Instituto Politécnico Nacional
Escuela Superior de Cómputo

Práctica #4: Manejo de archivos en android

Asignatura: Desarrollo de aplicaciones móviles nativas

Profesor: Gabriel Hurtado Avilés

Alumno: Díaz Fuentes Kevin

Boleta: 2021630259

Grupo: 7CV2

Fecha: 10/11/2025

Introducción

En la era digital contemporánea, el teléfono inteligente se ha consolidado como el centro neurálgico de nuestra vida personal y profesional. Actúa como nuestro principal dispositivo de comunicación, cámara, centro de entretenimiento y, crucialmente, como un repositorio de almacenamiento masivo. Diariamente, generamos y consumimos una cantidad exponencial de datos: fotografías, videos, documentos de trabajo, archivos descargados y música. Esta acumulación de información hace que la gestión eficiente de archivos no sea un lujo, sino una necesidad fundamental para mantener el dispositivo organizado, seguro y con un rendimiento óptimo.

Si bien el sistema operativo Android ofrece herramientas nativas para la gestión de archivos, a menudo estas pueden ser limitadas en funcionalidad o no ofrecer la experiencia de usuario intuitiva que las aplicaciones modernas demandan. Este contexto crea una oportunidad para el desarrollo de aplicaciones especializadas que brinden una interfaz más limpia, mayor control y características más robustas.

La presente práctica, titulada "GestorDeArchivos", nace como respuesta a esta necesidad. Se trata de una aplicación nativa de Android desarrollada desde cero con el objetivo de proporcionar una solución de gestión de almacenamiento externo que sea, al mismo tiempo, potente en su funcionalidad y moderna en su ejecución técnica.

Para lograrlo, la práctica se fundamenta en las tecnologías más actuales del ecosistema de desarrollo de Android. La totalidad de la interfaz de usuario (UI) ha sido construida utilizando Jetpack Compose, el moderno kit de herramientas declarativo de Google. Esta elección permite la creación de una UI reactiva, dinámica y menos propensa a errores, alejándose del tradicional sistema de vistas basado en XML.

Arquitectónicamente, la aplicación se adhiere estrictamente al patrón Model-View-ViewModel (MVVM). Esta separación de intereses es vital: la UI (los Composables) se mantiene limpia y se limita a observar los cambios de estado, mientras que el ViewModel (como FileViewModel) orquesta toda la lógica de negocio, desde la navegación por directorios hasta las complejas operaciones de archivos. Para manejar las operaciones de entrada/salida (I/O) del sistema de archivos (tareas inherentemente lentas que no deben bloquear la interfaz), la aplicación hace un uso intensivo de Corutinas de Kotlin y Dispatchers.IO, garantizando una experiencia de usuario fluida y sin interrupciones (ANR).

El "GestorDeArchivos" ofrece un conjunto completo de funcionalidades esenciales, que incluyen:

- Navegación Intuitiva: Exploración fluida del árbol de directorios del almacenamiento externo, complementada con un sistema de "migas de pan" (FileBreadcrumbs) que permite al usuario ubicarse y navegar a directorios padres con un solo toque.

- Gestión de Archivos (CRUD): Implementación completa de las operaciones de Copiar, Mover (Cortar), Renombrar y Borrar tanto archivos individuales como directorios completos.
- Visores Nativos: Capacidad para abrir y visualizar archivos comunes directamente dentro de la app, incluyendo un visor de archivos de texto y un visor de imágenes con funcionalidades avanzadas de zoom, paneo y rotación.
- Integración y Permisos: Manejo robusto del permiso MANAGE_EXTERNAL_STORAGE (Acceso a todos los archivos) e integración con el ecosistema de Android mediante la función de Compartir, implementada de forma segura a través de un FileProvider.
- Personalización: Demostración de persistencia de datos mediante Jetpack DataStore, permitiendo al usuario seleccionar y guardar su tema de preferencia (Guinda IPN o Azul ESCOM).

Este documento servirá como el reporte técnico del proyecto, detallando la arquitectura de la aplicación, el diseño de la interfaz, la implementación específica de las operaciones de archivos, los diagramas UML que modelan el sistema, las pruebas de funcionalidad realizadas y las conclusiones aprendidas durante el ciclo de desarrollo.

Desarrollo

Descripción técnica

La aplicación "GestorDeArchivos" está construida sobre un stack tecnológico moderno de Android, priorizando la separación de responsabilidades, la reactividad de la interfaz de usuario y la seguridad en el manejo de operaciones del sistema. La arquitectura central sigue el patrón **MVVM (Model-View-ViewModel)**.

1. Arquitectura y Componentes Centrales

La lógica de la aplicación se divide en las siguientes capas clave:

Capa de UI (Jetpack Compose):

- La interfaz de usuario es completamente declarativa, construida con Jetpack Compose. La actividad principal, MainActivity, actúa como el host.
- **Gestión de Estado de UI:** El estado de la UI se gestiona de forma reactiva. Las pantallas (Composable functions) observan los datos expuestos por los ViewModels. Por ejemplo, FileExplorerScreen utiliza observeAsState() para reaccionar a los cambios en el LiveData del FileViewModel (la lista de archivos).
- **Estado Efímero:** El estado local de la UI, como la visibilidad de un diálogo (showOptionsSheet) o el contenido de un campo de texto (RenameFileDialog), se maneja internamente en el Composable usando rememberSaveable y remember { mutableStateOf(...) }.

Capa de Lógica de UI (ViewModels):

- **FileViewModel**: Es el componente central de la lógica de negocio. Hereda de AndroidViewModel para obtener acceso al contexto de la aplicación (usado para Toasts y el FileProvider). Este ViewModel expone el estado de la aplicación a la UI a través de LiveData (currentPath, filesList). También contiene la lógica para todas las operaciones de archivos (cargar, borrar, renombrar, etc.).
- **ThemeViewModel**: Un ViewModel más pequeño y enfocado, responsable únicamente de gestionar el tema de la aplicación. Utiliza StateFlow para exponer el tema actual (AppThemeType) a la MainActivity.

Capa de Datos (Persistencia y Acceso a Archivos):

- **Sistema de Archivos:** La fuente de datos principal es el sistema de archivos del dispositivo, al cual se accede a través de la API clásica java.io.File. Todas estas llamadas se aíslan dentro del FileViewModel.
- **Preferencias (DataStore):** Para la persistencia de los ajustes del usuario, como el tema seleccionado, la aplicación utiliza Jetpack DataStore (específicamente preferencesDataStore). El ThemeViewModel gestiona la lectura y escritura asíncrona

de esta preferencia, asegurando que la elección del usuario (Guinda o Azul) se recuerde entre sesiones.

