BOOKNEST

DOCUMENTACIÓN DEL PROYECTO

José Martín Bellido

2º Desarrollo de Aplicaciones Multiplataforma

Programación Multimedia y Dispositivos Móviles

ÍNDICE

1. MANUAL DE USUARIO

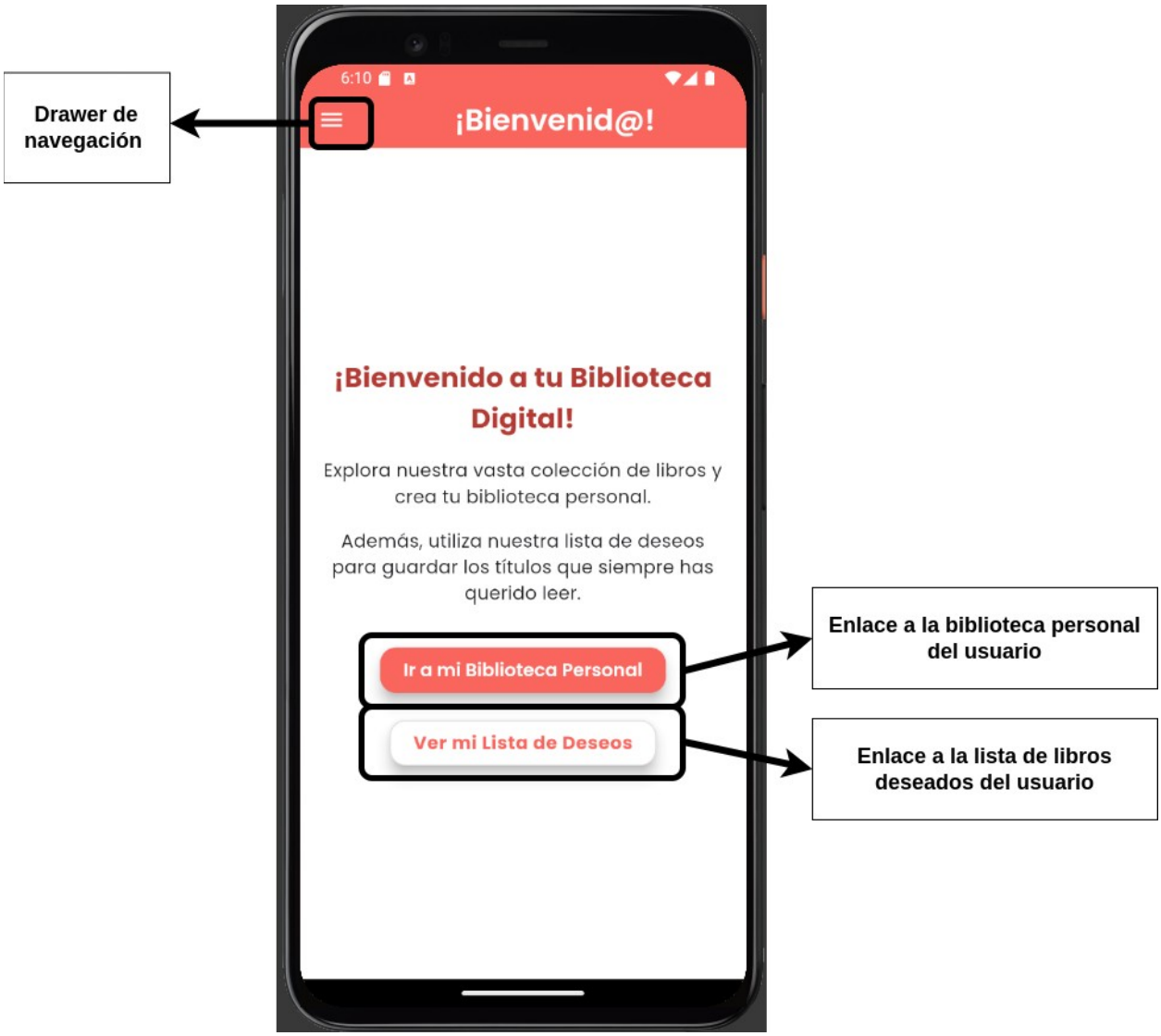
Se presenta a continuación el manual de uso para el usuario de la aplicación.

1.1 ENTRADA A LA APLICACIÓN

La entrada a la aplicación se basa en una página de bienvenida donde se explica a nivel general de qué trata ésta junto con dos enlaces de interés como acceso rápido al área personal del usuario.

El primero de ellos es su biblioteca personal, donde se almacenan los libros que el usuario tenga. El segundo es su lista de libros deseados (aquellos libros que no posea pero quiera tener o leer en un futuro).

Por último, en la parte superior se dispone de un drawer de navegación entre las distintas páginas de la aplicación.



