Universidad de Valladolid

Máster universitario

Ingeniería Informática

Trabajo Fin de Máster

Elaboración de una guía de migración de XML a Jetpack Compose en aplicaciones Android: un caso de estudio de modernización de software

Realizado por **Juan Carlos Garrote Gascón**

✠✠✠

Universidad de Valladolid

12 de julio de 2022

Tutores: Yania Crespo González-Carvajal y Abel García de Prada

Universidad de Valladolid

Máster universitario en Ingeniería Informática

D. Yania Crespo González-Carvajal, profesora del departamento de Informática, área de Lenguajes y Sistemas Informáticos, y D. Abel García de Prada, tutor por parte de la empresa.

Exponen:

Que el alumno D. Juan Carlos Garrote Gascón, ha realizado el Trabajo final de Máster en Ingeniería Informática titulado "Elaboración de una guía de migración de XML a Jetpack Compose en aplicaciones Android: un caso de estudio de modernización de software".

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Valladolid, 12 de julio de 2022

Vº. Bº. del Tutor:	Vº. Bº. del Tutor de empresa:
D. Yania Crespo González-Carvajal	D. Abel García de Prada

Agradecimientos

A mi familia, que me apoya con todo lo que me proponga.

A mis amigos, especialmente a los de mi compañía de baile, que me han ayudado y animado todos estos intensos meses siempre que lo he necesitado.

A los amigos del Máster que ya conocía y a los que he conocido, sin vosotros este curso no habría sido lo mismo.

A los profesores del Máster, por vuestra vocación y por formarnos como buenos profesionales.

A todos los compañeros de la oficina de Izertis de Valladolid, por hacerme sentir como uno más desde el primer momento y colaborar con lo que les he pedido.

A Raquel, jefa de la oficina de Valladolid de Izertis, por facilitarme todo lo posible mi paso por el Máster y confiar en mí para desarrollar este trabajo en convenio con la empresa.

A Abel, mi tutor de Izertis y compañero de proyecto, por compartir todo tu conocimiento conmigo, que no es poco, y ayudarme en todo lo que he necesitado.

A Yania, mi tutora académica, por su gran ayuda y compromiso no sólo conmigo, sino con todos los alumnos de la Escuela, a pesar de no tener ni un minuto de descanso para ella.

Muchas gracias

Resumen

Las interfaces de usuario de aplicaciones móviles han evolucionado mucho en los últimos años. En concreto, en el campo del desarrollo de apps Android, originalmente las interfaces de usuario sólo se podían definir mediante archivos XML basándose en un paradigma imperativo, pero recientemente se ha introducido el paradigma declarativo como una forma de definir interfaces de usuario mucho más efectiva, al menos en teoría, mediante un nuevo kit de herramientas: Jetpack Compose.

Dada la creciente popularidad de este fenómeno y que la gran mayoría de interfaces de usuario de aplicaciones Android aún siguen definidas en paradigma imperativo, el presente Trabajo de Fin de Máster se dedica a desarrollar un caso de estudio que se utiliza como medio de apoyo para elaborar lo que es el principal objetivo de este trabajo: una guía de migración para interfaces de usuario en Android de paradigma imperativo (XML) a paradigma declarativo (Jetpack Compose). Finalmente, basándose en el caso de estudio, se contrastan las hipótesis planteadas en relación a los beneficios que supone el uso de Jetpack Compose frente a XML en la construcción de interfaces de usuario de apps Android.

Descriptores

Android, Jetpack Compose, interfaz de usuario, paradigma declarativo, migración software, modernización software

Abstract

User interfaces in mobile applications have evolved a lot in the latest years. Specifically, in the Android app development field, originally user interfaces could only be defined through XML files based in an imperative paradigm, but recently the declarative paradigm has been introduced as a much more effective way of defining user interfaces, at least in theory, through a new toolkit: Jetpack Compose.

Given the growing popularity of this phenomenon and that the majority of Android apps' user interfaces are still defined in an imperative paradigm, the present Master Thesis aims to develop a case study which is used as means of support to work out what is the main objective in this work: a migration guide for user interfaces in Android from imperative paradigm (XML) to declarative paradigm (Jetpack Compose). Finally, based on the case study, the stated hypothesis related to Jetpack Compose benefits versus XML in Android user interfaces are contrasted.

Keywords

Android, Jetpack Compose, user interface, declarative paradigm, software migration, software modernization

Índice general

Índice general **IV**

Índice de figuras **VI**

Índice de tablas **VIII**

1. Introducción **1**
- 1.1. Planteamiento . . . 1
- 1.2. Contexto . . . 1
- 1.3. Motivación . . . 2
- 1.4. Objetivos . . . 5
- 1.5. Estructura del documento . . . 6

2. Marco teórico **7**
- 2.1. Interfaces de usuario móviles . . . 7
- 2.2. Paradigma imperativo vs paradigma declarativo . . . 8
- 2.3. Android Jetpack . . . 11
- 2.4. Jetpack Compose . . . 12

3. Estado del arte **14**
- 3.1. Estado del arte en modernización de software . . . 14
- 3.2. Estado del arte en Jetpack Compose . . . 17

4. Caso de estudio **20**
- 4.1. Descripción . . . 20
- 4.2. Proceso seguido . . . 22
- 4.3. Resultado . . . 35

5. Guía de migración **38**
- 5.1. Condiciones iniciales . . . 38
- 5.2. Estrategias de migración . . . 42

5.3. Glosario 46
5.4. Estado en *composables* 65

6. Resultados y discusión **68**
6.1. Resultados obtenidos y consecución de objetivos 68
6.2. Discusión final 71
6.3. Contraste de hipótesis 71

7. Conclusiones y líneas de trabajo futuras **76**

Apéndices **78**

Apéndice A Plan de proyecto **79**
A.1. Herramientas utilizadas 79
A.2. Planificación del trabajo 80
A.3. Ejecución del trabajo 81

Bibliografía **82**

Índice de figuras

2.1. Componente de historias con varias noticias [37] 9
2.2. Flujo de información de los eventos en una IU con paradigma declarativo [23] 9
2.3. Flujo de información de los datos en una IU con paradigma declarativo [23] . . 10
2.4. Arquitectura recomendada y flujo de datos en la capa de IU 11
4.5. Pantalla *“Share”* de la aplicación Android de ownCloud 21
4.6. *Composable* `SharedFileRow` con archivo PDF 25
4.7. *Composable* `SharedFileRow` con imagen 25
4.8. *Composable* `SharedFileRow` con carpeta 25
4.9. *Composable* `SectionHeader` con un título y con botón de añadir 26
4.10. *Composable* `SectionHeader` con otro título y sin botón de añadir 26
4.11. *Composable* `EmptyListText` para la sección *“Users and groups”* 27
4.12. *Composable* `EmptyListText` para la sección *“Public links”* 27
4.13. *Composable* `ShareUserItem` para una compartición con un usuario 28
4.14. *Composable* `ShareUserItem` para una compartición con un grupo 29
4.15. *Composable* `WarningText` 29
4.16. *Composable* `SharePublicLinkItem` con un enlace representado por su nombre 31
4.17. *Composable* `SharePublicLinkItem` con un enlace representado por su token . 31
4.18. Pantalla completa *“Share”* de una imagen no compartida después de la migración 35
4.19. Pantalla completa *“Share”* de una carpeta compartida después de la migración 35
4.20. Pantalla original de ownCloud (izquierda) y réplica con Jetpack Compose (derecha) 36
5.21. Esquema de la arquitectura recomendada para apps Android [14] 39
5.22. Esquema de la capa de IU de la arquitectura recomendada para apps Android [14] 39
5.23. Esquema de la capa de datos de la arquitectura recomendada para apps Android [14] 40
5.24. `Modifier` aplicando primero *padding* (izquierda) y aplicando primero el borde (derecha) 48
5.25. Tipos de `horizontalArrangement` para `Row` [6] 51
5.26. Tipos de `verticalArrangement` para `Column` [6] 51

6.27. Pantalla original de WhatsApp (izquierda) y réplica con Jetpack Compose (derecha) 69
6.28. Componente de encabezado de sección utilizado en la prueba 74

Índice de tablas

5.1. Correspondencias de las propiedades típicas de los componentes de IU 47
5.2. Correspondencias de la vista XML `LinearLayout` (horizontal) 50
5.3. Correspondencias de la vista XML `LinearLayout` (vertical) 50
5.4. Correspondencias de la vista XML `Spinner` 52
5.5. Correspondencias de la vista XML `RecyclerView` 53
5.6. Correspondencias de la vista XML `RecyclerView` (horizontal) 53
5.7. Correspondencias de la vista XML `CardView` 55
5.8. Correspondencias de la vista XML `MaterialCardView` 55
5.9. Correspondencias de las vistas XML `Toolbar` y `MaterialToolbar` 55
5.10. Correspondencias de la vista XML `BottomAppBar` 56
5.11. Correspondencias de la vista XML `BottomNavigationView` 56
5.12. Correspondencias de la vista XML `TextView` 57
5.13. Correspondencias de la vista XML `EditText` 58
5.14. Correspondencias de la vista XML `Button` 59
5.15. Correspondencias de la vista XML `MaterialButton` 59
5.16. Correspondencias de la vista XML `ImageButton` 60
5.17. Correspondencias de la vista XML `CheckBox` 60
5.18. Correspondencias de la vista XML `RadioButton` 60
5.19. Correspondencias de la vista XML `Switch` 61
5.20. Correspondencias de la vista XML `FloatingActionButton` 61
5.21. Correspondencias de la vista XML `ImageView` 62
5.22. Correspondencias de la vista XML `ImageView` (icono) 63
5.23. Correspondencias de la vista XML `ProgressBar` (circular) 64
5.24. Correspondencias de la vista XML `ProgressBar` (horizontal) 64
5.25. Correspondencias de la vista XML `SeekBar` 64
5.26. Correspondencias de la vista XML `SurfaceView` 65
6.27. Resultados de la prueba en desarrolladores con nivel básico en Android 74
6.28. Resultados de la prueba en desarrolladores con nivel medio en Android 74
6.29. Resultados de la prueba en desarrolladores con nivel avanzado en Android . . 74

1: Introducción

En este primer capítulo se presentarán algunos aspectos básicos de este trabajo. Se describirá el problema planteado, el contexto en el que se desarrolla, la motivación y justificación del mismo, los objetivos que se desean conseguir con la realización del trabajo y la estructura que sigue el presente documento.

1.1. Planteamiento

Actualmente, en el contexto de la programación de aplicaciones móviles, y en concreto en Android, el sistema operativo que domina el mercado de dispositivos móviles junto a iOS [41], la interfaz de usuario (o IU) posee dos maneras principales de definirse de forma nativa: bien mediante XML (*eXtensible Markup Language*) apoyándose en lógica desarrollada en Java o Kotlin con un paradigma imperativo, o bien directamente mediante código Kotlin con un paradigma declarativo, para lo que se utiliza Jetpack Compose [12], un kit de herramientas moderno de desarrollo de IU nativas. Todo indica que esta última manera será el claro futuro del desarrollo de interfaces de usuario en Android, pero el paso de apps construidas en XML a Jetpack Compose no es automático.

El problema que se pretende abordar en este trabajo es el de cómo convertir apps Android ya desarrolladas, con su interfaz de usuario definida en XML, a Jetpack Compose, para lo que se propone elaborar una **guía de migración** que permita consultar los pasos a seguir, las técnicas a emplear y buenas prácticas a aplicar en el proceso. Esta migración se trata de un proceso de modernización de software, es decir, un proceso de reingeniería en el que partes de un sistema legado son modificadas para satisfacer ciertos requisitos con el sistema resultante. En el Capítulo 3 se profundizará más sobre este tema.

1.2. Contexto

El trabajo se realiza en el contexto del **Trabajo de Fin de Máster** para el Máster en Ingeniería Informática de la Universidad de Valladolid. Los primeros pasos en este sentido

se dieron durante la parte práctica de la asignatura *I+D+i en Informática*, en concreto, a través de una estancia en la oficina de Valladolid de la empresa **Izertis S.A.** [40].

Izertis es una consultora tecnológica que facilita la transformación digital de las organizaciones a través de la tecnología. Su oficina de Valladolid se especializa en desarrollo de aplicaciones móviles tanto Android como iOS, y a pesar de que actualmente no tienen en cartera ningún proyecto que se desarrolle con Jetpack Compose, su introducción es inminente, por lo que el resultado y el desarrollo de este trabajo es de gran interés para la empresa. Para la puesta en práctica de Compose, se ha utilizado una aplicación Android de código abierto: **ownCloud** [51]. ownCloud es una aplicación que permite almacenar archivos en un servidor propio de forma segura, así como compartirlos, siempre con conocimiento de dónde residen esos datos. Esta aplicación es un proyecto en el que Izertis lleva trabajando más de 10 años, por lo que está ampliamente consolidada y es suficientemente grande.

Dado el interés y el alcance del trabajo comenzado en la estancia en empresa, se propuso darle continuidad en el TFM en convenio con Izertis. Por tanto, el objetivo del período de estancia en empresa ha sido obtener una formación sólida en Jetpack Compose, así como la creación de varios ejemplos y la migración de una pantalla real seleccionada de la aplicación de ownCloud para que pueda constituir el caso de estudio base. Por otro lado, durante el período de realización del TFM, el objetivo es elaborar una guía de migración de XML a Jetpack Compose en aplicaciones Android lo más completa posible dentro de las restricciones temporales. Además, si el resultado de migrar partes de dicha aplicación a Jetpack Compose fuera satisfactorio, cabe la posibilidad de ampliar el trabajo en un futuro y comenzar una migración completa en la aplicación partiendo del trabajo aquí desarrollado.

Actualmente no existe mucha documentación oficial ni muy extensa sobre migración de XML a Jetpack Compose. Existen ciertas páginas de documentación que introducen aspectos muy generales y métodos para poder realizar este proceso, pero en ningún caso se indica en un paso a paso o en un glosario de componentes de IU cómo transformar cada uno de los elementos, los cuales tienen sus propias características que los diferencian de los demás. Por tanto, el punto del que se ha partido es esta documentación general, que se podría considerar que está al mismo nivel que el resto de la documentación que se ha consultado en el período de formación. Debido a esto, el trabajo a desarrollar es en su mayor parte de generación de conocimiento propio, algo que encaja muy bien en el ámbito de la investigación. Esto se analiza en más detalle en el Capítulo 3, dedicado al estado del arte.

1.3. Motivación

Jetpack Compose se ha dado a conocer como un avance importante en la forma de crear interfaces de usuario en Android. En teoría, la utilización de Jetpack Compose frente a XML implica algunas ventajas: se necesita **menos código** para el mismo resultado, es **más intuitivo**, **acelera el desarrollo** de aplicaciones y es **más potente** puesto que

proporciona utilidades que no están presentes de forma nativa en XML [12]. Además, la tendencia en los últimos años para toda la industria de desarrollo de aplicaciones (no sólo móviles) ha sido moverse al paradigma declarativo [23], debido a que simplifica enormemente el proceso asociado a construir y actualizar IU, como ya se puede ver con herramientas muy populares como Swift UI para iOS o Flutter para aplicaciones móviles y web, entre otras. Por ello, uno de los intereses de este trabajo es contrastar empíricamente las hipotéticas ventajas que supone Jetpack Compose frente a XML.

Muchas aplicaciones populares e importantes hoy en día ya se están pasando a Jetpack Compose, como por ejemplo: Square [20], Cuvva [9], o más recientemente, un gigante como Twitter [24]. Esto también ayuda a fomentar que otras aplicaciones se decidan a llevar a cabo su transformación dadas las ventajas que ello supone, y a aquellas que se van a crear desde cero, comenzar con Compose desde un inicio.

Por otro lado, además de justificar la importancia del trabajo, cabe destacar por qué merece la pena poner el foco en este tema. La primera versión estable de Jetpack Compose fue publicada en julio de 2021, hace menos de un año a fecha de realización de este trabajo. Debido a ello, y a pesar de que Google ofrece muy buena documentación sobre la herramienta, actualmente no existe mucha literatura sobre cómo transformar aplicaciones que utilizan XML a Compose, que es precisamente el objetivo que se persigue aquí. Teniendo en cuenta la importancia que puede suponer esto en un futuro cercano, se considera que es un campo en el que se pueden realizar investigaciones que den lugar a aportaciones relevantes para la comunidad de desarrolladores Android.

Viabilidad de la automatización del proceso

Puesto que se propone realizar una guía de migración, resulta lógico preguntarse por qué no automatizar el proceso en su lugar. Existen varias razones por las que no es posible o viable realizar el proceso de transformación de XML a Jetpack Compose automáticamente.

La razón principal que imposibilita la migración automática es que en Jetpack Compose, a día de hoy, **aún no existe la totalidad de componentes de IU que existían en XML**, o algunos de sus atributos. Son pocos los que aún no es posible expresar en Jetpack Compose puro, pero para las interfaces de usuario que contengan alguno de esos componentes, no existe un correspondiente directo. Asimismo, algunos de los componentes para los que sí hay soporte, sobre todo los que se han añadido más recientemente, están aún en fase experimental y es posible que manifiesten fallos en su comportamiento debido a su carácter no estable. Para solventar este problema, se requiere hacer una migración manual, en cuyo caso sí sería muy útil una guía de migración.

Otras razones que no imposibilitan pero dificultan enormemente la migración automática son:

- La definición de interfaces de usuario en paradigma imperativo consta de archivos tanto XML como Java o Kotlin, mientras que con un paradigma declarativo, la IU se expresa exclusivamente con código Kotlin. A pesar de que XML es un lenguaje

estructurado, con algunas formas de gestionar las vistas es posible que el XML contenga lógica que se debe tener en cuenta también. Además, con un paradigma imperativo son necesarios mínimo 2 archivos para expresar la IU (uno XML y otro con código Java o Kotlin), mientras que con un paradigma imperativo, aunque se puede modularizar, todo puede estar contenido en un mismo archivo Kotlin. Todos estos factores aumentan la complejidad de automatización.

- Es posible que existan componentes de IU personalizados que no son los proporcionados de forma nativa por Android. Estos componentes poseen sus propias características, como dimensiones y estado interno, entre otras, lo que hace complejo poder migrarlos ya que habría que crear un componente personalizado a partir de los nativos que proporciona Android. En el caso de aquellos cuyo código se desconoce, es aún más complejo e incluso podría llegar a imposibilitar la migración automática, puesto que no siempre es fácil imitar el aspecto visual del componente e inferir cómo se gestiona su estado interno.

- La forma de migrar es altamente dependiente del código que se encuentre, puesto que no siempre se define de la misma manera programáticamente una IU, incluso dentro de un mismo paradigma, y tampoco se posee siempre una misma arquitectura de aplicación que permita asumir cómo se organiza el código, existiendo también numerosas variantes dentro de una misma arquitectura. Esto crea un gran número de posibilidades distintas, lo que aumentaría la complejidad de la migración automática exponencialmente.

- Aunque en el alcance de este proyecto no se contempla, la forma de probar las IU en un paradigma declarativo es totalmente distinta a como se hacía en paradigma imperativo, dado que la forma en la que está programada y el concepto de componente cambian.

- Existen diversas formas de gestionar una IU con XML. Las más típicas son el método `findViewById` para crear referencias a los distintos componentes, la técnica de *View Binding* [25] o la biblioteca de *Data Binding* [10], o incluso combinaciones de las mismas en una misma pantalla. Esto implica que para poder automatizar la migración, habría que o bien restringirse a uno de esos métodos, cuando no hay ninguno que sea más aceptado que los demás, o bien proporcionar soporte para todos ellos, lo cual no sería viable dado el número de posibilidades y la complejidad que cada una de ellas conlleva.

En conclusión, dada la característica que imposibilita la automatización que puede darse en algunos casos, y las varias razones que implicarían tener en cuenta demasiados escenarios distintos, se considera que **actualmente no es viable automatizar el proceso de migración de XML a Jetpack Compose**. No obstante, como en todo nuevo problema, se necesita un conocimiento profundo de su dominio antes de abordar su automatización, por lo que contar con esta guía de migración manual podría considerarse un primer paso

para obtener dicho conocimiento y detectar reglas y patrones que permitan en un futuro comenzar a abordar la automatización progresivamente.

1.4. Objetivos

Para establecer las metas que se quieren cumplir con la realización del trabajo, se han determinado diversos objetivos que han guiado el camino a seguir.

Objetivo general

Se ha definido un objetivo general que expresa la esencia del problema:

- **Crear una guía de migración de XML a Jetpack Compose en Android.** La consecución de este objetivo implica que se ha podido desarrollar de manera exitosa la investigación y que se han hallado una o varias maneras para transformar cada elemento de interfaz de usuario en XML a Jetpack Compose, manteniendo el mismo aspecto y comportamiento.

Objetivos específicos

A partir del objetivo general, se proponen los siguientes objetivos específicos:

- **Obtener formación y conocimiento sobre Jetpack Compose.** Puesto que es una herramienta relativamente nueva y en la experiencia personal no se había utilizado nunca, ni en proyectos personales ni profesionales, ha sido necesario aprender sobre ella para poder ponerla en práctica.
- **Convertir pantallas de aplicaciones con su IU en XML a Jetpack Compose.** Este objetivo implica que las pantallas transformadas conserven el mismo aspecto y el mismo comportamiento, aunque se definan de distinta manera.
- **Extraer patrones de las transformaciones de IU realizadas.** Para poder elaborar la guía de migración final, se debe partir de conclusiones obtenidas de la práctica de transformación de interfaces de usuario. Principalmente, se buscan patrones y reglas que por lo general se cumplen al convertir de un modo de definición de IU a otro.
- **Contrastar las hipótesis que se plantean con el uso de Jetpack Compose.** Tal y como se menciona en la documentación oficial, utilizando Jetpack Compose se necesita menos código, dicho código es más intuitivo, se acelera el desarrollo y permite desarrollar IU de manera más potente. Con la puesta en práctica de la herramienta, se contrastarán estas hipótesis de manera empírica.

1.5. Estructura del documento

El presente documento se estructura de la siguiente forma:

Capítulo 1: Introducción. En este capítulo se introduce el problema abordado con el trabajo, el contexto en el que se desarrolla el mismo, los aspectos que justifican su importancia y qué objetivos se persiguen.

Capítulo 2: Marco teórico. Para poder facilitar la comprensión del trabajo y el contexto teórico del mismo, en este capítulo se sintetizan y explican los conceptos necesarios.

Capítulo 3: Estado del arte. A pesar de que Jetpack Compose es una herramienta novedosa, existen ciertos trabajos relacionados con la misma. En este capítulo se analiza la situación de la que se parte en cuanto a conocimiento y estudios sobre el tema, así como el estado del arte en cuanto a modernización de software, lo que permite enmarcar teóricamente el trabajo en un contexto más general.

Capítulo 4: Caso de estudio. Para la posterior elaboración de la guía de migración, se ha desarrollado un caso de estudio que sirve de referencia. En este capítulo se describen los detalles de este caso de estudio base y el resultado final obtenido.

Capítulo 5: Guía de migración. Este capítulo conforma la parte principal del trabajo. En él, se sintetiza el conocimiento adquirido a través del desarrollo del caso de estudio para elaborar el objetivo final del proyecto: la guía de migración de XML a Jetpack Compose.

Capítulo 6: Resultados y discusión. En este capítulo se presentan los resultados obtenidos, se realiza un análisis de éstos a la vista de los objetivos planteados, y se hace un resumen del avance del conocimiento a lo largo de todo el trabajo. Finalmente, a partir de estos resultados, se contrastan las hipótesis planteadas sobre Jetpack Compose de forma empírica.

Capítulo 7: Conclusiones y líneas de trabajo futuras. Para finalizar, se exponen las conclusiones finales obtenidas, y se proponen distintas maneras de continuar el trabajo aquí comenzado en un futuro.

Apéndice A: Plan de proyecto. En este apéndice, se describen las herramientas utilizadas y la metodología seguida en el proyecto, se detalla el plan inicial de trabajo y se realiza un breve seguimiento del proyecto sobre la planificación.

2: Marco teórico

Para encuadrar teóricamente y poder comprender los distintos aspectos del trabajo, es necesario familiarizarse con algunos conceptos a priori, que son los que se describirán en este capítulo. Estos conceptos están esencialmente relacionados con interfaces de usuario en entornos móviles, las diferencias entre paradigma imperativo y paradigma declarativo en este contexto, y la naturaleza y características de la herramienta principal objeto de este proyecto: Jetpack Compose.

2.1. Interfaces de usuario móviles

La **interfaz de usuario** (o **IU**) de un sistema está formada por todos aquellos elementos gráficos que el usuario puede ver y con los que puede interactuar. En definitiva, una IU es la manera que tiene el usuario de comunicarse con la aplicación en cuestión. Existen ciertos principios de diseño y guías de estilo a las que atenerse para crear interfaces de usuario con un alto grado de usabilidad ("efectividad, eficiencia y satisfacción con las que unos usuarios específicos logran alcanzar unos objetivos específicos en un contexto específico", ISO 9241 [33]).