2. Concurrencia y Manejo de Hilos

Para garantizar que la aplicación sea fluida y nunca bloquee el hilo principal (UI), se utiliza un modelo de concurrencia basado en Corutinas de Kotlin:

- **Operaciones I/O:** Todas las operaciones que interactúan con el disco (leer directorio, borrar, renombrar, copiar, pegar) dentro del FileViewModel se ejecutan en un contexto de Dispatchers.IO. Esto se logra lanzando una nueva corutina en el viewModelScope (viewModelScope.launch(Dispatchers.IO)).
- **Actualización de UI:** Después de que la operación en segundo plano finaliza, la actualización de los LiveData (que debe ocurrir en el hilo principal) se realiza de forma segura usando withContext(Dispatchers.Main).
- **Carga de Visores:** De manera similar, el TextViewerScreen utiliza un LaunchedEffect para disparar una corutina que lee el contenido del archivo en Dispatchers.IO, mostrando un indicador de carga mientras tanto.

3. Navegación

La navegación dentro de la aplicación es manejada por Jetpack Navigation para Compose.

- AppNavigationHost.kt define el NavHost y todas las rutas (composable).
- La FileExplorerScreen recibe el navController como parámetro, permitiéndole invocar navController.navigate(...) para moverse a otras pantallas.
- **Paso de Argumentos:** Para abrir un visor de texto o imagen, la ruta del archivo se pasa como un argumento de navegación. Para manejar de forma segura los caracteres especiales y espacios en las rutas de los archivos, la ruta se codifica en URL (URLEncoder.encode) antes de ser añadida a la ruta de navegación y se decodifica (URLDecoder.decode) en la pantalla de destino.

4. Integración con el Sistema Android

- **Permisos:** La aplicación solicita el permiso MANAGE_EXTERNAL_STORAGE, que es necesario para una aplicación de gestión de archivos en Android 11 y superior. La lógica para verificar (Environment.isExternalStorageManager()) y solicitar este permiso (lanzando un Intent a Settings.ACTION_MANAGE_APP_ALL_FILES_ACCESS_PERMISSION) está encapsulada en PermissionManager.kt. MainActivity utiliza rememberLauncherForActivityResult para manejar el flujo de solicitud de permisos.
- **Compartir y Abrir Archivos (FileProvider):** Para compartir archivos de forma segura con otras aplicaciones o para permitir que otras aplicaciones abran un archivo, la app utiliza un FileProvider.

1. Está declarado en el AndroidManifest.xml con una autoridad única basada en el applicationId.
2. provider_paths.xml especifica que el provider puede acceder a todo el almacenamiento externo (<external-path ... path="." />).
3. Las funciones shareFile (en FileViewModel) y openFileWithIntent (en FileExplorerScreen) usan FileProvider.getUriForFile(...) para obtener una content:// URI segura.
4. Se obtiene el tipo MIME del archivo usando MimeTypeHelper.kt y se lanza un Intent (ACTION_SEND o ACTION_VIEW) con la bandera FLAG_GRANT_READ_URI_PERMISSION.

- **Carga de Imágenes (Coil):** La biblioteca **Coil** (io.coil-kt:coil-compose) se utiliza para la carga eficiente de imágenes. El composable AsyncImage se usa en dos lugares:
    1. En FileExplorerScreen, para cargar miniaturas en la lista de archivos, con ContentScale.Crop.
    2. En ImageViewerScreen, para cargar la imagen completa con ContentScale.Fit. Coil maneja automáticamente la carga en segundo plano, el almacenamiento en caché y la decodificación desde un objeto File.
- **Tematización:** El archivo Theme.kt define esquemas de color personalizados (lightColorScheme y darkColorScheme) para los dos temas: Guinda (IPN) y Azul (ESCOM). El composable FileExplorerTheme aplica el esquema de color correcto basándose en el valor del ThemeViewModel y si el sistema está en modo oscuro (isSystemInDarkTheme()).

Implementación

La implementación de las funcionalidades clave del "GestorDeArchivos" se centra en el FileViewModel. Este enfoque, acorde con la arquitectura MVVM, asegura que toda la lógica de operaciones de archivos (I/O) esté aislada de la capa de UI (Compose) y se ejecute de manera asíncrona para no bloquear el hilo principal.

Carga y Navegación de Directorios

La operación más fundamental es la carga del contenido de un directorio.

loadDirectory(path: String): Esta función se invoca desde una corutina en el viewModelScope con Dispatchers.IO.

1. Primero, comprueba que el permiso de almacenamiento esté concedido (_permissionsGranted.value == true).
2. Utiliza la API estándar java.io.File(path) para crear una instancia del directorio.
3. La llamada clave de I/O es directory.listFiles(). Esta llamada puede fallar (p.ej., por permisos de sistema en carpetas como /data) o devolver null, lo cual se maneja para evitar crasheos.
4. Ordenamiento: Los archivos obtenidos se ordenan usando una expresión sortedWith personalizada: compareBy({ !it.isDirectory }, { it.name.lowercase() }). Esta lógica de comparación dual asegura que las carpetas (!it.isDirectory evalúa a false, que se ordena antes que true) siempre aparezcan primero, y luego, tanto las carpetas como los archivos se ordenan alfabéticamente en minúsculas.
5. Finalmente, la lista ordenada y la ruta actual se publican en sus respectivos LiveData (_filesList.value y _currentPath.value) usando withContext(Dispatchers.Main) para garantizar que la actualización ocurra en el hilo de UI.

MapsUp(): Esta función implementa la lógica de "subir un nivel". Obtiene el _currentPath.value actual, crea un objeto File, y si no está en la raíz (initialPath), obtiene su ruta padre (current.parentFile!!.path) y llama a loadDirectory con esa nueva ruta.

Operaciones CRUD de Archivos

Las operaciones de Crear, Leer, Actualizar y Borrar se implementan de la siguiente manera:

Borrar (deleteFile): Esta operación también se ejecuta en Dispatchers.IO. La implementación distingue entre archivos y directorios:

- Si file.isDirectory es verdadero, se invoca file.deleteRecursively(). Esto es crucial ya que delete() solo funciona en directorios vacíos. deleteRecursively() borra la carpeta y todo su contenido.

- Si es un archivo, se usa file.delete(). Tras una eliminación exitosa (success == true), se llama internamente a loadDirectory para refrescar la lista de archivos en la UI. La UI simplemente invoca esta función después de una confirmación en el DeleteConfirmationDialog.

