
Profinder: una aplicación móvil para gestionar
servicios de expertos profesionales.
Profinder: a mobile app to manage professional
expert's services



Trabajo de Fin de Grado
Curso 2023–2024

Autor

Jaime Pablo Vázquez Martín

Director

Antonio Sarasa Cabezuelo

Grado en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

Profinder: una aplicación móvil para
gestionar servicios de expertos
profesionales.

Profinder: a mobile app to manage
professional expert's services

Trabajo de Fin de Grado en Ingeniería Informática

Autor

Jaime Pablo Vázquez Martín

Director

Antonio Sarasa Cabezuelo

Convocatoria: *Junio 2024*

Grado en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

27 de Mayo de 2024

Dedicatoria

*A mis padres Pablo y Elena, por siempre
apoyarme y ayudarme, sin ellos nada de esto
habría sido posible.*

Agradecimientos

A Antonio Sarasa, mi tutor, por estar siempre disponible. A Jorge Roselló, que me ayudó a iniciarme en lo que ha sido y será mi viaje en el mundo del desarrollo Android. A Aristides Guimerá (Aristidevs), creador de contenido online sin cuyos vídeos y cursos no habría avanzado tan comodamente en mi proceso de aprendizaje. Y a Marco Antonio Gómez Martín y Pedro Pablo Gómez Martín por facilitar esta plantilla tan útil.

Resumen

Profinder: una aplicación móvil para gestionar servicios de expertos profesionales.

Profinder es una aplicación que ha sido desarrollada para dispositivos con sistema operativo Android. Busca ser un sistema que ayude a profesionales de distintos campos a ponerse en contacto con nuevos clientes, mantener conversaciones con ellos y crear una reputación mediante la calificación que los clientes creen al terminar los servicios realizados por el profesional, a su vez los clientes serán calificados por los profesionales. Los usuarios podrán utilizar la funcionalidad de mapa para comprobar qué profesionales están trabajando por la zona, agilizando así los tiempos de espera. Tanto profesionales como clientes podrán ser agregados a una lista de favoritos para tener fácil acceso a estos si fuera necesario. Tanto profesionales como servicios estarán clasificados dentro de categorías que permitirán una mejor gestión y organización.

La aplicación ha sido desarrollada siguiendo las mejores prácticas posibles, tanto a nivel de código como de arquitectura. Se ha utilizado Kotlin como lenguaje de programación de la aplicación (más moderno, seguro, y eficiente que Java). Para las interfaces se ha utilizado el kit de herramientas de Jetpack Compose diseñado para simplificar el desarrollo de UI.

Palabras clave

profesional, usuario, servicio, mapa, chat, favoritos, categoría, android.

Abstract

Profinder: a mobile app to manage professional expert's services

Profinder is an application developed for devices running the Android operating system. It aims to serve as an app that assists professionals from various fields in connecting with new clients, engaging in conversations with them, and building a reputation through client ratings upon completion of services. Simultaneously, clients will be rated by professionals. Users can utilize the map functionality to check which professionals are working in their area, thus streamlining waiting times. Both professionals and clients can be added to a favorites list for easy access if needed. Professionals and services will be categorized for better management and organization.

The application has been developed following the best possible practices, both in terms of code and architecture. Kotlin has been used as the programming language for the application (which is modern, safer, and more efficient than Java). Jetpack Compose toolkit has been used for the interfaces, designed to simplify UI development.

Keywords

professional, user, service, map, chat, favourites, category, android.

Índice

1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
1.3. Plan de trabajo	3
2. Estado de la Cuestión	5
2.1. Uber	5
2.2. Habitissimo	6
2.3. Booksy	6
2.4. Upwork	7
3. Tecnologías empleadas	8
3.1. Aplicaciones y herramientas generales	8
3.1.1. Android Studio	8
3.1.2. Obsidian	8
3.1.3. Figma	9
3.1.4. Drawio	9
3.1.5. Git, Github y Lazygit	9
3.2. Tecnologías específicas de Android	10
3.2.1. Kotlin	10
3.2.2. Jetpack Compose	11
3.2.3. Testing	13
3.2.4. Dagger Hilt	13
3.2.5. Maps	14
3.2.6. Compose Destinations	15
3.2.7. Coil	15
3.2.8. Material design	15
4. Justificación del diseño y consideraciones técnicas fundamentales	17
4.1. Código abierto	17
4.2. Idiomas	18
4.3. Arquitectura y patrones	18

4.3.1. CLEAN architecture	18
4.3.2. MVI	19
4.3.3. Principios SOLID	21
4.4. Gestión de errores	22
5. Modelo de datos	25
5.1. Descripción del modelo de datos	25
5.2. Firebase	27
5.2.1. Authentication	28
5.2.2. Firestore	28
5.2.3. Realtime Database	29
5.2.4. Storage	30
5.3. Datastore	30
5.4. Uso del patrón Singleton como modelo de persistencia	30
6. Conclusiones y Trabajo Futuro	32
6.1. Conclusiones	32
6.2. Líneas de trabajo futuro	33
Introduction	35
Conclusions and Future Work	39
Bibliografía	41
A. Especificación de requisitos	42
A.1. Actores	42
A.2. Casos de Uso	42
A.2.1. Casos de uso generales	43
A.2.2. Casos de uso de usuarios	50
A.2.3. Casos de uso de profesionales	54
A.2.4. Casos de uso de administrador	61
B. Interfaces de Profinder diseñadas en figma	66
C. Diseño de las bases de datos	68
D. URLs referenciadas en la memoria	70

Índice de figuras

1.1. Diagrama de Gantt del proyecto	3
2.1. Logotipo Uber	5
2.2. Logotipo Habitissimo	6
2.3. Logotipo Booksy	6
2.4. Logotipo Upwork	7
3.1. Función en Java	11
3.2. Función en Kotlin	11
3.3. RecyclerView utilizando Android views (Khan, 2023)	12
3.4. LazyColumn utilizando Jetpack Compose	13
3.5. Ejemplo de inyección de un caso de uso usando Hilt	13
3.6. Ejemplo de módulo de dependencias	14
3.7. Ejemplo de viewmodel inyectado	14
3.8. Uso del <i>composable</i> GoogleMap para dibujar un mapa en la aplicación	14
3.9. Ejemplo de declaración de función como Destination.	15
3.10. Ejemplo de uso de variable navigator.	15
4.1. Ejemplo de estado.	19
4.2. Ejemplo de definición de los <i>intents</i> de una vista.	20
4.3. Representación gráfica del ciclo MVI (Fuentes, 2020).	20
4.4. Ejemplo de primer principio SOLID.	21
4.5. Ejemplo del quinto principio SOLID.	22
4.6. Interfaz Error.	22
4.7. Interfaz Result.	23
4.8. Ejemplo de tipo de error.	23
4.9. Ejemplo de implementación de Result.	24
4.10. Ejemplo de llamada a función implementando Result.	24
5.1. Clase ActorModel.	25
5.2. Clase ServiceModel.	26
5.3. Clase JobModel.	26
5.4. Clase ChatListItemModel.	26

5.5. Clase ChatMsgModel.	27
5.6. Clase LocationModel.	27
5.7. Ejemplo de enumerado (actores).	27
5.8. Captura de pantalla del panel de control de Firebase Authentication.	28
5.9. Captura de pantalla del panel de control de la base de datos Firestore.	29
5.10. Captura de pantalla de la estructura de Realtime Database	29
5.11. Ejemplo de función getData()	31
5.12. Ejemplo de Singleton con datos de usuario	31
6.1. Gantt chart of the project	37
B.1. Splash Screen, Login Screen y Sing Up Screen.	66
B.2. Home Screen, Services Screen (professional) y Chat Screen.	67
B.3. Services Screen (user), Profile Screen y Individual Chat Screen.	67
C.1. Diseño de la base de datos Realtime usando Drawio.	68
C.2. Diseño de la base de datos Firestore usando Drawio.	69