En el desarrollo de aplicaciones móviles, la interfaz de usuario y la experiencia de usuario son elementos muy significativos, puesto que juegan el papel más importante en el éxito de una app [27]. En general, la IU en apps móviles está constituida por diversas pantallas con las que el usuario interactúa esencialmente de manera táctil, percibiendo cambios visuales ante cambios de estado. Al igual que en el resto del campo de interfaces de usuario, existen algunos principios y guías a seguir para construir buenas IU móviles. Sin embargo, dada la rápida evolución de la movilidad en los últimos años, aún no existen estándares de diseño para interfaces móviles, sino que gran parte del diseño se basa en los principios y guías utilizados en aplicaciones para otras plataformas, como escritorio o web [55].

Aun así, existe una corriente ampliamente aceptada en el diseño de interfaces de usuario para aplicaciones móviles: el diseño centrado en el usuario. El objetivo de esta filosofía de diseño es poner el foco en el usuario final para maximizar su satisfacción y experiencia de

usuario con el uso de la aplicación. En el caso de Android, que es lo que concierne en este proyecto, estas ideas se plasman en *Material Design*.

Material Design [46] es un sistema de guías de estilo, componentes y herramientas que utiliza las mejores prácticas (*best practices*) del diseño de interfaces de usuario en sistemas interactivos. Entre las principales características del diseño presentado en estas guías, se busca que los elementos de la interfaz de usuario simulen texturas reales, como papel, y se fomenta la utilización de tarjetas y sombras para representar y organizar información en pantalla. Fue creado por Google, y comenzó siendo utilizado en sus aplicaciones y webs, pero más tarde se puso a disposición de todos los desarrolladores para poder utilizarlo libremente en todas las aplicaciones que se deseara. Aunque principalmente es una filosofía dirigida a las apps Android, también es extrapolable a las aplicaciones web.

Jetpack Compose aplica por defecto gran parte de las recomendaciones indicadas por *Material Design*, por lo que poner en práctica dicho sistema de diseño con el uso de esta herramienta es fácil y directo.

2.2. Paradigma imperativo vs paradigma declarativo

Jetpack Compose proporciona una API declarativa para construir interfaces de usuario en Android. Para entender qué implicaciones tiene esto, es necesario conocer la diferencia entre paradigma imperativo y paradigma declarativo en la definición de IU [23].

Con un paradigma imperativo, se debe indicar a cada uno de los componentes de IU cómo se debe comportar en respuesta a cada evento o interacción que ocurra, en caso de que tenga que cambiar, con el fin de modificar su estado interno. Cada componente almacena por sí mismo su estado interno, y ofrece métodos para leerlo o modificarlo. Es decir, construir una IU mediante programación imperativa implica indicar el *cómo* se debe comportar la interfaz de usuario en cada momento, con cada evento que ocurra.

Con un **paradigma declarativo**, los componentes no poseen estado interno en el sentido de que no ofrecen métodos para poder leer o modificarlo, sino que cada vez que se actualiza el estado de la aplicación en respuesta a eventos o interacciones, se construyen de nuevo con distintos argumentos. Es decir, construir una IU mediante programación declarativa implica indicar el *qué* se debe mostrar en la interfaz de usuario en función del estado actual de la aplicación.

Por ejemplo, consideremos una pantalla de noticias formada por distintos componentes de historias o *stories*, cada uno con varias noticias, tal y como se muestra en la Figura 2.1.

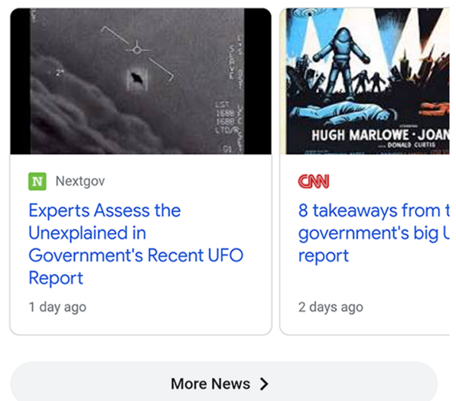


Figura 2.1. Componente de historias con varias noticias [37]

Con un paradigma imperativo, si se interactúa con una de esas noticias (por ejemplo, dando un click sobre ella), habría que actualizar la lógica de negocio e indicar cómo se debe comportar visualmente la noticia con la que se ha interactuado (por ejemplo, mostrarla en grande) y cada una del resto de noticias en pantalla (por ejemplo, ocultarlas). Con un paradigma declarativo, al interactuar con una de esas noticias, la información fluiría para actualizar la lógica de negocio, como se muestra en la Figura 2.2.

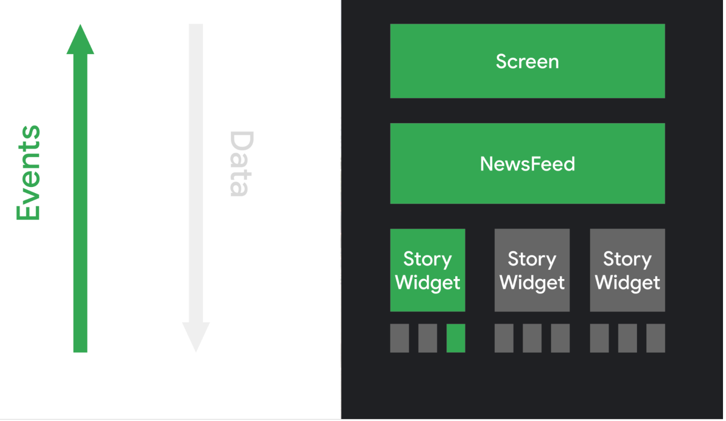


Figura 2.2. Flujo de información de los eventos en una IU con paradigma declarativo [23]

Al haber cambiado el estado de la aplicación, la IU se construiría de nuevo, y dado que se utilizaría para ello el estado que se acaba de actualizar, cada componente se construiría con los argumentos adecuados que muestran el estado actual, como se puede ver en la Figura 2.3. Este proceso de regeneración de componentes de IU se denomina **recomposición**.

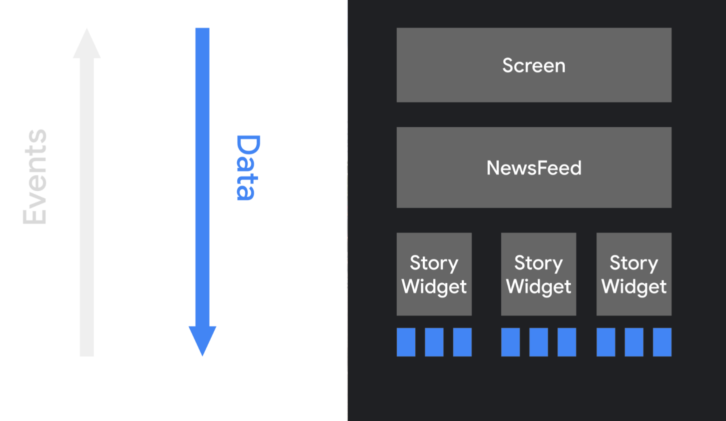


Figura 2.3. Flujo de información de los datos en una IU con paradigma declarativo [23]

Definir interfaces de usuario de forma declarativa posee una importante ventaja frente a definirlas de manera imperativa. En un paradigma imperativo, los componentes de IU se deben manipular cada vez que ocurre un evento. Modificar componentes de forma manual incrementa la probabilidad de aparición de errores, puesto que cuantos más componentes haya que mantener actualizados, más complejo será mantener un estado consistente, perjudicando así a la mantenibilidad del sistema. Por ello, la ventaja que ofrece el paradigma declarativo es que no es necesario manipular los componentes de IU, sino que indicando desde un inicio qué se debe mostrar en función del estado actual, la interfaz de usuario se regenera completamente de forma automática cuando ocurre un cambio de estado. Esto simplifica enormemente el proceso de actualización de IU, dado que con cada recomposición la IU se generará en el estado correcto sin necesidad de manipular sus componentes, disminuyendo por ende la probabilidad de errores.

Además, esta filosofía encaja actualmente muy bien con la arquitectura recomendada por Google para las aplicaciones Android, que consiste en una separación de responsabilidades por capas, y dentro de la capa de IU, la utilización del patrón MVVM (Model-View-ViewModel) [15]. Este patrón consiste en la utilización de observables[1] dentro de los ViewModels. Los ViewModels son las clases encargadas de manejar la lógica para obtener el estado actual de la aplicación y el nexo con las otras capas de más bajo nivel, tal y como se muestra en la Figura 2.4. Los observables son observados desde las clases de IU, y cuando hay un cambio en el estado de la aplicación, se notifica inmediatamente a los observadores de la IU. El paradigma declarativo puede beneficiarse de esto considerando que con cada cambio en los observables percibido por los observadores, se debe lanzar el proceso de recomposición para actualizar la interfaz de usuario, sin necesidad de hacer ninguna manipulación adicional.

[1]**Observable:** sujeto que mantiene una lista de observadores y los notifica cuando su estado cambia. **Observador:** sujeto que mantiene una referencia a un observable y es notificado cuando éste cambia. Estas entidades pertenecen a una instanciación de una arquitectura *publish-subscribe* [47] implementada mediante el patrón de diseño Observador [52].

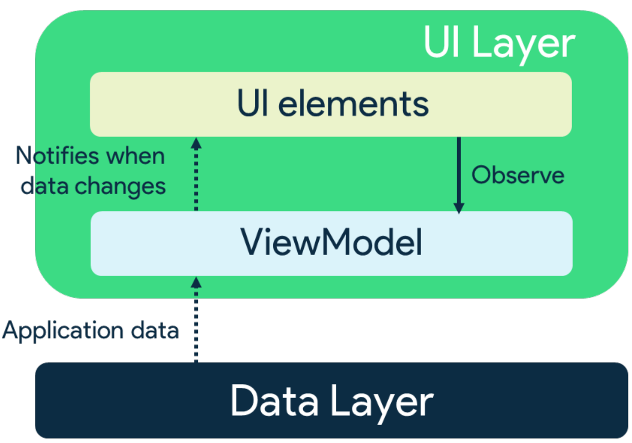


Figura 2.4. Arquitectura recomendada y flujo de datos en la capa de IU

2.3. Android Jetpack

Android Jetpack [4] es un conjunto de bibliotecas nativas de Android lanzado por Google en 2018, optimizadas para el lenguaje de programación Kotlin. Estas bibliotecas están diseñadas para ayudar a los desarrolladores a aplicar buenas prácticas en el código, reducir la cantidad de código repetitivo (*boilerplate code*) y escribir código que funciona de forma consistente en los distintos dispositivos y versiones de Android, con compatibilidad hacia atrás. El objetivo final de Jetpack es poder centrarse en el código importante para acelerar el desarrollo de aplicaciones Android y reducir el número de errores en el código para construir aplicaciones robustas.

Los componentes de Jetpack se dividen en cuatro categorías distintas, que se pueden adoptar independientemente pero que están diseñadas para funcionar en conjunto [44]:

- ***Foundation.*** En esta categoría se incluyen los componentes básicos del sistema, como las actividades de Android (pantallas), extensiones de Kotlin y bibliotecas de testing.
- ***Architecture.*** Las bibliotecas pertenecientes a esta categoría se centran en el soporte a la arquitectura de la aplicación, como puede ser la navegación, el ciclo de vida de la aplicación o la persistencia de los datos.
- ***Behaviour.*** Esta categoría incluye aquellas bibliotecas que permiten la interacción con el usuario a través de la IU, como aquellas relacionadas con permisos, multimedia o notificaciones.
- ***UI.*** En esta categoría se incluyen las bibliotecas relacionadas con la interfaz de usuario de la aplicación, como bibliotecas de animaciones, emojis, fragmentos, entre otras.

Jetpack Compose es parte de Android Jetpack, y aunque cada vez se incluyen más bibliotecas y la división en estas cuatro categorías se va dejando atrás por ser insuficiente, Compose se encuadra dentro de la categoría *UI* dada su clara relación con la interfaz de usuario de la aplicación.

2.4. Jetpack Compose

Jetpack Compose [12] es un kit de herramientas moderno para desarrollo de interfaces de usuario en aplicaciones Android. Es una utilidad nativa de Android y proporciona una API declarativa. Se puede utilizar exclusivamente con el lenguaje de programación Kotlin (recomendado por Google para el desarrollo de apps Android [5]) y a partir de la API 21 (Android 5.0 ó Android Lollipop).

El funcionamiento de Compose se basa en crear funciones que reciben datos por parámetro y emiten IU de forma idempotente (siempre que se llama con los mismos argumentos se obtiene el mismo resultado), sin efectos laterales. Estas funciones corresponden a los componentes de IU, que se denominan ***composables*** y en el código se indican con la anotación `@Composable`.

Una desventaja de la recomposición, en la que se basa el paradigma declarativo, es que la regeneración de toda la IU consume muchos recursos computacionales. Compose soluciona este aspecto mediante la **recomposición inteligente**, que en lugar de regenerar toda la IU, escoge qué componentes necesitan ser dibujados de nuevo y regenera sólo éstos [22]. Para ello, Compose detecta qué componentes necesitan ser actualizados a través de los datos que reciben por parámetro. Puesto que los *composables* deben regenerarse cuando los datos que muestran hayan cambiado, Compose se asegura de actualizar sólo aquellos componentes cuyos datos de entrada, pertenecientes al estado de la aplicación, hayan sido modificados.

A pesar de que los componentes de IU en un paradigma declarativo no poseen estado interno, Compose ofrece una manera de agregar **estado** a los *composables* [21]. Esto permite mantener un valor observable en memoria que persistirá aunque el componente se vuelva a generar, para casos en los que sea necesario, además de notificar a sus observadores cuando el valor cambia (por ejemplo, en un componente campo de texto, es necesario un estado para almacenar el valor del texto que se ha escrito hasta el momento y poder dibujarlo en la IU). Para ello, Compose proporciona en su API la cláusula `remember` y sus variantes, e incluso compatibiliza el uso de ViewModels para casos de uso más complejos en los que la información a mostrar provenga de otras capas de la arquitectura. Sin embargo, se recomienda hacer uso de esta funcionalidad sólo cuando sea necesario, puesto que los *composables* con estado interno por lo general son menos reutilizables y más difíciles de probar. En la Sección 5.4 se detallará el uso del estado en los *composables*.

Por otra parte, Jetpack Compose supone un nuevo modo de desarrollar interfaces de usuario más eficientemente, pero transformar una aplicación completa ya desarrollada de un paradigma a otro puede llegar a suponer un inmenso trabajo. Para poder hacer este

proceso viable, Compose ofrece una característica muy útil: la **interoperabilidad**. Esto significa que en una misma aplicación Android pueden convivir pantallas desarrolladas en XML y pantallas desarrolladas en Compose, e incluso dentro de una misma pantalla, es posible combinar elementos de ambas opciones para poder hacer una migración paso a paso o un desarrollo mixto [16]. Para realizar migraciones, que es lo que se busca en este trabajo, esta característica es esencial, puesto que permite partir de aplicaciones de gran tamaño desarrolladas en XML que se migrarán progresivamente sin interrumpir su funcionamiento si se encuentran en producción.

Con todo esto, Jetpack Compose establece que su uso supone una ventaja en cuatro aspectos principales frente al desarrollo de IU con XML y paradigma imperativo [26]. Estos cuatro puntos serán considerados hipótesis a contrastar con el resultado de este trabajo para obtener conclusiones sobre las mismas:

- **Menos código.** Uno de los puntos en los que Compose sugiere que destaca, es que se necesita menos código para hacer lo mismo que se haría con una IU en paradigma imperativo. Una de las razones es que todo el código está escrito en Kotlin y no en XML+Kotlin ó XML+Java, como sucedía con la programación imperativa.
- **Intuitivo.** Al tratarse de un paradigma declarativo, que se basa en simplemente realizar una descripción de la IU (el *qué* se va a mostrar) con el código a través de componentes en su mayoría sin estado y más modularizados, se considera que el código es mucho más legible e intuitivo que con un paradigma imperativo.
- **Acelera el desarrollo.** Debido a las características de Compose, como la interoperabilidad, que ofrece una gran flexibilidad en la forma de programar, o las previsualizaciones de IU, que permiten ver cómo está quedando la interfaz de usuario a la par que se desarrolla, así como a las características del paradigma declarativo empleado, que reduce la complejidad en el desarrollo y mantenimiento, gracias a no tener que capturar todos los eventos y modificar el estado de los diferentes elementos de la IU de forma consistente, se considera que desarrollar aplicaciones Android utilizando dicha herramienta reduce el tiempo de desarrollo.
- **Potente.** Dado que todo se desarrolla en Kotlin y es posible aprovechar la potencia que ofrece el lenguaje, como crear vistas repetitivas con bucles o vistas dinámicas con condiciones, y que se simplifica la manera de crear animaciones, aplicar tema oscuro, mejorar la accesibilidad y utilizar *Material Design*, Jetpack Compose sugiere que es más potente que su paradigma antecesor.

3: Estado del arte

Previo a este trabajo, se ha realizado un estudio sobre el estado del que se parte en cuanto a trabajos relacionados en general con modernización de software, y en particular, con migraciones de XML a Jetpack Compose en aplicaciones Android. En este capítulo se realizará una revisión de los mismos.

3.1. Estado del arte en modernización de software

Para comprender mejor el campo de la **modernización de software** y enmarcar correctamente el presente trabajo, se ha utilizado fundamentalmente el marco descrito en *"Software Modernization: Theoretical Framework"* [53]. Se trata de un marco de trabajo teórico común para trabajos relacionados con modernización de software, como lo es la migración, a través de diversas taxonomías que proporcionan un vocabulario que permite homogeneizar los términos y conceptos utilizados en este campo. Dicho marco de trabajo teórico se empleará en esta sección para contextualizar teóricamente el trabajo.

Como ya se ha introducido en la Sección 1.1, la modernización de software se define como un proceso al que se somete un sistema para redefinirlo parcial o totalmente, con el objetivo de mejorar algunas de sus características y cumplir con ciertos requisitos. Estos requisitos pueden tener relación con diversas propiedades del sistema, como su rendimiento, su seguridad o su calidad, entre muchas otras. En el marco teórico utilizado como referencia, se tratan varios aspectos de la modernización de software: cuándo es necesaria una solución de esta índole, los distintos tipos de modernización de software, los objetivos perseguidos y las razones que los motivan, diferentes aproximaciones que se pueden seguir, los procesos más adecuados a los que ceñirse cuando se lleva a cabo una solución de este tipo y otros aspectos a tener en cuenta. A continuación se hará un recorrido por cada uno de estos puntos en el dominio del problema a tratar.

En el caso de la migración de XML a Jetpack Compose en Android, se han detectado las dos características relacionadas con la necesidad de una solución de modernización de software: decadencia y obsolescencia. La **decadencia** del sistema de definición de IU mediante XML se justifica por el deterioro de algunas cualidades internas inherentes

del software, especialmente el incremento de complejidad, dado que al tratarse de un paradigma imperativo, cuanto más elementos contiene la IU, más difícil es de mantener. Esto obstaculiza la evolución del sistema tal y como es. Por otro lado, la **obsolescencia** se justifica con el deterioro de cualidades externas inherentes del software, dada la aparición de Jetpack Compose con el uso del paradigma declarativo. Esto causa la evolución del sistema hacia dichas nuevas tecnologías que ofrecen una mejor solución. Por tanto, dado que ambas características se exhiben en cierto grado, se considera que el sistema de definición de IU mediante XML en paradigma imperativo es un **sistema legado** o *legacy system* que se busca modernizar.

En cuanto a soluciones de modernización de software, en este caso la buscada se enmarcaría dentro de la categoría de **reingeniería**, puesto que se desea modificar el sistema de definición de IU ya existente, no reemplazarlo totalmente por uno nuevo. Dentro de la reingeniería, por una parte se aplicaría **modernización**, que permite subsanar la obsolescencia detectada mediante soluciones de **adaptación** (habilitando el uso de una nueva tecnología sin amenazar a la ya existente, tal y como permite la interoperabilidad de Jetpack Compose) o de **migración** (permitiendo moverse por completo a la nueva tecnología), y por otra parte se aplicarían soluciones de **renovación** para recuperarse de la decadencia del sistema legado a través de **reestructuración**, como por ejemplo aplicando refactorizaciones al código existente para adaptarse a la nueva tecnología.

Los objetivos específicos que motivan la aplicación de las mencionadas soluciones son, en este caso, un **cambio de paradigma** (de imperativo a declarativo) y una **traducción de IU** de un modo de definición (XML) a otro (Jetpack Compose).

Asimismo, también se han detectado varios **motivos** o *drivers* ofrecidos por las distintas soluciones consideradas, tanto **directos**, que poseen un impacto inmediato, como **indirectos**, que tienen unos efectos esperados. En el caso de las soluciones de modernización, el motivo directo es la habilitación de nuevas variables arquitectónicas como escalabilidad, gracias a la no-creciente complejidad que proporciona Jetpack Compose frente a XML, y modularidad, gracias a la potencia de código que ofrece Compose. Los motivos indirectos son la utilización de un servicio competitivo, la mejora de rendimiento del desarrollador y la reducción de costes, dadas las ventajas que sugiere la utilización de Jetpack Compose frente a XML. En el caso de las soluciones de renovación, como motivos directos se pueden encontrar también el fomento de escalabilidad y otras variables de calidad como descomponibilidad, mantenibilidad y facilidad de comprensión, y como motivos indirectos se encuentran también la mejora de rendimiento del desarrollador y la reducción de costes.

Desde el punto de vista del conocimiento del sistema, se tomará una **aproximación de caja gris**, puesto que es una solución híbrida entre una aproximación de caja negra (no se conoce la estructura interna del sistema, sólo sus entradas y salidas) y una de caja blanca (se conoce la estructura interna del sistema y se realiza ingeniería inversa para obtener conocimiento sobre el sistema). En este caso se conocen tanto entradas, como salidas, como estructura interna del sistema legado, pero no es necesario un proceso de ingeniería inversa porque se conoce ya el comportamiento del sistema, y tampoco es necesario profundizar

hasta el detalle de relaciones entre los distintos elementos que resulten en modelos u ontologías. Por tanto, la aproximación más adecuada es una mezcla, que permite seguir las pautas más pertinentes en cada caso.

El proceso de modernización más adecuado a este caso se ha considerado que es un **modelo en espiral**, puesto que toma como entrada un sistema y proporciona como salida ese sistema pero modificado, que es precisamente lo que se pretende conseguir con la transformación de interfaces de usuario de un paradigma a otro. En el Anexo A se trata este tema en más profundidad junto con el resto de plan de proyecto.

Se ha tenido en cuenta también, tal y como se recomienda, una **planificación estratégica**, que en el contexto de modernización de software implica seleccionar y priorizar qué partes del sistema se quieren modernizar. Esto se ha aplicado en el trabajo aquí descrito, puesto que en el caso de la migración aplicada en la app de ownCloud, se ha tenido que seleccionar una pantalla en concreto para hacer viable el proyecto. Esto ofrece una característica de **iteratividad** al proyecto que hace posible la migración. Por otra parte, también se ha realizado una **planificación operacional** acotando la duración de las tareas, paralelizándolas cuando es posible y haciendo cambios progresivos. Esto proporciona una característica de **incrementalidad**. Estas dos características mencionadas son fundamentales en cualquier proyecto de modernización de software.

Finalmente, las pruebas que se han propuesto para comprobar la validez de la solución son **pruebas de integración**, comprobando que una vez migrada la IU la app sigue funcionando de la misma manera que antes, y **pruebas de facilidad de comprensión**, para poder corroborar la hipótesis planteada sobre la intuitividad de Jetpack Compose.

Por lo tanto, sintetizando todo lo descrito, y tal y como se describe en el artículo referenciado anteriormente, "dado un sistema legado (definición de IU en XML) y unos motivos (los ya descritos), una solución de modernización de software es un proceso de reingeniería de migración o adaptación (en este caso ambas, siempre hacia Jetpack Compose) que aplica un método específico (aproximación de caja gris) para conseguir ciertos objetivos (cambio de paradigma y traducción de IU) que contribuyen a la satisfacción de los motivos dados impactando en partes específicas del sistema legado (su decadencia y obsolescencia)".

Algunos ejemplos de trabajos de modernización de software, referenciados en el artículo del marco teórico en el que se basa esta sección, y que además son de especial interés para el presente trabajo por tratarse de migraciones de interfaz de usuario son: *GUI Migration using MDE from GWT to Angular 6: An Industrial Case* [28], en el que se lleva a cabo la migración de la IU de una aplicación definida en GWT a Angular mediante un meta-modelo intermedio, *White-box modernization of legacy applications: The oracle forms case study* [43], que propone una transformación de caja blanca cambiando la arquitectura de la aplicación antigua (Oracle Forms), con un alto acoplamiento entre capa de IU y capa de datos, a una arquitectura multicapa sin degradar su calidad, y *Knowledge-based user interface migration* [50], que abunda en los problemas encontrados en las migraciones de IU y cómo es posible utilizar la ingeniería del conocimiento para migrar entre plataformas.