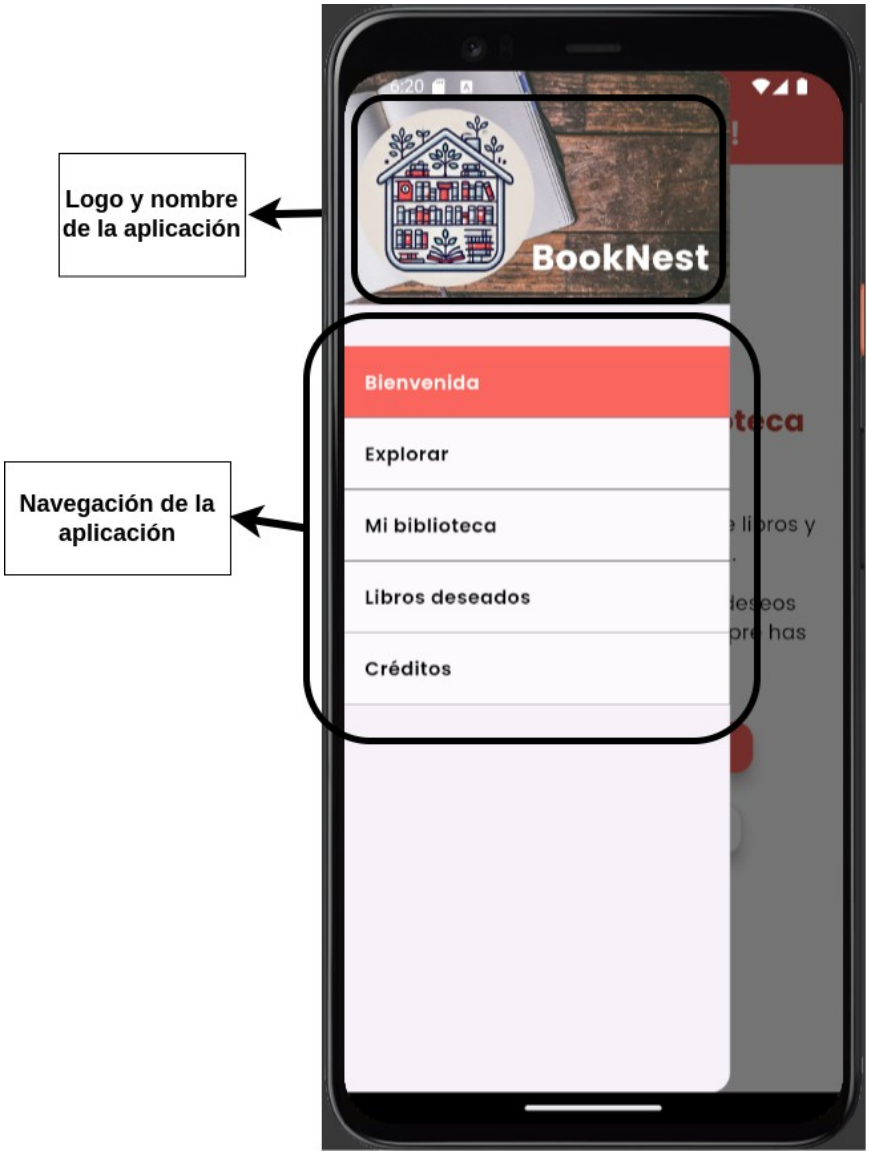
1.2 NAVEGACIÓN

La navegación en la aplicación se basa en un Widget denominado drawer.

En dicho drawer se dispone de los distintos enlaces de navegación, siendo aquel marcado la página en la que se encuentra actualmente el usuario.

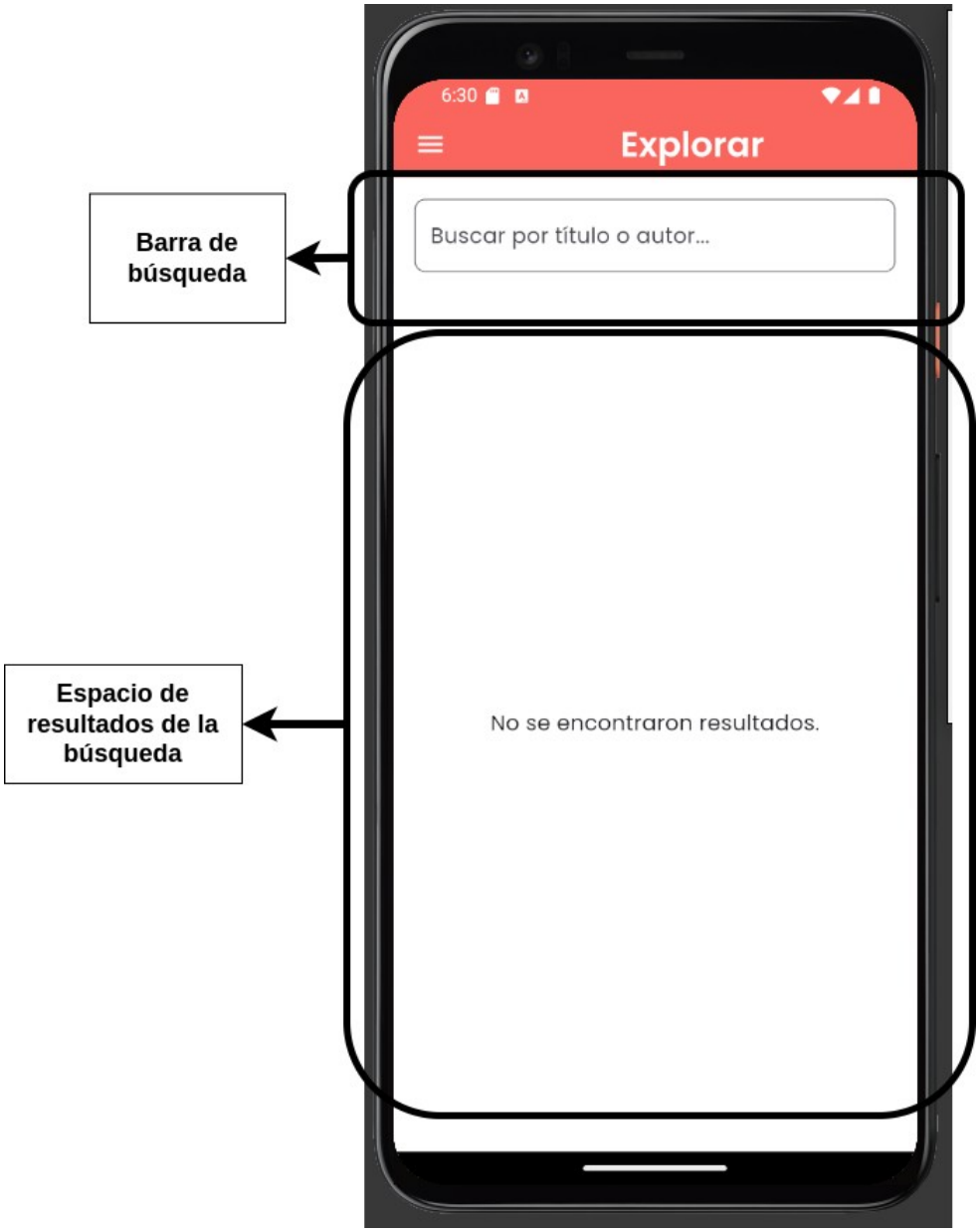
Se disponen de, principalmente, tres páginas funcionales:

- **Página de exploración**: Se trata de la página en la que se pueden buscar libros y desde la cual se pueden añadir a la biblioteca o a la lista de libros deseados.
- **Página de biblioteca**: Página en la cual se pueden ver los libros de la biblioteca del usuario y desde la cual, además, se pueden eliminar.
- **Página de libros deseados (wishlist)**: Página en la cual se pueden ver los libros que el usuario desea tener o leer, y desde la cual se pueden eliminar de la lista.

