salesians
SARRIÀ

Programación multimedia y dispositivos móviles.

Flutter: Widgets y listas de personajes de una API

Alumno: Damian Altamirano Ontiveros Rivero.

Índice

0. Introducción

Con el avance del tiempo, es mas común escuchar como las APIs han cambiado la forma en como se comunican los sistemas de manera rápida y eficiente. Dominar estas tecnologías dentro del entorno del desarrollo de aplicaciones con Flutter nos permitirá ser más versátiles y con las funcionalidades que podemos ofrecer, y tener conectividad e información más actualizada.

En esta documentación, veremos como crear una aplicación para almacenar listas de jugadores, usando una API de por medio, y el poder editar datos relevantes de estos a nuestro gusto (los cambios que hagamos no se verán reflejados en el host, solo de manera local)

1. APIs

Una API no es más que una interfaz que nos permite realizar comunicaciones o peticiones con un cliente o servicio para obtener datos, realizar cambios en bases de datos, eliminar elementos, etc.

Hoy en día son usados en gran manera para conectar sistemas que pueden estar desarrollados en diferentes entornos, de manera rápida y fácil.

Usualmente antes de empezar a trabajar con una API nuevo, se realizan una serie de pruebas y verificaciones previas que nos permiten verificar la velocidad y calidad de las peticiones, así como también, se usa para ver como esta semiestructurada la información y los metadatos, y saber trabajar con estos.

1.1 Endpoints

Un endpoint (o punto final) es esencialmente el **punto de contacto digital** al que un sistema cliente (por ejemplo, una aplicación móvil, un navegador web o un servidor) puede conectarse para acceder a un **servicio o recurso** que reside en un servidor remoto.

En la práctica, un endpoint es una combinación de la **URL base** de la API y una **ruta** específica, junto con el **método HTTP** utilizado para interactuar con ese recurso.

Estructura Típica

- **URL Base:** https://api.ejemplo.com/v1/
- **Ruta/Recurso:** /usuarios
- **Endpoint Completo:** https://api.ejemplo.com/v1/usuarios (más el método HTTP)

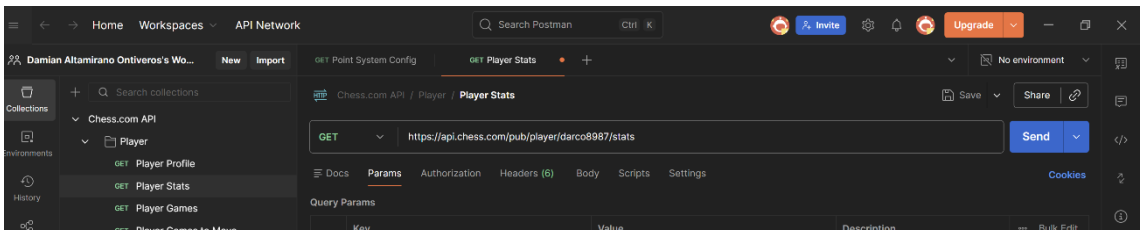
Los tipos de endpoints más usados son:

- Get: Recuperar datos de un recurso especificado por la URL. No modifica el estado del servidor.
- Post: Enviar datos al servidor para **crear un nuevo recurso** bajo la URL especificada. Los datos para crear se envían en el cuerpo (body) de la solicitud.
- Put/Patch: Modificar un recurso existente.
- Delete: Eliminar el recurso especificado por la URL.

Antes de comenzar con nuestro proyecto, haremos uso de la herramienta **Postman** para ver la información de retorno del endpoint **GET https://api.chess.com/pub/player/$username"**

1.2 Postman

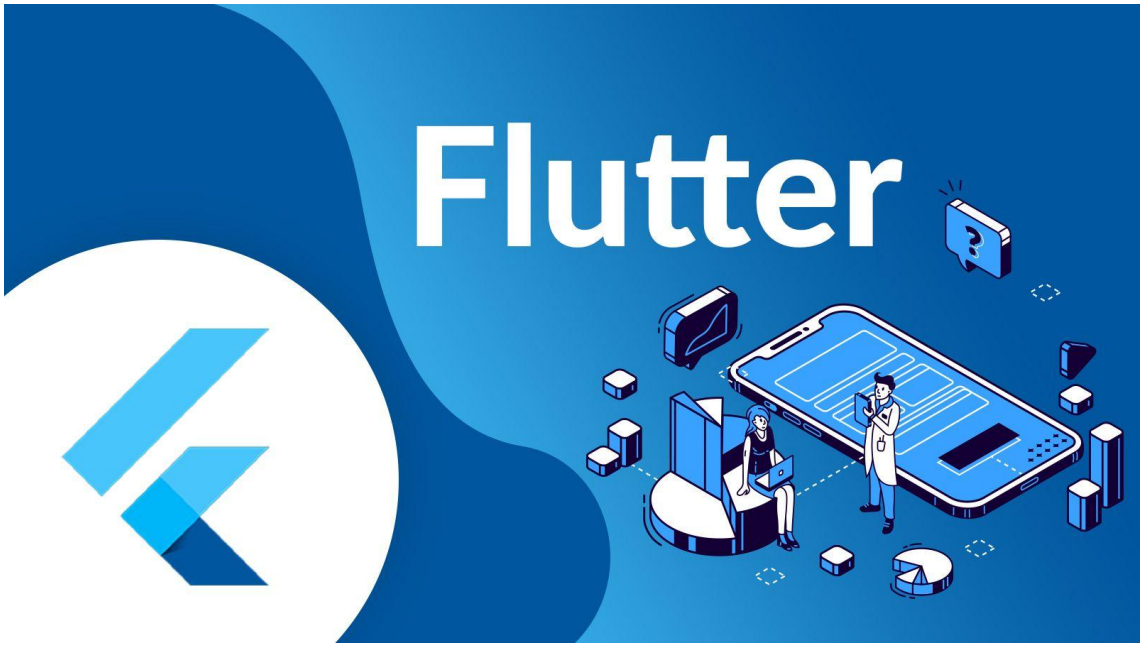
Postman es una herramienta multiplataforma que facilita el desarrollo y prueba de APIs, permitiendo enviar peticiones HTTP (GET, POST, PUT, DELETE, etc.), inspeccionar respuestas, organizar colecciones de endpoints y automatizar pruebas; se utiliza ampliamente por desarrolladores para documentar, depurar y colaborar en proyectos que consumen servicios web.



2. Librerías Principales

Antes de empezar con el desarrollo de nuestros proyectos, empezaremos hablando de librerías imprescindibles para construir módulos de conectividad con nuestra API y “builders” que nos ahorrarán tiempo a la hora de construir los formularios.

Todos los paquetes que importemos deberán de ser declarados en el archivo de dependencias del proyecto (pubpsec.yaml), y tener la última versión de este declarada he instalada.



2.1 http

Una librería que nos proporciona los desarrolladores de Dart para facilitar las peticiones http. Este paquete contiene un conjunto de funciones y clases de alto nivel que lo componen fácil de consumir recursos HTTP. Es multiplataforma (móvil, escritorio y navegador) y soporta múltiples implementaciones.

2.2 flutter_form_builder

Es una herramienta fundamental en nuestros proyectos Flutter. Desarrollado por los creadores del framework, esta librería ofrece una amplia gama de widgets preconfigurados que simplifican la creación de formularios interactivos, validados y altamente personalizables. Su objetivo principal es acelerar el desarrollo manteniendo la coherencia visual y funcional en toda la aplicación.

Entre sus componentes más destacados se encuentran:

- **FormBuilderCheckbox**: Campo de selección individual tipo checkbox.
- **FormBuilderCheckboxGroup**: Grupo de checkboxes para selección múltiple.
- **FormBuilderChoiceChip**: Chips que funcionan como botones de opción (radio).
- **FormBuilderDateRangePicker**: Selector de rangos de fechas.
- **FormBuilderDateTimePicker**: Entrada combinada de fecha y hora.
- **FormBuilderDropdown**: Menú desplegable para seleccionar un valor de una lista.
- **FormBuilderFilterChip**: Chips que actúan como checkboxes.
- **FormBuilderRadioGroup**: Grupo de botones de opción para selección única.
- **FormBuilderRangeSlider**: Selector de rango numérico mediante slider.
- **FormBuilderSlider**: Slider para seleccionar un valor numérico.
- **FormBuilderSwitch**: Interruptor tipo On/Off.
- **FormBuilderTextField**: Campo de texto con estilo Material Design.

Estos widgets están diseñados para integrarse fácilmente con el sistema de validación y estado de los formularios, permitiendo una experiencia de desarrollo más fluida y escalable.

2.3 form_builder_extra_fields

Este amplía las capacidades del paquete base **flutter_form_builder**, ofreciendo widgets especializados que cubren casos de uso más complejos y específicos. Estos componentes permiten integrar funcionalidades avanzadas sin necesidad de desarrollarlas desde cero, lo que acelera el desarrollo y mejora la experiencia del usuario final.

Entre los widgets más útiles se encuentran:

- **FormBuilderColorPicker**: Selector de color con soporte para paletas, RGB y HSV.
- **FormBuilderCupertinoDateTimePicker**: Selector de fecha y hora con estilo iOS.
- **FormBuilderImagePicker**: Permite seleccionar imágenes desde la galería o la cámara.
- **FormBuilderPhoneField**: Campo de entrada para números de teléfono con validación internacional.

- **FormBuilderRating**: Componente para capturar valoraciones mediante estrellas u otros íconos.
- **FormBuilderSignaturePad**: Área de dibujo para capturar firmas digitales.
- **FormBuilderTouchSpin**: Selector numérico con botones de incremento/decremento.
- **FormBuilderTypeAhead**: Campo de texto con sugerencias automáticas mientras el usuario escribe.

