Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Closet Assistant, tu estilista de bolsillo: arquitectura, persistencia y lógica de negocio

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Álvaro Gómez Cuenca

Tutores: David de Andrés Martínez

Juan Carlos Ruiz García

Curso 2020/2021

Resumen

El tiempo es oro. Esta expresión que se ha usado a lo largo de muchísimos años se ha visto acentuada por la situación en la que nos encontramos hoy en día. Esta pandemia nos ha hecho darnos cuenta de que hay que dedicarles tiempo a las cosas que de verdad importan. Así que, con esa premisa en mente, mi compañero Marc Ferrandis López y yo hemos decidido sacar adelante este proyecto.

Por ello, hemos decidido desarrollar esta aplicación de Android con la idea de poder ahorrar tiempo, tanto en el día a día, como en momentos más puntuales. Este ahorro se obtiene gracias a los diferentes mecanismos implementados y la idea base de la aplicación. La función principal de la aplicación es el almacenamiento de las diferentes prendas de ropa del usuario y la ordenación de estas por conjuntos. Este hecho nos aporta un ahorro de tiempo en una gran variedad de situaciones: lo primero es que no es necesario que cada mañana pensemos que prendas de ropa nos vamos a poner y cómo debemos combinarlas, sino que simplemente miramos en nuestra lista de conjuntos cual nos cuadra más para ese día. Además, en la acción de crear estos conjuntos también nos ahorramos tiempo, ya que, al tener las prendas ya guardadas y clasificadas en la aplicación, en lugar de ir buscando por el armario y los diferentes cajones qué prendas nos gustan más para formar el conjunto, simplemente deberemos deslizar el dedo por las diferentes pantallas de la aplicación. Pero si no queremos gastar tiempo ni siquiera creando nosotros mismos los conjuntos no hay de qué preocuparse, ya que la aplicación se puede encargar de ello, creando el conjunto óptimo para ese momento en base al clima y los eventos que haya ese día en tu calendario. Esto proporciona como beneficio adicional que a la gente que no le interesa la moda o cómo vestir pueda ir siempre de forma impecable. Pero la aplicación no solo nos permite ahorrar tiempo con la ropa que ya tenemos, sino también con la que vamos a tener en un futuro. Esto lo conseguimos gracias a la opción de “Tiendas” implementada en la propia aplicación, donde encontraremos los principales portales de venta online de moda agrupados en una única ventana, lo que nos permite ver las prendas de forma simultánea sin tener que gastar tiempo abriéndolas de una en una en el navegador o yendo al centro comercial.

El proyecto se ha decidido llevar a cabo como TFG de emprendimiento en el marco de Start.inf, el espacio de emprendimiento de la ETSInf, ya que nos ofrecía una sensación muy parecida a trabajar en una StartUp y además podíamos dedicarnos a partes diferentes del proyecto mientras que a la vez trabajábamos en equipo. En mi caso me voy a encargar de lo que vendría a ser el back-end de la aplicación, es decir, tareas como diseñar, especificar e implementar toda la base de datos, partiendo del prototipado más básico hasta alcanzar la versión final del primer MVP, implementar la arquitectura diseñada para la aplicación o hacer pruebas de aceptación, ya que es necesario comprobar que todo lo integrado anteriormente es correcto.

Palabras clave: Back-End, Android, Aplicación Móvil, Clima, Ropa, Calendario, LiveData, ViewModel, View Binding, MVVM, Room.

Abstract

Time is precious. This expression has been used during many years, but due to the actual situation, now is more realistic. This pandemic has help us to realize that we need to dedicate time to the things that really care. With this premise in mind, my partner Marc Ferrandis López and my we have decided to do this project.

We decide to develop this Andoid application with the idea to save time, in daily situations or in more special moments. This save is achieve thanks to different mechanisms implemented and the basic idea of the application. The main functionality of the application is storage the different clothes of the users and organize this in outfits. This fact gave us the capability to save time in different situations: the first of them, we do not need to waste time every morning choosing what we need to wear and how to match it, because we only need to check the outfit list more appropriate for that day. In addition, creating this outfits we also save time, because is not necessary go to the closet, from the application when we have all our clothes, we can create the outfits selecting it. Also, if the user wants to save more time, they do not even need to choose an outfit, the application suggests three every day basing on the weather and the calendar events. And the last way to save time is using the "Shops" part of the application, because instead of going to the mall we can buy the clothes faster thanks to the selection of shops that we found inside the application.

This project is an entrepreneur project developed in the framework of Start.inf, the entrepreneur space of ETSInf, due to the similarities to work like a real StartUp. Also, work of of this way, gave us the possibility to work in different parts of the project. In my case I developed the back-end of the application which involves everything related to design, specification and implementation of the database, starting from the most basic prototyping until reaching the final version for the first MVP, implementation of the architecture designed for the application or make acceptance testing, because it is necessary to check if everything previously integrated is right.

Keywords: Back-End, Android, Mobile App, Weather, Clothes, Calendar, LiveData, ViewModel, View Binding, MVVM, Room.

Índice

Anexos

Índice de Ilustraciones

Índice de Tablas

Índice de Gráficas

Índice de Diagramas

Índice de Fragmentos de Código

Capítulo *1*

Introducción

1.1. Motivación

El ser humano cada vez vive más estresado, esto no es ningún secreto. Un estudio realizado ya en 2018 por laboratorios Cinfa (Cinfa, s.f.) y avalado por la Sociedad Española para el Estudio de la Ansiedad y el Estrés (SEAS) nos mostraba como resultados que "Nueve de cada diez personas en España han sentido estrés en el último año y cuatro de cada diez lo ha hecho de manera frecuente o continuada", y eso que estamos hablando de datos previos a la pandemia. Este mismo estudio revela que con un 50,9% la falta de tiempo y el exceso de actividad es una de las causas más frecuentes que provocan este sentimiento. Por ello decidimos crear esta aplicación, porque si conseguimos reducir, aunque sea en un porcentaje pequeño, esta dolencia que golpea tan fuerte a la sociedad y que la gente pueda dedicarle un poco más de tiempo a las cosas que realmente importan, el desarrollo del proyecto ya habrá valido la pena.

Pero ¿realmente podemos ahorrar tiempo simplemente facilitando el hecho de escoger la ropa? Desde luego. Un estudio realizado por Marks & Spencer (Cliff, 2016) nos mostró que las mujeres pasan una media de 17 minutos cada mañana escogiendo un conjunto y los hombres 13. Esto se traduce en 95,2 y 72,8 horas al año respectivamente. Pero no solo eso, sino que el hecho de invertir más tiempo del necesario en escoger tu conjunto también afecta a tu vida laboral, ya que demostró que 1 de cada 10 veces que se llega tarde al trabajo es debido al tiempo que la persona había pasado escogiendo la ropa que se iba a poner ese día. De hecho, hay personas muy poderosas, como Steve Jobs o Mark Zuckerberg, que, para ahorrarse esta inversión de tiempo, simplemente vestían o visten prácticamente igual todos los días como podemos apreciar en la Figura 1.

Figura 1. Steve Jobs vistiendo el mismo conjunto a lo largo de los años

Así que como consecuencia directa de este hecho y sumando que gran parte del día lo pasamos con nuestro smartphone a mano, ¿no es una evolución natural disponer de una aplicación en nuestro móvil con el listado de todas las prendas y que además nos recomiende cual ponernos en cada momento? Siguiendo esta filosofía nace Closet Assistant, para que ahorres tiempo y dinero y los dediques a lo que realmente importa.

1.2. Objetivos

El objetivo principal de este proyecto es que las personas puedan ahorrar tiempo en su día a día gracias a que la aplicación será capaz de escogerles el outfit idóneo en cada situación, además de disponer de un listado completo de todas las prendas de ropa registradas y clasificadas en la aplicación para cuando le sea necesario consultarlo. Con esto en mente, nos hemos marcado los siguientes objetivos:

1. Dar al usuario la posibilidad de añadir diferentes prendas de ropa y diferentes conjuntos a la aplicación y que estos objetos estén organizados por etiquetas para su fácil acceso.

2. Que la aplicación sea capaz de componer autónomamente y recomendar tres conjuntos diferentes de forma diaria en función del clima y los eventos del calendario.

3. Dotar a la aplicación con una sección de tiendas donde aparezcan las webs de las principales empresas de venta de ropa.

4. Desarrollar una pantalla que permita al usuario consultar el tiempo en cualquier ciudad por si se tiene que ir de viaje y en base a esa nueva información y los eventos del calendario la aplicación le recomiende dos conjuntos cada día.

5. Que el usuario pueda acceder a su listado de prendas y conjuntos tanto si está conectado a internet como si no lo está.

6. Permitir a los usuarios registrados hacer una copia de seguridad de sus prendas y conjuntos en la nube.

7. Desarrollar dos MVP que irán acompañados de su respectiva encuesta con la opinión de los usuarios.

1.3. Estructura de la memoria

La memoria está estructurada en diferentes capítulos en cada cual se enfatizará en una parte clave del proyecto. Los capítulos 1, 2, 3 y 4 se han realizado de forma conjunta.

En el segundo capítulo hablaremos sobre la idea de negocio y la evaluación de esta. Desarrollaremos de forma más extensa el concepto de la aplicación y también su viabilidad. Para ver si es viable la compararemos contra sus actuales competidores y también que modelo de negocio vamos a seguir y la proyección económica que puede llegar a alcanzar. Se utilizarán también herramientas como el Lean Canvas y el análisis DAFO, para que la visión que tengamos sea lo más próxima a la realidad posible.

En el tercer capítulo se desarrollará la idea de negocio partiendo del análisis de un mapa de características el cual servirá de base para establecer los dos MVP que se van a realizar y se analizará los resultados de las pruebas realizadas con los early adopters del primer MVP

En el cuarto capítulo se desarrollarán los casos de uso de la aplicación necesarios para desarrollar la aplicación mediante diagramas y tablas.

En el quinto capítulo se explica la diferencia entre SQL y no SQL, el diseño de la base de datos y las interfaces necesarias para que el diseño funcione.

En el sexto capítulo se mencionan todas las tecnologías utilizadas en el proyecto, haciendo hincapié en las bibliotecas de Android Jetpack.

En el séptimo capítulo se explican las distintas arquitecturas propuestas para el proyecto y como se han implementado las mismas junto a las tecnologías referenciadas en el capítulo sexto.

En el octavo capítulo se muestran las verificaciones que se han realizado a la base de datos y a los métodos que aceden directamente a ella. Adicionalmente se muestra el resultado de la ejecución de los test mencionados en este capítulo.

En el noveno capítulo se exponen algunos problemas que han ido surgiendo durante el desarrollo del proyecto y como se ha abarcado el problema.

En el décimo capítulo se muestran algunas refactorizaciones y buenas prácticas que se han realizado en el proyecto.

Por último, en el capítulo undécimo y duodécimo se habla de las conclusiones sacadas del proyecto y el trabajo futuro a realizar.

Capítulo 2

Evaluación de la idea de negocio

2.1. Descripción de la idea

Desarrollar una aplicación para que los usuarios puedan ahorrar tiempo y dinero en todas las acciones que involucren la ropa que tienen en su armario, desde escoger los conjuntos de su día a día hasta la compra de una prenda especifica en un momento determinado. Para ello se han implementado diferentes pestañas:

Una primera pestaña, llamada "Mi armario", en la que encontraremos tres conjuntos sugeridos por la aplicación en función del clima y los eventos que tengamos ese día en el calendario, para ahorrarnos tiempo a la hora de escoger el conjunto que nos vamos a poner y también podremos en esa misma pestaña consultar las diferentes prendas de ropa, clasificadas por su tipo de prenda.

En la siguiente pestaña, que es la de "Mis conjuntos", podremos ver los diferentes conjuntos que ya tenemos formados, ordenados por tipo de ocasión, para poder escoger de forma rápida.

En la pestaña "Tiendas", encontraremos las páginas web de las tiendas de moda más famosas, lo que nos permitirá comparar de forma rápida prendas similares para así escoger la que más nos convenza, permitiéndonos ahorrar dinero y tiempo.

Por último, encontraremos la pestaña "Planificación", en la cual podremos indicar un lugar y día concretos y la aplicación nos dirá el tiempo que va a hacer para que nos hagamos una idea y además nos sugerirá dos conjuntos para esa ocasión, siendo una opción muy útil si por ejemplo nos vamos a ir de viaje y estamos preparando la maleta.

2.2. Lean Canvas

Ya que hemos enfocado el proyecto como si fuese una StartUp, hemos escogido utilizar Lean Canvas como herramienta de visualización de modelos de negocio. Es una herramienta muy útil que combina dos metodologías ya existentes que son el Lean Startup y el Business Model Canvas, lo que la hace idónea para modelos de negocio ligeros como una StartUp que es nuestro caso. A continuación, encontraremos la Tabla 1 que contiene el Lean Canvas que hemos desarrollado para nuestro proyecto:

Tabla 1. Lean Canvas

<table>
<tr>
<td rowspan="2">PROBLEMA

Las personas pasan más de 80 horas al año viendo que se ponen por las mañanas

La gente no tiene conciencia real de toda la ropa que tiene en el armario, por lo que siguen comprando ropa, aunque sea parecida a la que ya tienen malgastando dinero sumando lo que contamina cada prenda nueva fabricada.

Este mismo desconocimiento hace que a veces nos pongamos una prenda una única vez y acabe en el olvido al fondo del armario.

SOLUCIONES EXISTENTES
La aplicación Smart Closet</td>
<td>SOLUCIÓN

Nuestra aplicación nos permite tener un listado de nuestras prendas y conjuntos separados por categorías para que nunca se nos vuelva a olvidar la existencia de una prenda.

Además, la aplicación nos sugerirá tres conjuntos cada mañana en base a el clima y los eventos del calendario para que no perdamos nuestro tiempo.</td>
<td rowspan="2">PROPOSICIÓN DE VALOR ÚNICA

Hacer que las personas ahorren tiempo en su día a día teniendo todas sus prendas y conjuntos en la palma de la mano para realizar una elección rápida y precisa de que ropa desean ponerse.</td>
<td>VENTAJA ESPECIAL

La capacidad de que la propia aplicación te recomiende conjuntos sin tener que gastar tu tiempo buscando que ponerte.</td>
<td rowspan="2">SEGMENTO DE CLIENTES

Cualquier persona que tenga unas nociones mínimas de tecnología y no quiera perder tiempo todas las mañanas pensando en que ponerse.

EARLY ADOPTERS
Personas entre 20 y 30 años acostumbrados a usar la tecnología para todo y quieran ganar tiempo en su día a día y/o estén interesados en la moda</td>
</tr>
<tr>
<td>MÉTRICAS CLAVE

Descargas de la aplicación.

Personas que se crean un usuario en la aplicación.

Prendas compradas desde la aplicación.

Tiempo que las personas mantienen instalada la aplicación.</td>
<td>CANALES

A través de las redes sociales y publicitándonos en aplicaciones y páginas web.</td>
</tr>
<tr>
<td colspan="3">ESTRUCTURA DE COSTES

Marketing necesario para dar a conocer las aplicación y formar una base de usuarios</td>
<td colspan="3">FLUJO DE INGRESOS

Obteniendo un porcentaje de la venta de cada producto desde la web.

Ingresos realizados por las tiendas de ropa para que su web aparezca de las primeras en la lista de tiendas.</td>
</tr>
</table>

2.3. Análisis DAFO

Para realizar el análisis estratégico de la empresa, hemos utilizado el análisis DAFO como herramienta. Es una técnica muy utilizada que nos ayudará en la toma de decisiones ya que nos permite realizar un análisis tanto interno como externo de la empresa. A continuación, encontramos la Tabla 2 con el análisis DAFO especifico de nuestro proyecto:

Tabla 2. Análisis DAFO

	INTERNAS	**EXTERNAS**
NEGATIVAS	DEBILIDADES Inexperiencia a la hora de funcionar como empresa y lanzar un producto al mercado El proyecto es la primera aplicación compleja que desarrollamos Recursos económicos limitados	AMENAZAS Se trata de una aplicación muy específica y puede ser muy difícil de que se generalice
POSITIVAS	FORTALEZAS Interfaz cuidada y de uso sencillo Conceptos innovadores como la recomendación automática de conjuntos Uso completo de la aplicación totalmente gratuito para los usuarios	OPORTUNIDADES Es un concepto que no se ha explotado nada, solo existe una aplicación similar Las personas cada vez quieren dedicarle su tiempo a las cosas que realmente importan Todo el mundo lleva su Smartphone siempre encima

2.4. Estudio de mercado

CLOSET

2.4.1. Smart Closet

Ilustración 1. Logo Smart Closet

La única aplicación que encontramos en el mercado con un enfoque similar al nuestro es Smart Closet, cuyo logo se puede ver en la Ilustración 1. Smart Closet es una aplicación gratuita que nos permite tener un listado de todas las prendas de ropa, como vemos en la Ilustración 2, que tenemos y formar conjuntos con ellas. Una vez con conjuntos creados, la aplicación nos permite asignarlos a los diferentes días que tenga un mes.

La aplicación añade un componente social y nos permite compartir nuestros conjuntos dentro de la aplicación, además de consultar los conjuntos que hayan creado y compartido otras personas que tengan la aplicación, función que podemos ver en la Ilustración 3.

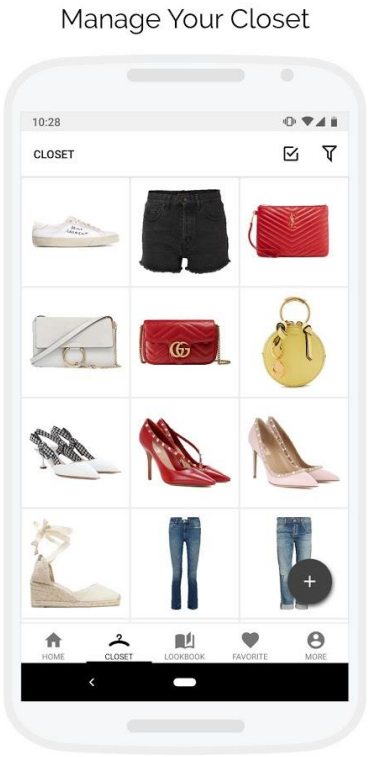


Ilustración 2. Captura de pantalla de closet de Smart Closet

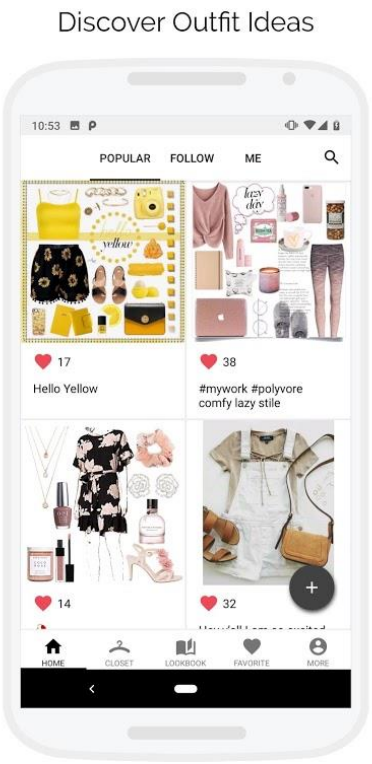


Ilustración 3. Captura de pantalla de inicio de Smart Closet

También dispone de una pestaña de tiendas, en la cual podemos filtrar por tienda, marca o tipo de prenda. Una vez con los filtros aplicados, nos aparecen distintas prendas de ropa con las que podemos interactuar para: añadirlas a nuestro armario o comprarlas en la web.

Podemos destacar también un par de funcionalidades que son: la posibilidad de crear un "Packing" donde podremos añadir diferentes conjuntos por si nos vamos de viaje y la opción de realizar una copia de seguridad de nuestras prendas y conjuntos en Google Drive.

Por último, hay que mencionar que dispone de una página web, que podemos ver en la Ilustración 4, que complementa la aplicación y nos permite ver nuestros conjuntos desde el ordenador.

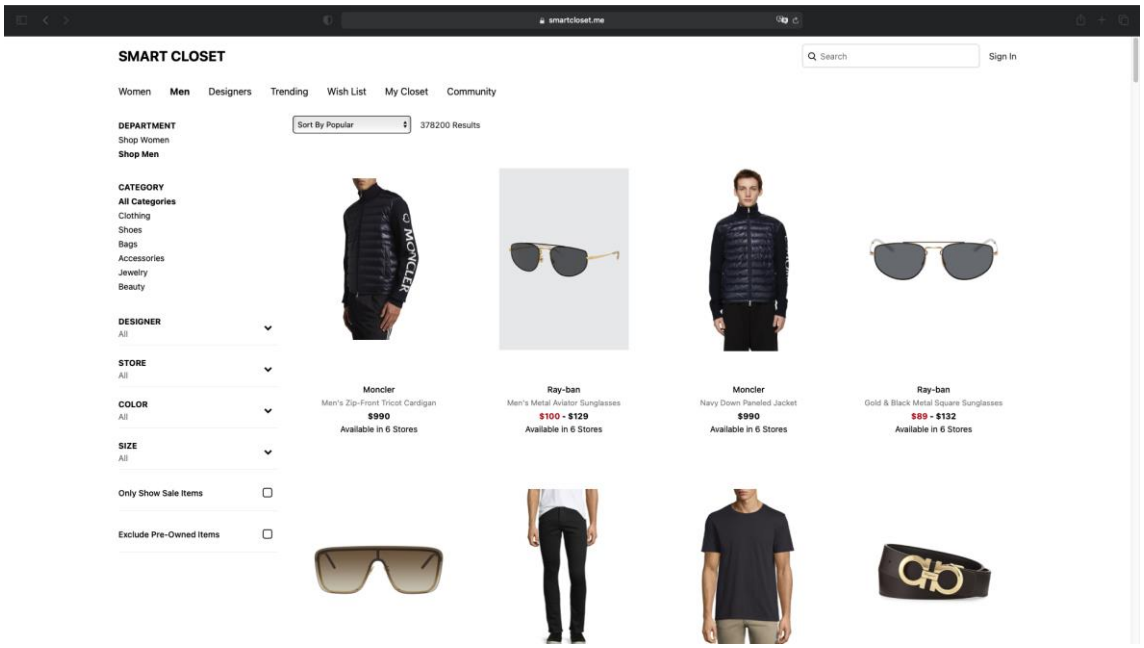


Ilustración 4. Captura de pantalla de la versión web de Smart Closet

2.4.2. Google Keep

Ilustración 5. Logo Google Keep

Keep es una aplicación gratuita desarrollada por Google que nos permite crear y administrar notas. A priori esta que es su funcionalidad principal no es competencia directa de nuestra aplicación. Pero debido a su interfaz básica pero cuidada, su disponibilidad multiplataforma, el gran desarrollador que tiene detrás y sobre todo que nos permite añadir imágenes a las diferentes notas y clasificarlas por etiquetas hace que mínimo valga la pena mencionarla.

Si que es cierto que como se ha mencionado, Keep no es una amenaza directa para nuestro proyecto ya que al fin y al cabo es simplemente una aplicación de notas, Ilustración 6 e Ilustración 7. Pero, si un usuario quisiese sí que es cierto que podría usarla para cubrir parte de la funcionalidad de nuestra aplicación , ya que podría realizar fotografías a sus prendas de ropa y ordenarla por etiquetas ya que a cada nota se le pueden asignar una o varias etiquetas que el usuario haya creado previamente, que es básicamente la opción de clasificado que tenemos nosotros en nuestro proyecto. Algo similar sucede con la opción de crear conjuntos. No es una opción que obviamente

incorpore Google Keep, pero si en lugar de añadir una imagen por nota como las prendas, añadimos más de una, en esencia tendríamos un conjunto, eso sí, hay que tener en cuenta que cada vez que queramos crear un conjunto nuevo deberíamos volver a añadir a la aplicación la imagen de la prenda que queramos agregar al conjunto. Por ello, además de la complejidad añadida frente a nuestra aplicación para realizar este tipo de acciones, la falta de características como planear el equipaje de un viaje o la recomendación de conjuntos nos hace darnos cuenta de que realmente no es un rival que batir, sino una alternativa que algún usuario puede usar para cubrir algunas de las características que posee nuestra aplicación, pero que por norma general no será así.

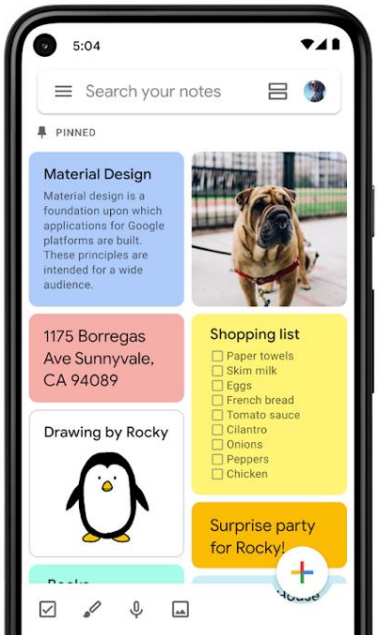


Ilustración 6. Captura de pantalla del menú de inicio de Google Keep

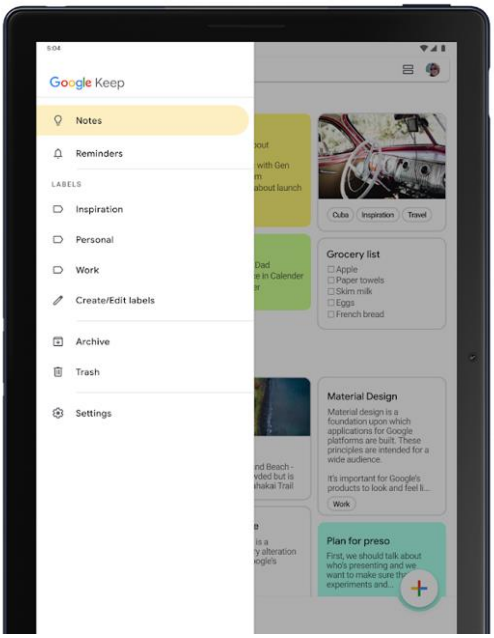


Ilustración 7. Captura de pantalla del menú desplegable de Google Keep

2.4.3. Notas con fotos - sencillo bloc de notas con fotos

Ilustración 8. Logo Notas con Fotos

Al igual que Google Keep, Notas con fotos se trata de una aplicación en la cual podemos añadir diferentes notas a las cuales les podemos agregar imágenes, por lo que podemos hacer el mismo "truco" de listar diferentes prendas de ropa o conjuntos añadiendo una foto de la prenda en cuestión y clasificándolo dentro de una carpeta o con etiquetas, Ilustración 9 e Ilustración 10. Pero si en mi opinión Google Keep no era un competidor, esta aplicación lo es menos, ya que estéticamente está menos cuidada que Keep, no tiene un desarrollador potente detrás y hay muchas funcionalidades como realizar una copia de seguridad en la nube u ordenar las notas que te obligan a realizar compras en la aplicación o adquirir el pack Premium.

