Ejercicio Fragment

Aplicación de gestión de tickets con Fragments

Autor: Ana Núñez Rejón

Curso: 2º DAM

Fecha: Noviembre 2025

https://github.com/AnaNunezRejon/EjercicioFragment

Centro: New Digital Talent - Granada

Índice

1. Introducción

El proyecto **Ejercicio Fragment** es una aplicación Android desarrollada en Java cuyo objetivo es practicar el uso de **Fragments**, el patrón **MVVM** y el manejo de **ficheros de texto** para almacenar información de forma persistente.

La aplicación simula un sencillo sistema de **gestión de tickets** o incidencias de soporte técnico. Un usuario introduce sus datos (nombre, apellidos y código de trabajador) en una pantalla de login y, una vez dentro, puede:

- Visualizar un listado de tickets en un *RecyclerView*.
- Filtrar los tickets según su estado.
- Crear nuevos tickets.
- Editar tickets existentes, modificando su resolución y estado.

Toda la información de los tickets se almacena en un fichero de texto `tickets.txt`, que inicialmente se incluye en la carpeta `assets` y se copia a la memoria interna (`/files`) la primera vez que se ejecuta la aplicación. A partir de ese momento, se trabaja siempre sobre el fichero de la memoria interna, de forma que los cambios permanecen persistentes entre ejecuciones.

2. Objetivos del proyecto

Los objetivos principales del proyecto son:

1. **Practicar el uso de Fragments** en lugar de Activities como pantallas principales de la aplicación.
2. **Aplicar el patrón MVVM**, separando la lógica en paquetes `model`, `view` y `viewmodel`.
3. **Trabajar con ficheros de texto** usando las clases vistas en el módulo de Acceso a Datos: `File`, `FileReader`, `FileWriter`, `BufferedReader`, etc.
4. Implementar un **RecyclerView** para mostrar listas dinámicas de elementos.
5. Manejar **spinners** y filtrado de datos según un estado.
6. Realizar **pruebas unitarias** sobre la lógica del modelo (clase `Ticket`) para entender cómo se usan los tests en proyectos reales.

3. Requisitos del sistema

3.1. Requisitos funcionales

- **RF1**: La aplicación debe permitir al usuario introducir su nombre, apellidos y código de trabajador en la pantalla de login.
- **RF2**: Tras el login, se mostrará una pantalla principal (*Home*) con un listado de todos los tickets.
- **RF3**: El usuario debe poder filtrar los tickets por estado: `NUEVO`, `ABIERTO`, `PENDIENTE`, `RESUELTO` o `CERRADO`.
- **RF4**: El usuario debe poder crear un nuevo ticket desde un botón en la pantalla principal.
- **RF5**: El usuario debe poder editar un ticket existente tocando sobre él en la lista.
- **RF6**: En modo *nuevo ticket*, solo se puede editar la descripción; el estado debe ser siempre `NUEVO`.
- **RF7**: En modo *editar ticket*, se puede modificar la resolución y el estado del ticket.
- **RF8**: Los cambios realizados en los tickets deben guardarse en el fichero de texto de la memoria interna.
- **RF9**: Debe existir un botón para cerrar sesión y volver al fragmento de login.

3.2. Requisitos no funcionales

- **RNF1**: La aplicación debe funcionar en dispositivos Android con versión 8.0 (API 26) o superior.
- **RNF2**: La interfaz debe ser sencilla, clara y usable, con estilos coherentes en todas las pantallas.
- **RNF3**: El código debe estar organizado y seguir una separación clara entre modelo, vista y lógica de negocio.
- **RNF4**: Los datos de los tickets deben mantenerse incluso después de cerrar la aplicación.
- **RNF5**: El proyecto debe ser fácilmente ampliable con nuevas funcionalidades en un futuro.

4. Arquitectura y estructura del proyecto

La aplicación sigue una arquitectura inspirada en **MVVM**, con la siguiente organización de paquetes:

```
com.example.ejerciciofragment

 model
    Ticket.java
    Estado.java

 view
    MainActivity.java
    LoginFragment.java
    HomeFragment.java
    EditarFragment.java
    TicketAdapter.java
    layouts XML (fragment_home, fragment_editar, item_ticket, ...)

 viewmodel
     controlador.java
```

Además de los paquetes Java, se utiliza la carpeta `assets` para incluir el fichero `tickets.txt` con los datos iniciales, y la carpeta `res/values` para definir los arrays de estados y los colores asociados a cada uno de ellos.