Índice de tablas

A.1. Registro/baja	43
A.2. Modificar datos	44
A.3. Login/Logout	45
A.4. Configurar app	46
A.5. Ver lista de favoritos	47
A.6. Valorar cliente/profesional	48
A.7. Chatear con profesional	49
A.8. Configurar búsqueda	50
A.9. Buscar servicio	50
A.10.Chatear con profesional	51
A.11.Contratar servicio	52
A.12.Contratar servicio	53
A.13.Cambiar estado de profesional	54
A.14.Dar de alta/baja servicio	55
A.15.Modificar servicio	56
A.16.Listar servicios dados de alta	57
A.17.Contestar solicitud de contratación.	58
A.18.Consultar cliente.	59
A.19.Añadir/Modificar/Quitar cliente de lista de favoritos.	60
A.20.Añadir/eliminar/Modificar categorías de servicios ofrecidos.	61
A.21.Consultar datos/estadísticas de profesionales/clientes.	62
A.22.Buscar clientes/profesionales.	63
A.23.Modificar datos de clientes/profesionales/servicios ofrecidos.	64
A.24.Dar de baja usuarios.	65

Introducción

“We are stubborn on vision. We are flexible on details...”
— Jeff Bezos

1.1. Motivación

Profinder nace del concepto de agilizar y simplificar las relaciones entre profesionales que ofrecen un servicio y usuarios que están dispuestos a consumirlo.

Se trata de una aplicación móvil, inicialmente desarrollada para el sistema operativo Android (pese a esto no se descarta la posibilidad de poder hacerla multiplataforma en un futuro, véase el capítulo 6) en la que tanto usuarios como profesionales se podrán registrar, creando una cuenta e interactuar entre ellos. Las funcionalidades para cada tipo de actor varían en distintos aspectos, manteniendo algunas funcionalidades comunes.

A continuación se explica la idea de flujo de publicación, contratación y calificación de servicios de la aplicación:

1. Al registrarse, los profesionales seleccionaran la categoría de profesional a la que pertenecen -esta categoría podrá ser modificada en cualquier momento desde la pantalla de ‘Editar perfil’-.
2. Una vez registrados, tendrán la posibilidad de crear servicios que a su vez estarán clasificados en categorías y podrán ser públicos o privados (activos o inactivos).
3. Los servicios activos aparecerán a los usuarios que podrán solicitarlos ya sea desde la pantalla de listado de servicios o a través del perfil del profesional.
4. Al crear una solicitud un usuario, esta le aparece al profesional que podrá aceptarla o rechazarla. En caso de aceptarla, el trabajo se pone en marcha y en adelante hasta que termine se mostrará como trabajo activo.
5. Una vez terminado el trabajo, el profesional lo marcará como tal y calificará con estrellas (del 1 al 5) al usuario. Para el profesional el trabajo habrá terminado.

6. Por último al usuario le aparecerá la opción de calificar al profesional de la misma forma. Una vez hecho esto, el trabajo se marca como completado y se da por terminado el flujo de servicios.

A parte de este flujo en el que cada actor tiene un papel marcado. Hay cierta funcionalidad común: usuarios y profesionales podrán chatear entre sí usando la pantalla de chat, donde podrán concretar detalles adicionales, además de fechas y horarios concretos. Todos los usuarios y profesionales podrán ser añadidos a listas de favoritos para un fácil acceso a sus perfiles. El perfil de cada actor en la aplicación podrá ser completado con atributos como una descripción o una foto de perfil.

1.2. Objetivos

El mundo del desarrollo android es inmenso, la forma de diseñar interfaces respecto a la programación web muy distinta, y se trata de un mundo que en los últimos años ha estado en constante cambio, cada año salen nuevas tecnologías que dejan obsoletas a las que ya había e incluso hace unos años cambió el lenguaje de programación usado para crear aplicaciones android.

El objetivo principal de este trabajo es aprender muchas de las cosas necesarias para ser un desarrollador android competente, partiendo de cero hasta llegar a ser capaz de crear una aplicación robusta, bonita y escalable en el tiempo.

Los objetivos relativos a la aplicación marcados desde el comienzo se muestran a continuación:

- definición de una especificación de requisitos inicial, sujeta a cambios a lo largo del proceso de desarrollo, en la que se definen los actores y casos de uso de la aplicación (vease el apéndice A).
- Una vez definida la especificación de requisitos, se marcarán una serie de hitos para la entrega de las funcionalidades.

También se han definido una serie de objetivos técnicos que serán llevados a cabo en paralelo a la consecución de los objetivos relativos a la aplicación:

- Aprender kotlin hasta alcanzar un nivel de competencia en el lenguaje apto para el desarrollo.
- Empezar en el desarrollo android utilizando el sistema de vistas ¹, forma clásica de desarrollar interfaces en android, que sirve como primer acercamiento pese a que luego se sustituya por Jetpack Compose.
- Realizar un acercamiento a las arquitecturas, patrones de diseño y buenas prácticas más utilizadas con el objetivo de empezar a hacer proyectos más robustos que se acerquen a la estructura de proyecto final.

¹Artículo que hace una introducción al sistema de vistas: <https://www.studytonight.com/android/introduction-to-views>

- Aprender los fundamentos de Jetpack Compose e ir adquiriendo nuevos conocimientos progresivamente haciendo distintos proyectos de prueba.
- Familiarizarse con las distintos servicios de Firebase para aplicaciones Android que serán usados como backend en la aplicación final.
- Al final del proceso de desarrollo de la aplicación se espera haber obtenido una base de conocimientos sólidos que sirvan como punto de partida de una posible carrera profesional futura.

1.3. Plan de trabajo

Para la consecución de los objetivos descritos en el apartado anterior, se ha establecido al inicio del proyecto un plan de trabajo representado con un diagrama de Gantt, los objetivos se han dividido -igual que en el apartado anterior- en objetivos técnicos y de aplicación ya que la idea inicial es ir cumpliendo los dos tipos de objetivos en paralelo en el tiempo.

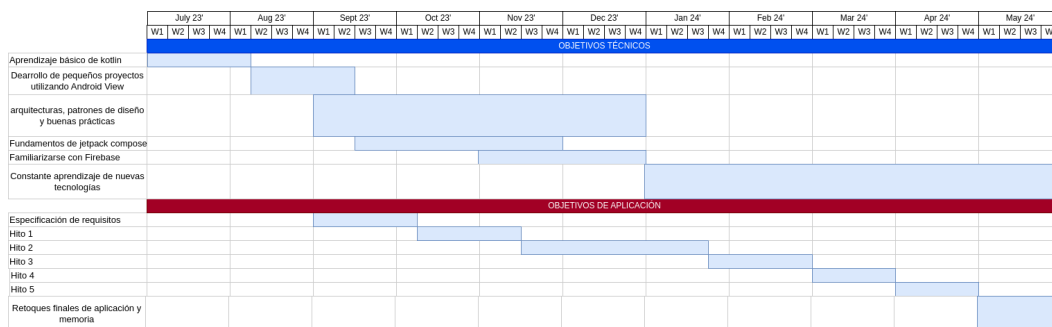


Figura 1.1: Diagrama de Gantt del proyecto

Respecto a los hitos del proyecto, a continuación se muestra una lista con los casos de uso a desarrollar en cada hito:

- Hito 1
 - Registro/baja de usuario.
 - Modificar datos usuario.
 - Login/Logout de usuario.
 - Configurar app.
- Hito 2
 - Cambiar estado de profesional.
 - Dar de alta/baja servicio.
 - Modificar servicio.
 - Listar servicios dados de alta.

- Añadir/eliminar/Modificar categorías de servicios ofrecidos.
- Hito 3
 - Buscar servicio.
 - Configurar búsqueda.
 - Consultar profesional.
 - Consultar cliente.
 - Añadir/Quitar Profesional de lista de favoritos.
 - Añadir/Modificar/Quitar cliente de lista de favoritos.
 - Ver lista de favoritos.
- Hito 4
 - Contratar servicio.
 - Chatear con profesional/cliente.
 - Valorar cliente/profesional.
 - Contestar solicitud de contratación.
- Hito 5
 - Consultar datos/estadísticas de profesional/clientes.
 - Buscar clientes/profesionales.
 - Modificar datos de clientes/profesionales/servicios ofrecidos.
 - Dar de baja usuarios.