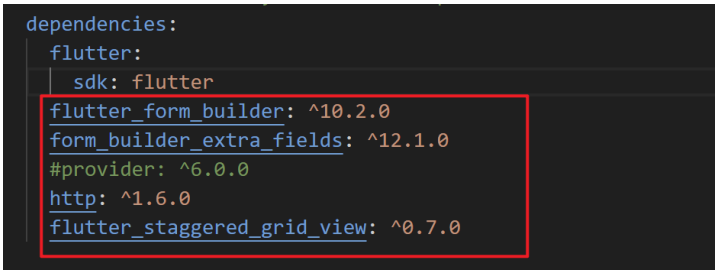
Estos widgets están diseñados para integrarse perfectamente con el sistema de validación y estado de flutter_form_builder, manteniendo la coherencia y escalabilidad del formulario.

2.4 flutter_staggered_grid_view

Esta librería permite crear cuadrículas flexibles y dinámicas donde los elementos no tienen un tamaño uniforme, ideal para interfaces tipo Pinterest, galerías de imágenes o dashboards; ofrece widgets como MasonryGridView, StaggeredGridView y AlignedGridView que facilitan organizar ítems con alturas o proporciones distintas, manteniendo un diseño atractivo y responsivo en aplicaciones móviles y de escritorio.

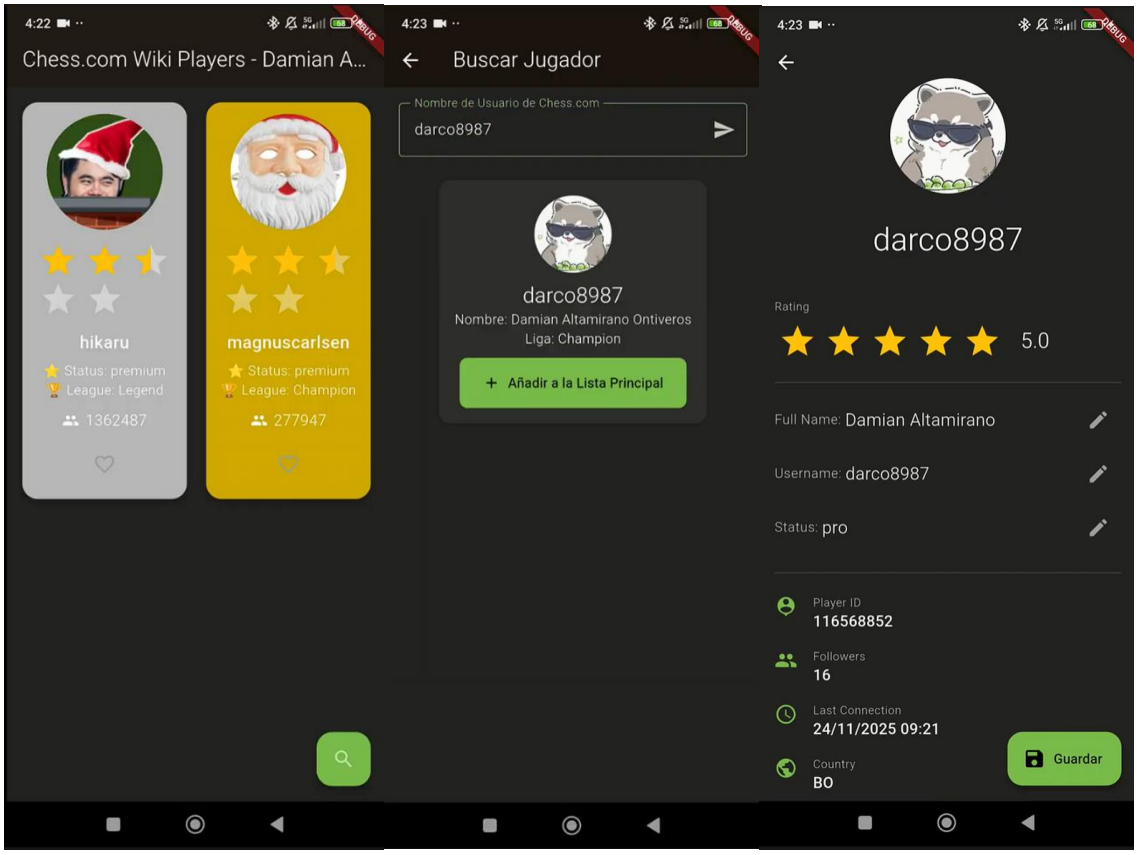
2.5 Versiones del projecto

Estas son las versiones que hemos usado de cada una de las librerías ya mencionadas para este proyecto:



3. Projecto: Chess_players

Nuestra aplicación constara de una serie de funcionalidades que nos permitirán buscar jugadores de **Chess.com** y guardalos en memoria, pudiendo acceder a sus datos y estadísticas y poder darles una calificación o notas personalizadas.



3.1 Chess.com API y Endpoints usados

La Chess.com API es una interfaz pública de tipo REST, de solo lectura, que permite acceder en formato JSON a datos abiertos de la plataforma Chess.com, como perfiles de jugadores, partidas, estadísticas, clubes, torneos y rompecabezas, pero sin incluir información privada ni la posibilidad de enviar movimientos o comandos. (documentación oficial)

Antes de comenzar con el desarrollo de nuestra aplicación, es importante saber de antemano y establecer los endpoints que usaremos, y el tipo de datos que retorna. Para ello, haremos uso de la herramienta Postman.

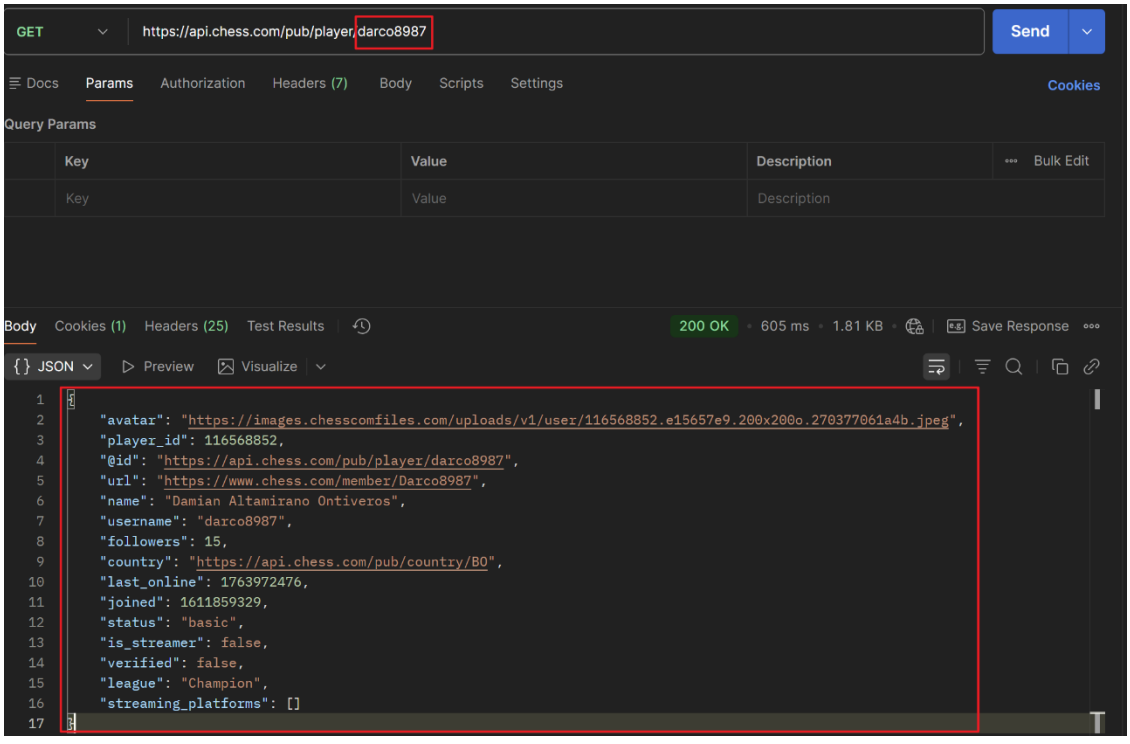
3.1.1 GET: Player Profile

> URL: https://api.chess.com/pub/player/{**username**}

Este endpoint nos permite obtener datos adicionales de los usuarios de chess.com a partir de su usuario. Si vemos la documentación veremos los datos que nos retorna:

```
{
  "@id": "URL", // the location of this profile (always self-referencing)
  "url": "URL", // the chess.com user's profile page (the username is displayed
with the original letter case)
  "username": "string", // the username of this player
  "player_id": 41, // the non-changing Chess.com ID of this player
  "title": "string", // (optional) abbreviation of chess title, if any
  "status": "string", // account status: closed, closed:fair_play_violations,
basic, premium, mod, staff
  "name": "string", // (optional) the personal first and last name
  "avatar": "URL", // (optional) URL of a 200x200 image
  "location": "string", // (optional) the city or location
  "country": "URL", // API location of this player's country's profile
  "joined": 1178556600, // timestamp of registration on Chess.com
  "last_online": 1500661803, // timestamp of the most recent login
  "followers": 17 // the number of players tracking this player's activity
  "is_streamer": "boolean", //if the member is a Chess.com streamer
  "twitch_url": "Twitch.tv URL",
  "fide": "integer" // FIDE rating
}
```

además, podemos hacer una pequeña validación con algún usuario que conozcamos para ver un ejemplo más claro con Postman:



3.2 Módulos

Ahora que tenemos todas las dependencias instaladas con sus respectivas versiones, nos centraremos en diseñar y organizar cada componente de nuestra aplicación y sus funcionalidades. En nuestro caso, los distribuiremos de la siguiente manera:

3.2.1 Entities

El módulo Entities actúa como el modelo de datos central para la aplicación. Su función principal es definir la estructura de las entidades clave que manejan el flujo de información, siendo la más importante la clase **Player**, la cual, encapsula todos los atributos de un jugador de Chess.com, incluyendo datos obtenidos de la API (como username, league y followers) y propiedades personalizadas de la aplicación (como rating, notes y isFavorite).

