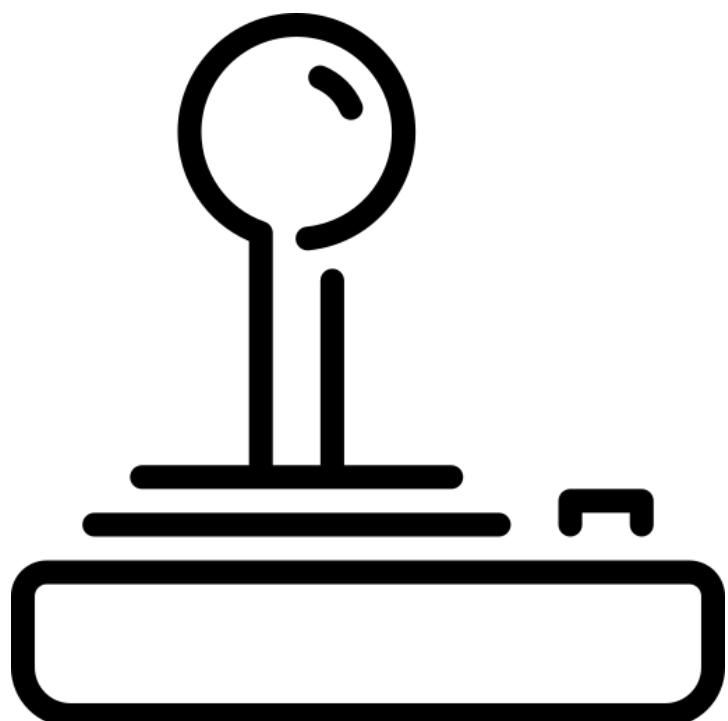


# Proyecto Integrado

## Keys Store App



Autor: Víctor Pareja Ramírez

# Índice

1.- Introducción.....	3
2.- Objetivos del proyecto.....	5
3.- Planificación del proyecto.....	7
3.1.- Planificación del lado del servidor.....	7
3.2.- Gestión de la información dentro del proyecto.....	9
3.3.- Diseño de la aplicación.....	11
4.- Análisis y diseño del sistema.....	12
4.1.- Modelo conceptual.....	12
4.2.- Creación de base de datos y consultas principales.....	14
5.- Especificaciones del sistema.....	16
5.1.- Instalación y configuración de la aplicación.....	16
5.2.- Elaboración de las especificaciones de hardware.....	17
6.- Especificaciones del software.....	19
6.1.- Descripción de operaciones, interfaz y navegación.....	20
7.- Código fuente relevante.....	34
8.- Conclusiones del proyecto.....	39
9.- Apéndice.....	42
10.- Bibliografía.....	44

# Introducción

KeysStoreApp es una aplicación nativa en la que el usuario puede comprar claves de juegos que pueden ser canjeadas en otras plataformas. Esta aplicación está desarrollada en Jetpack Compose, un marco de desarrollo de interfaz de usuario declarativa y moderna para aplicaciones Android. Además, Jetpack Compose ha recibido mucha atención por parte de la comunidad Android desde su lanzamiento, y cuenta con el respaldo de Google, por lo que es posible que Jetpack Compose tenga un papel importante en el futuro del desarrollo de aplicaciones Android.

El proyecto KeysStoreApp está más enfocado en el aprendizaje de esta nueva herramienta (Jetpack Compose) que en el desarrollo de una aplicación completa, es decir, que tenga frontend, un backend más complejo... El proyecto está desarrollado siguiendo la arquitectura propuesta por Google y utilizando las mejores prácticas para controlar el flujo de información, hacer las peticiones al servidor, controlar los estados de las vistas, etc.

KeysStoreApp es una aplicación sencilla en la que, de manera fácil, puedes comprar claves. Hoy en día, muchos jugadores, especialmente los de PC, compran estas claves de producto porque suelen estar bastante más baratas que en la tienda oficial donde se vende este producto. El precio de estas claves es reducido porque los que las venden compran el juego “al por mayor”, por así decirlo, y pueden vender estas claves más fácilmente ya que el juego tiene un precio mayor en las tiendas. Es por esto por lo que KeysStoreApp es una buena idea, porque la venta de claves de juegos no ha hecho más que crecer en estos últimos años y esta aplicación ofrece una manera fácil de comprar estas claves.

En internet ya hay plataformas que ofrecen este servicio, como puede ser Instant Gaming, que ofrece un servicio excelente, pero lo que ofrece

KeysStoreApp es una aplicación de fácil uso y, muy importante, con una buena arquitectura y un código que puede escalar de manera intuitiva sin que el desarrollo de nuevas funcionalidades sea excesivamente complejo.

La idea de KeysStoreApp surgió cuando probé aplicaciones de empresas que comercian con estas claves de juegos. Estas aplicaciones están más centradas en el negocio web, es por esto por lo que se me ocurrió hacer una aplicación nativa con este modelo de negocio, ya que las aplicaciones nativas de estas aplicaciones no iban tan bien como su aplicación web.

# Objetivos del proyecto

Una de las funcionalidades más importantes es el almacenamiento de información en KeysStoreApp, que se guarda/recupera la información desde varias bases de datos y preferencias del sistema.

En primer lugar, la información de la cuenta de usuario, como puede ser el email, nombre, configuración de la aplicación como, por ejemplo, el modo oscuro, etc. se almacena utilizando DataStorePreferences, librería que se utiliza para guardar y recuperar datos de forma persistente en la memoria del dispositivo. Esto significa que los datos se mantienen aún después de que la aplicación se cierre o se reinicie el dispositivo.

Otra forma de almacenamiento de información es la utilización de RoomDatabase, que proporciona una capa de abstracción sobre SQLite, lo que permite una fácil integración con el código de la aplicación. En esta base de datos interna de la aplicación, se almacenan los datos de los carritos de compra que tiene el usuario, es decir, todos los carros que haya creado el usuario junto a los productos que haya añadido a cada uno de ellos. Una de las razones del uso de Room en el proyecto es que proporciona un sistema de migración de datos, lo que permite actualizar el esquema de la base de datos de manera fácil y sin perder datos existentes.

Por último, la última herramienta para la gestión de datos es Firebase Database, en la que se gestiona la creación de cuentas de usuario, se almacenan todos los productos que hay en la aplicación, se gestionan los datos de las compras que haya hecho un usuario y, por último, se almacena una lista de deseos con los productos que haya marcado el usuario. Una de las ventajas de Firebase es que los datos cambian en tiempo real, lo que

hace que los datos se “recolecten” en cuánto están disponibles para ser leídos. Firebase también ofrece un monitoreo de la aplicación, en la que se pueden ver los errores que se han producido durante su ejecución (excepciones, errores anr, etc.).

Otro objetivo importante al margen de la gestión y almacenamiento de información en KeysStoreApp, es el diseño de la aplicación. La aplicación tiene un diseño moderno y original que ha seguido las pautas propuestas por Material Design, es una especificación de diseño creada por Google, que incluye aspectos como la interfaz de usuario, la interacción, la animación y, además, utiliza un diseño plano y minimalista, con una paleta de colores brillantes y fuertes.

Como he mencionado en la introducción, el objetivo principal del proyecto es aprender más sobre Jetpack Compose, porque ha venido para quedarse y estoy seguro de que en estos años que vienen muchas aplicaciones que están en el top descargas migrarán a Compose. El objetivo no es aprender Compose simplemente, sino aprenderlo siguiendo las guías y referencias de Google y Material Design.

# Planificación del proyecto

## Planificación del lado del servidor

KeysStoreApp no es una aplicación que tenga la información alojada en un servidor como tal, sino que esta información se almacena en Firebase. A diferencia de un servidor, como los de DigitalOcean, que tiene una base de datos relacional (mysql, mariadb...), este proyecto obtiene la información a partir de colecciones y documentos, que se almacenan dentro de cada colección. Estos documentos no tienen una relación directa, ya que no es una base de datos relacional, sino que son documentos que “simulan” ser un json en el que se almacena la información de la siguiente manera: campo - valor.

La elección del uso de Firebase en el proyecto era arriesgada, ya que nunca había usado este tipo de “base datos”, a diferencia de una base de datos relacional, con la que tengo más experiencia. Por suerte, la implementación de Firebase en el proyecto Android es bastante sencilla, además de contar con una muy buena documentación, por lo que esta implementación no llevó mucho tiempo.

El siguiente paso, una vez tuve Firebase implementado en el proyecto, era la creación de un modelo de datos para los documentos de cada colección. La colección principal, “products”, contiene documentos que, a su vez, almacenan la información de cada producto (nombre, precio, categoría, etc.). Las otras colecciones que se utilizan en el proyecto, son, en primer lugar, “wishlist”, documentos en los que se almacena un listado de productos seleccionados por el usuario; y “purchaseHistory”, que guarda la información de cada compra que haya hecho cualquier usuario. La creación de los modelos de las colecciones no ha supuesto una gran inversión de tiempo

porque Firebase proporciona bastante facilidad a la hora de añadir o quitar nuevos parámetros a los documentos.

Cuando se creó la colección principal, “products”, creé los documentos para poder empezar a hacer las peticiones desde la aplicación. La creación de estos productos y su almacenamiento en los documentos de la colección de Firebase si que llevó bastante tiempo, ya que se tenían que buscar las url de dos imágenes por cada documento, el nombre, la categoría, etc. Para que no supusiera esta tarea más tiempo del necesario, he hecho que en la aplicación haya dos campos estáticos con el mismo texto (la tagline y descripción) para cada uno de los productos, evitando así tener que copiar y pegar texto manualmente a cada documento de “products”. Para las otras colecciones, “wishlist” y “purchaseHistory”, no hizo falta una inserción de datos inicial, ya que se va rellenando a medida que el usuario usa la aplicación.