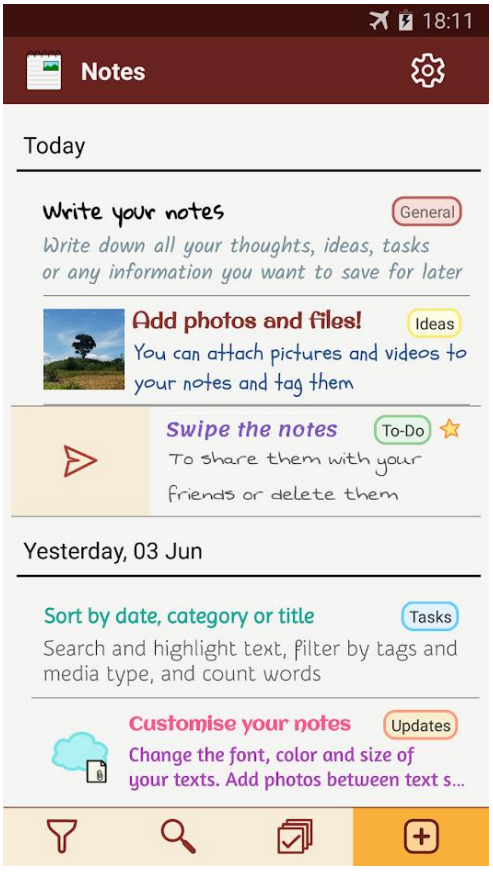


Ilustración 9. Captura de pantalla de Notas con Fotos

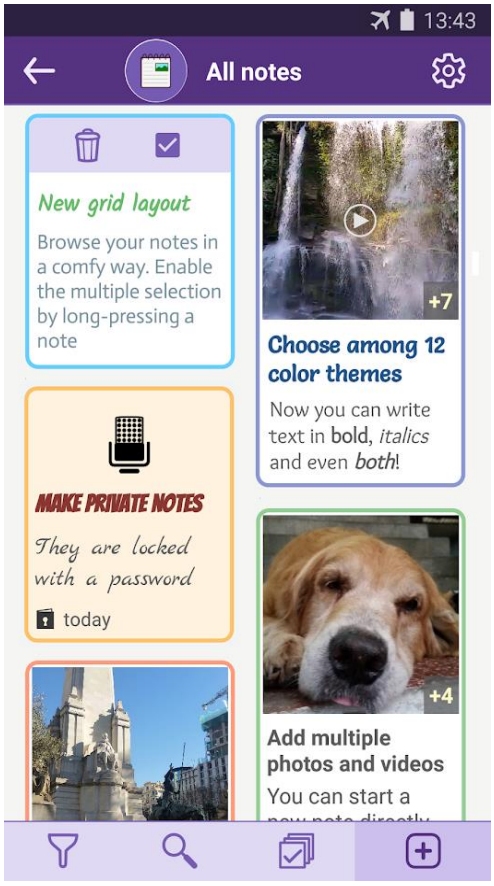
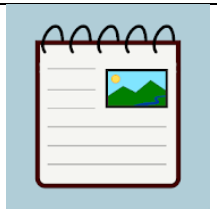


Ilustración 10. Otra captura de pantalla de Notas con Fotos

2.4.4. Comparación de las diferentes aplicaciones nombradas

A continuación, se ha añadido una tabla, Tabla 3, en la cual se va a listar las diferentes características que consideramos de mayor importancia en el ámbito en el que se ha desarrollado nuestra aplicación realizar una comparación de las tres aplicaciones nombradas anteriormente además de la nuestra.

Tabla 3. comparación de las diferentes aplicaciones similares

	Smart Closet	Google Keep	Notas con fotos	Closet Assistant
	CLOSET			
Crear un listado con nuestras prendas de ropa	✓	✓	✓	✓
Crear conjuntos en base a nuestras prendas de ropa	✓	✓	✓	✓
Filtrar por tipos de prendas	✓	✓	✓	✓
Filtrar los conjuntos por ocasión	✓	✓	✓	✓
Filtrar por temporada	✓	✓	✓	✓
Compartir conjunto dentro de la aplicación	✓	X	X	X
Compartir conjunto fuera de la aplicación	✓	X	X	✓
Recomendación automatizada de conjuntos en base al clima y los eventos del calendario	X	X	X	✓
Creación de conjuntos mediante inteligencia artificial	X	X	X	✓
Copia de seguridad de conjuntos y prendas en la nube	✓	✓	Solo Premium	✓
Web para ver los conjuntos desde el ordenador	✓	✓	X	X
Consultar el tiempo de un lugar y día concretos	X	X	X	✓
Recomendación de conjuntos a la hora de irte de viaje	X	X	X	✓
Crear paquetes de conjuntos	✓	X	Solo Premium	X
Diferentes tiendas dentro de la aplicación para comprar prendas	✓	X	X	✓
Poder añadir el precio que te ha costado a una prenda	✓	X	X	X
Interfaz cuidada y actual	X	✓	X	✓
Multilenguaje	X	✓	✓	✓
Precio de la aplicación	0€	0€	0€ - 7,99€	0€

2.4.5 Potenciales clientes

El principal perfil de cliente para el que hemos diseñado la aplicación es para hombres y mujeres de entre 15 y 40 años. Obviamente son edades orientativas ya que la aplicación no necesita de un gran conocimiento de las tecnologías ni ningún esfuerzo físico, pero hemos cogido estos dos extremos ya que por una parte los 15 es cuando empiezas a tener nociones de moda y una independencia total a la hora de vestir y como polo opuesto hemos escogido 35 años, ya que son personas que ya se han criado con tecnología y no tienen ningún problema en usarla e incorporar innovaciones en su día a día.

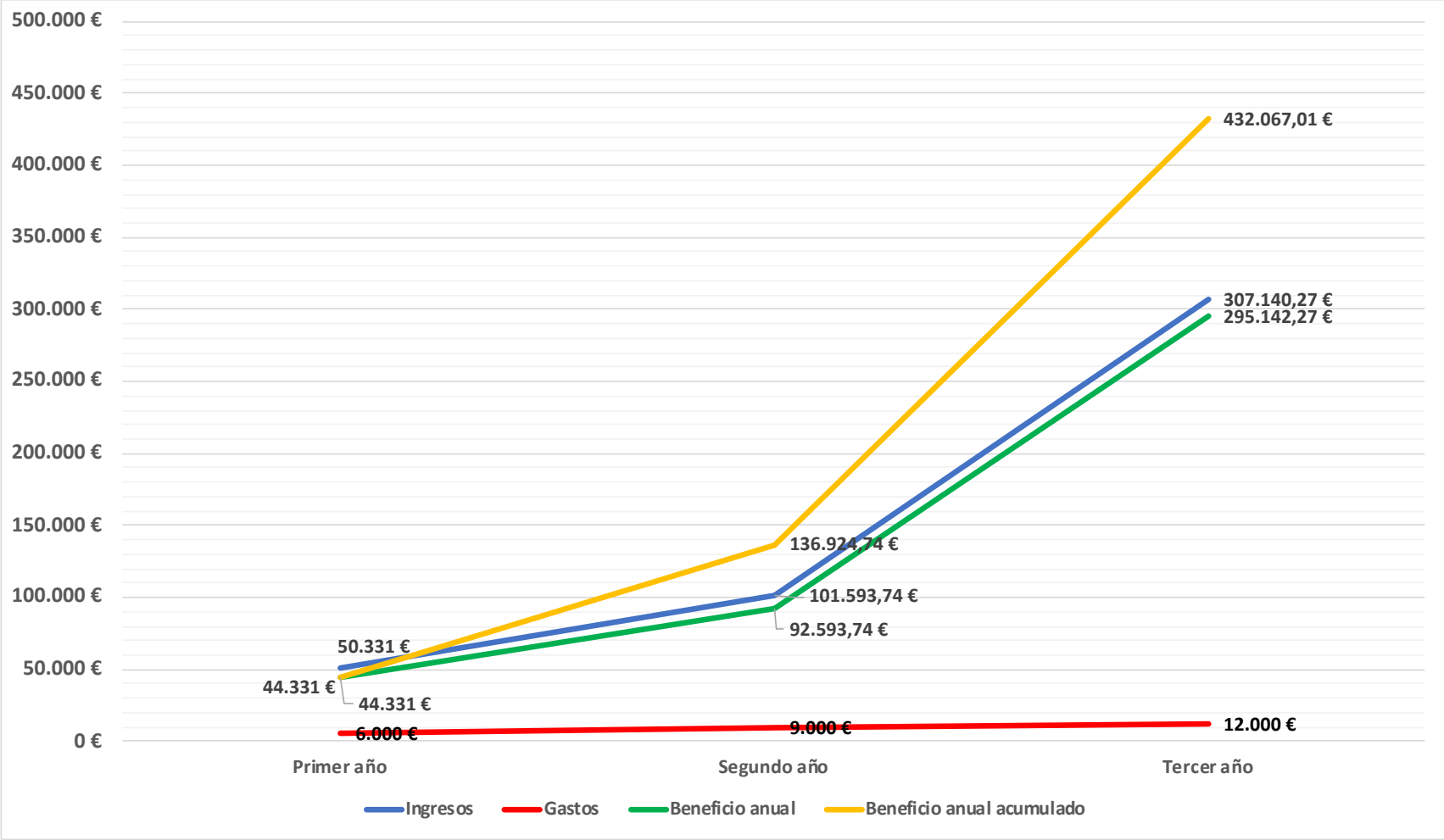
Pero como se ha comentado son perfiles meramente orientativos, cualquier persona puede usarla, pero si es verdad que hay altas posibilidades que una persona de unos 65 años no esté muy al día en las nuevas tecnologías u otra persona de 10 años no tenga conocimientos de combinación de ropa o ni siquiera Smartphone.

2.5 Modelo de negocio y proyección económica

La idea que teníamos clara a la hora de plantear el modelo de negocio es que debíamos de pensar una forma de que el dinero que ganásemos no saliera de los clientes, ya que empezar en un mercado tan grande como es el de aplicaciones para smartphones siendo un desarrollador nuevo y encima teniendo una aplicación de pago era una situación que se veía cuanto menos complicada. Así que con esta idea en mente hemos decidido dos fuentes de ingresos para la aplicación. El primero son los enlaces de referidos, esto es muy común en tema de promociones o ecomerce, se puede ver por ejemplo en canales de ofertas de Telegram. El sistema consiste en que si compras un producto en menos de X horas a través del enlace que se te proporciona, la persona o empresa que te ha proporcionado ese enlace se lleva una pequeña comisión del producto o productos adquiridos. El otro modo que tendríamos de obtener dinero sería que las tiendas estuviesen de forma predeterminada ordenadas alfabéticamente, pero si una empresa nos da una suma de dinero, su tienda aparecería mucho más adelante en la lista de tiendas. A más dinero, mejor posicionamiento frente a la competencia. Para tener una imagen más fiel de la proyección económica se ha realizado un análisis a 3 años vista, ya que es un tiempo suficiente para ver lo que sería una posible evolución. Para el cálculo de ingresos se han utilizado datos reales, como el porcentaje de beneficio mediante links de referidos o el precio que habría que pagar para posicionarse en Google, obviamente a menor escala ya que no podemos comparar posicionar una tienda en nuestra aplicación o en el buscador más famoso del mundo. A continuación, vamos a realizar una tabla, Tabla 4, y una gráfica, Gráfica 1, para tener de forma más visual la información:

Tabla 4. Ingresos - Gastos - Beneficios

	Primer año	Segundo año	Tercer año
Ingresos			
Numero de productos vendidos por enlaces de referidos	25000	50000	100000
Comisión media por enlace de referidos	10% de los ingresos	10% de los ingresos	10% de los ingresos
Dinero medio gastado por enlace de referidos	20€	20€	30€
Ingresos por enlace de referidos	50000	100000	300000
Tiendas participantes en el posicionamiento	3	10	22
Precio de posicionarse delante de otra tienda	10% más de lo que paga la tienda que se va a quedar detrás	10% más de lo que paga la tienda que se va a quedar detrás	10% más de lo que paga la tienda que se va a quedar detrás
Ingresos por posicionamiento	331€	1593,74€	7140,27€
Ingresos totales	50331€	101593,74€	307140,27€
Gastos			
Marketing	6000€	9000€	12000€
Beneficios			
Beneficio anual	44331€	92593,74€	295142,27€
Beneficio anual acumulado	44331€	136924,74€	432067,01€



Gráfica 1. Ingresos - Gastos - Beneficios

2.6. Conclusiones de la evaluación

Viendo los datos de la tabla y la gráfica obtenidos en el apartado anterior podemos obtener las siguientes conclusiones:

Lo primero que observamos como era de esperar es que el primer año es en el que se obtiene un menor beneficio. Esto es debido a que la aplicación todavía es joven, lo que hace que tanto los ingresos por enlaces de afiliados como sobre todo por el posicionamiento de las marcas sean relativamente comedidos. Esto sumado a el desembolso para la publicidad y el marketing digital, hace que el primer año, tal y como es lógico, tengamos los beneficios más bajos de esta hipótesis de ingresos, gastos y beneficios del proyecto. Como vemos después, este beneficio va aumentando año a año de forma exponencial, ya que el número de usuarios que usa la aplicación aumenta haciendo que aumente en función a esto el interés de las marcas en aparecer y posicionarse en la aplicación.

El segundo punto para destacar es que pese que el primer año los beneficios no son millonarios, hay beneficios, lo que hace el proyecto totalmente viable y resulta lógico llevarlo a cabo. Esto es así debido a que los gastos son muy bajos, ya que solo hay que invertir en marketing ya que toda la ejecución de la aplicación se realiza desde el propio teléfono o Tablet, sin necesidad de usar servidores externos y las copias de seguridad se suben al Google Drive de cada usuario. Todo este ahorro de infraestructuras es lo que nos permite tener tan buenos beneficios.

Capítulo 3

Desarrollo de la idea de negocio

Debido a que el proyecto se ha desarrollado en el contexto de start.inf como proyecto de emprendimiento, hemos decidido seguir como metodología de trabajo el proceso Lean Startup. Este método es muy conveniente para StartUps ya que al ir construyendo el producto en función al feedback de los usuarios a los que se les va pasando las diferentes versiones de la aplicación, se consigue reducir los riesgos y mejorar la aceptación, ya que en parte las características las han ayudado a escoger o afinar los propios usuarios debido a la naturaleza iterativa de esta metodología. Y para una empresa que parte desde cero, tener la posibilidad de reducir los peligros o incrementar la probabilidad de financiación, es una gran ayuda.

Para conseguir este valioso feedback se utilizan diferentes versiones parciales de la aplicación que van evolucionando llamadas Producto Mínimo Viable o MVP, que se les pasan a diferentes usuarios junto a una encuesta o algún otro método que nos ayude a obtener datos y métricas para saber primero de todo si el producto que estamos desarrollando va a ser utilizado por la gente y además saber si se está enfocando bien para saber qué cosas se deberían cambiar y cuales seguir de la misma forma.

Además, siguiendo con las metodologías de trabajo más actuales, para desarrollar estos nombrados MVP para nuestra aplicación hemos usado la conocida metodología de trabajo “scrum” ya que se trata de una metodología para desarrollo ágil, muy afín al Lean Canvas, que nos ha permitido dividir la faena en diferentes “sprints”. Para gestionar estos “sprints” y las diferentes tareas que hay en cada uno hemos utilizado la herramienta Worki, debido a su sencillo uso.

3.1. Mapa de características

Lo primero que hicimos fue un diagrama de características, donde encontraremos todas las características de la aplicación divididas en diferentes secciones y ordenadas por orden de prioridad para saber qué características había que implementar antes y cuales se podían dejar para más adelante.

Las diferentes secciones en las que se ha dividido el mapa de características son exactamente las mismas cuatro ventanas que encontramos en la aplicación, que son las siguientes:

- Mi Armario: se trata de la sección principal de la aplicación, en la cual se nos permite añadir las prendas de ropa y una vez añadidas, listarlas por tipo de prenda, eliminarlas o editarlas.

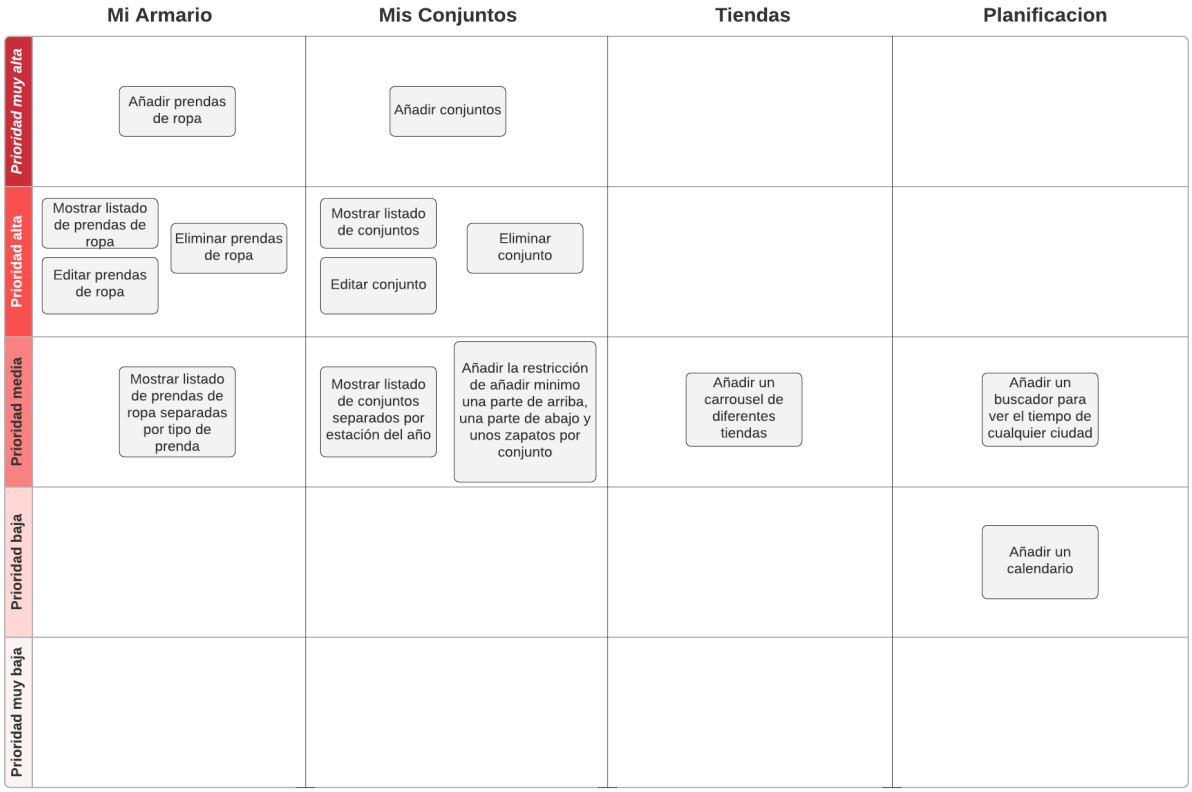
- Mis Conjuntos: esta sección es similar a la de Mi Armario, pero en lugar de añadir, modificar y eliminar prendas de ropa, lo que hacemos es añadir conjuntos partiendo de las prendas de ropa que ya tengamos y posteriormente poder modificarlos y eliminarlos.

- Tiendas: esta sección es muy sencilla, pero esencial. En esta parte de la aplicación encontraremos diferentes tiendas de ropa en las cuales podremos comprar prendas directamente sin salir de la aplicación, lo que añade un plus y además esta es la sección que nos puede servir para obtener beneficios, tanto llevándonos un porcentaje de cada producto vendido desde la aplicación como si las tiendas quieren pagar por posicionarse unas delante de otras.

- Planificación: la función actual de esta sección es mostrarnos un calendario para que veamos cómo está organizado el mes y además incluye un buscador para consultar el tiempo de cualquier ciudad del mundo.

A continuación, Tabla 5, encontraremos el mapa de características asociado a nuestra aplicación.

Tabla 5. Mapa de características de la aplicación



3.2. Desarrollo del primer MVP

Esta sección del capítulo la dividiremos en dos partes. En la primera haremos un resumen de las características que hemos implementado en esta primera versión de la aplicación y porque han sido esas las que hemos escogido. Y en la segunda parte analizaremos los resultados obtenidos de la encuesta que se les ha pasado a los primeros testers de la aplicación para de esta forma ver que tienes que mejorar o que nuevas características sería interesantes implementar para la siguiente versión.

3.2.1 Descripción del primer MVP

En este primer MVP queríamos por una parte implementar las funciones más características de esta aplicación que son la capacidad de almacenar todas tus prendas de ropa con sus diferentes etiquetas y que además puedas formar conjuntos con ellas y por otra parte tener una interfaz limpia y minimalista, ya que si una aplicación no te entra por los ojos o te parece fea es difícil que la sigas utilizando a no ser que sea esencial. Por ello se ha seguido el orden de prioridades que se muestra en la Tabla 5

Primero se implementó la sección de Mi Armario, ya que no tenía ningún sentido poder crear un conjunto si no se tenían prendas de ropa. Lo primero que implementamos fueron las operaciones CRUD. La prioridad máxima la teníamos a la hora de añadir una prenda a la base de datos con su imagen y etiquetas correspondientes. Una vez esta característica fue implementada ya nos pusimos con el resto de las acciones que podemos realizar con las prendas. Con las partes más críticas cubiertas, implementamos un filtro interno en la aplicación para que mostrara las prendas de ropa clasificadas por el tipo de prenda, lo que aporta más orden a la hora de visualizar las diferentes prendas además de facilitarnos la vida a la hora de añadir una prenda a un conjunto, ya que cuando vayamos a añadir una parte superior de un conjunto, solo se muestren las prendas con etiquetas correspondientes a partes superiores del cuerpo.

El siguiente bloque de características que implementamos fue el del apartado de Mis Conjuntos. La metodología seguida fue muy similar a el bloque de Mi Armario, implementar primero las CRUD priorizando la creación de conjuntos y luego mejoras menores. La primera mejora que añadimos fue la división de la lista de conjuntos en las diferentes estaciones del año y la segunda fue la restricción de no poder añadir un conjunto a no ser que se hubiese añadido una parte superior, una parte inferior y unos zapatos, ya que nos parecía que no tenía sentido tener un conjunto sin pantalones, por ejemplo.

Además de estas características, también se han implementado los cimientos de lo que sería un segundo MVP, que son las secciones de Tiendas y de Planificación. Aunque estas dos ventanas se han implementado de una forma muy básica, nos parecía interesante añadirlas ya en esta versión, para que los usuarios diesen su opinión para ver si les parecía útil o no y que opciones añadirían para completarlas. Aparte, la ventana de Tiendas queríamos implementarla lo antes posible, aunque solo fuese una aproximación, ya que es una parte de gran importancia para la aplicación porque es la que nos va a proporcionar ingresos y queríamos ver cómo era acogida entre los usuarios.

Por último, como se ha comentado en la introducción del capítulo, se ha utilizado la herramienta Worki para seccionar las tareas en diferentes unidades de trabajo o UTs y estas UTs las hemos englobado en 3 sprints, para así tener un mejor control de los tiempos y como debíamos ir avanzando. A continuación, podemos ver la Ilustración 11 con las UTs del primer sprint, la Ilustración 12 con las UTs del segundo y la Ilustración 13 con el tercer sprint.

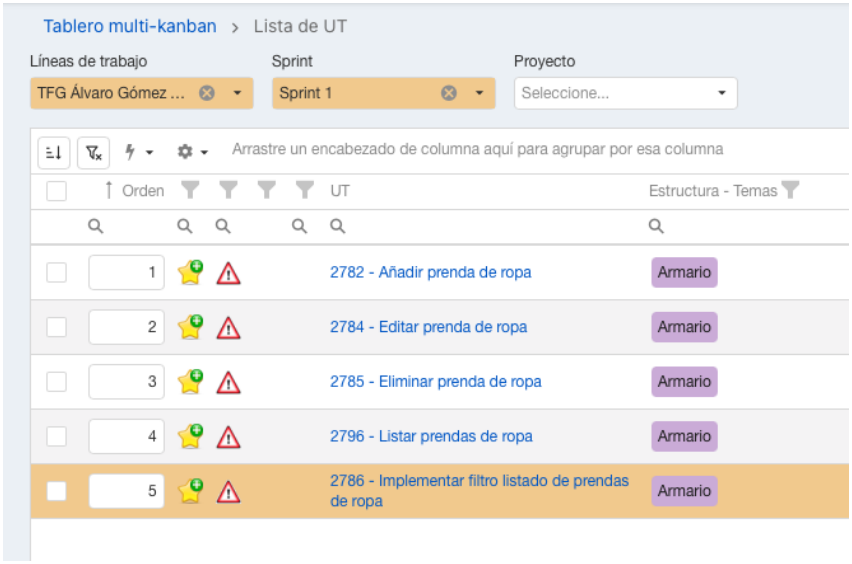


Ilustración 11. Primer sprint

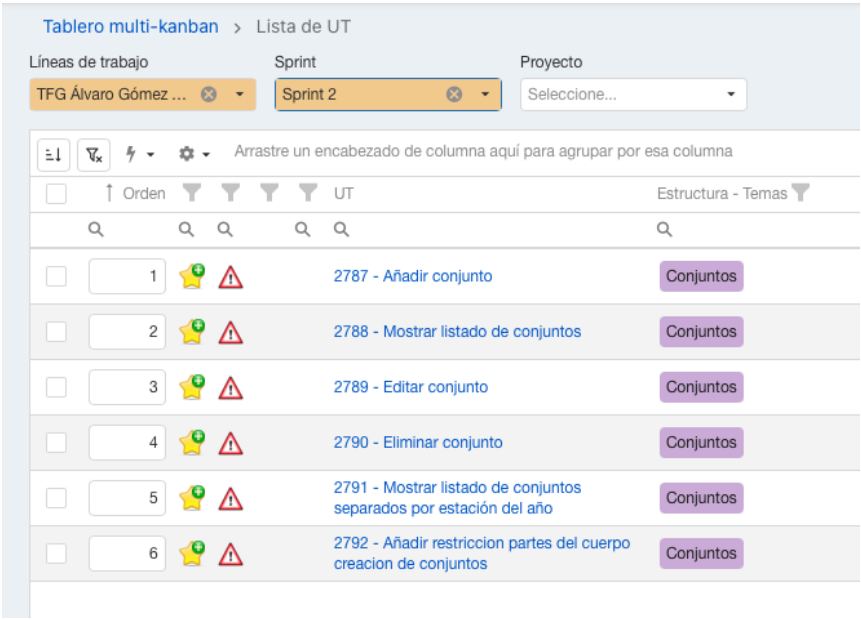


Ilustración 12. Segundo sprint

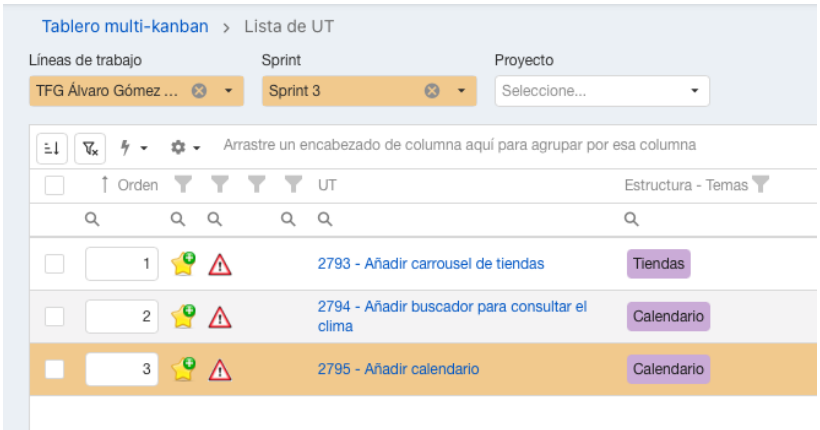
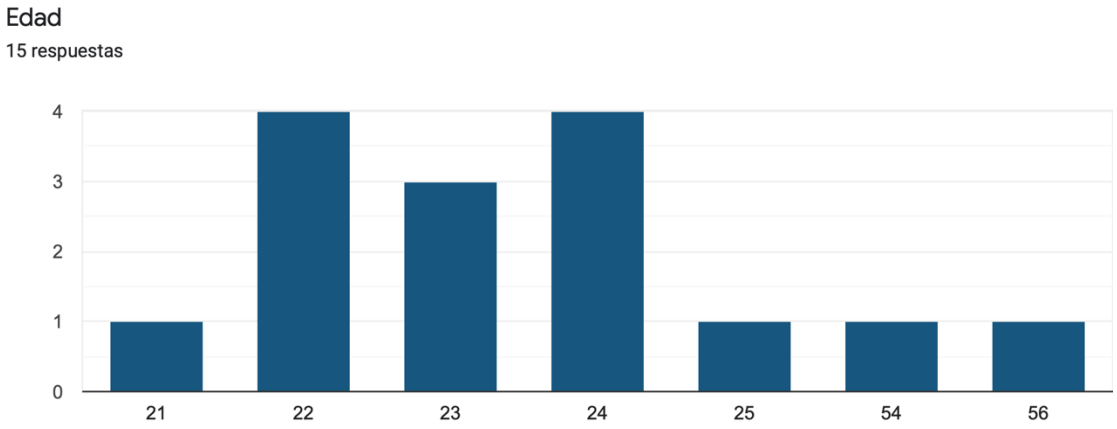


Ilustración 13. Tercer sprint

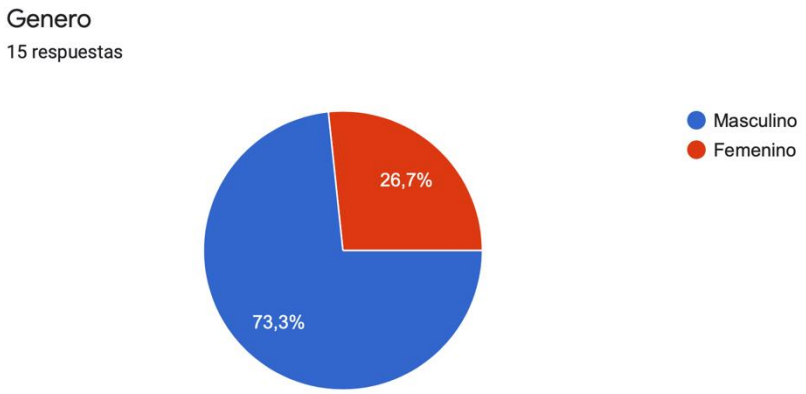
3.2.2 Análisis de las encuestas del primer MVP

En este apartado vamos a analizar las respuestas que nos han dado los 15 usuarios a los cuales se les ha pasado esta primera versión de la aplicación. Este análisis nos ayudará entre otras cosas a identificar fallos que no habíamos visto, implementar mejoras o enfocar algunas cosas desde otro punto de vista.