4.1. Diagrama de clases

En la Figura 1 se muestra un diagrama simplificado de las principales clases del proyecto.

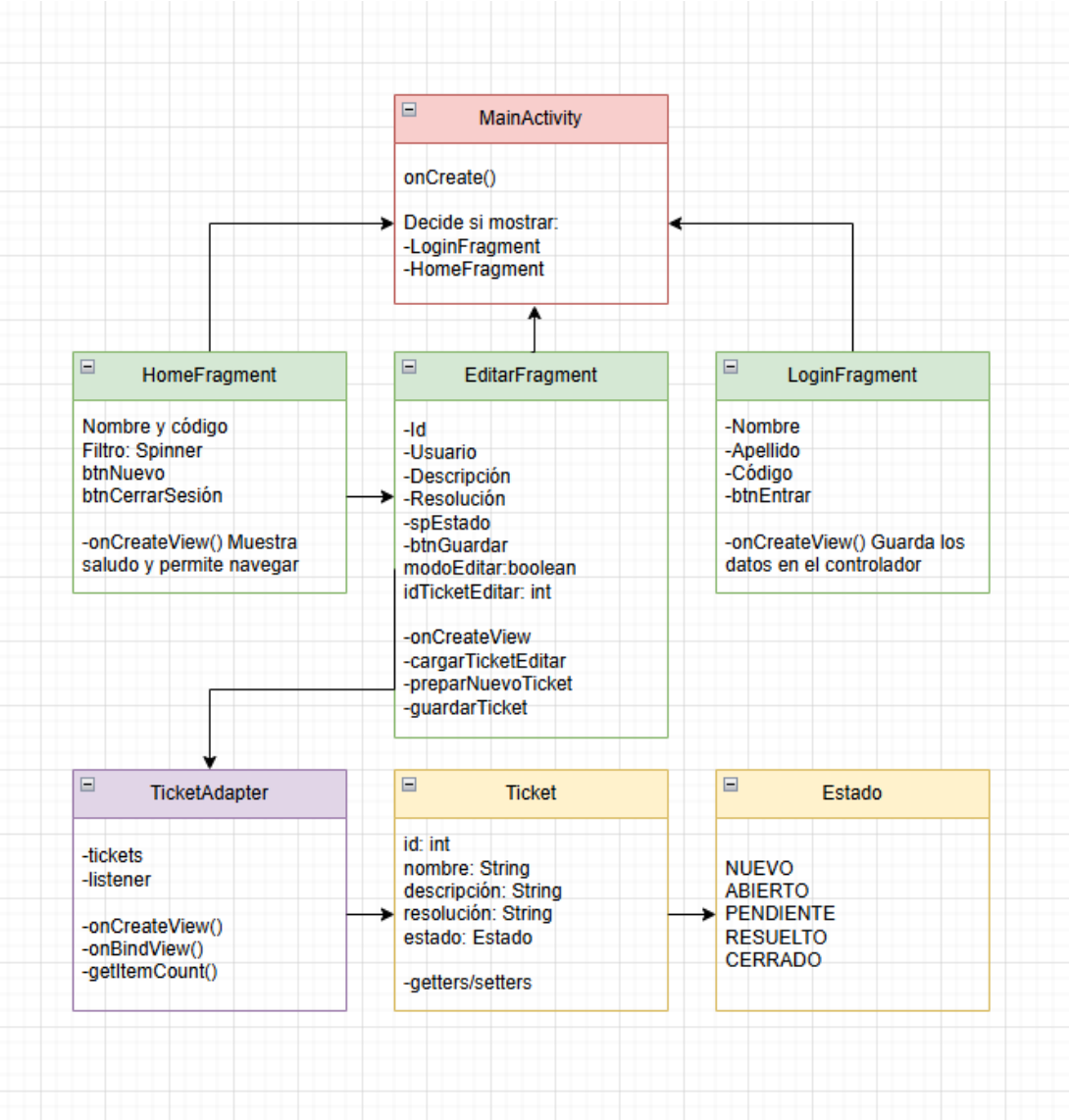

Figura 1: Diagrama de clases simplificado.