Debido a que los casos de uso eran una aproximación inicial, según ha ido avanzando el proceso de desarrollo, algunos casos de uso se han implementado antes, después, o se han descartado por no encajar bien en la aplicación. A pesar de esto, la planificación inicial ha sido de gran utilidad como guía y ha servido para marcar correctamente los plazos de entrega, asimilando el proceso a como sería en un entorno real.

Capítulo 2

Estado de la Cuestión

En este capítulo se ha realizado un análisis de otras aplicaciones en el mercado, viendo sus características generales y comprobando si se considerarían o no competencia de Profinder en el caso de que se tuviera la idea de comercializar en un futuro.

En todos los casos analizados se ha encontrado una diferencia significativa respecto a Profinder: todas son aplicaciones de código cerrado mientras que Profinder ha sido concebida desde sus inicios como una aplicación de código abierto, con todo su código fuente y diseños abiertos al público general para que cualquiera pueda ver y/o modificar el código (véase el apartado 4.1).

2.1. Uber



Figura 2.1: Logotipo Uber

Uber es una aplicación muy conocida que sirve para contratar servicios de desplazamiento en V.T.C. (Vehículo de Transporte con Conductor), el motivo por el que se considera una aplicación relacionada con Profinder es que los servicios se contratan por proximidad, se utiliza la ubicación del usuario -al igual que se hace en Profinder- para encontrar los conductores más próximos, reduciendo así tiempos de espera y costes de desplazamiento innecesario. Uber se ha utilizado como una de las aplicaciones de referencia ya que representa muy bien el concepto que se ha buscado desde el principio con Profinder.

2.2. Habitissimo



Figura 2.2: Logotipo Habitissimo

Habitissimo es una aplicación que opera principalmente en España e Italia, en la que distintos profesionales pueden publicar ofertas de servicios, que los usuarios pueden contratar pidiendo un presupuesto. Es la posible competencia más directa que se ha encontrado.

Esta aplicación está específicamente enfocada al sector de la reforma y reparación, a diferencia de Profinder, que no está enfocada a ningún sector particular, sino que cuenta con distintas categorías que pueden ir ampliándose y ajustándose con el tiempo. Profinder también busca diferenciarse por ser una aplicación global, al ser una aplicación que no depende de una infraestructura por países ya que no ofrece un servicio final sino una ayuda al contacto de usuarios y profesionales en su ámbito local. Asimismo, Profinder tiene un proceso más ágil que no depende de ningún tipo de tercero y en el que se puede tener un contacto más directo entre usuarios y profesionales, disminuyendo así posibles comisiones.

2.3. Booksy



Figura 2.3: Logotipo Booksy

Booksy es una aplicación centrada en el sector de la estética, en la que se pueden encontrar distintos profesionales como peluqueros, maquilladores o masajistas. Operan principalmente en Estados Unidos y cuentan con una batería específica de profesionales.

Las principales diferencias con Profinder es que están enfocados solo en belleza, ofreciendo un rango de categorías menor, a su vez, Booksy tiene una gestión centralizada de profesionales y servicios, no teniendo libertad los primeros para gestionar su trabajo de la forma en que deseen. Esto es también una desventaja respecto a Profinder ya que al tener una gestión descentralizada es una aplicación escalable a nivel internacional.

2.4. Upwork



Figura 2.4: Logotipo Upwork

Upwork es una plataforma que comenzó hace más de 20 años a operar, destinada a poner en contacto empresas con nuevo talento que busque trabajo. Se sitúan como uno de los líderes en su sector, siendo este volumen alto de usuarios una de sus principales ventajas frente a competidores.

El concepto de la aplicación tiene algunas cosas en común con Profinder en el sentido de crear un contacto directo entre contratante y contratado. Sin embargo, Upwork se mueve en el contexto empresarial, dejando de lado el particular, por ello no sería considerada competencia directa y pasaría más a un segundo plano en relación con este proyecto, sirviendo algunas de sus funcionalidades como referencia e inspiración.

Capítulo 3

Tecnologías empleadas

En este capítulo se explican todas la tecnologías, aplicaciones y herramientas que se han utilizado durante el proceso de desarrollo. El capítulo se ha dividido en dos apartados principales, uno dedicado a tecnologías generales y otro dedicado a tecnologías específicas de Android.

3.1. Aplicaciones y herramientas generales

3.1.1. Android Studio

Android Studio es el entorno de desarrollo integrado (IDE) oficial utilizado para el desarrollo de aplicaciones en Android. Esta basado en IntelliJ Idea, un editor multiplataforma con soporte para lenguajes como Kotlin, Java y Scala entre otros.

Android Studio ha sido utilizado a lo largo del proceso completo de desarrollo, siendo de gran utilidad sus modernas funcionalidades como ayuda al correcto flujo de trabajo. Este entorno te permite utilizar emuladores para probar las aplicaciones, sin embargo estos emuladores consumen gran cantidad de recursos del ordenador, esto ha sido un poco problemático en el caso de este proyecto ya que el ordenador con el que se contaba para el desarrollo no tenía recursos suficientes para ejecutar el emulador correctamente, provocando en algunos casos problemas de *lag* y *crashes* ocasionales. A pesar de esto, ha sido posible desarrollar la aplicación usando estos emuladores.

3.1.2. Obsidian

Obsidian es una aplicación de código abierto que sirve como sistema de notas, te permite organizar las mismas con gran cantidad de posibilidades. Cuenta con funcionalidades que han sido de gran utilidad como la referenciación entre notas, vista de grafo y multitud de extensiones desarrolladas por la comunidad.

En este proyecto, Obsidian ha servido como complemento al desarrollo. Se ha utilizado, entre otras cosas, para la planificación de hitos (apuntes personales sobre los casos de uso, fechas de entrega, dudas para tutoría...), reunión de ideas sobre las funcionalidades y documentación del ciclo de vida del proyecto.

3.1.3. Figma

Figma es una plataforma de diseño basada en la web que permite crear y prototipar diseños de interfaces de usuario, con ella ha sido posible crear diseños de una forma rápida y eficaz, que luego han sido materializados en Jetpack Compose.

En este proyecto en particular, figma ha sido utilizada para plasmar las ideas de diseño, sirviendo estas como una referencia no estricta de las interfaces finales. En el apéndice B se han adjuntado capturas de pantalla de todas las interfaces diseñadas. Como la aplicación se ha desarrollado utilizando Material 3 y Dynamic Colors (véase la subsección 3.2.8 para más detalles acerca de Material design) los colores varían dinámicamente en función del tema de cada dispositivo por lo que los colores de los diseños de figma no tienen por qué corresponderse con los colores que luego se han materializado en cada dispositivo.

3.1.4. Drawio

Drawio es una herramienta de código abierto que permite a los usuarios crear una amplia variedad de diagramas (flujo, Gantt...), seleccionada para el proyecto por ser cómoda, personalizable y con gran capacidad de portabilidad, utilizando archivos con extensión .io y dando la capacidad al usuario de cambiar de herramienta con facilidad.

Esta aplicación ha sido utilizada para la realización de todos los diagramas que han sido necesarios durante el proyecto. Ha sido especialmente útil para el diseño de las bases de datos del proyecto (véase el apéndice C con los diseños) ya que, al haberse utilizado bases de datos no SQL, ha sido necesario depurar el diseño en varias ocasiones para evitar la duplicación lo máximo posible.

3.1.5. Git, Github y Lazygit

Para el control de versiones de la aplicación se ha utilizado la conocida herramienta utilizada por la gran mayoría de los desarrolladores de software a nivel mundial: Git. Pese a haber sido un proyecto desarrollado por una sola persona, Git ha sido una herramienta esencial, sirviendo como backup en el desarrollo de nuevas funcionalidades así como herramienta de control general del estado de la aplicación.

El repositorio con el código de la aplicación y el código latex utilizado para la redacción de esta memoria se han almacenado de forma pública en Github, plataforma de desarrollo colaborativo basada en la web que utiliza Git como sistema de control de versiones.