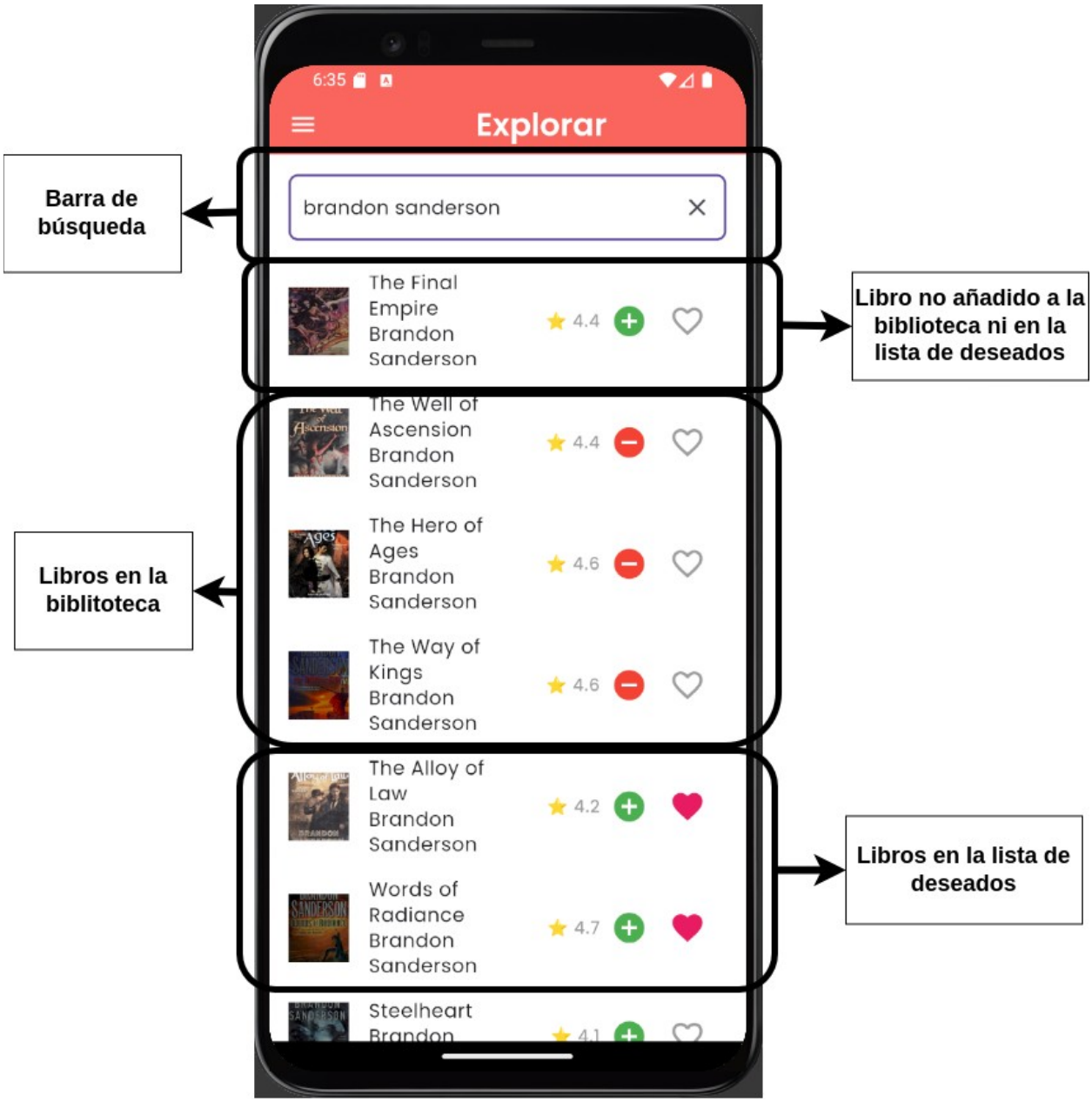


1.3 PÁGINA DE EXPLORACIÓN

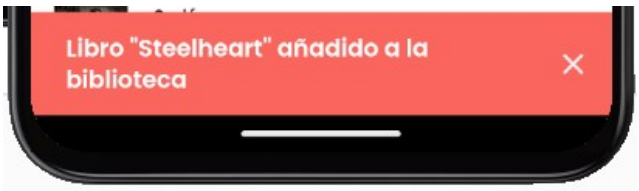
Inicialmente, la carga de la página de exploración contendrá únicamente un buscador (donde se podrán buscar tanto libros por título como por nombre de autor) y un mensaje de que actualmente no se han encontrado resultados. Esto se debe a que el usuario aún no ha hecho ninguna búsqueda.



Una vez se escriba en la barra de búsqueda, se activará una búsqueda automática y se presentarán los distintos resultados encontrados, disponiéndolos en una estructura denominada “scroll infinito”, según la cual se cargarán elementos conforme se descienda en la página. Además, cada libro mostrará su título, autor, puntuación media y los botones de interacción



Una vez se modifique el estado de un libro, es decir, tanto si se añade o elimina de la biblioteca como en caso de ser sobre la lista de deseados, se informará al usuario mediante un mensaje emergente en la parte inferior.



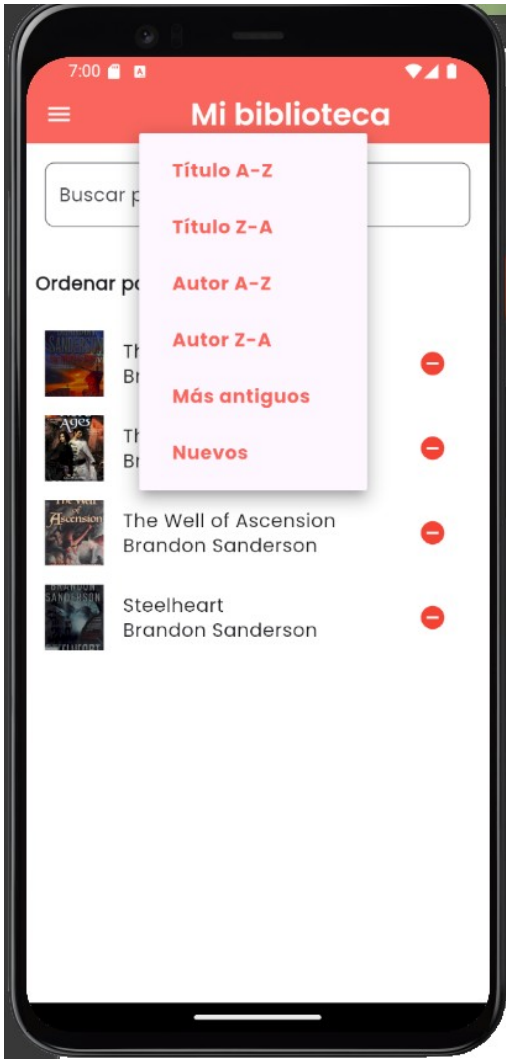
1.4 PÁGINA DE LA BIBLIOTECA

En la pestaña ‘Mi biblioteca’ se pueden observar todos aquellos libros añadidos previamente a la biblioteca personal. Además, se dispondrá de una barra de búsqueda como modo de filtrado por título o autor y un campo de ordenación donde los principales tipos serán:

- **Orden alfabético por Título (Ascendente o Descendente)**: Se aplica un orden alfabético sobre el título del libro, a elegir entre un orden ascendente (A-Z) o descendente (Z-A).
- **Orden alfabético por Autor (Ascendente o Descendente)**: Se aplica un orden alfabético sobre el autor o autores del libro, a elegir entre un orden ascendente (A-Z) o descendente (Z-A).
- **Nuevos primero**: Se establece un orden descendente en función del año de publicación de la primera edición del libro.
- **Más antiguos primero**: Se establece un orden ascendente en función del año de publicación de la primera edición del libro.

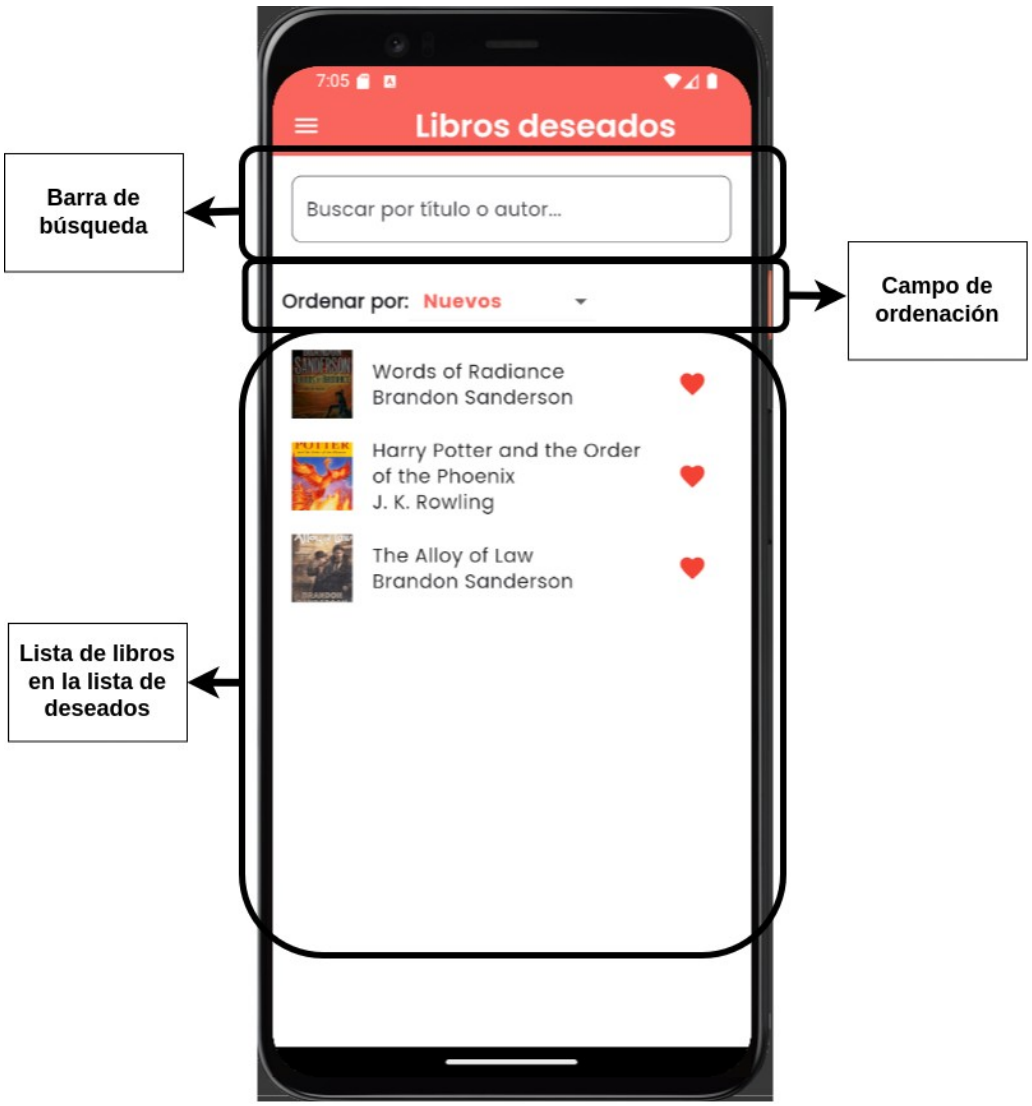
Cabe destacar la inclusión del botón para eliminar el libro de la biblioteca, el cual, al igual que en la página de exploración, avisará al usuario del cambio realizado.

A continuación se pueden ver dos capturas realizadas sobre la aplicación. La primera (izquierda) con el despliegue realizado sobre el campo de ordenación. La segunda, de forma que la búsqueda no produzca resultados.



1.5 PÁGINA DE LA LISTA DE DESEADOS

Debido a la funcionalidad de la lista de libros deseados, tanto la estructura de la página como su funcionalidad es extremadamente parecida a la anterior.



No se muestran capturas de la aplicación sobre los campos de ordenación ni sobre la búsqueda, ya que su funcionamiento es análogo.

1.6 PÁGINA DE CRÉDITOS

Se muestra a continuación la página de créditos de la aplicación, la cual se divide en dos secciones principalmente:

- **Créditos de la aplicación**: Se trata de la sección de créditos donde se hace referencia al desarrollador o desarrolladores de la aplicación, su información de contacto y una breve descripción de la aplicación.
- **Recursos usados**: Sección donde se describen tanto la API usada para la aplicación (para la obtención de datos sobre libros) y las bibliotecas usadas para el desarrollo.