4.2. Diseño de la aplicación en figma

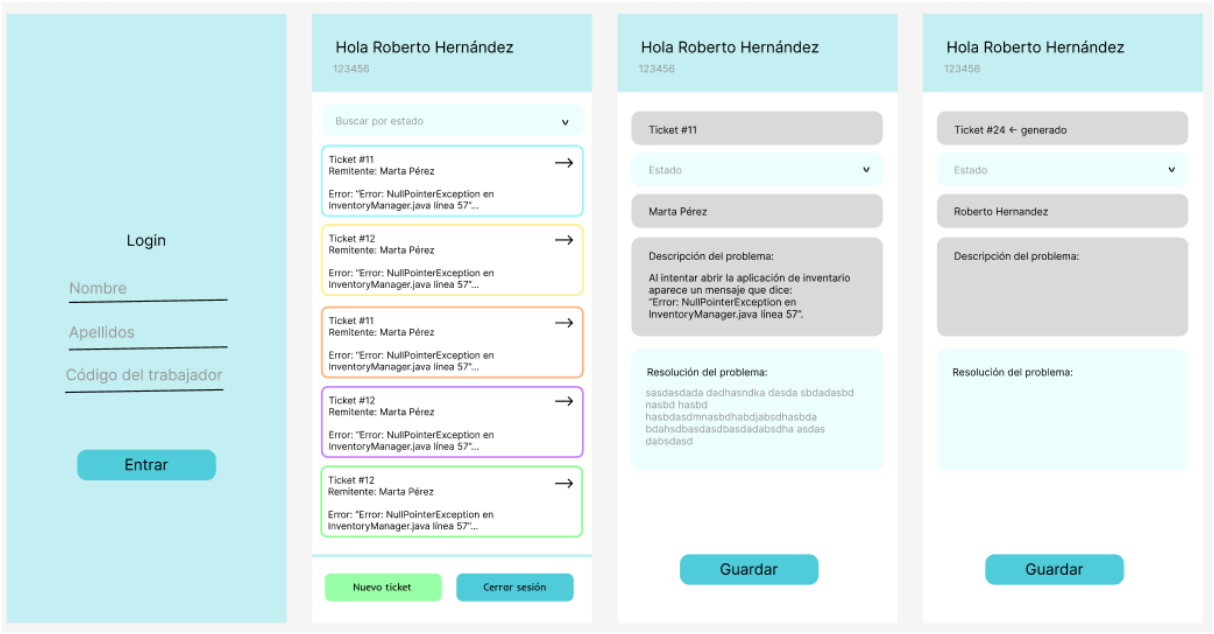

Figura 2: boceto previo en figma

4.3. Ideas e imágenes de referencia

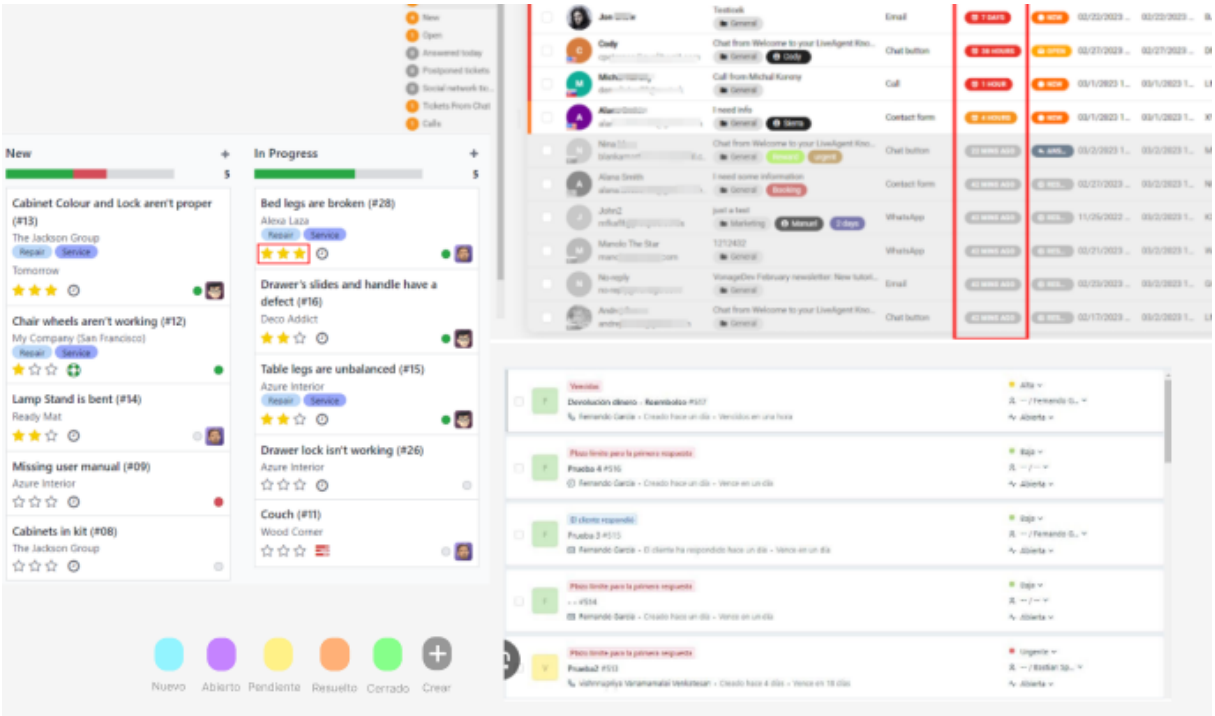

Figura 3: Ideas e imágenes de referencia

5. Modelo de datos

5.1. Clase Ticket

La clase `Ticket` representa una incidencia de soporte. Contiene:

- `id`: identificador numérico del ticket.
- `nombreUsuario`: nombre y apellidos de la persona que crea el ticket.
- `descripcion`: descripción del problema.
- `resolucion`: texto con la solución aplicada (puede estar vacío).
- `estado`: valor del enum `Estado`.

```java
// model/Ticket.java
public class Ticket {

    private int id;
    private String nombreUsuario;
    private String descripcion;
    private String resolucion;
    private Estado estado;

    public Ticket(int id, String nombreUsuario,
                  String descripcion, String resolucion,
                  Estado estado) {
        this.id = id;
        this.nombreUsuario = nombreUsuario;
        this.descripcion = descripcion;
        this.resolucion = resolucion;
        this.estado = estado;
    }

    // Getters y setters...
}
```

5.2. Enum Estado

El enum `Estado` define los posibles estados de un ticket:

```java
// model/Estado.java
public enum Estado {
    NUEVO,
    ABIERTO,
    PENDIENTE,
    RESUELTO,
    CERRADO
}
```

5.3. Formato del fichero de tickets

