica el JSON.
3. Crea un objeto Cancha con la información devuelta.
4. Actualiza el caché local con los nuevos datos de esa cancha.
5. Si la API no está disponible o la cancha no existe, se busca el registro en la base local (db.obtenerCanchas()).

Comportamiento Offline

Si el dispositivo está sin conexión:

* Busca la cancha localmente usando su id.
* Devuelve un objeto Cancha creado a partir de los datos guardados.
  + 1. Integración con la base local (Drift)

CanchaService utiliza los siguientes métodos del archivo app\_database.dart:

|  |  |
| --- | --- |
| Método | Descripción |
| cachearCanchas(List<Map<String, dynamic>> data) | Inserta o actualiza todas las canchas en la tabla local. |
| obtenerCanchas() | Devuelve una lista de todas las canchas almacenadas en caché. |

* + 1. Manejo de errores y depuración

El servicio utiliza debugPrint() para registrar errores en modo desarrollo.  
Esto incluye:

* Fallas de red.
* Respuestas HTTP con códigos distintos de 200.
* Problemas de parseo de datos JSON.
* Errores al acceder al caché local.

Ejemplo de log:

⚠️ Error o sin conexión. Cargando cache local: SocketException: Connection refused

* 1. **Servicio notification\_service.dart**

Ubicación: lib/services/notification\_service.dart

Propósito: Gestionar la configuración, programación y envío de notificaciones locales dentro de la aplicación SportReserve Mobile. Este servicio permite mostrar notificaciones inmediatas, programar recordatorios de reservas y manejar las preferencias del usuario sobre avisos y promociones.

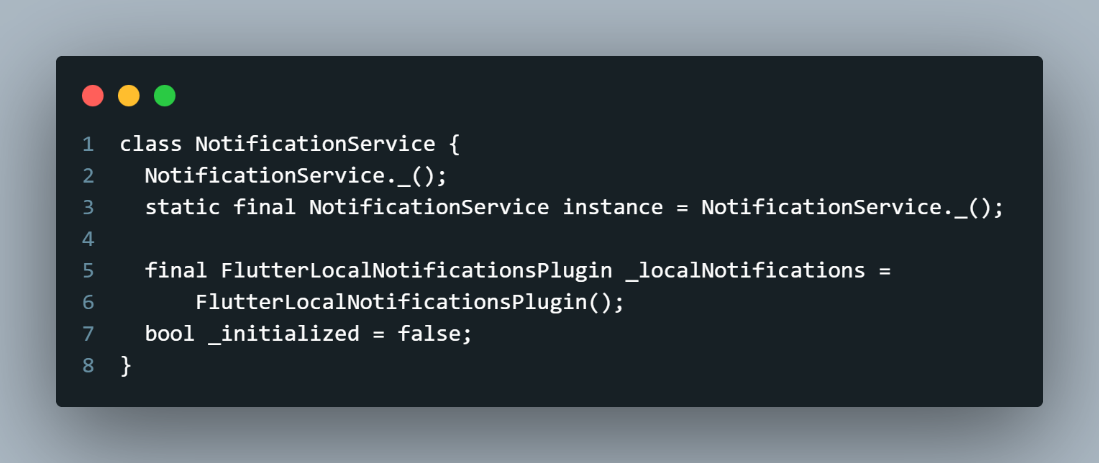
* + 1. Descripción general

El servicio NotificationService proporciona una interfaz centralizada para el manejo de notificaciones locales, utilizando la librería flutter\_local\_notifications.  
Su objetivo es mejorar la experiencia del usuario mediante alertas programadas, recordatorios automáticos y notificaciones en tiempo real, incluso cuando la aplicación no está abierta. El sistema está completamente integrado con la zona horaria local, garantizando la precisión de los recordatorios sin importar la ubicación del dispositivo.