2. DOCUMENTACIÓN TÉCNICA

Esta sección de la documentación estará dedicada a la parte técnica del proyecto, en la cual se explicará a rasgos generales el funcionamiento de la aplicación.

2.1 EXPLICACIÓN GENERAL

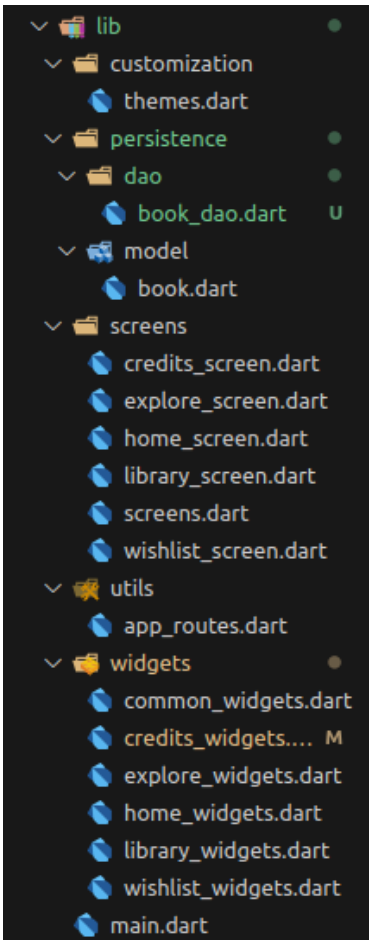
La aplicación BookNest se trata, en pocas palabras, de una aplicación desarrollada en **Flutter**. Dicha aplicación ha necesitado de:

- Una API externa de la cual se han extraído los datos acerca de los libros (**OpenLibrary**).
- Una serie de bibliotecas adaptadas a un uso desde flutter, entre las cuales cabe destacar **sqflite**, la cual está destinada al trabajo con bases de datos SQLite.

2.2 ESTRUCTURA DE CARPETAS

La estructura de carpetas del proyecto es la siguiente:

- **Customization**: Se trata de una carpeta que contendrá todas las clases de estilos y personalización de la aplicación.
- **Persistence**: Carpeta destinada a la persistencia de datos (modelos de datos, clases DAO, etc.)
- **Screens**: Clases con las estructuras principales de las distintas páginas de la aplicación.
- **Utils**: Carpeta con clases de utilidades.
- **Widgets**: Conjunto de clases con diferentes widgets, distribuidos por widgets comúnes o específicos de cada página.
- **Clase main**: Clase principal de la aplicación, la cual se ejecutará de inicio.



2.2.1 CLASE MAIN

La clase main será el punto de entrada a la aplicación, y desde el cual se hará referencia al resto de elementos. Será, por tanto, la entrada y el nexo de unión entre los estilos, la navegación y las distintas pantallas.

```dart
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  runApp(
    MaterialApp(
      title: 'BookNest',
      themeMode: ThemeMode.light,
      theme: customizedLightTheme,
      initialRoute: AppRoutes.home,
      routes: AppRoutes.routes,
      debugShowCheckedModeBanner: false,
    ),
  );
}
```

2.2.2 CUSTOMIZATION

Esta carpeta incluye únicamente los distintos temas a usar en la aplicación. En este caso, sólo se incluye un tema en modo claro, donde se detallan:

- Los **colores** básicos a usar en la aplicación.
- Los **estilos** a aplicar sobre los **scaffold**.
- Los **estilos** a aplicar sobre los distintos **botones**.
- **Tipo, color y tamaño** de las distintas **fuentes de letra**. En este caso, se ha elegido un tipo ***Poppins*** incluída a través de Google Fonts.

```dart
const mainColor = Color.fromARGB(255, 161, 189, 201);
const secondaryColor = Color.fromARGB(255, 250, 102, 94);
const darkSecondaryColor = Color.fromARGB(255, 182, 59, 53);

// Tema claro
final customizedLightTheme = ThemeData(

  // Básica
  brightness: Brightness.light,
  primaryColor: mainColor,
  secondaryHeaderColor: secondaryColor,
  dividerColor: Colors.black54,

  // Scaffold + AppBar
  scaffoldBackgroundColor: Colors.white,
  appBarTheme: const AppBarTheme(
    backgroundColor: secondaryColor,
    foregroundColor: Colors.white,
    iconTheme: IconThemeData(color: Colors.white)
  ), // AppBarTheme

  // ElevatedButton
  elevatedButtonTheme: ElevatedButtonThemeData(
    style: ElevatedButton.styleFrom(
      backgroundColor: Colors.white
    )
  ), // ElevatedButtonThemeData

  // FloatingActionButton
  floatingActionButtonTheme: const FloatingActionButtonThemeData(
    backgroundColor: Colors.white
  ), // FloatingActionButtonThemeData
```

```dart
    // Texto del cuerpo
    labelLarge: GoogleFonts.poppins(fontWeight: FontWeight.bold, fontSize: 18, color: secondaryColor),
    labelMedium: GoogleFonts.poppins(fontWeight: FontWeight.bold, fontSize: 16, color: secondaryColor),
    labelSmall: GoogleFonts.poppins(fontWeight: FontWeight.w500, fontSize: 14, color: secondaryColor),

    bodyLarge: GoogleFonts.poppins(fontSize: 18),
    bodyMedium: GoogleFonts.poppins(fontSize: 16),
    bodySmall: GoogleFonts.poppins(fontSize: 16),

  ), // TextTheme
```

2.2.3 PERSISTENCE

La carpeta persistence incluye dos subcarpetas principales.

2.2.3.1 Carpeta model

La carpeta model incluirá todos los modelos relacionados con base de datos. Se trata de clases con únicamente sus atributos, su constructor y métodos de conversión con formato JSON para su serialización.

