Instituto Politécnico Nacional

Escuela Superior De Computo

Desarrollo De Aplicaciones Móviles Nativas

Práctica 5

Consulta de Base de Datos vía APIs

Nombre Del Alumno:

García Quiroz Gustavo Ivan | 2022630278

Grupo: 7CV3

Nombre Del Profesor: Hurtado Avilés Gabriel

Fecha De Entrega: 19/04/2025

Índice

1 Introducción

La Práctica 5 "Consulta de Base de Datos vía APIs" tiene como objetivo principal desarrollar e integrar consultas avanzadas a bases de datos mediante APIs, añadiendo funcionalidades de búsqueda y recomendaciones en una aplicación móvil Android. Este proyecto permite adquirir experiencia en la implementación de sistemas que utilizan tanto bases de datos locales como servicios remotos, garantizando la funcionalidad continua incluso en situaciones sin conexión a internet.

En el desarrollo de esta práctica, se ha creado la aplicación "SystemBooks", un sistema de gestión y búsqueda de libros que se integra con la API de Open Library para obtener información bibliográfica actualizada. La aplicación permite a los usuarios buscar libros, autores, guardar favoritos y recibir recomendaciones personalizadas basadas en su historial de búsqueda y preferencias.

La arquitectura del proyecto se divide en dos componentes principales: un backend desarrollado con Spring Boot (SistemaRecomendacion2025-2) que gestiona la autenticación, usuarios y persistencia de datos en el servidor, y una aplicación móvil Android (SystemBooks) que implementa la interfaz de usuario y la lógica de consumo de APIs. Esta estructura permite una clara separación de responsabilidades y facilita el mantenimiento y escalabilidad del sistema.

2 Desarrollo

2.1 Ejercicio 1: Integración de Consultas a la Base de Datos vía API REST

2.1.1 Implementación de la conexión a la API REST

La aplicación móvil "SystemBooks" se conecta a dos APIs diferentes para su funcionamiento: la API REST propia desarrollada con Spring Boot (SistemaRecomendacion2025-2) y la API pública de Open Library. Esta arquitectura permite manejar tanto la autenticación y gestión de usuarios como la obtención de datos bibliográficos.

La implementación de la conexión a la API REST se realiza mediante las siguientes clases:

```java
public class ApiClient {
    // Cambiar la URL según corresponda a tu entorno:
    // Para emulador Android: "http://10.0.2.2:8088/"
    // Para dispositivo físico: "http://192.168.8.71:8088/"
    private static final String BASE_URL = "http://192.168.8.71:8088/"; // URL para el dispositivo

    private static Retrofit retrofit = null;
    private static AuthInterceptor authInterceptor;

    // Obtener cliente Retrofit sin autenticación (para login/registro)
    public static Retrofit getClient() {
        if (retrofit == null) {
            HttpLoggingInterceptor logging = new HttpLoggingInterceptor();
            logging.setLevel(HttpLoggingInterceptor.Level.BODY);

            OkHttpClient client = new OkHttpClient.Builder()
                    .addInterceptor(logging)
                    .connectTimeout(30, TimeUnit.SECONDS)  // Aumentar timeout
                    .readTimeout(30, TimeUnit.SECONDS)     // Aumentar timeout
                    .writeTimeout(30, TimeUnit.SECONDS)    // Aumentar timeout
                    .build();

            Gson gson = new GsonBuilder()
                    .setLenient() // Para manejo más flexible de JSON
                    .create();

            retrofit = new Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(LenientGsonConverterFactory.create(gson))
                    .client(client)
                    .build();
        }
        return retrofit;
    }
```

Figura 1 ApiClient.java - Configuración del cliente Retrofit

```java
public interface ApiService {
    // Autenticación
    @POST("api/auth/login")
    Call<LoginResponse> login(@Body LoginRequest loginRequest);

    @POST("api/auth/register")
    Call<ApiResponse<User>> register(@Body RegisterRequest registerRequest);

    // Gestión de usuarios (CRUD)
    @GET("api/users")
    Call<ApiResponse<List<User>>> getAllUsers();

    @GET("api/users/{id}")
    Call<ApiResponse<User>> getUserById(@Path("id") Long id);

    @PUT("api/users/{id}")
    Call<ApiResponse<User>> updateUser(@Path("id") Long id, @Body User user);

    @DELETE("api/users/{id}")
    Call<ApiResponse<Void>> deleteUser(@Path("id") Long id);

    // Gestión de imágenes de perfil
    @Multipart
    @POST("api/users/{id}/image")
    Call<ApiResponse<User>> uploadProfileImage(
            @Path("id") Long id,
            @Part MultipartBody.Part image
    );

    @GET("api/users/{id}/image")
    Call<ApiResponse<byte[]>> getProfileImage(@Path("id") Long id);
}
```

Figura 2 ApiService.java - Definición de endpoints

La clase ApiRepository sirve como intermediario entre la API y las actividades/fragmentos de la aplicación, proporcionando métodos para realizar operaciones con el servidor.