3.2.1.1 Player

A partir de los datos que obtenemos del endpoint, podemos hacer un modelo y construir nuestra clase con las propiedades que deseemos. En mi caso he tomado las mas relevantes y he establecido como constantes aquellas que no podran ser cambiadas a futuro, ademas de crear algunas nuevas:

```
{
  "@id": "URL", // the location of this profile (always self-referencing)
  "url": "URL", // the chess.com user's profile page (the username is displayed
with the original letter case)
  "username": "string", // the username of this player
  "player_id": 41, // the non-changing Chess.com ID of this player
  "title": "string", // (optional) abbreviation of chess title, if any
  "status": "string", // account status: closed, closed:fair_play_violations,
basic, premium, mod, staff
  "name": "string", // (optional) the personal first and last name
  "avatar": "URL", // (optional) URL of a 200x200 image
  "location": "string", // (optional) the city or location
  "country": "URL", // API location of this player's country's profile
  "joined": 1178556600, // timestamp of registration on Chess.com
  "last_online": 1500661803, // timestamp of the most recent login
  "followers": 17 // the number of players tracking this player's activity
  "is_streamer": "boolean", //if the member is a Chess.com streamer
  "twitch_url": "Twitch.tv URL",
  "fide": "integer" // FIDE rating
}
```

```dart
class Player {
  final String avatar;
  final int playerId;
  final String id;
  final String url;
  String name;
  String username;
  int followers;
  final String country;
  final int lastOnline;
  final int joined;
  String status;
  final bool isStreamer;
  final bool verified;
  String league;
  final List<String> streamingPlatforms;
  //optionals properties
  bool isFavorite;
  String? notes;
  double rating;
```

Una vez declaradas las propiedades, deberemos de crear el constructor según nuestras preferencias, pudiendo poner valores por defecto si asi lo desearamos.

```
Player({
  required this.avatar,
  required this.playerId,
  required this.id,
  required this.url,
  required this.name,
  required this.username,
  required this.followers,
  required this.country,
  required this.lastOnline,
  required this.joined,
  required this.status,
  required this.isStreamer,
  required this.verified,
  required this.league,
  required this.streamingPlatforms,
  this.isFavorite = false,
  this.rating = 2.5,
});
```

Por último, Crearemos un constructor especial para poder convertir el JSON decodificado a un objeto **Player** y instanciandolo en nuestro programa.

```
factory Player.fromJson(Map<String, dynamic> json) {
  List<String> platforms = [];

  return Player(
    avatar:
        json['avatar'] ??
        'https://static.vecteezy.com/system/resources/previews/004/511/281/original/default-avatar-photo-pla
    playerId: json['player_id'] ?? 0,
    name: json['name'] ?? 'N/A',
    id: json['@id'] ?? '',
    url: json['url'] ?? '',
    username: json['username'] ?? '',
    followers: json['followers'] ?? 0,
    country: json['country'] ?? '',
    lastOnline: json['last_online'] ?? 0,
    joined: json['joined'] ?? 0,
    status: json['status'] ?? '',
    isStreamer: json['is_streamer'] ?? false,
    verified: json['verified'] ?? false,
    league: json['league'] ?? '',
    streamingPlatforms: platforms,
  );
}
```

Este tipo de constructores son mas flexibles con los tipos de datos que podemos establecer y los diferentes casos a los que este se vea sometido. En nuestro caso, para cada propiedad hacemos referencia al valor asociado a la clave JSON. Mediante los simbolos "**??**" le indicamos al constructor, que de no encontrar la clave, este establezca un valor por defecto, controlando así futuras excepciones.

```
factory Player.fromJson(Map<String, dynamic> json) {
```

Cabe recalcar, que al declarar nuestra **factory**, si estamos tratando con JSON esta debera tener el argumento "**Map<String, dynamic>**" siempre.

Dynamic quiere decir que puede tener cualquier tipo de información como valor (strings, ints, listas, otros maps).

```
list.add(Player.fromJson(data));
```

Para hacer uso de nuestro **factory**, bastará con llamarlo como si fuese un método y pasarle el JSON obtenido como argumento.

3.2.2 Api Services

En este módulo se definen los servicios de comunicación con la API, encargados de gestionar las peticiones y respuestas entre la aplicación y el servidor. Su propósito es abstraer la lógica de acceso a datos, ofreciendo métodos reutilizables que transforman la información recibida en objetos manejables dentro de la aplicación. De esta manera, se facilita la integración de datos externos y se mantiene el código más organizado y modular.

3.2.2.1 Get players service

Antes de comenzar, importaremos las siguientes librerías:

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;
```

Usaremos **Converter** para decodificar el json que obtenemos de la peticion API y obtener un objeto JSON que usaremos para poder transformar los datos a objetos alterables.

También, usaremos **http** para simplificar las peticiones hechas al servidor API mediante Dart.

Además. Importaremos el package de **player** para poder castearlo correctamente.

```dart
import 'package:chess_players/entities/player.dart';
```

Creaeremos una clase llamada **PlayersApiService** la cual tendrá dos metodos estáticos muy importantes:

```dart
class PlayersApiService {
  static Future<Player> fetchSinglePlayer(String username) async {
    final response = await http.get(
      Uri.parse("https://api.chess.com/pub/player/$username"),
    );

    if (response.statusCode == 200) {
      final data = jsonDecode(response.body);
      return Player.fromJson(data);
    } else {
      throw Exception(
        'Player $username not found. HTTP Status: ${response.statusCode}',
      );
    }
  }

  static Future<List<Player>> fetchPlayers(List<String> playersUsername) async {
    final List<Future<Player>> fetchFutures = playersUsername.map((username) {
      return fetchSinglePlayer(username);
    }).toList();
    try {
      final List<Player> players = await Future.wait(fetchFutures);
      return players;
    } catch (e) {
      rethrow;
    }
  }
}
```

El metodo **fetchSinglePlayer** nos permite obtener los datos de un usuario en concreto mediante su **username** haciendo uso del endpoint ya antes mencionado y nos lo retorna.

```
static Future<Player> fetchSinglePlayer(String username) async {
  final response = await http.get(
    Uri.parse("https://api.chess.com/pub/player/$username"),
  );

  if (response.statusCode == 200) {
    final data = jsonDecode(response.body);
    return Player.fromJson(data);
  } else {
    throw Exception(
      'Player $username not found. HTTP Status: ${response.statusCode}',
    );
  }
}
```

El metodo nos retorna un Future<Player> el cual no es mas que una forma de decirle al programa que mientras se realiza la petición y el procesamiento de los datos, este nos devolverá un objeto tipo **Player**. Nuestro metodo ademas tendra como argumento un **String**, el cual sera el "username" y la **keyword** (modificador) **async**.

Future es usado mayormente para bloques de código **asíncronos** y constantemente guarda un estado, el cual puede ser: "uncompleted "o "completed". Será "uncompleted" mientras no se termine la operación .Será "completed" cuando la operación asincronea sea exitosa. Puede guardar el valor que deseamos, en este caso **Player**, o guardar un error.

Cuando hablamos de procesos asíncronos (async) nos refermios a tareas dentro del programa que se irán ejecutando solapadamente(no confundirlo con "paralelismo" o "multi-threading"), evitando el bloqueo o largas esperas (como lo sería si fuera de manera secuencial). Es ampliamente usado para taréas de tipo espera/suspensión, como lo sería una peticion http.

```
static Future<Player> fetchSinglePlayer(String username) async {
```

Con esto claro, veremos cómo funciona por dentro. Definiremos la respuesta del servidor como un **final** y usaremos la **keyword** await para realizar la petición.

Await le indica al programa que bloquerá la ejecución del código mientras espera la respuesta. Sin embargo, este no bloqueara el hilo principal, por lo que otras tareas podran ejecutarse de manera asíncrona, como componentes de la UI u otras peticiones.

```
final response = await http.get(
  Uri.parse("https://api.chess.com/pub/player/$username"),
);
```

Como resultado de usarlo. Podremos obtener del metodo **http.get()** un objeto de tipo **Future<Response>**, el cual contedra el **statusCode** con el resultadao de la petición según la nomenclatura http de errores, y el **body** con el JSON sin decodificar.

Antes de descodificar el body, revisaremos que el statusCode haya sido exitoso (200), en caso contrario, soltaremos una excepción con el error.

```
if (response.statusCode == 200) {
  final data = jsonDecode(response.body);
  return Player.fromJson(data);
} else {
  throw Exception(
    'Player $username not found. HTTP Status: ${response.statusCode}',
  );
}
```

```
class Player
```

Ahora, analizaremos el método **fecthPlayer**, el cual recibe una lista de usuarios (List<strings>) y crea una petición por cada uno de estos.