3.2. Estado del arte en Jetpack Compose

Una vez enmarcado el trabajo en términos más generales, se ha realizado una búsqueda de publicaciones científicas sobre **Jetpack Compose**. Como se ha mencionado anteriormente, no existe demasiada literatura actualmente sobre el tema dado que es relativamente nuevo, pero se ha procedido a realizar una búsqueda de artículos científicos en la base de datos científica en línea *Web of Science* [31]. Para una primera aproximación a la búsqueda, se ha utilizado la opción de buscar en todas las bases de datos y todas las ediciones con los siguientes parámetros:

Campo de búsqueda: `Topic` – Términos de búsqueda: `Jetpack Compose`

Dicha búsqueda, que es de las más básicas que se puede hacer, no arrojó ningún resultado, por lo que se ha procedido a realizar otras búsquedas variando tanto el campo como los términos de búsqueda. En concreto, los parámetros considerados han sido:

Campo de búsqueda: `Title` – Términos de búsqueda: `Jetpack Compose`

Campo de búsqueda: `All Fields` – Términos de búsqueda: `Jetpack Compose`

Campo de búsqueda: `All Fields` – Términos de búsqueda: `Android Compose`

Visto que con estos parámetros tampoco se conseguía encontrar ningún resultado (en los dos primeros casos) o publicaciones sin relación con el tema (tercer caso), se concluye que en esta base de datos no existen publicaciones científicas sobre Jetpack Compose. Por ello, se ha procedido a realizar una búsqueda con artículos lo más relacionados posible con el tema, con los siguientes parámetros:

Campo de búsqueda: `All Fields` – Términos de búsqueda: `Android migration`

De esta búsqueda se obtuvieron dos resultados relevantes principalmente. Se trata de casos de estudio no enmarcados en el campo de la modernización de software, sino de migración entre plataformas tecnológicas, pero es lo más relacionado que se ha conseguido encontrar al respecto. Uno de estos resultados es el artículo *"Implications on the Migration from Ionic to Android"* [29], que hace referencia a las implicaciones de migración a Android desde un framework de desarrollo de aplicaciones multiplataforma como es Ionic, tanto desde el punto de vista de los usuarios como de los desarrolladores. El otro artículo resultante de esta búsqueda que guarda algo de relación con el tema es *"TestMig: Migrating GUI Test Cases from iOS to Android"* [58]. Este paper no se centra solamente en el entorno Android, sino que pone el foco en la migración de tests de IU de aplicaciones iOS a Android, y aunque tampoco es exactamente lo que se busca, permite observar que las migraciones de distinto índole en plataformas móviles son comunes.

Fuera del ámbito científico, sí que existen ciertos materiales que ofrecen información sobre migración de XML a Jetpack Compose. Estos son artículos de la documentación oficial de Android, o incluso materiales preparados por usuarios de la comunidad de desarrolladores Android. A este tipo de textos se le denomina **literatura gris**, puesto que no están revisados por pares ni han seguido el mismo proceso que una publicación científica.

Dentro de la documentación oficial en línea de Android, y en concreto en la de Jetpack Compose, se puede encontrar un artículo que trata sobre la introducción de Jetpack Compose en apps ya existentes. En uno de sus apartados [3], se habla sobre cómo migrar vistas en XML a Jetpack Compose y mantener la app con el mismo funcionamiento y aspecto. Esto es precisamente lo que se busca con este trabajo, sin embargo, la explicación que se ofrece es muy escueta y con un solo ejemplo de migración, que concretamente corresponde a un texto. El objetivo de este trabajo es ampliar la información presente en dicho artículo y elaborar una guía detallada que comprenda todos los elementos de IU básicos para poder utilizarse como referencia a la hora de migrar una aplicación de XML a Jetpack Compose. De la misma manera, en la página de documentación dedicada a APIs de interoperabilidad [16], se explican las distintas posibilidades de migración, permitiendo mezclar XML y Jetpack Compose en una misma aplicación. Esta información se utilizará también en la elaboración de la guía para explicar las distintas formas de migración existentes actualmente.

En un contexto más extraoficial, existen distintos trabajos realizados por la comunidad de desarrolladores. Por ejemplo, [2] propone una serie de patrones, reglas y buenas prácticas para transformar interfaces de usuario de XML a Jetpack Compose. Además, proporciona una versión inicial reducida de un glosario de componentes (lo que denomina *cheatsheet*), indicando para ciertos elementos de IU en XML el correspondiente en Jetpack Compose. Este trabajo es muy similar a lo que se quiere conseguir aquí, puesto que se desarrolla un apartado de estrategias de migración y un glosario de componentes de IU que permite la consulta durante el proceso de migración, tal y como se describe en el Capítulo 5. Cabe destacar que el artículo es del 4 de abril de 2022, fecha posterior a la de comienzo de este proyecto. Dado el éxito de dicho artículo, el mismo usuario comenzó con el proyecto de escribir un libro basado en la información detallada en el artículo pero mucho más extendida y con ejemplos de los casos de uso más habituales [1], en este caso bajo suscripción de pago. Este libro es un trabajo que cumple objetivos muy similares a los que se desean cumplir con la elaboración de la guía de migración resultante de este Trabajo de Fin de Máster, por lo que en el Capítulo 6 se realizará una comparativa entre ambos.

Otro material encontrado que guarda relación con el tema aquí tratado es [49], una charla sobre estrategias de migración a Jetpack Compose impartida en la conferencia Android Worldwide. El material no está disponible en línea, puesto que se produjo en directo en abril de 2022, pero reafirma la idea de que este tema es de gran actualidad.

Tras hacer la revisión del estado del arte del campo en cuestión, se considera adecuado concluir que el trabajo aquí realizado es relevante, puesto que es útil, resuelve una necesidad real planteada por la comunidad de desarrolladores, y no posee similitudes con trabajos

accesibles públicamente hoy en día. Asimismo, tal y como se ha podido observar, comienzan a existir trabajos muy similares que se están desarrollando de manera simultánea a éste, lo que reafirma su importancia.

4: Caso de estudio

Para la posterior elaboración de la guía de migración, se ha desarrollado un caso de estudio al que se referenciará en la propia guía. En este capítulo se describirá en qué consiste dicho caso de estudio, se detallará el proceso seguido para su desarrollo y se mostrará el resultado final obtenido junto con algunas medidas.

4.1. Descripción

El caso de estudio desarrollado consiste en la migración de XML a Jetpack Compose de una de las pantallas de la aplicación Android *open-source* **ownCloud** [51]. ownCloud es una aplicación desarrollada en Java y Kotlin, actualmente en proceso de moverse a una arquitectura limpia organizada por capas, totalmente en Kotlin, y utilizando el patrón MVVM (Model-View-ViewModel). Las interfaces de usuario están descritas mediante el sistema de vistas que proporciona XML y el código en paradigma imperativo correspondiente en Java o Kotlin.

Como ya se ha mencionado, la app de ownCloud proporciona un sistema que permite almacenar y gestionar archivos en un servidor propio configurado de antemano. La ventaja principal que esto supone frente a otras aplicaciones de gestión de archivos en la nube, como por ejemplo Google Drive, Dropbox o OneDrive, es que el servidor donde los datos están almacenados es propio, lo que permite tener bajo control siempre la localización de los datos e incrementar su seguridad.

En concreto, la pantalla que se ha decidido migrar ha sido la denominada *"Share"*, mostrada en la Figura 4.5. En esta pantalla, a la que se accede a través de las opciones de un archivo o carpeta que se ha seleccionado, se realizan las tareas de compartición con otros usuarios o grupos de usuarios de dicho archivo o carpeta. Comenzando por la parte superior, se puede observar la *toolbar* de la aplicación con un botón que permite volver a la pantalla anterior y el título de dicha pantalla. Seguidamente, se encuentra una fila con un icono representativo del fichero o carpeta, su nombre y extensión (si es fichero), su tamaño (si es fichero) y un icono clicable que permite copiar el enlace privado al mismo en el servidor. Más abajo, se encuentra la sección *"Users and groups"*, en la que es posible

crear comparticiones privadas del archivo o carpeta con un usuario o grupo de usuarios concreto. Esta sección está compuesta por un encabezado que contiene el título de la sección y un botón que permite crear una nueva compartición privada, y por una lista de todas las comparticiones del fichero creadas. Para cada ítem de la lista, se muestra un icono que indica si se ha compartido con un usuario o un grupo, el nombre del usuario o el grupo, y dos botones que permiten eliminar dicha compartición o editar sus propiedades, respectivamente. Por último, debajo de la sección recién descrita, se encuentra la sección *"Public links"*, en la que es posible crear enlaces públicos al fichero o carpeta. Esta sección está formada por un encabezado análogo al de la sección anterior, un mensaje avisando de que la creación de un enlace público implica que todo aquel que lo tenga posee acceso al fichero o carpeta, y una lista con los distintos enlaces públicos creados. Cada ítem de esta lista está formado por un nombre identificativo del enlace y tres botones, que permiten copiar el enlace al portapapeles, eliminarlo o modificar sus propiedades, respectivamente.

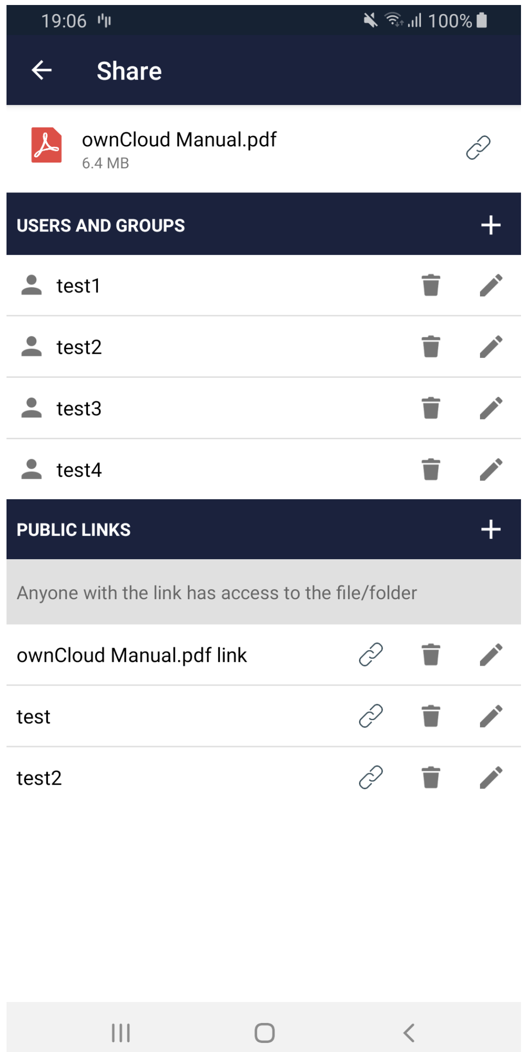


Figura 4.5. Pantalla *"Share"* de la aplicación Android de ownCloud

La decisión de seleccionar esta pantalla como caso de estudio reside, por una parte, en que la misma ya estaba migrada a la nueva arquitectura y por tanto se parten de las condiciones que se establecerán para poder seguir la guía de migración, y por otra parte, en que posee variedad de componentes de IU y una complejidad media a nivel de código, lo que la hace una pantalla completa y adecuada para constituir un ejemplo base al que referenciar.

El enlace al repositorio y rama donde se encuentra el código resultante del proceso que se explicará a continuación es el siguiente: `https://github.com/JuancaG05/owncloud-android-jetpack-compose/tree/compose/shares`. Dentro de él, la ruta que se debe seguir para encontrarlo es `owncloudApp` > `src` > `main` > `java` > `com/owncloud/android` > `presentation` > `ui` > `sharing` > `fragments` > `ShareFileComposeFragment.kt`.

4.2. Proceso seguido

El primer paso ha sido crear un *fork* del repositorio original de ownCloud [51], es decir, crear una copia del código original para poder trabajar a partir de ella. En esta nueva copia, se ha creado una nueva rama, `compose/shares`, con el objetivo de realizar los cambios pertinentes en el código y poder crear un *pull request* cuando se haya terminado el trabajo. El enlace a esta rama es el que se ha indicado anteriormente.

Aunque los comentarios de documentación interna del código se han mantenido intactos puesto que siguen siendo válidos, así como algunos de los atributos de la clase, esos y otros detalles se omitirán mediante puntos suspensivos en el código mostrado en esta sección por brevedad, poniendo el foco en los *composables* creados y su estructura.

En primer lugar, se ha creado un nuevo fragmento donde se alojará el nuevo código de Jetpack Compose, denominado `ShareFileComposeFragment.kt`. Este fragmento mantiene la misma estructura que el original, con los métodos básicos que se sobreescriben en los fragmentos de Android, y dado que se sigue la estrategia de migración por pantallas que permite mantener fragmentos utilizando Jetpack Compose (descrita en la Sección 5.2), el método `onCreateView` debe devolver un objeto `ComposeView`, tal y como se muestra en el Fragmento de código 4.1.

```
1  class ShareFileComposeFragment : Fragment() {
2      override fun onCreate(savedInstanceState: Bundle?) {
3          ...
4      }
5      override fun onCreateView(
6          inflater: LayoutInflater,
7          container: ViewGroup?,
8          savedInstanceState: Bundle?
9      ): View {
```

```
10          return ComposeView(requireContext()).apply {
11              ...
12              setContent {
13                  ...
14              }
15          }
16      }
17      override fun onAttach(context: Context) {
18          ...
19      }
20
21      override fun onDetach() {
22          ...
23      }
24
25      override fun onResume() {
26          ...
27      }
28  }
```

Fragmento de código 4.1. Estructura general del fragmento `ShareFileComposeFragment`

A partir de este momento, dentro del bloque `setContent` se pueden crear instancias de *composables*, que son los que conforman la interfaz de usuario de este fragmento.

Una vez creada la estructura básica del fragmento, el objetivo es crear *composables* personalizados para cada uno de los elementos que podemos ver en pantalla. Comenzando por la fila de detalles del fichero o carpeta a compartir, el *composable* creado es el mostrado en el Fragmento de código 4.2.

```
1   @Composable
2   private fun SharedFileRow(file: OCFile) {
3       Row(...) {
4           ...
5           if (file.isImage && thumbnail != null) {
6               Image(
7                   bitmap = thumbnail.asImageBitmap(),
8                   ...
9               )
10          } else {
11              Image(
12                  painter = painterResource(id = MimetypeIconUtil
                       .getFileTypeIconId(file.mimetype, file.
                       fileName)),
13                  ...
14              )
```

```
15            }
16            Column(...) {
17                Text(
18                    text = file.fileName!!,
19                    ...
20                )
21                if (!file.isFolder) {
22                    Text(
23                        text = DisplayUtils.bytesToHumanReadable(
                              file.fileLength, activity),
24                        ...
25                    )
26                }
27            }
28            if (!file.privateLink.isNullOrEmpty()) {
29                IconButton(
30                    onClick = { listener?.copyOrSendPrivateLink(
                          file) },
31                    ...
32                ) {
33                    Icon(
34                        painter = painterResource(id = R.drawable.
                              copy_link),
35                        ...
36                    )
37                }
38            }
39        }
40    }
```

Fragmento de código 4.2. *Composable* para la fila de detalles del fichero o carpeta a compartir

Nuevamente, y como se hará con todos los demás, se han omitido detalles no primordiales que se incluirán en el glosario de la guía de migración en la Sección 5.3 (configuración de colores, gestión de márgenes, características de accesibilidad, etc.), con el fin de centrarse en la estructura y características básicas del *composable*.

Este *composable* `SharedFileRow` recibe como parámetro un objeto `OCFile`, que representa al archivo o carpeta que se quiere compartir y del que se obtendrán los detalles para mostrar en la IU. Como se puede observar, todos los elementos del componente están contenidos en un *composable* `Row`, que permite una disposición horizontal de los elementos de IU que contiene. En primer lugar, se comprueba si el archivo a compartir es una imagen, y de ser así, se muestra una miniatura de la misma para poder identificarla más fácilmente mediante un componente `Image`. En caso de no serlo, se muestra un icono representativo para el tipo de fichero o carpeta, como por ejemplo un PDF. Tras esto, dentro de un elemento `Column`, que permite una disposición vertical de los elementos que

contiene, se encuentra un `Text` que muestra el nombre del fichero o carpeta, y en caso de no ser una carpeta, debajo del mismo aparece también el tamaño del fichero. Por último, si el enlace privado a dicho fichero o carpeta en servidor no es nulo o vacío, se añade un botón (`IconButton`) representado por un icono (`Icon`) de una cadena, que al pulsarlo permite copiar o compartir dicho enlace. El resultado de este código con distintas variantes de fichero/carpeta se muestra en las Figuras 4.6, 4.7 y 4.8.

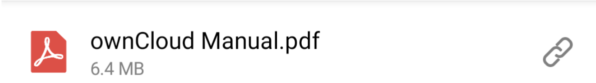


Figura 4.6. *Composable* `SharedFileRow` con archivo PDF

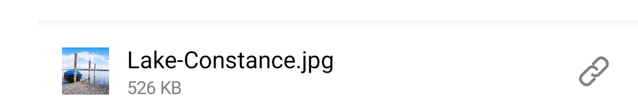


Figura 4.7. *Composable* `SharedFileRow` con imagen

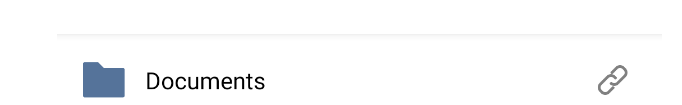


Figura 4.8. *Composable* `SharedFileRow` con carpeta

El siguiente *composable* que se ha creado es el encabezado de sección, que servirá tanto para la sección de comparticiones privadas como para la de enlaces públicos. Este *composable* es el que se muestra en el Fragmento de código 4.3.

```
1  @Composable
2  private fun SectionHeader(title: String, showAddButton: Boolean
       , onClickAddButton: () -> Unit) {
3      Row(...) {
4          Text(
5              text = title.uppercase(),
6              ...
7          )
8          if (showAddButton) {
9              IconButton(onClick = onClickAddButton) {
```

```
10                  Icon(
11                      painter = painterResource(id = R.drawable.
                           ic_add),
12                      ...
13                  )
14              }
15          }
16      }
17  }
```

Fragmento de código 4.3. *Composable* para el encabezado de sección

`SectionHeader` recibe por parámetro la cadena de texto con el título de la sección, una variable booleana que indica si se desea mostrar el botón de añadir o no, y la función que se ejecutará cuando se capture el click del botón de añadir. Todo el contenido del *composable* está englobado en una fila para que la disposición de los componentes sea horizontal. Dentro de esa fila podemos encontrar un `Text`, que imprime por pantalla en un texto el título que se pasa por parámetro en mayúsculas, y seguido, si se cumple la condición también pasada por parámetro, un icono (`Icon`) de un símbolo "+" dentro de un *composable* `IconButton` que al pulsarlo ejecuta la función pasada como tercer parámetro. Visualmente, el resultado sólo cambia cuando se indican títulos distintos o cuando se quiere mostrar el botón de añadir o no, tal y como se muestra en las Figuras 4.9 y 4.10.

Figura 4.9. *Composable* `SectionHeader` con un título y con botón de añadir

Figura 4.10. *Composable* `SectionHeader` con otro título y sin botón de añadir

Como se ha mencionado antes, las secciones que encabezan estos *composables* recién explicados contienen listas de ítems. Sin embargo, cuando la lista esta vacía, se muestra un texto con un mensaje indicándolo. Este *composable* es muy simple, como se muestra en el Fragmento de código 4.4.

```
1  @Composable
2  private fun EmptyListText(text: String) {
```

```
3       Text(
4           text = text,
5           ...
6       )
7   }
```

Fragmento de código 4.4. *Composable* para el mensaje de lista vacía

`EmptyListText` consiste en un texto que muestra el mensaje que recibe por parámetro, formateándolo con ciertas configuraciones de estilo. Para las secciones *"Users and groups"* y *"Public links"* los mensajes que indican la lista vacía son los de las Figuras 4.11 y 4.12, respectivamente.

No data shared with users yet

Figura 4.11. *Composable* `EmptyListText` para la sección *"Users and groups"*

No public links created yet

Figura 4.12. *Composable* `EmptyListText` para la sección *"Public links"*

En el caso de la sección *"Users and groups"* cada compartición privada, que es cada uno de los ítems de dicha lista, estará representada por un componente de Jetpack Compose también. Este *composable* es el que se muestra en el Fragmento de código 4.5.

```
1   @Composable
2   fun ShareUserItem(share: OCShare, unshare: () -> Unit, edit: ()
        -> Unit) {
3       Row(...) {
4           val iconId = if (share.shareType == ShareType.GROUP) R.
               drawable.ic_group else R.drawable.ic_user
5           var name = if (share.sharedWithAdditionalInfo!!.isEmpty
               ()) share.sharedWithDisplayName
6                else share.sharedWithDisplayName + " (" + share.
                   sharedWithAdditionalInfo + ")"
7           if (share.shareType == ShareType.GROUP) name =
               getString(R.string.share_group_clarification, name)
```

```
8            Icon(
9                painter = painterResource(id = iconId),
10               ...)
11           )
12           Text(
13               text = name!!,
14               ...
15           )
16           IconButton(onClick = unshare) {
17               Icon(
18                   painter = painterResource(id = R.drawable.
                        ic_action_delete_grey),
19                   ...
20               )
21           }
22           IconButton(onClick = edit) {
23               Icon(
24                   painter = painterResource(id = R.drawable.
                        ic_lead_pencil_grey),
25                   ...
26               )
27           }
28       }
29   }
```

Fragmento de código 4.5. *Composable* para un ítem de la lista de comparticiones privadas

El *composable* `ShareUserItem` recibe tres parámetros: un objeto `OCShare` que representa a la compartición y es de donde se obtienen los distintos detalles a mostrar, una función que permite eliminar la compartición en cuestión y una función que permite editar las propiedades de la compartición. El componente consiste en un `Row` que contiene un icono al principio, distinto en función de si la compartición es con un usuario o con un grupo, un texto que muestra el nombre de dicho usuario o grupo e información adicional si es pertinente, y dos iconos, uno para borrar y otro para editar la compartición, a los que se les asigna como manejadores del evento de click las dos funciones recibidas por parámetro correspondientes. Dos ejemplos del resultado visual de este código se pueden apreciar en las Figuras 4.13 y 4.14.

Figura 4.13. *Composable* `ShareUserItem` para una compartición con un usuario

group1 (group)

Figura 4.14. *Composable* `ShareUserItem` para una compartición con un grupo

En la sección *"Public links"*, lo primero que se encuentra es un mensaje avisando de que cualquiera que posea uno de los enlaces públicos tendrá acceso al archivo o carpeta. Este *composable* es sencillo y muy similar a `EmptyListText`, con la diferencia de que se colorea el fondo y que, puesto que sólo existe un mensaje de aviso en la pantalla por el momento, no es necesario que reciba por parámetro la cadena de texto ya que es única. El componente en cuestión se muestra en el Fragmento de código 4.6.

```
1 @Composable
2 private fun WarningText() {
3     Text(
4         text = stringResource(id = R.string.
              share_warning_about_forwarding_public_links),
5         ...
6     )
7 }
```

Fragmento de código 4.6. *Composable* para el mensaje de aviso

Puesto que en el presente código se omiten los detalles de estilo (como es el caso de dar color al fondo), `WarningText` luce similar a `EmptyListText` excepto porque el texto se obtiene por defecto en lugar de por parámetro. En la Figura 4.15 se muestra cómo es el resultado de instanciar este componente.

Anyone with the link has access to the file/folder

Figura 4.15. *Composable* `WarningText`

De forma análoga a la sección *"Users and groups"*, en la sección *"Public links"* se muestra una lista de ítems, cada uno de los cuales representa un enlace público creado. El *composable* empleado para cada uno de estos ítems se muestra en el Fragmento de código 4.7.

```
1 @Composable
2 fun SharePublicLinkItem(share: OCShare, copyOrSend: () -> Unit,
      remove: () -> Unit, edit: () -> Unit) {
```

```
3       Row(...) {
4           var name = if (share.name.isNullOrEmpty()) share.token
               else share.name
5           Text(
6               text = name!!,
7               ...
8           )
9           IconButton(onClick = copyOrSend) {
10              Icon(
11                  painter = painterResource(id = R.drawable.
                       copy_link),
12                  ...
13              )
14          }
15          IconButton(onClick = remove) {
16              Icon(
17                  painter = painterResource(id = R.drawable.
                       ic_action_delete_grey),
18                  ...
19              )
20          }
21          IconButton(onClick = edit) {
22              Icon(
23                  painter = painterResource(id = R.drawable.
                       ic_lead_pencil_grey),
24                  ...
25              )
26          }
27      }
28  }
```

Fragmento de código 4.7. *Composable* para un ítem de la lista de enlaces públicos

De manera similar a `ShareUserItem`, `SharePublicLinkItem` recibe por parámetro un objeto `OCShare`, haciendo referencia a la compartición que se está realizando con el enlace público, y tres funciones: una que realiza la acción de copiar o compartir el enlace público, otra que lo elimina y otra que permite editar sus propiedades. Dentro del *composable*, todo se encuentra contenido en un componente `Row` para mostrarse de manera horizontal. Lo primero que encontramos es un texto con el nombre identificativo que se ha dado al enlace, y si no tiene nombre, su token. Después encontramos tres iconos pulsables: uno para copiar o compartir el enlace, otro para borrarlo y otro para editarlo, a los que se les asigna como manejador del evento de click la función correspondiente de las que se ha recibido por parámetro. En las Figuras 4.16 y 4.17 se muestra cómo es el resultado de este *composable* en dos casos.