Por último, una de las mayores ventajas del uso de Firebase, es la gestión de errores, tanto excepciones como errores arn (ActivityNoResponse). Cuando implementas Firebase en el proyecto Android, automáticamente se enviará al servidor de Firebase toda la información y datos del usuario (dispositivo, versión de Android...) y almacena toda esta información en una tabla de errores del proyecto. De esta manera, todos los errores quedan registrados, y puedes ver tanto la información del error que sale en Android Studio cuando la aplicación falla, como los datos del dispositivo, fecha, localización, versión de Android, etc. También durante el desarrollo he ido añadiendo eventos de Firebase personalizados, para contar cuantas peticiones se hacían al entrar en una ventana, al añadir un producto a la wishlist, etc. En resumen, la gestión de errores y eventos de Firebase ha supuesto una gran mejora, comparándolo con el anterior proyecto que hice el curso pasado, ya que me ha facilitado la visualización de estos errores. Como he dicho anteriormente, esta parte de Firebase se puede utilizar desde que se

integra Firebase en el proyecto, sin necesidad de hacer ningún ajuste más, por lo que su implementación no me ha llevado nada de tiempo y me ha ahorrado mucho tiempo esta funcionalidad de Firebase.

## Gestión de la información dentro de la aplicación

KeysStoreApp está desarrollada completamente en Jetpack Compose, herramienta que hasta entonces nunca había probado, aunque la versión 1.0 de Compose se lanzó hace como un año. La idea de realizar el proyecto en Jetpack Compose sirve para aprender una nueva herramienta de Android que, probablemente, sea el futuro del desarrollo de los móviles. Al comienzo, Compose es demasiado abstracto y no es fácil la adaptación de los diseños cuando siempre los has realizado en archivos xml, pero en cuanto te adaptas, la creación de vistas y todo el potencial de reutilización de código que ofrece es impresionante. Me llevo cerca de 2 semanas el sentirme cómodo con el código de Compose y aún así siento que no sé absolutamente nada, ya que cuenta con muchísimas funciones interesantes relacionadas con el ciclo de vida, la persistencia de datos, etc.

Este proyecto sigue la arquitectura recomendada por Android, y sigue los patrones de diseño de Material Design de Google. Uno de los patrones de diseño que KeysStoreApp utiliza es el Singleton y la inyección de dependencias con Hilt, que ayuda al mantenimiento y desarrollo de la app. Con los Singleton he mejorado la jerarquía de clases, haciendo que dependan de abstracciones en lugar de otras clases en concreto. Otra de las ventajas es que garantiza que solo exista una instancia de una clase/objeto, haciendo este objeto accesible desde cualquier lado de la aplicación pero evitando múltiples objetos iguales.

KeysStoreApp está desarrollada con las dos capas de abstracción de datos que recomienda Android: la UI Layer, que muestra los datos y las vistas de la aplicación, además de atender cualquier interacción del usuario; y la DataLayer, que está compuesta por repositorios, que se implementan en los ViewModels, y por la fuente de datos, que hace la petición directamente al servidor y recibe los datos de manera “cruda”. Para la comunicación entre estas capas, se utilizan los Flows (flujos) de kotlin, una herramienta que proporciona la capacidad de realizar operaciones asíncronas, de manera que puede hacer operaciones en segundo plano sin bloquear el hilo de la interfaz del usuario y mostrando los datos cuando el flujo de datos se haya emitido.

La utilización de estos patrones, tanto el patrón Singleton, como la arquitectura de capas, eran nuevos para mi y el aprendizaje y comprensión han sido bastante progresivos, porque era mucha cantidad de información que no era fácil de entender. El uso de estos patrones facilita mucho el desarrollo de la aplicación, pero para que te lo facilite tienes que entenderlo y saber por qué se hace cada cosa si no, es muy difícil de implementar.

La lectura de datos desde los documentos de Firebase me dieron bastantes problemas en un comienzo, ya que los datos se pedían pero no conseguía que la conversión a objeto se hiciera correctamente. Una vez lo conseguí, estos objetos, los productos por ejemplo, se emiten mediante el flujo al ViewModel, que gestiona el estado de las peticiones y el estado de la interfaz de usuario. En cuanto a la lectura de datos con Room, se ha realizado con la interfaz Dao, que recomienda Android, que hace de capa de fuente de datos, ya que contiene todas las peticiones a la base de datos SQLite, y también, de igual manera, los flujos de kotlin. Por último, al haber usado DataStore, los datos almacenados en las preferencias de la aplicación también se leen mediante flujos. Cabe mencionar que estos flujos emite los datos dentro de una corutina ejecutada en el hilo IO, es decir el hilo de la entrada y salida de información.

## Diseño de la aplicación

En cuanto al diseño de la aplicación, la elección de colores de KeysStoreApp sigue la recomendación de paletas de Material Design, evitando así que algunos colores no encajen con el resto y dándole más consistencia al diseño de la app. He optado por una gama morada-rosada para los detalles de la aplicación y como colores principales para el fondo de la aplicación el blanco en modo día y un negro grisáceo para el modo oscuro. Para darle más personalidad al aspecto de los textos, he cambiado la fuente, para que no utilice la que viene por defecto. Aplicar los colores y fuentes a la KeysStoreApp ha sido muy sencillo, ya que Compose lo deja muy fácil y accesible el seteo tanto de los colores (ya sean colores de fondo o texto), como para la de la fuente.

Sobre el diseño gráfico de la aplicación, he elegido que las vistas tengan el borde redondeado, ya que le da un aspecto más moderno. La mayor parte de estas vistas que tienen el borde redondeado, cuentan con los detalles en los colores principales de la aplicación.

KeysStoreApp contiene la información organizada siguiendo el estándar de aplicaciones similares. La aplicación se compone de un menú de navegación inferior desde el que se puede navegar a través de las ventanas principales. En cuanto a la ficha de un producto, la información está estructurada de la manera estándar, pero con toques personales, como por ejemplo la animación del ícono de la ficha, animación que me supuso más tiempo del que debería, pero ha quedado muy bien.

# Análisis y diseño del sistema

## Modelo conceptual

El almacenamiento de datos en Firebase no es una base de datos de tipo relacional, por lo que no hay relación entre las directas entre las tablas. Aunque se almacene el id de un producto, por ejemplo, en una compra, realmente no hay una relación directa entre una tabla y otra, sino una simple referencia en el documento de la compra. En Firebase, cada documento vendría a ser una tupla dentro de una tabla y la tabla sería una colección. Las colecciones contienen documentos con un identificador para que pueda ser referenciado desde cualquier lado. Los documentos dentro de esta colección son los que contienen los datos reales de la tupla: id, nombre, precio, etc.

storeapp-2bb50		products	⋮	1	⋮
+ Iniciar colección		+ Agregar documento	+ Iniciar colección		
products	>	1	>	+ Agregar campo	
purchases		10		category: "Open-World"	
wishlists		11		iconUrl: "https://cdn.cloudflare.steamstatic.com/steamcommunity/public/images/apps/5bf6edd7efb1110b457da905e7ac696c	
		12			
		13		id: 1	(número)  
		14		imageUrl: "https://cdn.cloudflare.steamstatic.com/steam/apps/1174180/header.jpg?t=1656615305"	
		15			
		16			
		17		name: "Red Dead Redemption 2"	
		18		nameLowercase: "red dead redemption 2"	
		19		price: 559	
		2		tagline: "A tag line"	
		20		wishlist: false	
		21			

Por otro lado, la base de datos interna de la aplicación si es relacional. Esta base de datos consta de dos tablas, la tabla de carritos y la de los productos del carrito. Ambas tablas tienen una relación de muchos a muchos,

ya que un carrito puede contener muchos productos y un producto puede estar en muchos carritos al mismo tiempo. Esta base de datos cumple la tercera forma normal, ya que la tabla está libre de dependencias funcionales además de que todos los campos de la tabla dependen de la clave principal.

```
@Entity(  
    tableName = "cart_products",  
    foreignKeys = [  
        ForeignKey(  
            entity = CartEntity::class,  
            parentColumns = ["id"],  
            childColumns = ["cart_id"],  
            onDelete = ForeignKey.CASCADE  
        ),  
    ],  
    indices = [  
        Index(value = ["cart_id"])  
    ]  
)  
  
data class CartProductEntity(  
    @PrimaryKey(autoGenerate = true)  
    var id: Long = 0,  
    @ColumnInfo(name = "product_id")  
    val productId: Long,  
    @ColumnInfo(name = "name")  
    val name: String,  
    @ColumnInfo(name = "icon_url")  
    val iconUrl: String,  
    @ColumnInfo(name = "image_url")  
    val imageUrl: String,  
    @ColumnInfo(name = "price")  
    val price: Long,  
    @ColumnInfo(name = "category")  
    val category: String,  
    @ColumnInfo(name = "cart_id")  
    val cartId: Long,  
    val addDate: Long  
)  
  
@Entity(  
    tableName = "cart",  
)  
data class CartEntity // (   
    @PrimaryKey(autoGenerate = true)  
    var id: Long = 0,  
    @ColumnInfo(name = "name")  
    val name: String,  
    @ColumnInfo(name = "itemCount")  
    val itemCount: Int = 0,  
)  
()
```

## Creación de la base de datos y consultas principales

La creación de la base de datos de Firebase se hace de manera muy sencilla en el panel de control de Firestore Database. Para crear una base de datos en Firebase, hay que tener una cuenta de Firebase, que al ser de Google es la misma cuenta, y una vez ingresado en el panel dirigirse a Firestore Database. La primera vez que entras en esta pestaña, se iniciará una configuración rápida de la base de datos, y en pocos clics ya tienes la base de datos lista para poder crear colecciones y documentos. Lo único que hay que configurar está dentro de la pestaña Firestore Database, en concreto una sección llamada “Reglas”. Lo que hay que cambiar es la fecha de timestamp.date, que es la fecha hasta la que se pueden realizar consultas a la base de datos, tanto de escritura como de lectura.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
        request.time < timestamp.date(2023, 1, 31);
    }
  }
}
```

Por otro lado, la base de datos Room se genera automáticamente en tiempo de compilación cuando tienes una clase con la anotación @Database, además de que en esta clase abstracta extienda de RoomDatabase y tenga todos los @Dao declarados.