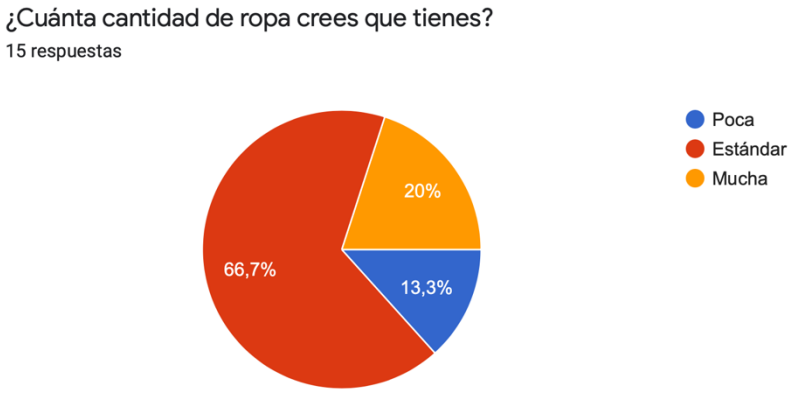
Gráfica 2. Gráfico sobre la edad de los usuarios.

La primera pregunta se trataba simplemente de responder la edad que tenían los usuarios. Se le decidió pasar la aplicación mayoritariamente a personas entre 20 y 25 años como podemos ver en la Gráfica 2 debido a que es la media de edad de nuestros conocidos y suponemos que será el rango de edad que más utilizará la aplicación. También se le paso la encuesta a dos personas de alrededor de 55 años para ver qué tan intuitiva les resultaba la aplicación.



Gráfica 3. Gráfico sobre el género de los usuarios

La otra pregunta personal realizada trataba sobre el género de la persona que estaba realizando la encuesta. Se la pasamos a un mayor porcentaje de hombres, factor que se ve reflejado en la Gráfica 3 que de mujeres ya que generalmente ellos se preocupan menos por la ropa así que supusimos que sería un público más difícil y si ellos lo verían útil ellas aún más.



Gráfica 4. Gráfico sobre la cantidad de ropa de los usuarios

Esta pregunta es la primera en la que realmente se trata el tema del proyecto en sí. Queríamos observar el volumen de ropa que tiene la mayoría de gente de estas edades, ya que, si su cantidad de ropa es poca, tener una aplicación para organizarla no acaba de ser del todo útil. Pero como vemos en la Gráfica 4 dos tercios de los encuestados consideran que tienen una cantidad estándar de ropa, eso sumado a las personas que dicen que tienen mucha nos da un porcentaje del 86,7%, así que consideramos que para esas personas si que les sería útil la aplicación.

Gráfica 5. Gráfico sobre la utilización de una aplicación de organización de ropa

En esta pregunta queríamos ver si los usuarios ya utilizaban una aplicación para organizarse la ropa o éramos nosotros los que estábamos creando esa necesidad. Como

vemos en la Gráfica 5 el dato es claro. Era una respuesta esperable también debido a la escasa oferta de este tipo de aplicaciones que hay en las tiendas.

Gráfica 6. Gráfico sobre la utilidad de la aplicación

Esta pregunta nos involucraba directamente ya que si el porcentaje de personas a las cuales no les parecía útil era muy alto, no tendría sentido seguir avanzando en el desarrollo. Pero como se puede ver en la Gráfica 6 a prácticamente tres cuartas partes de los usuarios les ha parecido útil.

Gráfica 7. Gráfico sobre el tiempo utilizado por las mañanas para escoger conjunto sin aplicación

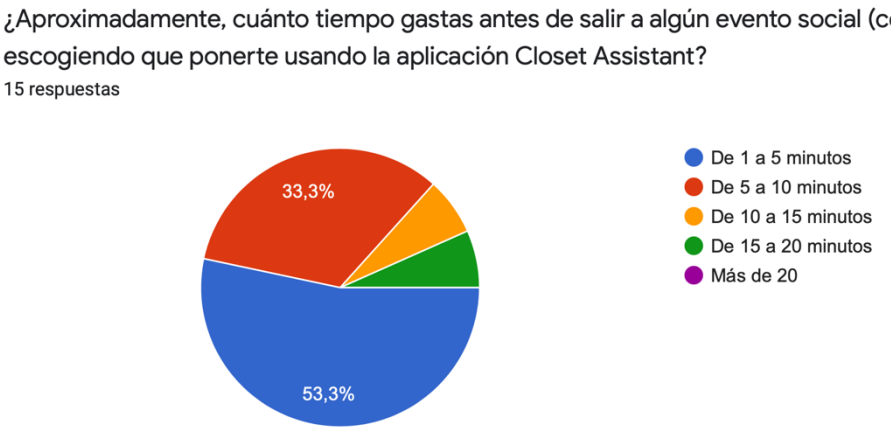
Comenzado ya con la parte de medida de tiempos, tenemos esta pregunta para ver cuánto tiempo tardan los usuarios de media a la hora de escoger que ropa ponerse por las mañanas. Como vemos en la Gráfica 7, los tiempos son bastante ajustados, así que la idea es que, con el uso de la aplicación, parte de las personas que tardan de 5 a 10 minutos baje a un tiempo de 1 a 5 minutos.

Gráfica 8. Gráfico sobre el tiempo utilizado antes de salir a un evento para escoger conjunto sin aplicación

Esta segunda pregunta de medida de tiempos como vemos en la Gráfica 8 nos arroja más variedad de información que la anterior Gráfica 7. Un 40% de la gente tarda entre 5 y 10 minutos y prácticamente la mitad entre 10 y 20, por lo que en este tipo de situaciones se espera que la aplicación ayude a los usuarios a reducir estos tiempos.

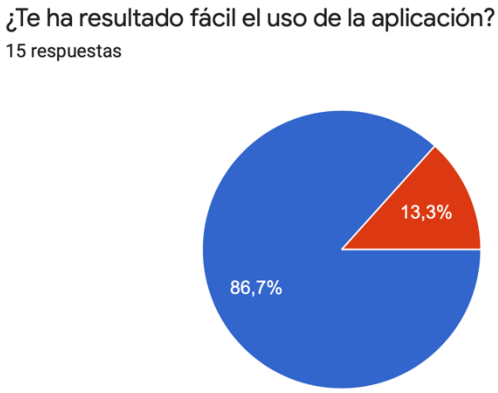
Gráfica 9. Gráfico sobre el tiempo utilizado por las mañanas para escoger conjunto con Closet Assistant

Como vemos en la Gráfica 9, la aplicación ha cumplido su cometido y hemos conseguido reducir los tiempos a la hora de escoger ropa por las mañanas incrementando un 20% las personas que han tardado de 1 a 5 minutos. Otro dato curioso que podemos observar es que ha habido una persona que le ha costado más que antes, echo que es comprensible ya que puede ser alguien que no se ha acabado de aclarar con la aplicación y tarda más usándola que sin usarla.

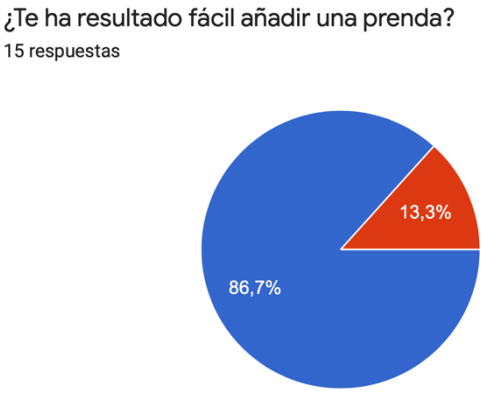


Gráfica 10. Gráfico sobre el tiempo utilizado antes de salir a un evento para escoger conjunto con Closset Assistant

Como vemos en este tipo de situaciones la aplicación también ha conseguido su objetivo y además con mejores resultados que en las situaciones diarias, evolución lógica ya que aquí había más margen de mejora. Si comparamos Gráfica 10 con la Gráfica 8 que reflejaban las mismas situaciones, pero la segunda sin el uso de la aplicación, vemos que la mejora es considerable ya que los usuarios que tardan de 1 a 5 minutos crecen prácticamente un 50% y casi todos los tiempos grandes han quedado absorbidos por el tiempo de 1 a 5 minutos.

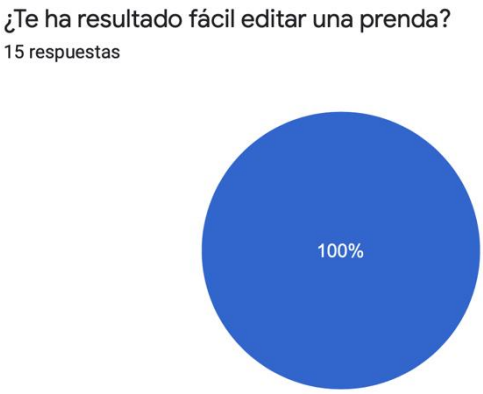


Gráfica 11. Gráfico sobre la dificultad de la app

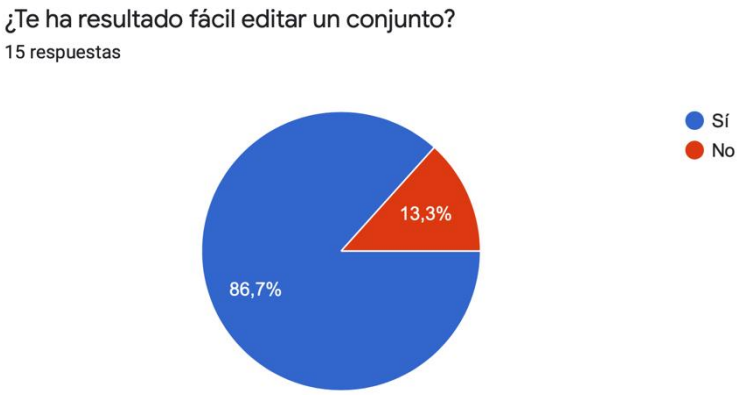


Gráfica 12. Gráfico sobre la dificultad de añadir una prenda

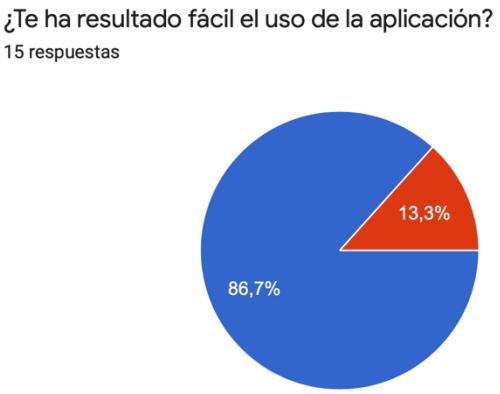
Gráfica 13. Gráfico sobre la dificultad de añadir un conjunto



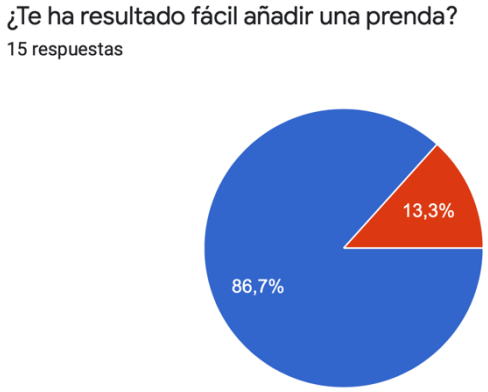
Gráfica 14. Gráfico sobre la dificultad de editar una prenda



Gráfica 15. Gráfico sobre la dificultad de editar un conjunto



Como podemos ver en la Gráfica **11**, la



Gráfica **12**, la Gráfica 13, la Gráfica 14 y la Gráfica 15 la aplicación ha resultado bastante fácil de usar para los usuarios. Solo dos personas han tenido problemas para averiguar cómo añadir una prenda o editar un conjunto y a nadie le ha costado ni editar una prenda o añadir un conjunto, así que podemos deducir que la aplicación es bastante intuitiva y que una vez aprendes a hacer una cosa como añadir una prenda puedes hacer sin problemas una similar como añadir un conjunto. La parte que parece que ha costado más, no solo por lo visto en la Gráfica 15 sino por los comentarios que veremos más adelante es la de modificar un conjunto y tiene sentido ya que es la menos intuitiva realmente así que una de las cosas que debemos cambiar para el siguiente MVP es hacer esa parte más visual.

Gráfica 16. Gráfico preferencia listar prendas

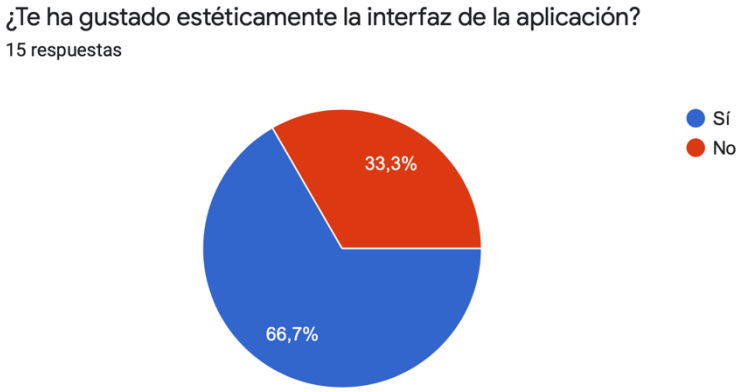
Como podemos ver en la Gráfica 16 a los usuarios no les ha acabado de convencer la forma en la que se muestran las prendas y preferirían que se mostraran por tipo de prenda como ahora, pero mostrando solo las de la época del año en la que nos encontramos. Así que a raíz de estos resultados hemos decidido dejar la visualización tal y como está, por si es verano y quieres ver alguna de tus prendas de invierno, pero que se pueda habilitar y deshabilitar un filtro por si solo te interesa ver las prendas de la época en la que te encuentras.

Gráfica 17. Gráfico preferencia listar conjuntos

Y de igual forma que pasaba en la Gráfica 16, como podemos ver en la Gráfica 17, a los usuarios no les convence la forma en la que están organizados los conjuntos y preferirían que se listaran por tipo de conjunto, pero mostrando solo los de la estación del año en la que nos encontramos. Para ello se hará lo mismo que en el apartado de armario, se listarán por tipos de conjunto y se le permitirá al usuario activar la opción de que solo se muestren los de la época del año actual.

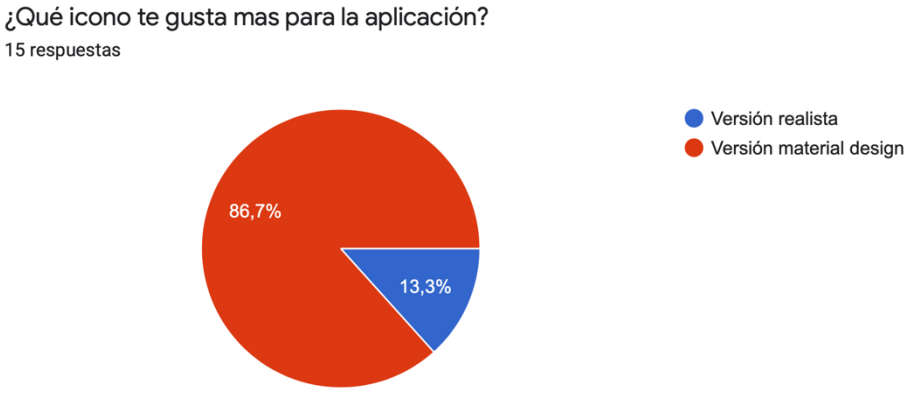
Gráfica 18. Gráfico que muestra si los usuarios consideran que falta algún tipo de prenda

En esta pregunta nos interesaba saber si nos habíamos dejado alguna etiqueta para la hora de clasificar las prendas, y hemos podido comprobar en la Gráfica 18 que sí, de hecho, casi la mitad de los usuarios opinan que faltan etiquetas. Algunas de las más repetidas son: pantalones cortos, vestidos, ropa interior, bañadores y zapatillas deportivas, además de hacer la categoría de accesorios menos genérica y dividirla en pulseras, collares, etc. Estas respuestas nos han sido de gran ayuda ya que es una característica fácil de mejorar y que contentará a los usuarios.



Gráfica 19. Gráfico sobre la opinión estética de la aplicación

En esta parte no hay mucho que comentar, simplemente que como vemos en la Gráfica 19 a dos terceras partes de los usuarios les ha gustado estéticamente la aplicación, lo que me parece un buen resultado ya que es algo muy subjetivo y aún faltan aspectos por pulir.

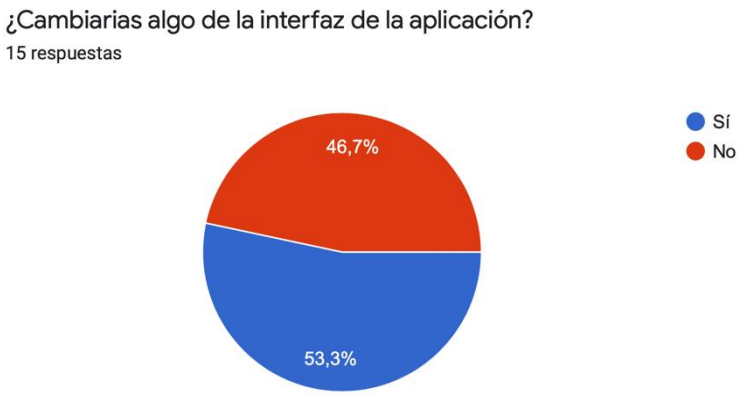


Gráfica 20. Gráfico sobre la comparativa de iconos

Ilustración 15. Icono estilo realista

Ilustración 14 Icono estilo material design

En esta parte se preguntaba a los usuarios sobre que icono de la aplicación preferían, ya que es una forma importante de identificar la aplicación, si una versión más realista, que podemos ver en la Ilustración 15 o una versión estilo material design Ilustración 14. Y como vemos en la Gráfica 20 la respuesta ha sido bastante unánime, escogiendo la versión material design más del 85% de los usuarios.



Gráfica 21. Gráfico sobre cambios en la interfaz

Continuando con el apartado visual, le preguntamos a los usuarios si cambiarían algo de la interfaz y como se ve en la Gráfica 21, más de la mitad lo harían. Casi todas las respuestas van en la misma línea: hacerla más atractiva visualmente con colores más vivos y un toque más moderno. Es difícil hacer una interfaz que encante a todos, pero intentaremos darle una vuelta de tuerca para que a la gente le resulte más atractiva.

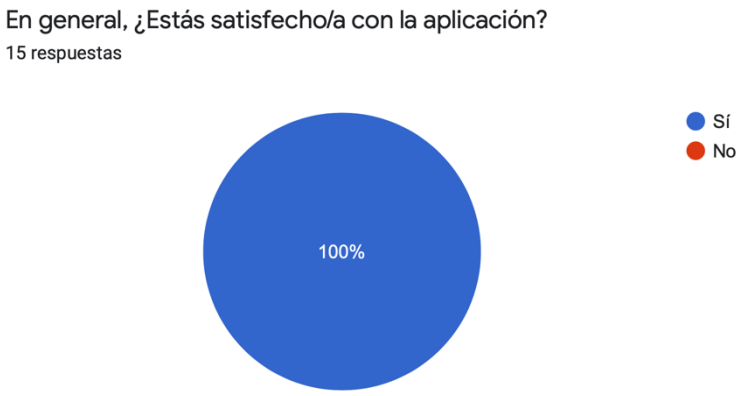
Gráfica 22. Gráfico sobre fallos en la aplicación

Sabíamos que era muy probable que la aplicación diese algún fallo debido al estar en una primera versión, de hecho, a nosotros nos había pasado alguna vez también testeándola, y como puede verse en la Gráfica 22 solo un tercio de los usuarios no han tenido ningún problema. Los tres comentarios de fallos más repetidos son los mismos que habíamos identificado nosotros:

- Al guardar una prenda la aplicación se cierra inesperadamente

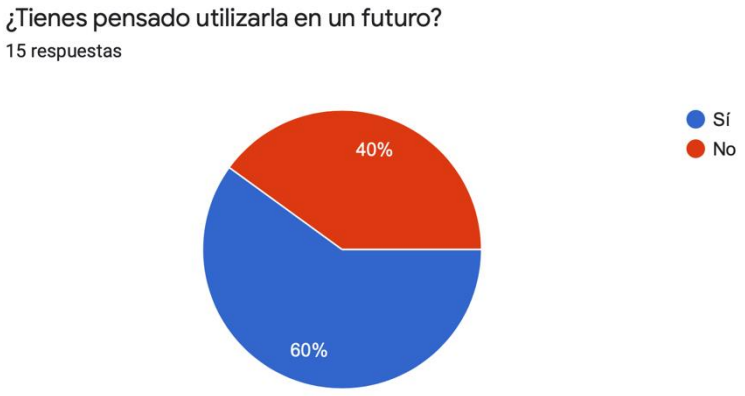
- La aplicación muestra una ventana de que no hay prendas, pero sí que se han añadido

- Algunas tiendas del apartado de Tiendas no cargan las imágenes, además de que se cambia de tienda sin querer debido a que la sensibilidad está muy alta.



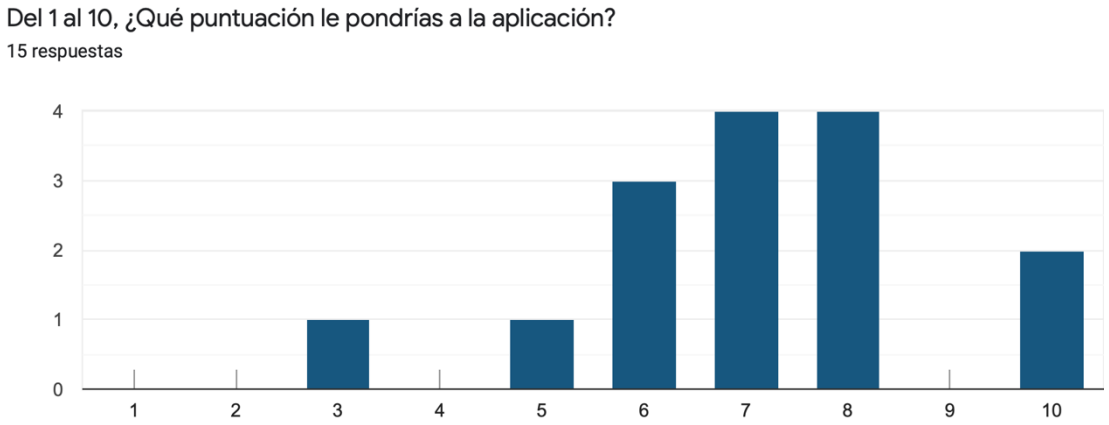
Gráfica 23. Gráfico sobre la satisfacción con la aplicación

Cuando le preguntamos a los usuarios si estaban satisfechos con la aplicación, como vemos en la Gráfica 23 la respuesta ha sido muy buena. Los 15 usuarios están satisfechos con el funcionamiento, suponemos que han asumido como era una primera versión era normal que a veces diese fallos o hubiese partes por pulir.



Gráfica 24. Gráfico sobre la futura utilización de la aplicación

Esta pregunta era muy importante para nosotros también, ya que veremos de los usuarios que han probado la aplicación, cuales realmente la usarían en su día a día, y para nuestra satisfacción, como vemos en la Gráfica 24, un 60% la seguiría utilizando. También podemos sacar como información interesante que toda la gente que no la utilizaría sería por lo mismo: considera que tiene un volumen de ropa tan bajo que la aplicación no le aportaría nada, de hecho, algunos dicen que le restaría más que aportaría.



Gráfica 25. Valoración numérica de los usuarios sobre la aplicación

Decidimos pedirles a los usuarios que nos diesen una valoración final de la aplicación y la verdad es que como vemos en la Gráfica 25 las opiniones son generalmente buenas, más teniendo en cuenta que se trata de una primera versión, tener una media de aproximadamente un 7 nos parece un resultado muy bueno.

Por último, le preguntamos a los early adopters si cambiarían o añadirían algo a la aplicación y la respuesta es prácticamente unánime: hacerla más intuitiva. Algunos sugieren hacer cambios en el botón de añadir prenda u otros a la hora de editar conjuntos. Intentaremos en la próxima versión hacer que estas partes sean más visuales añadiendo iconos o haciendo los botones más grandes.

Capítulo 4

Casos de uso

En este capítulo vamos a representar y comentar los diferentes casos de uso asociados a nuestra aplicación. Hemos decidido utilizar esta técnica ya que nos permite identificar desde antes de comenzar a desarrollar la aplicación las diferentes acciones que va a realizar el usuario y la acción o respuesta que va a realizar en sistema y como se va a comportar.

La representación de los casos de uso se compone de cuatro elementos: los casos de uso, los actores, la comunicación y el entorno del sistema. Los casos de uso son las acciones que va a poder realizar el usuario. Por otro lado, tenemos a los actores externos, que reflejan a los usuarios que interactúan con el sistema. Una vez con los actores y casos de uso identificados estos deben estar conectados mediante líneas de comunicación, donde además especificamos que tipo de comunicación hay entre ellos. Y por último encontramos el entorno del sistema, que identifica el entorno en el que se van a realizar las acciones.

A continuación, encontraremos la representación de los diferentes casos de uso que hemos redactado para nuestra aplicación. Hemos decidido dividirlos en cuatro bloques para que su representación quede más clara.

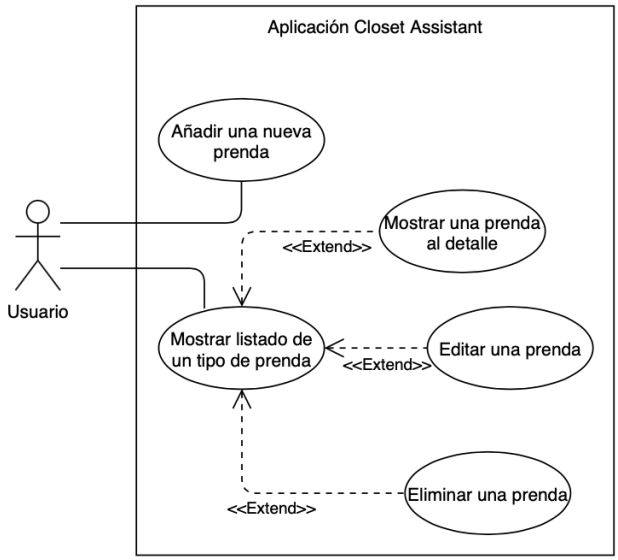


Diagrama 1. Casos de uso de la sección Mi Armario

En el Diagrama 1 encontramos todos los casos de uso relacionados con la creación y modificación de prendas, lo que vendría a ser la parte de Mi Armario. Como podemos ver tenemos un caso de uso que es independiente que se trata de “Añadir prenda”. Pero una vez tenemos una o más prendas de un tipo pasamos al caso de uso de “Mostrar listado de un tipo de prenda”, la cual es necesaria para el resto de casos que encontramos, ya que para mostrar, editar y eliminar una prenda primero ha de ser seleccionada desde una lista de prendas que la aplicación va creando de forma dinámica según se van añadiendo nuevas. Además, como podemos ver en la representación, estas actividades las puede realizar cualquier tipo de usuario, esté autenticado o no, ya que el actor representa un usuario genérico.