En el caso relacionado con la aplicación, el único modelo usado es la entidad Book, la cual se almacenará en una base de datos SQLite con sus datos más característicos: ID de la portada, título, año de primera publicación, puntuación, nombre del autor o autores y un booleano que indica si pertenece a la lista de deseados o está en la biblioteca.

```dart
/// Clase representación de un libro del sistema
You, hace 7 horas | 1 author (You)
class Book {
  /// ID del libro
  int? id;

  /// ID de la portada del libro
  int? coverId;

  /// Título del libro
  String? title;

  /// Primer año de publicación
  int? firstPublishYear;

  /// Media de puntuaciones
  double? rating;

  /// Lista de nombres de autores
  List<String>? authorNames;

  /// Booleano que indica si el libro está en la lista de deseados
  bool? isInWishlist;

  /// Constructor
  Book(
    this.id,
    this.coverId,
    this.title,
    this.firstPublishYear,
    this.rating,
    this.authorNames,
    this.isInWishlist,
  );
```

```dart
  /// Método para obtener los datos de un libro a partir de un JSON.
  /// Puede provenir tanto de base de datos como de API, por lo que se deben diferenciar los casos
  ///
  /// Params:
  ///   - json (Map<String, dynamic>): Mapa que actúa como JSON
  ///
  /// Return:
  ///   - Book: Libro generado
  Book.fromJson(Map<String, dynamic> json) {…

  /// Método que convierte el objeto en un mapa para poder trabajar con base de datos
  ///
  /// Params:
  ///
  /// Return:
  ///   - Map<String, dynamic>: Mapa generado a partir del libro
  Map<String, dynamic> toDBMap() {…

  /// Método toString
  @override
  String toString() {…
}
```

2.2.3.2 Carpeta dao

Se trata de la carpeta que incluye las clases DAO (Data Access Object). Dicho de otro modo, serán las clases que definirán el acceso a base de datos para obtener y tratar las distintas entidades.

En el caso que se está tratando en el momento, se define un único dao denominado **BookDao**, con los distintos métodos CRUD de interacción. Además, dicho DAO se basará en un patrón Singleton, definiendo una única instancia para el tratamiento de datos.

```dart
/// DAO de acceso a la base de datos de libros
class BookDao{

  /// Instancia de la clase. Patrón Singleton
  static final BookDao instance = BookDao._init();

  /// Base de datos
  static Database? bookDatabase;

  /// Constructor de clase
  BookDao._init();

  /// Getter para la base de datos
  ///
  /// Params:
  ///
  /// Return:
  ///   - Future<Database>
  Future<Database> get database async {

    if (bookDatabase != null) return bookDatabase!;
    bookDatabase = await initDB('books.db');
    return bookDatabase!;
  }
```

Se define así los siguientes métodos:

```
17 >  /// Getter para la base de datos…
23 >  Future<Database> get database async {…
29
30 >  /// Método para inicializar la base de datos…
37 >  Future<Database> initDB(String filePath) async {…
47
48 >  /// Método que define la estructura de la base de datos…
56 >  Future<void> createDB(Database db, int version) async {…
70
71 >  /// Método para añadir un libro en la biblioteca…
80 >  Future<void> addBookToLibrary(Book book) async {…
101
102 >  /// Obtiene todos los libros de la base de datos…
108 >  Future<List<Book>> getAllBooks() async {…
115
116 >  /// Obtiene todos los libros de la lista de la biblioteca…
122 >  Future<List<Book>> getLibraryBooks() async {…
128
129 >  /// Obtiene todos los libros de la lista de deseados…
135 >  Future<List<Book>> getWishlistBooks() async {…
141
142 >  /// Añade o elimina un libro de la lista de deseados. En caso de no estar en el sistema
150 >  Future<void> updateWishlistStatus(Book book, bool isInWishlist) async {…
187
188 >  /// Elimina un libro de la base de datos. …
197 >  Future<void> deleteBook(String title, String author) async {…
206
207    /// Cierra la base de datos
208 >  Future<void> close() async {…
213
```

Será necesario el trabajo con los distintos métodos de forma asíncrona, por lo que todos los métodos tienen como resultado un objeto envoltorio o Wrapper **Future**, además de ir acompañados por las palabras clave **async** y **await**.

Se muestra a continuación el desarrollo de uno de ellos como ejemplo, en el que se puede observar su uso.

```dart
/// Método para añadir un libro en la biblioteca
/// En caso de existir previamente, significa que está en la lista de deseados, por lo que se cambia su estado
/// Si no existe, se inserta
///
/// Params:
///   - book (Book): Libro a guardar en la biblioteca
///
/// Return:
///   - void
Future<void> addBookToLibrary(Book book) async {
  final db = await instance.database;

  var books = await db.query(
    'books', where: 'title = ? AND authorNames = ?',
    whereArgs: [book.title, book.authorNames!.join(', ')]
  );

  // Si no está incluido el libro en base de datos, se inserta
  if (books.isEmpty) {
    await db.insert(
      'books',
      book.toDBMap(),
      conflictAlgorithm: ConflictAlgorithm.replace,
    );

  // Si ya lo está, se actualiza porque se presupone que pertenece a la lista de deseados
  } else {
    await db.update(
      'books',
      {'isInWishlist': 0},
      where: 'title = ? AND authorNames = ?',
      whereArgs: [book.title, book.authorNames!.join(', ')],
    );
  }
}
```

2.2.4 SCREENS

La carpeta screens se subdivide a su vez en dos tipos de clases

2.2.4.1 Clase con la estructura de una pantalla

Cada pantalla usada en esta aplicación se basará en un Scaffold, de forma que tanto la parte superior como el drawer quedarán definidos en los widgets comunes, y usará como cuerpo de la pantalla un widget personalizado.

Tanto la pantalla home como la de créditos serán definidas como widgets sin estado o Stateless, puesto que no se modificarán durante su uso, mientras que el resto serán widgets con estado o Stateful, ya que serán modificadas mediante barras de búsqueda, criterios de ordenación y/o botones de acción.

A continuación se muestra un ejemplo de pantalla donde se indica tanto el texto superior, el elemento del drawer en el que se definirá la pantalla y el cuerpo de ésta.

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

  @override
  State<Explore> createState() => _ExploreState();
}

You, anteayer | 1 author (You)
class _ExploreState extends State<Explore> {
  @override
  Widget build(BuildContext context) {
    return const Scaffold(
      appBar: HeaderBar(headerText: 'Explorar'),
      drawer: MenuDrawer(markedLink: AppRoutes.explore),
      body: ExploreBody(),
    ); // Scaffold
  }
```

2.2.4.2 Clase de exportaciones

Ésta será una clase específica sobre exportaciones de clases, indicada especialmente para widgets que necesiten todas o la mayoría de las pantallas de la aplicación, como es por ejemplo el drawer.