Figura 4.16. *Composable* `SharePublicLinkItem` con un enlace representado por su nombre

Figura 4.17. *Composable* `SharePublicLinkItem` con un enlace representado por su token

Con este último *composable* creado, ya se poseen todos los necesarios para representar los elementos de la IU de esta pantalla. Sin embargo, en el fragmento original, existen ciertos elementos del ViewModel (la clase que mantiene el estado y conecta con las capas inferiores en la arquitectura [15]) que se observan para obtener datos y ser notificado cuando éstos cambien. En el nuevo fragmento estos datos serán necesarios también, y se tratarán como estado para poder generar la IU correcta en todo momento. Para ello, Jetpack Compose proporciona un método `observeAsState` que permite observar valores, en este caso del ViewModel, y tratarlos como estado para recomponer la IU cuando sean modificados. Los datos que necesitan ser observados son las *capabilities*, que es la configuración de las características disponibles en el servidor, y las propias comparticiones, tanto públicas como privadas. En el Fragmento de código 4.8 se muestra la manera de obtener estos datos como estado.

```
1 val capabilitiesState = ocCapabilityViewModel.capabilities.
     observeAsState()
2 val capabilitiesUiResult = capabilitiesState.value?.peekContent
     ()
3 val capabilities = capabilitiesUiResult?.getStoredData()
4 ...
5 val sharesState = ocShareViewModel.shares.observeAsState()
6 val sharesUiResult = sharesState.value?.peekContent()
7 val shares = sharesUiResult?.getStoredData()
```

Fragmento de código 4.8. Observación de datos como estado

De esta manera, en las variables `capabilities` y `shares` se dispone de los valores actualizados en todo momento puesto que cuando sean modificados, el ViewModel notificará al fragmento. De ellas se podrán extraer los datos necesarios para la lógica del fragmento.

Finalmente, una vez creados todos los *composables* precisos y disponiendo de los datos necesarios como estado, se puede dar forma a la interfaz de usuario de la pantalla. La estructura general de la IU, que se coloca dentro del bloque `setContent` del método `onCreateView` del fragmento, se muestra en el Fragmento de código 4.9. En este caso, por

brevedad, se ha omitido la lógica correspondiente a los eventos de click sobre los distintos componentes, puesto que es código que ya existía anteriormente perteneciente a otras clases, y no ha sido necesario adaptarlo.

```
1  LazyColumn {
2      item { SharedFileRow(ocFile) }
3
4      if (shareWithUsersAllowed && isShareApiEnabled) {
5          item {
6              SectionHeader(
7                  title = stringResource(id = R.string.
                       share_with_user_section_title),
8                  showAddButton = true,
9                  onClickAddButton = {
10                     ...
11                 }
12             )
13         }
14         if (privateShares.isEmpty()) {
15             item { EmptyListText(text = stringResource(id = R.
                   string.share_no_users)) }
16         } else {
17             items(privateShares) { share ->
18                 ShareUserItem(
19                     share = share,
20                     unshare = {
21                         ...
22                     },
23                     edit = {
24                         ...
25                     }
26                 )
27                 Divider(...)
28             }
29         }
30     }
31
32     if (shareViaLinkAllowed && isShareApiEnabled &&
           isPublicShareEnabled) {
33         item {
34             SectionHeader(
35                 title = stringResource(id = R.string.
                       share_via_link_section_title),
36                 showAddButton = isMultiplePublicSharingEnabled
                       || (!isMultiplePublicSharingEnabled &&
                       publicShares.isEmpty()),
```

```
37                    onClickAddButton = {
38                        ...
39                    }
40                )
41            }
42            if (shareWarningAllowed) {
43                item { WarningText() }
44            }
45            if (publicShares.isEmpty()) {
46                item { EmptyListText(text = stringResource(id = R.
                   string.share_no_public_links)) }
47            } else {
48                items(publicShares) { share ->
49                    SharePublicLinkItem(
50                        share = share,
51                        copyOrSend = {
52                            ...
53                        },
54                        remove = {
55                            ...
56                        },
57                        edit = {
58                            ...
59                        }
60                    )
61                    Divider(...)
62                }
63            }
64        }
65    }
```

Fragmento de código 4.9. Estructura general de la IU de la pantalla

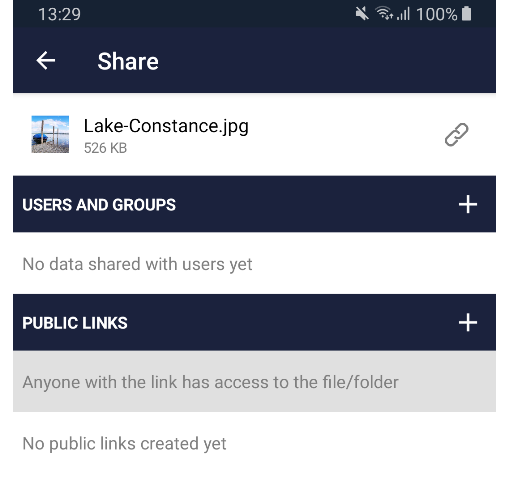
Como se puede ver, todo está contenido en un componente `LazyColumn`. Este componente permite que la disposición de sus hijos sea vertical, de la misma manera que `Column`, pero con la peculiaridad de que es posible hacer scroll por defecto y que dispone de soporte para mostrar un número indefinido de elementos de una lista, como es el caso de las listas de las secciones *"Users and groups"* y *"Public links"*. Dentro del alcance de `LazyColumn`, los elementos individuales se deben rodear con un bloque `item` y las listas de elementos con `items`, pasando por parámetro la lista con los elementos a pintar. En primer lugar, se encuentra el *composable* creado `SharedFileRow`, al que se le pasa el fichero en cuestión para pintar la fila de detalles como se ha mostrado. Después, haciendo unas comprobaciones provenientes de la configuración de la aplicación y de las *capabilities*, se crea la sección *"Users and groups"* si procede. Para ello, se crea un `SectionHeader` al que se le pasa el título de la sección, se le indica que sí se muestre el botón de añadir y se pasa la función que se ejecutará al capturar el evento del click del botón "+", y

después, se comprueba si la lista de comparticiones privadas está vacía o no, para crear un `EmptyListText` o la lista de ítems, respectivamente. En el segundo caso, al bloque `items` se le pasa por parámetro la lista con las comparticiones a representar, y para cada una de ellas se crea un componente `ShareUserItem` y un `Divider`, que no es más que la línea separatoria entre elementos. En el caso de la posterior sección *"Public links"*, se realizan algunas comprobaciones a partir de la configuración de la aplicación y las *capabilities* de nuevo, y si procede, se crea un encabezado `SectionHeader` con el título correspondiente, la opción de mostrar el botón de añadir dependiendo de los valores de ciertas *capabilities* y la función para el evento de click sobre él. Al mismo nivel que el encabezado de sección, se comprueba si se debe mostrar o no el mensaje de aviso `WarningText`, y finalmente se realiza el mismo proceso que en la anterior sección comprobando en este caso la lista de comparticiones públicas, añadiendo un `EmptyListText` si la lista está vacía o creando un `SharePublicLinkItem` y un `Divider` para cada elemento en caso contrario.

Con este código terminado, la pantalla estaría lista para mostrar la IU de manera muy similar a como lo hacía previamente y manteniendo el mismo comportamiento que antes. En las Figuras 4.18 y 4.19 se muestran dos ejemplos del resultado con dos estados distintos.

Tras finalizar el código, se ha realizado un *pull request* en el repositorio en el que se aloja el mismo. Todo lo desarrollado ha pasado un proceso de revisión de código (*code review*) por parte de un ingeniero desarrollador Android profesional de la oficina de Valladolid de Izertis, de manera que el código quede lo más limpio y simple posible, aplicando los cambios sugeridos cuando ha sido necesario. Finalmente, para concluir el desarrollo del caso de estudio, tras el proceso de revisión de código también se ha sometido lo desarrollado a un proceso de QA por parte de un ingeniero QA profesional de la oficina, de forma que se asegura que el funcionamiento es correcto y el mismo que el que ya se tenía previamente. Una vez pasados ambos procesos con éxito, se da por finalizado el desarrollo del caso de estudio.

El enlace al *pull request*, donde se puede consultar el código completo desarrollado con los detalles omitidos en este documento, el proceso de revisión de código y el proceso de QA, es el siguiente: https://github.com/JuancaG05/owncloud-android-jetpack-compose/pull/1.



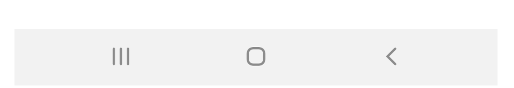


Figura 4.18. Pantalla completa *"Share"* de una imagen no compartida después de la migración

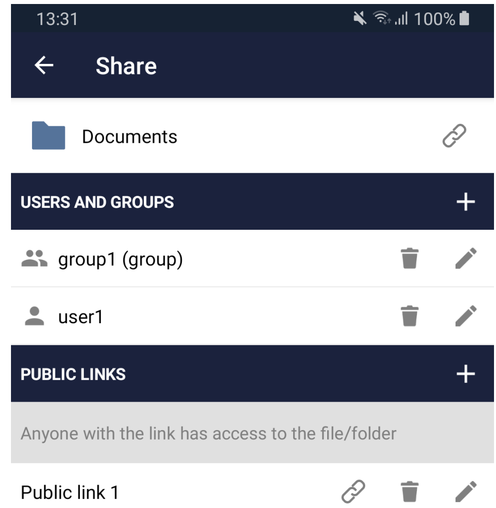


Figura 4.19. Pantalla completa *"Share"* de una carpeta compartida después de la migración

4.3. Resultado

El resultado obtenido de este caso de estudio es la pantalla *"Share"* de ownCloud migrada de un paradigma imperativo con su IU definida en XML y código Kotlin a un paradigma declarativo, utilizando exclusivamente código Kotlin y el kit de herramientas Jetpack Compose para definir la IU. En la Figura 4.20 se puede observar una comparativa de la pantalla original y la pantalla resultante para un mismo estado, pudiendo comprobar que la diferencia es mínima.

Para el desarrollo de este caso de uso, se han empleado **49 Horas-Persona**.

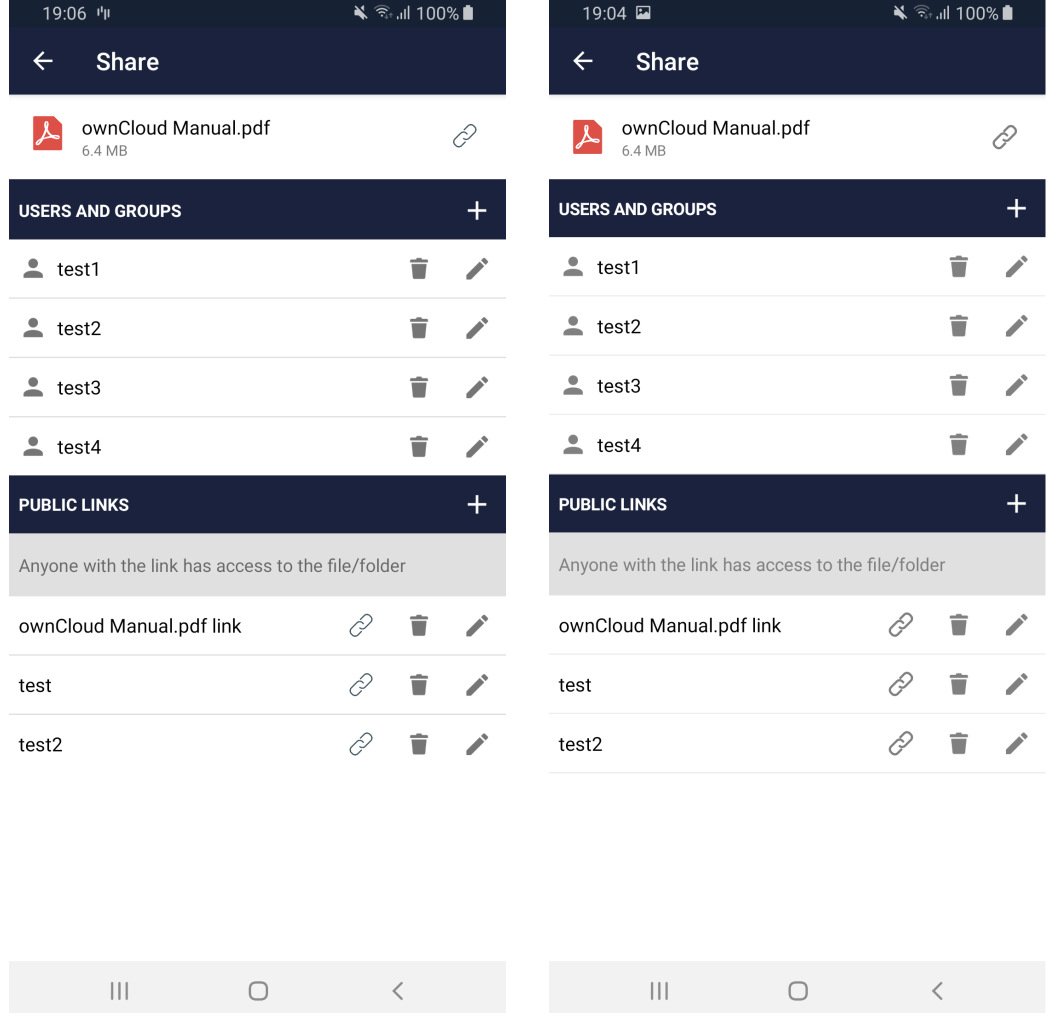


Figura 4.20. Pantalla original de ownCloud (izquierda) y réplica con Jetpack Compose (derecha)

Originalmente, para poder representar esta pantalla, los siguientes recursos de código eran necesarios, especificando el número de líneas lógicas de código (LLOC) para cada uno:

- `ShareFileFragment.kt`: 391 LLOC
- `ShareUserListAdapter.kt`: 58 LLOC
- `SharePublicLinkListAdapter.kt`: 42 LLOC
- `share_file_layout.xml`: 168 LLOC
- `share_user_item.xml`: 53 LLOC
- `share_public_link_item.xml`: 51 LLOC

Es decir, originalmente el código relativo a la pantalla *"Share"* consistía de un total de 6 archivos, tanto Kotlin como XML, que sumaban en total 763 LLOC.

Actualmente, tras la realización del caso de estudio, se poseen los siguientes recursos de código para representar la pantalla:

- `ShareFileComposeFragment.kt`: 507 LLOC

Es decir, actualmente el código relativo a la pantalla *"Share"* consiste de 1 archivo Kotlin con un total de **507 LLOC**.

5: Guía de migración

Durante la elaboración del caso de estudio, se ha extraído el conocimiento necesario para poder elaborar el artefacto objetivo del proyecto: la guía de migración de XML a Jetpack Compose para interfaces de usuario de apps Android. En este capítulo se sintetiza ese conocimiento y se dan algunas recomendaciones basadas en la experiencia.

5.1. Condiciones iniciales

Para poder seguir la guía aquí presentada, se considera que se debe partir de una aplicación Android que cumpla con ciertas condiciones.

Por una parte, la app Android debe seguir una arquitectura limpia que cumpla dos principios fundamentalmente [13]: separación de responsabilidades, de manera que cada clase se ocupe en lo posible de un solo ámbito de la aplicación, consiguiendo así una alta cohesión y un bajo acoplamiento, y control de la IU a partir del modelo de datos, de forma que los datos a mostrar sean independientes de las propias clases de IU y del ciclo de vida de sus componentes. Esto facilitará enormemente la migración, puesto que el código relativo a la IU (lo que se desea migrar) se encontrará prácticamente aislado de lo demás y no habrá que preocuparse del resto de lógica de la aplicación. Para conseguir cumplir estos principios, se propone seguir la arquitectura recomendada por Google para aplicaciones Android.

La arquitectura recomendada por Google [14] consiste en una **división por capas** y la utilización del **patrón MVVM**. En cuanto a la división por capas, se recomienda que existan mínimo 2: capa de IU y capa de datos, de manera que se cumplan los dos principios antes mencionados al separar responsabilidades por capas y hacer una distinción entre IU y modelo de datos. Opcionalmente, puede existir entre medias de ambas una capa de dominio para simplificar las interacciones entre capas. En la Figura 5.21 se puede ver un esquema con las dependencias entre capas.

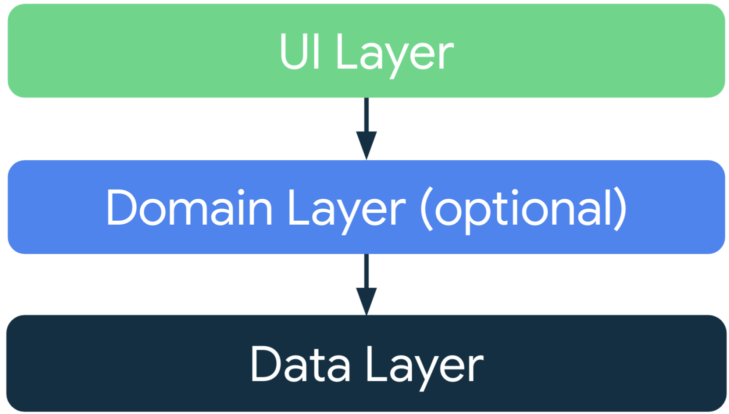


Figura 5.21. Esquema de la arquitectura recomendada para apps Android [14]

La capa de IU o capa de presentación permite mostrar los datos de la aplicación por pantalla. Para ello, la capa se subdivide en dos partes: **elementos de IU**, que son los que pintan en pantalla los distintos componentes mediante vistas XML o Jetpack Compose, y **contenedores de estado**, que mantienen datos de la aplicación, manejan la lógica relativa a éstos y los exponen a la IU. Cuando estos datos cambian, la IU debe actualizarse para reflejar dichos cambios. En la Figura 5.22 se muestra un esquema de la capa de IU con las dependencias entre sus distintas partes.

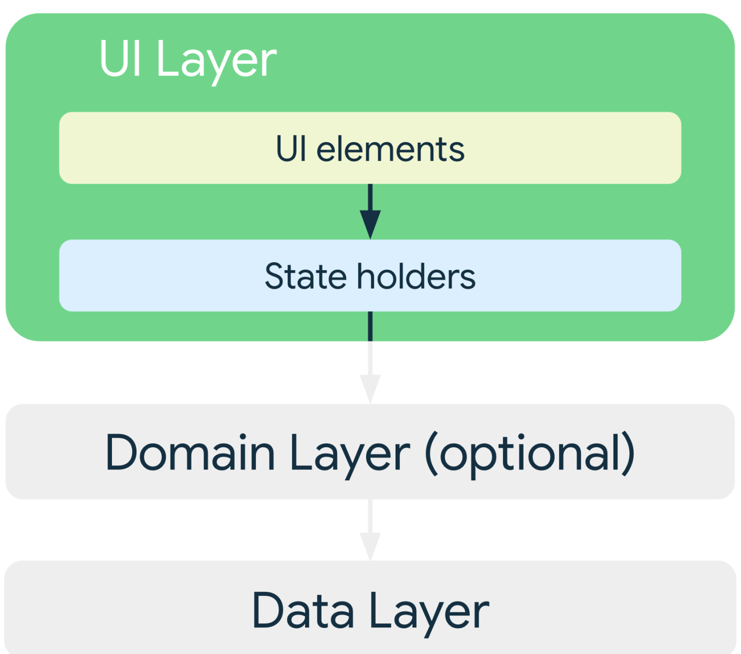


Figura 5.22. Esquema de la capa de IU de la arquitectura recomendada para apps Android [14]

En este caso, la guía se centra en la parte de elementos de IU. En una app con una arquitectura como la que se está describiendo, la migración consistiría simplemente en reemplazar dichos elementos de IU, que serían vistas en XML, por componentes de Jetpack Compose. En la parte de contenedores de estado es donde entra en juego el patrón MVVM (Model-View-ViewModel), puesto que un ViewModel es un contenedor de estado que permite gestionar datos de una forma compatible con el ciclo de vida de los componentes de IU. También, el patrón MVVM lleva implícita la utilización de una programación reactiva, donde los cambios se notifican a los observadores cuando se producen, tal y como se explica en la Sección 2.2.

La capa de datos se encarga de gestionar la lógica de negocio de la aplicación. Dentro de ella se pueden encontrar dos partes: los **repositorios**, que se encargan de operar con los datos de las fuentes de datos apropiadas y exponerlos al resto de la aplicación, y los propios **accesos a fuentes de datos**, que gestionan un solo origen cada uno de ellos, ya sea un fichero, la red o una base de datos local. En la Figura 5.23 se muestra la estructura interna de esta capa.

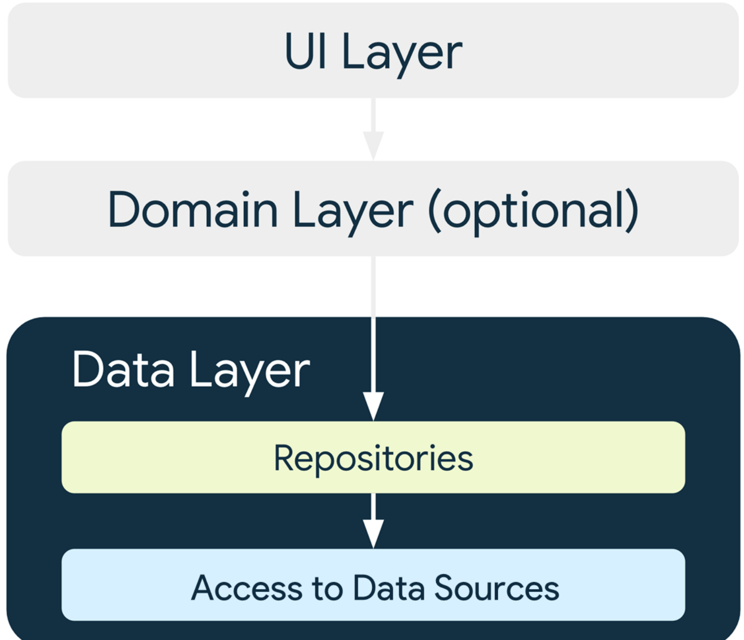


Figura 5.23. Esquema de la capa de datos de la arquitectura recomendada para apps Android [14]

Finalmente, la capa de dominio, que es opcional, se encarga de gestionar lógica de negocio compleja o que es reutilizada por varios contenedores de estado (ViewModels) de

la capa de IU. Dentro de ella, lo más común es encontrar la implementación de **casos de uso** o escenarios de historias de usuario que permiten ejecutar una funcionalidad en la aplicación.

Por otra parte, la aplicación Android a migrar debe tener al menos la **API 21 de Android SDK** (Android 5.0) como API mínima, puesto que Jetpack Compose está disponible a partir de esa versión. Asimismo, la versión de Jetpack Compose está ligada a la versión de Kotlin utilizada [7], por lo que para esta guía se ha utilizado la última versión estable de Compose: **Jetpack Compose 1.1.1**, que corresponde con la versión de **Kotlin 1.6.10**. Para establecer estas propiedades y habilitar el uso de Jetpack Compose, se debe añadir lo indicado en los Fragmentos de código 5.10 y 5.11 en los archivos `build.gradle` del nivel del proyecto y `build.gradle` del nivel de la app, respectivamente.

```
1  ext {
2      composeVersion = "1.1.1"
3      kotlinVersion = "1.6.10"
4  }
```

Fragmento de código 5.10. Propiedades necesarias en el `build.gradle` del nivel del proyecto para habilitar Jetpack Compose

```
1   ...
2   android {
3       ...
4       defaultConfig {
5           minSdkVersion 21
6           ...
7       }
8       compileOptions {
9           sourceCompatibility JavaVersion.VERSION_1_8
10          targetCompatibility JavaVersion.VERSION_1_8
11      }
12      buildFeatures {
13          compose true
14      }
15      composeOptions {
16          kotlinCompilerExtensionVersion composeVersion
17      }
18      kotlinOptions {
19          jvmTarget = JavaVersion.VERSION_1_8.toString()
20      }
21      ...
22  }
23  ...
```

Fragmento de código 5.11. Propiedades necesarias en el `build.gradle` del nivel de la app para habilitar Jetpack Compose

Finalmente, una vez establecidas estas propiedades, sólo queda añadir las **dependencias de Jetpack Compose** necesarias en el archivo `build.gradle` del nivel de la app. Las dependencias básicas, que son las que se han utilizado en el desarrollo de esta guía, son las que aparecen en el Fragmento de código 5.12.

```
...
dependencies {
    ...
    implementation "androidx.compose.ui:ui:$composeVersion"
    implementation "androidx.compose.material:material:
       $composeVersion"
    implementation "androidx.activity:activity-compose:1.4.0"
    implementation "androidx.compose.runtime:runtime-livedata
       :1.1.1"
    debugImplementation "androidx.compose.ui:ui-tooling:
       $composeVersion"
    ...
}
...
```

Fragmento de código 5.12. Dependencias de Jetpack Compose necesarias en el `build.gradle` del nivel de la app

Cumplidas las anteriores condiciones y preparado el entorno de desarrollo, ya se está en posición de poder comenzar a desarrollar código con Jetpack Compose.