2.1.2 Persistencia de la sesión de usuario

La persistencia de la sesión del usuario se logra mediante la clase SessionManager, que utiliza SharedPreferences para almacenar el token JWT y la información del usuario:

```java
public class SessionManager {
    private static final String KEY_USER_NAME = "user_name";
    private static final String KEY_USER_EMAIL = "user_email";
    private static final String KEY_USER_ROLE = "user_role";
    private static final String KEY_USER_DATA = "user_data";
    private static final String KEY_IS_LOGGED_IN = "is_logged_in";

    private SharedPreferences sharedPreferences;
    private SharedPreferences.Editor editor;
    private Context context;

    public SessionManager(Context context) {
        this.context = context;
        sharedPreferences = context.getSharedPreferences(PREF_NAME, Context.MODE_PRIVATE);
        editor = sharedPreferences.edit();
    }

    /**
     * Guarda los datos de sesión del usuario
     */
    public void createLoginSession(String token, User user, String role) {
        editor.putString(KEY_AUTH_TOKEN, token);
        editor.putLong(KEY_USER_ID, user.getId());
        editor.putString(KEY_USER_NAME, user.getNombre());
        editor.putString(KEY_USER_EMAIL, user.getEmail());
        editor.putString(KEY_USER_ROLE, role);

        // Guardar el objeto User completo en formato JSON
        Gson gson = new Gson();
        String userJson = gson.toJson(user);
        editor.putString(KEY_USER_DATA, userJson);

        editor.putBoolean(KEY_IS_LOGGED_IN, true);
        editor.apply();

        Log.d(TAG, "Sesión de usuario creada para: " + user.getNombre());
    }
```

Figura 3 SessionManager.java

Esta implementación permite mantener la sesión del usuario activa entre inicios de la aplicación y mostrar la información correspondiente en la interfaz (nombre de usuario, rol) en elementos como el nav_header.xml.

2.1.3 Pruebas de conexión con Retrofit

Se utilizó Retrofit como cliente HTTP para conectarse a las APIs. Las pruebas de conexión se realizaron mediante la implementación de callbacks para manejar las respuestas y errores. Para interceptar las peticiones y añadir el token de autenticación, se implementó la clase AuthInterceptor:

```java
public class AuthInterceptor implements Interceptor {
    private String authToken;

    public AuthInterceptor(String token) {
        this.authToken = token;
    }

    @NonNull
    @Override
    public Response intercept(@NonNull Chain chain) throws IOException {
        Request originalRequest = chain.request();

        // Si no hay token, procede sin modificar la solicitud
        if (authToken == null || authToken.isEmpty()) {
            return chain.proceed(originalRequest);
        }

        // Añadir el token en el encabezado de la solicitud
        Request newRequest = originalRequest.newBuilder()
                .header("Authorization", "Bearer " + authToken)
                .build();

        return chain.proceed(newRequest);
    }

    // Permite actualizar el token si cambia durante la sesión
    public void setAuthToken(String token) {
        this.authToken = token;
    }
}
```

Figura 4 AuthInterceptor.java

2.1.4 Almacenamiento local con SQLite

La aplicación implementa una base de datos local mediante SQLite utilizando clases auxiliares como DatabaseHelper, BooksDatabaseHelper, y contratos como FavoritesContract y SearchHistoryContract:

```java
public class DatabaseHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "systembooks.db";
    private static final int DATABASE_VERSION = 1;

    // Table names
    public static final String TABLE_SEARCH_HISTORY = "search_history";
    public static final String TABLE_FAVORITES = "favorites";

    // Common column names
    public static final String COLUMN_ID = "id";
    public static final String COLUMN_USER_ID = "user_id";

    // Search history columns
    public static final String COLUMN_QUERY = "query";
    public static final String COLUMN_SEARCH_DATE = "search_date";

    // Favorites columns
    public static final String COLUMN_BOOK_ID = "book_id";
    public static final String COLUMN_TITLE = "title";
    public static final String COLUMN_AUTHOR = "author";
    public static final String COLUMN_COVER_URL = "cover_url";
    public static final String COLUMN_DATE_ADDED = "date_added";

    // Create table statements
    private static final String CREATE_TABLE_SEARCH_HISTORY =
            "CREATE TABLE " + TABLE_SEARCH_HISTORY + " (" +
                    COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    COLUMN_USER_ID + " INTEGER NOT NULL, " +
                    COLUMN_QUERY + " TEXT NOT NULL, " +
                    COLUMN_SEARCH_DATE + " INTEGER NOT NULL);";

    private static final String CREATE_TABLE_FAVORITES =
            "CREATE TABLE " + TABLE_FAVORITES + " (" +
                    COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    COLUMN_USER_ID + " INTEGER NOT NULL, " +
                    COLUMN_BOOK_ID + " TEXT NOT NULL, " +
                    COLUMN_TITLE + " TEXT NOT NULL, " +
```