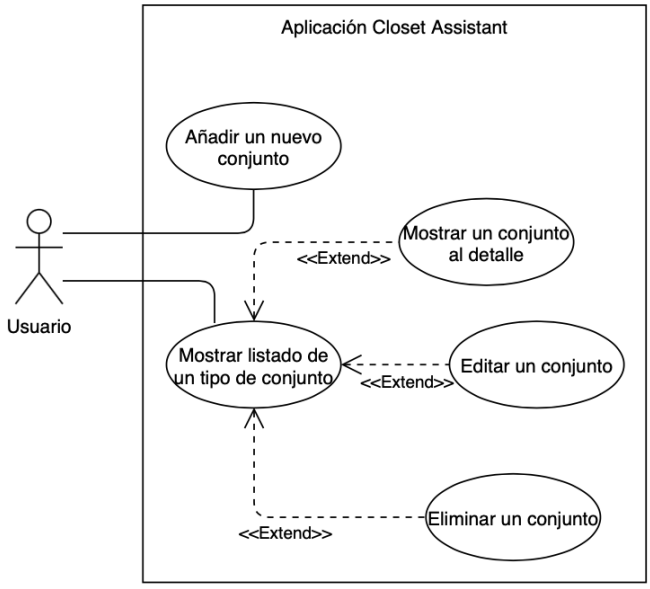


Diagrama 2. Casos de uso de la sección Mis Conjuntos

Como podemos ver, el Diagrama 2 es una representación muy similar en el diagrama anterior, Diagrama 1. Esto es lógico ya que encontrábamos representadas los casos de uso relacionados con las operaciones CRUD (Create, Read, Update, Delete) de los objetos de tipo “Prenda de ropa” y aquí encontramos los casos de uso relacionados con las CRUD de los objetos tipo “Conjunto”, por lo que el comportamiento es muy similar. Por un lado, tenemos el caso de uso de “Añadir un nuevo conjunto” y por otro lado mostrar, editar y eliminar un conjunto, que cuelgan del listado de conjuntos. E igual que en la Diagrama 1, también vemos que todas estas actividades las pueden realizar tanto usuarios autenticados como usuarios que no hayan iniciado sesión.

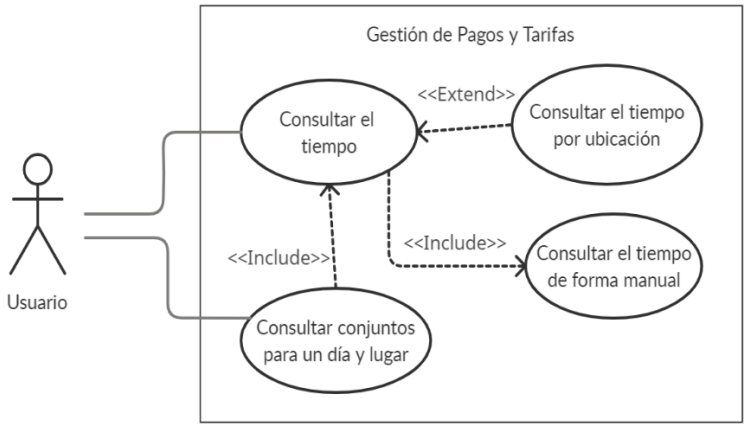


Diagrama 3. Casos de uso relacionados con la sección de planificación

Esta representación, Diagrama 3, ya es diferente a las anteriores. Por un lado, tenemos la consulta del tiempo que se puede realizar de dos maneras, cada una con un caso de uso relacionado. Tenemos la forma manual, que de hecho es necesaria hacerla si queremos que esta parte de la aplicación funcione y en la cual escribiremos nosotros de forma manual la ciudad que queremos que la aplicación use para la recomendación de conjuntos por si no queremos darle permisos a la aplicación para que conozca nuestra ubicación, como backup por si el servicio de ubicación falla o por si simplemente queremos que la aplicación use otra ciudad que no es la que nos encontramos por si queremos ver los conjuntos que debemos llevarnos para un viaje. La consulta del tiempo también se pude realizar de forma automática como vemos en el caso de uso "Consultar el tiempo por ubicación", de esta forma la aplicación utilizará la ubicación en la que nos encontremos en ese momento para las diferentes recomendaciones. El otro caso de uso, "Consultar conjuntos para un día y lugar", necesita del caso de uso "Consultar el tiempo" comentado anteriormente, debido a que en función a la información que este último le dé, la aplicación nos recomendará un par de conjuntos adecuados a esta información. Además, podremos movernos por el calendario para que los conjuntos cambien según cambia el tiempo de los diferentes días. Y de igual forma que los casos de uso anteriores, para la realización de estas actividades, la aplicación no distingue entre usuarios autenticados o no autenticados, ya que no queremos que los registros sean un impedimento para el uso de nuestra aplicación.

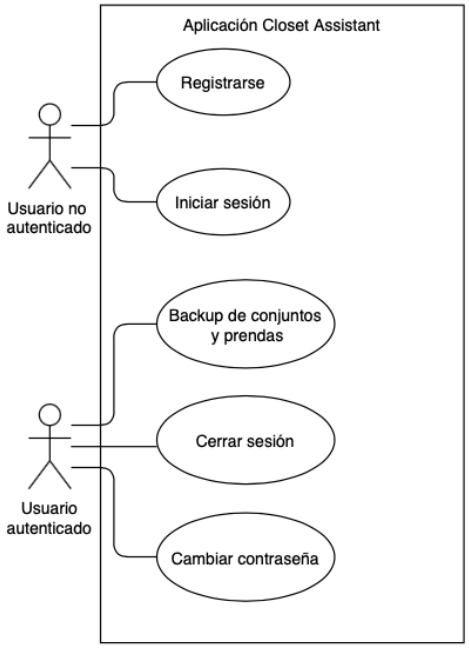


Diagrama 4. Casos de uso relacionados con las diferencias entre tipos de usuarios

Por último, encontramos la única representación que sí que distingue entre usuarios autenticados o no, que podemos ver en la Diagrama 4 que es la sección de los casos de uso relacionada con los perfiles de usuario. En el apartado de configuración, un usuario que no esté registrado solo podrá registrarse o iniciar sesión. En cambio, un usuario registrado, aparte de cerrar sesión o cambiar la contraseña, podrá acceder a una función importante de la aplicación y clave si se tienen varios dispositivos, que es realizar una copia de seguridad en la nube de las diferentes prendas y conjuntos, para no tener que empezar de cero si cambiamos de móvil o si utilizamos un móvil y una tablet por ejemplo.

Para terminar de ilustrar esta parte, a continuación se adjuntan las diferentes tablas de los casos de uso de las secciones de prendas de ropa: Tabla 6, Tabla 7, Tabla 8, Tabla 9 y Tabla 10. Las tablas referentes a consulta del tiempo y recomendación de conjuntos son: Tabla 11, Tabla 12 y Tabla 13. No se han decidido adjuntar las tablas de las secciones de Conjuntos y Perfiles de usuarios ya que la primera es muy similar a Prendas de ropa y la segunda es bastante genérica, pero de todas formas pueden consultarse en el apartado de anexos de la memoria.

Tabla 6. Tabla de caso de uso listar prendas

Caso de uso	Mostrar listado de un tipo de prenda		
Descripción	Se muestra un scroll vertical infinito con todas las prendas de ese tipo		
Actores	Usuario		
Precondición	El usuario debe de haber guardado mínimo una prenda de ese tipo		
Escenario Principal	Paso	Acción	
	1	El usuario accede a la ventana de Mi armario	
	2	Hace scroll hacia la derecha hasta llegar a la sección del tipo de prenda que desee	
	3	El sistema muestra todas las prendas de ese tipo registradas del usuario con un scroll vertical infinito	
Excepciones	Paso	Acción	
	3	El usuario no ha añadido ninguna prenda de ese tipo	
		E.1	No se muestra la sección de ese tipo de prendas

Tabla 7. Tabla de caso de uso añadir prenda

Caso de uso	Añadir nueva prenda	
Descripción	Se añade una nueva prenda a la base de datos de la aplicación	
Actores	Usuario	
Precondición		
Escenario Principal	Paso	Acción
	1	El usuario accede a la ventana de Mi armario
	2	Pulsa en el botón de añadir que se encuentra en la ventana
	3	Se abre la cámara del dispositivo
	4	El usuario realiza una fotografía
	5	El usuario decide si le gusta la fotografía que ha tomado o desea repetirla
	6	Una vez con la fotografía confirmada, elimina las partes de la imagen que no formen parte de la prenda
	7	El usuario confirma que ya ha eliminado todo el sobrante
	8	El usuario selecciona las etiquetas de la prenda y cuando termine le da a guardar
	9	El sistema almacena la nueva prenda en la base de datos
Postcondición	La prenda queda almacenada en la base de datos de la aplicación con sus etiquetas correspondientes	

Tabla 8. Tabla de caso de uso ver prenda en detalle

Caso de uso	Mostrar una prenda al detalle	
Descripción	Se muestra una nueva ventana con todas las etiquetas de una prenda	
Actores	Usuario	
Precondición		
Escenario Principal	Paso	Acción
	1	El usuario accede a la ventana de Mi armario
	2	Pulsa sobre una prenda
	3	El sistema muestra todas las etiquetas que tiene una prenda

Tabla 9. Tabla caso de uso eliminar prenda

Caso de uso	Eliminar una prenda		
Descripción	Se elimina una prenda de la base de datos		
Actores	Usuario		
Precondición	Que exista una prenda registrada		
Escenario Principal	Paso	Acción	
	1	El usuario accede a la ventana de Mi armario	
	2	Pulsa sobre la prenda que desee	
	3	Pulsa el botón eliminar	
	4	El sistema muestra una ventana de confirmación	
	5	El usuario afirma	
	6	El sistema elimina la prenda y todos los conjuntos que la contengan de la base de datos	
Postcondición	La prenda y todos los conjuntos que contengan esa prenda son eliminados de la base de datos		
Excepciones	Paso	Acción	
	6	La prenda es del tipo accesorio	
		E.1	No se eliminan los conjuntos a los que pertenece esa prenda

Tabla 10. Tabla de caso de uso editar prenda

Caso de uso	Editar una prenda		
Descripción	Se editan las etiquetas de una prenda		
Actores	Usuario		
Precondición	Que exista una prenda registrada		
Escenario Principal	Paso	Acción	
	1	El usuario accede a la ventana de Mi armario	
	2	Pulsa sobre la prenda que desee	
	3	Pulsa el botón editar	
	4	El sistema le permite modificar las etiquetas de la prenda	
	5	Una vez haya terminado, usuario confirma los cambios	
	6	El sistema sobrescribe la prenda de la base de datos con las nuevas etiquetas	
Postcondición	Se actualizan los datos de la prenda en la base de datos		
Excepciones	Paso	Acción	
	5	Debe haber mínimo una etiqueta seleccionada por categoría	
		E.1	No se le permite al usuario guardar los cambios

Tabla 11. Tabla de caso de uso consultar conjuntos para un día y lugar

Caso de uso	Consultar conjuntos para un día y un lugar		
Descripción	A través de un calendario integrado se podrán consultar los conjuntos que la aplicación haya creado para ese día		
Actores	Usuario		
Precondición			
Escenario Principal	Paso	Acción	
	1	El usuario accede a la ventana de Calendario	
	2	Pulsa un día del calendario	
	3	En base a los eventos de ese día y el tiempo el sistema crea dos conjuntos	
	4	El sistema muestra los dos conjuntos en la parte inferior de la pantalla	
Excepciones	Paso	Acción	
	3	Para el día seleccionado falta más de una semana	
		E.1	El sistema muestra un mensaje comunicándole al usuario que no es posible sugerir conjuntos ya que la predicción del tiempo para esa fecha no es fiable

Tabla 12. Tabla de caso de uso consulta de tiempo automática

Caso de uso	Consulta del tiempo (automático)	
Descripción	Consultar el clima de la ciudad en la que se encuentra	
Actores	Usuario	
Precondición	El usuario debe de permitir la ubicación	
Escenario Principal	Paso	Acción
	1	El usuario accede a la ventana de Calendario
	2	El sistema muestra temperatura, humedad y velocidad del viento de la ciudad en la que se encuentra

Tabla 13. Tabla de caso de uso consulta del tiempo manual

Caso de uso	Consulta del tiempo (manual)		
Descripción	Consultar el clima en una ciudad completa		
Actores	Usuario		
Precondición			
Escenario Principal	Paso	Acción	
	1	El usuario accede a la ventana de Calendario	
	2	El usuario introduce en un campo de texto una ciudad especifica	
	3	El sistema muestra temperatura, humedad y velocidad del viento en la ciudad escogida	
	4	El sistema actualiza los conjuntos sugeridos en función a los nuevos datos obtenidos	
Postcondición	Se actualizan los conjuntos mostrados en la ventana		
Excepciones	Paso	Acción	
	2	La ciudad introducida no existe	
		E.1	El sistema muestra un mensaje por pantalla informando que la información introducida no es correcta

Capítulo 5

Especificación de la capa de persistencia

Desde este capítulo la memoria se centrará en el back-end y la parte del *front-end* lo realizará Marc Ferrandis López en la memoria llamada: “Closet Assistant, tu estilista de bolsillo: desarrollo del front-end”.

En esta memoria se va a analizar cómo se ha diseñado e implementado la base de datos y arquitectura siguiendo los casos de usos definidos en el capítulo anterior.

5.1. Diseño y base de datos

Ilustración 16. SQL vs NoSQL

En este apartado se va a explicar que es SQL y NoSQL, con ventajas y desventajas, y el motivo de elección de SQL con su respectivo modelo de base de datos, el cual se va a usar en el proyecto

5.1.1. Que es SQL

Ilustración 17. Como se percibe SQL

SQL (Structured Query Languaje) es un lenguaje que fue creado por IBM para administrar y recuperar información de los sistemas de gestión de bases de datos relacionales.

La forma en la que trabaja SQL es con tablas interrelacionadas las cuales están formadas por filas y columnas. Las columnas hacen referencia al tipo del dato que hay que guardar (ej: números enteros, números decimales, texto, etc…) y las filas serán los datos a guardar siguiendo las pautas indicadas en las columnas.

Algunos sistemas de base de datos SQL más conocidos son: MySQL, SQL Server, MariaDB, SQLite…

Estas son algunas ventajas sobre SQL:

- **Atomicidad** en sus operaciones. En caso de que surja algún error durante la ejecución de la consulta todos los cambios realizados por la misma se anulan para asegurar la integridad de los datos, lo cual le da ventaja sobre algunos lenguajes NoSQL que no contienen esta característica.

- **Estándares bien definidos.** Al ser un lenguaje con tanto recorrido se han podido crear estándares que abarcan toda la concepción del lenguaje, desde la creación de tablas hasta las consultas de estas.

- **Sencillez** a la hora de diseñar la base de datos deseada y hacer consultas a la misma. Este lenguaje fue diseñado para que sea simple de usar para cualquier persona.

Y estas son algunas desventajas:

- **Difícil cambio del modelo.** SQL no es flexible en determinados campos por lo que hay veces que toca hacer un cambio al modelo de la base de datos, para lo que se necesita altos conocimientos sobre el modelo diseñado y cómo funcionan las tablas y relaciones.

- **Escalabilidad.** Según el modelo y los datos insertados aumentan en la base de datos se hace más difícil su mantenimiento, aumenta el tiempo de respuesta y el consumo de recursos, en las consultas principalmente serían de CPU – RAM y para mantener el servicio los HDD – SSD.

5.1.2. Que es NoSQL

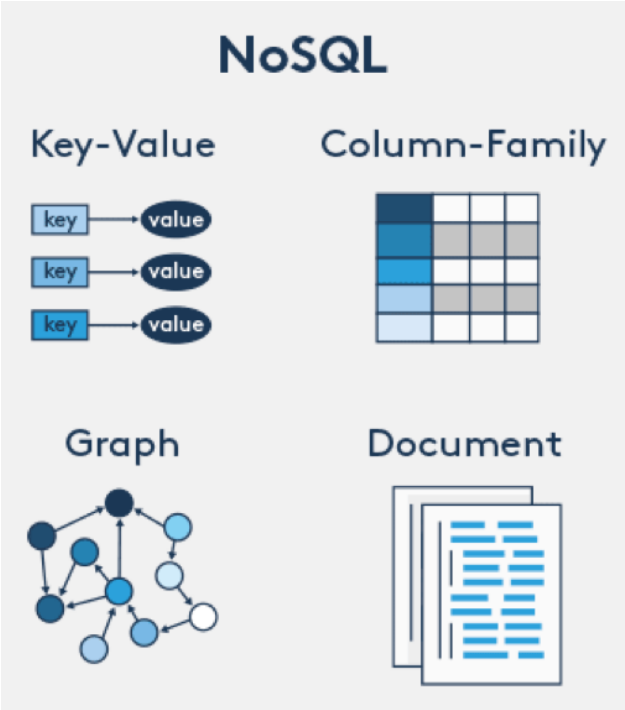


Ilustración 18. Modelos NoSQL

NoSQL (Not only SQL) es una forma alternativa de guardar los datos respecto a SQL, el cual al manejar grandes cantidades de datos las consultas pueden llegar a ser muy costosas lo que provoca que no sea escalable.

La motivación de la creación de NoSQL es romper con las limitaciones que contiene SQL, así se podrían crear BD mucho más extensas y eficientes.

A continuación, se exponen cuatro modelos para bases de datos NoSQL:

- Almacén de documentos: Los datos y los metadatos se almacenan jerárquicamente en documentos basados en JSON dentro de la base de datos.

- Almacén clave – valor: La más sencilla de las bases de datos NoSQL, los datos se representan como una colección de pares clave-valor.

- Almacén de columna ancha: Los datos relacionados se almacenan como un conjunto de pares clave-valor anidados dentro de una sola columna.

- Almacén de grafos: Los datos se almacenan en una estructura de grafo como propiedades de nodo, borde y datos.

Estas son algunas de las ventajas de los modelos NoSQL:

- No usan tablas, sino enlaces entre elementos. Si hiciéramos una traza de un elemento y sus hijos se podría apreciar algo similar a un despliegue de un árbol de datos.

- Escalabilidad del modelo, ya que al acceder directamente a los datos a consultar sus subconsultas muy probablemente serán un subconjunto muy pequeño del conjunto de datos total, lo cual aligera drásticamente el peso de la consulta.

- Uso de la notación JSON (Java Script Object Notation) para todas las comunicaciones que se realicen.

- Guardado de objetos sin seccionar, lo que agiliza la posterior extracción de estos, ya que no tiene la necesidad de hacer JOINS u otras operaciones adicionales.

Algunas de las desventajas de NoSQL son:

- Todas las ventajas que tiene NoSQL le limita a tener que relajar parte de las propiedades ACID (Atomicity, Consistency, Isolation, Durability), lo que peligrar su consistencia.

- Falta de estandarización respecto a SQL, ya que es un concepto algo novedoso no le ha dado tiempo a crear los estándares adecuados.

- Es más complicada la interoperabilidad entre otras BD NoSQL
- Existe menos soporte para la comunidad

5.1.3. Elección de SQL sobre NoSQL

El proyecto se ha planteado para desarrollarse con la tecnología Android y SQLite, de los cuales se hablarán en profundidad en el apartado de tecnologías utilizadas.

Android tiene integrado un sistema SQL llamado SQLite el cual es muy ligero y no necesita un servidor para poder ejecutar consultas a la base de datos, lo que lo hace un gran candidato como SGBD (sistema de gestión de bases de datos).

Si la aplicación a ejecutar tiene en las dependencias SQLite activado automáticamente se llamará al proceso para cuando se quiera abrir la BD ya esté todo preparado.

Como último motivo diría la experiencia que tengo con SQL respecto a NoSQL, el cual he manejado, pero no tengo tanta destreza como SQL.

5.1.1.1. Modelo SQL conceptual

A la hora de desarrollar un proyecto si se decanta por una base de datos SQL se necesita realizar un modelo conceptual sin mucha profundidad en el cual se definan las posibles tablas y relaciones.

En mi caso he optado por usar SQLite que es una implementación del estándar SQL92, por lo que he realizado un boceto UML (Diagrama 5) del modelo de base de datos que necesitará la aplicación.

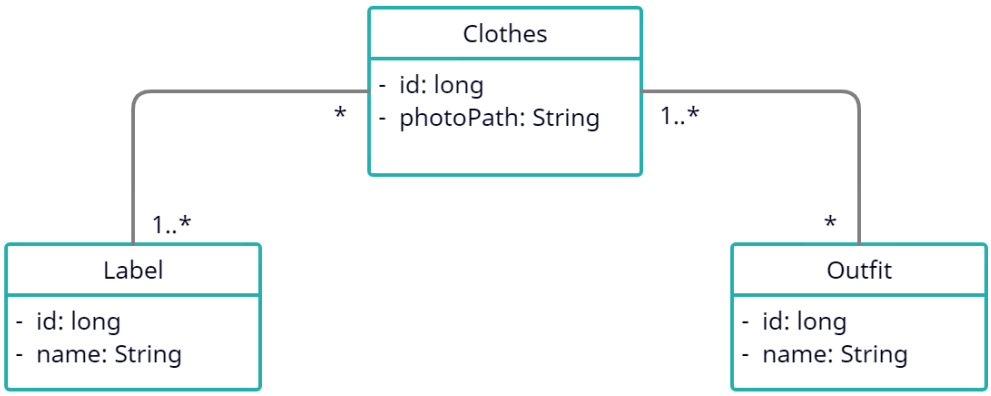


Diagrama 5. Boceto inicial de las clases UML a implementar en la base de datos

Descripción de las tablas y sus columnas:

- **Label**: Tabla con las etiquetas que se van a usar en las prendas
  - **id**: Identificador del Label en la BD (Base de Datos)
  - **name**: Nombre que va a tener la etiqueta

- **Clothes**: Tabla con las prendas de ropa que añade el usuario
  - **id**: Identificador de la Clothes en la BD
  - **photoPath**: URI de la imagen que contiene una prenda

- **Outfit**: Tabla con los conjuntos que crea el usuario
  - **id**: Identificador del Outfit en la BD
  - **name**: Nombre que se le da al conjunto

5.1.1.2. Modelo SQL actual

A lo largo del primer MVP (Minimum Viable Product) se han tenido que realizar múltiples cambios al boceto conceptual (Diagrama 5) empezando por romper las relaciones muchos a muchos, ya que no es posible implementarlas directamente en SQL, por lo que he creado un modelo UML (Diagrama 6) de la BD implementada para el primer MVP.

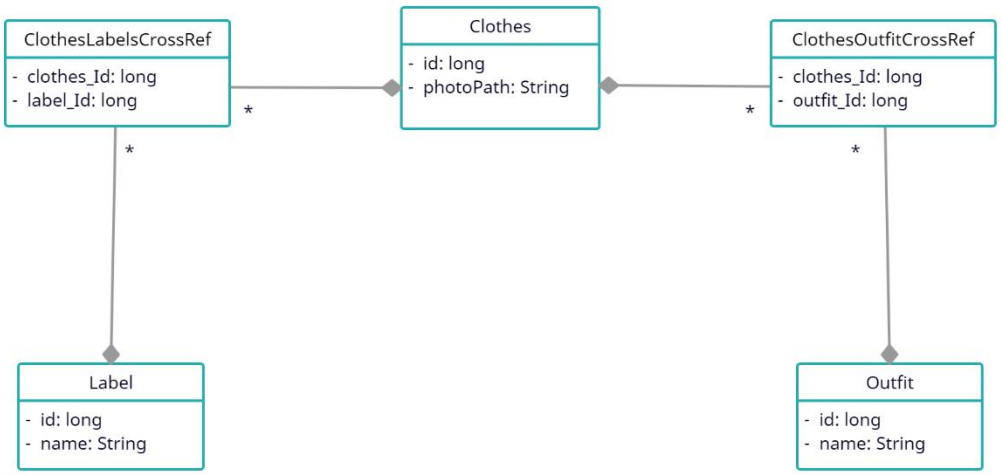


Diagrama 6. Modelo de clases actual UML de la base de datos

A continuación, se describen las nuevas tablas, las cuales son referencias cruzadas de las tablas del modelo conceptual, y el porqué de las relaciones:

- **ClothesLabelsCrossRef**: Tabla que contiene los identificadores del Clothes y Label relacionados.
    - **clothes_Id**: Identificador del objeto Clothes
    - **label_Id**: Identificador del objeto Label

- **ClothesOutfitCrossRef**: Tabla que contiene los identificadores del Clothes y Outfit relacionados.
    - **clothes_Id**: Identificador del objeto Clothes
    - **outfit_Id**: Identificador del objeto Outfit

Las relaciones de asociación se han cambiado por composición, ya que, si se elimina algún objeto Label, Clothes u Outfit interesa que todas las referencias que contengan el identificador del objeto eliminado también se eliminen.

5.2. Interfaces del modelo

A la hora de abarcar el proyecto se han tenido que plantear unas interfaces comunes y propias al menos para los tres objetos básicos: Label, Clothes y Outfit. Estas interfaces expondrán métodos públicos, los cuales usarán las clases encargadas de gestionar el correcto tratamiento de los tres objetos básicos.

El principal beneficio de usar interfaces es poder definir un “contrato” de métodos que va a tener la clase que implemente esa interfaz, por lo que el resto de clases en todo momento va a conocer que métodos expone, lo cual puede ayudar a reducir las inconsistencias en el proyecto.

Se han creado cuatro interfaces: IDbHelper<T>, IDbLabel, IDbClothes y IDbOutfit; cómo se puede apreciar la primera de ellas tiene generalización, por lo que simplemente poniendo el tipo de objeto a tratar rellenará todos los campos necesarios.

Exposición de la especificación de las interfaces y la labor respectiva de cada método:

- **IDbHelper<T>**: Interfaz esencial para poder añadir, editar y eliminar datos del modelo proporcionado.
    - **long add(T entity)**: Inserta a la BD el elemento de tipo T indicado al importar la interfaz y devuelve el id del mismo.
    - **Integer update(T entity)**: Dado el objeto a actualizar devuelve el número de filas afectadas por el cambio.
    - **Integer delete(T entity)**: Dado el objeto a eliminar devuelve el número de filas eliminadas.
    - **T getById(long id)**: Dado el id del objeto a recuperar devuelve el objeto solicitado si existe.
    - **List<T> getAll()**: Devuelve todos los objetos que contenga la tabla que implemente este método.
    - **long[] addList(List<T> entityList)**: Funcionalidad equivalente a *long add(T entity)*, pero ahora puedes añadir una lista de elementos y te devuelve una array con las posiciones donde se han insertado.
    - **Integer deleteList(List<T> entityList)**: Funcionalidad equivalente a *Integer delete(T entity)*, pero ahora puedes eliminar una lista de elementos y devuelve el número de filas eliminadas.
    - **long deleteAll()**: Elimina todas las filas de la tabla que implementa este método y devuelve el número de filas eliminadas.
    - **Integer getSize()**: Solicita el número de filas que contiene la tabla que implementa este método.

- **IDbLabel**: Interfaz para extraer objetos relacionados con los Label:
    - **HashMap<Label, List<Clothes>> getClothesPerLabel()**: Devuelve un HashMap con cada Label y todas las Clothes que hacen referencia a ese Label.
    - **List<ClothesLabelsCrossRef> getLabelWithClothesCrossRef(long labelId)**: Dado el id de un Label te devuelve todas las referencias

cruzadas que contengan ese id en el campo de *label_Id* de la tabla *ClothesLabelCrossRef*.

- **long[] insertLabelWithClothesCrossRef(Label label, List<Clothes> clothesList)**: Método que inserta los elementos en la BD si es necesario y luego con esos ids crea referencias cruzadas, las cuales también insertan en la tabla de *ClothesLabelsCrossRef*, el método devuelve las ids de las referencias cruzadas insertadas.
- **void updateLabelWithClothesCrossRef(Label label, List<Clothes> clothesList)**: Método que edita las referencias cruzadas que tiene asignadas el Label, la lista de Clothes proporcionada serán los únicos elementos que tendrán referencias cruzadas con el Label seleccionado, por lo que todos los que no estén contenidos en la lista serán desechados.
- **int deleteLabelWithClothesCrossRef(Label label, List<Clothes> clothesList)**: Método que elimina las referencias cruzadas de la lista de Clothes con el Label seleccionado.
- **List<Clothes> getClothesOfALabel(long labelId)**: Dado el id de un Label devuelve una lista de Clothes enlazadas con el mismo.