También merece mención en este apartado la herramienta de código abierto Lazygit, interfaz gráfica de terminal para linux que proporciona una forma fácil y visual de interactuar con los repositorios de Git. Está diseñado para simplificar el flujo de trabajo al proporcionar una interfaz gráfica intuitiva dentro de la terminal. Ha sido de gran ayuda para la agilización del control de versiones en conjunción con la extensión de Android Studio para Git.

3.2. Tecnologías específicas de Android

3.2.1. Kotlin

Kotlin es un lenguaje de programación de código abierto creado por JetBrains, es un lenguaje de tipado estático, lo que significa que se puede desarrollar sobre la JVM (Java Virtual Machine) y es totalmente compatible e interoperable con Java lo que facilita la migración desde el primero ¹.

Este lenguaje de programación ha sido seleccionado para el proyecto ya que, a parte de ser el recomendado por Google y ser muy cómodo y útil de usar, con el paso de los años se está consolidando como el lenguaje de referencia para el desarrollo Android, dejando obsoleto el anterior lenguaje de referencia: Java.

3.2.1.1. Kotlin vs. Java

Kotlin empezó a considerarse una opción viable para el desarrollo Android en 2017, cuando Google le empezó a dar soporte. En ese momento, Java era el lenguaje universal utilizado para el desarrollo de aplicaciones en Android. A partir de este punto, en los siguientes años comenzó el debate de si Kotlin era mejor alternativa. En estos últimos años este debate ha quedado prácticamente resuelto en favor de Kotlin, que ha demostrado ser un lenguaje mucho más moderno y útil dejando obsoleto a Java.

Algunas de las características -entre muchas otras- que hacen a kotlin destacar por encima de java son:

- Una sintaxis más moderna, que permite hacer más con menos código.
- Al ser el lenguaje principal utilizado para Android hoy en día ofrece un mayor -y más actualizado- número de bibliotecas, más comunidad y recursos disponibles.
- Inmutabilidad de variables: una variable no puede cambiar a no ser que se indique explícitamente que sí. Esto ahorra muchos posibles errores.
- Kotlin está diseñado ser un lenguaje *null-safe*, es decir ofrece seguridad sobre valores nulos (¡No más NullPointerException!). Esto quiere decir que los valores no pueden ser nulos por defecto, para serlo se debe indicar expresamente con el operador ‘?’.
- Relativo a Android, ofrece soporte nativo para corrutinas y flows, muy útiles en el desarrollo de aplicaciones.

A continuación se muestra una breve comparación del mismo código -que declara una variable, sumándola a un número entero, guardándola en otra variable y mostrándola por consola- en los dos lenguajes para intentar ilustrar algunas de las diferencias entre ambos:

¹Artículo que explica en detalle qué es kotlin: Plain Concepts

```
void javaFunction() {
    Integer exampleVariable = null;

    int exmpleSum = 3 + exampleVariable; //<- NullPointerException

    System.out.println(exmpleSum); //Crash
}
```

Figura 3.1: Función en Java

```
fun kotlinFunction(){
    val exampleVariable: Int = null // ← No puede ser nulo (error de compilación)
                                   // inmutabilidad; no más punto y comas;
    val exampleSum: Int = 3 + exampleVariable //← inferencia del tipo (Int);
                                             //← Sabemos que nunca saltará NullPointerException
                                             // porque exampleVariable no puede ser nulo y es
                                             // immutable

    println(exampleSum) //Sintaxis más sencilla
}
```

Figura 3.2: Función en Kotlin

3.2.2. Jetpack Compose

Jetpack Compose es un moderno kit de herramientas de desarrollo de interfaces de usuario (UI) para Android, creado por Google. Permite a los desarrolladores construir interfaces de usuario de aplicaciones de forma más fácil y eficiente, utilizando un enfoque declarativo. Jetpack Compose fue elegida como tecnología principal para el desarrollo de las interfaces porque suponía un reto apasionante, es una tecnología puntera y muy demandada actualmente en el mundo profesional, además de divertida de usar y con gran cantidad de posibilidades a nivel programático.

Jetpack Composes es una forma más cómoda, sencilla y mejor de desarrollar aplicaciones frente al sistema tradicional de vistas con XML. A continuación se intenta explicar el por qué de esta afirmación.

3.2.2.1. Jetpack Compose vs. Android views

Android Views es el enfoque tradicional para construir interfaces de usuario en aplicaciones Android. Se definen las vistas utilizando archivos XML y se gestionan programáticamente en Java o Kotlin. Esta era la forma de desarrollar interfaces en Android hasta que a partir de 2019, cuando salió Jetpack Compose en modo *preview*, ambas fueron poniéndose a la par. Fue en 2021 cuando Google estableció Jetpack Compose como la forma recomendada de aprender a desarrollar aplicaciones en Android. Hoy en día, Jetpack Compose está totalmente asentado y es un hecho que cada vez va dejando más atrás el sistema de vistas, siendo algo lógico puesto que presenta numerosas ventajas:

- Todo en Kotlin: mientras que usando Views las interfaces se definen en XML y se manipulan programáticamente en kotlin -o Java-, en Jetpack Compose

se unifica todo en Kotlin. Esto reduce enormemente la cantidad de código necesaria y hace el código mucho más entendible.

- Jetpack Compose ofrece un estilo de programación declarativa y reactiva que se adapta mucho mejor al paradigma de programación funcional de Kotlin, apoyándose mucho por ejemplo, en el uso de lambdas para añadir robustez al código.
- Debido a que Jetpack Compose es hoy en día la principal forma de programar interfaces, constanmente van saliendo actualizaciones y nuevas utilidades como el *live edit*, que permite ver los cambios en el diseño de tu aplicación en tiempo real mientras que esta está ejecutandose; o las *preview*s que permiten comprobar el diseño en distintos dispositivos y tamaños.
- Jetpack Compose permite programar para distintos tipos de dispositivos, esto significa que se podría por ejemplo, crear en el mismo proyecto una aplicación de móvil su versión de wearable y su versión de escritorio sin más complicación que la que tuviero adaptar las interfaces a cada tamaño.

Para terminar esta comparativa, a continuación se muestran dos formas de hacer exactamente lo mismo (una columna que renderice y muestre los elementos en pantalla) utilizando las dos formas analizadas en este apartado. Como se puede comprobar lo que se hace en 6 líneas de código en Jetpack Compose necesita de cuatro clases en el sistema de vistas (Adapter, ViewHolder, XML y Activity):



```
<!-- item_layout.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <TextView
        android:id="@+id/text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:padding="8dp" />
</LinearLayout>
```

```
// RecyclerView Adapter
class MyAdapter(private val dataList: List<String>) :
    RecyclerView.Adapter<MyAdapter.ViewHolder>() {

    class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val textView: TextView = itemView.findViewById(R.id.text_view)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.item_layout, parent, false)
        return ViewHolder(view)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = dataList[position]
        holder.textView.text = item
    }

    override fun getItemCount(): Int {
        return dataList.size
    }
}
```

```
// USAGE
val recyclerView = findViewById<RecyclerView>(R.id.recycler_view)
val adapter = MyAdapter(dataList)
recyclerView.adapter = adapter
recyclerView.layoutManager = LinearLayoutManager(this)
```

Figura 3.3: RecyclerView utilizando Android views (Khan, 2023)


```
LazyColumn { this: LazyListScope
    items(state.userList) { this: LazyItemScope it: ActorModel
        MySpacer(size = 8)
        ProfileItem(actor = it, onClick = onSeeProfile)
    }
}
```

Figura 3.4: LazyColumn utilizando Jetpack Compose

3.2.3. Testing

Durante el proceso de desarrollo de Profinder, se han encontrado numerosos desafíos que resolver pero sin duda el testing ha sido el más complicado. La cantidad de cosas que se pueden hacer y conocimientos que aprender son un mundo y debido a los ajustados tiempos del proyecto solo ha sido posible rascar la superficie.