```
static Future<List<Player>> fetchPlayers(List<String> playersUsername) async {
  final List<Future<Player>> fetchFutures = playersUsername.map((username) {
    return fetchSinglePlayer(username);
  }).toList();
  try {
    final List<Player> players = await Future.wait(fetchFutures);
    return players;
  } catch (e) {
    rethrow;
  }
}
```

Nuestra clase será estática, retornandonos una lista de jugadores **Future<List<Player>>**. Tendrá como argumento una lista de "usernames" y tendra un **keyword** de tipo **async** para evitar bloquear el hilo principal.

```
static Future<List<Player>> fetchPlayers(List<String> playersUsername) async {
```

Ahora, declaramos una lista de futuros **Players** y llamamos al metodo .map() de nuestra lista de usernames, para poder itearar por cada uno de los elementos, llamando al metodo **fetchSinglePlayer** y retornando una promesa (**Future<Player>**), permitiendo pasar al siguiente elemento.

```
final List<Future<Player>> fetchFutures = playersUsername.map((username) {
  return fetchSinglePlayer(username);
}).toList();
```

Por último, recolectaremos todos los resultados provenientes de las peticiones API que se iniciaron asíncronamente. Y en caso de que alguna petición falle, agregaremos un **rethrow** para que esta excepción siga subiendo a otras clases o metodos que la usen.

```
try {
  final List<Player> players = await Future.wait(fetchFutures);
  return players;
} catch (e) {
  rethrow;
}
```

Una vez hecho este modulo, ya estarémos listos para poder comenzar con el diseño de la interfaz.

3.2.3 Main

Este es el centro de nuestra aplicación, donde se comienzan todos los procesos subyacentes y la estructura de nuestra interfaz, veremos construir una interfaz responsiva y dinámica. En nuestro caso, separaremos algunos elementos importantes del módulo main, como los widgets, que explicaremos más adelante.

Comenzaremos importando los siguientes módulos y librerías.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_staggered_grid_view/flutter_staggered_grid_view.dart';

import 'package:chess_players/widgets/player_card.dart';
import 'package:chess_players/entities/player.dart';
import 'package:chess_players/api_services/get_players_service.dart';
import 'package:chess_players/pages/custom_player.dart';
import 'package:chess_players/pages/search_player.dart';
```

Empezamos llamando a la función **main**, la cual es el punto de partida para cualquier aplicación de Dart. Dentro de esta llamaremos al método **runApp,** un método nativo de flutter al cual le pasaremos como argumento nuestra clase **MyApp** como constante, que hereda de **StatelessWidget** y definiremos su constructor con el parámetro **key**, para poder identificar elementos dentro del árbol de widgets.

```dart
Run | Debug | Profile
void main() {
  runApp(const MyApp());
}

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

Una vez hecho esto, sobreescriviremos el metodo **Build**, donde pasaremos como argumento el BuildContext y nos retornará un Widget. Además definiremos los estilos y colores principales de nuestra aplicación.

El Widget que nos retornará será un **MediaQuery**, el cual nos permite consultar información sobre el tamaño y la configuración de la pantalla y del sistema operativo en el que se está ejecutando tu aplicación.

Es esencial para hacer que el diseño de tu aplicación sea **adaptable y responsivo** a diferentes dispositivos (teléfonos, tabletas, y escritorios).

```dart
@override
Widget build(BuildContext context) {
  const Color chessGreen = Color.fromARGB(255, 120, 186, 73);
  const Color darkBackground = Color.fromARGB(255, 34, 35, 32);
  return MediaQuery(
```

La configuración inicia al envolver el **MaterialApp** en un MediaQuery. Esta es una técnica de **sobrescritura de responsividad** crucial: MediaQuery.of(context).copyWith(textScaleFactor: 1.0) fuerza a la aplicación a **ignorar** la configuración de tamaño de texto del sistema operativo del usuario. Esto asegura que elementos críticos del diseño, como el texto en las tarjetas y el RatingBar, no se vean desproporcionadamente grandes en pantallas pequeñas, manteniendo la integridad del *layout*.

Dentro del MaterialApp, se establece el **ThemeData**, que define el lenguaje de diseño general:

- **Paleta de Colores Oscura**: La aplicación está configurada para el modo oscuro (brightness: Brightness.dark), utilizando un fondo principal (scaffoldBackgroundColor)

y superficies de tarjeta (cardColor y colorScheme.surface) de tonos oscuros (gris/negro).

- **Color Primario**: El acento principal (primaryColor y colorScheme.primary) es el verde ajedrez (chessGreen), utilizado para elementos interactivos y de enfoque.

La configuración asegura la consistencia de los principales *widgets* de Material Design:

- **AppBar**
- **Botones (elevatedButtonTheme)**
- **Esquema de Texto**

```
data: MediaQuery.of(context).copyWith(textScaleFactor: 1.0),
child: MaterialApp(
  title: 'Chess.com Players - Salesians de Sarrià 2526 ',
  theme: ThemeData(
    brightness: Brightness.dark,
    primaryColor: chessGreen,
    scaffoldBackgroundColor: darkBackground,
    cardColor: const Color.fromARGB(255, 45, 46, 43),
    colorScheme: ColorScheme.fromSeed(
      seedColor: chessGreen,
      brightness: Brightness.dark,
      background: darkBackground,
      surface: const Color.fromARGB(255, 45, 46, 43),
      onSurface: Colors.white,
      primary: chessGreen,
    ).copyWith(secondary: chessGreen), // ColorScheme.fromSeed
    appBarTheme: AppBarTheme(
      backgroundColor: const Color.fromARGB(255, 26, 20, 13),
      foregroundColor: Colors.white,
      elevation: 0,
    ), // AppBarTheme
    elevatedButtonTheme: ElevatedButtonThemeData(
      style: ElevatedButton.styleFrom(
        backgroundColor: chessGreen,
        foregroundColor: Colors.black,
        padding: const EdgeInsets.symmetric(horizontal: 24, vertical: 16),
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(8),
        ), // RoundedRectangleBorder
      ),
    ), // ElevatedButtonThemeData

    textTheme: Typography.whiteMountainView.apply(
      bodyColor: Colors.white,
    ),
    useMaterial3: true,
  ), // ThemeData
```

Por último definiremos la propiedad Home, con la clase **MyHomePage**, la cual hereda de la clase **StateFulWidget**.

La propiedad **home** es una de las propiedades más importantes de MaterialApp. Su función es definir el **widget que será la pantalla inicial o la primera ruta** que se mostrará cuando la aplicación se inicie.

```
        home: MyHomePage(),
      ), // MaterialApp
    ); // MediaQuery
  }
}

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

definiremos createState(), el cual es un método obligatorio que debe ser implementado por **todo** StatefulWidget. Es el único método abstracto que tienes que definir al heredar de StatefulWidget.

Su trabajo es crear y devolver la instancia de la clase que contendrá el estado mutable (que puede cambiar) para este *widget*.

```
@override
State<MyHomePage> createState() => _MyHomePageState();
class _MyHomePageState extends State<MyHomePage> {
```

Antes de sobreescribir el **Build**, definiremos previamente las constantes que usaremos, ademas de inizializar la lista de de usernames que tendremas nada mas entrar a la aplicación. Ademas, tendremos metodos adicionales que nos permitirán realizar interacciones útiles con nuestra aplicación (las explicaremos mas adelante).

```
class _MyHomePageState extends State<MyHomePage> {
  final String title = 'Chess.com Wiki Players - Damian Altamirano O.';
  final List<String> usernamesList = ['hikaru', 'magnuscarlsen'];
  late Future<List<Player>> playersFuture;
  List<Player> _players = [];

  void _handleCardTap(Player player) async {···

  void _handleFavoriteToggle(Player player) {···

  Color _getLeagueColor(String league) {···

  Future<List<Player>> _loadInitialPlayers() async {···
```

Empezaremos sobreescribiendo el metodo initState, el cual se lanza cuando la clase estado se inicia, llamando al metodo **_loadInitialPlayers**, el cual sera una función asíncrona que nos retornará un **Future**, el cual será una lista de players. Este método llama a la clase **PlayersApiService**, al metodo **fetchPlayers**, al cual le pasamos la lista de usuarios predeterminados que declaramos incialemente. Agregamos el **Keyword** await para evitar bloquear el hilo principal. Por último, hacemos una llamada al metodo **setState** para cambiar el estado de **_MyHomePageState**, he indicarle a la aplicación que refresque la información con los nuevos datos.

```dart
Future<List<Player>> _loadInitialPlayers() async {
  final players = await PlayersApiService.fetchPlayers(usernamesList);
  setState(() {
    _players = players;
  });
  return players;
}

@override
void initState() {
  super.initState();
  playersFuture = _loadInitialPlayers();
}
```

Sobreescribiremos el metodo build, para retornar un Scaffold y sentar la estructura de nuestra aplicación. Dentro de este, configuraremos los estilos básicos de los slots, como el appBar.

Ademas, configuraremos el botón **floatingActionButton** presente durante toda la HomePage, el cual al presionarlo, lanzara el evento onPressed. La función **onPressed** del FAB maneja todo el flujo de búsqueda de un nuevo jugador utilizando programación asíncrona:

La función se marca como async para poder usar await. **await Navigator.of(context).push(...)**: Inicia la navegación a la pantalla **PlayerSearchScreen**. El uso de await es crucial aquí porque hace que el programa **espere** hasta que el usuario regrese de la pantalla de búsqueda (ya sea tocando "Atrás" o añadiendo un jugador).

El resultado de esta navegación (el jugador encontrado o null) se almacena en final newPlayer. Si el usuario regresó y **newPlayer** no es null, entonces este actuailiza el estado del widget, añadiéndolo a la lista de **Players** y generando un objeto nuevo(card), y saltando un **snackBar** al usuario advirtiéndole del cambio hecho.