```

@Database(
    entities = [
        CartEntity::class,
        CartProductEntity::class
    ],
    version = 1,
    exportSchema = true,
)
abstract class StoreAppDatabase : RoomDatabase() {
    abstract fun cartDao(): CartDao
    abstract fun cartProductsDao(): CartProductDao
}

```

Las consultas principales de esta base de datos son:

- `SELECT * FROM cart ORDER BY id DESC` - Para obtener todos los carritos existentes del usuario en una lista de carros.
- `SELECT * FROM cart WHERE id == :id` - Obtener un carro en concreto dando como referencia un ID de carro.
- `SELECT EXISTS(SELECT * FROM cart_products WHERE cart_id = :cartId OR name = :cartName)` - Comprobar si, dado un ID de carro, existe ya esa tupla en base de datos.
- `DELETE FROM cart WHERE id in (:ids)` - Eliminar un carro dado un ID de carro.
- `SELECT EXISTS(SELECT * FROM cart_products WHERE product_id = :productId AND cart_id = :cartId)` - Comprobar si ya se ha añadido un producto a un carrito.
- `DELETE FROM cart_products WHERE product_id in (:productId) AND cart_id in (:cartId)` - Eliminar un producto de un carrito dando el id del producto y del carrito.

Para las consultas tanto de inserción como de actualización de una entidad, no es necesario crear consultas, ya que la anotación de Room `@Insert` y `@Update` se encarga de hacer la consulta.

# Especificaciones del sistema

## Uso de tecnología y software empleado

KeysStoreApp es un proyecto Android desarrollado en Jetpack Compose, que es un marco de desarrollo que ayuda al desarrollo de la aplicación gracias al lenguaje declarativo. El lenguaje de programación de Compose es Kotlin, un lenguaje de código abierto basado en Java pero más conciso y fácil de entender.

Para la base de datos de la aplicación he utilizado Firebase Database, que ofrece de manera gratuita un espacio para poder crear los documentos (lo que sería una tupla en una tabla de mysql) necesarios para el proyecto. Además Firebase se actualiza en tiempo real por lo que la lectura de los datos a través de los flujos de kotlin es instantánea.

## Instalación y configuración de la aplicación

La instalación de la aplicación es muy sencilla, ya que el proyecto solo consta de una aplicación Android. Para descargar la aplicación, escanea el QR que hay abajo. Para instalarla, pulsa en el fichero descargado y el sistema debería empezar la instalación o pedir permisos para instalar aplicaciones de terceros (permiso que hay que otorgarle al sistema para el uso de la aplicación).

Una vez instalada la aplicación, el usuario tiene que crear una cuenta (sin necesidad de verificación) para poder acceder al contenido de la aplicación. Cuando el usuario tenga la cuenta creada, podrá realizar todas las posibles operaciones que permite realizar KeysStoreApp.

## Elaboración de las especificaciones hardware del sistema

El hardware en el que se ha realizado el proyecto es un MacBook Pro de 14 pulgadas. El sistema cuenta con 500GB de memoria interna y 16GB de memoria RAM. El procesador es un Apple M1 Pro, la primera generación de los procesadores de Apple. El sistema operativo actual del dispositivo es macOS Ventura, la última versión del SO de Apple.



La aplicación ha sido probada en un Samsung Galaxy S22 Ultra. El dispositivo cuenta con un procesador **Exynos 2200**, desarrollado por AMD en el último año. Tiene 256GB de memoria interna y 12 GB de memoria RAM. El sistema operativo es Android 13, con la última capa de Android de Samsung One UI 5.0.

3:18 100% •

## About phone



Victor's S22 Ultra

Edit

Phone number	Unknown
Product name	Galaxy S22 Ultra
Model name	SM-S908B/DS
Serial number	R3CT204B7WV
IMEI (slot 1)	354321590176121
IMEI (slot 2)	354324430176129

Status information

Legal information

Software information

Battery information

3:19 100% •

## Software information

One UI version  
5.0

Android version  
13

Google Play system update  
1 July 2022

Baseband version  
S908BXXU2BVKB