Figura 5 DatabaseHelper.java

Los contratos definen la estructura de las tablas:

```java
public final class FavoritesContract {

    // To prevent someone from accidentally instantiating the contract class
    private FavoritesContract() {}

    // Table name
    public static final String TABLE_NAME = "favorites";

    // Column names
    public static class Columns implements BaseColumns {
        public static final String COLUMN_USER_ID = "user_id";
        public static final String COLUMN_BOOK_ID = "book_id";
        public static final String COLUMN_TITLE = "title";
        public static final String COLUMN_AUTHOR = "author";
        public static final String COLUMN_COVER_URL = "cover_url";
        public static final String COLUMN_TIMESTAMP = "timestamp";
    }

    // SQL to create the table
    public static final String CREATE_TABLE =
            "CREATE TABLE " + TABLE_NAME + " (" +
                    Columns._ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    Columns.COLUMN_USER_ID + " INTEGER NOT NULL, " +
                    Columns.COLUMN_BOOK_ID + " TEXT NOT NULL, " +
                    Columns.COLUMN_TITLE + " TEXT NOT NULL, " +
                    Columns.COLUMN_AUTHOR + " TEXT NOT NULL, " +
                    Columns.COLUMN_COVER_URL + " TEXT, " +
                    Columns.COLUMN_TIMESTAMP + " INTEGER NOT NULL, " +
                    "UNIQUE(" + Columns.COLUMN_USER_ID + ", " + Columns.COLUMN_BOOK_ID + "))";
}
```

```java
public final class SearchHistoryContract {

    // To prevent someone from accidentally instantiating the contract class
    private SearchHistoryContract() {}

    // Table name
    public static final String TABLE_NAME = "search_history";

    // Column names
    public static class Columns implements BaseColumns {
        public static final String COLUMN_USER_ID = "user_id";
        public static final String COLUMN_QUERY = "query";
        public static final String COLUMN_TIMESTAMP = "timestamp";
    }

    // SQL to create the table
    public static final String CREATE_TABLE =
            "CREATE TABLE " + TABLE_NAME + " (" +
                    Columns._ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    Columns.COLUMN_USER_ID + " INTEGER NOT NULL, " +
                    Columns.COLUMN_QUERY + " TEXT NOT NULL, " +
                    Columns.COLUMN_TIMESTAMP + " INTEGER NOT NULL)";
}
```

Figura 6 Tablas de favoritos y del historial.

2.1.5 Mecanismos de sincronización de datos

La sincronización entre la base de datos local y la API REST se implementa mediante repositorios que manejan tanto el almacenamiento local como las llamadas a la API.

Este enfoque garantiza que la aplicación funcione correctamente incluso sin conexión a internet, manteniendo la coherencia de los datos entre la base de datos local y el servidor cuando la conectividad está disponible.

Para optimizar el tráfico de red y mejorar la experiencia del usuario, se implementó un CacheInterceptor que permite el almacenamiento en caché de las respuestas HTTP.

Este sistema completo de persistencia y sincronización asegura que la aplicación pueda funcionar offline y mantener los datos actualizados cuando hay conexión disponible.

2.2 Ejercicio 2: Consumo de APIs Públicas

2.2.1 Descripción de la API seleccionada

La Open Library API es una interfaz de programación de aplicaciones que proporciona acceso a la extensa base de datos de libros de Open Library, un proyecto de Internet Archive. Esta API permite buscar y recuperar información detallada sobre millones de libros, autores y editoriales. Para esta práctica, se seleccionó esta API debido a su riqueza de datos bibliográficos, facilidad de uso, y su naturaleza gratuita y de código abierto.

Características principales de la Open Library API:

- **Acceso gratuito**: No requiere clave de API para consultas básicas.
- **Formato de respuesta**: Devuelve datos en formato JSON, fácilmente procesable por aplicaciones.
- **Endpoints principales**:
    - /search.json: Para búsqueda de libros por título, autor, etc.
    - /books/{id}.json: Para obtener detalles específicos de un libro.
    - /authors/{id}.json: Para obtener información sobre autores.

- **Variedad de datos**: Proporciona títulos, autores, fechas de publicación, ISBN, portadas, y más.

2.2.2 Implementación de la integración

La integración con la Open Library API se implementó mediante Retrofit, un cliente HTTP para Android que facilita la comunicación con servicios web RESTful. A continuación, se detalla la implementación:

```java
public interface OpenLibraryApi {

    // Buscar libros por consulta
    @GET("search.json")
    Call<SearchResponse> searchBooks(
            @Query("q") String query,
            @Query("page") int page,
            @Query("limit") int limit
    );

    // Obtener detalles de un libro por ID
    @GET("works/{workId}.json")
    Call<BookResponse> getBookDetails(
            @Path("workId") String workId
    );

    // Obtener libros por categoría/tema
    @GET("subjects/{subject}.json")
    Call<CategoryResponse> getBooksInCategory(
            @Path("subject") String subject,
            @Query("limit") int limit,
            @Query("offset") int offset
    );

    // Obtener libros destacados
    @GET("trending/daily.json")
    Call<SearchResponse> getTrendingBooks(
            @Query("limit") int limit
    );
}
```

Figura 7 OpenLibraryApi.java - Interface que define los endpoints de la API

2.2.3 Procesamiento de datos JSON

La respuesta de la API de Open Library viene en formato JSON, por lo que se definieron modelos de datos para facilitar su procesamiento:

```java
public class SearchResponse {

    @SerializedName("numFound")
    private int numFound;

    @SerializedName("start")
    private int start;

    @SerializedName("docs")
    private List<BookDoc> docs;

    public static class BookDoc {
        @SerializedName("key")
        private String key;

        @SerializedName("title")
        private String title;

        @SerializedName("author_name")
        private List<String> authorNames;

        @SerializedName("cover_i")
        private Long coverId;

        @SerializedName("first_publish_year")
        private Integer firstPublishYear;

        @SerializedName("publisher")
        private List<String> publishers;
```