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      backgroundColor: const Color.fromARGB(255, 34, 27, 23),
      title: Text(title),
    ), // AppBar
    floatingActionButton: FloatingActionButton(
      onPressed: () async {
        final newPlayer = await Navigator.of(context).push(
          MaterialPageRoute(builder: (context) => const PlayerSearchScreen())
        );
        if (newPlayer != null && newPlayer is Player) {
          setState(() {
            _players.add(newPlayer);
          });
          ScaffoldMessenger.of(context).showSnackBar(
            SnackBar(content: Text('Jugador ${newPlayer.username} añadido!'))
          );
        }
      },
      child: const Icon(Icons.search),
      backgroundColor: Theme.of(context).primaryColor,
    ), // FloatingActionButton
```

Ahora pasaremos al **body** de nuestro Scaffold, la parte principal de nuestro programa. Para construir nuestras tarjetas interactivas de jugadores haremos uso de un **FutureBuilder<List<Player>>**.

El *widget* **FutureBuilder** es el mecanismo de Flutter para trabajar con datos que se cargan en el futuro (como tu lista de jugadores de la API). Su trabajo es decidir qué mostrar en la pantalla basándose en el estado de la promesa de datos (Future). Previamente en la propiedad **Future,** definiremos el objeto Future que debe monitorear. En nuestro caso, es playersFuture, ya que se inicializa en initState con la llamada _loadInitialPlayers().

Luego, dentro de la propiedad **builder**, pasaremos el context y el snapshot, el cual llama cada vez que el estado del Future cambia. El objeto snapshot contiene el estado actual de los datos (connectionState, hasError, data).

```
body: FutureBuilder<List<Player>>(
  future: playersFuture,
  builder: (context, snapshot) {
    if (snapshot.connectionState == ConnectionState.waiting) {
      return const Center(child: CircularProgressIndicator());
    }
    if (snapshot.hasError) {
      return Center(child: Text("Error: ${snapshot.error}"));
    }
```

Una vez que el estado del FutureBuilder este correcto, retornaremos el widget MasonryGridView, el cual es el *widget* que organiza nuestras tarjetas de jugador en una grilla dinámica de dos columnas. Con crossAxisCount: 2 definimos que la grilla tendrá dos columnas. Con itemCount: _players.length definimos que el número total de tarjetas a construir, basado en el tamaño de la lista local (_players) que fue inicializada por _loadInitialPlayers. El **itemBuilder** es la función que construye cada elemento de la grilla.

Aquí, la complejidad de la tarjeta se delega al *widget* PlayerCard. Se le pasan todos los datos y, fundamentalmente, se le pasan las **funciones de callback** (_handleCardTap, _handleFavoriteToggle) definidas en el estado (_MyHomePageState), permitiendo que el *widget* hijo interactúe con el estado del padre.

```
return MasonryGridView.count(
  padding: const EdgeInsets.all(12),
  crossAxisSpacing: 12,
  mainAxisSpacing: 12,
  crossAxisCount: 2,
  itemCount: _players.length,
  itemBuilder: (context, index) {
    final player = _players[index];

    return PlayerCard(
      player: player,
      onCardTap: _handleCardTap,
      onFavoriteToggle: _handleFavoriteToggle,
      cardColor: _getLeagueColor(player.league),
    ); // PlayerCard
  },
); // MasonryGridView.count
```

Los métodos a los que llama son estos.

```
void _handleCardTap(Player player) async {
  await Navigator.of(context).push(
    MaterialPageRoute(
      builder: (context) => PlayerDetailScreen(player: player),
    ), // MaterialPageRoute
  );
  setState(() {}); // Refresca la lista al volver
}

void _handleFavoriteToggle(Player player) {
  setState(() {
    player.isFavorite = !player.isFavorite;

    if (player.isFavorite) {
      player.followers += 1;
    } else {
      player.followers -= 1;
    }
  });
}
```

```
Color _getLeagueColor(String league) {
  // Aseguramos que la comparación no distinga mayúsculas y minúsculas
  switch (league.toLowerCase()) {
    case 'legend':
      // Color que simula el Oro/Leyenda (Amarillo intenso)
      return const Color.fromARGB(255, 183, 183, 183);
    case 'champion':
      // Color para Campeón (Rojo/Borgoña)
      return const Color.fromARGB(255, 207, 169, 0);
    case 'elite':
      // Color para Elite (Cian/Azul Brillante)
      return const Color.fromARGB(255, 154, 0, 0);
    case 'crystal':
      // Color para Cristal (Azul claro/Acuático)
      return Colors.blue.shade400;
    case 'silver':
      // Color para Plata (Gris claro/Plateado)
      return Colors.blueGrey.shade300;
    case 'bronze':
      // Color para Bronce (Marrón/Naranja oscuro)
      return const Color.fromARGB(255, 161, 94, 27);
    case 'stone':
      // Color para Piedra (Gris piedra oscuro)
      return Colors.grey.shade700;
    case 'wood':
      // Color para Madera (Marrón base)
      return Colors.brown.shade800;
    default:
      // Color por defecto si la liga no coincide, usando el cardColor original del tema.
      return Theme.of(context).cardColor;
  }
}
```

3.2.4 Pages

Una vez definido las funcionalidades principales del Main, echaremos un vistazo a las diferentes páginas de nuestra aplicación, y veremos como estas interactúan con nuestra lista de personas y la grillia en nuestra **HomePage**.

3.2.4.1 Search Player

Se nos direcciona a esta página cuando presionamos el floating button situado en el main. Esta página cumple la función de buscar nuevos jugardores y agregarlos a nuestra lista local.

Empezaremos importando las siguientes librerías, las cuales serán importantes para usar los objetos Player y los métodos para realizar las llamadas http:

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

import 'package:chess_players/api_services/get_players_service.dart';
import 'package:chess_players/entities/player.dart';
import 'package:chess_players/widgets/search_result_card.dart';
```

Crearemos una nueva clase que heredará de **StatefulWidget**. Dentro de esta crearemos un nuevo estado para nuestro widget mediante el método **createState()**.

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

  @override
  State<PlayerSearchScreen> createState() => _PlayerSearchScreenState();
}
```

Procederemos a crear nuestro estado heredándolo de la clase base **State<>**. Hecho esto definiremos las variables globales de la clase, así como las constantes.

```dart
class _PlayerSearchScreenState extends State<PlayerSearchScreen> {
  final TextEditingController _searchController = TextEditingController();
  Player? _foundPlayer;
  bool _isLoading = false;
  String? _errorMessage;
```

Ahora, declararemos la función Future<List<Player>> _loadInitialPlayers(), la cual es crucial para la pantalla principal, ya que se encarga de la **obtención inicial de datos** sin bloquear la aplicación. La función se marca como async para permitir el uso de await, que detiene temporalmente la ejecución local mientras se espera la respuesta de la red. Al encontrar la línea final players = await PlayersApiService.fetchPlayers(usernamesList), la función llama al servicio para obtener la lista de jugadores de la API. Gracias a que fetchPlayers utiliza Future.wait, esta solicitud se realiza de forma **concurrente**, minimizando el tiempo total de espera.

Una vez que todas las peticiones han finalizado y la lista de objetos Player está lista, la función procede a actualizar el estado de la aplicación. Esto se realiza dentro del método **setState(() { ... });**, que notifica al *framework* Flutter que los datos han cambiado. Dentro de este bloque, la lista de jugadores cargada se asigna a la variable mutable de estado (_players = players;). Al ejecutarse setState, Flutter programa la **reconstrucción** del *widget* MyHomePage. Cuando el método build se ejecuta de nuevo, el FutureBuilder ya ha resuelto su promesa y el MasonryGridView se puede construir utilizando la lista _players ya poblada, mostrando las tarjetas en pantalla. Finalmente, la función devuelve la lista de jugadores, confirmando el éxito de la operación al FutureBuilder.

```dart
Future<void> _searchPlayer(String username) async {
  if (username.isEmpty) {
    setState(() {
      _errorMessage = 'Insert an username';
    });
    return;
  }

  setState(() {
    _isLoading = true;
    _foundPlayer = null;
    _errorMessage = null;
  });

  try {
    final Player player = await PlayersApiService.fetchSinglePlayer(username);

    setState(() {
      _foundPlayer = player;
    });
  } catch (e) {
    setState(() {
      _errorMessage = 'Error: Player not found';
    });
  } finally {
    setState(() {
      _isLoading = false;
    });
  }
}
```

Además, definiremos un método que retornará un jugador encontrado para ser guardado:

```dart
void _addPlayerToList() {
  if (_foundPlayer != null) {
    Navigator.pop(context, _foundPlayer);
  }
}
```

Y además, crearemos un método que construirá un widget de tipo **TextField**, el cual llamará al método _serchPlayer() una vez demos a enter o pulsemos sobre el icono.