- **IDbClothes**: Interfaz para extraer objetos relacionados con las Clothes.
  - **HashMap<Clothes, List<Label>> getLabelsPerClothes()**: Devuelve un HashMap con cada Clothes y todos los Label que hacen referencia a ese Clothes.
  - **HashMap<Clothes, List<Outfit>> getOutfitsPerClothes()**: Devuelve un HashMap con cada Clothes y todos los Outfit que hacen referencia a ese Clothes.
  - **List<ClothesLabelsCrossRef> getClothesWithLabelsCrossRef (long clothesId)**: Dado el id de un Clothes te devuelve todas las referencias cruzadas que contengan ese id en el campo de clothes_*Id* de la tabla *ClothesLabelCrossRef*.
  - **List<ClothesOutfitCrossRef> getClothesWithOutfitsCrossRef(long clothesId)**: Dado el id de un Clothes te devuelve todas las referencias cruzadas que contengan ese id en el campo de clothes_*Id* de la tabla *ClothesOutfitCrossRef*.
  - **long[] insertClothesWithLabelsCrossRef(Clothes clothes, List<Label> labelList)**: Método que inserta los elementos en la BD si es necesario y luego con esos ids crea referencias cruzadas, las cuales también insertan en la tabla de *ClothesLabelsCrossRef*, el método devuelve las ids de las referencias cruzadas insertadas.
  - **void updateClothesWithLabelsCrossRef(Clothes clothes, List<Label> labelList)**: Método que edita las referencias cruzadas

que tiene asignadas la Clothes, la lista de Label proporcionada serán los únicos elementos que tendrán referencias cruzadas con la Clothes seleccionada, por lo que todos los que no estén contenidos en la lista serán desechados.

- **int deleteClothesWithLabelsCrossRef(Clothes clothes, List<Label> labelList)**: Método que elimina las referencias cruzadas de la lista de Label con la Clothes seleccionada.
- **long[] insertClothesWithOutfitsCrossRef(Clothes clothes, List<Outfit> outfitList)**: Método que inserta los elementos en la BD si es necesario y luego con esos ids crea referencias cruzadas, las cuales también insertan en la tabla de *ClothesOutfitCrossRef*, el método devuelve las ids de las referencias cruzadas insertadas.
- **void updateClothesWithOutfitsCrossRef(Clothes clothes, List<Outfit> outfitList)**: Método que edita las referencias cruzadas que tiene asignadas la Clothes, la lista de Outfit proporcionada serán los únicos elementos que tendrán referencias cruzadas con la Clothes seleccionada, por lo que todos los que no estén contenidos en la lista serán desechados.
- **int deleteClothesWithOutfitsCrossRef(Clothes clothes, List<Outfit> outfitList)**: Método que elimina las referencias cruzadas de la lista de Outfit con la Clothes seleccionada.
- **List<Label> getLabelsOfAClothes(long clothesId)**: Dado el id de un Clothes devuelve una lista de Label enlazados con el mismo.
- **List<Outfit> getOutfitsOfAClothes(long clothesId)**: Dado el id de un Clothes devuelve una lista de Outfit enlazados con el mismo.

- **IDbOutfit**: Interfaz para extraer objetos relacionados con los Outfit.
  - **HashMap<Outfit, List<Clothes>> getClothesPerOutfit()**: Devuelve un HashMap con cada Outfit y todas las Clothes que hacen referencia a ese Outfit.
  - **List<ClothesOutfitCrossRef> getOutfitWithClothesCrossRef(long outfitId)**: Dado el id de un Outfit te devuelve todas las referencias cruzadas que contengan ese id en el campo de *outfit_Id* de la tabla *ClothesOutfitCrossRef*.
  - **long[] insertOutfitWithClothesCrossRef(Outfit outfit, List<Clothes> clothesList)**: Método que inserta los elementos en la BD si es necesario y luego con esos ids crea referencias cruzadas, las cuales también insertan en la tabla de *ClothesOutfitCrossRef*, el método devuelve las ids de las referencias cruzadas insertadas.
  - **void updateOutfitWithClothesCrossRef(Outfit outfit, List<Clothes> clothesList)**: Método que edita las referencias

cruzadas que tiene asignadas el Outfit, la lista de Clothes proporcionada serán los únicos elementos que tendrán referencias cruzadas con el Outfit seleccionado, por lo que todos los que no estén contenidos en la lista serán desechados.

- **int deleteOutfitWithClothesCrossRef(Outfit outfit, List<Clothes> clothesList)**: Método que elimina las referencias cruzadas de la lista de Clothes con el Outfit seleccionado.
- **List<Clothes> getClothesOfAOutfit (long outfitId)**: Dado el id de un Outfit devuelve una lista de Clothes enlazadas con el mismo.

Capítulo 6

Tecnologías utilizadas

En este capítulo se van a enumerar y explicar las tecnologías más relevantes usadas en el proyecto, desde entornos de trabajo, pasando por herramientas y acabando en bibliotecas utilizadas.

6.1. Android

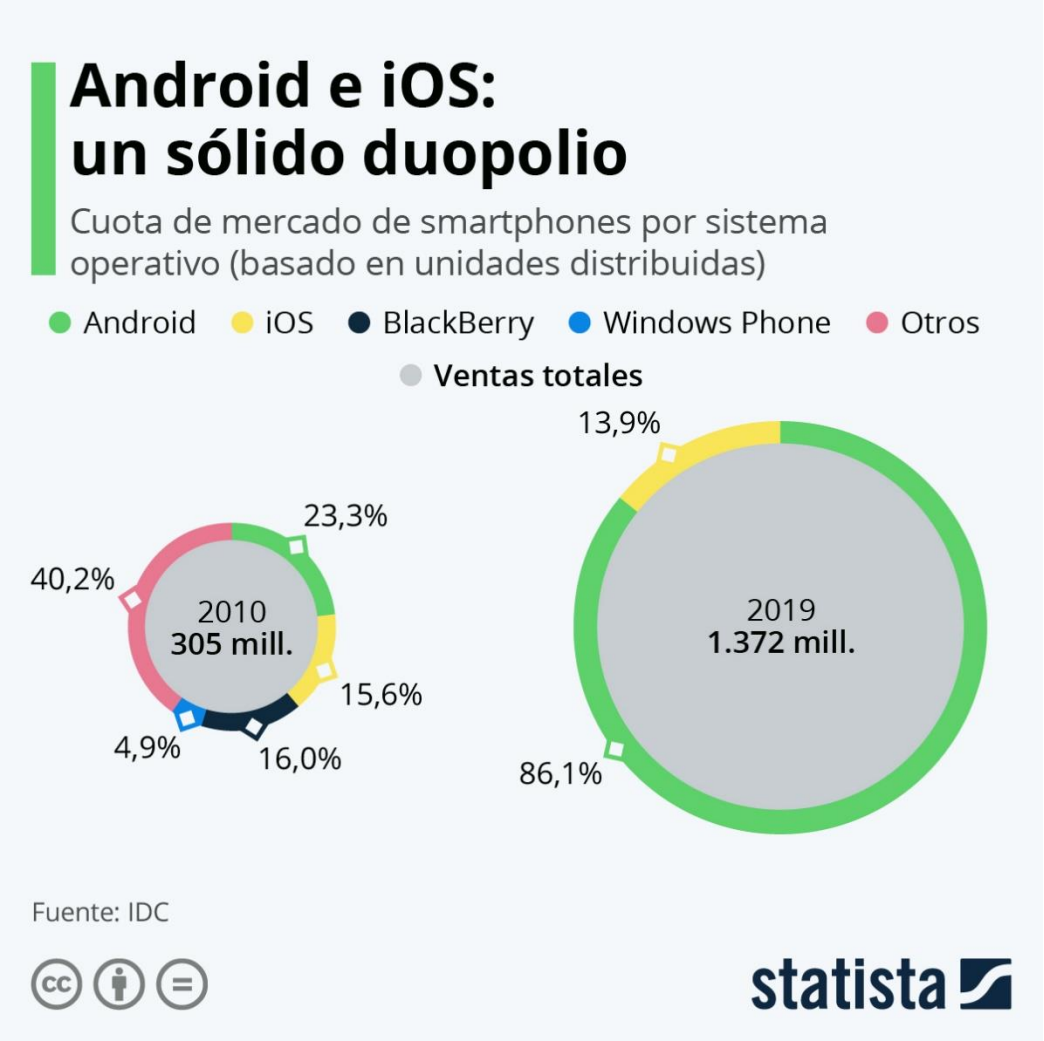
Ilustración 19. Logo de Android

Android[1] es un sistema operativo móvil el cual está basado principalmente en el kernel de Linux. Tal y como se puede apreciar en la Gráfica 26 , Android actualmente es el sistema operativo móvil más usado en el mundo, lo que lo hace un gran candidato como tecnología base para desarrollar la aplicación.

Según el país seleccionado puede haber una gran diferencia entre la proporción de sistemas operativos, por ejemplo, en la "Gráfica 27. Sistemas operativos, móviles en España" se puede apreciar que el 83.6% son Android, pero por lo contrario en la "Gráfica 28. Sistemas operativos, móviles en EEUU" es tan solo del 54,1%.

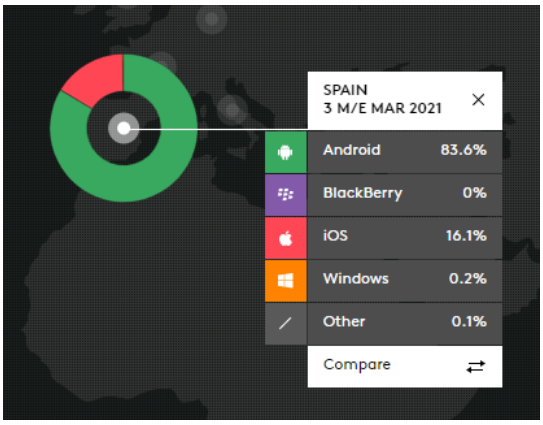
Pese a que en cada país la proporción de gente que usa Android puede fluctuar, ésta siempre alcanza la mayoría.

[1] https://www.android.com/

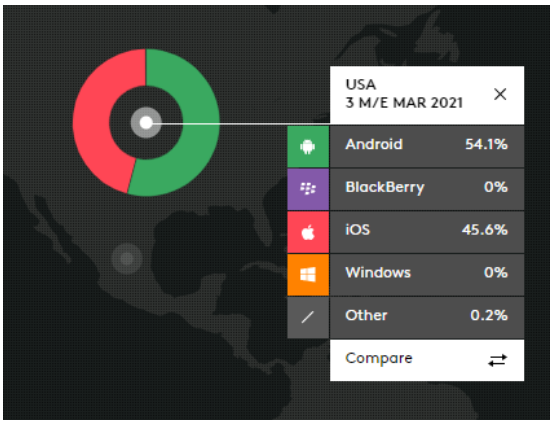


Gráfica 26. Sistemas operativos por smartphone distribuido [1]

Como se puede apreciar Android siempre ha ido en cabeza en número de dispositivos integrados con su sistema operativo, también potenciado por la compra de Google y su intento de crear una estandarización de sistema operativo para smartphones, ya que como podemos apreciar en la gráfica superior en 2010 era un caos y los desarrolladores tenían que contemplar las limitaciones y casos propios de cada sistema lo que repercutía negativamente en el tiempo de desarrollo y calidad del producto.



Gráfica 27. Sistemas operativos, móviles en España

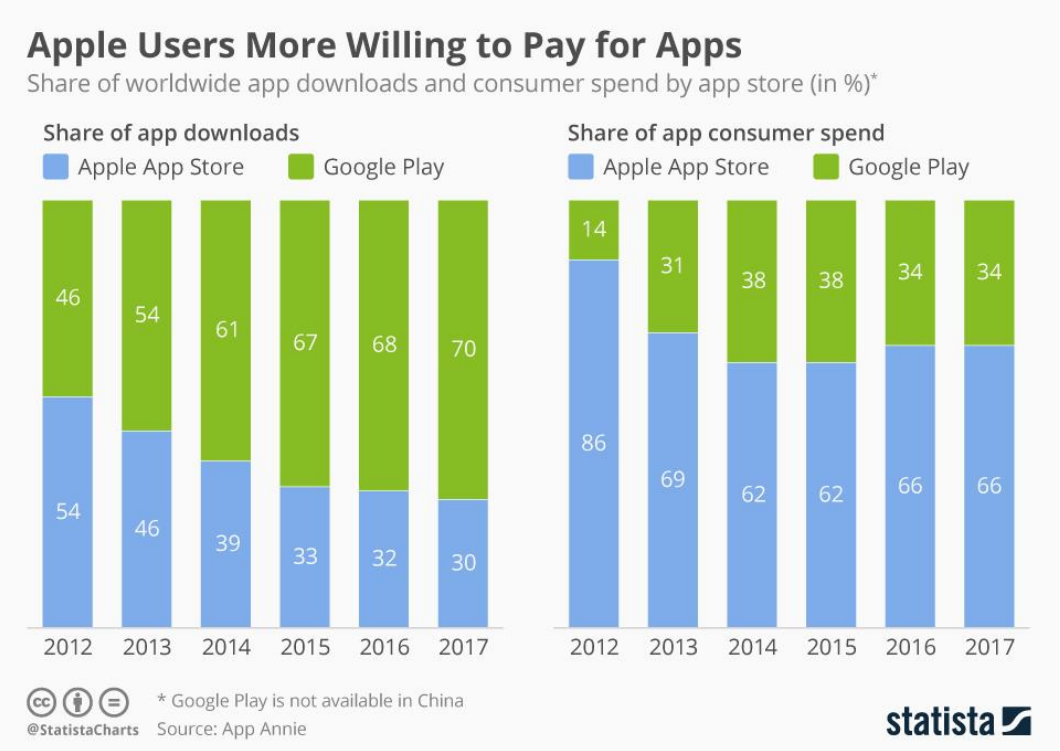


Gráfica 28. Sistemas operativos, móviles en EEUU

Gráficas extraídas de la página interactiva Kantar[2] en la que se puede ver un listado de países y la cuota de mercado de cada sistema operativo de cada uno desde 2012 hasta hoy.

A la hora de publicar una aplicación se necesita una licencia de desarrollador, tanto para Android como para iOs, pero las licencias cuestan 25€ un único pago y 100€ anuales respectivamente, por lo que es mucho más atractivo Android por su baja cuota de desarrollador.

[2] https://www.kantarworldpanel.com/global/smartphone-os-market-share/



Gráfica 29. Descargas y ganancias iOs vs Android [2]

Como se expone anteriormente, Android tiene una cuota de desarrollador mucho inferior a iOs y muchos más usuarios, pero pese a todo la store de iOs tiene muchos más beneficios que las de Android. Para hacernos una idea entre las 100 aplicaciones más populares de ambas stores, en el primer trimestre de 2019 iOs ganó $84M y Android $51M[3], aunque como es natural en Android se consiguen más descargas (Gráfica 29).

A la hora de desarrollar en Android hay múltiples alternativas entre las cuales están:

- **Java**[3] es la opción clásica.

- **Kotlin**[4], la alternativa creada por JetBrains[5] y potenciada por Google como lenguaje oficial para Android al mismo nivel que Java. Este lenguaje es capaz de eliminar código repetitivo el cual aparece en varias partes del proyecto sin ninguna o muy poca alteración simplificando las declaraciones de tipos, clases, funciones, etc…

  Adicionalmente Kotlin permite exportar la lógica de negocio a Android e iOs, pero no el código especifico de cada plataforma como puede ser las vistas de usuario.

[3] https://www.java.com/
[4] https://kotlinlang.org/
[5] https://www.jetbrains.com/

- **Flutter**[6] en conjunto con Dart[7], otra alternativa creada por Google para desarrollar un proyecto multiplataforma el cual se puede exportar a Android, iOs y web (beta).

- **Xamarin**[8], un proyecto mantenido por Microsoft el cual permite programar en C#[9] y junto a unos archivos .xaml (algo similar a HTML) permite exportar la aplicación a escritorio, Android e iOs.

Al final nos hemos decantado por el desarrollo en Android nativo con Java, ya que he cursado una optativa de desarrollo de aplicaciones para dispositivos móviles donde nos enseñan a como programar nativamente en Android con Java.

A parte de la optativa, desde principios de la carrera nos han enseñado lenguajes orientados a objetos de entre los cuales mayoritariamente se han centrado en Java, lenguaje apto para programar en Android nativo.

Para finalizar, el desarrollo nativo de aplicaciones consigue una mejor optimización de la aplicación en comparación con lenguajes multiplataforma, ya que se ahorra realizar transformaciones adicionales y no tiene ninguna limitación del SDK (Software Development Kit) propio del sistema operativo ni del hardware del dispositivo.

6.2. SQLite

Ilustración 20. Logo de SQLite

SQLite[10] es un sistema gestor de bases de datos relacional de dominio público, autocontenido, no necesita un servidor, no necesita configuración, transicional (cumple las reglas ACID) e implementa el estándar SQL92, por lo que es capaz de soportar todas las consultas que se pueden realizar en SQL y más.

[6] https://flutter.dev/
[7] https://dart.dev/
[8] https://docs.microsoft.com/es-es/xamarin
[9] https://docs.microsoft.com/es-es/dotnet/csharp/
[10] https://www.sqlite.org/

También es la base de datos más desplegada del mundo y es usada por grandes compañías como: Adobe, Airbus, Apple, Google, Facebook, Microsoft… [11]

Algunas de sus ventajas y logros son:

- El **formato de archivos** es **multiplataforma**, por lo que puede trasladarse en cualquier momento a otra base de datos con sistemas entre 32 y 64 bits.

- La biblioteca del congreso de Estados Unidos ha seleccionado este **formato de almacenamiento** como el **recomendado** para **guardar conjunto de datos**[12], otros formatos también recomendados por la misma institución son: XML, JSON y CSV.

- Es una **biblioteca muy compacta**, con todas las funcionalidades activadas puede llegar a ocupar menos de 600KB (dependiendo de los parámetros de optimización del compilador)[13], por lo que lo hace una muy buena opción para los dispositivos móviles entre tantos otros campos.

- Ha **pasado millones de test automáticos** de llamadas SQL en los cuales ha conseguido un 100% de cobertura de ramas.

- Es capaz de usar **múltiples hilos de ejecución** para hacer las consultas, siempre cumpliendo la atomicidad de sus consultas.

Uno de los inconvenientes de SQLite es que funciona mejor cuanta mayor RAM tenga el dispositivo, pero aun así es capaz de defenderse en situaciones con poca disponibilidad de RAM.

[11] https://www.sqlite.org/famous.html
[12] https://www.sqlite.org/locrsf.html
[13] https://www.sqlite.org/about.html

6.3. Android Jetpack

Ilustración 21. Logo de Android Jetpack

Android Jetpack[14] es un conjunto de bibliotecas que intentan ayudar a los desarrolladores a: seguir las prácticas recomendadas, reducir el código repetido y escribir código de forma coherente. Adicionalmente tiene cursos de conceptos teóricos, prácticos y ejemplos de aplicaciones, de las cuales proporcionan el código fuente, desarrolladas en Android siguiendo los patrones y bibliotecas que recomiendan Android Jetpack.

En los siguientes subapartados voy a enumerar y explicar algunas de las bibliotecas usadas en el proyecto que están contenidas en el conjunto de Android Jetpack.

6.3.1. Room

La gestión de SQLite en Android puede llegar a ser algo complejo, pero por suerte existe Room[15], una biblioteca de persistencia la cual da una capa de abstracción sobre SQLite para poder acceder a la base de datos aprovechando toda su potencia y sin problemas adicionales.

[14] https://developer.android.com/jetpack

[15] https://developer.android.com/jetpack/androidx/releases/room

Al proporcionar una capa de abstracción es capaz de manejar las tablas de SQLite con los datos guardados y crear instancias de POJO (Plain Old Java Object) sin necesitad de crear nuevas tablas que los respalden, aunque si se desea se puede habilitar esta funcionalidad marcando el POJO con @Entity.

Todos los objetos o clases que vaya a manejar Room necesitarán usar decoradores (ej: @Entity, @PrimaryKey, @Relation, …) y Room se encargará de hacer el resto de las operaciones necesarias para la creación de esos objetos, por lo que no sería necesario escribir nada en SQL más que las consultas complejas. Como se puede apreciar todo lo explicado sobre Room muestra que cumple con la premisa de eliminar el código repetido tal y como busca el conjunto de bibliotecas Android Jetpack.

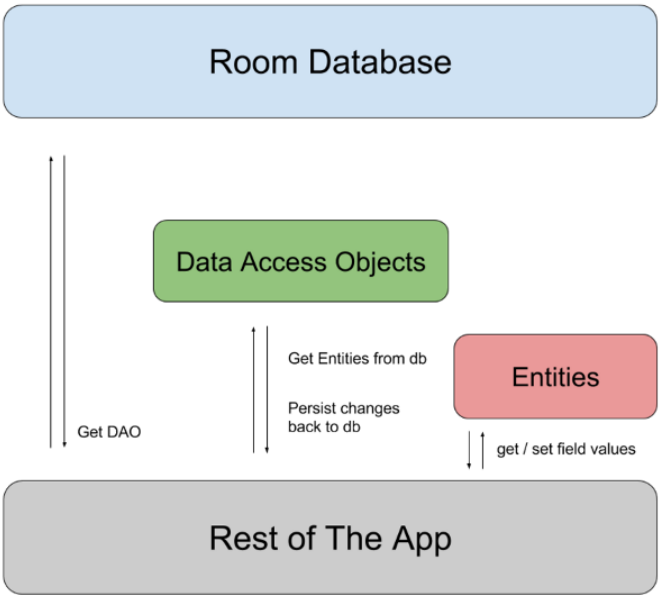


Diagrama 7. Arquitectura de Room [16]

6.3.2. LifeCycle

LifeCycle[17] es una librería la cual contiene componentes escuchando el ciclo de vida de otros componentes como las actividades o fragmentos. Es una solución más estandarizada y estructurada de hacer listeners y observers en proyectos Android.

Los componentes que contiene LifeCycle y han sido usado en el proyecto son los siguientes:

[16] https://developer.android.com/training/data-storage/room
[17] https://developer.android.com/jetpack/androidx/releases/lifecycle

- **Livedata**: Componente capaz de superar la funcionalidad de listener y observers clásicos de forma simplificada, ya que al objeto de tipo Livedata<T> le añades el elemento que quieres guardar y automáticamente todos los objetos que lo estén observando recibirán el nuevo valor guardado. Adicionalmente livedata escucha los lifecycle de la aplicación y los respeta, lo que significa que no se activarán los observers que su lifecycle esté pausado o destruido

- **ViewModel**: Componente que se encarga de guardar valores para poder comunicarlo entre distintas vistas. Teóricamente en Android dos vistas no pueden tener intercambio de datos constante, por ello se tiene que usar ViewModel que guarda los datos que una vista quiera exponer y desde otra vista se pueden recuperar.

ViewModel y Livedata tienen muy buena sinergia, ya que ViewModel puede hacer de repositorio que guarda los datos y Livedata se encargaría de comunicarlo al resto de vistas respondiendo a sus observers tal y como se puede apreciar en el Diagrama 8.

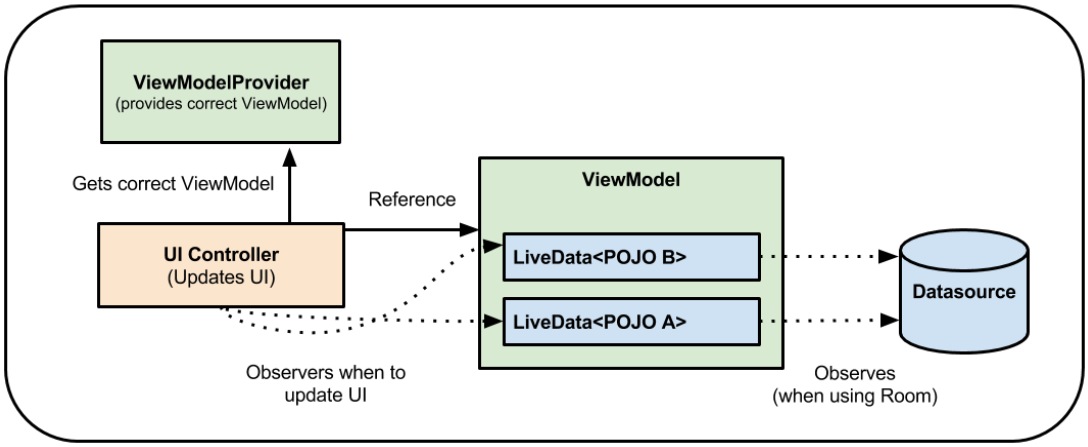


Diagrama 8. Funcionamiento de ViewModel y Livedata [18]

En el Diagrama 8 se puede apreciar el funcionamiento de los módulos de esta biblioteca. El Livedata se dedica a observar los datos que le proporciona Room y luego notifica a los observadores que controlan el IU (Interfaz de Usuario) para actualizarlo. El ViewModel únicamente se encarga de mantener las referencias a los objetos que se les ha asignado, para que en caso de que la instancia de la IU haya sido destruida el ViewModelProvider sea capaz de enlazar nuevamente la nueva instancia de la IU al ya existente ViewModel con los datos intactos.

[18] https://developer.android.com/topic/libraries/architecture/viewmodel

6.3.3. Paging

Paging[19] es una biblioteca la cual es capaz de extraer grandes cantidades de información de la base de datos y mostrar un pequeño conjunto en la capa de presentación, a este concepto se le llama paginación.

Su funcionamiento es extraer una página con el número de elementos que deseamos y cuando lo tenga listo lo devolverá. En caso de que acabemos la página de elementos solicitará otra más y así hasta que no queden más elementos que devolver.

Un ejemplo sencillo sería una llamada que devuelve un conjunto de elementos y tarda en devolverlos todos 20 segundos, pues con paginación podríamos hacer páginas del tamaño que deseemos y que tardara tan solo un segundo o incluso menos, en función de las necesidades que tengamos.

Esta biblioteca tiene una dependencia total para su implementación con el apartado 6.3.2. LifeCycle, el cual se puede apreciar en el Diagrama 9.

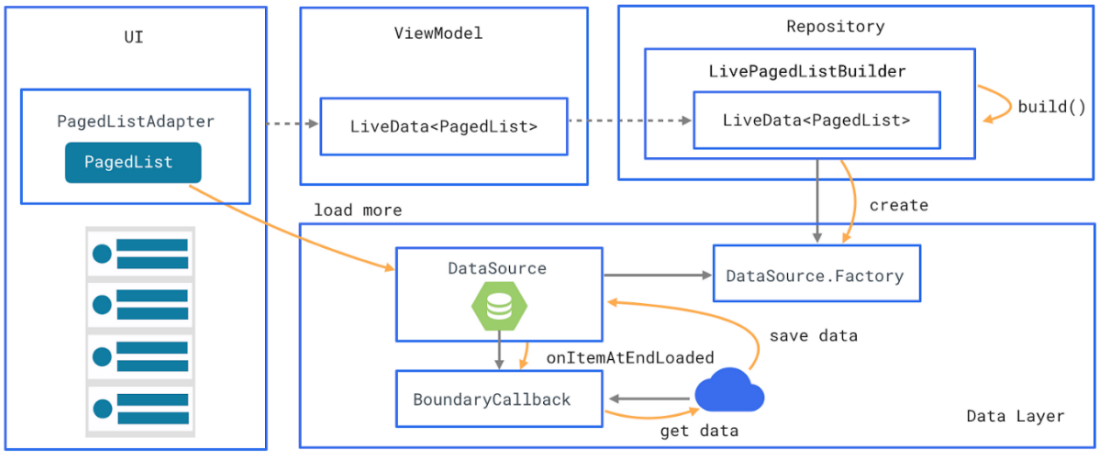


Diagrama 9. Paging y LifeCycle [4]

En el Diagrama 9 se puede apreciar con más profundidad el funcionamiento y sinergia que tienen la biblioteca LifeCycle y Paging. A continuación, voy a explicar la traza del Diagrama 8 y Diagrama 9 unidas:

1. Desde la IU se solicita una instancia de ViewModel mediante el ViewModelProvider.

2. El ViewModel crea un objeto Livedata<PagedList> del tipo requerido por la IU.

[19] https://developer.android.com/jetpack/androidx/releases/paging

3. La IU se suscribe al objeto Livedata<PagedList> contenido en el ViewModel solicitado.

4. La IU solicita los datos a la BD

5. La BD devuelve una "página" de datos pura(subconjunto de datos solicitado por la IU)

6. La lógica de negocio recibe los datos puros y construye una PagedList

7. La lógica de negocio guarda la PagedList construida anteriormente en el objeto Livedata<PagedList> que está contenido en el mismo ViewModel que tiene referenciado la IU. El ViewModelProvider se encarga de que se hayan realizado correctamente las referencias al ViewModel.