* + 1. Dependencias principales

|  |  |
| --- | --- |
| Dependencia | Función |
| flutter\_local\_notifications | Gestión de notificaciones locales para Android. |
| shared\_preferences | Almacenamiento de preferencias de usuario (activación o desactivación de promociones). |
| timezone | Control y conversión de zonas horarias. |
| dart:math | Generación de identificadores aleatorios para notificaciones. |
| flutter/foundation.dart | Registro de logs durante la depuración (debugPrint). |

* + 1. Arquitectura de la clase



|  |  |
| --- | --- |
| Elemento | Descripción |
| Constructor privado (NotificationService.\_()) | Impide la creación de múltiples instancias. |
| Singleton instance | Garantiza un único punto de acceso global. |
| \_localNotifications | Controlador del sistema de notificaciones locales. |
| \_initialized | Bandera que indica si el servicio ya fue inicializado. |

* + - 1. Inicialización del servicio

Método: initialize()

Future<void> initialize() async {

if (\_initialized) return;

await \_initializeTimezone();

await \_configureLocalNotifications();

await \_applyPromotionPreference();

\_initialized = true;

}

Descripción:

Inicializa el servicio de notificaciones locales.  
Incluye la configuración de zonas horarias, canales de notificación y aplicación de preferencias guardadas del usuario.

Comportamiento:

* Se ejecuta una sola vez por ciclo de vida de la aplicación.
* Llama internamente a tres métodos privados:
  + \_initializeTimezone()
  + \_configureLocalNotifications()
  + \_applyPromotionPreference()
    - 1. Configuración de zona horaria

Método: \_initializeTimezone()

Future<void> \_initializeTimezone() async {

tz.initializeTimeZones();

try {

tz.setLocalLocation(tz.getLocation(DateTime.now().timeZoneName));

} catch (\_) {

tz.setLocalLocation(tz.getLocation('America/Bogota'));

}

}

Descripción:

Configura la zona horaria local del dispositivo para que las notificaciones se programen correctamente.  
En caso de no detectar la ubicación, establece una zona horaria predeterminada: “America/Bogotá”.

* + - 1. Configuración del sistema de notificaciones

Método: \_configureLocalNotifcations()

Future<void> \_configureLocalNotifications() async {

const androidSettings = AndroidInitializationSettings('@mipmap/ic\_launcher');

const darwinSettings = DarwinInitializationSettings();

await \_localNotifications.initialize(

const InitializationSettings(android: androidSettings, iOS: darwinSettings),

onDidReceiveNotificationResponse: \_onNotificationResponse,

);

}

Descripción:

Inicializa la configuración de notificaciones locales para Android y iOS.  
Define los iconos, permisos y el comportamiento cuando el usuario toca una notificación.

Importante:

* En Android utiliza el ícono @mipmap/ic\_launcher.
* En iOS aplica los permisos definidos por DarwinInitializationSettings.
  + - 1. Notificaciones inmediatas

Método: showInstantNotification()

Future<void> showInstantNotification({

required String title,

required String body,

}) async

Función: Muestra una notificación inmediata dentro de la aplicación.

Características:

* Se ejecuta instantáneamente sin programación.
* Útil para mensajes de confirmación, alertas de sistema o recordatorios inmediatos.

Ejemplo de uso:

await NotificationService.instance.showInstantNotification(

title: 'Reserva confirmada',

body: 'Tu cancha de fútbol está lista para mañana a las 8:00 AM',

);

* + - 1. Programación de recordatorios

Método: scheduleReminder()

Future<void> scheduleReminder({

required DateTime eventDateTime,

required String title,

required String body,

Duration advance = const Duration(hours: 1),

})

Función: Programa una notificación que se mostrará antes del inicio de una reserva o evento.

Parámetros:

|  |  |  |
| --- | --- | --- |
| Nombre | Tipo | Descripción |
| eventDateTime | DateTime | Fecha y hora del evento (por ejemplo, inicio de reserva). |
| title | String | Título de la notificación. |
| body | String | Descripción o detalle del recordatorio. |
| advance | Duration | Tiempo de antelación para enviar el aviso (por defecto, 1 hora antes). |

Ejemplos de uso:

await NotificationService.instance.scheduleReminder(

eventDateTime: DateTime(2025, 11, 10, 18, 0),

title: 'Recordatorio de reserva',

body: 'Tu partido comienza en 1 hora',

);

Comportamiento:

* Calcula la hora programada restando el tiempo definido en advance.
* Si la fecha ya pasó, el método no hace nada.
* Crea una notificación persistente mediante zonedSchedule() con prioridad alta.
  + - 1. Interacción del usuario

Método: \_onNotificationResponse()

Future<void> \_onNotificationResponse(NotificationResponse response) async {

if (kDebugMode) {

debugPrint('Notificación tocada: ${response.payload}');

}

}

**Función:** Maneja la acción del usuario al tocar una notificación. Actualmente, muestra un log en modo depuración, pero puede adaptarse para abrir páginas específicas (por ejemplo, la reserva correspondiente).