```dart
Widget _buildSearchField() {
  return TextField(
    controller: _searchController,
    decoration: InputDecoration(
      labelText: 'Chess.com User name',
      suffixIcon: _isLoading
          ? const Padding(
              padding: EdgeInsets.all(8.0),
              child: CircularProgressIndicator(strokeWidth: 2),
            ) // Padding
          : IconButton(
              icon: const Icon(Icons.send),
              onPressed: () => _searchPlayer(_searchController.text),
            ), // IconButton
      border: const OutlineInputBorder(),
    ), // InputDecoration
    onSubmitted: _searchPlayer,
  ); // TextField
}
```

Hecho esto, definiremos la función principal de nuestra clase. El método build comienza devolviendo un **Scaffold**, que proporciona la estructura fundamental de Material Design a la pantalla, incluyendo la **AppBar** superior. La barra de aplicaciones muestra el título "Search

Player" y utiliza el color de fondo configurado en el tema principal de la aplicación. El contenido principal de la pantalla se coloca dentro de la propiedad body.

El body está envuelto en un **SingleChildScrollView** con un padding uniforme. Esto es crucial para la robustez de la aplicación, ya que permite que el contenido se desplace verticalmente, **evitando errores de desbordamiento** (*overflow*) si el teclado virtual aparece o si hay resultados largos. Dentro, el *widget* **Column** organiza todos los elementos verticalmente, y la propiedad crossAxisAlignment: CrossAxisAlignment.center asegura que el campo de búsqueda, los mensajes y la tarjeta de resultados estén centrados horizontalmente en la pantalla.

La parte más importante del Column es la lógica condicional que muestra la información basada en el estado de la búsqueda:

1. **Campo de Búsqueda**: Se muestra primero, definido por el método auxiliar **_buildSearchField()**, que contiene el TextField donde el usuario introduce el nombre de usuario.

2. **Manejo de Errores**: Si la variable de estado **_errorMessage** no es nula (lo cual ocurre si la API falla o si el jugador no se encuentra), se muestra un mensaje de texto de retroalimentación en color rojo.

3. **Resultado Exitoso**: Si la variable de estado **_foundPlayer** contiene un objeto Player (indicando una búsqueda exitosa), se muestra el *widget* **SearchResultCard**. Esta tarjeta encapsula la visualización del jugador encontrado y proporciona un botón que, al presionar, ejecuta la función _addPlayerToList para devolver el jugador a la lista principal de la aplicación.

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Search Player'),
      backgroundColor: Theme.of(context).appBarTheme.backgroundColor,
    ), // AppBar
    body: SingleChildScrollView(
      padding: const EdgeInsets.all(16.0),
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.center,
        children: [
          _buildSearchField(),

          const SizedBox(height: 20),
          if (_errorMessage != null)
            Text(_errorMessage!, style: const TextStyle(color: Colors.red)),

          if (_foundPlayer != null)
            SearchResultCard(player: _foundPlayer!, onAdd: _addPlayerToList),
        ],
      ), // Column
    ), // SingleChildScrollView
  ); // Scaffold
}
```

3.2.4.2 Custom Player

En esta página, podremos customizar los datos de nuestros jugadores, darles una calificación y poder guardarlos localmente.

Antes de comenzar, importaremos las siguientes librerías:

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

import 'package:chess_players/entities/player.dart';
import 'package:chess_players/widgets/player_details_widgets.dart';
```

Definiremos una nueva clase de tipo **StatefulWidget** en Flutter, lo que significa que seremos capaces de mantener y gestionar datos que cambian con el tiempo (el estado), como los controladores de texto para la edición del nombre o el estado de edición del *rating*. Este *widget* recibe el objeto **player** (los datos del jugador a mostrar) a través de su constructor como una propiedad **inmutable** (final). El propósito del método @override createState() => _PlayerDetailScreenState(); es decirle a Flutter que, cada vez que se construya esta pantalla, debe crear una instancia del objeto **_PlayerDetailScreenState**, el cual se encarga de contener toda la lógica mutable (las variables _isEditingName, _currentRating, las funciones setState, etc.) y gestionar el ciclo de vida de la interfaz de usuario.

```
class PlayerDetailScreen extends StatefulWidget {
  final Player player;
  const PlayerDetailScreen({super.key, required this.player});

  @override
  State<PlayerDetailScreen> createState() => _PlayerDetailScreenState();
}
```

Ahora, la clase **_PlayerDetailScreenState** es la encargada de gestionar el estado (los datos mutables) y la lógica de la pantalla de detalle de un jugador. Se inicializa con variables de estado como **_currentRating** y controladores de texto (TextEditingController) para campos editables como **_notesController** (para notas personales), **_nameController**, **_usernameController** y **_statusController**.

Además, incluye variables booleanas (bool) como **_isEditingName** que funcionan como **"interruptores"** (o *toggles*) para determinar si un campo específico debe mostrarse como un campo de texto editable o como texto estático. Toda esta estructura permite que el usuario edite la información del jugador en la pantalla de forma interactiva y que esos cambios se guarden localmente al pulsar el botón "Guardar".

```
class _PlayerDetailScreenState extends State<PlayerDetailScreen> {
  late double _currentRating;
  late TextEditingController _notesController;

  // Controladores para la edición de Name, Username y Status
  late TextEditingController _nameController;
  late TextEditingController _usernameController;
  late TextEditingController _statusController;

  // Estados de edición para cada campo (toggle con el lápiz)
  bool _isEditingName = false;
  bool _isEditingUsername = false;
  bool _isEditingStatus = false;
```

El método initState se llama exactamente **una vez** cuando el objeto de estado (_PlayerDetailScreenState) se crea e inserta en el árbol de *widgets*. Su propósito principal es inicializar las variables de estado y los recursos que dependen del *widget* inmutable asociado

(widget.player). Dentro de este método, se inicializa el valor mutable **_currentRating** con el *rating* inicial del jugador.

Más importante aún, se inicializan todos los **TextEditingController** (_notesController, _nameController, _usernameController, etc.). Estos controladores son esenciales para gestionar y capturar los cambios en los TextField de la interfaz de usuario, cargando el texto inicial directamente desde las propiedades del objeto widget.player

```
@override
void initState() {
  super.initState();
  // Inicializar campos editables personalizados
  _currentRating = widget.player.rating;
  _notesController = TextEditingController(text: widget.player.notes ?? '');

  // Inicializar campos editables de la API
  _nameController = TextEditingController(text: widget.player.name);
  _usernameController = TextEditingController(text: widget.player.username);
  _statusController = TextEditingController(text: widget.player.status);
}
```

El método dispose también forma parte del ciclo de vida del estado, pero se llama justo antes de que el objeto de estado sea **permanentemente eliminado** del árbol de *widgets* y liberado de la memoria. Su función es realizar la **limpieza de recursos**.

En este caso, es absolutamente crucial llamar al método .dispose() en **todos** los objetos **TextEditingController** que fueron creados. Si no se llama a dispose(), estos controladores de texto seguirán existiendo en la memoria incluso después de que el usuario haya salido de la pantalla de detalle, lo que provocaría una **fuga de memoria** (*memory leak*) y degradaría el rendimiento de la aplicación con el tiempo.

```
@override
void dispose() {
  _notesController.dispose();
  _nameController.dispose();
  _usernameController.dispose();
  _statusController.dispose();
  super.dispose();
}
```

La función **_saveChanges()** se activa al pulsar el botón "Guardar" y es responsable de transferir todos los datos editados por el usuario desde los controladores de texto del estado (como _currentRating, _notesController, _nameController, _usernameController y _statusController) directamente a las propiedades del objeto **widget.player** (que es la entidad mutable en la memoria).

Después de actualizar el objeto de datos, el método realiza dos acciones de interfaz de usuario: primero, muestra una notificación emergente (SnackBar) utilizando **ScaffoldMessenger.of(context).showSnackBar** para confirmar al usuario que los cambios se han guardado localmente, y segundo, utiliza **Navigator.pop(context)** para cerrar la pantalla de detalle y regresar a la pantalla anterior (la lista principal), donde los cambios guardados serán visibles.

```
void _saveChanges() {
  const SnackBar(
    content: const Text('Changes saved in local!!'),
    duration: const Duration(milliseconds: 50),
  ); // SnackBar

  widget.player.rating = _currentRating;
  widget.player.notes = _notesController.text;

  widget.player.name = _nameController.text;
  widget.player.username = _usernameController.text;
  widget.player.status = _statusController.text;

  // 2. Notificación y navegación
  ScaffoldMessenger.of(
    context,
  ).showSnackBar(const SnackBar(content: Text('Changes saved in local!!')));
  Navigator.pop(context);
}
```

Por último, una función auxiliar diseñada para tomar una marca de tiempo (timestamp) en formato entero y convertirla a una cadena de fecha y hora legible por humanos.

```
String formatLastOnline(int timestamp) {
  if (timestamp == 0) return 'Never';
  final DateTime dateTime = DateTime.fromMillisecondsSinceEpoch(
    timestamp * 1000,
  );
  return DateFormat('dd/MM/yyyy HH:mm').format(dateTime);
}
```

Ahora, comenzamos con la estructura principal de nuestra página. Al construir nuestro **Scaffold**, definiremos un floating action button, el cual al ser pressionado llamara al método antes definido _saveChanges.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    floatingActionButton: FloatingActionButton.extended(
      onPressed: _saveChanges,
      backgroundColor: Theme.of(context).primaryColor,
      icon: const Icon(Icons.save, color: Colors.black),
      label: const Text('Save', style: TextStyle(color: Colors.black)),
    ), // FloatingActionButton.extended
```

Al definer Nuestro **body**, haremos uso de un CustomScrollView. Este *widget* proporciona un desplazamiento personalizado y es el contenedor necesario para albergar los *widgets* llamados **Sliver**. A diferencia de un ListView estándar, CustomScrollView permite combinar múltiples efectos de desplazamiento y tipos de listas en una sola área de desplazamiento. Slivers es la lista de *widgets* hijos que se colocan dentro de CustomScrollView.

```
body: CustomScrollView(
  slivers: <Widget>[
    buildDetailHeader(context, widget.player, _usernameController.text),

    SliverList(
      delegate: SliverChildListDelegate([
        Padding(
          padding: const EdgeInsets.all(16.0),
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: <Widget>[
```

De aquí en adelante, cederemos toda la lógica de los widgets al modulo **Player Details Widget**, con el cual, construiremos los diferentes widgets donde se verán los datos de los jugadores, además de poder editar alguno de ellos. Haciendo una mención puntual de estos, serían:

- EditableRating.

```
EditableRating(
  currentRating: _currentRating,
  onRatingUpdate: (rating) {
    setState(() {
      _currentRating = rating;
    });
  },
), // EditableRating
```

- EditableRow.

```
EditableRow(
  title: 'Full Name',
  controller: _nameController,
  isEditing: _isEditingName,
  onToggleEdit: () => setState(() {
    _isEditingName = !_isEditingName;
  }),
), // EditableRow
```

- StaticDetailRow.

```
StaticDetailRow(
  icon: Icons.person_pin,
  label: 'Player ID',
  value: widget.player.playerId.toString(),
), // StaticDetailRow
```

- EditableNotesField.

```
EditableNotesField(controller: _notesController),
```

3.2.5 Widgets

Uno de los módulos mas importantes de nuestro programa. Aquí se almacenan los diferentes widgets y sus lógicas para cada un de las páginas de nuestra aplicación.