8. El objeto Livedata<PagedList> notifica a todos los observadores que están suscritos.

9. Los observadores reciben los datos y actualizan la IU.

6.3.4. AppCompat

AppCompat[20] es una biblioteca que se hace cargo de que la aplicación sea retrocompatible con versiones API antiguas de Android y con las nuevas que vayan saliendo. Hay casos en los que hay que limitar la API mínima objetivo, pero no suele suceder mucho, por lo contrario, con APIs futuras no da problemas.

6.3.5. View Binding

View Binding[21] es una biblioteca que permite escribir código más fácilmente para interactuar con las vistas. Una vez habilitado el módulo cuando se compile el proyecto automáticamente se generará un archivo .java el cual tendrá todos los componentes de la vista .XML a representar.

[20] https://developer.android.com/jetpack/androidx/releases/appcompat
[21] https://developer.android.com/topic/libraries/view-binding

El archivo .java te dará más facilidad de acceso a los objetos del .XML con sus propiedades y tipos en vez de tener que solicitar al sistema que te dé un único elemento de la vista y luego hacer una conversión de tipo al elemento deseado.

6.3.6. Constraint Layout

Constraint Layout[22] es una biblioteca que se usa únicamente en los .XML que definen las vistas. Permite la creación de contenedores y evita la creación de layouts anidados, por lo que mejora el rendimiento de las vistas complejas.

6.3.7. Fragment

Fragment[23] es una biblioteca que permite la división de las vistas de la aplicación en fragmentos, los cuales estarán contenidos en una Activity. La ventaja principal respecto a una Activity es que puedes tener tantos fragmentos en pantalla como espacio haya, por lo contrario, las Activity solo se puede mostrar una al mismo tiempo.

6.3.8. ViewPager2

ViewPager2[24] es una biblioteca que permite la implantación de un módulo que es capaz de contener vistas y fragmentos con un formato deslizable. El resultado final sería algo parecido al efecto de pasar una hoja de un libro.

6.3.9. Arch Core

Arch Core[25] es una biblioteca de asistencia para las dependencias de la arquitectura la cual tiene incluida las reglas de prueba JUnit y permite el uso de Livedata mencionado en el punto 6.3.3. LifeCycle.

[22] https://developer.android.com/jetpack/androidx/releases/constraintlayout
[23] https://developer.android.com/jetpack/androidx/releases/fragment
[24] https://developer.android.com/jetpack/androidx/releases/viewpager2
[25] https://developer.android.com/jetpack/androidx/releases/arch-core

6.4. Android Studio

Ilustración 22. Logo de Android Studio

Android Studio[26] es el IDE (Integrated Development Environment) recomendado por la página de Android Developer para desarrollar aplicaciones en Android.

Este IDE está basado en el software IntelliJ IDEA[27] de JetBrains. Este incluye: soporte de construcción Gradle[28], refactorización específica de Android, integración de firma de aplicaciones, renderizado en tiempo real, un emulador que puede ejecutar aplicaciones en un sistema operativo Android, etc…

6.5. Git

Ilustración 23. Logo de Git

Git[29] es un sistema de control de versiones gratuito y de código abierto el cual es esencial para el desarrollo de cualquier aplicación.

[26] https://developer.android.com/studio
[27] https://www.jetbrains.com/idea/
[28] https://gradle.org/
[29] https://git-scm.com/

Git es un software de control de versiones el cual es capaz de guardar versiones o estados del código de la aplicación. Cada vez que guarda una nueva versión o estado no guarda todo el código, sino solo los cambios realizados, por lo hace ideal para poder trabajar entre varios desarrolladores. Si se da el caso de que hay conflictos con varios cambios Git tiene herramientas que permiten al desarrollador solucionar los conflictos a mano.

6.6. Worki Process

Ilustración 24. Logo de Worki Process

Worki Process[30] es una herramienta enfocada al desarrollo ágil en el cual múltiples desarrolladores son capaces de gestionar las tareas a completar por cada uno y hacer un seguimiento de este.

Esta herramienta nos permite usar un gran abanico de módulos como puede ser: workflows de trabajo, tableros Kanban, dashboards, roles para colaboradores, apartado con gráficas de todo el proyecto para poder hacer análisis de retrospectiva, etc...

Hemos elegido esta herramienta por la experiencia adquirida en la rama de ingeniería del software y no hemos encontrado otras alternativas que nos aporten tanto valor como Worki.

[30] http://www.tuneupprocess.com/

Capítulo 7

Implementación

En este capítulo se describe la arquitectura de la aplicación y la implementación de esta. Adicionalmente se expondrán con ejemplos distintas implementaciones de la aplicación entre las que están: ViewModel, Livedata, capa intermedia con la base de datos, etc…

7.1. Arquitectura

La arquitectura en un entorno software describe los patrones y las técnicas que se utilizan para diseñar y desarrollar aplicaciones. La arquitectura proporciona un plan y prácticas recomendadas que se deben seguir al momento de diseñar una aplicación, de modo que se obtenga una aplicación bien estructurada.

En los siguiente subapartados se va a hablar de los distintos tipos de arquitecturas que se van a implementar.

7.1.1. Arquitectura a tres capas

Desde momento cero ya sabía que quería una división por capas de la aplicación, ya que, aunque tenga el inconveniente de hacer la división inicial y tener que hacer un buen diseño el resto son todo ventajas como pueden ser:

- **Reutilización de capas**: Nos permite eliminar mucho código repetido a lo largo de la aplicación, lo que aumenta la eficiencia del proyecto.

- **Facilita la estandarización**: La reutilización de capas aumenta la estandarización del proyecto.

- **Dependencias intra-capa**: Puede parecer algo malo, pero donde antes podíamos tener dependencias en cualquier parte del proyecto ahora están limitadas a intra-capa.

- **Contención de los cambios**: Al aumentar la estandarización los cambios a realizar en el proyecto deberían ser mínimos, ya que solo tienen que hacerse a las clases centrales en vez de repetir los mismos cambios en todas las clases repetidas.

La arquitectura a tres capas se puede especializar en dos ramas, la primera, (Diagrama 10. Arquitectura a tres capas local sería implementando totalmente una lógica de negocio y base de datos local.

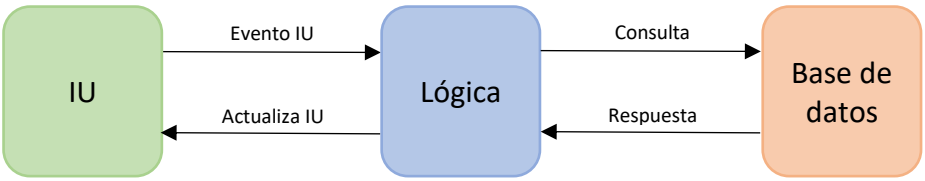


Diagrama 10. Arquitectura a tres capas local

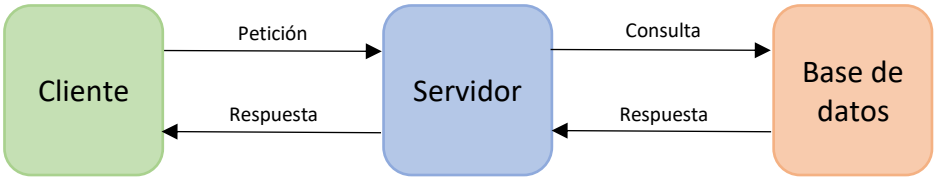


Diagrama 11. Arquitectura a tres capas cliente-servidor

La segunda, (Diagrama 11) sería realizando una arquitectura cliente-servidor en la cual normalmente habrá una lógica de negocio y base de datos más ligera en la parte del cliente, pero todo lo que te ahorras en la parte de cliente hay que completarlo en el servidor, por lo que este contendrá la base de datos más pesada y la lógica de negocio necesaria para tratar los datos y gestionarla.

Como se puede apreciar ambas ramas son muy parecidas pero cada una tiene sus ventajas y desventajas. La arquitectura a tres capas local es adecuada para aplicaciones que no necesitan conexión a internet, ya que como su propio nombre dice es local. Por lo contrario, la arquitectura cliente-servidor si que necesitaría conexión a internet para funcionar correctamente, pero tenemos la ventaja de que la aplicación cliente sería más ligera y consumiría menos recursos.

Tras todo lo expuesto anteriormente se generó un dilema, ya que pese que las arquitecturas parecen muy parecidas a la hora de la implementación son muy distintas, por lo que no sería una opción cambiar de arquitectura una vez comenzado el proyecto.

La aplicación fue pensada para usarse sin conexión a internet, pero en el futuro nos gustaría crear un servidor que respalde una copia de seguridad de los datos del cliente. A la hora de la implementación de la arquitectura hay que tener en mente dejarlo todo preparado para el futuro híbrido entre persistencia local y cliente-servidor, tal y como se puede ver en el Diagrama 12

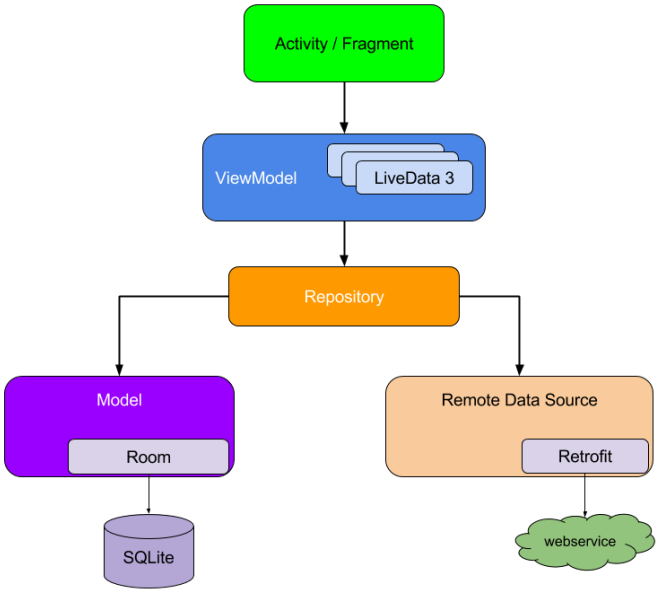


Diagrama 12. Arquitectura a tres capas mezclando persistencia local y cliente-servidor

7.1.2. Arquitectura MVVM

A parte de la arquitectura a tres capas, si queremos que la IU se actualice dinámicamente y facilitar la consistencia de esta de acuerdo con los eventos que ha lanzado, sin morir en el proceso, hay que usar la arquitectura MVVM (Model View ViewModel) tal y como se puede apreciar en el Diagrama 13

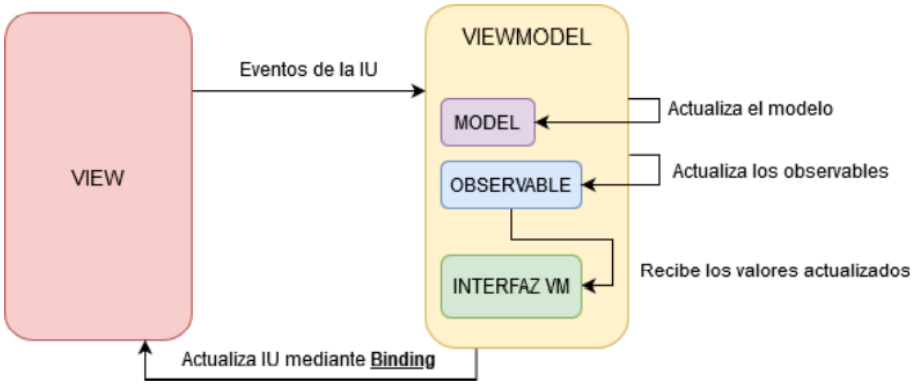


Diagrama 13. Arquitectura MVVM

En esta arquitectura los eventos de la IU en vez de ir todos directamente a la lógica de negocio, como se hace en la arquitectura a tres capas expuesta anteriormente, parte de ellos irán al ViewModel.

El ViewModel se encargará de hacer las consultas necesarias a la lógica de negocio, las cuales actualizarán las variables observables de tipo Livedata alojadas en el ViewModel. Estos observables responderán a los métodos que estén suscritos a ellos, siempre y cuando el proceso de la clase que contiene los métodos esté activo, de lo contrario se esperará hasta que vuelvan a estarlo para responder.

Los ViewModel proporcionaran una persistencia en memoria a todos los objetos declarados dentro de él mientras el proceso de la aplicación siga vivo.

7.1.3. Arquitectura final

En los dos apartados anteriores se exponen las dos arquitecturas a implementar en el proyecto, pero aún hay unificarlas.

En el Diagrama 14 se puede apreciar como quedaría la arquitectura actual del proyecto. Las líneas naranjas indican la traza que sigue una respuesta de la base de datos, así mismo, una consulta activada por un evento de la IU haría la traza inversa.

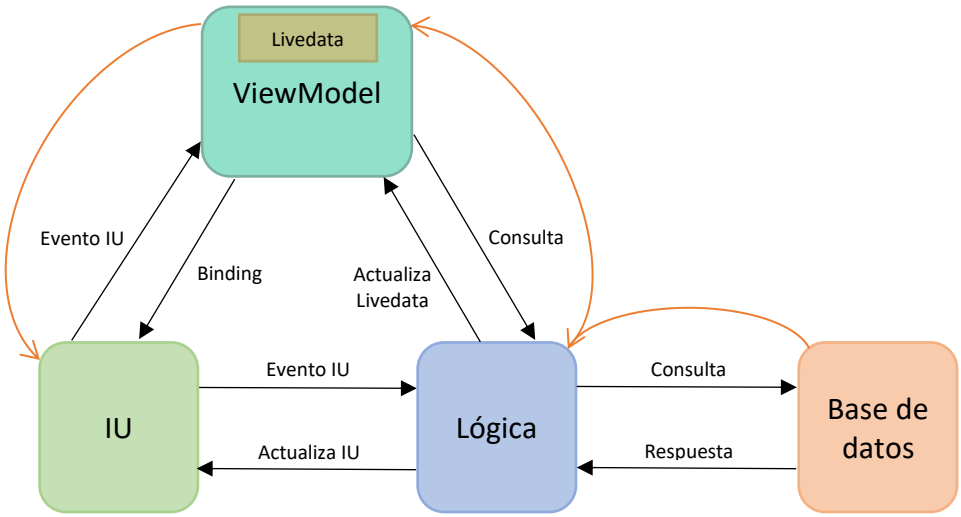


Diagrama 14. Arquitectura final

Adicionalmente se muestran capturas de la jerarquía de carpetas del proyecto y el significado de cada una.

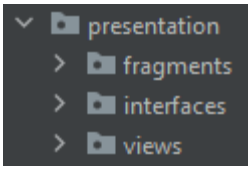


Ilustración 25. Jerarquía capa presentación

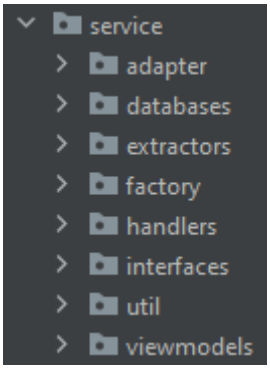


Ilustración 26. Jerarquía capa lógica de negocio con ViewModel

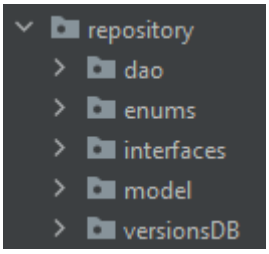


Ilustración 27. Jerarquía capa base de datos

- **Presentation**: Como se puede apreciar en la Ilustración 25 la capa de presentación tiene tres carpetas y su funcionalidades son:
    - **Fragments**: Aloja todo el código referente a los controladores de las vistas llamadas Fragment que son como las vistas Activity, pero tienen la peculiaridad de que los Fragment pueden anidarse en las **Activity**, pero las Activity no pueden alojarse a si mismas.
    - Interfaces: Tal y como dice su nombre esta carpeta contiene todas las interfaces necesarias para poder comunicar las vistas.
    - **Views**: Carpeta que aloja todo el código referente a los controladores de las Activity, las cuales harán de contenedores para mostrar los Fragment.

- **Service**: Como se puede apreciar en la Ilustración 26 la capa de lógica de negocio tiene ocho carpetas, de las cuales la carpeta Viewmodels es esencial para la implementación de la arquitectura MVVM. La funcionalidad de las carpetas son las siguientes:
    - **Adapter**: Contiene los adaptadores necesarios que implementan la funcionalidad del RecyclerView creado y cargan los valores en la vista según le indiquemos.
    - **Databases**: Contiene una clase gestora de la base de datos y las clases gestoras de las distintas tablas existentes en la base de datos, estas últimas se comunican con una DAO que realiza las peticiones a la BD.
    - **Extractors**: Contiene una clase que extrae los nombres de las variables globales de una clase. La he marcado como obsoleta marcando la clase y sus métodos con el decorador @Deprecated, ya que no se va a utilizar más en el proyecto, pero ya que implementé la funcionalidad la dejo para conocimiento futuro de cómo usar la librería “java.lang.reflect”.
    - **Factory**: Contiene fábricas de objetos como puede ser las referencias cruzadas o las que necesita los ViewModel con argumentos de entrada, ya que a no es tan fácil como crear un constructor con argumentos, también es necesario crear un objeto nuevo el cual implemente ViewModelProvider.Factory.
    - **Handlers**: Contiene distintos manejadores de componentes que pueden ser útiles para toda las capas como el manejador de la cámara, de los fragmentos y de los Toast (pequeña notificación que aparece en la parte inferior de la pantalla).
    - **Interfaces**: Contiene interfaces que necesitarán los ViewModel, los contenedores de fragmentos y los RecyclerView.

    - **Util**: Contiene clases estáticas con métodos estáticos los cuales se repetían en muchas partes del código, por lo que se han refactorizado e introducido en clases anteriormente mencionadas.
    - **ViewModels**: Contiene todos los ViewModel implementados en el proyecto necesarios para la arquitectura MVVM.

- Repository: Como se puede apreciar en la Ilustración 27 la capa de persistencia tiene cinco carpetas y su funcionalidades son:
    - **DAO**: También llamada Data Access Object, contiene todas las interfaces con métodos que hacen solicitudes a la BD. El hecho que remarque que son interfaces es importante, ya que he querido separar claramente las DAO de los manejadores de las tablas mencionados en la jerarquía de Service. Si te estás preguntando como se puede hacer consultas desde una interfaz la respuesta son los decoradores. En la interfaz simplemente tienes que escribir los métodos con lo que se supone que van a devolver y los parámetros de entrada y encima de estos métodos se añadirán los decoradores que indicarán a Room cuál es la operación que tiene que realizar.
    - **Enums**: Contiene todos los enum del proyecto, pero no son los enum de Java, son unos especiales que usan “java.lang.annotation” y “androidx.annotation”, ya que los enum clásicos de Java consumen mucha memoria, lo que puede provocar que el gestor de aplicaciones de Android decida terminar el proceso por exceso de recursos.
    - **Interfaces**: Contiene las interfaces con los métodos declarados en el apartado 5.2. Interfaces del modelo, las cuales se implementarán en las clases gestoras de las tablas de la BD
    - **Model**: Contiene los tres objetos básicos y las dos referencias cruzadas mencionadas en el Diagrama 6 a parte de los objetos POJO que utiliza Room sin necesidad de crear tablas adicionales.
    - **VersionsDB**: Contiene un archivo autogenerado por Room el cual muestra por las distintas versiones que ha pasado la BD con el código SQLite generado.

7.2. Room

Como se explica en el apartado de tecnologías, Room es una biblioteca que da una capa de abstracción sobre SQLite, pero se necesitan etiquetar las clases que vayamos a usar con decoradores, todos empiezan con una @.

Antes comenzar la implementación de éste, a modo de introducción, se explica una de ellas con cuatro clases nuevas (Diagrama 15) que Room es capaz de crear con los datos extraídos del modelo.

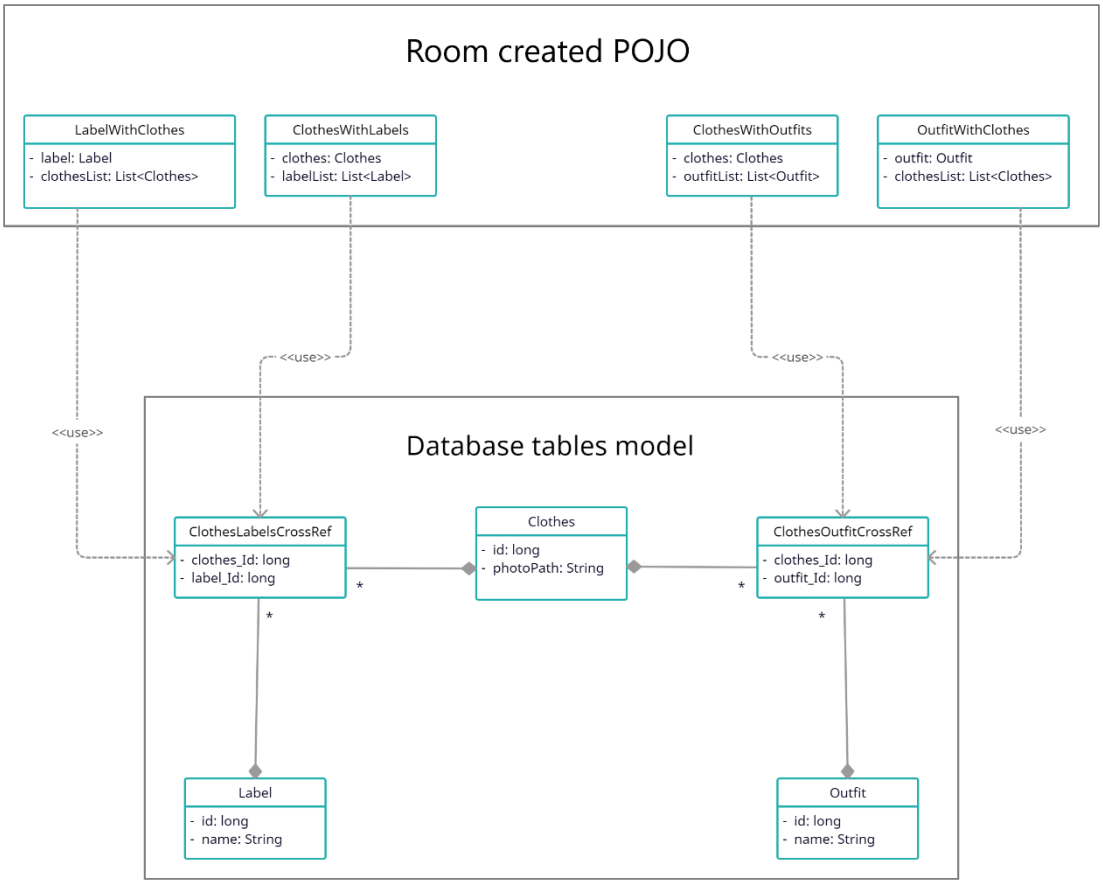


Diagrama 15. Especificación UML de las clases creadas por Room y el modelo de datos actual

Como podemos ver, los nuevos objetos creados con Room únicamente necesitan las CrossRef con las que posteriormente extraerán los objetos deseados gracias a los ids:

- **LabelWithClothes**: Objeto que muestra por cada Label una lista con todas las Clothes relacionadas con el anterior

- **ClothesWithLabels**: Objeto que muestra por cada Clothes una lista con todas las Labels relacionadas con el anterior

- **ClothesWithOutfits**: Objeto que muestra por cada Clothes una lista con todas los Outfit relacionadas con el anterior

- **OutfitWithClothes**: Objeto que muestra por cada Outfit una lista con todas las Clothes relacionadas con el anterior

Ahora voy a explicar con una traza como se ha implementado Room.

```
61      def room_version = "2.2.6"
62      implementation "androidx.room:room-runtime:$room_version"
63      annotationProcessor "androidx.room:room-compiler:$room_version"
64      testImplementation "androidx.room:room-testing:$room_version"
```

Fragmento de Código 1. Declaración de dependencias de Room en build.gradle (:app)

Primero de todo hay que añadir Room a las dependencias del proyecto, Fragmento de Código 1.

```
23  // Important add all pojo that have @Entity to the entities set of @Database
24  @Singleton
25  @Database(entities = {
26              Clothes.class, Label.class, Outfit.class,
27              ClothesLabelsCrossRef.class, ClothesOutfitCrossRef.class
28              },
29              version = 1)
30  public abstract class DataBaseManager extends RoomDatabase {
```

Fragmento de Código 2. Declaración de la clase gestora de la BD

El segundo paso es marcar con el decorador @Database la clase que se va a encargar de gestionar el abrir y cerrar la base de datos, que la clase extienda el componente “RoomDatabase” de la biblioteca “androidx.room” y en el apartado “entities” nombrar las .class que queramos que tengan tablas en la BD, Fragmento de Código 2.

```
@Entity (indices = {@Index("clothes_Id")})
public class Clothes implements IdModel, Parcelable {
    //* Autoincremental primary key
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "clothes_Id")
    private long id;

    @ColumnInfo(name = "uri")
    @NonNull
    private String photoPath;

    public Clothes (String photoPath) { this.photoPath = photoPath; }

    protected Clothes(Parcel in) {...}

    @Override
    public long getId() { return id; }

    @Override
    public void setId(long id) { this.id = id; }

    public String getPhotoPath() { return photoPath; }

    public void setPhotoPath(String photoPath) { this.photoPath = photoPath; }
```

Fragmento de Código 3. Declaración de POJO respaldada con tabla en la BD

El tercer paso es crear la clase del POJO que queramos guardar en la BD y marcarlo con @Entity, el nombre de la tabla será el mismo que el de la clase, si se desea cambiar solo hay que modificar la variable “tableName” dentro de los paréntesis de @Entity, Fragmento de Código 3.

El decorador @Index es opcional e indica el elemento por el que debería indexar la entrada en la BD. Usar @Index aumenta la velocidad de las consultas SELECT, aunque desafortunadamente ralentiza los INSERT y UPDATE.

Posteriormente hay que ponerle los decoradores adecuados a cada variable del POJO, aunque no es necesario ponerles a todas. El indispensable sería el decorador @PrimaryKey el cual indica cual será la clave primaria del POJO, opcionalmente se le puede marcar que el id de la clave primaria sea autogenerado. El decorador @ColumnInfo sirve por si se le quiere dar algún nombre especial a la columna que guarde esa variable en la BD, de lo contrario se usará el mismo nombre que tiene la variable.

Como último paso para la correcta implementación de las POJO hay que crear un constructor con los parámetros de entrada de todas las variables que no puedan ser nulas y tantos getters y setters como variables haya, para que Room sea capaz de acceder y sobrescribir las variables.

```java
@Dao
public interface ClothesDao {
    @Insert (onConflict = OnConflictStrategy.IGNORE)
    long insert(Clothes clothes);

    @Update
    Integer update(Clothes clothes);

    @Delete
    Integer delete(Clothes clothes);

    @Query("SELECT * FROM Clothes WHERE clothes_Id = :clothesId")
    Clothes getClothesById(long clothesId);

    @Query("SELECT * FROM Clothes ORDER BY clothes_Id")
    DataSource.Factory<Integer, Clothes> getAllClothesWithPaging();

    @Query("SELECT * FROM Clothes ORDER BY clothes_Id")
    List<Clothes> getAllClothes();

    @Transaction
    @Query("SELECT * FROM Clothes ORDER BY clothes_Id")
    DataSource.Factory<Integer, ClothesWithLabels> getAllLabelsForEachClothes();
```

Fragmento de Código 4. Declaración de la DAO

El cuarto paso es crear una interfaz o clase abstracta marcada con el decorador @Dao, este decorador le servirá a Room para autogenerar el código necesario para que estos métodos hagan las consultas adecuadas y devuelvan el tipo solicitado, Fragmento de Código 4.

Como se puede apreciar dentro de la interfaz hay cinco decoradores distintos los cuales hacen:

- **@Insert**: Inserta el POJO indicado con el decorador @Entity mencionado anteriormente. Si el método que está decorando solicita que devuelva un tipo long este devolverá la posición en la que se ha insertado en la BD y en caso de que no haya sido posible devolverá -1.

- **@Update**: Dado un POJO que contiene un id ya existente en la BD actualiza sus columnas. Si el método que decora solicita que devuelva un tipo Integer este devolverá el número de filas afectadas por el cambio.