Se han implementado pequeños test unitarios usando librerías como mockk sobre todo del repositorio, en el que se han podido probar todas las funciones. También se han podido probar algunos casos de uso y se han intentado probar los viewmodel, pero debido a la complejidad de sus pruebas no ha dado tiempo ha hacer algo que de verdad valga la pena, por eso los tests se clasificarán en el capítulo 6 como una línea de trabajo futuro.

3.2.4. Dagger Hilt

Todo el proyecto ha sido implementado utilizando inyección de dependencias mediante Dagger Hilt.

Dagger Hilt es un framework que realiza operaciones en tiempo de compilación (autogenerando ficheros) para Android, el cual se encarga de crear y administrar la creación de objetos en toda la aplicación, esto tiene muchas ventajas pero la principal es que facilita la codificación inmensamente ya que evita tener que saber donde se crea cada objeto.

```
@HiltViewModel
class SearchViewModel @Inject constructor(
    private val getAllUsersUseCase: GetAllUsersUseCase
) : BaseViewModel<SearchState, SearchIntent>(SearchState.initial) {
```

Figura 3.5: Ejemplo de inyección de un caso de uso usando Hilt

Se han creado módulos para inyectar todos los servicios remotos de Firebase, así como para el servicio de localización, creando automáticamente instancias que serán utilizadas por todas las interfaces que las requieran.

```
@Module
@InstallIn(SingletonComponent::class)
object NetworkModule {
    new *
    @Singleton
    @Provides
    fun provideDataStoreRepository(
        @ApplicationContext app: Context
    ): DataStoreRepository = DataStoreRepositoryImpl(app)
}
```

Figura 3.6: Ejemplo de módulo de dependencias

Por último, en todas las vistas se han inyectado los *viewmodels*:

```
@Destination
@Composable
fun SearchScreen(
    navigator: DestinationsNavigator,
    searchViewModel: SearchViewModel = hiltViewModel()
) {
```

Figura 3.7: Ejemplo de viewmodel inyectado

3.2.5. Maps

Para la funcionalidad del mapa, se ha usado el SDK de Google Maps para Android. Para utilizarlo es necesario darse de alta en la consola de Google para desarrolladores e introducir un método de pago (aunque el límite de uso gratuito es bastante alto por lo que esto no supone un problema) para recibir la clave de API necesaria.

Una vez obtenida la clave de API ya es posible implementar mapas en la aplicación, para ello se utilizó una adaptación del SDK para el uso en Jetpack Compose creada por múltiples desarrolladores de Google como herramienta *open source*².

```
GoogleMap(
    cameraPositionState = cameraPositionState,
    properties = mapProperties,
    uiSettings = mapUiSettings
) {
    otherLocations.forEach { locationModel →
        MarkerComposable(
            state = MarkerState(locationModel.location),
            onClick = { ... }
        ) { ... }
    }
}
```

Figura 3.8: Uso del *composable* GoogleMap para dibujar un mapa en la aplicación

²Link al repositorio: [android-maps-compose](#)

3.2.6. Compose Destinations

La navegación nativa que ofrece Jetpack Compose no es demasiado cómoda por el momento ya que usa navegación por rutas en formato de *string*, lo cual puede llegar a ser confuso y poco seguro.

Para solucionar este problema se ha encontrado una biblioteca creada por la comunidad llamada Compose Destinations que utiliza como base la navegación nativa pero añade una capa encima que permite estructuración y tipado de los argumentos, ofreciendo una navegación más segura y sencilla de utilizar. Este recurso ha sido muy útil, haciendo el desarrollo con múltiples vistas sustancialmente más sencillo.

Para utilizarlo basta con marcar con el tag *@Destination* las funciones que participen en la navegación, esta etiqueta autogenera el código necesario -usando la navegación nativa- en tiempo de compilación.

```
@Destination
@Composable
fun ServicesScreen(
    navigator: DestinationsNavigator,
    servicesViewModel: ServicesViewModel = hiltViewModel()
) {
```

Figura 3.9: Ejemplo de declaración de función como Destination.

Una vez generado este código, se podrá utilizar la variable *navigator* para navegar a esas pantallas de forma sencilla.

```
navigator.navigate(
    IndividualChatScreenDestination(
        otherNickname = userToSee?.nickname ?: "Guess",
        otherProfilePhoto = userToSee?.profilePhoto,
        otherUid = uidToSee,
        viewProfileViewModel.combineUids(uid, uidToSee)
    )
)
```

Figura 3.10: Ejemplo de uso de variable navigator.

3.2.7. Coil

Coil es una biblioteca creada para Jetpack compose (equivalente a Glide en Android views). Sirve para cargar imágenes en las interfaces, proporciona el componente *AsyncImage* que se ha utilizado para mostrar las fotos de perfil de los usuarios en la aplicación. Se ha optado por Coil en lugar de bibliotecas como Picasso ya que ofrece un rendimiento e integración con Jetpack compose mucho mejor.

3.2.8. Material design

Material Design es un sistema de diseño creado por Google que se utiliza para crear experiencias de usuario coherentes y atractivas en aplicaciones.

Para esta aplicación se ha utilizado Material 3 que ofrece la funcionalidad de Dynamic colors, esto significa que no tiene una paleta de colores estática, sino que dependiendo del tema de cada dispositivo específico, crea una paleta de colores personalizada. De esta forma se consigue una mejor experiencia de usuario, que gracias al algoritmo que se usa para crear la paleta de colores mantiene la armonía entre aplicaciones que usen Material.

Gracias a este sistema, también ha sido posible hacer la aplicación con tema claro y oscuro, para conseguir esto se declaran dos colores para cada elemento de la paleta de colores (dinámicamente) y en función de una variable *booleana* controlada por la aplicación se decide que tema se utiliza.

Capítulo 4

Justificación del diseño y consideraciones técnicas fundamentales

En este capítulo se han desarrollado en profundidad tanto los aspectos técnicos del proyecto como las decisiones que se han tomado en cuanto al rumbo a seguir de la aplicación en distintos niveles.

4.1. Código abierto

El código libre o abierto ¹ (significando esto que el código desarrollado está público para cualquiera que lo quiera ver) es un movimiento socio-tecnológico que comenzó en la década de 1980, cuando Richard Stallman (considerado por muchos el padre del movimiento de código abierto) fundó la *Free Software Foundation* (Fundación de código libre), así como el sistema operativo GNU (*GNU's not Unix*), este creía que el software debía ser creado de forma colaborativa y libremente compartido. Otro gran exponente de este movimiento (aunque con distintas motivaciones) fue el finlandés Linus Torvalds, creador del kernel Linux (GNU-Linux es el sistema operativo con el que se está escribiendo esta memoria y se ha realizado todo el proceso de desarrollo).

En este contexto, se ha intentado desarrollar Profinder siguiendo estos principios e intentando utilizar la mayor cantidad de herramientas de código abierto posible. Algunos ejemplos a parte de los que se han ido mencionando a lo largo de esta memoria pueden ser Firefox, Neovim y Linux entre otros. Asimismo se ha decidido que todo el código de este proyecto sea libre -siendo posible revisarlo, distribuirlo y modificarlo- así como el proceso seguido para hacerlo.

¹La mayoría del contenido en este apartado ha tomado las referencias del libro *The innovators* (Isaacson, 2015)

4.2. Idiomas

El idioma principal utilizado para el desarrollo de Profinder (código, apuntes, *commits*) ha sido el inglés con el objetivo de hacer el proyecto lo mas ‘universal’ posible, por tanto, el idioma original de la aplicación es el inglés. Sin embargo, se ha desarrollado de tal forma que también tiene una versión en español, de la misma forma podría ser traducida a cualquier otro idioma.

4.3. Arquitectura y patrones

4.3.1. CLEAN architecture

La arquitectura CLEAN, originaria del libro Clean architecture (Martin, 2017), es una forma de estructurar el software, dividiendolo en distintas capas que abstraigan el código y separen responsabilidades. Gracias al uso de esta arquitectura, se ha podido crear un proyecto escalable y robusto. También se han ahorrado muchos dolores de cabeza en los distintos procesos de refactorización ya que al haber tenido una estructura ordenada y bien definida, se han evitado posibles efectos secundarios por causa de los distintos cambios.