Kernel version  
5.10.66-android12-9-25489753-abS908BXXU2BVKB  
#2 Tue Nov 15 16:02:42 KST 2022

Build number  
TP1A.220624.014.S908BXXU2BVKB

SE for Android status  
Enforcing  
SEPF\_SM-S908B\_12\_0001  
Tue Nov 15 16:13:15 2022

Knox version  
Knox 3.9  
Knox API level 36  
Knox ML 1.3  
DualDAR 1.5.1  
HDM 2.0 - 1D

Service provider software version  
SACMG\_GM\_S908B\_GYM\_EUY\_12\_0076

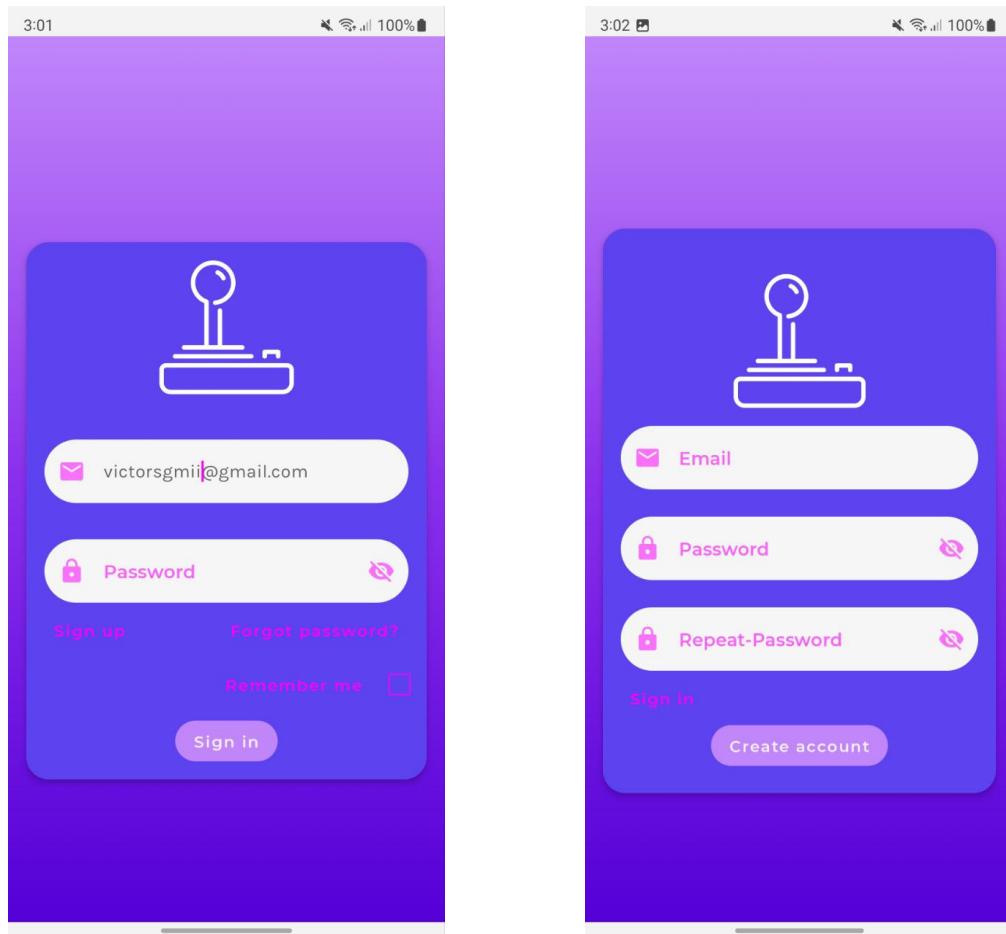
# Especificaciones del software

El proyecto KeysStoreApp no utiliza librerías de terceros, sólo utiliza librerías oficiales y verificadas tanto por Android como por Jetpack Compose. El proyecto implementa muchas librerías, ya que en Jetpack Compose está todo desarrollado en módulos pequeños, evitando así que haya funcionalidades en la librería que no se vayan a usar y solo aumentan el peso de la aplicación.

Aunque no sean librerías de terceros, cabe mencionar que las librerías más importantes que he utilizado para desarrollar la aplicación, obviando todas las relacionadas con Compose, son: Hilt, un marco de inyección de dependencias que proporciona clases y anotaciones que hacen más fácil la implementación de dependencias; Coil, una biblioteca de imagen que proporciona la carga de imágenes para aplicaciones Android; y por último las librerías relacionadas con Firebase, tanto para la autorización de usuario como para la lectura de documentos de Firebase, y Room junto a su compilador.

## Descripción de operaciones, interfaz y navegación de la aplicación

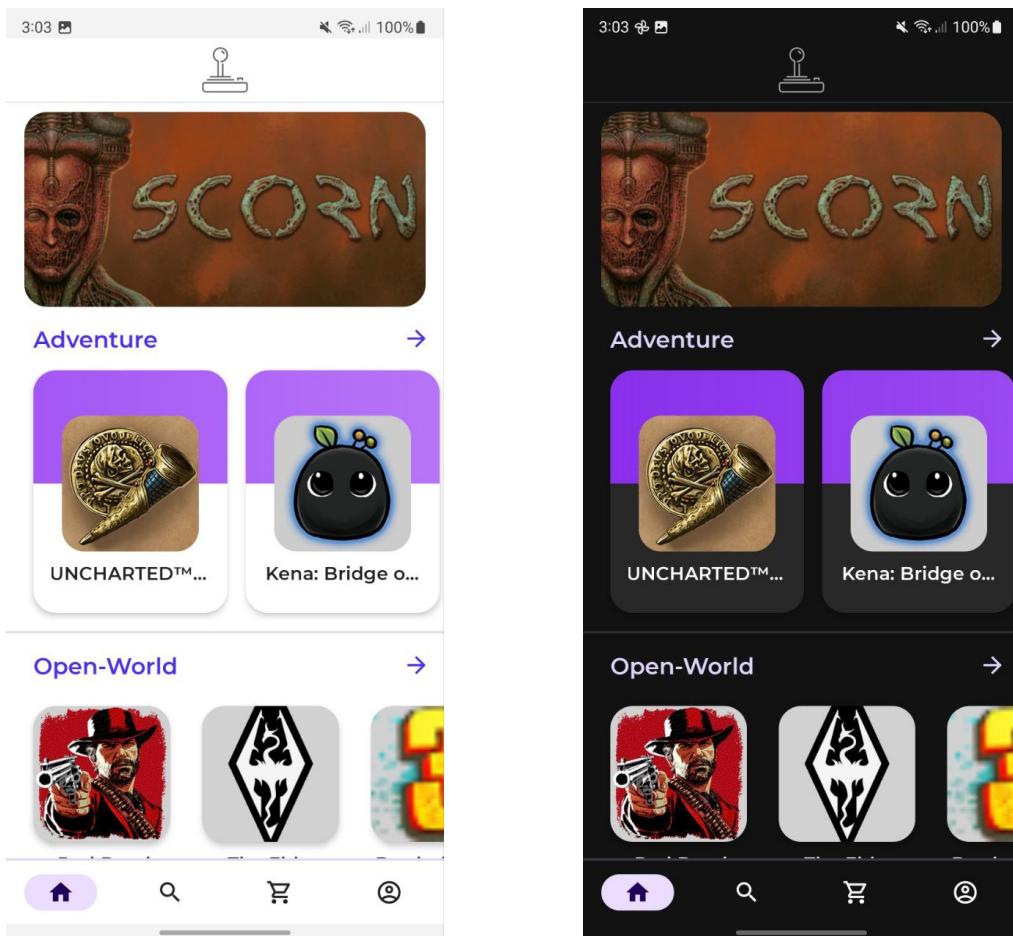
### SignUp y Login



La primera ventana o actividad de la aplicación cuando la abres por primera vez al instalarla es la ventana de Login. En esta ventana, el usuario podrá iniciar sesión mediante email y password en la aplicación, pero será necesaria una cuenta de usuario. En esta ventana, debajo del campo password, hay varias opciones: en primer lugar dirigirse a la ventana de creación de cuenta (Signup), la segunda es la opción de recuperar contraseña y la tercera es un botón seleccionable que si está activado al hacer inicio de sesión no es necesario introducir las credenciales cuando se cierra y se vuelve a abrir la app en otro momento.

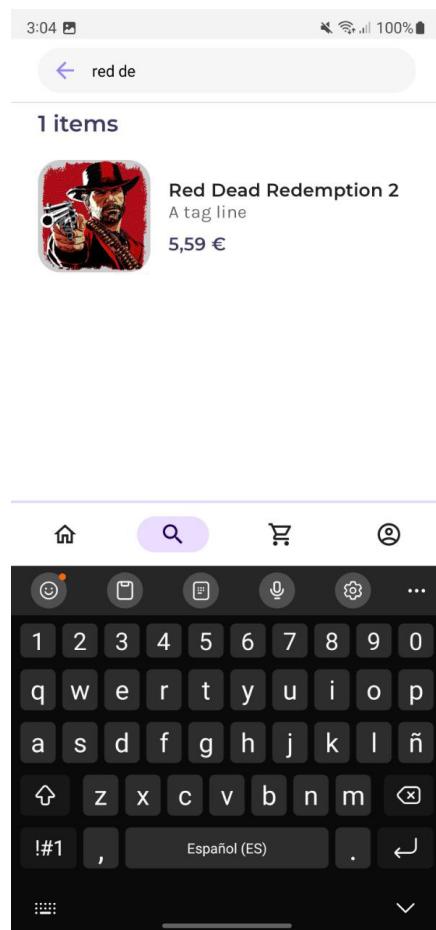
En la ventana de signup, el usuario podrá crear una cuenta de usuario con email y password. Desde esta ventana se puede volver a la ventana de inicio sesión, ventana a la que se dirige la navegación de la aplicación cuando la cuenta está creada. Cuando se crea la cuenta, no es necesario que el usuario verifique esta cuenta para poder entrar en la aplicación.

#### Ver la Feed



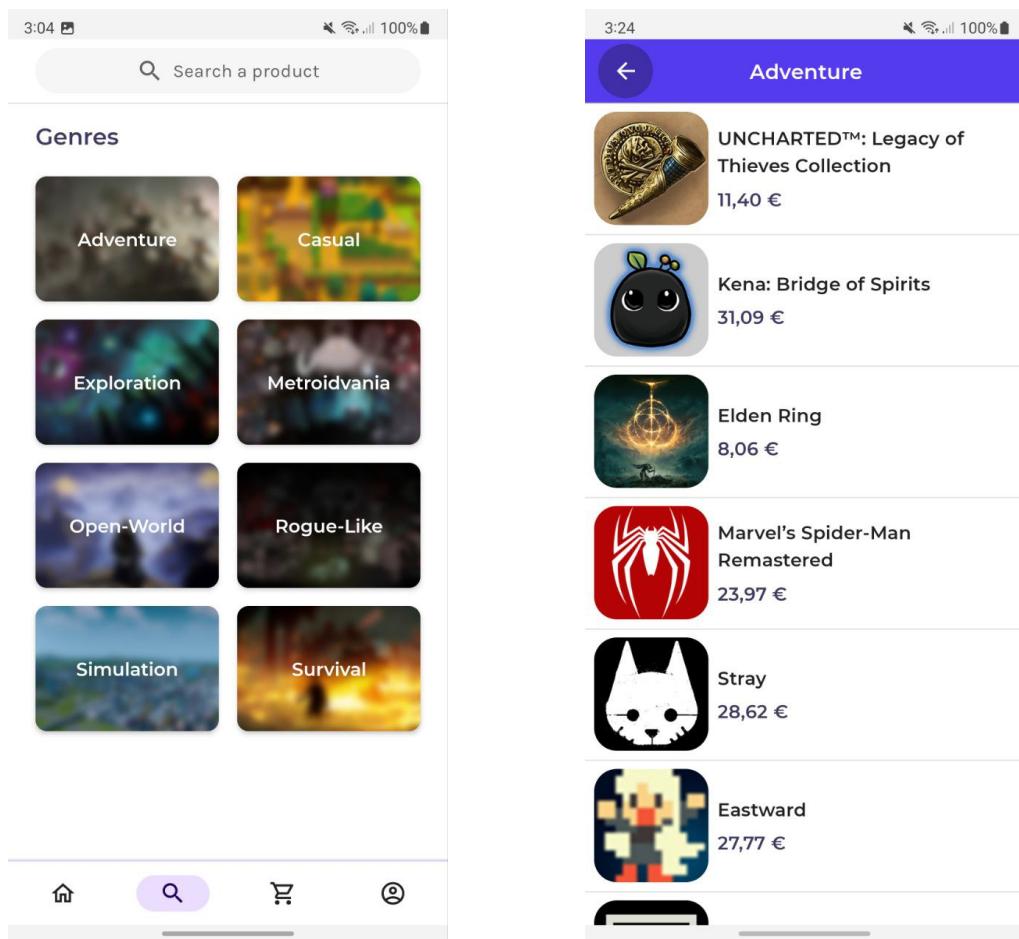
El inicio de la navegación de la aplicación es esta ventana, donde está el contenido de la aplicación. Desde esta ventana el usuario podrá ver cualquiera de las fichas que se presentan en la ventana o ver un listado por categoría en caso de pulsar sobre la flecha que está al final del nombre de la categoría. También se podrá navegar a las otras ventanas de navegación de la aplicación.

## Buscador



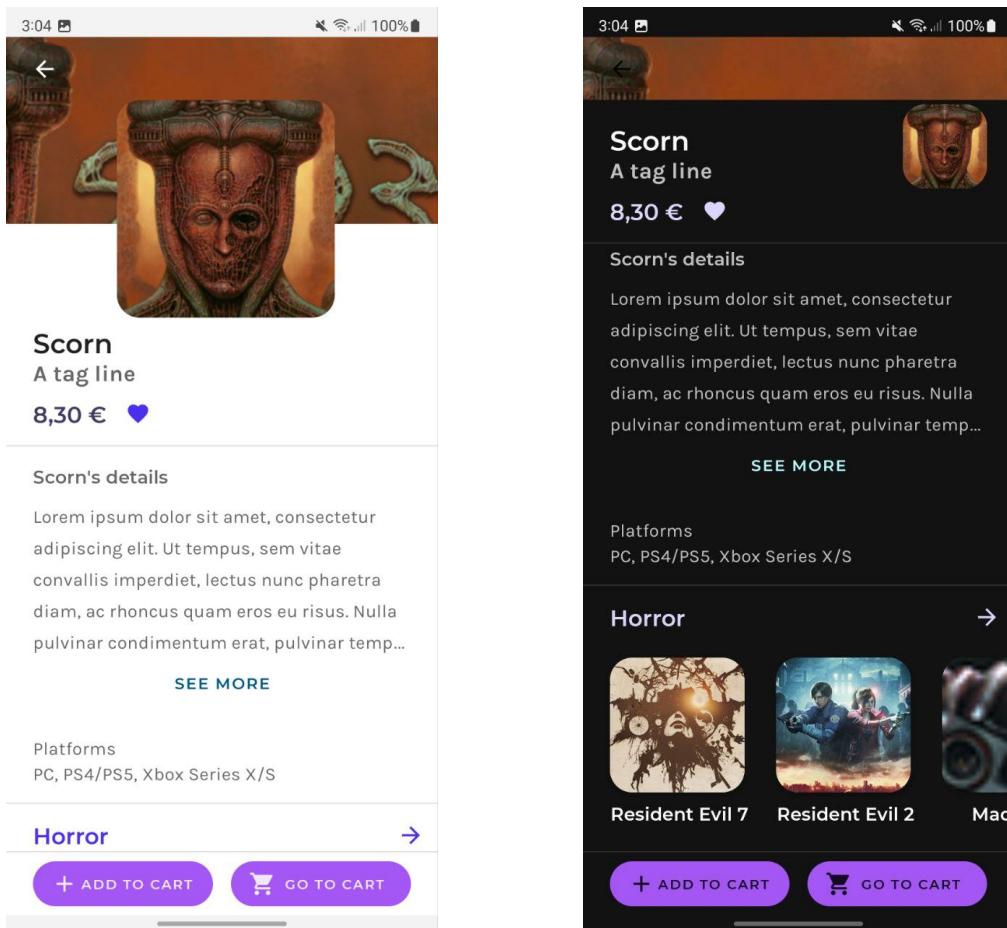
La ventana del buscador, la segunda opción en el menú de navegación, tiene dos funcionalidades; la primera (próxima captura) muestra un listado de cards con cada una de las categorías de juegos que hay en la aplicación y la segunda es el propio buscador, en el que se puede realizar una búsqueda por nombre de un juego en concreto, como vemos en la captura de arriba. Al desarrollar esta funcionalidad, para realizar una petición mas simple y Firebase no hiciera más trabajo del necesario, limité la búsqueda al nombre del juego pero en minúsculas, ya que de la otra manera se tendrían que hacer varias consultas por cada letra que se introducía en el campo de búsqueda y no era eficiente.

## Ver listados por categorías



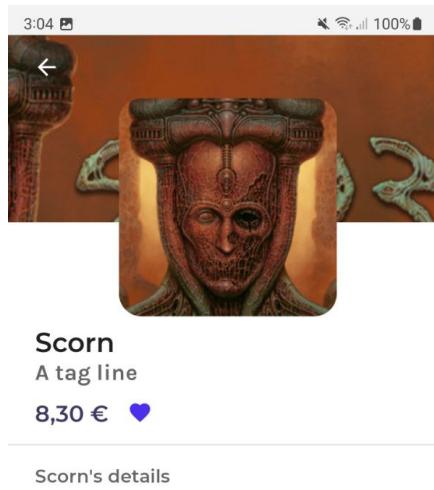