5.2. Estrategias de migración

Existen tres opciones principales para migrar interfaces de usuario de XML a Jetpack Compose [16] [45]. A continuación se describirán las alternativas existentes y finalmente se realizará una recomendación al respecto.

En cualquiera de las tres estrategias que se describen, se pueden utilizar dos enfoques distintos a la hora de migrar la IU:

- **De abajo hacia arriba *(bottom-up)*.** Se comienza migrando los elementos de IU más pequeños, y de forma progresiva se va construyendo la pantalla completa.
- **De arriba hacia abajo *(top-down)*.** Se comienza migrando los elementos de IU que contienen a todos los demás, y de forma progresiva se van migrando todos los elementos contenidos.

Migración completa

La alternativa más simple consiste en realizar una migración completa de la aplicación, pantalla a pantalla, hacia Jetpack Compose. Para ello, simplemente habría que crear una

nueva actividad Android con *composables* que representen las pantallas ya existentes y mantengan su lógica, para así poder reemplazar totalmente la antigua definición de IU por la nueva. Para adaptar la navegación existente al nuevo sistema, se hace uso del componente de navegación para Compose, cuya explicación se sale del ámbito de este trabajo (para más información consultar la documentación oficial al respecto [19]). Se trata de una solución conceptualmente simple, pero con la desventaja de que habría que paralizar la aplicación (o al menos no sacar versiones nuevas) hasta que todas las pantallas estén migradas y se haya probado que su funcionamiento es el mismo que poseía anteriormente.

El resultado será una aplicación con su IU completamente definida mediante *composables* de Jetpack Compose y sin utilización de XML para la IU.

Migración por pantallas

Otra alternativa más flexible es realizar una migración progresiva por pantallas completas. Esta forma de trabajo permite que las nuevas características o pantallas que se añaden a la aplicación se definan totalmente en Jetpack Compose, o bien transformar pantallas completas ya existentes al paradigma de Compose. Para ello, existen dos casos posibles.

En el caso de que la pantalla que se desea migrar esté alojada en una actividad Android, se debe crear una nueva actividad que sustituirá a la anterior de la forma que se expresa en el Fragmento de código 5.13. En los fragmentos de código de esta sección se omiten las importaciones de las dependencias necesarias por brevedad y por la automatización de esta tarea desde el IDE Android Studio.

```
1 class NewComposeActivity : AppCompatActivity() {
2     override fun onCreate(savedInstanceState: Bundle?) {
3         super.onCreate(savedInstanceState)
4         setContent {
5
6         }
7     }
8 }
```

Fragmento de código 5.13. Nueva actividad Android con su IU definida completamente con Compose

De esta manera, dentro del bloque `setContent` ya se podrían empezar a añadir *composables* para definir la IU completamente con Jetpack Compose.

En el caso de que la pantalla a migrar se encuentre alojada en un fragmento y se quiera mantener dicho fragmento, se debe proceder como se muestra en el Fragmento de código 5.14.

```
1 class NewComposeFragment : Fragment() {
2     override fun onCreateView(
3         inflater: LayoutInflater,
```

```
4           container: ViewGroup?,
5           savedInstanceState: Bundle?
6       ): View {
7           return ComposeView(requireContext()).apply {
8               setContent {
9                   setViewCompositionStrategy(
                       ViewCompositionStrategy.
                       DisposeOnViewTreeLifecycleDestroyed)
10
11              }
12          }
13      }
14  }
```

Fragmento de código 5.14. Nuevo fragmento Android con su IU definida completamente con Compose

Dentro del bloque `setContent` y después de establecer la estrategia de composición que permite mantener el comportamiento original de los fragmentos (no desechar cuando se deje de ver en pantalla, sino cuando se destruya), ya sería posible comenzar a añadir *composables* para definir la IU. En este caso, se mantiene la navegación usada anteriormente, puesto que lo único que se modifica es la forma en la que se define la IU.

Realizando este proceso, se consigue una migración progresiva de la aplicación llegando a un estado con algunas pantallas definidas completamente mediante *composables* y otras mediante vistas XML, manifestando así la interoperabilidad entre ambos métodos de definición de IU. Una vez migradas todas las pantallas, la definición de IU estaría expresada completamente en Jetpack Compose y sería posible adoptar la navegación para Compose, de manera que se consigue una migración total.

Compose y vistas XML en conjunto

La forma más flexible de migrar es aprovechar al máximo la interoperabilidad y utilizar XML y Jetpack Compose en una misma pantalla. Esto ofrece dos posibilidades: utilizar Compose en una pantalla con vistas XML como base o utilizar vistas XML en una pantalla con Compose como base.

Vistas XML como base + Compose

Para poder mantener el código XML de las vistas y añadir *composables* a la IU, es necesario añadir un elemento `ComposeView` al XML existente, como se indica en el Fragmento de código 5.15.

```
1  <androidx.compose.ui.platform.ComposeView
2          android:id="@+id/compose_view"
3          android:layout_width="match_parent"
```

```
4           android:layout_height="match_parent"/>
```

Fragmento de código 5.15. Elemento `ComposeView` en una pantalla definida en XML

Una vez añadida esta vista, desde el código Java o Kotlin que maneja la pantalla se referencia a la misma y se invoca su método `setContent`, que permite añadir *composables* que se mostrarán en la IU en el sitio ocupado por el elemento `ComposeView`. Considerando que las vistas se gestionan mediante `findViewById`, esto se realizaría como se muestra en el Fragmento de código 5.16.

```
1  val composeView = findViewById<ComposeView>(R.id.compose_view)
2  composeView.setContent {
3
4  }
```

Fragmento de código 5.16. Añadiendo código Compose a una pantalla definida en XML

De esta forma, como en los otros casos, dentro del bloque `setContent` ya es posible añadir componentes de Jetpack Compose que definirán parte de la IU.

Compose como base + vistas XML

Si el caso que nos encontramos es que se posee una pantalla definida en Compose pero se quieren utilizar vistas XML dentro de ella, por ejemplo porque aún no existe soporte en Compose para dichas vistas y sólo están disponibles en XML, se debe utilizar el *composable* `AndroidView`, tal y como se muestra en el Fragmento de código 5.17.

```
1  AndroidView(
2      factory = { context ->
3          val xmlView = LayoutInflater.from(context)
4                  .inflate(R.layout.xml_view, null, false)
5          xmlView
6      }
7  )
```

Fragmento de código 5.17. Elemento `AndroidView` en una pantalla definida con Compose

Utilizando este *composable*, es posible utilizar vistas ya definidas en XML, como en este caso la que se encuentra en el recurso `R.layout.xml_view`.

Con estas dos maneras de migración progresiva se consigue una aplicación con pantallas definidas en parte con Jetpack Compose y en parte con vistas XML, que pueden ir moviéndose hacia Jetpack Compose de forma completa según se vaya completando la migración de otras partes de la aplicación o según se ofrezca soporte en Jetpack Compose para algunas funcionalidades no disponibles actualmente.

La recomendación establecida en esta guía es utilizar la estrategia de **migración por pantallas** manteniendo las actividades o fragmentos ya existentes, tal y como se ha hecho en el caso de estudio descrito en el Capítulo 4. Esto permite realizar una migración progresiva de un grano ni demasiado fino, como el que se consigue con la estrategia de Compose y vistas XML en conjunto, que puede llegar a aumentar la complejidad y comprensibilidad del código, ni demasiado grueso, como en la estrategia de migración completa. Además, de esta manera es posible centrarse solamente en cómo definir la IU y mantener su gestión tal y como estaba, por ejemplo con la navegación. El enfoque a seguir recomendado es **de abajo hacia arriba**, puesto que de esta manera es posible ir visualizando el resultado con cada pequeño componente que se migre, y se puede comprender cómo se construye la pantalla completa detectando y solucionando fallos lo antes posible.

5.3. Glosario

Para convertir los elementos de IU de XML a Jetpack Compose, es útil saber cuál es el correspondiente de cada uno de ellos en ambos modos de definición, así como de las propiedades de cada uno que se han considerado más comunes. En este glosario se presentan tablas con las correspondencias de los componentes de IU nativos básicos de Android y sus propiedades más comunes, es decir, de aquellos que aparecen en la paleta de componentes del IDE oficial Android Studio, además de ejemplos referenciando al caso de estudio descrito en el Capítulo 4 para ilustrar mejor su uso. Estos componentes se dividen en distintas categorías: *layouts*, contenedores, texto, botones y *widgets*.

Cabe destacar que se han excluido las siguientes categorías presentes en la paleta de diseño: *helpers*, puesto que los elementos de esta categoría eran útiles para el `ConstraintLayout` disponible en XML y no se va a profundizar en su uso en Jetpack Compose en este trabajo, *Google*, dado que el único componente que tiene soporte en Jetpack Compose y además muy reciente es `MapView` [38], y *legacy*, puesto que son componentes que tampoco tienen soporte en Jetpack Compose dado que se han dejado de utilizar incluso en XML.

En primer lugar, puesto que es algo común a todos los componentes descritos posteriormente, se presentan las correspondencias de las propiedades más típicas de todos los componentes de IU. En XML, estas propiedades se expresan con atributos que comienzan con `android:` o `app:`, según si provienen del SDK de Android o de bibliotecas de soporte, respectivamente. En Jetpack Compose, la mayoría de estas propiedades se pueden expresar con los propios parámetros de los *composables* o mediante el parámetro de tipo `Modifier` que poseen todos ellos y sirve para editar ciertos aspectos y comportamientos del componente. En la Tabla 5.1 se muestran las propiedades más típicas de los componentes de IU.

XML	Jetpack Compose
`android:id="@+id/X"`	`Modifier.layoutId("X")`
`android:layout_width="match_parent"`	`Modifier.fillMaxWidth()`
`android:layout_width="wrap_content"`	Por defecto
`android:layout_width="Xdp"`	`Modifier.width(X.dp)`
`android:layout_height="match_parent"`	`Modifier.fillMaxHeight()`
`android:layout_height="wrap_content"`	Por defecto
`android:layout_height="Xdp"`	`Modifier.height(X.dp)`
`android:gravity`	Parámetros `-Alignment` y `-Arrangement` de `Row` y `Column`
`android:visibility`	Condiciones lógicas en el código
`android:padding="Xdp"`	`Modifier.padding(X.dp)`
`android:paddingHorizontal="Xdp"`	`Modifier.padding(horizontal = X.dp)`
`android:paddingVertical="Xdp"`	`Modifier.padding(vertical = X.dp)`
`android:paddingStart="Xdp"`	`Modifier.padding(start = X.dp)`
`android:paddingTop="Xdp"`	`Modifier.padding(top = X.dp)`
`android:paddingEnd="Xdp"`	`Modifier.padding(end = X.dp)`
`android:paddingBottom="Xdp"`	`Modifier.padding(bottom = X.dp)`
`android:layout_margin="Xdp"`	`Modifier.padding(X.dp)`
`android:layout_marginHorizontal="Xdp"`	`Modifier.padding(horizontal = X.dp)`
`android:layout_marginVertical="Xdp"`	`Modifier.padding(vertical = X.dp)`
`android:layout_marginStart="Xdp"`	`Modifier.padding(start = X.dp)`
`android:layout_marginTop="Xdp"`	`Modifier.padding(top = X.dp)`
`android:layout_marginEnd="Xdp"`	`Modifier.padding(end = X.dp)`
`android:layout_marginBottom="Xdp"`	`Modifier.padding(bottom = X.dp)`
`android:scrollbars="vertical"`	`Modifier.scrollable(state = rememberScrollState(), orientation = Orientation.Vertical)`
`android:scrollbars="horizontal"`	`Modifier.scrollable(state = rememberScrollState(), orientation = Orientation.Horizontal)`
`android:focusable="true/false"`	`Modifier.focusable(enabled = true/false)`
`android:clickable="true/false"`	`Modifier.clickable(enabled = true/false, onClick = {})`
`android:background="@drawable/forma"`	`Modifier.clip(Forma)`
`android:background="@drawable/borde"`	`Modifier.border(width = X.dp, color = Color.Color)`
`android:backgroundTint="@color/color"`	`Modifier.background(color = Color.Color)`

Tabla 5.1. Correspondencias de las propiedades típicas de los componentes de IU

A partir de la tabla anterior, se pueden destacar algunos detalles y hacer comentarios aclaratorios sobre algunas de las propiedades:

- Si se desea aplicar más de una propiedad a un componente, éstas se pueden encadenar en el parámetro `Modifier`. El orden de las mismas es significativo puesto que las propiedades que aparecen primero se aplicarán primero, es decir, con un modificador `Modifier.propiedad1().propiedad2()`, primero se aplicará la `propiedad1()` al componente y posteriormente la `propiedad2()`, que puede no dar el mismo resultado que si se aplican en el orden inverso. Por ejemplo, el resultado de aplicar a un texto el modificador `Modifier.padding(20.dp).border(2.dp, Color.Blue)` se puede observar en la parte izquierda de la Figura 5.24, y el resultado de aplicar `Modifier.border(2.dp, Color.Blue).padding(20.dp)` a la derecha, manifestando la dependencia del orden de las propiedades.

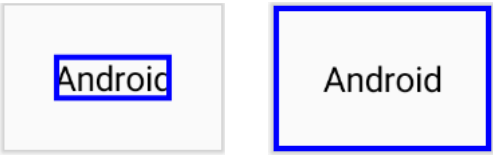


Figura 5.24. `Modifier` aplicando primero *padding* (izquierda) y aplicando primero el borde (derecha)

- La propiedad `android:gravity` ha dejado de existir, y ahora la alineación y organización de los elementos se gestiona mediante los parámetros `-Alignment` y `-Arrangement` de `Row` y `Column`, que se explicarán más en profundidad en la sección de *layouts*.

- La propiedad `android:visibility` ya no es necesaria debido al cambio de paradigma, que gestiona lo que sí es necesario mostrar o no en función del estado de la aplicación, utilizando condiciones desde el propio código. Un ejemplo de esto puede encontrarse en el *composable* `SectionHeader` del caso de estudio (Figura 4.9, Fragmento de código 4.3), donde el icono de añadir se muestra o no en función de una variable `showAddButton`:

```
1  @Composable
2  private fun SectionHeader(...) {
3      ...
4      if (showAddButton) {
5          IconButton(...)
6      }
7      ...
8  }
```

Fragmento de código 5.18. Manejo de la visibilidad de un componente de IU en paradigma declarativo

- Como se puede observar, la propiedad `android:layout_margin` y todas sus variantes han dejado de existir. Actualmente, todos los espacios entre elementos se gestionan mediante *paddings*, con los que es posible conseguir el mismo resultado que con los márgenes pero aplicándolos en el elemento contenedor. Por ejemplo, en el *composable* `SharedFileRow` del caso de estudio (Figura 4.6, Fragmento de código 4.2), anteriormente a los dos textos se les añadía un margen a la izquierda para separarlos del icono del fichero. Ahora, esta separación se maneja con un *padding* a la izquierda en el elemento contenedor de ambos textos, que es una columna:

```
1  @Composable
2  private fun SharedFileRow(file: OCFile) {
3      Row(...) {
4          ...
5          Column(modifier = Modifier.padding(start = 12.dp)) {
6              ...
7          }
8          ...
9      }
10 }
```

Fragmento de código 5.19. Manejo de los espacios con *paddings*

- Asociado a la propiedad `android:focusable="true"`, anteriormente existía un `OnFocusChangeListener` que permitía implementar la función que capturaba el evento de poner el foco en el elemento de IU en cuestión. En Jetpack Compose, gracias al lenguaje funcional Kotlin, esto es posible realizarlo mediante el modificador `Modifier.onFocusEvent({ focusState -> })`, al que se pasa como parámetro la función que se desea ejecutar cuando se capture el evento.
- De la misma manera, asociado a `android:clickable="true"`, anteriormente existía un `OnClickListener` que capturaba el evento de click sobre el elemento. En Jetpack Compose, esto se puede realizar mediante el parámetro `onClick` de `Modifier.clickable`, tal y como se especifica en la tabla.
- Para la propiedad `android:background="@drawable/forma"`, que se traduce en `Modifier.clip(Forma)` en Jetpack Compose, la clase `Forma` debe ser una de las proporcionadas por Android o bien implementar la interfaz `Shape`.

La lista completa de modificadores se puede encontrar en la documentación oficial de Android [18].

Layouts

Los *layouts* en Android son el tipo principal de contenedores, que permiten definir la disposición de los elementos que contienen. Los *layouts* disponibles de forma nativa y sus propiedades específicas más relevantes aparecen a continuación.

XML	Jetpack Compose
`LinearLayout` con `android:orientation="horizontal"`	`Row`
`android:gravity="top"`	`Row(verticalAlignment = Alignment.Top)`
`android:gravity="bottom"`	`Row(verticalAlignment = Alignment.Bottom)`
`android:gravity="center_vertical"`	`Row(verticalAlignment = Alignment.CenterVertically)`

Tabla 5.2. Correspondencias de la vista XML `LinearLayout` (horizontal)

XML	Jetpack Compose
`LinearLayout` con `android:orientation="vertical"`	`Column`
`android:gravity="end"`	`Column(horizontalAlignment = Alignment.End)`
`android:gravity="start"`	`Column(horizontalAlignment = Alignment.Start)`
`android:gravity="center_horizontal"`	`Column(horizontalAlignment = Alignment.CenterHorizontally)`

Tabla 5.3. Correspondencias de la vista XML `LinearLayout` (vertical)

Los *layouts* lineares permiten organizar elementos en una fila o una columna, uno detrás de otro. Como se puede observar en las Tablas 5.2 y 5.3, una de las propiedades más importantes de los *layout* lineares es `verticalAlignment` y `horizontalAlignment`, respectivamente. Estas propiedades `-Alignment` permiten definir cómo se posicionan los elementos en el eje perpendicular a la dirección del *layout*, es decir, en el caso de `Row` en el eje vertical y en el caso de `Column` en el eje horizontal. Para definir el posicionamiento de los elementos en el eje de la dirección del *layout*, existen las propiedades `-Arrangement`, que poseen unos valores por defecto `Arrangement.TipoDeArrangement`. En el caso de `Row`, los tipos de `horizontalArrangement` son los mostrados en la Figura 5.25, y en el caso de `Column`, los tipos de `verticalArrangement` los mostrados en la Figura 5.26.

Además de estos *layouts* lineares existe otro *layout* básico en Jetpack Compose, denominado `Box`. Este contenedor permite colocar elementos unos encima de otros ocupando el mismo espacio y posicionándose de forma relativa a su contenedor padre, de manera similar a como se podía hacer con `FrameLayout`, otro de los *layouts* disponibles en la paleta de componentes de Android, o `RelativeLayout`, ya deprecado en Android (se encuentra en la categoría *legacy*). Puesto que `Box` se basa exclusivamente en su modificador `Modifier.align(...)` para posicionar los elementos contenidos y no tiene correspondiente directo en XML, no se presenta en forma de tabla.

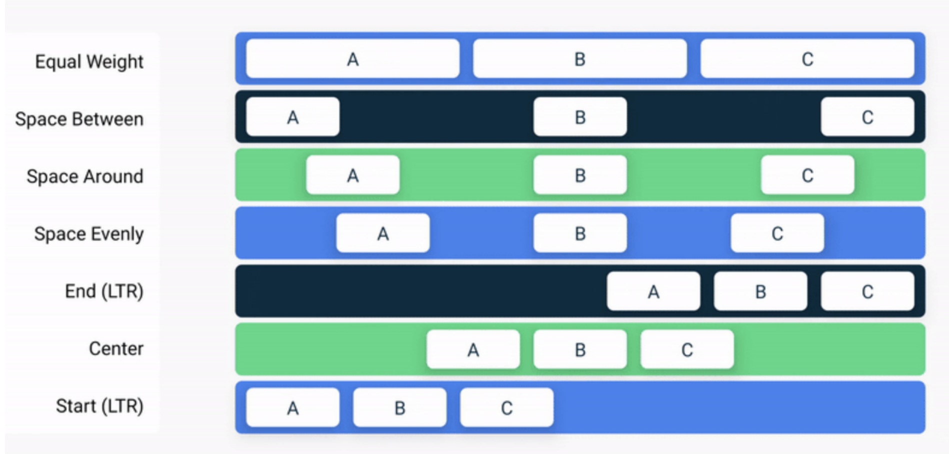


Figura 5.25. Tipos de `horizontalArrangement` para `Row` [6]

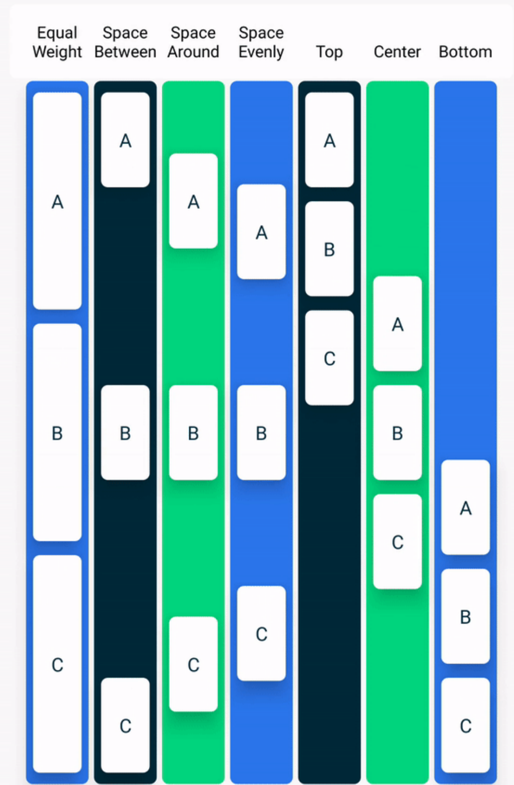


Figura 5.26. Tipos de `verticalArrangement` para `Column` [6]

Disponiendo de los *layouts* `Row`, `Column` y `Box`, es posible confeccionar la gran mayoría de pantallas que se podían hacer mediante *layouts* XML. Un ejemplo se puede observar en cualquiera de los *composables* del caso de estudio que incluyen más de un elemento,

como en `SharedFileRow` (Figura 4.6, Fragmento de código 4.2), que se puede ver cómo se organiza internamente mediante filas y columnas.

En XML, uno de los *layouts* más populares es `ConstraintLayout`, que se basa en especificar la posición de cada uno de los elementos en relación a los demás. Este tipo de contenedor es de los más usados en XML puesto que evita anidar muchos *layouts*, algo que perjudica a la aplicación en términos de rendimiento cuando se define mediante vistas XML. Sin embargo, en Compose, anidar muchos *layouts* no supone un problema, puesto que se pueden manejar jerarquías muy profundas de *layouts* de forma eficiente. Por ello, la recomendación es definir las pantallas previamente definidas con `ConstraintLayout` mediante los tres *layouts* básicos explicados. Aun así, en Jetpack Compose existe el análogo de este tipo de *layout*, denominado también `ConstraintLayout` y recomendado si ayuda a aumentar la legibilidad y mantenibilidad del código en pantallas altamente complejas. Este *layout* necesita otra dependencia puesto que no se encuentra entre los básicos, y toda la información relativa al mismo se puede consultar en la documentación oficial [8].

Entre los *layouts* básicos de Android de la paleta de componentes XML también se pueden encontrar `TableLayout` y `TableRow`, utilizados para crear un *layout* en forma de tabla y cada una de sus filas, respectivamente. De la misma manera que con el tipo anterior, no existe un correspondiente directo en Jetpack Compose puesto que una tabla se puede realizar fácilmente mediante una combinación de componentes `Row` y `Column`.

Finalmente, el último componente disponible en la paleta de diseño XML es `Space`, que como su propio nombre indica, sirve para añadir espacios entre elementos. El correspondiente en Jetpack Compose es `Spacer`, que posee un solo parámetro de tipo `Modifier` con el que se le puede dar el mismo aspecto que su correspondiente en XML utilizando las equivalencias comunes detalladas en la Tabla 5.1.

Contenedores

Los contenedores son una categoría de componentes de IU destinados a tener elementos dentro de sí. Las correspondencias de los pertenecientes a la paleta de componentes Android aparecen a continuación.

XML	Jetpack Compose
`Spinner`	`Row` que contiene `DropdownMenu` con diversos `DropdownMenuItem`
`android:entries="@array/valores"`	`Text` dentro de cada `DropdownMenuItem`

Tabla 5.4. Correspondencias de la vista XML `Spinner`

El componente `Spinner`, que se trata de un menú desplegable de opciones, no posee un correspondiente directo, tal y como se puede observar en la Tabla 5.4. Para conseguir uno de aspecto similar al de XML, se define una `Row` que al hacer click sobre ella despliegue un `DropdownMenu` (mediante su parámetro booleano `expanded`). Este `DropdownMenu` a su vez contendrá tantos `DropdownMenuItem` como entradas se desee que tenga el menú desplegable.

A pesar de no existir una vista directamente correspondiente, Jetpack Compose ofrece mucha flexibilidad al respecto, dado que mientras que antes con el `Spinner` nativo solo se podían tener entradas con texto, ahora se puede añadir cualquier *composable* en cada entrada, ofreciendo la posibilidad de mostrar texto de la misma manera que antes, iconos, imágenes, etc. Un ejemplo de menú desplegable en Jetpack Compose se puede consultar en este tutorial [39].