- **@Delete**: Dado un POJO que está contenido en la BD lo elimina. Si el método que decora solicita que devuelva un tipo Integer este devolverá el número de filas afectadas por el borrado.

- **@Query**: Este decorador necesita que escribas una consulta SQL para que el la ejecute y devuelva el tipo solicitado por el método.

- **@Transaction**: Este decorador sirve para que todas las subconsultas se realicen en una única transacción. Esto es útil por si se quiere devolver un POJO nuevo que no contiene tabla en la BD, ya que estos necesitan muchas subconsultas. Este decorador nunca va solo, tiene que acompañar a alguno de los cuatro decoradores mencionados anteriormente.

```
36      //DAO to access database operations
37      public abstract ClothesDao clothesDao();
38
39      public abstract LabelDao labelDao();
40
41      public abstract OutfitDao outfitDao();
42
43      public static DataBaseManager getDB(Context context) {
44          if (dataBaseManager == null) {
45              dbContext = context;
46              dataBaseManager = Room
47                      .databaseBuilder(context, DataBaseManager.class,  name: "ClosetAssistantDB")
48                      //.addMigrations(MIGRATION_1_2)
49                      .allowMainThreadQueries() //Permite hacer consultas en el hilo principal
50                      .fallbackToDestructiveMigration() //Si da error borra todas las tablas
51                      .build();
52          }
53
54          return dataBaseManager;
55      }
```

Fragmento de Código 5. Conseguir una instancia de la base de datos y de las DAOs

El quinto paso es crear un método el cual devuelva una instancia Singleton del gestor de la BD, Fragmento de Código 5, antes mencionado en el segundo paso.

.databaseBuilder() necesita el contexto de la aplicación, el .class del gestor de BD (es decir en la clase que estamos ahora mismo) y el nombre que se le quiera dar a la BD.

.addMigrations() sirve para añadir migraciones a la BD en caso de que se haya cambiado el modelo.

.allowMainThreadQueries() permite ejecutar consultas en el hilo principal de la aplicación, aunque esto está muy desaconsejado, ya que una consulta pesada puede llegar a congelar por segundos la IU, lo cual es totalmente inviable para el usuario.

.fallbackToDestructiveMigration() en caso de que se quiera migrar la base de datos y de problemas la borrará y creará una nueva con la nueva migración.

.build() construye la BD.

Adicionalmente si se ha optado por hacer las DAOs interfaces y no clases abstractas se necesitan tantos métodos abstractos que devuelvan las DAOs como hayan de estas, ya que las clases que gestionen las DAOs necesitarán una instancia de esta.

```java
@Entity(primaryKeys = {"clothes_Id", "label_Id"},
        foreignKeys = {
            @ForeignKey(onDelete = CASCADE, entity = Clothes.class,
                    parentColumns = "clothes_Id", childColumns = "clothes_Id"),
            @ForeignKey(onDelete = CASCADE, entity = Label.class,
                    parentColumns = "label_Id", childColumns = "label_Id")
        },
        indices = {
                @Index("clothes_Id"), @Index("label_Id")
        }
)
public class ClothesLabelsCrossRef {
    public long clothes_Id;
    public long label_Id;

    public ClothesLabelsCrossRef(long clothes_Id, long label_Id) {
        this.clothes_Id = clothes_Id;
        this.label_Id = label_Id;
    }
```

*Fragmento de Código 6. Declaración de referencias cruzadas para un modelo * ... * (muchos a muchos)*

El sexto paso totalmente opcional, ya que solo es necesario si tu modelo necesita relaciones * ... * (muchos a muchos), en este paso hay que volver lo mismo que en el paso 3 (la variable “primaryKeys = {…}” es equivalente al decorador @PrimaryKey) y adicionalmente definir las claves ajenas con el decorador @ForeignKey, Fragmento de Código 6.

Adicionalmente he puesto con la variable “onDelete = CASCADE” que cuando se borre alguna clave ajena que se haga borrado en cascada de todas las filas de la BD que contengan ese valor.

```java
public class ClothesWithLabels {
    @Embedded
    public Clothes clothes;

    @Relation(
            parentColumn = "clothes_Id",
            entityColumn = "label_Id",
            associateBy = @Junction(ClothesLabelsCrossRef.class)
    )
    public List<Label> labelList;

    public Clothes getClothes() { return clothes; }

    public List<Label> getLabels() { return labelList; }
}
```

Fragmento de Código 7. Declaración de un POJO sin respaldo de tabla en la BD

Y, por último, en caso de que se haya decidido crear POJOs nuevos mediante Room, los cuales no están respaldados con una tabla en la BD, hay que poner el decorador @Embedded para permitir recolectar variables anidadas, Fragmento de Código 7.

7.3. ViewModel

Tal y como he explicado en el apartado de tecnologías es un componente que se dedica a guardar valores para comunicarlos entre las vistas. Esta funcionalidad es vital para poder alojar los componentes de Livedata y darles una persistencia en memoria.

El conjunto de ViewModel + Livedata formal el corazón de la arquitectura MVVM la cual voy a exponer en una traza paso a paso de como la he implementado en el proyecto.

```
    def lifecycle_version = "2.2.0"
    implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-extensions:$lifecycle_version"
```

Fragmento de Código 8. Declaración de dependencias de ViewModel + Livedata en build.gradle (:app)

Primero de todo hay que añadir LifeCycle a las dependencias del proyecto, el cual contiene los componentes de ViewModel y Livedata, Fragmento de Código 8.

```
public class AddClothesToOutfitViewModel extends ViewModel {
    private final MutableLiveData<Clothes> clothesToAdd = new MutableLiveData<>();

    public LiveData<Clothes> getClothesToAdd() { return clothesToAdd; }

    public void addClothesToOutfit(Clothes addClothes) { clothesToAdd.setValue(addClothes); }
}
```

Fragmento de Código 9. Implementación de un ViewModel con Livedata

El segundo paso es crear una clase la cual extienda el componente "ViewModel" de la biblioteca "androidx.lifecycle", Fragmento de Código 9.

Posteriormente solo hay que declarar una variable de tipo MutableLiveData<T> la cual podrá ser observada desde la IU o la lógica de negocio.

Una vez declarada la variable global que aloja la funcionalidad de Livedata hay que asígnale un getter y un setter, los cuales expondrá el ViewModel. El setter será ejecutado desde el hilo principal y actualizará el valor guardado en el Livedata, pero se pueden guardar con dos métodos distintos:

- **.setValue(T)**: este método sobrescribe el valor de Livedata, pero si por algún motivo entre los observers hay alguno que no está en el hilo principal de ejecución puede llegar a crearse inconsistencias en los datos, para este caso usar .postValue().

- **.postValue(T)**: exactamente la misma funcionalidad de .setValue(T), pero con la excepción de que si contempla los distintos hilos de ejecución. En caso de que se ejecute múltiples veces .postValue(T) todas intentarán sobrescribir el valor al mismo tiempo, para que no se generen inconsistencias si durante una escritura recibe otra adicional la primera se descartará, así hasta que la última escritura no sea interrumpida.

Entre un .setValue(T) y un .postValue(T) se ejecutaría primero el set y posteriormente el post.

```
addClothesToOutfitViewModel = new ViewModelProvider( owner: this).get(AddClothesToOutfitViewModel.class);
addClothesToOutfitViewModel.getClothesToAdd().observe(getViewLifecycleOwner(), clothes -> {
    if(clothes != null && selectedBodyPart != 0) {
        addToBodyPartsAndAdapter(clothes, selectedBodyPart);
    }
} );
```

Fragmento de Código 10. Referenciación del ViewModel y implementación de observer de Livedata

Por último, si se quiere declarar un observador del componente de Livedata primero tendremos que conseguir la instancia correcta del ViewModel que lo contiene, Fragmento de Código 10.

Para conseguir la instancia correcta de ViewModel se usa ViewModelProvider(), el cual necesita una instancia de Fragment o Activity para funcionar, y posteriormente se usa el método .get() el cual necesita el .class del ViewModel a extraer.

En caso de que se le quiera usar un constructor en el ViewModel hay que crear una nueva clase la cual implemente “ViewModelProvier.Factory” de la librería “androidx.lifecycle”, tal y como se muestra en el Fragmento de Código 11.

Los únicos cambios adicionales serían poner un constructor igual al que acabamos de crear en el ViewModel deseado y en el ViewModelProvider añadir el nuevo constructor de la siguiente forma “new ViewModelProvider(this, new ClothesViewModelFactory(…, …)) .get(ClothesViewModel.class);”

```
13  public class ClothesViewModelFactory implements ViewModelProvider.Factory {
14      private FragmentActivity mFragmentActivity;
15      private IPublishToRecyclerViewWithPaging<Clothes> mAdapter;
16
17      public ClothesViewModelFactory(FragmentActivity fragmentActivity, IPubli
18          mFragmentActivity = fragmentActivity;
19          mAdapter = adapter;
20      }
21
22      @NotNull
23      @Override
24      public <T extends ViewModel> T create(@NotNull Class<T> modelClass) {
25          return (T) new ClothesViewModel(mFragmentActivity, mAdapter);
26      }
27  }
```

Fragmento de Código 11. Creación de constructor para ViewModel

Volviendo al Fragmento de Código 10 podemos apreciar en la línea 103 como se declara el observador del Livedata. Se llama al método del ViewModel que devuelva un tipo LiveData<T> y lo que devuelve se le aplica un “.observe(getViewLifecycleOwner(), elementoRecuperado -> {*código a ejecutar*});”. El segundo parámetro del .observe tiene que ser un Observer, pero he decidido usar una función Lambda, ya que ahorra muchas líneas de declaración que a fin de cuentas es azúcar sintáctico teniendo esta alternativa, Fragmento de Código 11.

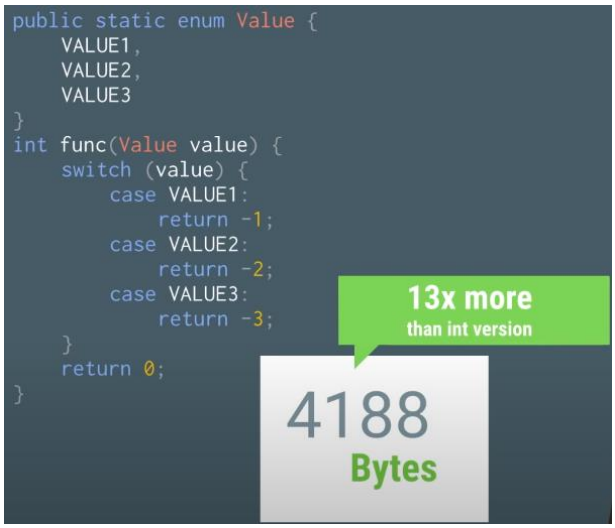
7.4. Mejores enum

Los enum clásicos de Java en Android no son eficientes en consumo de recursos, ya que requieren mucha memoria al aumentar el tamaño del DEX (archivo de código que hace referencia a la aplicación Android compilada) y si la aplicación supera el límite asignado por el sistema el gestor de recursos podría detener la aplicación.

Dada una aplicación con un tamaño de DEX de 2556 Bytes se puede ver la comparación de no usar enums, Fragmento de Código 12 el cual ha añadido 124 Bytes adicionales, y de usar los enums de Java, Fragmento de Código 13 el cual ha añadido 1632 Bytes adicionales.

Como se puede apreciar implementar un enum de java consume hasta 13 veces más espacio que una implementación equivalente sin enum[5], esto sucede porque internamente Java intenta crear un array de elementos contenido en una clase anónima, lo cual es muy caro y por si fuera poco cada entrada del enum cuesta entre 12-16 Bytes adicionales en la memoria de ejecución de la aplicación.

Fragmento de Código 12. Consumo del DEX antes de usar enum



Fragmento de Código 13. Consumo del DEX usando enum

Recapitulado, si se usa muy pocos enum no debería pasar nada, pero si tenemos implementados 20 enums de java como los del Fragmento de Código 13 el DEX de la aplicación pesaría 32640 Bytes adicionales respecto a los 2480 Bytes que pesaría si no se usan enums, lo cual es inaceptable sabiendo que hay mejores alternativas.

Para evitar este problema se pueden usar los enum de “androidx.annotation”, los cuales son mucho más eficientes que los de Java.

A continuación, se muestra una traza de como implementar los nuevos enum.

```
72      //For better Enums
73      implementation 'com.android.support:support-annotations:28.0.0'
```

Fragmento de Código 14. Declaración de dependencias de annotations

Primero de todo hay que añadir annotations a las dependencias del proyecto, Fragmento de Código 14.

```java
public class InflateFragmentEnum {
    @Retention(RetentionPolicy.SOURCE)
    @IntDef({NO_RESTRICTIONS, FRAGMENT_VIEW_HOLDER, OUTFIT_VIEW_HOLDER})
    public @interface SelectedActivity {}

    public static final int NO_RESTRICTIONS = -1;
    public static final int FRAGMENT_VIEW_HOLDER = 1;
    public static final int OUTFIT_VIEW_HOLDER = 2;
}
```

Fragmento de Código 15. Implementación del nuevo enum

El segundo paso es declarar las variables del enum de tipo “public static final int” y usar un nombre acorde a lo que va a hacer referencia en el enum, Fragmento de Código 15.

Una vez declaradas las variables hay que declarar una interfaz la cual tendrá los decoradores @IntDef, este decorador indicará que elementos contendrá la interfaz y el enum.

Ahora para extraer un elemento del enum simplemente hay que poner “InflateFragmentEnum.<nombre de la constante a seleccionar>” y si queremos que un método solo pueda aceptar elementos que estén contenidos en el enum se pondría de la siguiente manera“… … inflateFragment(…, @InflateFragmentEnum.SelectedActivity int selectedActivity)”. Como hemos podido apreciar la interfaz declarada en el enum lo podemos usar como decorador del argumento de tipo int.

7.5. Paging

Tal y como se explica en el apartado de tecnologías, paging es una biblioteca la cual es capaz de extraer grandes cantidades de información de la base de datos y mostrar un pequeño conjunto en la capa de presentación.

Esta librería tiene gran sinergia con Room, por lo que voy a explicar una pequeña traza de cómo se implementa Paging en el proyecto y como trabaja con Room.

```
    def paging_version = "2.1.2"
    implementation "androidx.paging:paging-runtime:$paging_version"
```

Fragmento de Código 16. Declaración de dependencias de paging

Primero de todo hay que añadir paging a las dependencias del proyecto, Fragmento de Código 16.

```
98      @Transaction
99      @Query("SELECT * FROM Label WHERE label_Id IN" +
100             "(SELECT label_Id FROM ClothesLabelsCrossRef WHERE clothes_Id = :clothesId)")
101     DataSource.Factory<Integer, Label> getLabelsOfAClothesWithPaging(long clothesId);
```

Fragmento de Código 17. Implementación de paging con Room

El segundo paso es añadir a la consulta que hace Room "DataSource.Factory<Integer, T>" como parámetro de retorno, el cual devolverá un objeto configurable para seleccionar el tamaño de página entre tantas otras cosas, Fragmento de Código 17.

```
239     @Override
240     public LiveData<PagedList<Label>> getLabelsOfAClothesWithPaging(long clothesId) {
241         return new LivePagedListBuilder<>(clothesDao.getLabelsOfAClothesWithPaging(clothesId),
242                 pageSize: 20).build();
243     }
```

Fragmento de Código 18. Implementación de paging en el gestor de tablas de la BD

Por último, mediante el módulo LivePagedListBuilder<>() podemos crear una lista con paginación con los elementos extraídos de la DAO, únicamente se necesita un "DataSource.Factory" el cual ha sido proporcionado por Room y el tamaño de página deseado, Fragmento de Código 18.

Una vez construido el módulo LivePagedListBuilder<>() si no se desea añadir más configuraciones se puede usar el método .build() y así poder devolver un objeto de tipo LiveData<PagedList<T>> preparado para añadirle un observador desde la IU y actualizala dinámicamente según se vayan recibiendo más páginas.

7.6. Interfaces de la base de datos

En este apartado se expone como se han declarado finalmente las interfaces especificadas en el apartado 5.2. Interfaces del modelo.

```java
public interface IDbHelper<T> {
    void destroyInstance();

    long add(T entity);

    Integer update(T entity);

    Integer delete(T entity);

    T getById(long id);

    LiveData<PagedList<T>> getAllWithPaging();

    List<T> getAll();

    long[] addList(List<T> entityList);

    int deleteList(List<T> entityList);

    long deleteAll();

    Integer getSize();
}
```

Fragmento de Código 19. Implementación de la interfaz IDbHelper<T>

```java
public interface IDbLabel {
    LiveData<PagedList<LabelWithClothes>> getClothesPerLabel();

    List<ClothesLabelsCrossRef> getLabelWithClothesCrossRef(long labelId);

    long[] insertLabelWithClothesCrossRef(Label label, List<Clothes> clothesList);

    void updateLabelWithClothesCrossRef(Label label, List<Clothes> clothesList);

    int deleteLabelWithClothesCrossRef(Label label, List<Clothes> clothesList);

    List<Clothes> getClothesOfALabel(long labelId);

    LiveData<PagedList<Clothes>> getClothesOfALabelWithPaging(long labelId);
}
```

Fragmento de Código 20. Implementación de la interfaz IDbLabel

```java
public interface IDbClothes {
    LiveData<PagedList<ClothesWithLabels>> getLabelsPerClothes();

    LiveData<PagedList<ClothesWithOutfits>> getOutfitsPerClothes();

    List<ClothesLabelsCrossRef> getClothesWithLabelsCrossRef(long clothesId);

    List<ClothesOutfitCrossRef> getClothesWithOutfitsCrossRef(long clothesId);

    long[] insertClothesWithLabelsCrossRef(Clothes clothes, List<Label> labelList);

    void updateClothesWithLabelsCrossRef(Clothes clothes, List<Label> labelList);

    int deleteClothesWithLabelsCrossRef(Clothes clothes, List<Label> labelList);

    long[] insertClothesWithOutfitsCrossRef(Clothes clothes, List<Outfit> outfitList);

    void updateClothesWithOutfitsCrossRef(Clothes clothes, List<Outfit> outfitList);

    int deleteClothesWithOutfitsCrossRef(Clothes clothes, List<Outfit> outfitList);

    List<Label> getLabelsOfAClothes(long clothesId);

    List<Outfit> getOutfitsOfAClothes(long clothesId);

    LiveData<PagedList<Label>> getLabelsOfAClothesWithPaging(long clothesId);

    LiveData<PagedList<Outfit>> getOutfitsOfAClothesWithPaging(long clothesId);
}
```

Fragmento de Código 21. Implementación de la interfaz IDbClothes

```java
public interface IDbOutfit {
    LiveData<PagedList<OutfitWithClothes>> getClothesPerOutfit();

    List<ClothesOutfitCrossRef> getOutfitWithClothesCrossRef(long outfitId);

    long[] insertOutfitWithClothesCrossRef(Outfit outfit, List<Clothes> clothesList);

    void updateOutfitWithClothesCrossRef(Outfit outfit, List<Clothes> clothesList);

    int deleteOutfitWithClothesCrossRef(Outfit outfit, List<Clothes> clothesList);

    List<Clothes> getClothesOfAOutfit(long outfitId);

    LiveData<PagedList<Clothes>> getClothesOfAOutfitWithPaging(long outfitId);
}
```

Fragmento de Código 22. Implementación de la interfaz IDbOutfit

Capítulo 8

Verificación de la base de datos

En este capítulo se describe que son las pruebas unitarias, para que sirven, pruebas unitarias realizadas en la base de datos y se va a mostrar los resultados de estos.

8.1. Que es una prueba unitaria

Las pruebas unitarias son fragmentos de código que su único propósito es verificar el funcionamiento de otros métodos del proyecto.

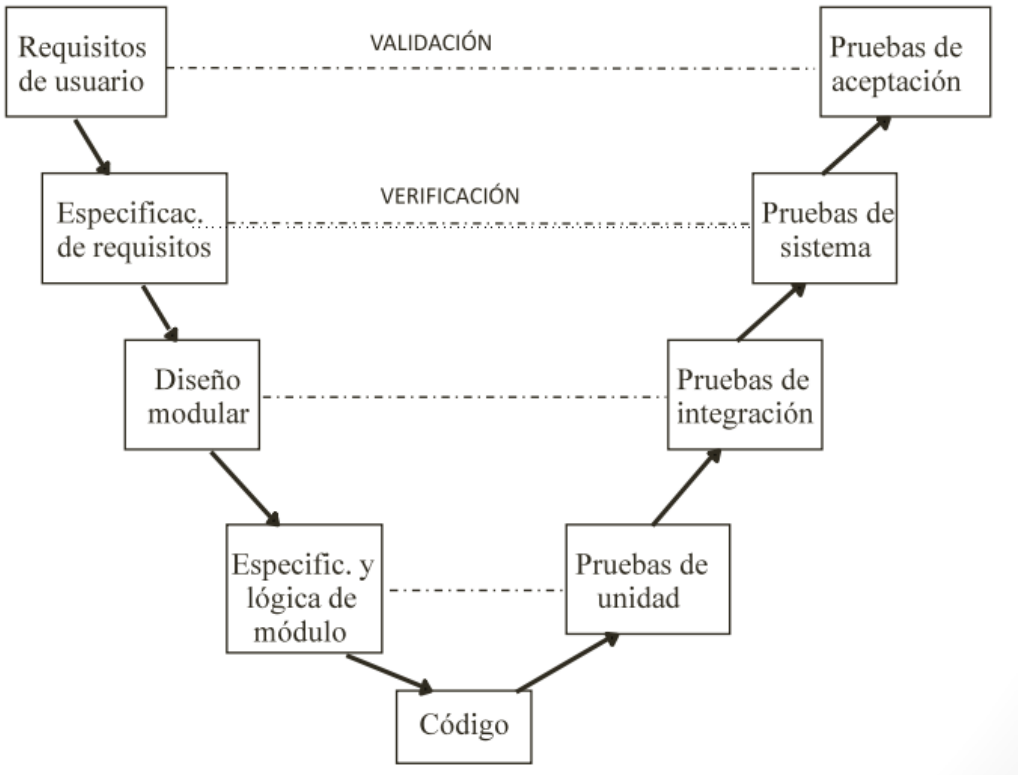


Diagrama 16. Proceso de las pruebas

Como se puede apreciar en el Diagrama 16 principalmente en un proyecto para que sea lo más correcto posible hay que hacer cuatro tipos de pruebas: unitarias, integración, sistema y aceptación. En mi caso he implementado 50 pruebas unitarias

que comprueban todos los métodos de las interfaces expuestas en el punto 5.2. Interfaces del modelo.

En todo proyecto es vital la implementación de pruebas automáticas, ya que tras la implementación o modificación de métodos se pueden ejecutar y dar una confirmación rápida de si lo que se ha implementado está correcto.

Desde mi punto de vista, los test son un arma de doble filo ya que son muy útiles para verificar la funcionalidad de la aplicación, pero si por algún motivo se implementan mal los test pueden llegar a dar falsos negativos, cosa que ralentiza el desarrollo y obliga a comenzar una sesión de depuración de errores para al final descubrir que todo el problema estaba en los test mal implementados que no contemplaban un caso extremo (experiencia propia).

Como corolario de lo expuesto anteriormente, quien vaya a implementar los test tiene que ser alguien experimentado en este campo para reducir al máximo el número de test mal implementados.

8.2. Que es JUnit

JUnit[31] es un framework el cual permite realizar pruebas automáticas mediante el uso de decoradores en los métodos que vayan a ser los test y el uso de los métodos "assert".

A continuación, voy a exponer los decoradores más se usan de JUnit:

- **@RunWith()**: Decora la clase que vaya a contener los test y se le tiene que pasar como argumento el .class que JUnit deseado por ejemplo "AndroidJUnit4.class" de la biblioteca "androidx.test.ext.junit.runners".

- **@Before**: Decora los métodos que se tengan que ejecutar antes de cada test. Remarco que es antes de cada test y no antes de todos los test, para eso está @BeforeClass.

- **@BeforeClass**: Decora los métodos que se tengan que ejecutar antes de que se ejecute ningún test de la clase. Este decorador se suele usar si el código a ejecutar es muy pesado de ejecutar y/o solo se desea ejecutar

[31] https://junit.org/junit4/

una única vez. Este decorador tiene el requerimiento que el método que decora tiene que ser estático.

- **@After**: Decora los métodos que se tengan que ejecutar después de cada test. Remarco que es después de cada test y no después de todos los test, para eso está @AfterClass.

- **@AfterClass**: Decora los métodos que se tengan que ejecutar después de que se ejecute todos los test de la clase. Este decorador tiene el requerimiento que el método que decora tiene que ser estático.

- **@Test**: Decora los métodos que vayan a ser test. Todo test necesita ejecutar un assert de la biblioteca "org.junit.Assert", de lo contrario el test siempre devolverá que se ha completado correctamente a no ser que salte una excepción durante la ejecución.

8.3. Estructura de los test

En la Ilustración 28 se puede apreciar la división de dos carpetas: "testSuite" contiene clases las cuales son capaces de concatenar ejecuciones de clases de test y "unitTest" la cual contiene las clases con las pruebas unitarias a ejecutar.

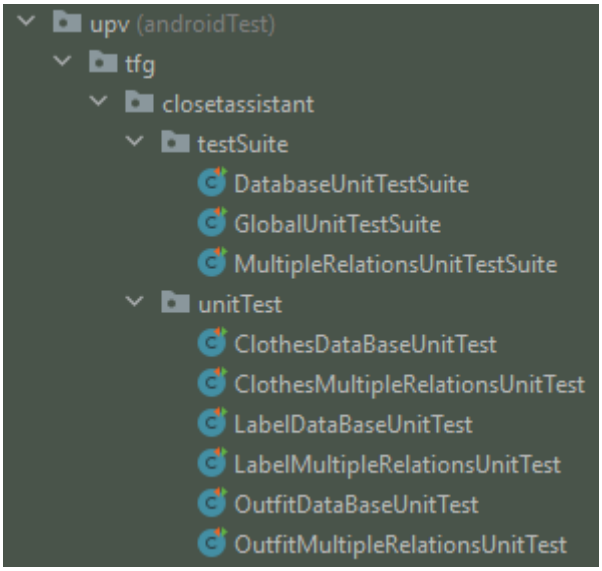


Ilustración 28. Jerarquia de los test

A continuación, voy a mostrar fragmentos de código que exponen cómo funcionan los "testSuite" y las clases de "unitTest", Ilustración 28.

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
        MultipleRelationsUnitTestSuite.class,
        DatabaseUnitTestSuite.class
})
public class GlobalUnitTestSuite {
    @AfterClass
    public static void cleanTest() { DataBaseManager.destroyInstance(); }
}
```

Fragmento de Código 23. Implementación de un Suite inicial para ejecutar el resto de Suites

Si se desea ejecutar todos los test desde una clase se puede usar el decorador @Suite y declarando la clase actual con @RunWith(Suite.class), Fragmento de Código 23. En mi caso he creado un Suite el cual ejecuta otros dos Suites que son:

- DatabaseUnitTestSuite.class: Se especializa en test unitarios de los 3 objetos básicos del modelo que contienen tablas en la BD: Label, Clothes, Outfit; tal y como se puede apreciar en el Fragmento de Código 24
- MultipleRelationsUnitTestSuite.class: Se especializa en test unitarios de los 2 objetos de relaciones cruzadas que tienen tablas en la BD y de los 4 POJO creados por Room mostrados en el Diagrama 15

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
        ClothesDataBaseUnitTest.class,
        LabelDataBaseUnitTest.class,
        OutfitDataBaseUnitTest.class
})
public class DatabaseUnitTestSuite {
}
```

Fragmento de Código 24. Implementación de un Suite específico para ejecutar clases con los test unitarios

```java
@RunWith(AndroidJUnit4.class)
public class ClothesDataBaseUnitTest {
    static DataBaseManager db;
    static ClothesDataBase clothesDataBase;

    @Rule
    public InstantTaskExecutorRule instantTaskExecutorRule = new InstantTaskExecutorRule();


    @BeforeClass
    public static void initializeDb() {
        db = TestUtil.initializeDB(ApplicationProvider.getApplicationContext());

        clothesDataBase = db.getClothesDataBase();
    }

    @After
    public void cleanDb() { db.clearAllTables(); }

    @Test
    public void addClothes() {
        Clothes clothesToTest = TestUtil.createClothes( numberOfClothes: 1).get(0);
        long pos = clothesDataBase.add(clothesToTest);

        //If cant insert value the method returns -1
        assertTrue( condition: pos > 0);
    }
```

Fragmento de Código 25. Implementación básica de una clase de test unitarios con JUnit4

Tras ejecutar los Suites llegaremos a la clase que ejecute los test, para indicar que esta clase es capaz de ello hay que decorarla con @RunWith(<biblioteca JUnit.class>), Fragmento de Código 25.