Como he mencionado anteriormente, en esta ventana se encuentra tanto el buscador como este listado de cards de categorías. Al pulsar en alguna de estas cards, el usuario navegará a una ventana que contiene un listado de juegos cuya categoría es la que se ha pulsado. El usuario podrá ver cual es la categoría de estos juegos en la barra de tareas superior de la aplicación. En estos listados por categorías, el usuario puede clicar en los juegos para dirigirse a la ficha, en la que se encuentra la información de cada uno.

## Ver la ficha de un producto



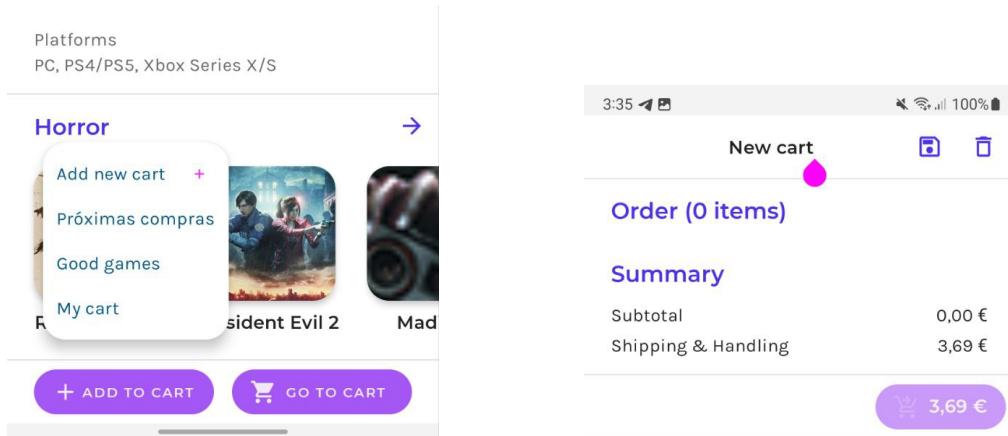
En la ficha de un juego se muestran detalles del producto que no se ven desde las cards de la feed o cualquiera de los listados. En la ficha se pueden realizar varias acciones: añadir a la lista deseos, crear un carro, añadir el producto al carrito, visitar otras fichas de productos con la misma categoría e ir al carro directamente, funcionalidades que iré describiendo mas adelante. En la ficha no he puesto descripción para cada uno de los productos porque no afecta a nada, simplemente sería un campo mas que leer, por eso he el texto de las descripciones de los productos es siempre el mismo.

## Añadir/eliminar a la wishlist



En la ficha de un juego, a la derecha del precio, hay un ícono que es un corazón, que al ser pulsado por el usuario se añadirá a una lista de deseos privada del usuario. Si el corazón tiene un único color (está relleno) significa que ese producto ya está añadido a la lista de deseos, si se ve hueco significa que no está en la lista y puede ser añadido.

## Crear/eliminar carritos

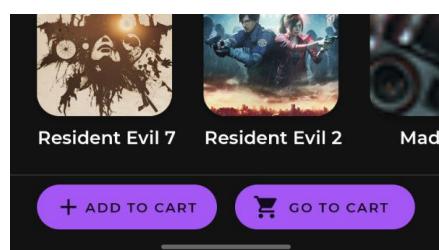


En la ventana de la ficha, al pulsar en el botón de añadir carrito se mostrará un desplegable que muestra todos los carritos que el usuario ha creado y una opción para crear un carrito. Al pulsar en esta opción, aparecerá el teclado y el usuario podrá nombrar su nuevo carrito y guardándolo

pulsando sobre el ícono de guardado que aparece a la derecha del nombre del nuevo carro.

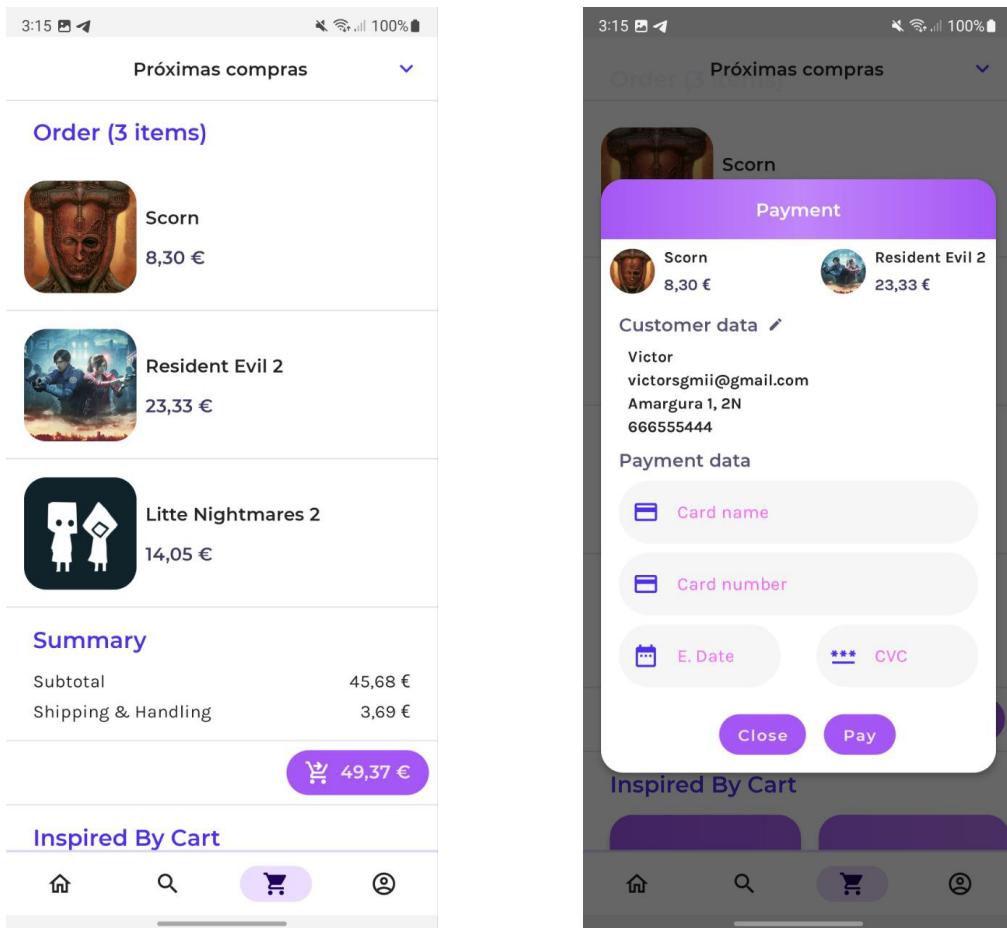
Para eliminar un carrito, el usuario puede clicar en ir al carrito, a la derecha del botón de añadir un producto al carrito. Cuando se pulsa en este botón, la aplicación navega hasta la ventana del carro, en el que el usuario puede, pulsando el nombre del carro, cambiar el nombre del carro seleccionado o borrarlo pulsando en la papelera que está en el lado derecho del ícono de guardar.

Añadir/eliminar producto al carrito



Desde la ventana de la ficha, el usuario podrá añadir el producto actual que se está mostrando en la ficha al carrito de compra. Al pulsar en añadir al carrito, aparecerá el desplegable de la captura de más arriba en el que puedes añadir el producto a cualquiera de los carritos que hayas creado. El mismo juego no lo puedes añadir dos veces al mismo carrito y, una vez añadido, se podrá eliminar desde el propio carrito pulsando en la X que se muestra en la parte superior derecha del producto.

## Comprar



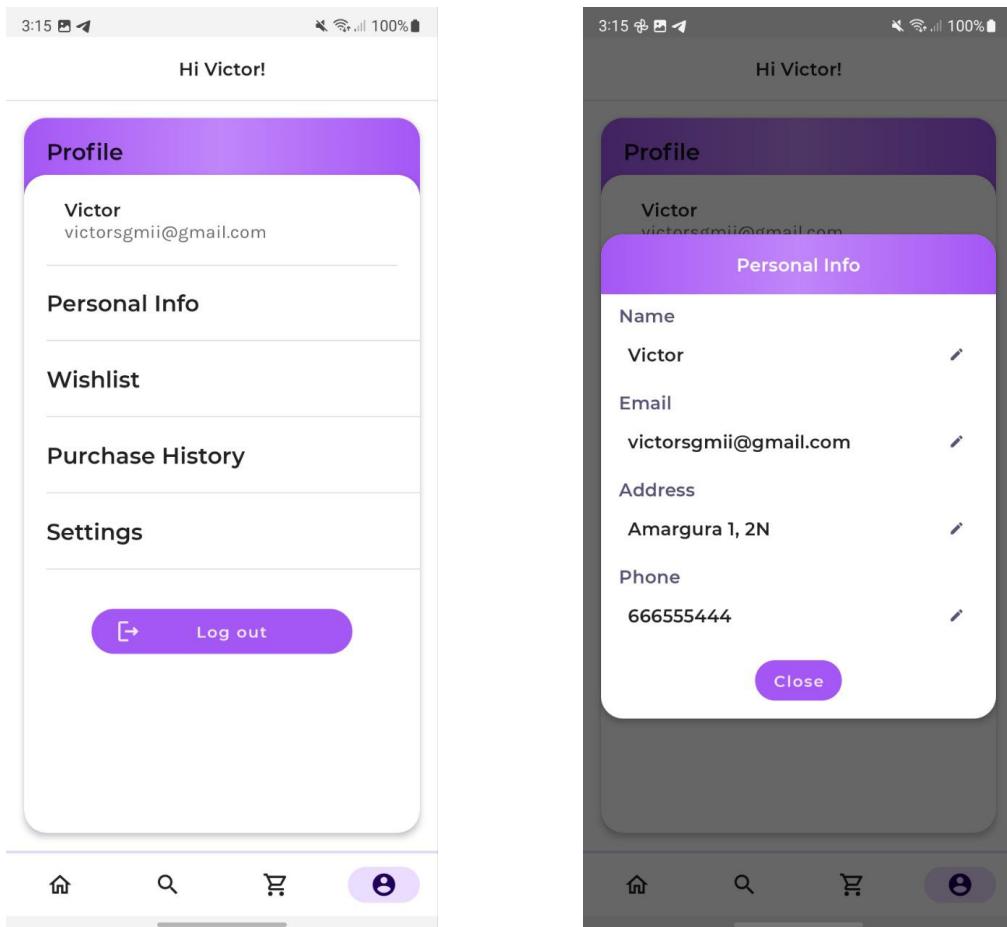
En la ventana de compra, la tercera pestaña en la navegación principal de la aplicación, se pueden realizar varias acciones:

- Ver los productos de cada carro. En cada carro se ve un listado de productos que el usuario ha añadido anteriormente. Pulsando en algún producto del carro se abre la ventana de la ficha del producto seleccionado.
- Cambiar entre los distintos carros que el usuario haya creado. El usuario puede cambiar entre los carros que haya creado pulsando en el ícono de expandir. Cuando se pulsa este botón aparece el mismo menú desplegable que aparece en la ficha cuando se añade un producto al carro. Con este menú desplegado, el usuario puede cambiar a otro carro y

automáticamente se actualizará la ventana para mostrar la información conveniente.

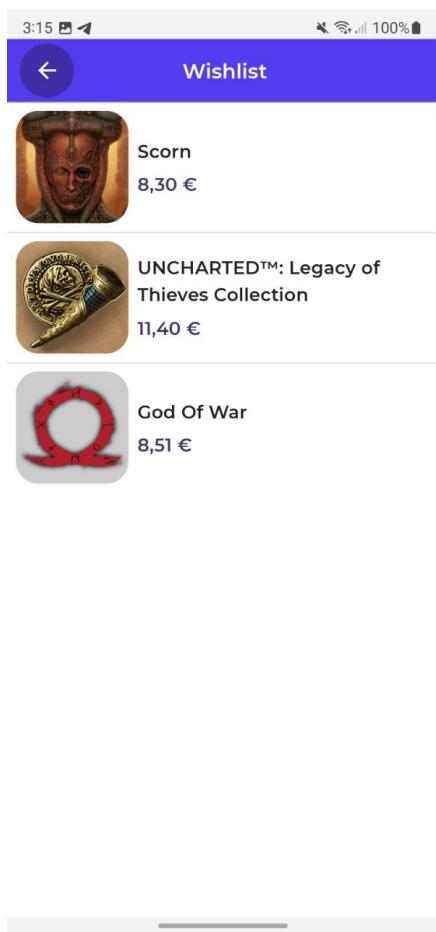