Renombrar (renameFile): Invocada desde el RenameFileDialog, esta función se ejecuta en Dispatchers.IO.

1. Realiza una validación para asegurar que el newName no esté vacío.
2. Crea un nuevo objeto File con la ruta de destino: File(file.parent, newName).
3. Prevención de Conflictos: Se realiza una comprobación de seguridad clave: if (newFile.exists()). Si un archivo con el nuevo nombre ya existe, la operación se aborta y se notifica al usuario.
4. La operación de I/O es file.renameTo(newFile).
5. Si tiene éxito, se refresca el directorio actual llamando a loadDirectory.

Leer (Visores de Archivos): La lectura se implementa de tres maneras, decididas en FileExplorerScreen:

1. Visor de Texto: Para archivos identificados por isTextFile(), se navega a TextViewerScreen. Esta pantalla usa un LaunchedEffect que lanza una corutina en Dispatchers.IO para ejecutar file.readText(). El String resultante se almacena en un estado de Compose (fileContent) y se muestra en un TextField con readOnly = true para permitir selección y scroll.
2. Visor de Imágenes: Para archivos isImageFile(), se navega a ImageViewerScreen. La carga de la imagen se delega a la biblioteca Coil a través del composable AsyncImage(model = file, ...). Los gestos de zoom y paneo se implementan usando rememberTransformableState y aplicando las transformaciones de scale y offset al AsyncImage mediante el modificador graphicsLayer.
3. Visor Externo (Fallback): Para todos los demás tipos de archivo (PDF, MP3, etc.), se llama a la función openFileWithIntent. Esta función es crucial para la integración con el sistema (ver sección "Integración del Sistema" más abajo).

Implementación del Portapapeles (Copiar y Mover)

Se implementó un portapapeles interno simple usando un LiveData en el FileViewModel.

Estado del Portapapeles: private val _clipboard = MutableLiveData<Pair<File, ClipboardAction>?>(null). Almacena el archivo a operar y la acción a realizar (COPY o MOVE).

Copiar/Mover: Las funciones copyFileToClipboard(file) y moveFileToClipboard(file) simplemente actualizan el valor de _clipboard.

Pegar (pasteFile): Esta es la función principal, ejecutada en Dispatchers.IO.

1. Lee el _clipboard.value. Si es nulo, no hace nada.
2. Obtiene el (fileToPaste, action).
3. Define el destinationDir basándose en el _currentPath.value.
4. Valida que se pueda escribir (canWrite()) y que el archivo no exista ya.
5. Un bloque when(action) determina la operación de I/O:
    1. ClipboardAction.COPY: Ejecuta fileToPaste.copyRecursively(newFile). copyRecursively es esencial para copiar carpetas con contenido.
    2. ClipboardAction.MOVE: Ejecuta fileToPaste.renameTo(newFile). En la mayoría de los sistemas de archivos de Android, si el origen y el destino están en el mismo volumen, renameTo es una operación de metadatos casi instantánea que "mueve" el archivo.
6. Limpieza: El finally del bloque try-catch contiene una lógica importante: solo si la acción fue MOVE, se limpia el portapapeles (clearClipboard()). Esto permite pegar un archivo copiado múltiples veces.
7. Finalmente, se refresca el directorio actual.

Integración con el Sistema Android (Intents y FileProvider)

Para interactuar de forma segura con otras aplicaciones, la implementación no expone rutas de archivo (file://) directas, sino que utiliza un FileProvider.

1. Configuración: El FileProvider está definido en AndroidManifest.xml y sus rutas accesibles se definen en res/xml/provider_paths.xml para cubrir todo el almacenamiento externo.
2. shareFile(context, file): Esta función obtiene la autoridad del provider (${context.packageName}.provider).
3. Genera una URI segura: val fileUri = FileProvider.getUriForFile(context, authority, file).
4. Obtiene el tipo MIME usando el MimeTypeHelper.
5. Crea un Intent(Intent.ACTION_SEND) y añade la URI como Intent.EXTRA_STREAM.
6. Añade la bandera Intent.FLAG_GRANT_READ_URI_PERMISSION para dar permiso temporal a la app que recibe el intent.

7. Lanza el intent usando un Intent.createChooser().

- openFileWithIntent(context, file): La implementación en FileExplorerScreen es idéntica a shareFile, pero con dos diferencias clave:
    - La acción es Intent.ACTION_VIEW.
    - Los datos se establecen usando setDataAndType(fileUri, mimeType) en lugar de EXTRA_STREAM.

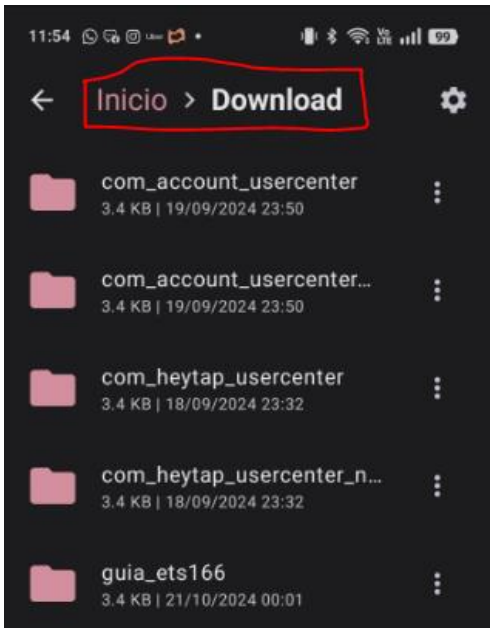
Esta implementación dual de FileProvider permite que la aplicación delegue de forma segura y robusta la visualización de tipos de archivo no soportados y la funcionalidad de compartir al ecosistema de aplicaciones del usuario.