Una vez declarada la clase que va a ejecutar los test hay que definir las acciones que se deseen ejecutar antes y después de cada test y antes y después de todos los test, solo en caso de que sea necesario.

El decorador @Rule permite declarar reglas a la hora de ejecutar los test, por ejemplo, en mi caso he declarado que todas las consultas asíncronas se ejecuten de forma instantánea y por ello se convierten en síncronas.

Por último, tocaría declarar los test con el decorador @Test en el método deseado. Como he explicado en el apartado 8.2 Que es JUnit, hay que añadirle el assert deseado al método que se le haga el test o dará siempre como test pasado correctamente a no ser que salte una excepción.

8.4. Listado de test

Antes de nada, hay que exponer los resultados de la ejecución de todos los Suite (Ilustración 29), de los que posteriormente se hará una descripción del contenido de cada uno.

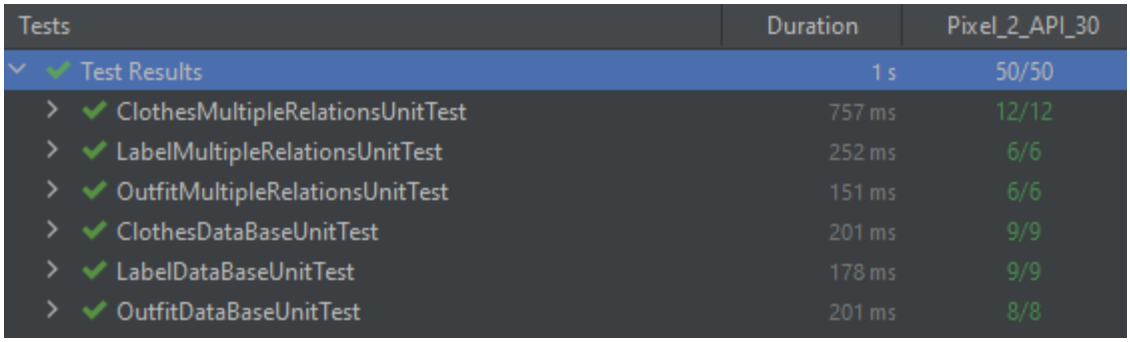


Ilustración 29. Resultado de los test JUnit

- **LabelDataBaseUnitTest**: Clase que verifica los métodos de la interfaz IDbHelper<T> (Fragmento de Código 19) implementados en la clase LabelDataBase.
    - **public void addLabel():** Verifica añadir Label a la BD
    - **public void getLabel():** Verifica solicitar Label de la BD
    - **public void editLabel():** Verifica editar Label de la BD
    - **public void removeLabel():** Verifica eliminar Label de la BD
    - **public void addListLabel():** Verifica añadir una lista de Label a la BD
    - **public void getViewModelListLabel():** Verifica solicitar una lista de Label a la BD mediante el ViewModel
    - **public void removeListLabel():** Verifica eliminar una lista de Label de la BD
    - **public void getAllWithPaging():** Verifica solicitar una lista de Label con paginación a la BD
    - **public void getAll():** Verifica solicitar una lista con todas las Label a la BD

- **ClothesDataBaseUnitTest**: Clase que verifica los métodos de la interfaz IDbHelper<T> (Fragmento de Código 19) implementados en la clase ClothesDataBase.
    - **public void addClothes():** Verifica añadir Clothes a la BD
    - **public void getClothes():** Verifica solicitar Clothes de la BD
    - **public void editClothes():** Verifica editar Clothes de la BD
    - **public void removeClothes():** Verifica eliminar Clothes de la BD
    - **public void addListClothes():** Verifica añadir una lista de Clothes a la BD
    - **public void getViewModelListClothes():** Verifica solicitar una lista de Clothes a la BD mediante el ViewModel

- **public void removeListClothes():** Verifica eliminar una lista de Clothes de la BD
- **public void getAllWithPaging():** Verifica solicitar una lista de Clothes con paginación a la BD
- **public void getAll():** Verifica solicitar una lista con todas las Clothes a la BD

- **OutfitDataBaseUnitTest**: Clase que verifica los métodos de la interfaz IDbHelper<T> (Fragmento de Código 19) implementados en la clase OutfitDataBase.
  - **public void addOutfit():** Verifica añadir Outfit a la BD
  - **public void getOutfit():** Verifica solicitar Outfit de la BD
  - **public void editOutfit():** Verifica editar Outfit de la BD
  - **public void removeOutfit():** Verifica eliminar Outfit de la BD
  - **public void addListOutfit():** Verifica añadir una lista de Outfit a la BD
  - **public void getViewModelListOutfit():** Verifica solicitar una lista de Outfit a la BD mediante el ViewModel
  - **public void removeListOutfit():** Verifica eliminar una lista de Outfit de la BD
  - **public void getAllWithPaging():** Verifica solicitar una lista de Outfit con paginación a la BD
  - **public void getAll():** Verifica solicitar una lista con todas las Outfit a la BD

- **LabelMultipleRelationsUnitTest**: Clase que verifica los métodos de la interfaz IDbLabel (Fragmento de Código 20) implementados en LabelDataBase.
  - **public void insertLabelWithClothesCrossRef():** Verifica añadir referencias cruzadas de un Label a muchos Clothes distintos.
  - **public void updateLabelWithClothesCrossRef():** Verifica la actualización de las referencias cruzadas de un Label a muchos Clothes distintos.
  - **public void deleteLabelWithClothesCrossRef():** Verifica el borrado de las referencias cruzadas de un Label a muchos Clothes distintos.
  - **public void getClothesOfALabel():** Verifica la solicitud de una lista de Clothes que tienen referencias cruzadas con un Label concreto.
  - **public void getClothesOfALabelWithPaging():** Verifica la solicitud de una lista con paginación de Clothes que tienen referencias cruzadas con un Label concreto.
  - **public void getClothesPerLabel():** Verifica la solicitud de una lista del POJO LabelWithClothes

- **ClothesMultipleRelationsUnitTest**: Clase que verifica los métodos de la interfaz IDbClothes (Fragmento de Código 21) implementados en ClothesDataBase.

- **public void insertClothesWithLabelsCrossRef():** Verifica añadir referencias cruzadas de un Clothes a muchos Label distintos.
- **public void updateClothesWithLabelsCrossRef():** Verifica la actualización de las referencias cruzadas de un Clothes a muchos Label distintos.
- **public void deleteClothesWithLabelsCrossRef():** Verifica el borrado de las referencias cruzadas de un Clothes a muchos Label distintos.
- **public void insertClothesWithOutfitsCrossRef():** Verifica añadir referencias cruzadas de un Clothes a muchos Outfit distintos.
- **public void updateClothesWithOutfitsCrossRef():** Verifica la actualización de las referencias cruzadas de un Clothes a muchos Outfit distintos.
- **public void deleteClothesWithOutfitsCrossRef():** Verifica el borrado de las referencias cruzadas de un Clothes a muchos Outfit distintos.
- **public void getLabelsOfAClothes():** Verifica la solicitud de una lista de Label que tienen referencias cruzadas con un Clothes concreto.
- **public void getOutfitsOfAClothes():** Verifica la solicitud de una lista de Outfit que tienen referencias cruzadas con un Clothes concreto.
- **public void getLabelsOfAClothesWithPaging():** Verifica la solicitud de una lista con paginación de Label que tienen referencias cruzadas con un Clothes concreto.
- **public void getOutfitsOfAClothesWithPaging():** Verifica la solicitud de una lista con paginación de Outfit que tienen referencias cruzadas con un Clothes concreto.
- **public void getLabelsPerClothes():** Verifica la solicitud de una lista del POJO ClothesWithLabels.
- **public void getOutfitsPerClothes():** Verifica la solicitud de una lista del POJO ClothesWithOutfit.

- **OutfitMultipleRelationsUnitTest**: Clase que verifica los métodos de la interfaz IDbOutfit (Fragmento de Código 22) implementados en OutfitDataBase.
  - **public void insertOutfitWithClothesCrossRef():** Verifica añadir referencias cruzadas de un Outfit a muchos Clothes distintos.
  - **public void updateOutfitWithClothesCrossRef():** Verifica la actualización de las referencias cruzadas de un Outfit a muchos Clothes distintos.
  - **public void deleteOutfitWithClothesCrossRef():** Verifica el borrado de las referencias cruzadas de un Outfit a muchos Clothes distintos.
  - **public void getClothesOfAOutfit():** Verifica la solicitud de una lista de Clothes que tienen referencias cruzadas con un Outfit concreto.
  - **public void getClothesOfAOutfitWithPaging():** Verifica la solicitud de una lista con paginación de Clothes que tienen referencias cruzadas con un Outfit concreto.

- **public void getClothesPerOutfit():** Verifica la solicitud de una lista del POJO OutfitWithClothes

Capítulo 9

Problemas y soluciones

En este capítulo se explican algunos problemas que han ido ocurriendo durante el desarrollo del proyecto y las soluciones correctivas implementadas.

En función de la dificultad de los elementos que se describen, les he asignado un título en inglés, algunos de ellos con un ápice cómico.

9.1. The forbidden singleton database

Error: Cuando se ejecutan test en la base de datos en memoria pensaba que el @Before daría una nueva instancia de la BD, pero como el método que llamaban es un @Singleton devuelve la misma instancia para todos los test. Si se inserta un valor con una Id hardcoded puede ser rechazado, ya que en el test anterior podría haberse insertado otro elemento con la misma id.

Solución: Añadir un @After el cual realizará 'db.clearAllTables();' después de cada test y se encargará de que todas las tablas estén vacías para el siguiente test.

9.2. Wrong DataBase Manager

Error: Cuando ejecutaba el UnitTestSuite aparentemente todo iba bien, pero en realidad había un problema oculto, que después de ejecutar una clase de test cerraba la base de datos. Aparentemente no es mayor problema ya que al entrar a cada clase de test pido una nueva instancia, pero justamente ahí está el error, ya que puede pasar que ya exista una instancia de la base de datos (db != null) pero esa misma base de datos esté cerrada.

Solución: Añadir una comprobación adicional al solicitar una instancia de la base de datos en la cual aparte de comprobar si es null se mira si está cerrada.

9.3. Can’t obtain LifeCycleOwner in testing class for observers

Problema: No es posible obtener un LifeCycleOwner para observar un objeto desde las clases de testing, ya que no se llega a ejecutar la aplicación para la ejecución de estos.

Solución: Suscribirse para siempre al objeto que se necesite observar, así no te pedirá un LifeCycleOwner.

9.4. Unexpected response of observer

Error: Al realizar test con objetos que tienen LiveData es necesario usar un observer en el cual se pondrán los assert. Pensaba que debía asignar primero un observer al objeto con LiveData y luego añadir los objetos para que los detecte el observer, pero eso lo único que daba eran resultados erróneos en los assert.

Solución: Por cómo funciona los observers cuando se enlazan a un objeto en ese momento comprueban los valores de este y responden al listener onChanged con los valores anteriormente mencionados, en vez de esperar a que se escriban nuevos valores y responder con esos. Simplemente añadiendo los valores deseados antes de enlazar el observer con el objeto a observar soluciona el problema.

9.5. Unexpected response of observer 2: Electric Boogaloo

Error: Hay veces que cuando se comprueba valores con LiveData en un Listener puede llegar a responder dos veces, la primera con los valores deseados y la segunda con todo vacío lo que provocaría que el test de incorrecto.

Solución: Al final del Listener se añade una línea para desuscribirse del mismo para que no haya más respuestas inesperadas.

9.6. Can’t find some test classes

Error: Cuando intentaba ejecutar el test de la clase “ClothesMultipleRelationsUnitTest.java” por algún motivo que desconozco Android Studio no encontraba los test, pero si los ejecutaba individualmente si funcionaba.

Solución: Editar las configuraciones de arranque y añadir una a mano indicando la ruta del test, Ilustración 30.

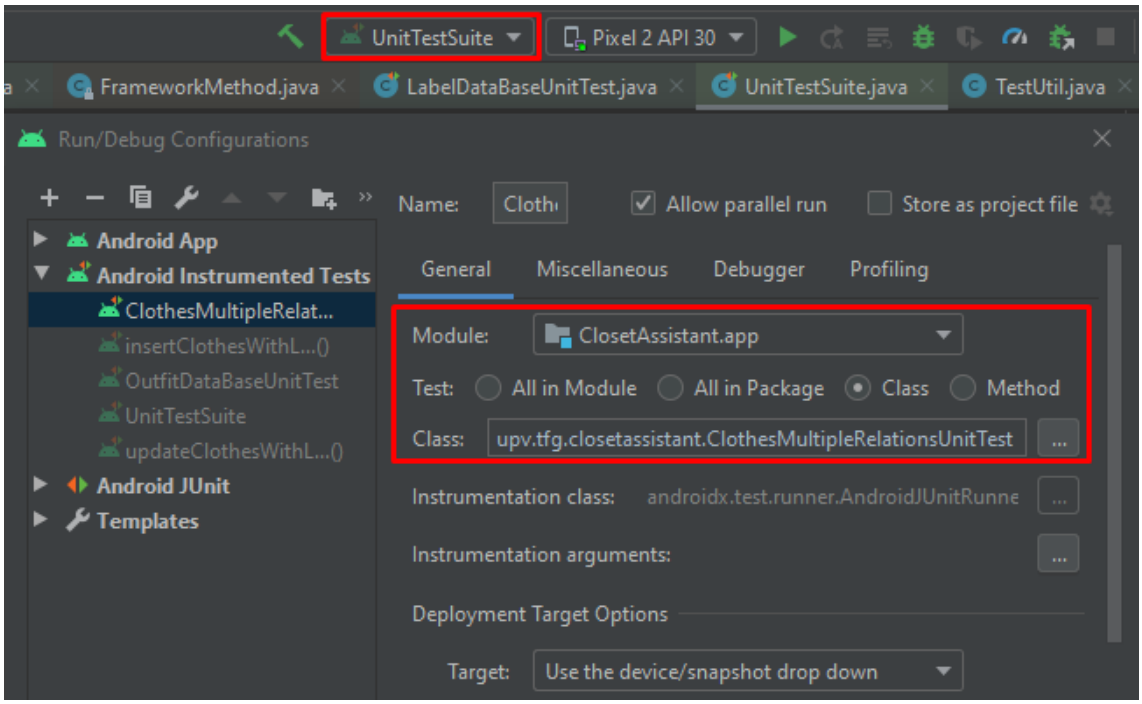


Ilustración 30. Editando la configuración de arranque de los test

9.7. Foreach can’t modify size while looping

Error: Al intentar recorrer dos foreach anidados resulta que no puedes eliminar elementos de la lista mientras estos están en ejecución, Fragmento de Código 26.

```
for (ClothesOutfitCrossRef CRDB: crossRefsDB) {
    for (ClothesOutfitCrossRef CR: crossRefs) {
        if (CRDB.clothes_Id == CR.clothes_Id) {
            crossRefsDB.remove(CRDB);
            crossRefs.remove(CR);
            break;
        }
    }
}
```

Fragmento de Código 26. Foreach no deja modificar elementos que está recorriendo

Solución: Sacar una intersección en otra lista y luego eliminarla de las dos listas, aunque por cuestión de optimización he optado por usar Stream, Fragmento de Código 27.

```
List<ClothesOutfitCrossRef> intersection =
        crossRefsDB.stream().filter(crossRefs::contains).collect(Collectors.toList());

crossRefsDB.removeAll(intersection);
crossRefs.removeAll(intersection);
```

Fragmento de Código 27. Usando Streams para comprobar la sección común de dos listas

9.8. Race Conditions

Contexto: Los ViewModel trabajan con MutableLiveData, a los cuales se les puede hacer ".setValue(…)" o ".postValue(…)", si el MutableLiveData tiene observadores se activarán tantas veces como se haya llamado a setValue, pero si se hace postValue muy rápido se hará skip a todos los post anteriores y se quedará el último. La ventaja del postValue sobre el setValue es que se puede ejecutar esta función desde hilos distintos al principal

Error: se hacía postValue muy rápido y se saltaban ejecuciones vitales para el correcto funcionamiento de la aplicación

Solución: cambiar en el ViewModel el postValue por un setValue

9.9. Inmutable PagedList

Problema: Las PagedList son inmutables por lo que no se pueden aplicar delete(Object o) o deleteAll(ArrayList a)

Alternativa: Al no poderse usar paginación se tendrá que usar una lista normal

Capítulo 10

Refactorizaciones

En este capítulo se van a mostrar algunas las refactorizaciones y buenas prácticas realizadas durante el proyecto.

10.1. Uso de Stream

En múltiples partes del proyecto he estado usando la estructura for o foreach para recorrer listas de datos, pero resulta que las streams tienen mucho más potencial que cualquier otro iterador por lo que he refactorizado la gran mayoría de for / foreach y los he convertido en Stream.

Las Stream pueden llegar a ser complejas de escribir y digerir, pero como recompensa recibimos un iterador más eficiente para listas muy grandes, ya que trabaja mejor con los buses del sistema y tiene una función para paralelizar en múltiples hilos las llamadas que vaya a realizar ".parallel()".

10.2. Generalización

En este apartado voy a exponer algunos ejemplos de generalización que he implementado en el proyecto.

10.2.1. Observer para los ViewModel

En los ViewModel he decidido extraer la declaración del observador, Fragmento de Código 28, ya que es un segmento que se repite en muchas partes del código

Método: ViewModelUtil.getObserver(…);

```
public static <E> Observer<PagedList<E>> getObserver(
        MediatorLiveData<PagedList<E>> mediatorLiveData,
        LiveData<PagedList<E>> query) {
```

Fragmento de Código 28. Declaración de observer genérico para ViewModel

10.2.2. Tratamiento de listas

Para los test hay que arreglar las Ids, ya que cuando se insertan listas de cualquier tipo de objeto en la base de datos no los actualiza, a no ser que los recuperes directamente de la base de datos, Fragmento de Código 29.

Método: TestUtil.fixId(…)

```
public static <E extends IdModel> List<E> fixId(List<E> list, long[] ids) {
```

Fragmento de Código 29. Declaración de método que trabaja con listas genéricas

Adicionalmente ha sido necesario implementar la interfaz del Fragmento de Código 30 para poder usar los métodos necesarios de los objetos genéricos que recibimos.

```
3  public interface IdModel {
4      long getId();
5
6      void setId(long id);
7  }
```

Fragmento de Código 30. Interfaz implementada para el Fragmento de Código 29

Capítulo 11

Conclusiones

Respecto a los objetivos del apartado 1.2 no se han podido completar todos, ya que este TFG solo abarca hasta el primer MVP, por lo que voy a nombrar el estado de cada uno de los objetivos:

- Objetivos completados: 1, 3 y 5

- Objetivos completados parcialmente:
  - 4: Actualmente se puede consultar el tiempo, pero no es capaz de recomendarte conjuntos en función del tiempo que haga
  - 7: Solo se ha desarrollado hasta el primer MVP y las encuestas relacionadas con este.

- Objetivos no completados: 2 y 6

Todos los objetivos parcialmente completados o no completados se realizarán para el segundo MVP, el cual estará en la memoria de mi compañero: “Closet Assistant, tu estilista de bolsillo: desarrollo del front-end”.

Gracias a este proyecto se ha aprendido en profundidad como funciona Android y alguna de sus tecnologías recomendadas por Android Jetpack. Adicionalmente, se ha aprendido a la unificar e implementar dos arquitecturas distintas (arquitectura a tres capas con MVVM).

Respecto a las asignaturas cursadas en la carrera, las que han sido más útiles para la concepción e implementación del backend en Android son:

- DADM (Desarrollo de aplicaciones para dispositivos móviles – 14093) para tener una base de programación en entornos móviles, ya que pese a ser parecido al desarrollo de otras plataformas tiene sus peculiaridades y es necesario pulirlas.

- BDA (Bases de datos y sistemas de información – 11548) al haber usado una base de datos SQL ha sido muy útil para hacer las consultas y conocer cómo funciona internamente SQLite.

- DGS (Diseño y gestión de sistemas de información genómicos – 14104) que pese a no haber usado una base de datos NoSQL ha servido para ver la diferencia entre SQL y NoSQL con casos prácticos, lo cual puede ser útil para el futuro si decidimos usar Firebase con NoSQL

- DDS (Diseño de software – 11565) para conocer y aprender a utilizar diferentes patrones software y arquitectónicos que han resultado ser útiles en la aplicación.

- MES (Mantenimiento y evolución de software – 11569) ha servido para gestionar tener un buen historial de commits en Git, utilizar buenos estilos de programación y para realizar buenos refactoring

- EDA (Estructuras de datos y algoritmos – 11551) ha sido útil para conocer distintos tipos de estructuras de algoritmos necesarias para tratar los datos que se vayan a insertar o provengan de la base de datos.

Capítulo 12

Trabajo futuro

Como se ha explicado en el apartado anterior el proyecto no está cerca de estar finalizado, aún queda un MVP completo por realizar, pero si todo sale como está planeado en un plazo de dos meses estarán totalmente implementados los objetivos propuestos.

A largo plazo se tiene pensado expandir la base de clientes de la aplicación reescribiendo todo en un SDK multiplataforma como podría ser Flutter + Dart. En caso de que se usara Flutter gozaríamos de widgets con Material Design integrado y la posibilidad de exportar a Android, iOs y web la aplicación.

[1] Cinfa, «Infocop Online,» [En línea]. Available: http://www.infocop.es/view_article.asp?id=7365.

[2] M. Cliff, «Women spend SIX MONTHS of their working lives deciding what to wear - and suffer 'wardrobe rage' over trying to choose the right outfit,» MailOnline, 5 Junio 2016. [En línea]. Available: https://www.dailymail.co.uk/femail/article-3626069/Women-spend-SIX-MONTHS-working-lives-deciding-wear.html. [Último acceso: 28 abril 2021].

[3] M. Mena, "Android e iOS dominan el mercado de los smartphones", [En línea] Disponible: https://es.statista.com/grafico/18920/cuota-de-mercado-mundial-de-smartphones-por-sistema-operativo, [Último acceso: 05 de julio de 2021].

[4] F. Richter, "Apple Users More Willing to Pay for Apps", [En línea] Disponible: https://www.statista.com/chart/14590/app-downloads-and-consumer-spend-by-platform, [Último acceso: 05 de julio de 2021].

[5] O. Sodiq, "Why do iOS apps generate more revenue than Android apps?", [En línea] Disponible: https://www.dignited.com/48795/why-do-ios-apps-generate-more-revenue-than-android-apps, [Último acceso: 05 de julio de 2021].

[6] D. Sharma, "Android Paging Library Step By Step Implementation Guide", [En línea] Disponible: https://sharmadhiraj.medium.com/android-paging-library-step-by-step-implementation-guide-75417753d9b9, [Último acceso: 05 de julio de 2021].

[7] Android Developers, "The price of ENUMs", [En línea] Disponible: https://youtu.be/Hzs6OBcvNQE, [Último acceso: 06 de julio de 2021].

Anexo A

Formulario MVP

Encuesta sobre el primer MVP

Datos personales

- Edad
  - Respuesta numérica abierta
- Genero
  - Masculino
  - Femenino
  - Otro

Organización de la ropa

- ¿Cuánta cantidad de ropa crees que tienes?
  - Poca
  - Estándar
  - Mucha
- ¿Actualmente utilizas alguna aplicación para organizar o listar tus prendas de ropa?
  - Sí
  - No
- ¿Te parece útil tener una aplicación para hacerlo?
  - Sí
  - No

Optimización del tiempo

- ¿Aproximadamente, cuánto tiempo gastas por las mañanas escogiendo que ponerte?
  - De 1 a 5 minutos
  - De 5 a 10 minutos
  - De 10 a 15 minutos
  - De 15 a 20 minutos
  - Más de 20 minutos
- ¿Aproximadamente, cuánto tiempo gastas antes de salir a algún evento social (cena, fiesta, etc.) escogiendo que ponerte?
  - De 1 a 5 minutos
  - De 5 a 10 minutos
  - De 10 a 15 minutos
  - De 15 a 20 minutos

    - Más de 20 minutos
- ¿Aproximadamente, cuánto tiempo gastas por las mañanas escogiendo que ponerte usando la aplicación Closet Assistant?
    - De 1 a 5 minutos
    - De 5 a 10 minutos
    - De 10 a 15 minutos
    - De 15 a 20 minutos
    - Más de 20 minutos
- ¿Aproximadamente, cuánto tiempo gastas antes de salir a algún evento social (cena, fiesta, etc.) escogiendo que ponerte usando la aplicación Closet Assistant?
    - De 1 a 5 minutos
    - De 5 a 10 minutos
    - De 10 a 15 minutos
    - De 15 a 20 minutos
    - Más de 20 minutos

Interfaz y comportamiento

- ¿Te ha resultado fácil el uso de la aplicación en general?
    - Sí
    - No
- ¿Te ha resultado fácil añadir una prenda?
    - Sí
    - No
- ¿Te ha resultado fácil editar una prenda?
    - Sí
    - No
- ¿Te ha resultado fácil añadir un conjunto?
    - Sí
    - No
- ¿Te ha resultado fácil editar un conjunto?
    - Sí
    - No
- ¿Cómo preferirías que se organizaran las prendas?
    - Por tipo de prenda (como está ahora)
    - Por estación del año
    - Por tipo de prenda, pero mostrando solo los de la estación del año en la que nos encontramos
- ¿Cómo preferirías que se organizaran los conjuntos?
    - Por tipo de conjunto
    - Por estación del año (como está ahora)
    - Por tipo de conjunto, pero mostrando solo los de la estación del año en la que nos encontramos
- ¿Hay algún tipo de prenda que eches en falta para poder etiquetar alguna prenda?
    - Sí
    - No

- ¿En caso afirmativo, ¿Qué etiqueta o etiquetas añadirías?
    - Respuesta abierta
- ¿Te ha gustado estéticamente la interfaz de la aplicación?
    - Sí
    - No

- ¿Qué icono te gusta más para la aplicación?

- ¿Cambiarias algo de la interfaz de la aplicación?
    - Sí
    - No
- En caso afirmativo, ¿Qué cambiarias?
    - Respuesta abierta
- ¿Te ha dado algún fallo la aplicación?
    - Si
    - No
- En caso afirmativo, ¿Cuál?
    - Respuesta abierta

Valoración final

- En general, ¿Estás satisfecho/a con la aplicación?
    - Sí
    - No
- En caso de no, ¿Por qué?
    - Respuesta abierta
- ¿Tienes pensado utilizarla en un futuro?
    - Sí
    - No
- En caso de no, ¿Por qué?
    - Respuesta abierta
- Del 1 al 10, ¿Qué puntuación le pondrías a la aplicación?
    - 1
    - 2
    - 3
    - 4
    - 5
    - 6

    - 7
    - 8
    - 9
    - 10
- Para finalizar, ¿Qué cambiarias o añadirías a la aplicación?
    - Respuesta abierta

Anexo A

Formulario MVP

Guion para testers de la aplicación Closet Assistant

Muchas gracias por hacer el esfuerzo de probar la aplicación que estamos desarrollando como trabajo de final de carrera. A continuación, se van a enumerar las diferentes acciones que se deben de realizar con la aplicación antes de completar la encuesta que se os ha pasado. Si tenéis cualquier problema o consulta no dudéis en preguntarme.

- Instala y ejecuta la aplicación (para ello debes habilitar la opción de orígenes desconocidos/aplicaciones desconocidas en tu teléfono)
- Añade 5 prendas de ropa de diferente tipo (para obtener mejores resultados te recomendamos poner el formato de imagen en 1:1 en la aplicación de cámara)
- Navega por los diferentes tipos de ropa
- Modifica varias prendas de ropa, cambiado la temporada, la ocasión o el tipo de prenda
- Elimina una prenda de ropa
- Visita la pestaña de mis conjuntos
- Añade 3 conjuntos
- Navega por diferentes tipos de conjuntos
- Modifica las prendas de un conjunto
- Elimina un conjunto
- Visita la pestaña de tiendas
- Navega por diferentes tiendas de ropa
- Visita la pestaña de planificación
- Consulta el tiempo que hace en otra ciudad
- Utiliza la aplicación un par de días para escoger que ponerte