XML	Jetpack Compose
`RecyclerView`	`LazyColumn`
`app:reverseLayout="true/false"`	`LazyColumn(reverseLayout = true/false)`

Tabla 5.5. Correspondencias de la vista XML `RecyclerView`

XML	Jetpack Compose
`RecyclerView` con `app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"` y `android:orientation="horizontal"`	`LazyRow`
`app:reverseLayout="true/false"`	`LazyRow(reverseLayout = true/false)`

Tabla 5.6. Correspondencias de la vista XML `RecyclerView` (horizontal)

El componente XML `RecyclerView` es uno de los más importantes y su uso se ve muy simplificado con Jetpack Compose. Se trata de un contenedor para mostrar listas dinámicas de elementos haciendo un uso eficiente de la memoria, a través del reciclaje de las vistas que salen de pantalla para mostrar otros elementos que entran en pantalla.

Como se puede ver en las Tablas 5.5 y 5.6, según la orientación del `RecyclerView`, el correspondiente en Compose es `LazyColumn` o `LazyRow`. Estos dos *composables* gestionan la alineación de los elementos que contienen mediante los parámetros `-Alignment` y `-Arrangement`, de la misma manera que como se ha explicado en las correspondencias de `Column` y `Row`, respectivamente. La gran ventaja que suponen frente a XML es que ya no es necesario mantener un `RecyclerView`, un `Adapter`, un `ViewHolder` y los correspondientes archivos XML del `RecyclerView` y de los elementos a mostrar, sino que es posible hacerlo todo dentro de un mismo *composable*. Dentro del alcance de cualquiera de estos dos *composables* `Lazy-`, los elementos individuales deben rodearse con un bloque `item` y las listas de elementos con un bloque `items`, pasando por parámetro la propia lista de elementos a pintar. Por ejemplo, en el caso de estudio se puede observar la estructura general de la pantalla, formada por un `LazyColumn` y con elementos individuales y listas de elementos dentro de ella (Figura 4.19, Fragmento de código 4.9). De esta manera, se mostrarían primero tres elementos individuales, cada uno con su *composable* correspondiente, y posteriormente una lista de elementos, en la que para cada elemento de la lista `privateShares` se pintará un `ShareUserItem` y un `Divider`:

```
LazyColumn {
    item { SharedFileRow(...) }
    ...
    item { SectionHeader(...) }
    ...
    item { EmptyListText(...) }
    ...
    items(privateShares) { share ->
        ShareUserItem(...)
        Divider(...)
    }
}
```

Fragmento de código 5.20. Uso de `LazyColumn` mostrando elementos individuales y listas de elementos

Los siguientes componentes XML que aparecen en la paleta de diseño son `ScrollView`, `HorizontalScrollView` y `NestedScrollView`. Estos elementos eran necesarios para poder hacer *scroll* en pantallas demasiado grandes que no se pudieran visualizar enteras. En Jetpack Compose, no existe un correspondiente directo puesto que el concepto de *scroll* se asocia ahora a los propios *layouts* básicos, es decir, a `Row`, `Column` y `Box`, en lugar de tratarlo como otro componente separado. Simplemente, para proveer de *scroll* a cualquiera de estos *layouts*, se le debe pasar como modificador un `Modifier.verticalScroll(rememberScrollState())` si se desea tener *scroll* vertical o `Modifier.horizontalScroll(rememberScrollState())` si se desea tener *scroll* horizontal. Otra alternativa pasa por utilizar directamente un componente `LazyColumn` o `LazyRow`, que son *scrollables* por defecto, pero se recomienda reservar éstos para las listas dinámicas de elementos puesto que su utilización implica la introducción de código repetitivo con los bloques `item/items`.

El siguiente componente, `ViewPager2`, proporciona un carrusel de elementos. En Jetpack Compose no existe un equivalente a dicho componente dada la flexibilidad que permite la utilización libre de elementos más básicos, como en el caso del `Spinner`. Existen diversas formas de crear un `ViewPager2` personalizado en Jetpack Compose, pero dado que cada solución es distinta y todas suponen una agregación de otros elementos básicos, no se detallará. Un ejemplo de código equivalente a dicho componente se puede encontrar en este artículo [34].

Las correspondencias del componente `CardView`, que representa una tarjeta, se pueden observar en la Tabla 5.7. El componente `MaterialCardView`, a pesar de no aparecer en la paleta de componentes básicos, es una variante del mismo componente pero que aplica por defecto los principios de *Material Design*, por lo que el correspondiente en Jetpack Compose es aún más similar a éste, como se puede ver en la Tabla 5.8. A diferencia de `CardView`, entre sus propiedades más típicas `MaterialCardView` permite añadir un borde

con un determinado color y grosor a la tarjeta, algo que el *composable* `Card` ya permite por defecto también.

XML	Jetpack Compose
`CardView`	`Card`
`app:cardCornerRadius="Xdp"`	`Card(shape = RoundedCornerShape(X.dp))`
`app:cardBackgroundColor="@color/color"`	`Card(backgroundColor = Color.Color)`
`app:cardElevation="Xdp"`	`Card(elevation = X.dp)`

Tabla 5.7. Correspondencias de la vista XML `CardView`

XML	Jetpack Compose
`MaterialCardView`	`Card`
`app:cardCornerRadius="Xdp"`	`Card(shape = RoundedCornerShape(X.dp))`
`app:cardBackgroundColor="@color/color"`	`Card(backgroundColor = Color.Color)`
`app:cardElevation="Xdp"`	`Card(elevation = X.dp)`
`app:strokeWidth="Xdp"` + `app:strokeColor="@color/color"`	`Card(border = BorderStroke(width = X.dp, color = Color.Color))`

Tabla 5.8. Correspondencias de la vista XML `MaterialCardView`

El siguiente componente XML, `AppBarLayout`, proporciona un *layout* que permite añadir una barra de aplicación fácilmente siguiendo las líneas de *Material Design*. En Jetpack Compose no existe un correspondiente directo de este componente, pero sí existe `Scaffold`, un *composable* que permite añadir a una pantalla en huecos reservados elementos de IU que siguen los principios de *Material Design*, como por ejemplo la barra de aplicación superior, barra de aplicación inferior, botón flotante de acción, menú lateral, entre otras, por lo que es más flexible y potente que `AppBarLayout` puesto que no solo sirve para barras de aplicación sino para otros varios componentes. Para más información sobre `Scaffold` se puede consultar la documentación oficial [17].

XML	Jetpack Compose
`Toolbar` o `MaterialToolbar`	`TopAppBar`
`app:title="Texto"`	`TopAppBar(title = { /* Un Text */ })`
`app:navigationIcon="@drawable/icono"`	`TopAppBar(navigationIcon = { /* Un IconButton */ })`
`app:menu="@menu/menu"`	`TopAppBar(actions = { /* Tantos IconButton como se desee */ })`
`app:title="Texto"`	`TopAppBar(title = { /* Un Text */ })`
`android:elevation="Xdp"`	`TopAppBar(elevation = X.dp)`

Tabla 5.9. Correspondencias de las vistas XML `Toolbar` y `MaterialToolbar`

XML	Jetpack Compose
`BottomAppBar`	`BottomAppBar`
`app:backgroundTint="@color/color"`	`BottomAppBar(backgroundColor = Color.Color)`
`app:menu="@menu/menu"`	`BottomAppBar() { /* Tantos IconButton como se desee */ }`
`app:elevation="Xdp"`	`BottomAppBar(elevation = X.dp)`

Tabla 5.10. Correspondencias de la vista XML `BottomAppBar`

XML	Jetpack Compose
`BottomNavigationView`	`BottomNavigation`
`app:backgroundTint="@color/color"`	`BottomNavigation(backgroundColor = Color.Color)`
`app:menu="@menu/menu"`	`BottomNavigation() { /* Tantos BottomNavigationItem como se desee */ }`
`app:elevation="Xdp"`	`BottomNavigation(elevation = X.dp)`

Tabla 5.11. Correspondencias de la vista XML `BottomNavigationView`

En las Tablas 5.9, 5.10 y 5.11 se muestran las correspondencias de tres de los elementos que se pueden pasar como parámetros a un *composable* `Scaffold`, de manera que se puedan colocar como barras de aplicación superiores o inferiores. Anteriormente, el atributo `app:menu` indicaba qué elementos se van a mostrar en la barra de aplicación, pero actualmente debido a la flexbilidad de Compose, es posible definir dichos elementos como contenido de los *composables*, y sin limitarse a iconos, sino que es posible añadir cualquier tipo de componente IU.

Otro de los componentes XML de este tipo es `NavigationView`. Con este componente se contruye aspecto del menú lateral de la aplicación. En Jetpack Compose no existe correspondiente directo, pero tampoco es necesario puesto que nuevamente el *composable* `Scaffold` permite pasarle en otro de sus parámetros una agrupación de *composables* que serán los que se muestren en el menú lateral. Un ejemplo de código para esto mismo se puede consultar en este artículo [42].

Para el correspondiente de `TabLayout` y cada uno de los `TabItem` que contiene, que forman una agrupación de pestañas para navegar entre distintas pantallas, también es necesario recurrir a la creatividad y partir desde cero realizando una creación propia. Lo más parecido a estos elementos en Jetpack Compose son `TabRow` para el *layout* contenedor, y `Tab` o `LeadingIconTab` para cada una de las pestañas, en función de si poseen un icono o no. El estado de estas pestañas y los gestos de navegación entre las mismas no se proporcionan de manera automática, sino que es necesario recurrir a otros elementos. Este artículo muestra una de las maneras de conseguir un código equivalente a `TabLayout` con distintos elementos `TabItem` y navegación entre ellos [30].

Los elementos restantes en esta categoría de la paleta de componentes, o bien no poseen equivalente en Jetpack Compose debido a la no existencia de fragmentos en este paradigma, como en el caso de `FragmentContainerView` o `NavHostFragment`, o bien son útiles para el sistema de vistas XML exclusivamente, como son `ViewStub`, `<include>`, `<view>` y `<requestFocus>`.

Texto

Los componentes de IU de la categoría de texto son aquellos que se utilizan para mostrar texto por pantalla o bien para recibir una entrada de texto por parte del usuario. A continuación se describen todos ellos y sus propiedades más típicas.

XML	Jetpack Compose
`TextView`	`Text`
`android:text="Texto"`	`Text(text = "Texto")`
`android:textColor="@color/color"`	`Text(color = Color.Color)`
`android:textSize="Xsp"`	`Text(fontSize = X.sp)`
`android:textStyle="normal"`	`Text(fontStyle = FontStyle.Normal)`
`android:textStyle="italic"`	`Text(fontStyle = FontStyle.Italic)`
`android:textStyle="bold"`	`Text(fontWeight = FontWeight.Bold)`
`android:fontFamily="monospace"`	`Text(fontFamily = FontFamily.Monospace)`
`android:fontFamily="serif"`	`Text(fontFamily = FontFamily.Serif)`
`android:fontFamily="sans-serif"`	`Text(fontFamily = FontFamily.SansSerif)`
`android:fontFamily="cursive"`	`Text(fontFamily = FontFamily.Cursive)`
`android:letterSpacing="X"`	`Text(letterSpacing = X.sp)`
`android:textAlignment="textStart"`	`Text(textAlign = TextAlign.Start)`
`android:textAlignment="textEnd"`	`Text(textAlign = TextAlign.End)`
`android:textAlignment="center"`	`Text(textAlign = TextAlign.Center)`
`android:ellipsize="end"`	`Text(overflow = TextOverflow.Ellipsis)`
`android:singleLine="true"`	`Text(maxLines = 1)`
`android:maxLines="X"`	`Text(maxLines = X)`

Tabla 5.12. Correspondencias de la vista XML `TextView`

En la Tabla 5.12 se muestran las correspondencias del componente XML `TextView`, utilizado para mostrar fragmentos de texto en pantalla. Este *composable* `Text` es muy común, como se puede observar en el caso de estudio para cualquier elemento que contenga texto, como el propio `WarningText` (Figura 4.15, Fragmento de código 4.6).

El otro componente XML básico relativo a textos es `EditText`, que permite la entrada de texto del usuario. Su equivalente en Compose es `TextField`, como se observa en la Tabla 5.13. Aunque en la tabla se omite por brevedad, este componente tiene como parámetros

XML	Jetpack Compose
`EditText`	`TextField`
`android:enabled="true/false"`	`TextField(enabled = true/false)`
`android:hint="Texto"`	`TextField(placeholder = { /* Un Text */ })`
`android:singleLine="true/false"`	`TextField(singleLine = true/false)`
`android:maxLines="X"`	`TextField(maxLines = X)`
`android:inputType="textPersonName"`	`TextField(keyboardOptions = KeyboardOptions(capitalization = KeyboardCapitalization.Sentences, keyboardType = KeyboardType.Text))`
`android:inputType="textPassword"`	`TextField(keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password), visualTransformation = PasswordVisualTransformation())`
`android:inputType="numberPassword"`	`TextField(keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.NumberPassword), visualTransformation = PasswordVisualTransformation())`
`android:inputType="textEmailAddress"`	`TextField(keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email))`
`android:inputType="phone"`	`TextField(keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Phone)`
`android:inputType="textPostalAddress"`	`TextField(keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Text))`
`android:inputType="textMultiLine"`	`TextField(keyboardType = KeyboardType.Text, imeAction = ImeAction.None))`
`android:inputType="number"`	`TextField(keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number))`

Tabla 5.13. Correspondencias de la vista XML `EditText`

obligatorios `value` y `onValueChange`, dedicados al texto que se debe mostrar en el propio campo de texto y al evento que captura el cambio de ese texto llevado a cabo por el usuario, respectivamente. Para su correcto funcionamiento, el parámetro `value` se debe gestionar con un estado, como se explicará en la Sección 5.4.

En cuanto a los otros componentes XML de texto disponibles en la paleta, `AutoComplete-TextView` y `MultiAutoCompleteTextView` aún no tienen soporte en Jetpack Compose,

por lo que si se desea tener uno se debe hacer partiendo desde cero. En este artículo se muestra un ejemplo de cómo poder hacerlo [48]. El componente XML `CheckedTextView` tampoco tiene correspondiente directo en Jetpack Compose, dado que es posible añadir al *composable* `TextField` mediante sus parámetros `leadingIcon` o `trailingIcon` un *checkbox* al principio o al final del campo de texto, que puede cumplir la función del botón seleccionable de `CheckedTextView`. Finalmente, el último componente disponible, `TextInputLayout`, tampoco tiene correspondiente directo en Compose puesto que lo que le hace especial es que posee una etiqueta flotante con el texto de su *hint* cuando ésta no es visible en el campo de texto porque se ha introducido texto, siguiendo así las líneas de *Material Design*. Esta característica está disponible en el *composable* ya descrito `TextField` mediante su parámetro `label`, que hace referencia a esta etiqueta flotante y unifica lo que eran dos componentes distintos en XML en uno solo.

Botones

Existen varios componentes que permiten representar botones de distinto tipo. En este apartado se presentan las correspondencias de aquellos presentes en la paleta XML.

XML	Jetpack Compose
`Button`	`Button`
`android:text="Texto"`	`Button() { /* Un Text */ }`
`android:enabled="true/false"`	`Button(enabled = true/false)`
`android:background="@drawable/forma"`	`Button(shape = Forma`

Tabla 5.14. Correspondencias de la vista XML `Button`

XML	Jetpack Compose
`MaterialButton`	`Button`
`android:text="Texto"`	`Button() { /* Un Text */ }`
`android:enabled="true/false"`	`Button(enabled = true/false)`
`android:background="@drawable/forma"`	`Button(shape = Forma`
`app:strokeWidth="Xdp"` + `app:strokeColor="@color/color"`	`Button(border = BorderStroke(width =` `X.dp, color = Color.Color))`
`app:cornerRadius="Xdp"`	`Button(shape =` `RoundedCornerShape(10.dp))`
`app:icon="@drawable/icono"`	`Button() { /* Un Icon */ }`

Tabla 5.15. Correspondencias de la vista XML `MaterialButton`

Aunque en las Tablas 5.14 y 5.15 no se muestra por brevedad, el *composable* `Button` posee un parámetro obligatorio: `onClick`. Éste recibirá la función que se utilizará para manejar el evento de click sobre el botón. Además, de la misma manera que ocurre con los *composables* `CardView` y `MaterialCardView` (Tablas 5.7 y 5.8), aunque no está entre los componentes básicos, el botón tiene su correspondiente adaptado a *Material Design*, `MaterialButton` (Tabla 5.15), que posee propiedades más similares al botón de Jetpack

Compose y por ello posee alguna correspondencia más.

XML	Jetpack Compose
`ImageButton`	`IconButton`
`app:srcCompat="@drawable/icono"`	`IconButton() { /* Un Icon */ }`
`app:srcCompat="@drawable/imagen"`	`IconButton() { /* Un Image */ }`
`android:enabled="true/false"`	`IconButton(enabled = true/false)`

Tabla 5.16. Correspondencias de la vista XML `ImageButton`

De la misma manera que con el botón normal, el *composable* `IconButton` requiere como parámetro obligatorio `onClick`, recibiendo también la función para manejar el evento de click sobre el botón. El uso de este componente se puede observar en el caso de estudio en el componente creado `SharePublicLinkItem`, para representar los iconos pulsables de las acciones disponibles (Figura 4.16, Fragmento de código 4.7).

El componente XML `Chip` y por tanto su componente agrupador `ChipGroup`, no están disponibles en Jetpack Compose, y es necesario crearlos a partir de otros *composables* más básicos. Una forma de hacerlo se muestra en este artículo [54].

XML	Jetpack Compose
`CheckBox`	`Checkbox`
`android:text="Texto"`	`CheckBox` y `Text` dentro de un componente `Row`
`android:enabled="true/false"`	`Checkbox(enabled = true/false)`

Tabla 5.17. Correspondencias de la vista XML `CheckBox`

Al igual que con `TextField`, para manejar el estado de marcado o no de `Checkbox`, se hará uso del concepto de estado mediante los parámetros obligatorios `checked` y `onCheckedChange`. Este concepto se introducirá en la Sección 5.4 de este capítulo.

XML	Jetpack Compose
`RadioButton`	`RadioButton`
`android:text="Texto"`	`RadioButton` y `Text` dentro de un componente `Row`
`android:enabled="true/false"`	`RadioButton(enabled = true/false)`

Tabla 5.18. Correspondencias de la vista XML `RadioButton`

Nuevamente, para el estado de marcado de `RadioButton` se utilizará el concepto de estado, como se explicará en la siguiente sección. El parámetro obligatorio `selected` permite gestionar el estado de marcado del botón, y `onClick`, también obligatorio, permite definir qué ocurre cuando se hace click sobre el `RadioButton`. En XML también existe un componente `RadioGroup` que sirve para gestionar un conjunto de `RadioButton`, puesto

que la idea tras estos componentes es que se ofrezca al usuario una agrupación de distintas opciones y sólo se pueda seleccionar una de ellas. Como en Compose se puede gestionar el estado directamente, realizar una gestión conjunta del estado de varios botones de este tipo, asegurando que únicamente esté marcado uno de ellos, sustituye la función del `RadioGroup`.

El componente XML `ToggleButton` consiste en un botón de aspecto normal, pero con dos estados en lugar de uno como el `Button` original. En Jetpack Compose, no existe correspondiente directo de este componente puesto que es posible dotar de esta característica a un botón normal mediante el modificador `Modifier.toggleable()`, al que se ha de pasar un valor booleano en su parámetro `value`, que también se manejará con un estado de Compose (Sección 5.4), y una función al parámetro `onValueChange`, para manejar el evento de cambio de estado del botón. Cabe destacar que en Compose sí que existe un *composable* `IconToggleButton` que permite crear un botón de icono de dos estados, gestionado mediante sus parámetros `checked` y `onCheckedChange` de manera análoga al modificador `Modifier.toggleable()`.

XML	Jetpack Compose
`Switch`	`Switch`
`android:text="Texto"`	`Switch` y `Text` dentro de un componente `Row`
`android:enabled="true/false"`	`Switch(enabled = true/false)`

Tabla 5.19. Correspondencias de la vista XML `Switch`

`Switch` es un botón de dos estados pero con distinto aspecto a un botón original. En este caso también se hace uso del estado en Compose para gestionar si el botón se encuentra activado o desactivado, como se explicará en la Sección 5.4. Los parámetros obligatorios del *composable* que ofrecen esta funcionalidad son el booleano `checked` y la función `onCheckedChange`.

XML	Jetpack Compose
`FloatingActionButton`	`FloatingActionButton`
`app:srcCompat="@drawable/icono"`	`FloatingActionButton() { /* Un Icon */ }`
`app:shapeAppearanceOverlay="@style/forma"`	`FloatingActionButton(shape = Forma)`
`app:backgroundTint="@color/color"`	`FloatingActionButton(backgroundColor = Color.Color)`
`app:elevation="Xdp"`	`FloatingActionButton(elevation = FloatingActionButtonDefaults .elevation(X.dp))`

Tabla 5.20. Correspondencias de la vista XML `FloatingActionButton`

El componente `FloatingActionButton` posee un parámetro obligatorio `onClick` que recibe la función para manejar el click sobre el botón. Junto a otros de los componentes

explicados en la categoría de contenedores, el *composable* `FloatingActionButton` puede pasarse como parámetro al componente `Scaffold`, que reserva un hueco para dicho botón en la parte inferior derecha de la pantalla.

Widgets

En la última categoría, *widgets*, se encuentran otros componentes de IU que no encajan en las otras categorías, pero no por ello son menos importantes. A continuación se presenta la correspondencia entre ambos modos de definición de IU de los *widgets* disponibles en la paleta de componentes XML.

El primer componente que se encuentra es `View`. `View` es un componente genérico y en realidad es a partir del que se han creado todos los demás, por lo que posee las propiedades comunes a todos los componentes (en la Tabla 5.1 se indican los más típicos). Su uso por sí mismo es típicamente como un bloque que ocupa una zona de la IU, al que se le pueden dar distintas formas (círculo, óvalo, línea...) para conseguir un elemento de IU básico que no se podría conseguir con los demás componentes. En Jetpack Compose no existe una correspondencia directa para este componente, pero dado el uso que se le da como vista genérica, se podría considerar la utilización del *composable* `Surface`, que es precisamente la metáfora de vista principal en Jetpack Compose.

XML	Jetpack Compose
`ImageView`	`Image`
`app:srcCompat="@drawable/imagen"`	`Image(painter = painterResource(id = R.drawable.imagen)`
`android:contentDescription="Texto"`	`Image(contentDescription = "Texto")`
`android:scaleType="center"`	`Image(contentScale = ContentScale.None)`
`android:scaleType="centerInside"`	`Image(contentScale = ContentScale.Inside)`
`android:scaleType="centerCrop"`	`Image(contentScale = ContentScale.Crop)`
`android:scaleType="fitStart"`	`Image(alignment = Alignment.CenterStart)`
`android:scaleType="fitEnd"`	`Image(alignment = Alignment.CenterEnd)`
`android:scaleType="fitCenter"`	Por defecto
`android:scaleType="fitXY"`	`Image(contentScale = ContentScale.FillBounds)`
`android:alpha="X"`	`Image(alpha = X)`

Tabla 5.21. Correspondencias de la vista XML `ImageView`

El *composable* `Image`, correspondiente de `ImageView` tal y como se muestra en la Tabla 5.21, posee como parámetros obligatorios `painter`, donde se le pasa el recurso que se desea dibujar en pantalla, y `contentDescription`, de manera que por defecto haya que

proporcionar una descripción del contenido de la imagen (se puede dejar a `null`). Aquí se manifiesta la forma en la que Jetpack Compose fomenta la accesibilidad de las IU construídas, como ocurre con otros componentes de esta misma categoría. En XML, este componente se utilizaba también para representar iconos, pero en Jetpack Compose se ha añadido un nuevo *composable* `Icon` más adecuado para manipular las propiedades más básicas de un icono. La correspondencia con este componente se muestra en la Tabla 5.22.

XML	Jetpack Compose
`ImageView`	`Icon`
`app:srcCompat="@drawable/imagen"`	`Icon(painter = painterResource(id = R.drawable.imagen)`
`android:contentDescription="Texto"`	`Icon(contentDescription = "Texto")`
`app:tint="@color/color"`	`Icon(tint = Color.Color)`

Tabla 5.22. Correspondencias de la vista XML `ImageView` (icono)

De la misma manera que con `Image`, los parámetros `painter` y `contentDescription` son obligatorios. En el caso de estudio se puede encontrar el uso de estos dos *composables*, bien en el caso de `SharedFileRow` para `Image` (Figura 4.7, Fragmento de código 4.2) o bien en el caso de `SharePublicLinkItem` para `Icon` (Figura 4.16, Fragmento de código 4.7), con el objetivo de colocar en la IU una imagen o un icono, respectivamente.