En Profinder se ha adaptado esta arquitectura a la programación en Android, dividiéndose así el código en cuatro capas explicadas a continuación:

1. core: en esta capa van los recursos comunes que pueden ser accedidos por todas las demás capas, esta es una relación unidireccional ya que core no puede acceder a las demás capas. En este proyecto en particular se ha utilizado para guardar todo el boilerplate necesario para la aplicación del MVI (explicado en la apartado 4.3.2), modelos de datos para pruebas, y utilidades como la clase encargada de formatear las fechas y combinar los identificadores de usuarios.
2. data: en esta capa es donde se gestionan todos los servicios y se preparan para ser enviados a la capa domain (explicada en el siguiente punto). Esta capa tiene toda la lógica de aplicación y es la encargada de decidir de dónde sacar los datos, todas las llamadas de red se hacen en esta capa. En muchos casos, esta capa tiene un modelo de datos propio (los datos tal cual llegan de los *endpoints*) que adapta mediante el uso de *mappers* para ser enviados a la siguiente capa.
3. domain: En esta capa se encuentra la lógica de negocio de la aplicación, sería común a todas las versiones de la aplicación si se hicieran para otros sistemas operativos (IOS, Windows...) a excepción de la adecuación con el lenguaje de programación específico de cada sistema operativo, esto significa que no le ‘importa’ de donde vienen los datos, cuenta con una interfaz (denominada repositorio) en la que se ha definido el contrato con la capa de data y es ahí donde pide los datos que le deben llegar según su propio modelo (mapeados). Es en esta capa donde se definen los casos de uso, que actúan como contratos entre la capa de UI (explicada en el siguiente punto) y el repositorio. Esta implementación abstrae responsabilidades y ayuda a la encapsulación.

4. ui: Esta es toda la capa de interfaz de usuario de la aplicación, realizada utilizando Jetpack compose y siguiendo el patrón arquitectónico MVI (4.3.2). Las vistas como tal (denominadas *Composables*) se comunican con los casos de uso (capa domain) a través de unas clases denominadas *Viewmodels*.

4.3.2. MVI

Debido a que en una aplicación móvil la mayor parte del trabajo se realiza en la interfaz de usuario, es habitual seguir patrones de diseño (mejor denominados: pseudo-arquitecturas). En el caso de Profinder había dos posibilidades: MVVM (Model-View-Viewmodel) o MVI (Model-View-Intent). Se decidió seguir el patrón MVI por ser más organizado y viable para aplicaciones de tamaño grande, a su vez es un patrón que se adapta muy bien al paradigma de programación en kotlin (programación funcional), su única desventaja es que introduce código de relleno para la estructuración (denominado boilerplate), pero esto a su vez ayuda a que sea más entendible por terceros a medida que aumenta el tamaño del proyecto. A continuación se explica en qué consiste esta pseudo-arquitectura y como se ha aplicado en el proyecto:

- **Model:** es un término que puede variar en distintas pseudo-arquitecturas, en MVI es el que representa el estado de los datos, es inmutable y cuando se quiere cambiar algún atributo se destruye y se vuelve a crear con las nuevas modificaciones, esto ayuda a aumentar la testabilidad del código. Por ejemplo podemos tener un estado para la carga de datos (una variable loading) y otro para el usuario (empieza siendo nulo y cuando se han cargado los datos se reescribe el estado con la nueva variable).

```
data class ProfileState(  
    val logout: Boolean,  
    val user: ActorModel?,  
    val error: UiText?,  
    val loading: Boolean,  
    val darkTheme: Boolean,  
) : State {  
    @Jaimevzkz +1  
    companion object {  
        val initial: ProfileState = ProfileState(  
            logout = false,  
            user = null,  
            error = null,  
            loading = true,  
            darkTheme = false  
        )  
    }  
}
```

Figura 4.1: Ejemplo de estado.

- **View:** se encarga de observar el estado y renderizar la vista de forma reactiva. Por ejemplo, en el caso de observar que cambia el valor de la variable loading, dibujaríamos un shimmer (pantalla de carga), cuando el valor de loading se

pusiera a *true* y tras comprobar que no hubiera errores, se dibujaría la vista como tal. Esto hace que la vista sea ‘tonta’ en el sentido de que no tiene que tomar ningún tipo de decisión, solo recibe unos datos en el estado y los pinta.

- **Intent:** se define como una acción (No confundir con Intent de Android) que se lanza cuando el usuario realiza una acción (o el sistema solicita un cambio de estado). Estas acciones se procesan en un *pipe* del viewmodel y van sobrescribiendo el estado. Es la forma que tiene la vista de cambiar los datos, cuando el usuario realiza una acción (como por ejemplo pulsar un botón), la vista lanza un *intent* que se procesa, haciendo los cambios de estado necesarios.

```
sealed class ProfileIntent : Intent {  
    ▲ Jaimevzkz  
    data object Logout : ProfileIntent()  
    ▲ Jaimevzkz  
    data class SetUser(val user: ActorModel?) : ProfileIntent()  
    ▲ Jaimevzkz  
    data class Error(val error: UiText?) : ProfileIntent()  
    ▲ Jaimevzkz  
    data class SetState(val state: ProfState) : ProfileIntent()  
    ▲ Jaimevzkz  
    data object CloseError: ProfileIntent()  
    ▲ Jaimevzkz  
    data class SetTheme(val theme: Boolean): ProfileIntent()  
}
```

Figura 4.2: Ejemplo de definición de los *intents* de una vista.

Esta estructura define un flujo unidireccional de datos: La vista lanza intents que sobrescriben el estado, como consecuencia de este cambio de estado cambia la vista. Este flujo tiene muchas ventajas que ayudan a crear un código entendible, organizado y escalable.

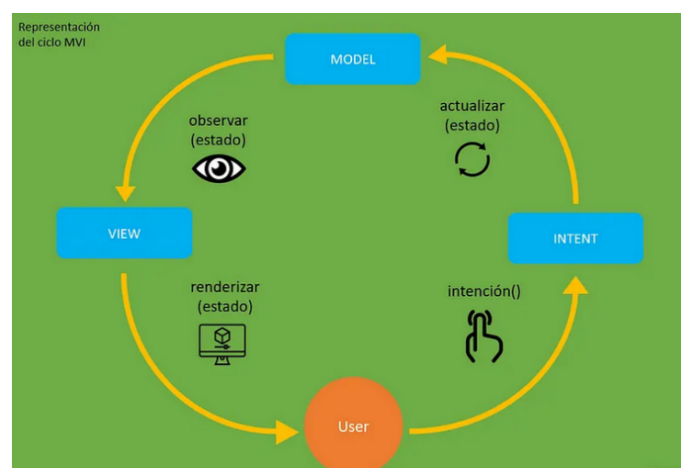


Figura 4.3: Representación gráfica del ciclo MVI (Fuentes, 2020).

4.3.3. Principios SOLID

Los principios SOLID son un conjunto de cinco principios de diseño de software que fueron introducidos en el libro *Clean Code* (Martin, 2008). Están destinados a guiar a los desarrolladores hacia la creación de código fuente limpio, modular, mantenible y extensible. A continuación se ha descrito brevemente cada uno de ellos, adjuntando una captura del código de la aplicación que lo implementa o una explicación de en qué parte del código se ha implementado cuando lo anterior no sea posible:

1. Principio de Responsabilidad Única (*Single Responsibility Principle*): establece que una clase debería tener una única razón para cambiar. En otras palabras, una clase debe tener una sola responsabilidad o función dentro del sistema. En el ejemplo, cualquier clase que implemente la interfaz tendrá la responsabilidad única de implementar el metodo *invoke*.

```
interface ChangeStateUseCase {  
    @Jaimevzkz  
    suspend operator fun invoke(  
        uid: String,  
        state: ProfState  
    ): Result<Unit, FirebaseError.Firestore>  
}
```

Figura 4.4: Ejemplo de primer principio SOLID.

2. Principio Abierto/Cerrado (*Open/Closed Principle*): sostiene que las entidades de software (clases, módulos, funciones, etc.) deberían estar abiertas para su extensión pero cerradas para su modificación.