Pruebas Realizadas

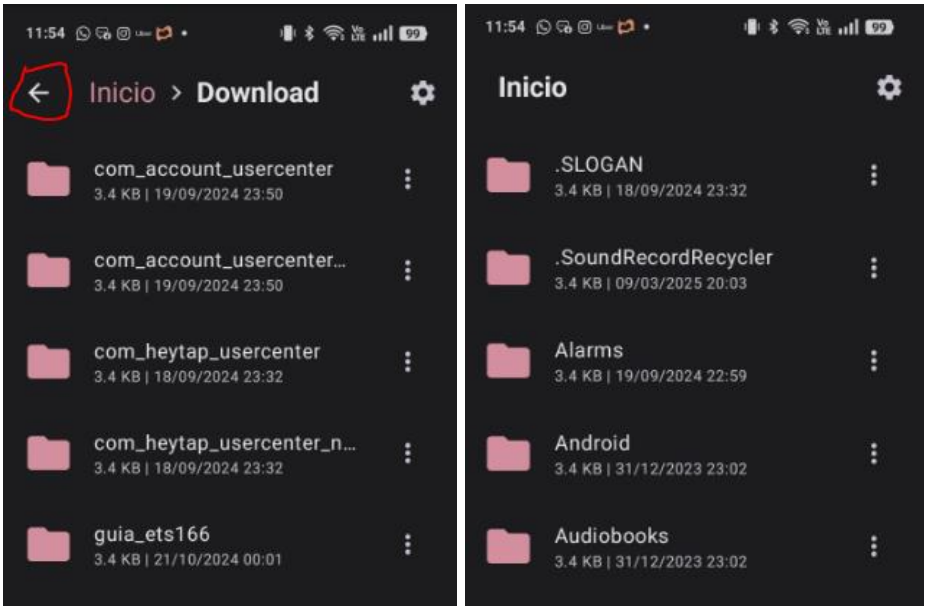
Plan de Pruebas

Pruebas de Navegación:

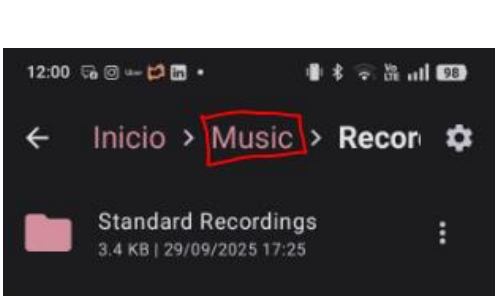
- Caso 1.1 (Entrar): En FileExplorerScreen, hacer clic en una carpeta (ej. "Download"). Verificar que la lista se actualiza y muestra el contenido de "Download". Verificar que las "migas de pan" (FileBreadcrumbs) se actualizan (ej. "Inicio > Download").

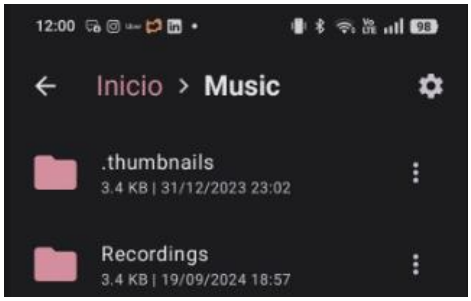


- Caso 1.2 (Subir): Desde "Download", presionar el icono de flecha "atrás" en la TopAppBar. Verificar que se vuelve a la raíz ("Inicio").



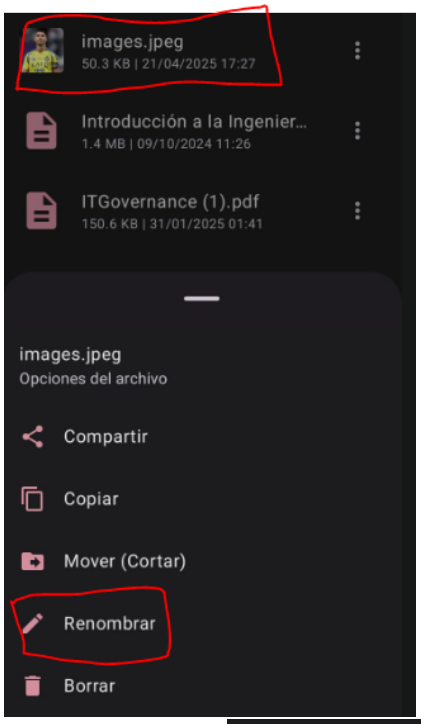
- Caso 1.3 (Migas de Pan): Navegar a una subcarpeta (ej. "Inicio > Download > Subcarpeta"). Hacer clic en "Inicio" en las migas de pan. Verificar que se navega directamente a la raíz.



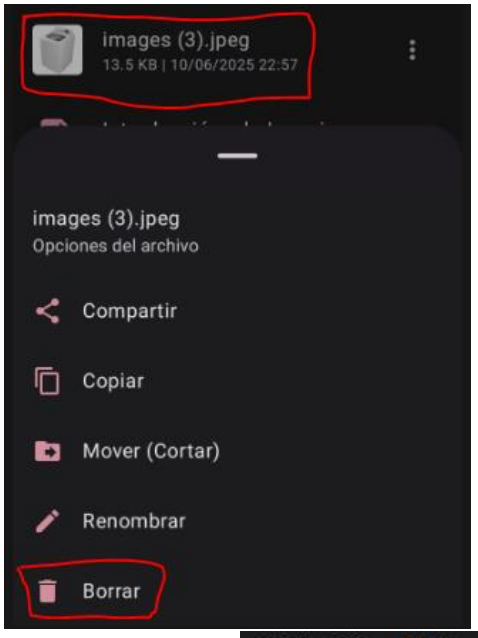


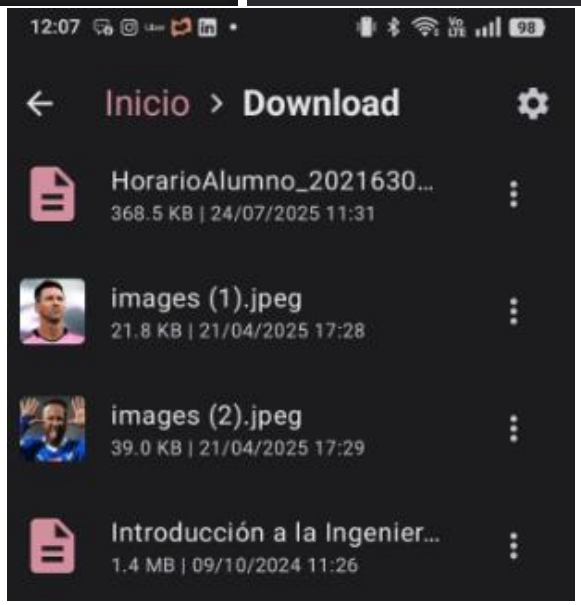
Pruebas de Operaciones (CRUD):

- Caso 2.1 (Renombrar): Seleccionar un archivo. Elegir "Renombrar". Introducir un nombre vacío. Verificar que el botón "Confirmar" está deshabilitado. Introducir un nombre válido. Confirmar. Verificar que el archivo aparece con el nuevo nombre.