Para los componentes `WebView`, utilizado para mostrar una pantalla de navegador web interno de la aplicación, y `VideoView`, utilizado para reproducir un archivo de vídeo en la IU, aún no existe soporte en Jetpack Compose. Además, éstos son unos de los pocos casos en los que no se puede construir algo similar con otros *composables* más básicos, por lo que es necesario hacer uso del *composable* `AndroidView`, que permite interoperabilidad con XML en una misma pantalla tal y como se ha explicado en la Sección 5.2, o bien utilizar bibliotecas externas que lo tengan implementado. En el caso de `WebView`, existe una biblioteca de útiles de Google Accompanist [36].

En el caso del componente `CalendarView`, que proporciona una vista de calendario, tampoco existe soporte aún en Jetpack Compose. Tampoco existen bibliotecas oficiales de Google que lo implementen aunque sean externas, por lo que en este caso la única solución es hacer uso de `AndroidView` para incluir la vista XML en el código en Compose (Sección 5.2) o utilizar bibliotecas de terceros.

Para los *composables* de barra de progreso, mostrados en las Tablas 5.23 y 5.24, si no se da valor al parámetro `progress`, aparecerá una animación de progreso indeterminado, informando al usuario de que tiene que esperar sin indicar el progreso concreto. Para los casos en los que se desee mostrar el progreso real, rellenando la barra según se vaya progresando, se deberá utilizar un estado con el parámetro `progress`, como se explicará en la Sección 5.4.

El componente XML `SeekBar` y su variante discreta se convierten ahora en `Slider`, como se puede observar en la Tabla 5.25. Para gestionar correctamente el valor indicado por el usuario en la barra en todo momento, será necesario utilizar un estado (Sección

XML	Jetpack Compose
`ProgressBar` con `style="?android:attr/progressBarStyle"`	`CircularProgressIndicator`
`android:progress="X"`	`CircularProgressIndicator(progress = X)`
`android:indeterminateTint="@color/color"`	`CircularProgressIndicator(color = Color.Color)`
`android:indeterminateDrawable= "@drawable/bordeGrosorPersonalizado"`	`CircularProgressIndicator(strokeWidth = X.dp)`

Tabla 5.23. Correspondencias de la vista XML `ProgressBar` (circular)

XML	Jetpack Compose
`ProgressBar` con `style="?android:attr/progressBarStyleHorizontal"`	`LinearProgressIndicator`
`android:progress="X"`	`LinearProgressIndicator(progress = X)`
`android:progressTint="@color/color"`	`LinearProgressIndicator(color = Color.Color)`
`android:progressBackgroundTint= "@color/color"`	`LinearProgressIndicator( backgroundColor = Color.Blue)`

Tabla 5.24. Correspondencias de la vista XML `ProgressBar` (horizontal)

XML	Jetpack Compose
`SeekBar`	`Slider`
`android:progress="X"`	`Slider(value = X)`
`android:min="X"` + `android:max="Y"`	`Slider(valueRange = valueRange = X.rangeTo(Y))`
`style="@style/Widget.AppCompat .SeekBar.Discrete"` + `android:max="X"`	`Slider(steps = X-1)`

Tabla 5.25. Correspondencias de la vista XML `SeekBar`

5.4) a través del parámetro obligatorio `value`, cuyos cambios se manejarán mediante la función pasada al parámetro también obligatorio `onValueChange`.

Al igual que ocurría con los componentes `WebView` y `VideoView`, para los componentes `RatingBar` y `SearchView` no existe soporte en Compose actualmente. Sin embargo, en estos casos sí es posible crear componentes personalizados a partir de *composables* más básicos, por ejemplo, para `RatingBar` como se muestra en este artículo [32] o para `SearchView` como se muestra en esta contribución de un usuario de Stack Overflow [56].

Para el componente XML `TextureView`, utilizado para mostrar por pantalla un flujo de datos de contenido principalmente multimedia, como por ejemplo la previsualización de la cámara, tampoco existe soporte actualmente. La solución, por tanto, pasa por utilizar el *composable* `AndroidView` que permite la interoperabilidad entre la vista XML ya existente

y Jetpack Compose, como se ha explicado en la Sección 5.2, dado que no es posible construir un componente similar a este con *composables* básicos.

XML	Jetpack Compose
`SurfaceView`	`Surface`
`android:background="@drawable/forma"`	`Surface(shape = Forma)`
`android:backgroundTint="@color/color"`	`Surface(color = Color.Color)`
`android:elevation="Xdp"`	`Surface(elevation = X.dp)`

Tabla 5.26. Correspondencias de la vista XML `SurfaceView`

A diferencia del componente `SurfaceView`, el *composable* `Surface` (Tabla 5.26) además de adoptar el aspecto que le queramos dar, puede servir como contenedor para otros elementos. Es muy similar al *composable* `Box` explicado en la categoría de *layouts*, pero `Surface` permite aplicar propiedades inherentes a *Material Compose* de manera mucho más sencilla, como por ejemplo la elevación.

Finalmente, la paleta XML presenta dos componentes: `Horizontal Divider` y `Vertical Divider`. En el código XML, estos componentes son realmente un `View` cada uno, a los que se les da cierta altura, anchura y color mediante las propiedades básicas, en función del aspecto que queramos darle. En Jetpack Compose existe un *composable* específico para este uso: `Divider`, pero no se presenta en forma de tabla puesto que es el correspondiente solamente para este uso concreto. Mediante los modificadores básicos podemos indicarle altura y anchura, y como parámetros propios es posible indicar color (`color`), grosor del divisor (`thickness`) e identación a la izquierda (`startIndent`). En el caso de estudio, este *composable* es utilizado para colocar un divisor después de cada ítem de las listas de comparticiones privadas y enlaces públicos, dentro de los bloques `items` (Figura 4.19, Fragmento de código 4.9).

5.4. Estado en *composables*

Como se ha explicado en la Sección 2.4 del capítulo de marco teórico, Jetpack Compose proporciona una forma de agregar estado a los *composables*, y es importante saber cómo gestionarlo.

El **estado** en un *composable* es necesario cuando existen ciertos valores que deben persistir entre recomposición y recomposición de la IU, y que pueden cambiar a lo largo del tiempo como respuesta a eventos. Por ejemplo, en un componente campo de texto, `TextField`, se debe mantener el texto introducido hasta el momento. Si este valor no se manejara como estado, al componente se le indicaría en su parámetro `value` qué texto debe mostrar, y cuando el usuario introduzca texto, se produciría una recomposición que daría lugar a un campo de texto con el mismo valor que antes, puesto que el cambio en el valor pasado a `value` no se mantiene entre recomposiciones. Por esa razón, este valor mutable se debe mantener como un estado. Lo mismo ocurre con componentes como botones con más de un estado, como `CheckBox` y `Switch`, barras de progreso, y en general,

cualquier componente de IU cuyo aspecto sea cambiante y dependa de ciertos valores de la aplicación.

Además, el concepto de estado en Compose implica que el valor que se quiere persistir entre recomposiciones es **observado**, y cuando éste cambia, se notifica automáticamente a los observadores, que suelen ser los propios *composables*. De esta manera, cuando el estado cambia, la IU de la aplicación es notificada, dando lugar a una recomposición que resulta en la generación de una IU con los valores observados actualizados en todo momento.

Existen dos maneras principales de gestionar el estado en Jetpack Compose, según la gestión del estado en la aplicación [21]:

***Composables* como fuente de verdad.** Si la gestión del estado es simple y se mantiene en el propio *composable*, se utiliza la expresión `remember` y sus variantes. Además, en el caso típico, cuando el estado es mutable, se utiliza `mutableStateOf` y sus variantes para indicar esta propiedad del valor. Para mantener un valor mutable como estado en un *composable*, se debe definir como `var stateName by remember { mutableStateOf(valorInicial) }`, y a partir de ahí, es posible utilizar el estado encapsulado en `stateName` como si de una variable normal se tratara. En el Fragmento de código 5.21 se muestra un ejemplo para un *composable* formado por un `Text` y un `TextField` dentro de un `Column`.

```
@Composable
fun Saludo() {
   Column {
       var nombre by remember { mutableStateOf("") }
       if (nombre.isNotEmpty()) {
           Text(text = "Saludos, $name!")
       }
       TextField(
           value = nombre,
           onValueChange = { nombre = it }
       )
   }
}
```

Fragmento de código 5.21. Ejemplo de estado mantenido en un *composable*

En el código anterior se puede observar cómo el estado se mantiene en `nombre`, que se inicializa como una cadena vacía. Como es posible utilizar el estado como una variable normal, se comprueba si la cadena de texto no está vacía para mostrar un saludo con el nombre introducido por el usuario. Al parámetro `value` de `TextField` se le pasa el propio estado, de tal manera que en dicho campo de texto siempre se mostrará el valor actualizado del estado, y la función pasada al parámetro `onValueChange` indica que cada vez que se cambie el valor del campo de texto, el estado se actualice con este nuevo valor. El resultado de mantener esta cadena de texto como estado es

que tanto el componente `Text` como el componente `TextField` mostrarán siempre el texto introducido por el usuario, sobreviviendo a las recomposiciones que ocurran.

ViewModels como fuente de verdad. Si la gestión del estado es más compleja y no es conveniente mantenerlo en el propio *composable*, sino en el ViewModel correspondiente, se utiliza la expresión `observeAsState`. Para ello, se debe instanciar el ViewModel, y al LiveData que se desea observar se le añade dicha expresión para convertirlo en estado, de la siguiente manera: `var stateName = viewModel.liveData.observeAsState()`. En el caso de estudio esto se ha llevado a cabo para mantener como estado la lista de comparticiones y las *capabilities* provenientes del servidor (Fragmento de código 4.8):

```
1  val capabilitiesState = ocCapabilityViewModel.capabilities.
      observeAsState()
2  ...
3  val capabilitiesUiResult = capabilitiesState.value?.
      peekContent()
4  val capabilities = capabilitiesUiResult?.getStoredData()
5  val isShareApiEnabled = capabilities?.filesSharingApiEnabled
      == CapabilityBooleanType.TRUE
6  ...
7  if (... && isShareApiEnabled) {
8  item { SectionHeader(...) }
9  items(privateShares) {
10     ShareUserItem(...)
11     Divider(...)
12 }
```

Fragmento de código 5.22. Ejemplo de estado mantenido en un ViewModel

En este caso, de la instancia del ViewModel (`ocCapabilityViewModel`) se recupera el LiveData que se desea observar (`capabilities`) y se convierte en estado mediante la expresión `observeAsState()`. A partir de este estado se recuperan datos hasta llegar al dato de interés, en este caso `isShareApiEnabled`, que permite determinar si se va a mostrar la sección de comparticiones privadas. En caso de que su valor cambie, la IU se actualizará inmediatamente mostrando o no la sección según indique el valor. Como se puede observar, es posible realizar una migración a Compose cuando los datos observados están en el ViewModel sin tener que modificar más que la clase que aloja el código de la interfaz de usuario.

A pesar de la existencia de métodos para gestionar el estado en Jetpack Compose, la recomendación es que esto sólo se haga cuando sea realmente imprescindible, ya que los *composables* con estado interno tienden a ser menos reutilizables y más difíciles de probar. En numerosas ocasiones, la utilización de estados se puede evitar mediante paso de valores a los *composables* por parámetro, ya que el cambio de los mismos también producirá una recomposición de la IU sin necesidad de mantener un estado que se observa continuamente.

6: Resultados y discusión

Una vez realizado el trabajo principal, para ponerlo en valor se realiza una recopilación de los resultados obtenidos y cómo éstos han ayudado a conseguir los objetivos planteados. En este capítulo se describen estos dos puntos y se realiza también el contraste de las hipótesis sobre Jetpack Compose planteadas en la Sección 2.4.

6.1. Resultados obtenidos y consecución de objetivos

Los resultados más significativos que se han obtenido a lo largo del desarrollo del trabajo y un análisis de cada uno en vista a los objetivos planteados en la Sección 1.4 se presentan a continuación:

- **Conocimiento sobre Jetpack Compose.** Tras las primeras fases de formación, y con lo aprendido a lo largo de todo el desarrollo de la guía de migración, se considera que se poseen conocimientos sólidos sobre la herramienta principal utilizada en el trabajo: Jetpack Compose. Este resultado es transversal, aplicable al resto del trabajo y en la generación del resto de resultados.

  Como parte de las fases de formación, se elaboró una réplica de una de las pantallas de la aplicación Android de WhatsApp [57] con Jetpack Compose. Dicha pantalla y la comparativa con la pantalla de la aplicación original se puede observar en la Figura 6.27. Dado el interés por el resultado obtenido, el código desarrollado y el conocimiento adquirido también dieron pie a la elaboración de un seminario para el personal de la oficina de Izertis en Valladolid sobre construcción de apps Android con Jetpack Compose, y como plan de futuro cercano, la redacción de un artículo para el blog de Izertis sobre Jetpack Compose.

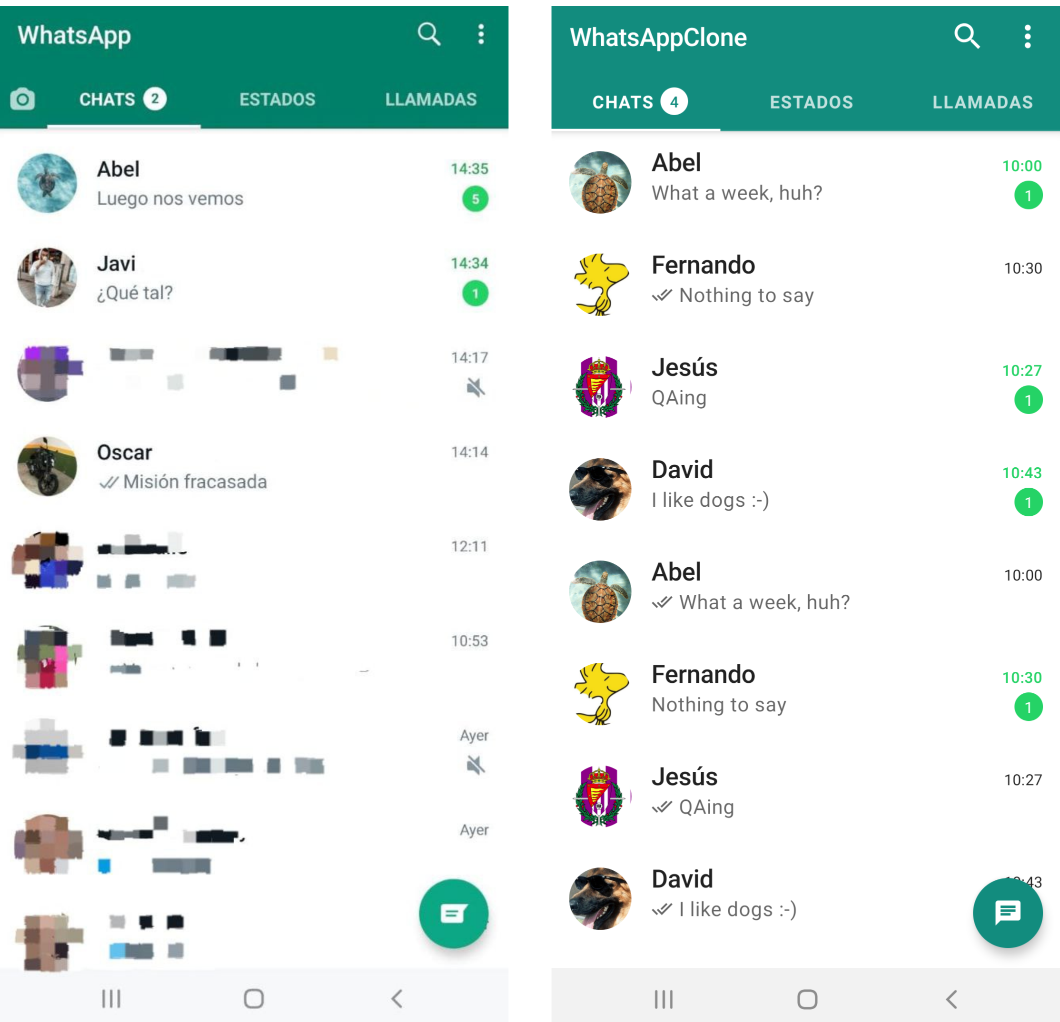


Figura 6.27. Pantalla original de WhatsApp (izquierda) y réplica con Jetpack Compose (derecha)

El conocimiento sobre Jetpack Compose es la base que ha permitido avanzar en el resto del proyecto. Junto con la réplica de la pantalla de WhatsApp en Jetpack Compose, que permitió asentar los conocimientos ya adquiridos en las primeras fases de formación y ponerlos en práctica en un caso de uso "real" al crear una app desde cero, se cumple el primer objetivo específico: **"Obtener formación y conocimiento sobre Jetpack Compose"**. Además, el seminario técnico impartido al personal de la oficina de Valladolid de Izertis y la futura publicación en el blog de Izertis permite otorgar un extra de utilidad a este resultado, puesto que se ha podido utilizar para realizar transferencia de conocimiento útil, vista la creciente popularidad de Jetpack Compose. La transferencia de conocimiento es una de las partes fundamentales del proceso de investigación, por lo que se adecúa a lo que se pretende conseguir en parte con este Trabajo de Fin de Máster, que es hacer un primer acercamiento al ámbito de la investigación.

- **Migración de una pantalla de la app Android de ownCloud hacia Jetpack Compose.** Se ha conseguido migrar exitosamente de XML a Jetpack Compose la parte de IU de una pantalla de una aplicación existente en producción. Este es precisamente el problema que se quería resolver con este trabajo, y la realización de este ejemplo concreto ha dado lugar al caso de estudio en el que se ha basado la realización de la guía de migración. En la Sección 4.3 del capítulo dedicado al caso de estudio se describe en más profundidad dicho resultado obtenido.

  Con dicha migración, se ha trabajado con transformaciones de IU en XML a Compose, lo cual complementa el conocimiento base adquirido con los anteriores resultados mientras que se genera nuevo conocimiento más próximo al tema tratado en la guía de migración final. Esto ha permitido cumplir los dos siguientes objetivos específicos: **“Convertir pantallas de aplicaciones con su IU en XML a Jetpack Compose”**, dado que se ha conseguido replicar la pantalla en cuestión manteniendo el mismo aspecto y comportamiento, y **“Extraer patrones de las transformaciones de IU realizadas”**, puesto que dada la variedad de componentes de IU migrados ha sido posible obtener reglas y patrones a seguir para realizar una transformación efectiva, que han sido parte del conocimiento fundamental plasmado en la guía final. Además, una vez finalizado el desarrollo de este caso de estudio, se han obtenido métricas y conclusiones que han permitido comprobar la veracidad de las hipotéticas ventajas establecidas en la documentación de Jetpack Compose, dando lugar así a la consecución del último objetivo específico: **“Contrastar las hipótesis que se plantean con el uso de Jetpack Compose”**.

- **Guía de migración de interfaces de usuario en Android de XML a Jetpack Compose.** Basándose en el conocimiento obtenido previamente sobre Compose y en el caso de estudio elaborado, se ha conseguido elaborar una guía de migración para transformar una aplicación con su IU definida en XML a Jetpack Compose. El Capítulo 5 de este documento constituye dicha guía.

  Este resultado, que conforma el artefacto principal de este trabajo, permite dar por cumplido el objetivo general planteado: **“Crear una guía de migración de XML a Jetpack Compose en Android”**.

Teniendo en cuenta el avance del conocimiento obtenido, que como se puede observar ha ido desde una perspectiva más general (fundamentos de Jetpack Compose) a una cada vez más centrada en el tema objetivo (migración de IU en XML a Jetpack Compose), y los resultados obtenidos, que han permitido cumplir satisfactoriamente con todos los objetivos planteados inicialmente, se considera que el trabajo se ha completado de manera exitosa.

6.2. Discusión final

El trabajo realizado ha requerido un esfuerzo extra de investigación puesto que como ya se ha mencionado varias veces a lo largo del documento, no existe demasiada documentación al respecto del tema que se trata aquí. Conseguir información útil para la guía ha consistido en alguna ocasión en prueba y error, lo que ha aumentado la complejidad de su elaboración.

Por otra parte, Jetpack Compose es una herramienta muy reciente, lo que implica que está en continua evolución y que algunas funcionalidades de la misma aún no se encuentran en un estado estable. Este hecho ha limitado la posibilidad de hacer una guía más exhaustiva dado que muchas partes dejarían de ser válidas en poco tiempo. Asimismo, esta idea de cambio continuo también apoya la premisa inicial de que actualmente no es viable automatizar el proceso de migración de XML a Compose.

Sin embargo, a pesar del poco tiempo que lleva publicada la herramienta, ha tenido muy buena aceptación por la comunidad de desarrolladores Android y se poseen cada día más recursos sobre documentación básica, cursos y ejemplos variados para poder iniciarse en ella. Tal y como comentan los expertos, y tras la experiencia de la realización de este trabajo, se considera que Jetpack Compose se convierta en la herramienta dominante de definición de IU en los próximos tiempos.

En cuanto al trabajo existente [2] comentado en el Capítulo 3 cuyos objetivos son muy similares al trabajo aquí desarrollado, se considera que la guía de migración de este documento es más completa que lo presentado en dicho artículo. El trabajo de extender ese artículo está actualmente en proceso por el autor, pero no se pondrá a disposición del público general, como sí se hará con el presente trabajo, sino que requerirá suscripción. Además, se considera que la guía desarrollada en este TFM es más exhaustiva, ya que aunque no es posible visualizar el contenido completo del trabajo mencionado, el mismo publicita que se ofrecerán ejemplos de los casos de uso más habituales, mientras que en el presente trabajo se proporcionan las correspondencias de cada uno de los componentes básicos de Android además de un caso de estudio en el que es posible ver ejemplos del uso de algunos de ellos.

6.3. Contraste de hipótesis

A partir de los resultados y métricas obtenidas del caso de estudio (Sección 4.3), en esta sección se realizará un contraste de las cuatro hipótesis planteadas inicialmente en la Sección 2.4 sobre el uso de Jetpack Compose frente a XML.

Menos código

Según se observa en las métricas de la Sección 4.3, la definición original en paradigma imperativo mediante el **sistema de vistas XML** de la pantalla desarrollada en el caso de estudio, está constituida por 6 archivos Kotlin y XML, sumando un total de **763 LLOC**. La misma pantalla definida en paradigma declarativo mediante **Jetpack Compose**, está

constituida por 1 archivo Kotlin con un total de 507 LLOC. Es decir, en este caso, con Jetpack Compose se ha necesitado un 34 % menos de líneas de código para definir la misma pantalla que con XML+Kotlin, por lo que se concluye que con Jetpack Compose **sí se necesita menos código**.

Intuitivo

Dada la subjetividad de esta característica, se ha decidido realizar una prueba a pequeña escala con desarrolladores de la oficina de Izertis de Valladolid, trabajando en distintos ámbitos y con distinto nivel de conocimiento de Android.

La prueba consistía en convocar individualmente a cada desarrollador, mostrar el código de un elemento de IU bien en XML o bien en Jetpack Compose y pedir que dibujara a mano lo que cree que se está definiendo con dicho código. De esta manera, es posible comprobar el modelo mental que el desarrollador crea al tratar de comprender el código. Para obtener conclusiones, se ha medido el **tiempo** que se ha tardado en finalizar el dibujo del modelo mental y la **similitud** con el componente real en una escala de 0 a 5, evaluada por un desarrollador Android experto de la oficina de Izertis de Valladolid, de tal manera que se espera que cuanto más intuitivo sea el código, menos tiempo se tarde en realizar dicho dibujo y mayor sea la similitud con el componente real.

Los códigos utilizados se pueden observar en los Fragmentos de código 6.23 y 6.24. Ambos son adaptaciones de código extraído del caso de estudio y se han tratado de “anonimizar” para no dar pistas sobre el resultado final, por ejemplo cambiando el nombre del *composable* o eliminando los identificadores con nombres descriptivos de los elementos en XML. En ambos casos, lo que se está definiendo es el componente de encabezado de sección, y el resultado visual de ambos modos de definición sería el mostrado en la Figura 6.28.

```
1  <RelativeLayout
2      android:layout_width="match_parent"
3      android:layout_height="wrap_content"
4      android:background="@color/blue"
5      android:paddingBottom="4dp"
6      android:paddingTop="4dp">
7
8      <TextView
9          android:layout_width="wrap_content"
10         android:layout_height="wrap_content"
11         android:layout_centerVertical="true"
12         android:paddingStart="8dp"
13         android:text="Título"
14         android:textAllCaps="true"
15         android:textColor="@color/white"
16         android:textStyle="bold" />
17
```

```
18      <ImageButton
19          android:layout_width="wrap_content"
20          android:layout_alignParentEnd="true"
21          android:layout_marginEnd="8dp"
22          android:background="@color/white"
23          android:src="@drawable/ic_add" />
24
25  </RelativeLayout>
```

Fragmento de código 6.23. Código en XML para representar el componente de encabezado de sección

```
1   @Composable
2   private fun Test() {
3       Row(
4           verticalAlignment = Alignment.CenterVertically,
5           horizontalArrangement = Arrangement.SpaceBetween,
6           modifier = Modifier
7               .fillMaxWidth()
8               .background(color = colorResource(id = R.color.blue
                  ))
9       ) {
10          Text(
11              text = "Título".uppercase(),
12              modifier = Modifier.padding(
13                  start = 8.dp,
14                  top = 15.dp,
15                  bottom = 15.dp),
16              color = colorResource(id = R.color.white),
17              fontWeight = FontWeight.Bold
18          )
19          IconButton(
20              onClick = { }
21          ) {
22              Icon(
23                  painter = painterResource(id = R.drawable.
                      ic_add),
24                  contentDescription = null,
25                  tint = colorResource(id = R.color.white)
26              )
27          }
28      }
29  }
```

Fragmento de código 6.24. Código en Jetpack Compose para representar el componente de encabezado de sección

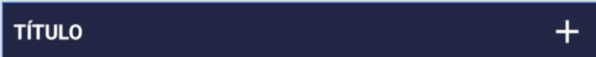


Figura 6.28. Componente de encabezado de sección utilizado en la prueba

La población escogida para la prueba han sido **6 desarrolladores**, 2 de ellos con nivel básico en Android, 2 con nivel medio y 2 con nivel avanzado. Para poder comparar más equitativamente, dentro de cada grupo de distinto nivel en Android a un desarrollador se le ha mostrado el código en XML y al otro el código en Jetpack Compose. Los resultados obtenidos se pueden observar en las Tablas 6.27 a 6.29.