```dart
export 'credits_screen.dart';
export 'explore_screen.dart';
export 'home_screen.dart';
export 'library_screen.dart';
export 'wishlist_screen.dart';

export '../utils/app_routes.dart';
```

2.2.5 UTILS

Carpeta o paquete de utilidades de la aplicación. En este caso específico, será necesaria únicamente una clase de utilidades destinada a la conexión de rutas, puesto que la aplicación se conecta mediante rutas nombradas.

```dart
class AppRoutes {

  // Nombres de rutas
  static const String credits = "/credits";
  static const String explore = "/explore";
  static const String home = "/";
  static const String library = "/library";
  static const String wishlist = "/wishlist";


  // Mapas de rutas
  static final Map<String, WidgetBuilder> routes = {
    credits: (context) => const Credits(),
    explore: (context) => const Explore(),
    home: (context) => const MainApp(),
    library: (context) => const Library(),
    wishlist: (context) => const Wishlist()
  };
```

2.2.6 WIDGETS

Por último, se tratará la carpeta widgets. Se trata de la carpeta que incluye los widgets de la aplicación, los cuales definirán su comportamiento y cómo se visualiza el conjunto de cara al usuario.

Se explicará a continuación cada uno de ellos a nivel teórico pero sin profundizar en su funcionamiento línea a línea, puesto que para ello se puede visualizar el código de la aplicación, el cual está comentado para su mejor entendimiento y mantenimiento.

Cabe distinguir dentro de la carpeta widgets dos tipos de clases.

2.2.6.1 Widgets comunes

Una clase será la común, donde se incluirán elementos como la barra superior que muestra la página en la que se encuentra el usuario, el drawer junto con sus elementos o el selector de ordenación a usar en diferentes páginas.

```
 5   /// HeaderBar de todas las páginas
     You, anteayer | 1 author (You)
 6 > class HeaderBar extends StatelessWidget implements PreferredSizeWidget {…
30
31   /// Elemento de la lista del drawer
     You, hace 6 horas | 1 author (You)
32 > class InkedDrawerText extends StatelessWidget {…
130
131
132  /// Drawer personalizado de la aplicación
     You, hace 6 horas | 1 author (You)
133 > class MenuDrawer extends StatelessWidget {…
200
201  /// Widget para seleccionar el criterio de ordenación
     You, hace 6 horas | 1 author (You)
202 > class SortSelector extends StatelessWidget {…
```

2.2.6.2 Widgets de una pantalla

El resto de widgets de la carpeta estará distribuido en clases que hacen referencia a la pantalla para la cual sirven. Poniendo varios ejemplos clave dentro del código, se explicarán tanto los widgets de la página de exploración como los de la biblioteca personal.

La **página de exploración** se basa en los siguientes widgets:

```
 8   /// Clase que representa el cuerpo de la página de explorar
     You, anteayer | 1 author (You)
 9 > class ExploreBody extends StatefulWidget {…
17
     You, hace 9 horas | 1 author (You)
18 > class _ExploreBodyState extends State<ExploreBody> {…
310
311  /// Widget Barra de búsqueda
     You, anteayer | 1 author (You)
312 > class SearchBar extends StatelessWidget {…
352
353  /// Elemento de la lista que contendrá la información de un libro
     You, hace 9 horas | 1 author (You)
354 > class BookTile extends StatelessWidget {…
441
442  /// Lista de libros completa
     You, hace 9 horas | 1 author (You)
443 > class BookList extends StatelessWidget {…
501
```

De forma que se tiene un widget general para el cuerpo de la página (ExploreBody) y que hará llamadas sobre los widgets SearchBar (barra de búsqueda) y BookList (Componente que contendrá la lista de libros, la cual se compondrá de elementso BookTile).

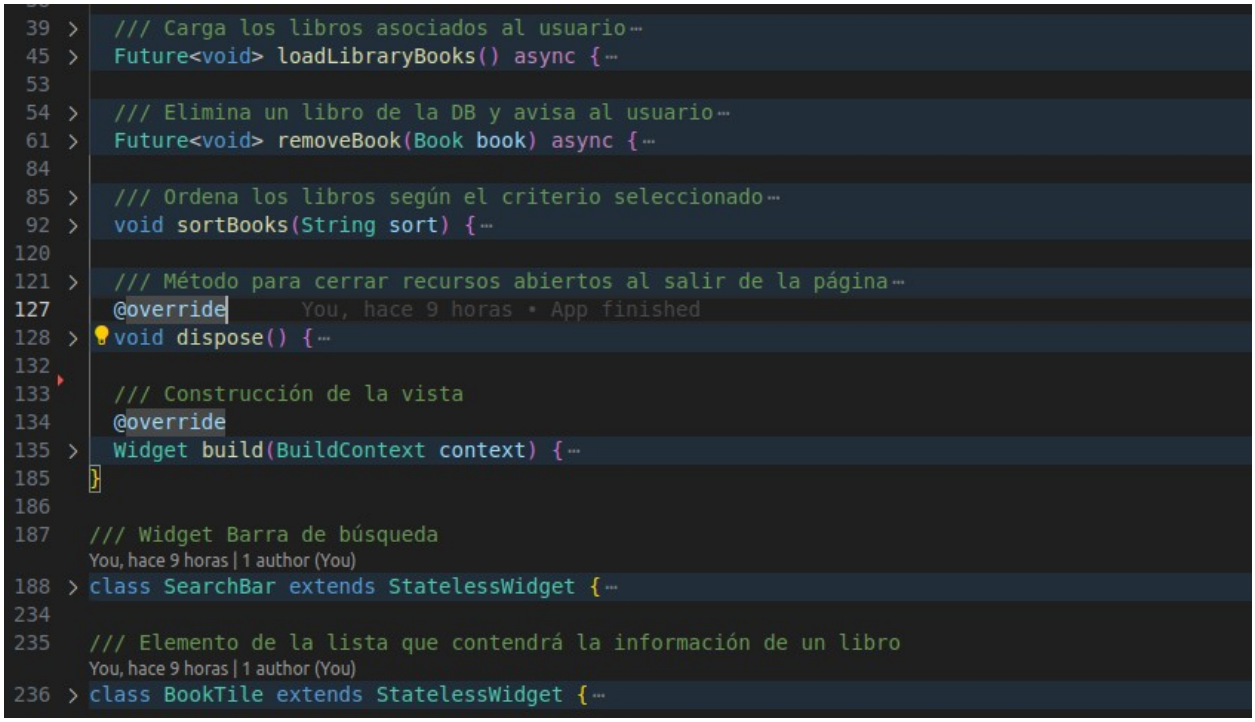
El funcionamiento se construye en cadena, transmitiendo por parámetros los atributos y métodos necesarios para el funcionamiento del widget general. Básicamente, se definen tres tipos de métodos desembocados por acción:

- Acción por pulsación de botones: Estos son los clicks sobre los métodos que interaccionan con la entidad Book, almacenando un nuevo libro en la biblioteca o eliminándolo, así como su acción análoga sobre la lista de libros deseados, a través de su DAO.
- Acción por escritura en la barra de búsqueda: Se define un temporizador que, tras 700ms, realiza una llamada a la API externa para traer los resultados necesarios, siguiendo la URL junto con los parámetros proporcionados por OpenLibrary. Este temporizador, además, necesita de un controlador específico denominado TextEditingController.
- Acción por movimiento de scroll: Se establece una acción para que, cuando se encuentre el scroll en la parte inferior de la pantalla, y en caso de existir más elementos a cargar, se realice una búsqueda sobre la API para obtener más resultados y mostrarlos al final de la página. Esta técnica de paginación es conocida como “scroll infinito”, y necesita de un controlador específico denominado ScrollController.

Teniendo desarrolladas estos tres conjuntos de métodos, así como ciertos métodos de utilidades, se construyen los widgets mostrados anteriormente.

Por otro lado, la **página de la biblioteca** se basa en los siguientes widgets:

```dart
/// Clase que representa el cuerpo de la página de la biblioteca
You, hace 9 horas | 1 author (You)
class LibraryBody extends StatefulWidget {…

You, hace 9 horas | 1 author (You)
class _LibraryBodyState extends State<LibraryBody> {

  /// Lista de libros de la biblioteca
  List<Book> libraryBooks = [];

  /// Lista de libros de la biblioteca mostrados al usuario
  List<Book> shownLibraryBooks = [];

  /// Controlador de búsqueda
  final TextEditingController searchController = TextEditingController();

  /// Tipo de ordenación seleccionada
  String sortOrder = 'year_desc';

  /// Inicializa el estado de la página. Añade los listeners sobre los controladores y carga los libros del usuario
  @override
  void initState() {…
```



Debido a que los widgets de la biblioteca son menos extensos, se ha dispuesto un resumen de métodos más ampliado que en el anterior ejemplo.

Al igual que antes, se necesita de un widget general que sea el cuerpo de la pantalla, la cual a su vez llama a la barra de búsqueda y a los elementos a mostrar. Los métodos necesarios, sin embargo, se reducen, puesto que se trabaja únicamente con los libros de la base de datos, es decir, con el DAO.

- Acción por pulsación de botones: Estos son los clicks sobre los métodos que interaccionan con la entidad Book. En este caso, se recurre a un único método de eliminación de libro y a un método que obtenga todos los libros.
- Acción por escritura en la barra de búsqueda: Se define la acción de forma que se trabaje localmente y no actuando sobre APIs ni sobre bases de datos. Se puede realizar de esta forma porque, tras cargar todos los libros inicialmente, se pueden filtrar y ordenar sobre una lista local de forma más eficiente.
- Acción por ordenación: Definido en los widgets comunes. Trabaja de forma local sobre la lista pasada como atributo, junto con el método de ordenación.

Aunque en esencia la metodología sea muy similar a la página de exploración, es necesario tener en cuenta que tanto esta página como la de la lista de libros deseados son mucho más eficientes que la de exploración, puesto que el rendimiento de la base de datos SQLite y el filtrado y ordenación en local es mucho mayor que una acción sobre una API externa.

3. ASPECTOS DE MEJORA

Evidentemente, esta aplicación no es más que una idea primaria y desarrollada en un tiempo muy acotado. Incluye aspectos como el trabajo simultáneo y compaginado entre dos fuentes de datos, siendo una la API externa y otra una base de datos local en fichero SQLite.

Existen diversos aspectos de mejora y ampliación:

- Podría añadirse un **sistema de puntuaciones** personal, de forma que agregue un nuevo tipo de ordenación sobre los libros. Además de esto, podrían agregarse comentarios personales acerca del libro. Se tratarían de comentarios personales para recordar ciertos aspectos a largo plazo, tanto positivos como negativos, a gusto del usuario.
- A largo plazo, podría implementarse una **base de datos en servidor (firebase, supabase, etc)** de forma que puedan verse bibliotecas de otras personas, así como sus opiniones acerca de otros libros.
- A consecuencia del punto anterior, un **sistema de amigos** combinaría perfectamente con el servidor de base de datos, de forma que se tuviese un acceso rápido a opiniones y puntuaciones de amigos.
- Debido al mismo motivo, podrían desarrollarse aspectos como **perfiles de usuario** donde se pueda contactar con otras personas para charlar sobre libros, o incluso implementar un **sistema de chat** dentro de la misma aplicación con este fin, llegando a generar un entorno seguro y controlado donde poder contactar con personas que tienen gustos afines en la lectura.

En cuanto a código, estas mejoras resultarían en una ampliación enorme, puesto que implicarían la **creación de nuevas entidades** como usuario y nuevas relaciones entre entidades como usuario – libro o usuario – usuario (amistad).

Esto, a su vez, llevaría a la necesidad de crear un DAO común abstracto y DAOs por cada entidad. Obviamente, por otra parte, el patrón Singleton dejaría de tener sentido como acceso único a base de datos, sino que serían necesario aplicar un patrón Factoría para tener un pool de conexiones.

Además, esta ampliación necesitaría de un rediseñado completo de la aplicación. Sería necesario redistribuir los espacios de los elementos para no sobrecargar la pantalla. A más elementos en pantalla, más difícil será que el usuario encuentre lo que necesite, llevándolo a una sensación de frustración que lo impulsaría a dejar de usar la aplicación.

La lógica interna de los widgets, al igual que en la construcción de elementos visuales, necesitaría de una reestructuración para evitar clases excesivamente grandes.

BIBLIOGRAFÍA

- https://docs.flutter.dev/
- https://e200.medium.com/flutter-infinite-scroll-with-rest-api-2b11f64b9d02
- https://www.scaler.com/topics/texteditingcontroller-flutter/
- Apuntes del temario de Programación Multimedia y Dispositivos Móviles, realizados por Francisco Miguel Fernández Banderas

  Ubicado en: https://moodle.iespablopicasso.es/
- Aplicación referencia: **GoodReads**