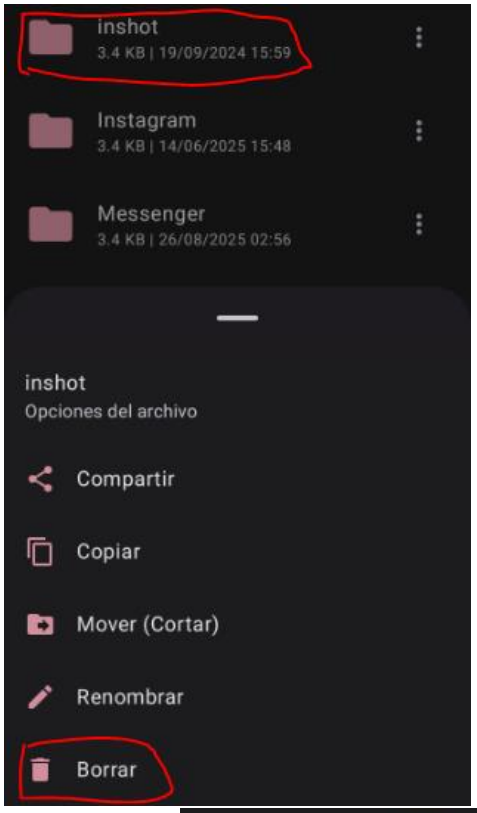


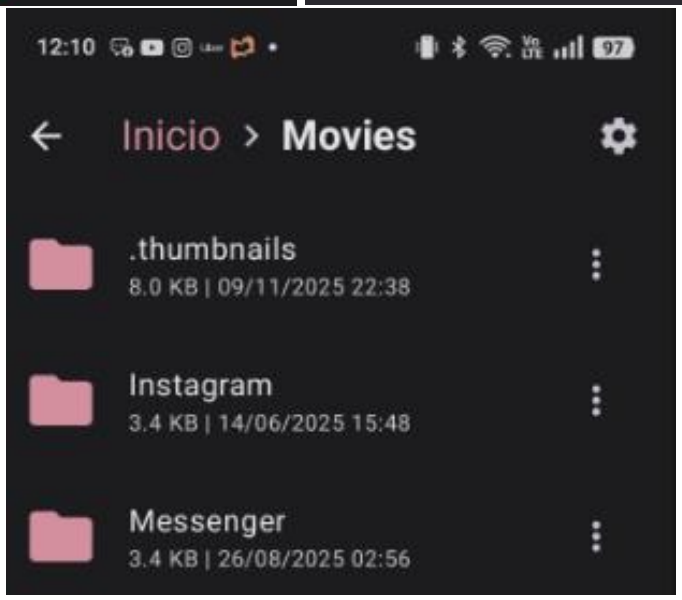
- Caso 2.2 (Borrar Archivo): Seleccionar un archivo. Elegir "Borrar". Cancelar en el diálogo de confirmación. Verificar que el archivo sigue allí. Repetir y "Confirmar". Verificar que el archivo desaparece de la lista.