3.2.5.1 Player Card

Este widget es usado para mostrar los diferentes jugadores, en un formato estilo Card, usado en el main.

Antes, importaremos las siguientes librerías.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_rating_bar/flutter_rating_bar.dart';
import 'package:chess_players/entities/player.dart';
```

Declararemos la clase PlayerCard, la cual es un **StatelessWidget**, lo que significa que su apariencia es inmutable y no gestiona su propio estado interno. Toda la información que necesita para construirse (el *widget* player, la función onCardTap, etc.) se pasa a través de su constructor. Este constructor está diseñado para ser eficiente, exigiendo la entidad **player** y las funciones de *callback* (**onCardTap** y **onFavoriteToggle**) como argumentos requeridos. La función de *callback* es un patrón de diseño clave que permite que el *widget* hijo (la tarjeta) le comunique eventos al *widget* padre (la lista MyHomePage) sin depender de su estado.

```dart
class PlayerCard extends StatelessWidget {
  final Player player;
  final Function(Player player) onCardTap;
  final Function(Player player) onFavoriteToggle;
  final Color cardColor;

  const PlayerCard({
    super.key,
    required this.player,
    required this.onCardTap,
    required this.onFavoriteToggle,
    required this.cardColor,
  });
```

El método build de la tarjeta comienza envolviendo el contenido en un **InkWell**, lo que proporciona el efecto visual de "toque" (ripple effect) al pulsar la tarjeta. Al tocar, se dispara el *callback* onCardTap, que en la aplicación principal se encarga de la navegación a la pantalla de detalle.

Dentro del InkWell se encuentra el **Card**, que proporciona el estilo visual (sombra, esquinas redondeadas de 16px) y utiliza el **cardColor** específico (determinado por la liga del jugador) pasado por el constructor. El contenido interno está organizado verticalmente por un **Column** centrado.

```dart
@override
Widget build(BuildContext context) {
  return InkWell(
    onTap: () => onCardTap(player),
    child: Card(
      elevation: 4,
      color: cardColor,
      shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(16)),
      child: Padding(
        padding: const EdgeInsets.all(8),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.center,
          children: [
```

El cuerpo de la tarjeta presenta los datos del jugador de forma concisa. En la parte superior, se encuentra el **CircleAvatar** que muestra la imagen de perfil del jugador, envuelto en un *widget* **Hero**. La propiedad tag del Hero (player-avatar-${player.username}) es crucial, ya que permite la **animación de transición fluida** de la imagen cuando el usuario navega a la pantalla de detalle.

```
children: [
  Hero(
    tag: 'player-avatar-${player.username}',
    child: CircleAvatar(
      radius: 45,
      backgroundImage: NetworkImage(player.avatar),
    ), // CircleAvatar
  ), // Hero
```

Debajo de la imagen, se muestran el **RatingBar** (configurado para ser estático ya que ignoreGestures: true) y las propiedades principales del jugador (nombre de usuario, estatus, y liga). En la parte inferior, el **IconButton** de favorito llama a la función onFavoriteToggle pasada por el padre al ser pulsado, permitiendo que la lógica de actualizar el contador de seguidores y cambiar el ícono del corazón se gestione en el estado superior (_MyHomePageState).

```
const SizedBox(height: 10),
RatingBar.builder(
  initialRating: player.rating,
  minRating: 1,
  direction: Axis.horizontal,
  allowHalfRating: true,
  ignoreGestures: true,
  itemCount: 5,
  itemSize: 18,
  itemPadding: const EdgeInsets.symmetric(horizontal: 2.0),
  itemBuilder: (context, _) =>
      const Icon(Icons.star, color: Colors.amber),
  onRatingUpdate: (rating) {},
), // RatingBar.builder
```

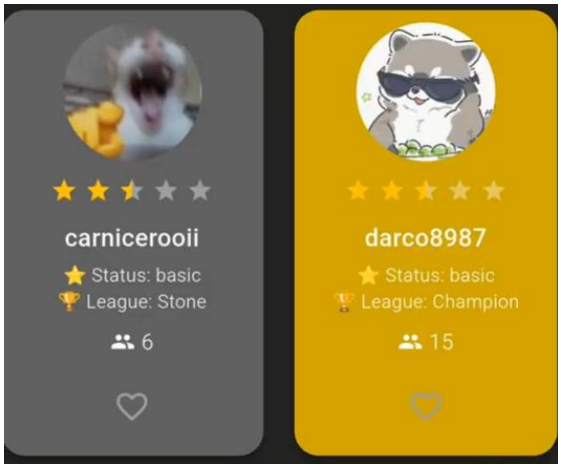
Mediante el objeto Text mostramos campos estaticos del jugador.

```
Text(
  player.username,
  style: const TextStyle(
    fontWeight: FontWeight.bold,
    fontSize: 16,
  ), // TextStyle
  textAlign: TextAlign.center,
), // Text

const SizedBox(height: 4),
Text(
  "⭐ Status: ${player.status}",
  style: const TextStyle(fontSize: 12),
), // Text
Text(
  "🏆 League: ${player.league}",
  style: const TextStyle(fontSize: 12),
), // Text
```

Y por último podremos ver un apartado de follors, donde podremos interactuar con este añadiendo un IconButton que aumente o disminuya la cantidad de followers.

```
const SizedBox(height: 8),
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: [
    const Icon(Icons.people, size: 16),
    const SizedBox(width: 4),
    Text(
      player.followers.toString(),
      style: const TextStyle(fontSize: 14),
    ), // Text
  ],
), // Row

const SizedBox(height: 8),
IconButton(
  icon: Icon(
    player.isFavorite ? Icons.favorite : Icons.favorite_border,
    color: player.isFavorite ? Colors.red : Colors.grey,
  ), // Icon
  onPressed: () => onFavoriteToggle(player),
), // IconButton
```

Hecho esto, podremos usar este widget para generar cards como estas:



3.2.5.2 Search Result Card

Este módulo es usado para la página de **Search Result,** este nos permitirá construir una tarjeta en base del resultado obtenido al buscar un usuario por el campo de texto.
Antes de comenzar importamos las siguientes librerías.

```
import 'package:flutter/material.dart';
import 'package:chess_players/entities/player.dart';
```

Comenzaremos definiendo la clase SearchResultCard, la cual es un **StatelessWidget** cuyo único propósito es mostrar la información de un jugador que ha sido encontrado exitosamente en la búsqueda de la API. Como *widget* sin estado, recibe toda la información que necesita a través de su constructor: la entidad **player** (con los datos a mostrar) y el *callback* **onAdd** (VoidCallback), que es la función que se ejecutará cuando el usuario decida añadir el jugador. Este diseño de *callback* permite que el *widget* padre (la pantalla de búsqueda) mantenga el control sobre la lógica de agregar a la lista principal.

```
class SearchResultCard extends StatelessWidget {
  final Player player;
  final VoidCallback onAdd;

  const SearchResultCard({
    super.key,
    required this.player,
    required this.onAdd,
  });
```

El método build de esta clase construye una **Card** que encapsula los detalles del jugador, añadiendo márgenes verticales y utilizando el color de superficie definido en el tema de la aplicación.

```
@override
Widget build(BuildContext context) {
  return Card(
    margin: const EdgeInsets.symmetric(vertical: 10),
    color: Theme.of(context).cardColor,
```

El contenido está centrado dentro de un **Column** (crossAxisAlignment: CrossAxisAlignment.center), lo que garantiza que todos los elementos (avatar, texto y botón) se presenten de manera uniforme. El Column utiliza mainAxisSize: MainAxisSize.min para ocupar solo el espacio vertical necesario, optimizando el *layout* dentro del SingleChildScrollView de la pantalla de búsqueda.

```
child: Padding(
  padding: const EdgeInsets.all(16.0),
  child: Column(
    crossAxisAlignment: CrossAxisAlignment.center,
    mainAxisSize: MainAxisSize.min,
```

La tarjeta muestra la información esencial del jugador: el **CircleAvatar** con la imagen de perfil, el **username** destacado, y detalles como el nombre completo y la liga del jugador. La funcionalidad principal es proporcionada por el **ElevatedButton.icon** con la etiqueta "Añadir a la Lista Principal". Este botón está directamente enlazado al *callback* **onAdd** pasado al constructor, lo que significa que al pulsarlo, se ejecuta la función _addPlayerToList() definida en la clase de estado superior, logrando así la adición del jugador y la navegación fuera de la pantalla de búsqueda.