El fichero `tickets.txt` utiliza un formato sencillo de texto plano, donde cada línea representa un ticket con los campos separados por punto y coma:

```
id;nombreUsuario;descripcion;resolucion;ESTADO
```

Ejemplo de línea real:

```
1;Marta Pérez;No carga la app;;NUEVO
```

6. Gestión de persistencia (ficheros)

La lógica relacionada con la lectura y escritura del fichero de tickets se encuentra en la clase `controlador` dentro del paquete `viewmodel`.

6.1. Copia inicial desde assets a files

En la primera ejecución de la aplicación, si el fichero `tickets.txt` no existe en la carpeta `files` de la app, se copia desde `assets`:

```java
// viewmodel/controlador.java
public static void copiarTicketsDesdeAssets(Context ctx) {

    File destino = new File(ctx.getFilesDir(), "tickets.txt");

    if (destino.exists()) {
        return; // Ya existe, no copiamos otra vez
    }

    try {
        InputStream is = ctx.getAssets().open("tickets.txt");
        FileOutputStream fos = new FileOutputStream(destino);

        int dato = is.read();
        while (dato != -1) {
            fos.write(dato);
            dato = is.read();
        }

        is.close();
        fos.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

6.2. Lectura de tickets desde fichero

Para leer los tickets se usa `FileReader` y `BufferedReader`, tal y como se ha trabajado en Acceso a Datos:

```
public static ArrayList<Ticket> leerTickets(Context ctx) {

    ArrayList<Ticket> lista = new ArrayList<>();
    File fichero = new File(ctx.getFilesDir(), "tickets.txt");

    try {
        BufferedReader br = new BufferedReader(new FileReader(
           fichero));
        String linea = br.readLine();

        while (linea != null) {
            String[] partes = linea.split(";");

            if (partes.length == 5) {
                int id = Integer.parseInt(partes[0]);
                String nombre = partes[1];
                String desc = partes[2];
                String resol = partes[3];
                Estado estado = Estado.valueOf(partes[4]);

                lista.add(new Ticket(id, nombre, desc, resol,
                   estado));
            }

            linea = br.readLine();
        }

        br.close();

    } catch (Exception e) {
        e.printStackTrace();
    }

    return lista;
}
```

6.3. Guardado de tickets en fichero

Cuando se crean o modifican tickets desde la aplicación, se vuelve a escribir el fichero completo con todos los tickets actualizados:

```
public static void guardarTickets(Context ctx,
                                  ArrayList<Ticket> lista) {

    File fichero = new File(ctx.getFilesDir(), "tickets.txt");

    try {
        BufferedWriter bw = new BufferedWriter(
                new FileWriter(fichero, false)); // sobrescribe

        for (Ticket t : lista) {
            bw.write(
                t.getId() + ";" +
                t.getNombreUsuario() + ";" +
                t.getDescripcion() + ";" +
                t.getResolucion() + ";" +
                t.getEstado().toString()
            );
            bw.newLine();
        }

        bw.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

7. Interfaz de usuario y navegación

7.1. Fragments principales

La aplicación utiliza tres fragments principales:

LoginFragment Permite introducir nombre, apellidos y código.

HomeFragment Muestra la lista de tickets, filtro por estado y botones para nuevo ticket y cerrar sesión.

EditarFragment Se utiliza tanto para crear tickets nuevos como para editar tickets existentes.

7.2. Flujo de navegación

El flujo de navegación es el siguiente:

1. La aplicación se inicia en `MainActivity`, que carga el `LoginFragment`.
2. Tras introducir los datos y pulsar el botón de acceso, se guarda la sesión en `SharedPreferences` y se muestra `HomeFragment`.
3. Desde `HomeFragment`, el usuario puede:
   - Pulsar sobre un ticket para abrir `EditarFragment` en modo edición.
   - Pulsar el botón "Nuevo ticket" para abrir `EditarFragment` en modo creación.
4. Al guardar los cambios desde `EditarFragment`, se actualiza el fichero de tickets y se vuelve atrás a `HomeFragment`.
5. Desde el botón "Cerrar sesión" se vacía la sesión y se vuelve al `LoginFragment`.

7.3. HomeFragment: listado y filtrado

`HomeFragment` muestra un `RecyclerView` con la lista de tickets y un `Spinner` para filtrar por estado. Además, cada tarjeta de ticket muestra:

- El ID del ticket.
- El estado del ticket como texto.
- El remitente.
- La descripción.
- Un borde de color que indica visualmente el estado.

8. Lógica de negocio principal

8.1. Creación de nuevo ticket

Cuando el usuario pulsa "Nuevo ticket" en el `HomeFragment`, se abre el `EditarFragment` sin argumentos. En este caso:

- Se calcula un nuevo ID a partir del máximo existente.
- El estado se fija a `NUEVO` y no es editable.
- El nombre del usuario se toma de la sesión activa.
- Solo se permite editar la descripción del problema.

8.2. Edición de ticket existente

Cuando el usuario pulsa sobre un ticket en la lista:

- Se abre `EditarFragment` con el ID del ticket en un `Bundle`.
- Se carga el ticket correspondiente desde la lista.
- La descripción no se puede modificar.
- La resolución y el estado sí son editables.
- Al pulsar guardar, se actualiza el ticket en la lista y se persiste el cambio en el fichero.

8.3. Filtrado de tickets por estado

El filtrado se realiza manteniendo dos listas en `HomeFragment`:

- `listaOriginal`: contiene todos los tickets leídos del fichero.
- `listaTickets`: contiene únicamente los tickets que se muestran, según el filtro.

Cuando el usuario selecciona un estado en el `Spinner`:

- Si elige "Todos", se copian todos los elementos de `listaOriginal`.
- Si elige un estado concreto, se recorre `listaOriginal` y se añaden a `listaTickets` solo los tickets cuyo estado coincide.
- Finalmente se llama a `adapter.notifyDataSetChanged()` para refrescar la lista.