Medición	**Desarrollador 1** (XML)	**Desarrollador 2** (Jetpack Compose)
Tiempo	5:04 minutos	2:43 minutos
Similitud	2/5	4/5

Tabla 6.27. Resultados de la prueba en desarrolladores con nivel básico en Android

Medición	**Desarrollador 3** (XML)	**Desarrollador 4** (Jetpack Compose)
Tiempo	2:28 minutos	3:29 minutos
Similitud	2/5	3/5

Tabla 6.28. Resultados de la prueba en desarrolladores con nivel medio en Android

Medición	**Desarrollador 5** (XML)	**Desarrollador 6** (Jetpack Compose)
Tiempo	1:49 minutos	2:23 minutos
Similitud	4/5	5/5

Tabla 6.29. Resultados de la prueba en desarrolladores con nivel avanzado en Android

Con los anteriores resultados, es posible observar que en todos los niveles, la similitud con el componente real del dibujo creado por los desarrolladores a los que se les ha mostrado el código en Jetpack Compose es mayor que la del dibujo creado por los desarrolladores con el código XML. Sin embargo, tanto los desarrolladores de nivel medio como los de nivel avanzado a los que se les ha mostrado el código en Compose han tardado algo más de tiempo en realizar el dibujo, probablemente debido a su experiencia con XML y la falta de conocimiento y práctica con Jetpack Compose. A pesar de ello, haciendo balance de ambas características, se concluye que en general Jetpack Compose **sí es más intuitivo**.

Acelera el desarrollo

En la Sección 4.3, se ha determinado que definir la IU de la pantalla del caso de estudio mediante Jetpack Compose ha supuesto un total de **49 Horas-Persona**. Puesto que el desarrollo en XML y Kotlin de dicha pantalla ocurrió hace varios años y no se poseen registros sobre el tiempo que se tardó en completar, se ha utilizado el criterio de "opinión de experto" preguntado a uno de los desarrolladores Android más experimentados del proyecto de ownCloud. Teniendo en cuenta que se trata de un desarrollo realizado por alguien sin demasiada experiencia aún con la herramienta, el experto estima que la realización de dicha pantalla con XML por un desarrollador en una situación similar en cuanto a XML habría llevado aproximadamente **60 Horas-Persona**. De haberse tratado de un desarrollador más experimentado, se hubiera reducido el tiempo estimado y muy probablemente el tiempo realmente dedicado. Dadas estas dos medidas, se concluye que utilizando Jetpack Compose **sí se acelera el desarrollo**.

Potente

Con el desarrollo del caso de estudio se ha podido comprobar cómo la utilización de Kotlin exclusivamente hace que la IU se pueda definir de una manera mucho **más dinámica**, aprovechando las características que proporciona el lenguaje de programación. Un claro ejemplo es la definición de las listas de elementos indefinidos, cuyos archivos relacionados se enumeran en la Sección 4.3: mientras que en paradigma imperativo es necesario definir un XML con el contenedor de la lista (`share_file_layout.xml`), otro XML con el aspecto de un elemento de la lista (`..._item.xml`), una clase con el adaptador que permite mostrar dichos elementos en el contenedor de la lista (`...ListAdapter.kt`) y la propia clase que maneja el fragmento que aloja la IU de dicha pantalla (`ShareFileFragment.kt`), en paradigma declarativo se puede definir todo en el mismo archivo (`ShareFileComposeFragment.kt`), y además de forma muy sencilla mediante un *composable* `LazyColumn` y su bloque `items`, como se puede ver en el Fragmento de código 4.9.

Además, dada la flexibilidad que ofrece la utilización de Kotlin, el código es muy modularizable. En este caso, se ha decidido desarrollar todo el código en un único archivo Kotlin, pero se podrían haber creado distintos archivos Kotlin con distintos *composables* cada uno, con el objetivo de facilitar la organización del código de la aplicación.

Cabe destacar que en Jetpack Compose también se aplican ciertos principios de accesibilidad por defecto, que ayudan a crear una IU de más calidad evitando escribir código que de otra manera sería necesario. Por ejemplo, los *composables* `IconButton` poseen por defecto un área de 48 dp x 48 dp, que permite proporcionar una zona suficientemente grande de click para el botón. Con XML, estos principios no se aplican, y si se quiere hacer una interfaz de usuario accesible se debe añadir este tipo de parámetros de forma manual. Análogamente, también se aplican por defecto principios de *Material Design* para proporcionar una IU altamente usable.

Dadas estas razones, se considera razonable concluir que Jetpack Compose **sí es más potente**.

7: Conclusiones y líneas de trabajo futuras

En el presente trabajo, se ha seguido un proceso combinado de investigación y desarrollo, basado en un caso de estudio, para elaborar una guía de migración de IU expresadas en paradigma imperativo (XML) a paradigma declarativo (Jetpack Compose) en aplicaciones Android. Para ello, inicialmente se ha justificado la importancia del tema y se han determinado ciertos objetivos a cumplir para poder calificar el trabajo como exitoso. Posteriormente, se ha enmarcado teóricamente el contexto del trabajo y se ha realizado una investigación sobre el estado del arte en modernización de software, y más en concreto, en Jetpack Compose. Como parte de desarrollo del trabajo, se ha desarrollado como caso de estudio la migración de una pantalla de la app Android de ownCloud de XML hacia Jetpack Compose, pudiendo obtener conocimiento que posteriormente se ha plasmado en la guía de migración final. Para finalizar, a partir del resultado del caso de estudio se han contrastado empíricamente las hipótesis que se plantean con el uso de Jetpack Compose, concluyendo que efectivamente esta forma de desarrollar interfaces de usuario implica ciertas ventajas frente a XML.

Se considera que el resultado es satisfactorio puesto que se han cumplido todos los objetivos, consiguiendo crear una guía de migración no existente en la actualidad de manera oficial, que puede ser de gran utilidad para la comunidad de desarrolladores Android. Además, se han obtenido resultados que realmente ponen de manifiesto la utilidad del trabajo realizado, como un seminario sobre Jetpack Compose para la oficina de Izertis de Valladolid y la planificación de un futuro artículo en el blog oficial de la empresa. Teniendo en cuenta esto, la creciente popularidad de la herramienta y que la conclusión final del trabajo es que en base al caso de estudio desarrollado Jetpack Compose supone menos código, es más intuitivo, acelera el desarrollo y es más potente que XML, se piensa que el resultado obtenido es exitoso y que las numerosas horas invertidas en este trabajo han merecido la pena.

Asimismo, se ha constatado que quizás se ha infravalorado el tiempo que llevaría la elaboración de este trabajo. Partir de un trabajo relacionado con el tema casi nulo y que la mayor parte de valor se aporte mediante generación de conocimiento propio no

es algo sencillo, pero esto ha ayudado a tener un primer contacto con el ámbito de la investigación, que hasta ahora se desconocía, y también a mejorar las estimaciones de tiempo para trabajos que requieran investigación en futuras ocasiones.

En cuanto a líneas de trabajo futuras que podrían continuar el trabajo aquí comenzado, se proponen las siguientes:

- **Actualizar la guía con las futuras versiones de Jetpack Compose que se publiquen.** Como se ha mencionado en varias ocasiones en la guía, a día de hoy existen componentes XML que aún no tienen disponible un componente correspondiente en Jetpack Compose. Puesto que Compose es una herramienta relativamente nueva y está en desarrollo continuo, periódicamente se publican actualizaciones que incluyen nuevas características y componentes que puedan ayudar a completar el catálogo de correspondencias, como ha ocurrido hasta ahora.
- **Completar la guía con más componentes.** En la guía aquí elaborada se ha partido de los componentes disponibles en la paleta de componentes XML de Android Studio, que son los que se han considerado básicos para construir una interfaz de usuario. Sin embargo, existen diversos componentes también nativos de Android que no aparecen en la paleta y que actualmente poseen un análogo en Jetpack Compose.
- **Tests en Jetpack Compose.** Una parte muy importante del código de una aplicación para asegurar la calidad de la misma son los tests. Como se ha mencionado, al cambiar de paradigma, en Jetpack Compose la manera de probar la IU también cambia, por lo que sería una parte fundamental a añadir a la guía de migración que en este trabajo no se ha podido tratar por restricciones temporales.
- **Estilos en Jetpack Compose.** Otra parte relevante en las aplicaciones, y sobre todo pertinente en la parte de IU, son los estilos que se aplican a los elementos de interfaz de usuario. Jetpack Compose posee una forma distinta a XML de gestionar los estilos, aunque existen maneras de migrar los ya existentes. Esta sería otra parte interesante a tratar para incluir en la guía de migración, que se sale del ámbito de este trabajo por limitaciones temporales.

Apéndices

Apéndice A

Plan de proyecto

Para llevar a cabo el presente trabajo, se ha seguido una metodología específica que ha permitido elaborar una planificación de las tareas a realizar. En este apéndice se describe dicha planificación, cómo ha discernido la realidad con la planificación y se enumeran las herramientas utilizadas.

A.1. Herramientas utilizadas

Las herramientas empleadas fundamentalmente para la realización del trabajo se enumeran a continuación:

- **Kotlin.** Lenguaje de programación recomendado por Google para desarrollar aplicaciones Android. Aunque también es posible hacer desarrollo Android con Java, Jetpack Compose sólo está disponible en Kotlin, por lo que es el lenguaje que se ha utilizado durante toda la fase de formación, el desarrollo del caso de estudio y la posterior elaboración de la guía de migración.
- **Android Studio** [11]**.** Entorno de desarrollo (IDE) oficial para aplicaciones Android. A través de él, se han programado, compilado y probado las distintas aplicaciones con las que se ha trabajado. Además, la paleta de componentes XML disponible en este IDE ha servido como referencia para conformar el conjunto de componentes básicos que se tratan en la guía.
- **GitHub** [35]**.** Sistema de repositorios en línea que utiliza el sistema de control de versiones *Git*. Mediante él, se ha obtenido el código de aplicaciones desarrolladas en Jetpack Compose para la fase de formación, el código de la aplicación Android de ownCloud, y se ha utilizado para alojar el código desarrollado en el caso de estudio.

A.2. Planificación del trabajo

Para la elaboración de la planificación, se han tomado como referencia las recomendaciones establecidas en el artículo empleado en el capítulo del estado del arte para enmarcar el trabajo en el ámbito de la modernización de software, *"Software Modernization: Theoretical Framework"* [53].

Puesto que lo que se pretende es seguir un proceso que tenga como entrada un sistema y como salida el mismo sistema pero modificado, haciendo referencia a una aplicación con su IU en XML y a la misma aplicación pero con ciertas partes de su IU en Jetpack Compose, respectivamente, se ha decidido seguir un **modelo en espiral**. Este modelo está basado en **iteraciones**, y cada una de ellas implica planificar, entender el sistema original, entender qué se espera del sistema objetivo, transformar el conocimiento y modificar el sistema.

La metodología aplicada para la planificación del Trabajo de Fin de Máster en su totalidad, incluyendo las tareas de documentación, será una metodología por fases.

Teniendo en cuenta que el Trabajo de Fin de Máster tiene una carga de 6 ECTS, correspondiente a 150 horas, las fases planificadas y la dedicación estimada a cada una de ellas son:

- **FASE 1: Finalización del desarrollo del caso de estudio.** Puesto que este trabajo se inició en la estancia en empresa de la asignatura *I+D+i en Informática*, el desarrollo del caso de estudio se pudo comenzar pero no terminar, por ello se planifica como primera fase con la que se comienza este TFM. En esta fase, aplicándose el modelo en espiral antes mencionado basado en iteraciones, se trata de realizar la última iteración del proceso de migración por modernización.
Dedicación estimada: 20 horas.
- **FASE 2: Documentación inicial del TFM.** Una vez completado todo el código del caso de estudio, se comienza con las primeras partes de documentación del TFM. Esta fase incluye la elaboración de los capítulos "Introducción", "Marco teórico" y "Estado del arte".
Dedicación estimada: 45 horas.
- **FASE 3: Documentación central del TFM.** Habiendo introducido ya el problema a resolver y los conceptos básicos relacionados, así como investigado el estado del arte del tema, se documenta el caso de estudio que se ha desarrollado y se sintetizan los conocimientos obtenidos de él en la guía de migración final. Esta fase incluye la elaboración de los capítulos "Caso de estudio" y "Guía de migración".
Dedicación estimada: 50 horas.
- **FASE 4: Documentación final del TFM.** Para concluir con la documentación del trabajo, se hace una recopilación de los resultados obtenidos a lo largo de todo el proceso, se realiza el contraste de hipótesis final y se concluye y proponen formas

de continuar el trabajo realizado. Esta fase incluye la elaboración de los capítulos "Resultados y discusión" y "Conclusiones y líneas de trabajo futuras", así como del apéndice "Plan de proyecto".
Dedicación estimada: 25 horas.

- **FASE 5: Revisión y correcciones.** El último paso antes de dar por concluido totalmente el trabajo es recibir el visto bueno de los tutores, y para eso se revisa el trabajo entero y se aplican las correcciones y cambios sugeridos por los mismos.
Dedicación estimada: 10 horas.

Dedicación total estimada: 150 horas.

A.3. Ejecución del trabajo

En la ejecución real del trabajo, se ha tratado de seguir la planificación propuesta, pero la dedicación real a cada fase discierne de lo estimado.

El trabajo enmarcado en el contexto del TFM se comenzó el **16 de mayo de 2022** y se finalizó el **11 de julio de 2022**, teniendo así una duración de aproximadamente 2 meses.

La dedicación real a cada fase planificada ha sido la siguiente:

- **FASE 1: Finalización del desarrollo del caso de estudio.** 24 horas.
- **FASE 2: Documentación inicial del TFM.** 54 horas.
- **FASE 3: Documentación central del TFM.** 66 horas.
- **FASE 4: Documentación final del TFM.** 15 horas.
- **FASE 5: Revisión y correcciones.** 10 horas.

Por tanto, la dedicación total real para la realización del Trabajo de Fin de Máster ha sido de **169 horas**.

Bibliografía

[1] ALEX STYL. From View to Composable: Learn Jetpack Compose from a Android View mindset. https://viewtocomposable.com/. Accedido el 15/06/2022.

[2] ALEX STYL (MEDIUM). From View to Composable: A Quick jump to Jetpack Compose from an Android View mindset (with cheat sheet). https://medium.com/@alexstyl/views-to-composables-d715b92c6055. Accedido el 15/06/2022.

[3] ANDROID DEVELOPERS. Adding Jetpack Compose to your app - Start migrating to Compose. https://developer.android.com/jetpack/compose/interop/adding#start-migrate. Accedido el 24/04/2022.

[4] ANDROID DEVELOPERS. Android Jetpack. https://developer.android.com/jetpack. Accedido el 07/06/2022.

[5] ANDROID DEVELOPERS. Android's Kotlin-first approach. https://developer.android.com/kotlin/first. Accedido el 08/06/2022.

[6] ANDROID DEVELOPERS. Arrangement. https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/Arrangement. Accedido el 05/07/2022.

[7] ANDROID DEVELOPERS. Compose to Kotlin Compatibility Map. https://developer.android.com/jetpack/androidx/releases/compose-kotlin. Accedido el 27/06/2022.

[8] ANDROID DEVELOPERS. ConstraintLayout in Compose. https://developer.android.com/jetpack/compose/layouts/constraintlayout. Accedido el 05/07/2022.

[9] ANDROID DEVELOPERS. Cuvva builds faster and with higher quality with Compose. https://developer.android.com/stories/apps/cuvva-compose. Accedido el 22/04/2022.

[10] Android Developers. Data Binding Library. https://developer.android.com/topic/libraries/data-binding. Accedido el 23/06/2022.

[11] Android Developers. Download Android Studio & App Tools. https://developer.android.com/studio. Accedido el 10/07/2022.

[12] Android Developers. Get started with Jetpack Compose. https://developer.android.com/jetpack/compose/documentation. Accedido el 21/04/2022.

[13] Android Developers. Guide to app architecture - Common architectural principles. https://developer.android.com/topic/architecture#common-principles. Accedido el 27/06/2022.

[14] Android Developers. Guide to app architecture - Recommended app architecture. https://developer.android.com/topic/architecture#recommended-app-arch. Accedido el 27/06/2022.

[15] Android Developers. Guide to app architecture - UI layer. https://developer.android.com/topic/architecture#ui-layer. Accedido el 07/06/2022.

[16] Android Developers. Interoperability APIs. https://developer.android.com/jetpack/compose/interop/interop-apis. Accedido el 23/04/2022.

[17] Android Developers. Material Components and layouts - Scaffold. https://developer.android.com/jetpack/compose/layouts/material#scaffold. Accedido el 07/07/2022.

[18] Android Developers. Modifier. https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier. Accedido el 30/06/2022.

[19] Android Developers. Navigating with Compose. https://developer.android.com/jetpack/compose/navigation. Accedido el 28/06/2022.

[20] Android Developers. Square sees increase productivity with Compose. https://developer.android.com/stories/apps/square-compose. Accedido el 22/04/2022.

[21] Android Developers. State and Jetpack Compose. https://developer.android.com/jetpack/compose/state. Accedido el 08/06/2022.

[22] Android Developers. Thinking in Compose - Recomposition. https://developer.android.com/jetpack/compose/mental-model#recomposition. Accedido el 08/06/2022.

[23] Android Developers. Thinking in Compose – The declarative programming paradigm. https://developer.android.com/jetpack/compose/mental-model#paradigm. Accedido el 22/04/2022.

[24] Android Developers. Twitter sees increased developer efficiency and velocity with Compose. https://developer.android.com/stories/apps/twitter-compose. Accedido el 22/04/2022.

[25] Android Developers. View Binding. https://developer.android.com/topic/libraries/view-binding. Accedido el 23/06/2022.

[26] Android Developers. Why adopt Compose. https://developer.android.com/jetpack/compose/why-adopt. Accedido el 08/06/2022.

[27] Ashni Sharma (Codementor.io). UI Design Principles Every Mobile App Developer/Designer Must Follow. https://www.codementor.io/design/tutorial/mobile-app-ui-design-principles. Accedido el 06/06/2022.

[28] B. Verhaeghe, A. Etien, N. Anquetil, A. Seriai, L. Deruelle, S. Ducasse, M. Derras. GUI Migration using MDE from GWT to Angular 6: An Industrial Case. *26th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering* (2019). https://hal.archives-ouvertes.fr/hal-02019015. Accedido el 11/07/2022.

[29] Caulo, M., Francese, R., Scanniello, G., Tortora, G. Implications on the Migration from Ionic to Android. *Product-Focused Software Process Improvement* (2021), 3–19. https://doi.org/10.1007/978-3-030-91452-3_1. Accedido el 14/06/2022.

[30] @chaitanyamunje (GeeksforGeeks). Tab Layout in Android using Jetpack Compose. https://www.geeksforgeeks.org/tab-layout-in-android-using-jetpack-compose/. Accedido el 07/07/2022.

[31] Clarivate. Web of Science. https://www.webofscience.com/wos/alldb/basic-search. Accedido el 14/06/2022.

[32] Codeplayon. Jetpack compose custom rating bar in android 2022. https://www.codeplayon.com/2022/05/jetpack-compose-custom-rating-bar/. Accedido el 09/07/2022.

[33] Cognitios.co. ISO 9241 y la usabilidad. https://www.cognitios.co/usabilidad-con-iso-9241/. Accedido el 06/06/2022.

[34] Francesc Vilarino Guell. Creating a ViewPager in Jetpack Compose. https://fvilarino.medium.com/creating-a-viewpager-in-jetpack-compose-332d6a9181a5. Accedido el 06/07/2022.

[35] GitHub. GitHub. https://github.com/. Accedido el 10/07/2022.

[36] Google Accompanist. WebView wrapper for Jetpack Compose. https://google.github.io/accompanist/webview/. Accedido el 09/07/2022.

[37] Google Developers. Answers to some common questions about appearing in Google News. https://developers.google.com/search/blog/2021/07/google-news-top-questions. Accedido el 22/04/2022.

[38] Google Developers. Maps Compose Library. https://developers.google.com/maps/documentation/android-sdk/maps-compose. Accedido el 30/06/2022.

[39] Intense Coder. Spinner in Jetpack compose (DropDown). https://intensecoder.com/spinner-in-jetpack-compose-dropdown/. Accedido el 05/07/2022.

[40] Izertis. Izertis. https://www.izertis.com/es/. Accedido el 24/04/2022.

[41] Jacinto Araque (Business Insider). Android vs iPhone: Este mapa muestra la preferencia en móviles de cada país. https://www.businessinsider.es/android-vs-iphone-mapa-muestra-preferencia-cada-pais-903103. Accedido el 21/04/2022.

[42] John Codeos. How to create a Navigation Drawer with Jetpack Compose. https://johncodeos.com/how-to-create-a-navigation-drawer-with-jetpack-compose/. Accedido el 07/07/2022.

[43] K. Garcés, R. Casallas, C. Álvarez, E. Sandoval, A. Salamanca, F. Viera, F. Melo, J.M. Soto. White-box modernization of legacy applications: The oracle forms case study. *Computer Standards & Interfaces 57* (2018), 110–122. https://doi.org/10.1016/j.csi.2017.10.004. Accedido el 11/07/2022.

[44] Kevin D Moore (raywenderlich.com). Introduction to Android Jetpack. https://www.raywenderlich.com/5376-introduction-to-android-jetpack. Accedido el 07/06/2022.

[45] Manuel Vicente Vivo (Android Developers). Migrating to Jetpack Compose. https://developer.android.com/codelabs/jetpack-compose-migration. Accedido el 12/04/2022.

[46] Material Design. Design. https://material.io/design. Accedido el 06/06/2022.

[47] Matthew O'Riordan (Ably). Everything You Need To Know About Publish/-Subscribe. https://ably.com/topic/pub-sub. Accedido el 11/07/2022.

[48] MK (Dev Bits and Bytes). Jetpack Compose – A Simple Opiniated AutoCompleteTextView. https://www.devbitsandbytes.com/jetpack-compose-a-simple-opiniated-autocompletetextview/. Accedido el 08/07/2022.

[49] Mohit Sarveiya. Strategies for Migrating to Jetpack Compose. https://twitter.com/androidww_/status/1515465077826240516?t=oxG0sTdbHkdveYkiR_WlqA&s=19. Accedido el 15/06/2022.

[50] Moore, Rugaber, Seaver. Knowledge-based user interface migration. *Proceedings 1994 International Conference on Software Maintenance* (1994), 72–79. https://ieeexplore.ieee.org/document/336788. Accedido el 11/07/2022.

[51] ownCloud (GitHub). The ownCloud Android App. https://github.com/owncloud/android. Accedido el 24/04/2022.

[52] Refactoring.Guru. Observer. https://refactoring.guru/design-patterns/observer. Accedido el 11/07/2022.

[53] S. Bragagnolo, N. Anquetil, S. Ducasse, A. Seriai, M. Derras. Software Modernization: Theoretical Framework. *Information and Software Techonology* (2022), 1–25. https://ssrn.com/abstract=4054806. Accedido el 12/06/2022.

[54] Sebastian Rieger (Medium). Create a ChipGroup with Jetpack Compose. https://medium.com/@Rieger_san/create-a-chipgroup-with-jetpack-compose-f4744b94fa34. Accedido el 08/07/2022.

[55] Steven Hoober & Eric Berkman. Designing Mobile Interfaces: Patterns for Interaction Design. http://4ourth.com/wiki/Designing%20Mobile%20Interfaces. Accedido el 06/06/2022.

[56] Thracian (Stack Overflow). Jetpack Compose UI: How to create SearchView? https://stackoverflow.com/a/69605371. Accedido el 09/07/2022.

[57] WhatsApp LLC (Play Store). WhatsApp Messenger. https://play.google.com/store/apps/details?id=com.whatsapp. Accedido el 09/07/2022.

[58] Xue Qin, Hao Zhong, Xiaoyin Wang. TestMig: Migrating GUI Test Cases from iOS to Android. *PROCEEDINGS OF THE 28TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS (ISSTA '19)* (2019), 284–295. https://dl.acm.org/doi/10.1145/3293882.3330575. Accedido el 14/06/2022.