* + - 1. Preferencias del usuario

Método: updatePromotionPreference(bool enabled)

Guarda la preferencia del usuario sobre si desea recibir notificaciones promocionales.

Future<void> updatePromotionPreference(bool enabled) async {

final prefs = await SharedPreferences.getInstance();

await prefs.setBool('promotionsEnabled', enabled);

}

Método: \_applyPromotionPreference()

Carga la preferencia guardada al iniciar la aplicación y aplica su configuración.

Future<void> \_applyPromotionPreference() async {

final prefs = await SharedPreferences.getInstance();

final enabled = prefs.getBool('promotionsEnabled') ?? true;

if (kDebugMode) {

debugPrint(enabled

? '🔔 Promociones activadas localmente.'

: '🔕 Promociones desactivadas.');

}

}

Uso: Estas funciones permiten controlar si el usuario desea recibir notificaciones de tipo “promocional” sin afectar los recordatorios de reservas.

* 1. Servicio reservation\_service.dart

Ubicación: lib/services/reservation\_service.dart

Propósito: Gestionar todas las operaciones relacionadas con reservas deportivas dentro de la aplicación SportReserve Mobile, incluyendo creación, cancelación, consulta de disponibilidad y horarios, así como la sincronización con el backend Laravel mediante autenticación protegida con tokens.

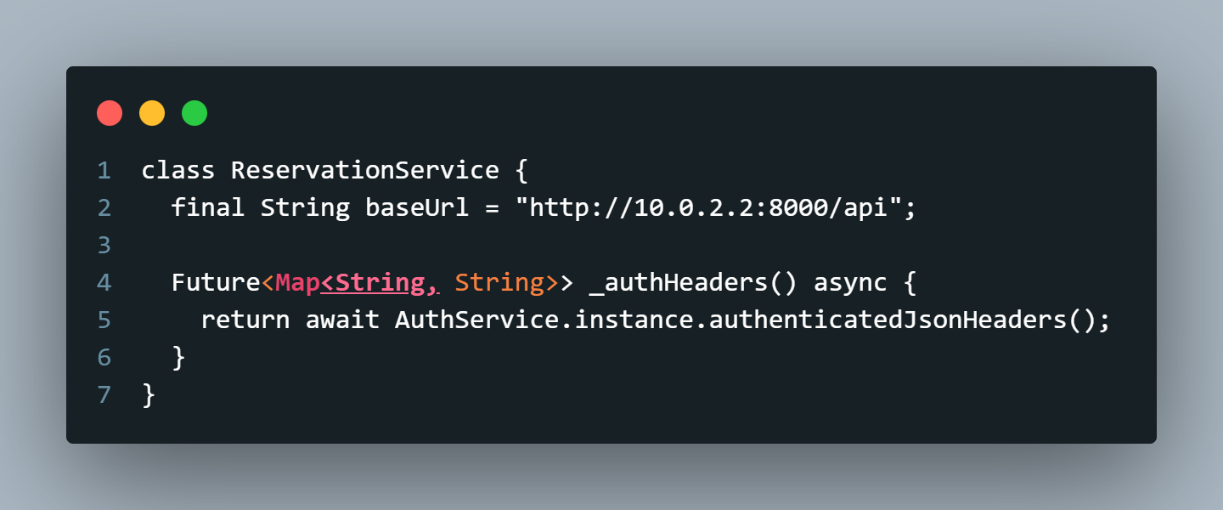
* + 1. Descripción general

El servicio ReservationService constituye el **módulo central de gestión de reservas** en SportReserve Mobile. Se comunica directamente con la API REST de Laravel, utilizando los encabezados autenticados proporcionados por AuthService, para garantizar que solo usuarios válidos puedan crear o consultar reservas.Además, ofrece un manejo flexible de los datos recibidos del servidor, permitiendo procesar tanto respuestas en formato Map como List, según la estructura definida por el backend.

* + 1. Dependencias principales

|  |  |
| --- | --- |
| Dependencia | Función |
| http | Comunicación HTTP con el backend Laravel. |
| dart:convert | Codificación y decodificación de datos JSON. |
| auth\_service.dart | Obtención del token de autenticación y encabezados seguros. |
| reservation\_slot.dart | Representación de intervalos de tiempo disponibles por cancha. |

* + 1. Estructura de la clase



|  |  |
| --- | --- |
| Elemento | Descripción |
| baseUrl | Define la dirección base del backend Laravel (usa 10.0.2.2 para conexión desde el emulador Android). |
| \_authHeaders() | Obtiene los encabezados HTTP con el token JWT actual para solicitudes protegidas. |

* + 1. Funcionalidadesprincipales
       1. Crear una nueva reserva

Future<Map<String, dynamic>> crearReserva({

required int canchaId,

required String fecha,

required String hora,

required int cantidadHoras,

required double precioPorCancha,

})

Descripción

Envía una solicitud POST al endpoint /api/reservas para crear una nueva reserva en el sistema.