Figura 8 SearchResponse.java - Modelo para respuestas de búsqueda

```java
public class BookResponse {

    @SerializedName("key")
    private String key;

    @SerializedName("title")
    private String title;

    @SerializedName("description")
    private Description description;

    @SerializedName("covers")
    private List<Long> covers;

    @SerializedName("authors")
    private List<Author> authors;

    @SerializedName("publish_date")
    private String publishDate;

    @SerializedName("publishers")
    private List<Publisher> publishers;

    @SerializedName("number_of_pages")
    private Integer numberOfPages;
```

Figura 9 BookResponse.java - Modelo para la información de un libro

2.2.4 Funcionalidades de búsqueda de libros y autores

La aplicación SystemBooks implementa tres tipos principales de búsqueda:

1. **Búsqueda general de libros**: Permite buscar libros por título, autor o cualquier término.
2. **Búsqueda por autor**: Facilita encontrar libros escritos por un autor específico.
3. **Exploración por categorías**: Permite navegar por libros clasificados en diferentes categorías.

Estas funcionalidades se implementan en SearchFragment.java:

```java
public class SearchFragment extends Fragment {
    private void performSearch(String query) {
        if (!NetworkUtils.isNetworkAvailable(requireContext())) {
            showErrorView(getString(R.string.error_network));
            return;
        }

        showLoadingView();

        bookRepository.searchBooks(query, 1, 20, new BookRepository.BookCallback<List<Book>>() {
            @Override
            public void onSuccess(List<Book> result) {
                swipeRefreshLayout.setRefreshing(false);

                // Guardar la búsqueda en el historial si el usuario está logueado
                if (sessionManager.isLoggedIn()) {
                    // Ejecutar en un hilo separado para no bloquear la UI
                    new Thread(() -> {
                        long userId = sessionManager.getUserId();
                        searchHistoryRepository.saveSearchQuery(userId, query);
                    }).start();
                }

                if (result.isEmpty()) {
                    showEmptyView();
                } else {
                    bookAdapter.updateBooks(result);
                    showResultsView();
                }
            }

            @Override
            public void onError(String message) {
                swipeRefreshLayout.setRefreshing(false);
                showErrorView(message);
            }
        });
```

Figura 10 SearchFragment.java - Fragmento para búsqueda de libros

2.2.5 Sistema de recomendaciones personalizadas

El sistema de recomendaciones se implementa analizando el historial de búsquedas y favoritos del usuario para sugerir libros similares:

```java
public class RecommendationEngine {
    */
    private void generateRecommendations(long userId, BookRepository.BookCallback<List<Book>> callback) {
        // Step 1: Get user's favorites
        List<FavoriteBook> favorites = favoritesRepository.getFavorites(userId);

        // Step 2: Get user's recent search history
        List<SearchHistoryItem> searchHistory = searchHistoryRepository.getSearchHistory(userId);

        // If user has no favorites or search history, return featured books
        if ((favorites == null || favorites.isEmpty()) && (searchHistory == null || searchHistory.isEmpty())) {
            Log.d(TAG, "User has no favorites or search history, returning featured books");
            bookRepository.getFeaturedBooks(MAX_RECOMMENDATIONS, callback);
            return;
        }

        // Step 3: Extract relevant keywords for recommendation (from titles, authors, search queries)
        Set<String> keywords = extractKeywords(favorites, searchHistory);

        if (keywords.isEmpty()) {
            Log.d(TAG, "No keywords extracted, returning featured books");
            bookRepository.getFeaturedBooks(MAX_RECOMMENDATIONS, callback);
            return;
        }

        // Step 4: Use extracted keywords to search for recommendations
        searchRecommendations(keywords, callback);
    }
```

Figura 11 RecommendationEngine.java - Sistema de recomendaciones

El sistema de recomendaciones implementado proporciona sugerencias personalizadas basadas en:

1. **Autores favoritos**: Recomienda libros de autores que el usuario ha guardado en favoritos.
2. **Historial de búsqueda**: Utiliza las búsquedas recientes para sugerir libros relacionados.
3. **Categorías populares**: Como respaldo, ofrece recomendaciones de categorías populares.

Esta implementación garantiza que el usuario siempre reciba recomendaciones relevantes, incluso si es nuevo en la aplicación o tiene poca actividad registrada.

Además, el sistema de recomendaciones funciona de manera eficiente tanto online como offline, aprovechando la base de datos local para proporcionar sugerencias incluso sin conexión a internet.