Este principio se podría ver implementado por ejemplo en la funcionalidad *profile*, en la que se podría añadir todo el código que se deseara para añadir funcionalidad sin tener que cambiar nada de lo ya implementado.

3. Principio de Sustitución de Liskov (*Liskov Substitution Principle*): establece que los objetos de un programa deben ser reemplazables por instancias de sus subtipos sin afectar la integridad del programa.

En Profinder, este principio ha sido implementado por ejemplo en los *viewmodels*, que heredan de la clase *Baseviewmodel*. Cualquier instancia del objeto *Baseviemodel* podría ser sustituido por la implementación de alguna de sus clases hijas sin que cambiara el comportamiento del programa.

4. Principio de Segregación de la Interfaz (*Interface Segregation Principle*): sostiene que es mejor el uso de muchas interfaces pequeñas y concretas, mejor que una monolítica para evitar dependencias innecesarias.

Esto se aplica en los casos de uso, cada caso de uso tiene una interfaz específica y concreta.

5. Principio de Inversión de Dependencias (*Dependency Inversion Principle*): establece que los módulos de alto nivel no deben depender (ni al revés) de los módulos de bajo nivel, ambos deben depender de abstracciones.

Esto se cumple por ejemplo en la implementación *reduce* de los *viewmodel*, esta función no tiene ninguna dependencia específica de la que dependa, sino que se adapta a cada caso recibiendo tipos genéricos.

```
abstract fun reduce(state: S, intent: I): S
```

Figura 4.5: Ejemplo del quinto principio SOLID.

4.4. Gestión de errores

Para conseguir una gestión de errores eficiente y generalizada se han desarrollado una serie de clases ² que en conjunto sirven como sistema bien estructurado que proporciona una forma cómoda de gestionar errores, respetando todos los principios de arquitectura limpia mencionados en el apartado 4.3.1. A continuación se explican todas las clases que se han creado, es posible que solo con la explicación que viene a continuación no se entienda el sistema en su totalidad, pero el lector tendrá disponible el código fuente para verlo aplicado y si es necesario clonar el repositorio para probarlo haciendo cambios.

En primer lugar se ha creado la interfaz *Error* que no hará nada pero servirá para identificar todos los tipos de errores declarados.

```
sealed interface Error
```

Figura 4.6: Interfaz Error.

En segundo lugar se ha creado la interfaz *Result* (se ha llamado así a pesar de que ya haya una clase con este nombre en la biblioteca estandar por considerarse el más apropiado). Esta es la interfaz principal que utilizan todas las funciones de la aplicación que devuelvan datos. Tiene dos parámetros genéricos que cambiarán en cada caso específico, uno para los datos devueltos y otro para el tipo de error, también cuenta con los dos posibles resultados que pueda devolver una función que implemente esta interfaz:

- **Success:** para el caso de éxito. Llevará como parámetro los datos devueltos.
- **Error:** distinto de la interfaz explicada anteriormente, será lo que se devuelva en caso de error en la llamada a función y llevará como parámetro una clase que implemente la interfaz *Error*.

²Este sistema de gestión de errores se ha realizado siguiendo el propuesto en este tutorial: Philipp Lackner on Error handling y ha sido personalizado según las necesidades específicas de la aplicación.

```

typealias RootError = Error
@Jaimevzkz
sealed interface Result<out D, out E: RootError> {
    @Jaimevzkz
    data class Success<out D, out E: RootError>(val data: D): Result<D, E>
    @Jaimevzkz
    data class Error<out D, out E: RootError>(val error: E): Result<D, E>
}

```

Figura 4.7: Interfaz Result.

Result y **Error** son las interfaces base del sistema. Lo siguiente será crear todas las interfaces que sean necesarias -e implementen *Error*- para cada tipo de error. En Profinder solo ha sido necesario crear una que englobara todos los errores relativos a Firebase, sin embargo, es un sistema escalable a futuro para múltiples tipos de errores.

Dentro de esta interfaz se declaran todos los tipos de errores (relativos a Firebase, por ejemplo) como clases enumeradas, consiguiendo así que cada uno sea un enumerado que se gestiona de forma independiente en las distintas capas de la aplicación.

```

sealed interface FirebaseError: Error {
    @Jaimevzkz *
    enum class Authentication : FirebaseError {
        WRONG_EMAIL_OR_PASSWORD,
        USERNAME_ALREADY_IN_USE,
        ACCOUNT_WITH_THAT_EMAIL_ALREADY_EXISTS,
        UNKNOWN_ERROR,
    }
    new *
    enum class Realtime : FirebaseError {
        NULL_REALTIME_DATA,
        RECENT_CHAT_UPDATE_ERROR,
        ERROR_GETTING_RECENT_CHATS,
        REALTIME_ACCESS_INTERRUPTED,
        UNKNOWN_ERROR,
    }
}

```

Figura 4.8: Ejemplo de tipo de error.

Con todo lo anterior, la gestión de errores se convierte en algo sencillo, las funciones deberán devolver un tipo *Result*:

```
override suspend fun login(email: String, password: String): Result<ActorModel, FirebaseError>
```

Figura 4.9: Ejemplo de implementación de Result.

Y en las llamadas a función se podrá gestionar si hay un error dividiendo los casos con una expresión *when*:

```
when(val login : Result<ActorModel, FirebaseError> = loginUseCase(email, password)){  
    is Result.Success → {  
        withContext(Dispatchers.IO) { saveUidDataStoreUseCase(login.data.uid) }  
        dispatch(LoginIntent.Login(login.data))  
    }  
    is Result.Error → dispatch(LoginIntent.Error(login.error.asUiText()))  
}
```

Figura 4.10: Ejemplo de llamada a función implementando Result.

Capítulo 5

Modelo de datos

En este capítulo se explicará todo el modelo de datos de la aplicación. En primer lugar, se hará una descripción de todas las estructuras de datos utilizadas, explicándolas brevemente para dar un poco de contexto, sin embargo, siempre se recomienda ver y tocar el código para entenderlo en profundidad. En segundo lugar se explicarán todos los servicios utilizados para mantener la persistencia: Firebase (principal herramienta utilizada como base de datos remota), Datastore (se ha utilizado para guardar en local preferencias del sistema) y por último se explicará como se ha utilizado el patrón Singleton para evitar tener que usar una base de datos local como Room, logrando así una aplicación completa que pueda interactuar con otros dispositivos a través de la red.

5.1. Descripción del modelo de datos

- **ActorModel:** es la estructura de datos que almacena todos los datos de usuario, se usa la misma tanto para usuarios como para profesionales ya que admite campos nulos para aquellos atributos que solo corresponden a un tipo de actor, en esta estructura se guarda también el tipo de actor, la lista de favoritos (lista con otros *ActorModel*), la calificación y el número de calificaciones.

```
data class ActorModel(  
    val nickname: String,  
    val uid: String,  
    val firstname: String,  
    val lastname: String,  
    val description: String? = null,  
    val actor: Actors,  
    val profession: Professions? = null, //Only for professionals  
    val state: ProfState? = null, //Only for professionals  
    val profilePhoto: Uri? = null,  
    val favourites: List<String> = emptyList(),  
    val rating: Double? = null,  
    val reviewNumber: Int? = null  
)
```

Figura 5.1: Clase ActorModel.

- **ServiceModel**: guarda el modelo de un servicio, cuenta con distintos campos como la categoría del servicio, si está activo o no, y el propietario del servicio (un *ActorModel*).

```
data class ServiceModel(  
    val sid: String,  
    val uid: String,  
    val name: String,  
    val isActive: Boolean,  
    val category: Categories,  
    val servDescription: String,  
    val owner : ActorModel,  
    val price: Double  
) {
```

Figura 5.2: Clase ServiceModel.

- **JobModel**: esta clase guarda tanto solicitudes de trabajo como trabajos activos, en un principio se hicieron dos clases separadas para esto, pero se comprobó que la funcionalidad era la misma y se unificó en una sola clase para evitar duplicaciones de código, relacionan el profesional, el usuario y el servicio.

```
data class JobModel(  
    val id: String,  
    val otherNickname: String,  
    val otherUid: String,  
    val serviceId: String,  
    val serviceName: String,  
    val price: Double,  
    val isRatingPending: Boolean = false  
)
```

Figura 5.3: Clase JobModel.