8.4. Colores según estado

Cada tarjeta de ticket tiene un fondo `GradientDrawable` definido en `ticket_bg.xml`. Desde el adaptador se cambia el color del borde según el estado del ticket:

- `NUEVO`: color verde.
- `ABIERTO`: color azul.
- `PENDIENTE`: color amarillo o naranja.
- `RESUELTO`: color morado.
- `CERRADO`: color gris.

Esto mejora la usabilidad visual y, además, se acompaña de una etiqueta de texto con el estado para facilitar la lectura a personas con problemas de visión de colores.

9. Pruebas unitarias

Para practicar el uso de pruebas unitarias, he creado varios tests.

Estas pruebas utilizan **JUnit** y se centran en la lógica del modelo (`Ticket`) y en operaciones sencillas sobre colecciones de tickets.

9.1. Ejemplos de pruebas realizadas

9.1.1. Comprobación del estado de un ticket

```java
// TicketEstadoTest.java
@Test
public void test_getEstado_devuelveEstadoCorrecto() {
    Ticket t = new Ticket(1, "Ana", "Prueba", "", Estado.NUEVO);
    assertEquals(Estado.NUEVO, t.getEstado());
}

@Test
public void test_setEstado_modificaElEstadoCorrectamente() {
    Ticket t = new Ticket(1, "Ana", "Prueba", "", Estado.NUEVO);
    t.setEstado(Estado.ABIERTO);
    assertEquals(Estado.ABIERTO, t.getEstado());
}
```

9.1.2. Generación del siguiente ID

```java
@Test
public void generarSiguienteId_esCorrecto() {
    ArrayList<Ticket> lista = new ArrayList<>();
    lista.add(new Ticket(1, "Ana", "Desc", "", Estado.NUEVO));
    lista.add(new Ticket(5, "Luis", "Desc", "", Estado.NUEVO));
    lista.add(new Ticket(3, "Marta", "Desc", "", Estado.NUEVO));

    int max = 0;
    for (Ticket t : lista) {
        if (t.getId() > max) max = t.getId();
    }

    int siguiente = max + 1;
    assertEquals(6, siguiente);
}
```

9.1.3. Lectura de una línea de fichero

```
@Test
public void leerTicketDesdeLinea() {
    String linea = "10;Ana␣N   ez;No␣funciona;Reiniciar;ABIERTO"
       ;
    String[] p = linea.split(";");

    Ticket t = new Ticket(
        Integer.parseInt(p[0]),
        p[1],
        p[2],
        p[3],
        Estado.valueOf(p[4])
    );

    assertEquals(10, t.getId());
    assertEquals("Ana␣N   ez", t.getNombreUsuario());
    assertEquals("No␣funciona", t.getDescripcion());
    assertEquals("Reiniciar", t.getResolucion());
    assertEquals(Estado.ABIERTO, t.getEstado());
}
```

10. Posibles mejoras

Algunas mejoras que podrían implementarse en futuras versiones del proyecto son:

- Sustituir el fichero de texto por una base de datos SQLite o Room.
- Añadir campos de prioridad o categoría a los tickets.
- Permitir la búsqueda por texto en la descripción o remitente.
- Implementar un sistema de autenticación real con usuarios y contraseñas.
- Añadir paginación o agrupación por estado o fecha.
- Incorporar notificaciones cuando un ticket cambie de estado.

11. Documentación técnica consultada

Durante el desarrollo de la aplicación se hizo uso de la documentación oficial de Android, con el objetivo de implementar correctamente las funcionalidades necesarias. A continuación se detallan las principales páginas de referencia, junto con su relación con las partes del código implementado.

11.1. 7.1. Uso de Fragments

Referencia: https://developer.android.com/guide/fragments

Los fragments constituyen la base de la arquitectura de la aplicación. Se emplearon para dividir la interfaz en tres pantallas principales:

- `LoginFragment`: formulario de acceso del usuario.
- `HomeFragment`: visualización y filtrado de tickets.
- `EditarFragment`: creación y actualización de tickets.

La documentación oficial permitió comprender el ciclo de vida de un fragment, la inflado de layouts mediante `onCreateView()` y la navegación a otros fragments con `FragmentManager` y `addToBackStack()`.