2.3 Ejercicio 3: Funcionalidades de Búsqueda, Favoritos y Recomendaciones

2.3.1 Historial de búsquedas

El sistema implementa un historial de búsquedas que permite a los usuarios acceder rápidamente a sus consultas anteriores. Esta funcionalidad se gestiona a través de la clase SearchHistoryRepository que interactúa tanto con la base de datos local como con el servidor:

```
public class SearchHistoryRepository {
    public List<SearchHistoryItem> getSearchHistory(long userId) {
        List<SearchHistoryItem> historyItems = new ArrayList<>();
        SQLiteDatabase db = dbHelper.getReadableDatabase();

        String selection = DatabaseHelper.COLUMN_USER_ID + " = ?";
        String[] selectionArgs = {String.valueOf(userId)};
        String orderBy = DatabaseHelper.COLUMN_SEARCH_DATE + " DESC";

        Cursor cursor = db.query(
                DatabaseHelper.TABLE_SEARCH_HISTORY,
                null,
                selection,
                selectionArgs,
                null,
                null,
                orderBy
        );

        try {
            int idColumnIndex = cursor.getColumnIndex(DatabaseHelper.COLUMN_ID);
            int userIdColumnIndex = cursor.getColumnIndex(DatabaseHelper.COLUMN_USER_ID);
            int queryColumnIndex = cursor.getColumnIndex(DatabaseHelper.COLUMN_QUERY);
            int dateColumnIndex = cursor.getColumnIndex(DatabaseHelper.COLUMN_SEARCH_DATE);

            while (cursor.moveToNext()) {
                long id = cursor.getLong(idColumnIndex);
                long historyUserId = cursor.getLong(userIdColumnIndex);
                String query = cursor.getString(queryColumnIndex);
                long timestamp = cursor.getLong(dateColumnIndex);

                SearchHistoryItem item = new SearchHistoryItem(
                        id, historyUserId, query, new Date(timestamp)
                );
                historyItems.add(item);
            }
```

Figura 12

Figura 13 SearchHistoryRepository.java

La visualización del historial se realiza a través del fragmento SearchHistoryFragment, que utiliza el adaptador SearchHistoryAdapter para mostrar las búsquedas previas. Este adaptador permite al usuario repetir búsquedas anteriores con un simple clic.

2.3.2 Sistema de favoritos

Perspectiva del usuario

El sistema de favoritos permite a los usuarios guardar los libros que les interesan para acceder rápidamente a ellos en el futuro. Los usuarios pueden:

- Añadir libros a favoritos desde la vista de detalles del libro o desde los resultados de búsqueda
- Ver su lista de favoritos en el fragmento FavoritesFragment
- Eliminar elementos de la lista de favoritos

La implementación del fragmento de favoritos muestra los libros guardados y permite interacciones:

```
public class FavoritesFragment extends Fragment {

    private void loadFavoriteBooks() {
        if (!sessionManager.isLoggedIn()) {
            showEmptyView();
            return;
        }

        showLoading();

        new Thread(() -> {
            final List<FavoriteBook> favorites = favoritesRepository.getFavorites(sessionManager.getUserId());

            // Convert FavoriteBook objects to Book objects
            final List<Book> favoriteBooks = new ArrayList<>();
            for (FavoriteBook favorite : favorites) {
                Book book = new Book(favorite.getBookId(), favorite.getTitle(), favorite.getAuthor());
                book.setCoverUrl(favorite.getCoverUrl());
                favoriteBooks.add(book);
            }

            if (getActivity() != null) {
                getActivity().runOnUiThread(() -> {
                    if (favoriteBooks != null && !favoriteBooks.isEmpty()) {
                        adapter.updateBooks(favoriteBooks);
                        showContent();
                    } else {
                        showEmptyView();
                    }
                });
            }
        }).start();
    }
```

Figura 14 FavoritesFragment.java

Perspectiva del administrador

Los administradores tienen acceso a funcionalidades adicionales a través del fragmento UserManagementFragment, donde pueden:

- Ver la lista de todos los usuarios
- Acceder a los favoritos de cada usuario
- Administrar los favoritos de los usuarios

La implementación permite a los administradores gestionar los datos y obtener información sobre los intereses de los usuarios.

2.3.3 Algoritmo de recomendaciones

El sistema implementa un algoritmo de recomendaciones basado en los favoritos del usuario y su historial de búsquedas. Este algoritmo analiza patrones en las preferencias del usuario para sugerir nuevos libros que podrían interesarle:

```
public class RecommendationEngine {
     */
    private void generateRecommendations(long userId, BookRepository.BookCallback<List<Book>> callback) {
        // Step 1: Get user's favorites
        List<FavoriteBook> favorites = favoritesRepository.getFavorites(userId);

        // Step 2: Get user's recent search history
        List<SearchHistoryItem> searchHistory = searchHistoryRepository.getSearchHistory(userId);

        // If user has no favorites or search history, return featured books
        if ((favorites == null || favorites.isEmpty()) && (searchHistory == null || searchHistory.isEmpty())) {
            Log.d(TAG, "User has no favorites or search history, returning featured books");
            bookRepository.getFeaturedBooks(MAX_RECOMMENDATIONS, callback);
            return;
        }

        // Step 3: Extract relevant keywords for recommendation (from titles, authors, search queries)
        Set<String> keywords = extractKeywords(favorites, searchHistory);

        if (keywords.isEmpty()) {
            Log.d(TAG, "No keywords extracted, returning featured books");
            bookRepository.getFeaturedBooks(MAX_RECOMMENDATIONS, callback);
            return;
        }

        // Step 4: Use extracted keywords to search for recommendations
        searchRecommendations(keywords, callback);
    }
```