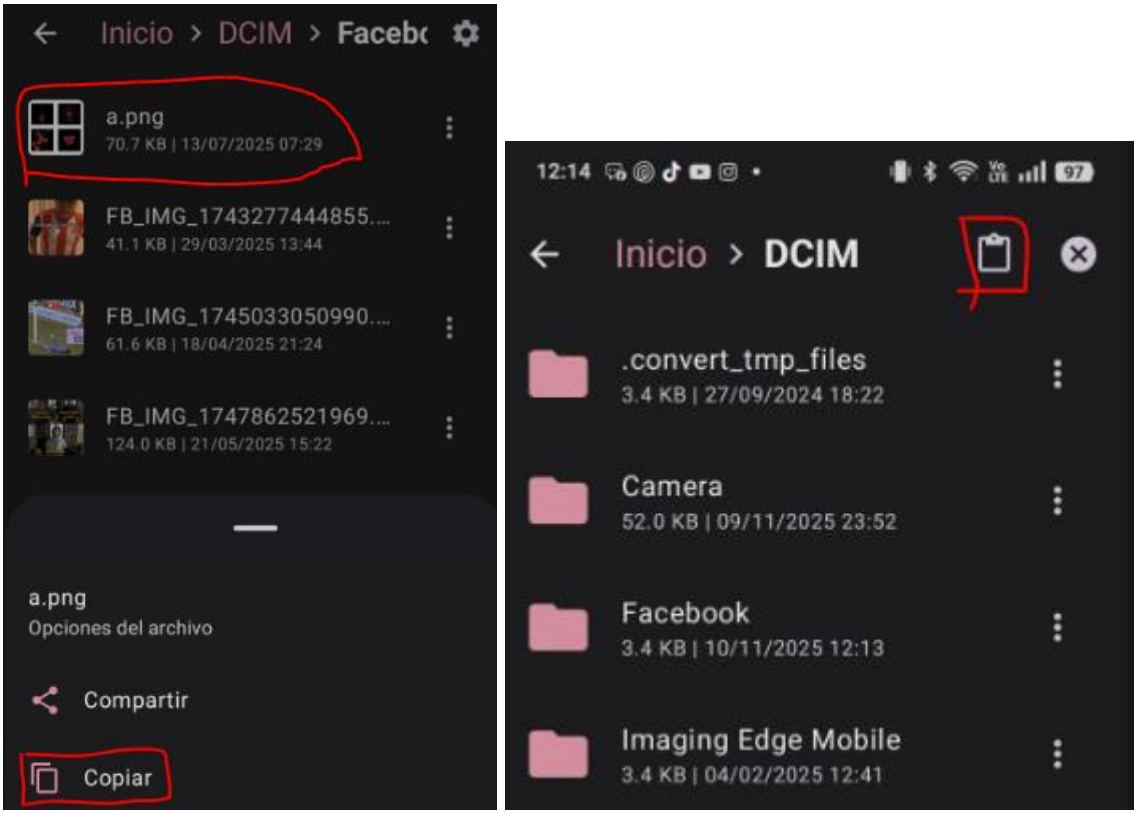


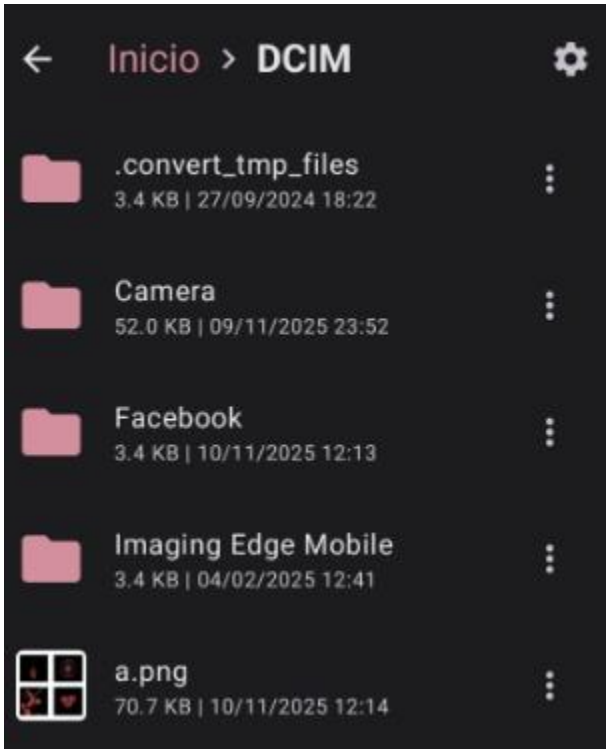
- Caso 2.3 (Borrar Carpeta): Repetir Caso 3.2 con una carpeta que contenga archivos. Verificar que la carpeta y todo su contenido se eliminan.



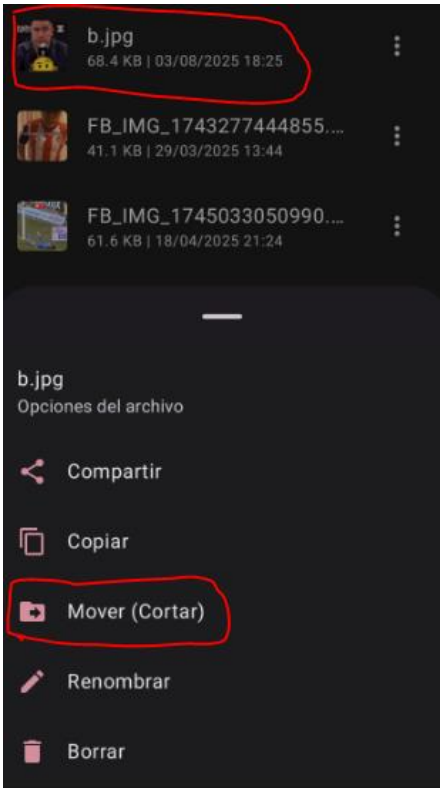


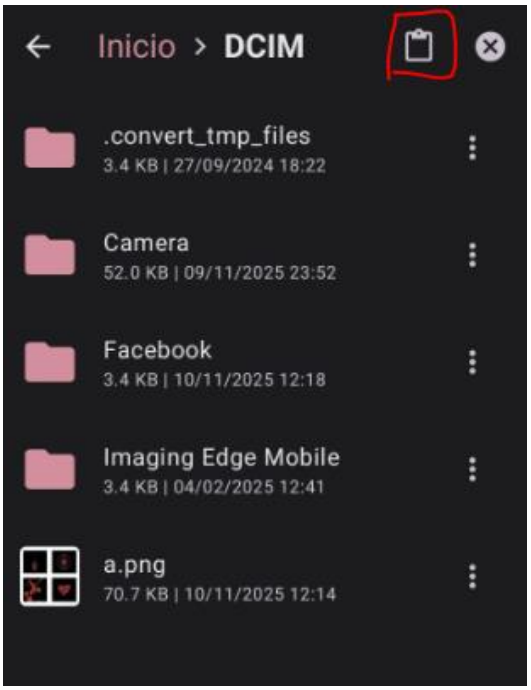
- Caso 2.4 (Copiar): Seleccionar un archivo. Elegir "Copiar". Navegar a otra carpeta. Presionar "Pegar". Verificar que el archivo aparece en la nueva carpeta y *también* permanece en la original.

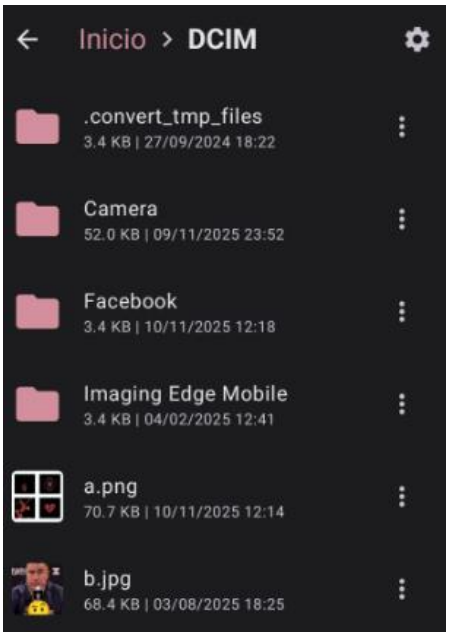


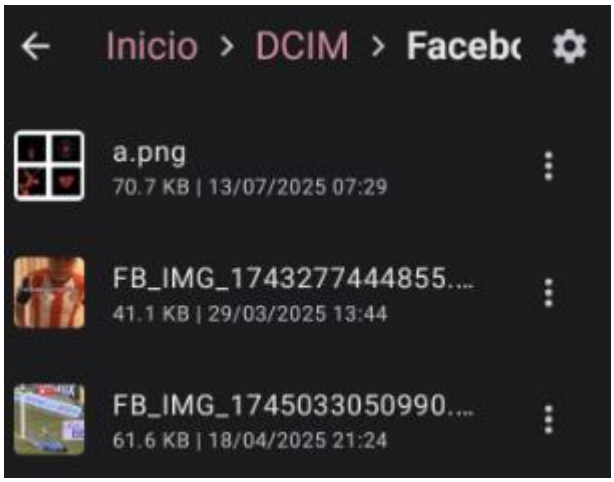


- Caso 2.5 (Mover): Seleccionar un archivo. Elegir "Mover". Navegar a otra carpeta. Presionar "Pegar". Verificar que el archivo aparece en la nueva carpeta y *desaparece* de la original.