- Cambiarle el nombre al carro que esté seleccionado. Pulsando en el nombre del carro, aparecerá el teclado para poder cambiarle el nombre al carro y aparecerán dos botones a la derecha del nombre: el primero para guardar el nuevo nombre y el otro para eliminar el carro de la base de datos.
- Eliminar el carro seleccionado. Como he mencionado, al pulsar en el nombre aparece la opción de eliminar el carro.
- Iniciar el proceso de compra. Si hay productos en el carro, el botón para iniciar el pago está activo, en cambio, si no hay productos el botón (en el que pone el precio) está desactivado. Al pulsar en este botón, si hay productos en el carro, aparecerá el diálogo de compra. En este diálogo el usuario puede ver un listado horizontal scrolleable con todos los productos del carro; datos personales del usuario que se vinculan con la compra, como el nombre, teléfono, dirección, etc. estos datos se pueden cambiar desde la ventana del perfil o pulsando en el botón de editar a la derecha del título de los datos del comprador; por último están los datos de pago, que incluye el nombre, número de tarjeta, fecha de vencimiento y el código de seguridad. Cuando los datos estén correctamente puestos, el botón de pago estará activo para realizar la compra. Para que los datos estén correctos, el formato del número de tarjeta es un número de 8 dígitos que empieza por 4, la fecha ha de tener el formato 00/00 y el CVC es un número de 3 dígitos. Con los datos correctos, al pulsar en el botón de pago la compra se realizará y aparecerá en el historial de compras del usuario.
- Ver un listado de aplicaciones. Por último, en la parte inferior de la ventana de los carritos, se puede ver un listado de aplicaciones horizontal similar a los de la Feed.

## Cambiar los datos de la cuenta



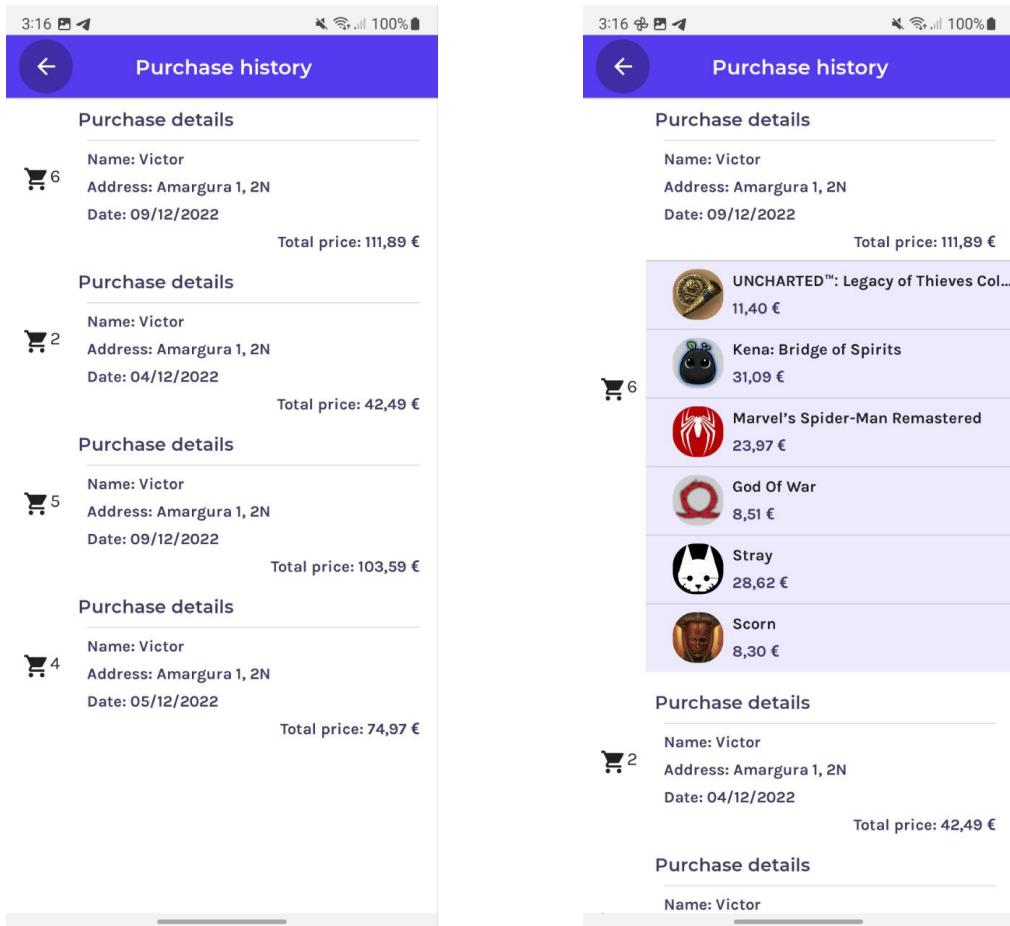
La última ventana de navegación de la aplicación es la del usuario. En esta ventana, el usuario puede realizar varias acciones. La primera opción abre un diálogo en el que el usuario puede establecer o modificar los campos nombre, email, dirección y teléfono, que son datos útiles para realizar una compra.

## Ver el la lista de deseos



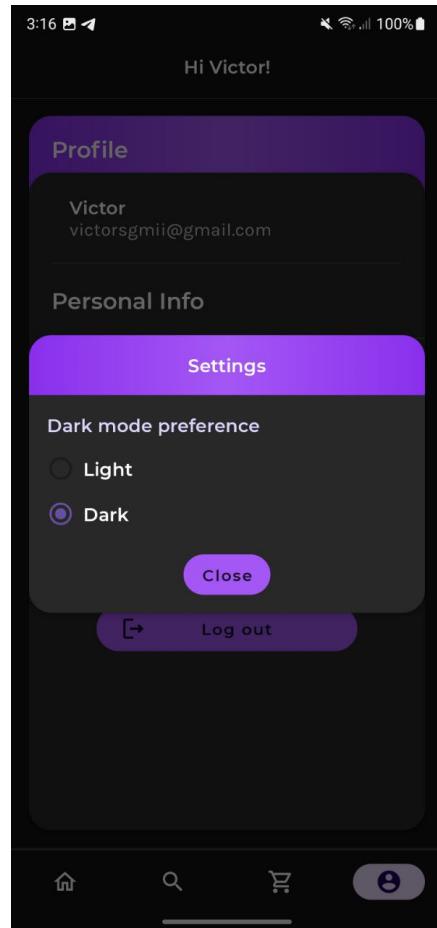
La segunda opción en el panel de usuario es la lista de deseos. En esta lista están todos los productos que el usuario ha añadido a su lista. Es un listado simple, similar al listado por categoría, en el que el usuario puede navegar a la ficha del juego al clicar sobre él.

## Ver el historial de compras



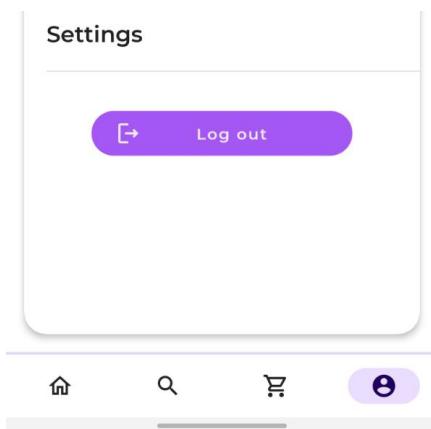
La tercera opción de la ventana del usuario es el historial de compras. En el historial de compras aparecen todas las compras realizadas por el usuario. En cada ítem de compra sale la información básica de la compra, algunos datos del usuario, la fecha de compra y el precio total de la compra. Si el usuario pulsa sobre alguna de estas compras, se despliega un listado con los juegos que se han comprado junto con su precio y logo, cada uno de estos ítems es cliclable y navega hacia la ficha del producto. Al pulsar en la compra estando el listado desplegado, se pliega de nuevo.

## Cambiar de modo claro a oscuro



Por último, la última opción en la ventana de usuario son los ajustes. En los ajustes, en esta primera versión de la aplicación, el usuario puede alternar entre el modo claro y oscuro de la aplicación. El cambio de los colores se hace en tiempo real y al instante, por lo que no hace falta recargar la ventana ni la aplicación.

Logout



En la ventana de usuario, debajo de todas las opciones, hay un botón para cerrar la sesión de usuario activa. Al pulsar este botón, devuelve al usuario a la ventana de login, la cual tendrá el email guardado, ya que se guarda en las preferencias de la aplicación al hacer inicio de sesión.

# Código fuente relevante

A mi parecer, hay bastante código relevante en la aplicación, pero me quedaré con ciertos fragmentos de código que han sido nuevos para mí, tanto por el uso de Compose como por el patrón Singleton con Hilt.