Figura 15 Algoritmo de recomendaciones

3 Capturas de Pantalla

3.1 Inicio de Sesión

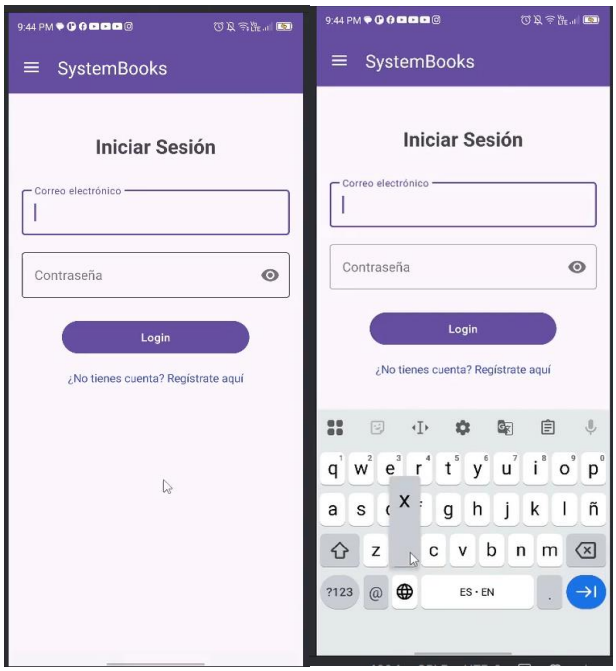


Figura 16 Pantalla de Inicio de Sesión.

3.2 Registro

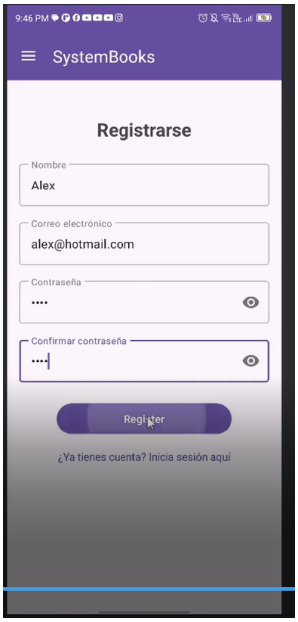


Figura 17 Pantalla de Registro

3.3 Perfil de Usuario

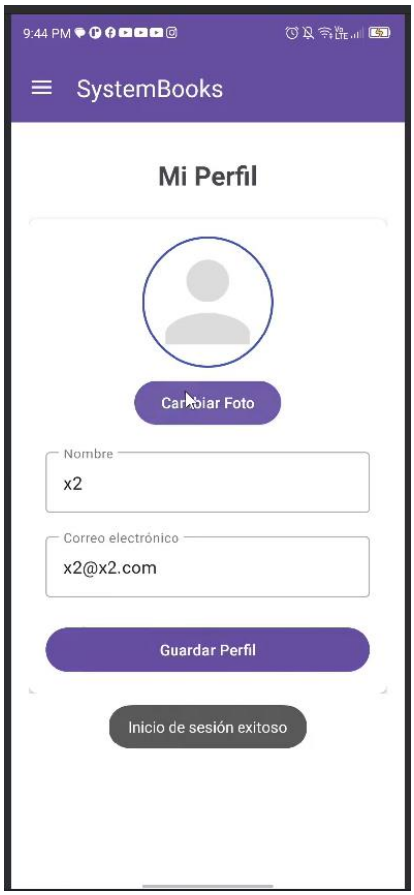
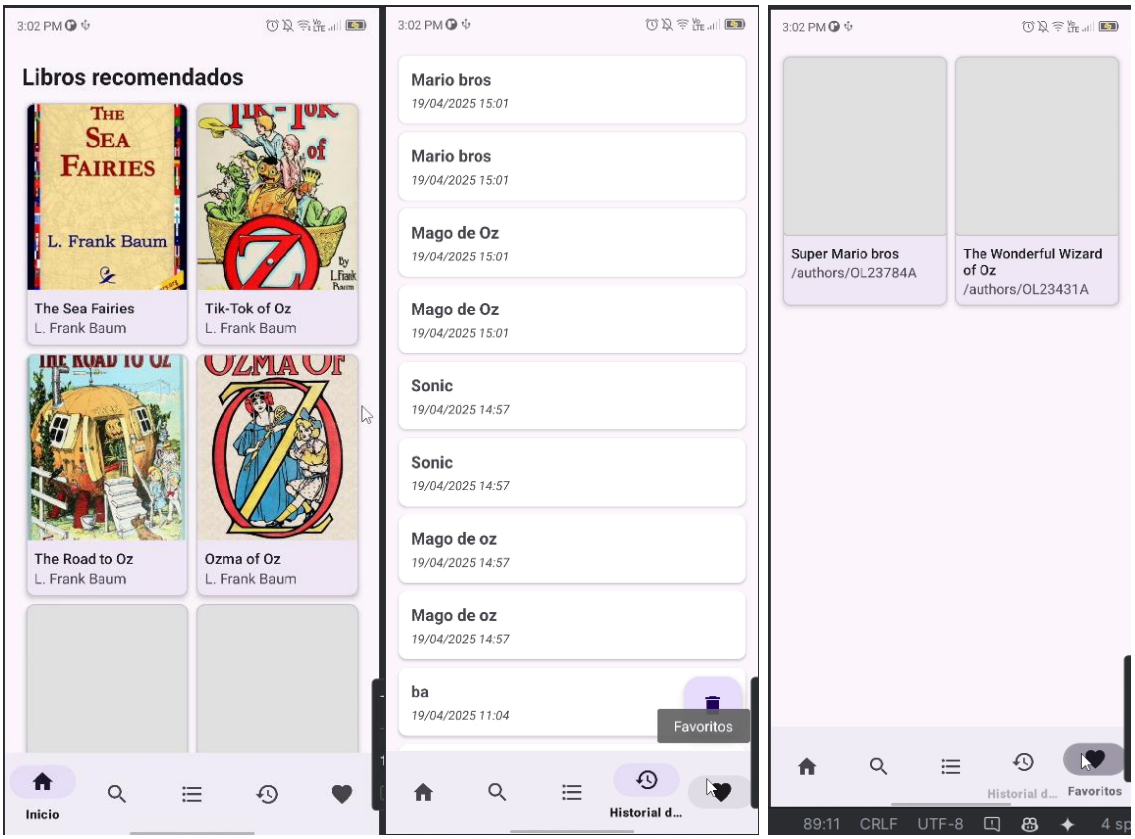
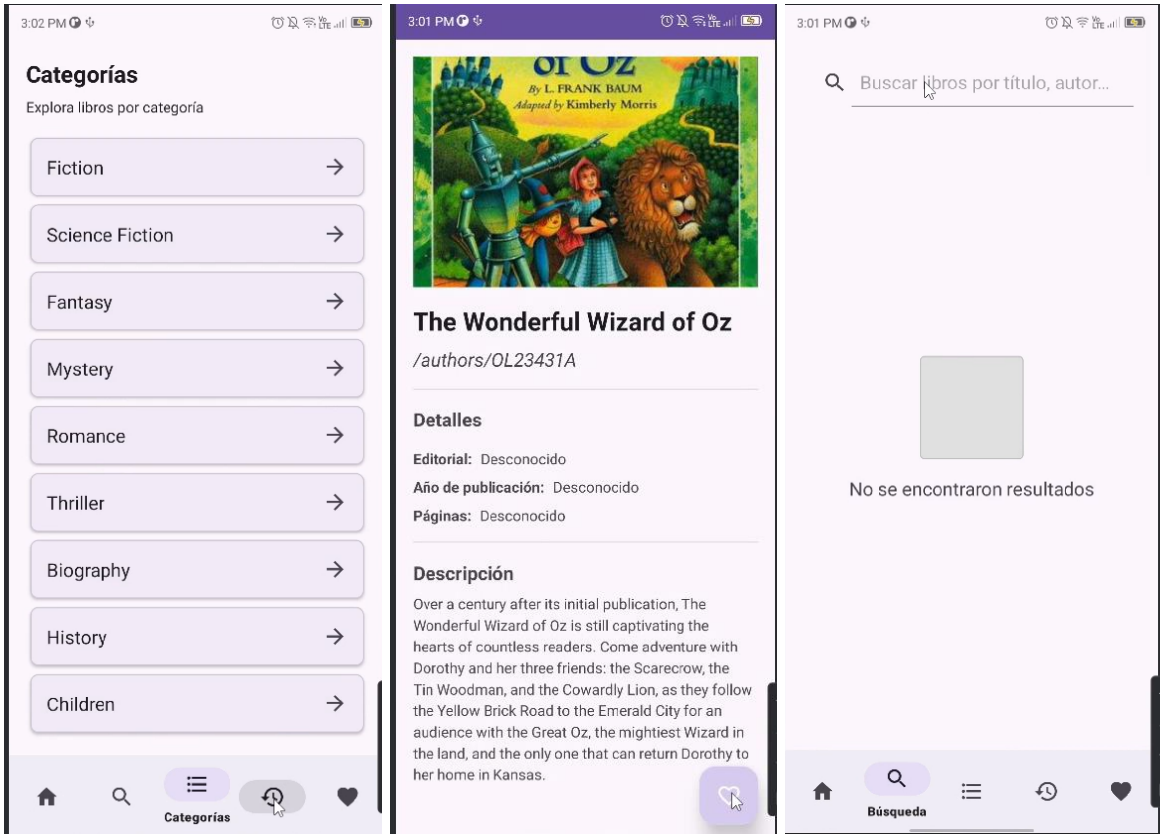