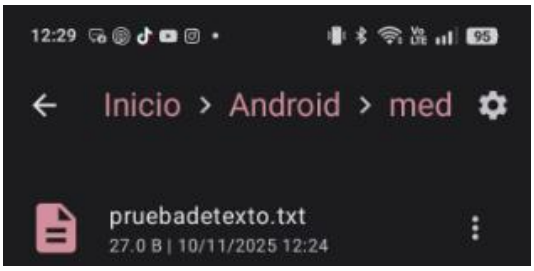


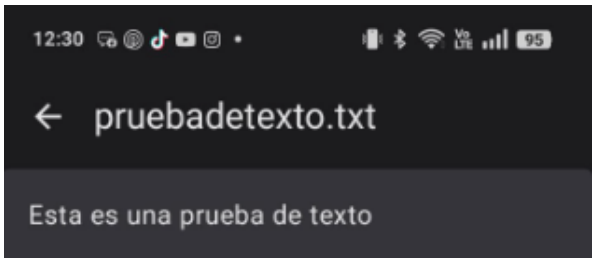




Pruebas de Visores:

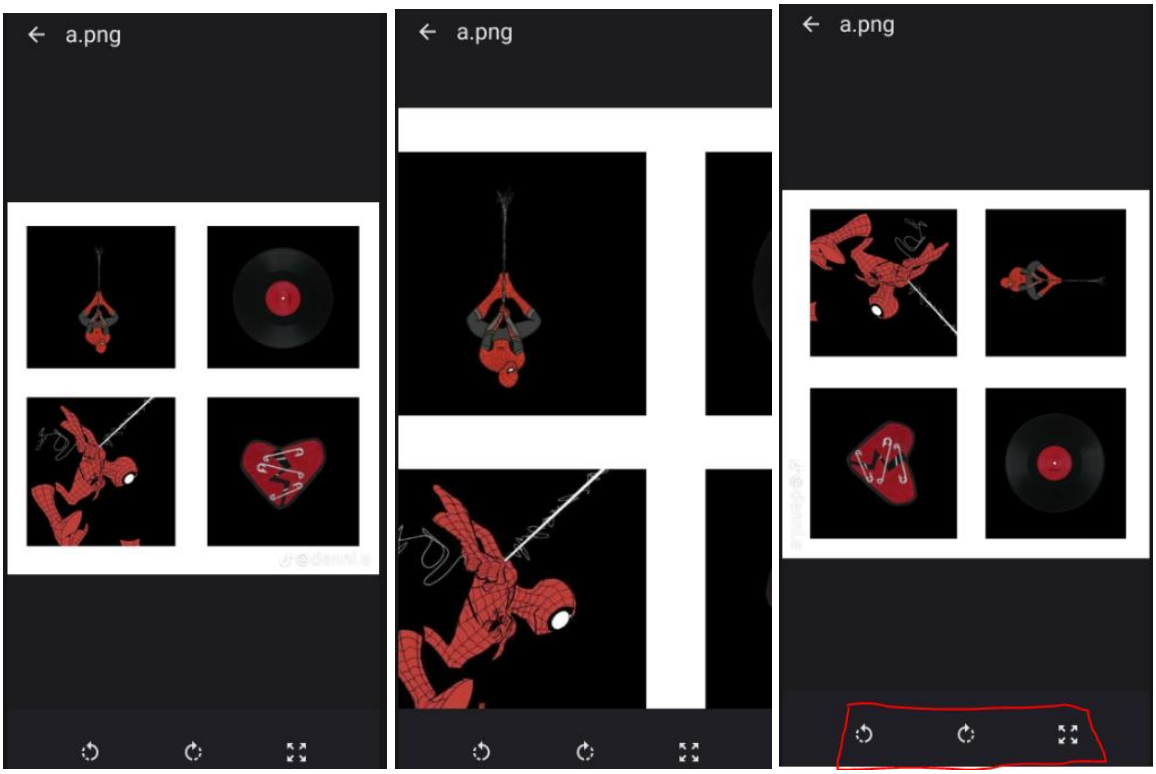
- Caso 3.1 (Texto): Hacer clic en un archivo .txt o. kt. Verificar que se abre TextViewerScreen y muestra el contenido.





- Caso 3.2 (Imagen): Hacer clic en un archivo .jpg o .png. Verificar que se abre ImageViewerScreen.

- Caso 3.3 (Zoom/Rotación): En ImageViewerScreen, pellizcar para hacer zoom. Arrastrar para mover. Usar los botones de la BottomAppBar para rotar. Verificar que la imagen responde correctamente.

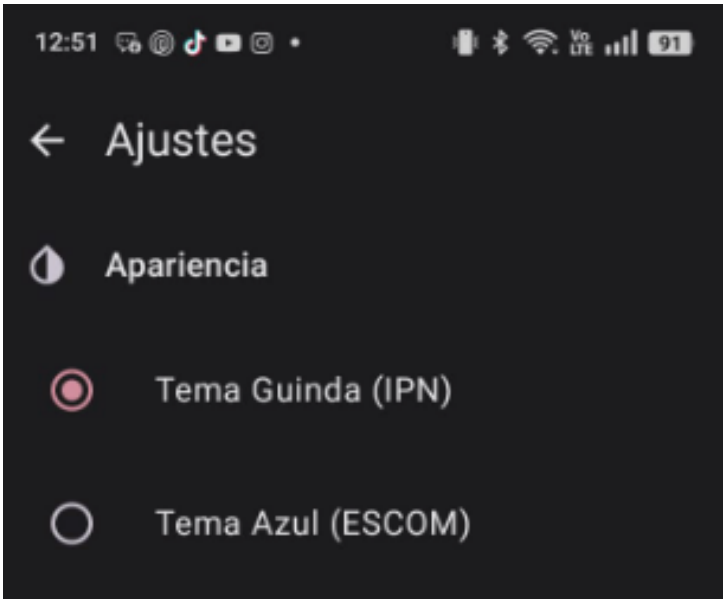


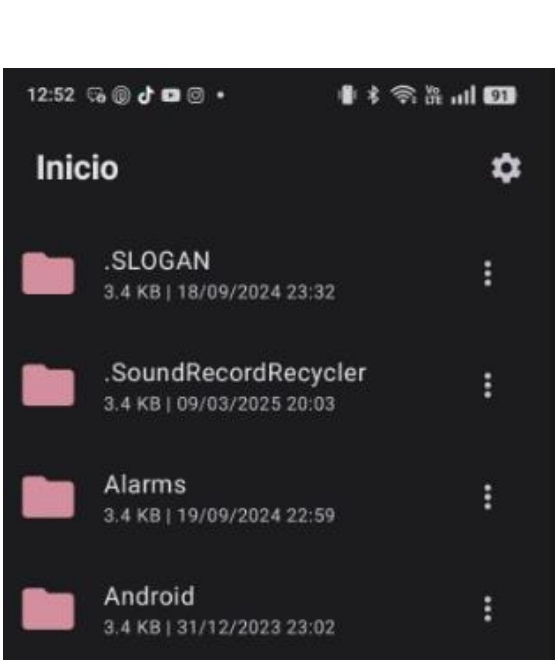
- Caso 3.4 (Abrir con): Hacer clic en un archivo no soportado (ej. .pdf). Verificar que el sistema operativo muestra el diálogo "Abrir con..." de Android.