Parámetros

|  |  |  |
| --- | --- | --- |
| Campo | Tipo | Descripción |
| canchaId | int | Identificador de la cancha seleccionada. |
| fecha | String | Fecha del evento en formato YYYY-MM-DD. |
| hora | String | Hora de inicio de la reserva (HH:mm). |
| cantidadHoras | int | Duración de la reserva en horas. |
| precioPorCancha | double | Valor cobrado por hora o total. |

* + - 1. Flujo de ejecución
* Obtiene los encabezados autenticados desde AuthService.
* Envía la solicitud POST con los datos en formato JSON.
* Si el token es inválido (401), lanza una excepción Unauthenticated.
* Devuelve la respuesta decodificada del backend como Map<String, dynamic>.

Ejemplo de uso

final service = ReservationService();

final result = await service.crearReserva(

canchaId: 2,

fecha: '2025-11-08',

hora: '18:00',

cantidadHoras: 2,

precioPorCancha: 40000,

);

* + - 1. Consultar disponibilidad de una cancha

Future<Map<String, dynamic>> obtenerDisponibilidad({

required int canchaId,

required String fecha,

})

Descripción

Consulta los horarios disponibles de una cancha para una fecha específica mediante el endpoint /api/canchas/{id}/disponibilidad.

Flujo

* Llama a la API con los encabezados autenticados.
* Si la respuesta indica error 401, lanza excepción.
* Devuelve el resultado JSON con los horarios disponibles.

Ejemplo de respuesta esperada

{

"cancha\_id": 2,

"fecha": "2025-11-08",

"disponible": true,

"slots": [

{ "start\_at": "18:00", "end\_at": "19:00", "is\_available": true },

{ "start\_at": "19:00", "end\_at": "20:00", "is\_available": false }

]

}

* + - 1. Obtener reservas del usuario autenticado

Future<dynamic> obtenerMisReservas()

Descripción

Consulta las reservas asociadas al usuario que ha iniciado sesión, mediante el endpoint /api/mis-reservas.

Flujo

* Realiza solicitud GET con encabezados de autenticación.
* Devuelve la respuesta JSON del backend (puede ser lista o mapa según configuración del servidor).
* Maneja errores de sesión (401).

Ejemplo de respuesta

[

{

"id": 5,

"cancha": { "nombre": "Cancha FET Sur" },

"fecha": "2025-11-08",

"hora": "18:00",

"estado": "confirmada"

}

]

Ejemplo de uso

final reservas = await ReservationService().obtenerMisReservas();

print(reservas);

* + - 1. Cancelar una reserva existente

Future<Map<String, dynamic>> cancelarReserva(int reservaId)

Descripción

Permite al usuario cancelar una reserva específica mediante el endpoint /api/reservas/{id}/cancelar.

Flujo

* Envía una solicitud PUT al backend.
* Si la reserva pertenece al usuario autenticado, se actualiza su estado a “cancelada”.
* Devuelve la respuesta JSON del backend con la confirmación del cambio.

Ejemplo de respuesta

{

"success": true,

"message": "La reserva ha sido cancelada exitosamente."

}

* + - 1. Obtener horarios disponibles por día

Future<List<ReservationSlot>> fetchDailySlots({

required int canchaId,

required DateTime day,

})

Descripción

Recupera los intervalos de tiempo disponibles (ReservationSlot) para una cancha en un día determinado.  
Usa el endpoint /api/canchas/{id}/horarios.

Flujo

* Construye la fecha en formato YYYY-MM-DD.
* Envía una solicitud GET autenticada.
* Si la respuesta no es 200, lanza excepción con el código de error.
* Convierte la lista JSON en una lista de objetos ReservationSlot.

Ejemplo de uso

final horarios = await ReservationService().fetchDailySlots(

canchaId: 1,

day: DateTime.now(),

);

for (final slot in horarios) {

print('${slot.startAt} → ${slot.isAvailable ? "Disponible" : "Ocupado"}');

}