Figura 18 Perfil de Usuario

3.4 Funcionalidades de Búsqueda, Favoritos y Recomendaciones


Libros recomendados
The Sea Fairies
L. Frank Baum
Tik-Tok of Oz
L. Frank Baum
The Road to Oz
L. Frank Baum
Ozma of Oz
L. Frank Baum
Inicio
Mario bros
19/04/2025 15:01
Mario bros
19/04/2025 15:01
Mago de Oz
19/04/2025 15:01
Mago de Oz
19/04/2025 15:01
Sonic
19/04/2025 14:57
Sonic
19/04/2025 14:57
Mago de oz
19/04/2025 14:57
Mago de oz
19/04/2025 14:57
ba
19/04/2025 11:04
Favoritos
Historial d...
Super Mario bros
/authors/OL23784A
The Wonderful Wizard of Oz
/authors/OL23431A
Historial d...
Favoritos


Categorías
Explora libros por categoría
Fiction
Science Fiction
Fantasy
Mystery
Romance
Thriller
Biography
History
Children
Categorías
By L. FRANK BAUM
Adapted by Kimberly Morris
The Wonderful Wizard of Oz
/authors/OL23431A
Detalles
Editorial: Desconocido
Año de publicación: Desconocido
Páginas: Desconocido
Descripción
Over a century after its initial publication, The Wonderful Wizard of Oz is still captivating the hearts of countless readers. Come adventure with Dorothy and her three friends: the Scarecrow, the Tin Woodman, and the Cowardly Lion, as they follow the Yellow Brick Road to the Emerald City for an audience with the Great Oz, the mightiest Wizard in the land, and the only one that can return Dorothy to her home in Kansas.
Buscar libros por título, autor...
No se encontraron resultados
Búsqueda