```
setContent {  
    val systemUiController = rememberSystemUiController()  
    val darkTheme = shouldUseDarkTheme(uiState)  
  
    // Update the dark content of the system bars to match the theme  
    DisposableEffect(systemUiController, darkTheme) { this: DisposableEffectScope  
        systemUiController.systemBarsDarkContentEnabled = !darkTheme  
        onDispose {} ^DisposableEffect  
    }  
  
    StoreApp(darkTheme = darkTheme)  
}
```

En MainActivity, la única actividad de todo el proyecto, tiene este fragmento de código, en el que se declara el contenido de la aplicación. La función setContent es una función de Compose para declarar funciones Compostables, como shouldUseDarkTheme(), DisposableEffect o StoreApp. Primero, se hace una consulta a las preferencias del usuario para saber en qué modo tiene la aplicación (oscuro o claro). Después se llama a la función StoreApp, que contiene todo el contenido de la aplicación mediante funciones anidadas. Jetpack Compose, a diferencia de los proyectos convencionales de Android, es un lenguaje declarativo por lo que no se necesitan actividades ligadas a un diseño en XML, sino que estas vistas se declaran en funciones kotlin bajo la anotación @Composable.

```
@Composable
private fun ProductListItemContent(
    product: Product
) {
    Row(
        modifier = Modifier
            .padding(vertical = 8.dp)
            .padding(start = 8.dp)
            .fillMaxWidth(),
        verticalAlignment = Alignment.CenterVertically,
    ) {
        this: RowScope
        ProductImage(
            imageUrl = product.iconUrl,
            contentDescription = null,
            modifier = Modifier.size(100.dp)
        )
        Column(
            verticalArrangement = Arrangement.spacedBy(8.dp)
        ) {
            this: ColumnScope
            Text(
                text = product.name,
                style = MaterialTheme.typography.subtitle1,
                color = StoreAppTheme.colors.textSecondary,
                overflow = TextOverflow.Ellipsis
            )
        }
    }
}
```

Como ejemplo de función composable pondré un fragmento de una función que lo que hace es pintar un producto de un listado. Este fragmento corresponde al diseño de un ítem del listado por categorías de la aplicación.

```

@Module
@InstallIn(SingletonComponent::class)
object DataStoreModule {

    @Provides
    @Singleton
    fun providesUserPreferencesDataStore(
        @ApplicationContext context: Context,
    ): DataStore<Preferences> =
        PreferenceDataStoreFactory.create(
            corruptionHandler = ReplaceFileCorruptionHandler(
                produceNewData = { emptyPreferences() }
            ),
            migrations = listOf(SharedPreferencesMigration(context, USER_PREFERENCES)),
            scope = CoroutineScope(context + SupervisorJob()),
            produceFile = { context.preferencesDataStoreFile(USER_PREFERENCES) }
        )
}

```

Otra parte del código que me parece muy importante, es la declaración de módulos Singleton. De esta manera, se consigue crear un módulo completamente accesible desde cualquier parte de la aplicación y usar esa misma instancia sin estar creando instancias nuevas por cada vez que se llama. Al crear este objeto se le pasan ciertos parámetros de configuración, en este caso, para las preferencias de la aplicación, como el scope, es decir, en qué hilo se va a ejecutar cualquier petición que se haga a estas preferencias.

```

navigation{
    route = MainDestinations.HOME_ROUTE,
    startDestination = TopLevelDestination.FEED.route
} (this: NavGraphBuilder)
    addHomeGraph(onProductSelected, onNavigateTo, onClearAndNavigate, onNavigateTo)
}

composable{
    route = "${MainDestinations.PRODUCT_DETAIL_ROUTE}/{$MainDestinations.CATEGORY_ID_KEY}/{$MainDestinations.PRODUCT_ID_KEY}",
    arguments = listOf(
        navArgument(MainDestinations.PRODUCT_ID_KEY) { type = NavType.LongType },
        navArgument(MainDestinations.CATEGORY_ID_KEY) { type = NavType.StringType }
    )
} (navBackStackEntry ->
    ProductDetail(
        upPress = upPress,
        onProductClick = { id, category -> onProductSelected(id, category, navBackStackEntry) },
        onNavigateTo = { route -> onNavigateTo(route, navBackStackEntry) }
)
}

```

La navegación en la aplicación no se hace declarando actividades en AndroidManifest, ni utilizando las funciones de AppCompatActivity para iniciar actividades. Para declarar una ventana en la aplicación se hace de esta manera, creando un composable, pasándole como parámetros los argumentos (como los extras en una actividad) y la ruta (como si fuera un directorio). Dentro de este composable se ha de llamar a la función @Composable que inicia la ventana, en el caso de la captura, ProductDetail, que también recibe unos parámetros, como por ejemplo una función que se encarga de navegar entre distintos destinos.