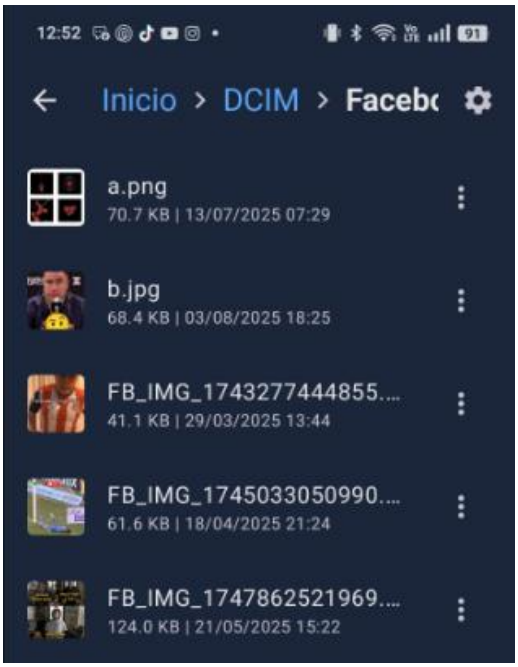
Pruebas de Interfaz (Multi-dispositivo):

- Caso 4.1 (Rotación): Girar el dispositivo a modo horizontal. Verificar que la UI se adapta y sigue siendo funcional.

- Caso 4.2 (Temas): Ir a Ajustes. Cambiar de "Tema Guinda" a "Tema Azul". Verificar que los colores primarios (iconos de carpeta, TopAppBar) cambian inmediatamente. Cerrar y volver a abrir la app. Verificar que el "Tema Azul" persiste.




12:52
Inicio
.SLOGAN
3.4 KB | 18/09/2024 23:32
.SoundRecordRecycler
3.4 KB | 09/03/2025 20:03
Alarms
3.4 KB | 19/09/2024 22:59
Android
3.4 KB | 31/12/2023 23:02


12:52
Inicio > DCIM > Faceb
a.png
70.7 KB | 13/07/2025 07:29
b.jpg
68.4 KB | 03/08/2025 18:25
FB_IMG_1743277444855....
41.1 KB | 29/03/2025 13:44
FB_IMG_1745033050990....
61.6 KB | 18/04/2025 21:24
FB_IMG_1747862521969....
124.0 KB | 21/05/2025 15:22

Conclusiones

La realización de la presente práctica ha representado una inmersión profunda y una aplicación integral de los conceptos que definen el desarrollo moderno de aplicaciones Android. Más allá de cumplir con el objetivo funcional de crear un gestor de archivos, esta práctica sirvió como un ejercicio cohesivo para ensamblar un stack tecnológico contemporáneo, enfrentando desafíos realistas del ecosistema de Android.

El pilar fundamental de la práctica fue la adopción de Jetpack Compose para la totalidad de la interfaz de usuario. Esta elección demostró ser un acierto que va más allá de la simple estética. La naturaleza declarativa de Compose simplificó drásticamente la gestión de estados complejos, como la visibilidad condicional de diálogos (RenameFileDialog, FileOptionsDialog) y la actualización reactiva de la LazyColumn en FileExplorerScreen. La gestión de estado efímero con rememberSaveable y el flujo de navegación con AppNavigationHost resultaron ser herramientas mucho más intuitivas y menos propensas a errores que el manejo de fragmentos y vistas XML tradicionales.

Arquitectónicamente, la adhesión estricta al patrón MVVM fue crucial para el éxito de la práctica. El FileViewModel se convirtió en el cerebro centralizado de toda la lógica de negocio. Al aislar las operaciones de I/O y la gestión del estado de la UI (como la ruta actual y la lista de archivos), la capa de Compose pudo mantenerse limpia, centrada únicamente en "renderizar" el estado que el ViewModel le proporcionaba. Esta separación no solo facilita las pruebas y el mantenimiento, sino que también maneja de forma nativa eventos del ciclo de vida, como la rotación de pantalla, sin perder el estado.

El aprendizaje más significativo en el ámbito del rendimiento provino del uso de Corutinas de Kotlin. Cada operación de archivo, desde un simple listFiles() hasta un deleteRecursively(), se ejecutó dentro del viewModelScope en el despachador Dispatchers.IO. Esta práctica de sacar el trabajo pesado del hilo principal es mandatoria en el desarrollo profesional para evitar errores de "Aplicación no Responde" (ANR) y garantizar una experiencia de usuario fluida, algo que los visores de texto e imágenes demuestran con su carga asíncrona.

Los desafíos más complejos no fueron de lógica de UI, sino de integración con las estrictas políticas de seguridad de Android. El manejo del permiso MANAGE_EXTERNAL_STORAGE representó un obstáculo significativo, ya que rompe el flujo de solicitud de permisos estándar y requiere guiar al usuario a una pantalla de configuración del sistema. Asimismo, la implementación de las funciones "Compartir" y "Abrir con" forzó una comprensión profunda del FileProvider, el uso de URIs content:// y la gestión de banderas de permisos de URI (FLAG_GRANT_READ_URI_PERMISSION).

Finalmente, la práctica se enriqueció con la implementación de Jetpack DataStore en el ThemeViewModel. Esto no solo añadió una característica de personalización (temas Guinda y Azul), sino que también demostró la forma moderna y asíncrona de persistir datos simples, superando las limitaciones de las SharedPreferences tradicionales.

Bibliografía

Google. (2024a). *Descripción general de ViewModel*. Android Developers. Recuperado de https://developer.android.com/topic/libraries/architecture/viewmodel

Google. (2024b). *Descripción general de los permisos de Android*. Android Developers. Recuperado de https://developer.android.com/guide/topics/permissions/overview

Google. (2024c). *Acceso a archivos desde el almacenamiento externo*. Android Developers. Recuperado de https://developer.android.com/training/data-storage/manage-all-files

Google. (2024d). *Corutinas de Kotlin en Android*. Android Developers. Recuperado de https://developer.android.com/kotlin/coroutines

Google. (2024e). *Jetpack DataStore*. Android Developers. Recuperado de https://developer.android.com/topic/libraries/architecture/datastore

Google. (2024f). *Navegación con Compose*. Android Developers. Recuperado de https://developer.android.com/jetpack/compose/navigation

Google. (2024g). *Configuración de uso compartido de archivos (FileProvider)*. Android Developers. Recuperado de https://developer.android.com/training/secure-file-sharing/set-up-sharing