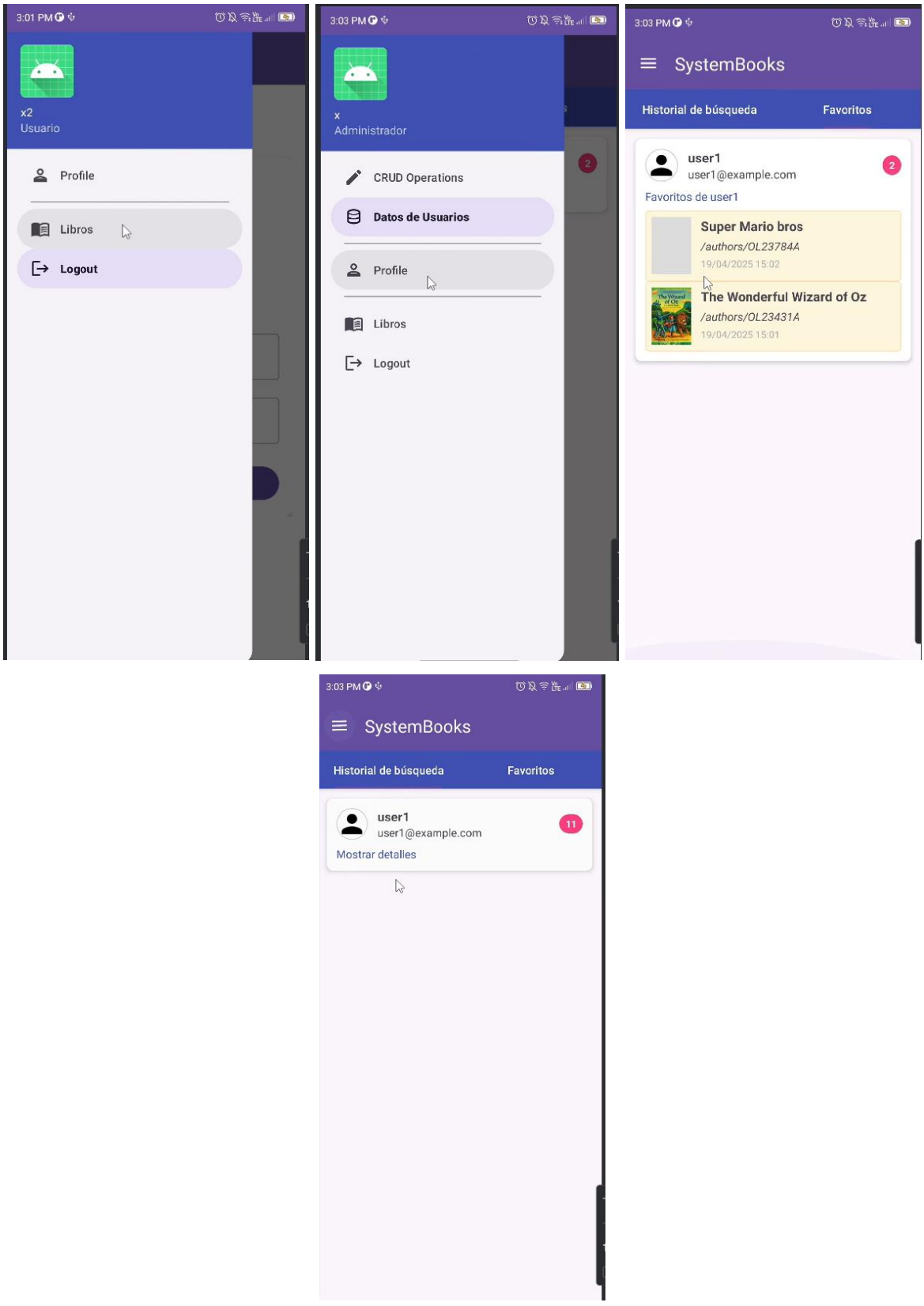


Figura 19 Funcionalidades de Búsqueda, Favoritos y Recomendaciones

4 Conclusiones

El desarrollo de la Práctica 5 ha permitido adquirir competencias avanzadas en la integración de APIs y bases de datos en aplicaciones móviles Android, enfrentando retos significativos como la gestión del estado de conexión, la sincronización de datos entre el sistema local y remoto, y la implementación de mecanismos efectivos de búsqueda y recomendación. La aplicación "SystemBooks" demuestra cómo es posible crear sistemas robustos que funcionan tanto online como offline, utilizando SQLite para el almacenamiento local y Retrofit para comunicarse con servicios externos como la API REST propia y Open Library. Los principales logros incluyen la implementación de un sistema de autenticación seguro, una interfaz de usuario intuitiva y un mecanismo eficiente de persistencia de datos que mantiene la coherencia entre el dispositivo y el servidor. Para futuras iteraciones, sería valioso expandir el sistema de recomendaciones incorporando algoritmos más sofisticados de machine learning que mejoren la personalización de sugerencias, así como añadir soporte para más formatos de libros y fuentes de información bibliográfica adicionales.

5 Bibliografía

Android Developers. (2025). *Android Developer Documentation*. Recuperado de https://developer.android.com/docs

Spring Framework. (2025). *Spring Boot Reference Documentation*. Recuperado de https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/