```
Get product details by productId

fun getProductDetailsById(productId: Long): Flow<Product> = flow {
    var product = Product()
    db.collection(collectionPath: "products") CollectionReference
        .whereEqualTo(field: "id", productId) Query
        .get() Task<QuerySnapshot!>
        .addOnSuccessListener { it: QuerySnapshot!
            for (doc in it) {
                product = doc.toObject(Product::class.java)
            }
        }
        .addOnFailureListener { it: Exception
        }
        .await()
    emit(product)
}
```

Por último, como código relevante respecto a las consultas que se hacen a Firebase, he puesto una consulta para obtener los datos de un producto dado un ID. La función se ejecuta dentro de un flow, que emite el producto en forma de flow a las funciones superiores. En esta función se recibe un documento y mediante el parseo toObject() pasan los datos de formato JSON a objeto Product. La función await espera que se ejecute todo

lo anterior antes de emitir nada, para que no emita un producto vacío mientras no haya obtenido el producto aún al hacer la llamada a la función.

# Conclusiones del proyecto

El desarrollo de este proyecto ha sido bastante importante para mi ya que me he introducido en una nueva manera de desarrollar aplicaciones Android, gracias a Jetpack Compose.

Yo tenía ya un proyecto hecho y que pude haber presentado, pero leí información sobre Compose y que su versión 1.0 había salido recientemente y decidí empezar un proyecto de cero en Jetpack Compose. El proyecto, KeysStoreApp, ha llevado un desarrollo, en general, algo lento, ya que me he tenido que documentar sobre muchas tecnologías y librerías con las que no estaba familiarizado, pero aún así estoy bastante contento con el estado de la aplicación actual, a pesar de haberme dejado cosas que me hubiera gustado implementar.

La aplicación en sí no tiene funcionalidades complejas puesto que no las necesita al ser una tienda de productos. No me he querido centrar en que la aplicación haga muchas operaciones distintas, en primer lugar porque el desarrollo de estas me podría haber llevado mucho tiempo porque no tengo tanta práctica con Compose, y en segundo lugar porque he querido centrar el proyecto en el aprendizaje de Compose. Me alegra que haya podido sacar este proyecto adelante sin muchos problemas durante el desarrollo, ya que necesito práctica con esta nueva forma de desarrollar aplicaciones porque no paro de leer cosas buenas sobre Compose, o sea que en un futuro se estandarizará el uso de este marco de desarrollo.

Mi idea es seguir con el desarrollo de este proyecto e ir practicando, mejorando y actualizando las funcionalidades que ya están implementadas, una vez haga la presentación del mismo. Hay funcionalidades en las que no me he centrado para no consumir el tiempo del que disponía para el desarrollo de funcionalidades que eran obligatorias para mi, como por

ejemplo la verificación de usuario, o una estricta validez de los datos con regex. Ambas funcionalidades no están implementadas porque mi proyecto no se centraba en eso, en cambio tiene un registro de usuario sin verificación de email, para que sea más rápido el testeo de la aplicación; por el otro lado, todos los campos de la aplicación están validados mediante regex (los datos de la tarjeta, email, passwords...), pero no es un regex tan estricto para que, de nuevo, el testeo sea más rápido y fácil.

Cuando empecé el proyecto, quise plantearlo bien desde un principio y que todo esté bien organizado y encapsulado para que sea de fácil comprensión. He podido llevar a cabo este orden, aunque hay cosas que se pueden mejorar, fragmentos de código que se pueden cambiar pero se han quedado tal y como están porque no quería invertir mas tiempo del necesario para alguno de estos aspectos. Una de mis ideas en un principio con este proyecto, es hacerlo completamente público en Github, junto con una documentación más detallada, para que otros desarrolladores puedan utilizarlo como referencia para hacer alguna operación.

La idea de que sea una tienda de productos, en este caso de claves de juegos aunque podría valer cualquier producto, es muy útil, porque es un modelo de aplicación que nunca va a desaparecer y, de hecho, cada vez habrá más, por lo que me parece una muy buena idea desarrollar, aunque sea un proyecto completamente personal, una *Store*.

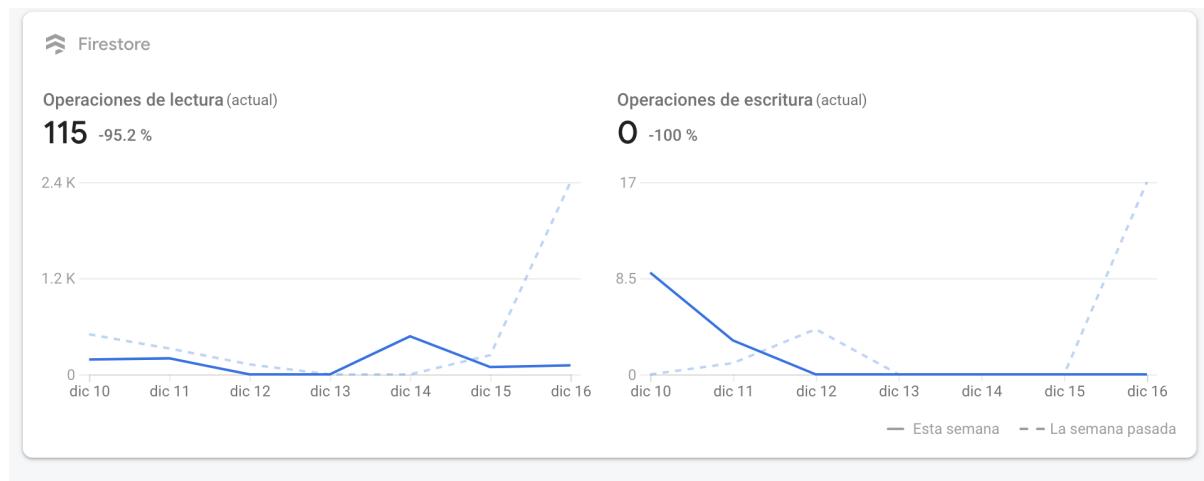
En resumen, estoy muy contento con el desarrollo de KeysStoreApp porque me ha despertado un interés y ambición muy grande por esta tecnología, Compose. Como he dicho antes, este proyecto no se quedará aquí, sino que seguirá creciendo poco a poco hasta ser un buen modelo estándar de lo que debería ser una aplicación de tienda de productos.



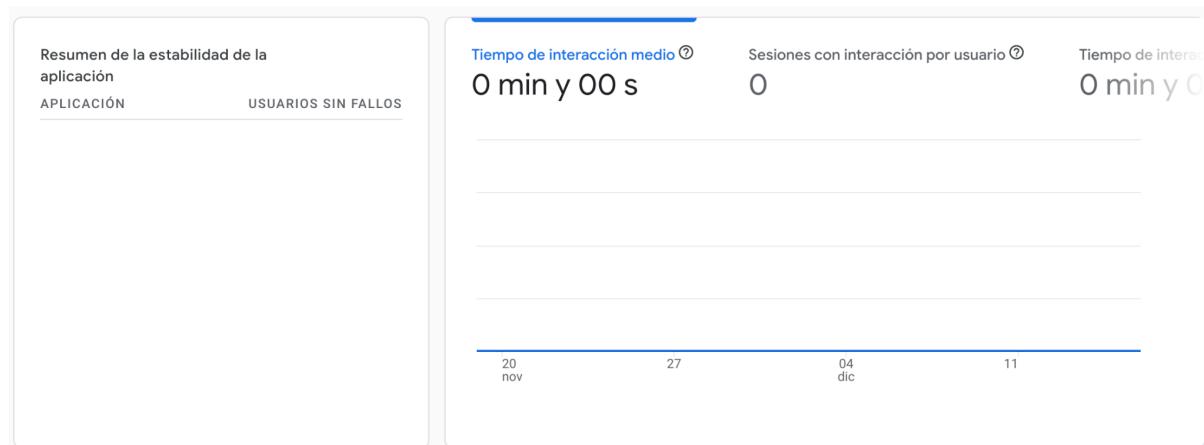
# Apéndices

El proyecto no cuenta con tests unitarios, que será algo que implementaré en futuras versiones ya que realizar tests a una aplicación es muy importante si quieras evitar errores en ejecución.

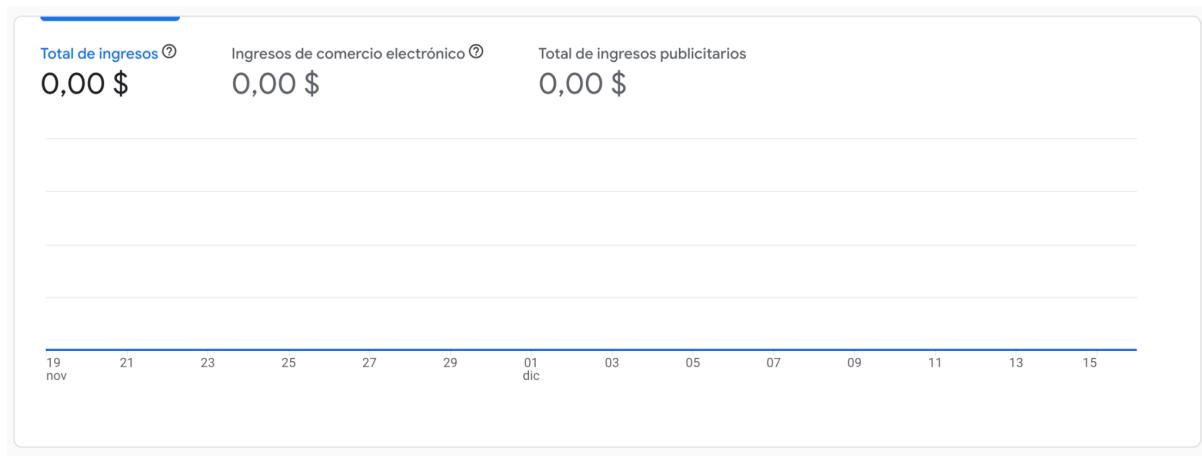
En cuanto a los informes, gracias a Firebase, se pueden obtener informes con información muy útil del uso de tu aplicación. Para obtener estos informes, hay que habilitar analytics en Firebase e implementarlo en la aplicación. Se pueden ver datos como cuantas operaciones de escritura y lectura se han hecho a lo largo de la semana.



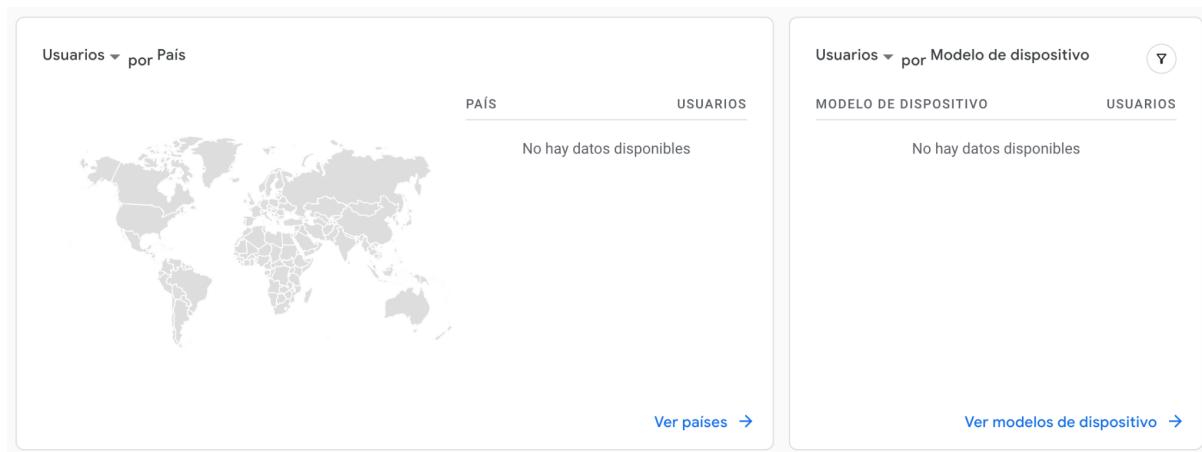
También se pueden ver el tiempo de sesión medio de un usuario en la aplicación. Estos datos están vacíos porque solo la he probado yo y en una versión debug de la aplicación, por lo que no se registran este tipo de datos.



Se puede ver la cantidad de ingresos que tenga la aplicación tanto por IAPs (compras en la app) como de publicidad. En este caso no hay ni compras reales en la app ni publicidad, por lo que nunca habrá ingresos.



Por último, de los datos más curiosos para mí, es el uso de la aplicación por país, modelo de dispositivo, versión de sistema operativo, etc. Para hacer estadísticas del uso de una aplicación real esta es una herramienta muy útil que ayuda al desarrollo continuo de la app.



# Bibliografía

- <https://stackoverflow.com/>
- <https://developer.android.com/>
- <https://developer.android.com/jetpack/compose/documentation>
- <https://kotlinlang.org/docs/home.html>
- <https://www.geeksforgeeks.org/>
- <https://www.linkedin.com/>
- <https://twitter.com/>
- <https://www.youtube.com/>
- <https://stackexchange.com/>
- <https://www.reddit.com/>
- <https://github.com/>
- <https://gitlab.com/>
- <https://issuetracker.google.com/issues>
- <https://firebase.google.com/>
- <https://m3.material.io/>
- <https://dagger.dev/hilt/>
- <https://coil-kt.github.io/coil/>