```
children: [
  CircleAvatar(
    radius: 40,
    backgroundImage: NetworkImage(player.avatar),
  ), // CircleAvatar
  const SizedBox(height: 10),
  Text(
    player.username,
    style: Theme.of(context).textTheme.titleLarge,
  ), // Text
  Text('Nombre: ${player.name}'),
  Text('Liga: ${player.league}'),
  const SizedBox(height: 10),
  ElevatedButton.icon(
    onPressed: onAdd,
    icon: const Icon(Icons.add),
    label: const Text('Añadir a la Lista Principal'),
  ), // ElevatedButton.icon
```

Una vez hecho todo esto, podremos generar widgets de este estilo:

3.2.5.3 Player Details Widgets

Aquí se encuentran los diferentes widgets bases usados para crear campos editables de nuestra página **Custom Player**, la cual nos facilitara a la hora de crear nuestras pantallas de customización.

Empezamos importando las siguientes librerías:

```
import 'package:flutter/material.dart';
import 'package:flutter_rating_bar/flutter_rating_bar.dart';
import 'package:chess_players/entities/player.dart';
```

La función auxiliar buildDetailHeader es responsable de crear el encabezado dinámico de la pantalla, que es un **SliverAppBar**

```
// Se define como una función sep
SliverAppBar buildDetailHeader(
  BuildContext context,
  Player player,
  String username,
) {
```

Este *widget* se integra en el CustomScrollView para crear un efecto de paralaje y colapso. El encabezado está configurado para tener una altura expandida de 250px y la propiedad pinned: true asegura que la barra superior se mantenga visible (fijada) en la parte superior de la pantalla al desplazarse. Dentro del **FlexibleSpaceBar**, se define el fondo, que muestra el CircleAvatar del jugador, envuelto en un *widget* **Hero** con un tag único para la animación fluida desde la pantalla anterior. El título superpuesto muestra el nombre de usuario centrado, asegurando una experiencia visual pulida al colapsar.

```
  return SliverAppBar(
    expandedHeight: 250.0,
    pinned: true,
    backgroundColor: Theme.of(context).scaffoldBackgroundColor,
    flexibleSpace: FlexibleSpaceBar(
      centerTitle: true,
      title: Text(
        username,
        style: const TextStyle(
          shadows: [Shadow(blurRadius: 5.0, color: Colors.black)],
        ), // TextStyle
      ), // Text
      background: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Hero(
            tag: 'player-avatar-${player.username}',
            child: CircleAvatar(
              radius: 60,
              backgroundImage: NetworkImage(player.avatar),
            ), // CircleAvatar
          ), // Hero
          const SizedBox(height: 12),
        ],
      ), // Column
    ), // FlexibleSpaceBar
  ); // SliverAppBar
}
```

La clase EditableRating encapsula la funcionalidad de establecer una valoración personalizada con estrellas. Recibe el valor actual (currentRating) y un *callback* (onRatingUpdate) para comunicar los nuevos valores de vuelta al estado principal. El *widget* utiliza un **RatingBar.builder** y se centra horizontalmente, mostrando el valor numérico exacto junto a las estrellas. Cada vez que el usuario modifica la valoración, se ejecuta el *callback*

onRatingUpdate, que en el estado principal llama a setState para actualizar la variable_currentRating.

```
class EditableRating extends StatelessWidget {
  final double currentRating;
  final ValueChanged<double> onRatingUpdate;

  const EditableRating({
    super.key,
    required this.currentRating,
    required this.onRatingUpdate,
  });

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.symmetric(vertical: 8.0),
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.center,
        children: [
          Text('Rating', style: Theme.of(context).textTheme.bodySmall),
          const SizedBox(height: 8),
          Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              RatingBar.builder(
                initialRating: currentRating,
                minRating: 0,
                direction: Axis.horizontal,
                allowHalfRating: true,
                itemCount: 5,
                itemPadding: const EdgeInsets.symmetric(horizontal: 4.0),
                itemBuilder: (context, _) =>
                    const Icon(Icons.star, color: Colors.amber),
                onRatingUpdate: onRatingUpdate,
              ), // RatingBar.builder
              const SizedBox(width: 16),
              Text(
                currentRating.toStringAsFixed(1),
                style: Theme.of(context).textTheme.titleLarge,
```

La clase EditableRow es un componente de UI reutilizable diseñado para campos de texto editables (como Nombre, Usuario o Status) que deben alternar entre el modo de visualización y el modo de edición. Recibe un controlador de texto (TextEditingController), el estado booleano isEditing, y el *callback* onToggleEdit.

```
class EditableRow extends StatelessWidget {
  final String title;
  final TextEditingController controller;
  final bool isEditing;
  final VoidCallback onToggleEdit;

  const EditableRow({
    super.key,
    required this.title,
    required this.controller,
    required this.isEditing,
    required this.onToggleEdit,
  });
```

- **Lógica de Visualización**: El contenido principal utiliza un **Expanded** para ocupar el espacio disponible. Muestra el texto estático si isEditing es falso, o un TextField sin bordes si isEditing es verdadero.

```
@override
Widget build(BuildContext context) {
  const TextStyle textStyle = TextStyle(
    fontSize: 18,
    fontWeight: FontWeight.bold,
  );

  return Padding(
    padding: const EdgeInsets.only(bottom: 8.0),
    child: Row(
      crossAxisAlignment: CrossAxisAlignment.center,
      children: [
        Text(
          '$title: ',
          style: Theme.of(
            context,
          ).textTheme.bodySmall!.copyWith(fontSize: 14),
        ), // Text
        Expanded(
```

- **Toggle de Edición**: El IconButton en el extremo derecho muestra un lápiz (Icons.edit) o un *check* (Icons.check). Al pulsarlo, se ejecuta onToggleEdit, invirtiendo el valor de isEditing y haciendo que el *widget* padre llame a setState para cambiar dinámicamente el *widget* de visualización a edición (o viceversa).

```
Expanded(
  child: isEditing
      ? Padding(
          padding: const EdgeInsets.only(top: 8.0),
          child: TextField(
            controller: controller,
            decoration: const InputDecoration(
              border: InputBorder.none,
              isDense: true,
              contentPadding: EdgeInsets.zero,
            ), // InputDecoration
            style: textStyle.copyWith(fontWeight: FontWeight.normal),
          ), // TextField
        ) // Padding
      : Text(
          controller.text.isNotEmpty ? controller.text : 'N/A',
          style: textStyle.copyWith(fontWeight: FontWeight.normal),
          overflow: TextOverflow.ellipsis,
        ), // Text
), // Expanded
IconButton(
  icon: Icon(isEditing ? Icons.check : Icons.edit),
  color: isEditing ? Theme.of(context).primaryColor : Colors.grey,
  onPressed: onToggleEdit,
), // IconButton
```

La clase EditableNotesField proporciona un campo de texto grande dedicado a la entrada de notas personales y observaciones. Este *widget* envuelve un **TextField** configurado para ser multilínea (maxLines: 4) y utiliza un OutlineInputBorder para delimitar visualmente la caja de entrada. Está directamente conectado a un TextEditingController del estado principal,

permitiendo que el texto se capture de manera continua para guardarlo al pulsar el FAB principal.

```dart
class EditableNotesField extends StatelessWidget {
  final TextEditingController controller;

  const EditableNotesField({super.key, required this.controller});

  @override
  Widget build(BuildContext context) {
    return Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        Text(
          'Notes and Observations',
          style: Theme.of(context).textTheme.titleSmall,
        ), // Text
        const SizedBox(height: 10),
        TextField(
          controller: controller,
          decoration: const InputDecoration(
            hintText: 'Añade notas sobre el jugador...',
            border: OutlineInputBorder(),
            alignLabelWithHint: true,
          ), // InputDecoration
          maxLines: 4,
          keyboardType: TextInputType.multiline,
          style: Theme.of(context).textTheme.bodyLarge,
        ), // TextField
      ],
    ); // Column
  }
}
```

La clase StaticDetailRow se utiliza exclusivamente para presentar información que **no es editable** (datos crudos de la API, como ID, Followers o última conexión).

```dart
class StaticDetailRow extends StatelessWidget {
  final IconData icon;
  final String label;
  final String value;

  const StaticDetailRow({
    super.key,
    required this.icon,
    required this.label,
    required this.value,
  });
```

Recibe un icon, un label y el value de la información. Este *widget* organiza los elementos en una **Row**, con un ícono temático a la izquierda, y una Column que muestra el título del campo (label) en texto pequeño y el valor del dato (value) en texto negrita y más grande. Esto proporciona una presentación clara y consistente para la información estática del jugador.

```dart
@override
Widget build(BuildContext context) {
  return Padding(
    padding: const EdgeInsets.symmetric(vertical: 8.0),
    child: Row(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: <Widget>[
        Icon(icon, color: Theme.of(context).primaryColor, size: 24),
        const SizedBox(width: 16),
        Expanded(
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: <Widget>[
              Text(label, style: Theme.of(context).textTheme.bodySmall),
              Text(
                value.isNotEmpty && value != 'N/A' ? value : 'N/A',
                style: Theme.of(context).textTheme.titleMedium!.copyWith(
                  fontWeight: FontWeight.bold,
                ),
```

Recursos

Demos

Chess.com Players Demo

Enlace:

- Chess players App Demo

Librerías

http

- http | Dart package

flutter_form_builder

- flutter_form_builder | Flutter package

form_builder_extra_fields

- form_builder_extra_fields | Flutter package

flutter_staggered_grid_view

- flutter_staggered_grid_view | Flutter package

APIs

Chess.com API

- Published-Data API - Chess.com

Repositorio GitHub

Enlace:

- Mobile_projects_2526