11.2. 7.2. RecyclerView y su Adapter

Referencia: https://developer.android.com/guide/topics/ui/layout/recyclerview

La lista principal del Home se implementó mediante un `RecyclerView`. Se consultó la documentación oficial para:

- Crear el `TicketAdapter`.
- Definir el `ViewHolder`.
- Inflar el layout `item_ticket.xml`.
- Actualizar la lista mediante `notifyDataSetChanged()`.

Este recurso fue fundamental para mostrar los tickets de forma eficiente.

11.3. 7.3. ArrayAdapter y Spinners

Referencia: https://developer.android.com/reference/android/widget/ArrayAdapter

Los spinners utilizados tanto en `HomeFragment` como en `EditarFragment` se construyeron con la clase `ArrayAdapter`, cargando valores desde archivos XML en `res/values/spinner_arrays.x`

Esto permitió:

- Cargar dinámicamente los estados de un ticket.
- Filtrar los tickets desde el Home.
- Seleccionar o bloquear estados en modo edición o creación.

11.4. 7.4. SharedPreferences (gestión de sesión)

Referencia: https://developer.android.com/training/data-storage/shared-preferences

La clase `controlador` utiliza `SharedPreferences` para almacenar la sesión del usuario:

- Guardar nombre, apellido y código.
- Recuperar la sesión al iniciar la app.
- Cerrar sesión eliminando los datos.

Gracias a esta API, la aplicación recuerda quién ha iniciado sesión sin necesidad de volver a introducir credenciales.

11.5. 7.5. Lectura de archivos en assets

Referencia: https://developer.android.com/guide/topics/resources/providing-resources
OriginalFiles

El fichero `tickets.txt` se aloja en la carpeta `assets`. Se consultó la documentación para:

- Acceder al archivo mediante `getAssets().open()`.
- Copiarlo a `/data/data/.../files`.
- Trabajar con él como fichero persistente del usuario.

11.6. 7.6. Lectura y escritura con FileReader / FileWriter

Referencia: https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.
html

La persistencia de los tickets se implementa leyendo y escribiendo líneas en un archivo de texto plano mediante:

- `BufferedReader`
- `BufferedWriter`
- `FileReader`
- `FileWriter`

La estructura de cada ticket fue definida usando campos separados por punto y coma.

11.7. 7.7. Navegación entre pantallas

Referencia: https://developer.android.com/guide/navigation
La transición entre fragments se implementó con:

- `FragmentTransaction.replace()`
- `addToBackStack(null)`

Esta documentación ayudó a realizar correctamente la navegación desde un item del RecyclerView hacia la pantalla de edición.

11.8. 7.8. Estilos, colores y drawables

Referencia: https://developer.android.com/guide/topics/resources/drawable-resource
Se consultó para definir:

- Bordes de colores en `ticket_bg.xml`.
- Colores personalizados del estado.
- Estilos limpios para los items y los botones.

11.9. 7.9. Ciclo de vida de la aplicación

Referencia: https://developer.android.com/guide/components/activities/activity-life
Fue clave para entender por qué `HomeFragment.onResume()` debe recargar los tickets actualizados.

12. Conclusiones

El desarrollo de la aplicación **EjercicioFragment** ha permitido consolidar varios conceptos clave del ciclo de formación de DAM:

- Uso de **Fragments** como base de la interfaz de usuario en Android, sustituyendo a las Activities tradicionales.
- Aplicación del patrón **MVVM** para separar claramente la lógica de negocio (controlador), el modelo de datos y la interfaz.
- Manejo de ficheros de texto con las clases `File`, `FileReader`, `FileWriter` y `BufferedReader`, tal y como se ha estudiado en el módulo de Acceso a Datos.
- Trabajo con **RecyclerView** y adaptadores personalizados para mostrar listas dinámicas con distinto formato según el estado.
- Creación y ejecución de **pruebas unitarias** con JUnit, entendiendo cómo ayudan a detectar errores y a validar la lógica.

En conjunto, el proyecto supone un ejercicio completo que combina diseño de interfaz, gestión de datos y buenas prácticas de programación orientada a objetos, acercándose a la estructura de aplicaciones Android reales.