- **ChatListItemModel**: representa un ítem en la lista de chats recientes, guarda solo los campos necesarios para mostrar al usuario sin tener que volver a acceder a base de datos para sacar el *ActorModel* completo.

```
data class ChatListItemModel(  
    val profilePhoto : Uri?,  
    val nickname: String,  
    val uid: String,  
    val timestamp: Long,  
    val lastMsg: String,  
    val unreadMsgNumber: Int,  
    val lastMsgUid: String  
)
```

Figura 5.4: Clase ChatListItemModel.

- **ChatMsgModel:** esta clase representa un mensaje en el chat, guarda el contenido del mensaje, un *timestamp* para saber el momento exacto en el que se envió de cara a ordenarlo en la lista de mensajes, así como para mostrarle la hora al usuario y si el usuario es propietario (lo ha enviado él) o no (lo ha recibido) del mensaje.

```
data class ChatMsgModel(  
    val msgId: String,  
    val msg: String,  
    val timestamp: Long,  
    val isMine: Boolean  
)
```

Figura 5.5: Clase ChatMsgModel.

- **LocationModel:** representa la localización de un usuario, solo guarda los campos que es necesario mostrar en el mapa, la localización y el id del usuario (para poder extraer sus datos si fuera necesario).

```
data class LocationModel(  
    val uid: String,  
    val nickname: String,  
    val location: LatLng  
)
```

Figura 5.6: Clase LocationModel.

- **Enums:** También se han usado una serie de enumerados para guardar cosas como los tipos de actores, las categorías de servicios, profesiones, etc. Esto hace que sea muy sencillo añadir nuevos en caso de necesidad sin tener que cambiar código existente, solo añadir nuevo (*Open/Closed Principle*, véase el apartado 4.3.3).

```
enum class Actors(  
    val icon: ImageVector,  
    @StringRes val string: Int  
) {  
    User(icon = Icons.Outlined.Person, "User"),  
    Professional(icon = Icons.Outlined.Engineering, "Professional")  
}
```

Figura 5.7: Ejemplo de enumerado (actores).

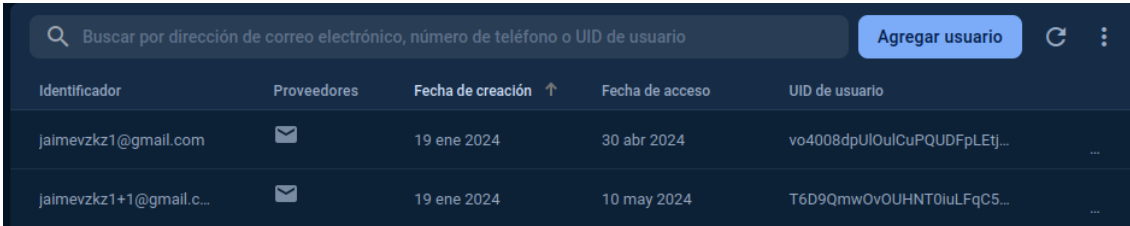
5.2. Firebase

Firebase es una plataforma web desarrollada por Google que ofrece una amplia gama de servicios para ayudar a los desarrolladores a crear y mejorar aplicaciones. En Profinder, se ha utilizado como *backend* para toda la aplicación y ha aportado

la capacidad de plasmar el modelo de datos de la aplicación a una base datos, así como autenticación y almacenamiento para fotos. A continuación se ha explicado cada servicio utilizado.

5.2.1. Authentication

Firebase Authentication es un servicio de autenticación que ahorra todo el proceso de gurdado y cifrado de contraseñas, también gestiona las sesiones de cada usuario, permitiendo mantener la sesión iniciada. Tiene compatibilidad para configurar diversos métodos de autenticación, sin embargo, en esta aplicación solo se ha considerado oportuno implementar la autenticación con *email* y contraseña aunque si se quisiera sería sencillo añadir otros métodos.



The screenshot shows the Firebase Authentication console interface. At the top, there is a search bar with the placeholder text 'Buscar por dirección de correo electrónico, número de teléfono o UID de usuario' and a blue button labeled 'Agregar usuario'. Below the search bar is a table with the following columns: 'Identificador', 'Proveedores', 'Fecha de creación', 'Fecha de acceso', and 'UID de usuario'. The table contains two rows of user data.

Identificador	Proveedores	Fecha de creación	Fecha de acceso	UID de usuario
jaimevzkz1@gmail.com		19 ene 2024	30 abr 2024	vo4008dpUIOuICuPQUDFpLEtj...
jaimevzkz1+1@gmail.c...		19 ene 2024	10 may 2024	T6D9QmwOvOUHNT0iuLFqC5...

Figura 5.8: Captura de pantalla del panel de control de Firebase Authentication.

5.2.2. Firestore

Firebase Firestore ha sido la principal base de datos del proyecto, es de tipo no SQL lo cual ha presentado algunas ventajas pero también muchos desafíos ya que en este tipo de bases de datos es muy fácil caer en la duplicación de datos y ha sido necesario gastar mucho tiempo en pensar la mejor forma de implementarla.

Esta base de datos ha sido dividida en tres colecciones explicadas a continuación (véase el apéndice C con el diagrama del diseño):

- **users:** cada documento de esta colección se guarda con un id generado automáticamente y es donde se almacenan los datos de cada usuario, así como los *jobs* y *requests*.
- **services:** cada documento de esta colección se guarda con un id generado automáticamente y es donde se almacenan los datos de cada servicio.
- **locations:** cada elemento de esta colección representa la localización de un usuario, se generan con un id igual al id del usuario y guardan el *nickname* a parte de la localización

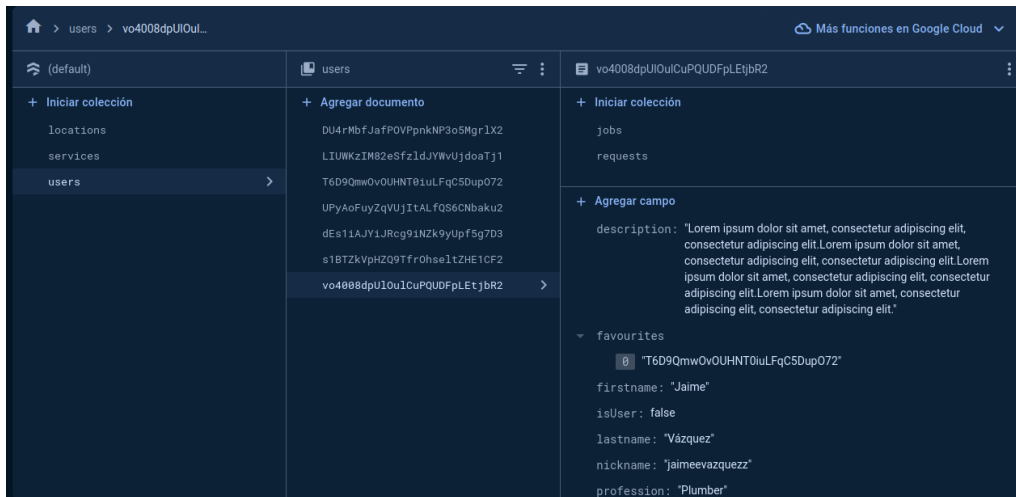


Figura 5.9: Captura de pantalla del panel de control de la base de datos Firestore.

5.2.3. Realtime Database

Realtime database es una base de datos de baja latencia, menos sofisticada que Firestore pero cuyas funcionalidades han encajado perfectamente con las necesidades del proyecto. Se ha utilizado para toda la funcionalidad de chat ya que al ser los mensajes en tiempo real, era necesario que la latencia fuera mínima.

En Realtime, los datos se guardan en formato JSON (*JavaScript Object Notation*). Su estructura se ha dividido en la lista de chats recientes (conteniendo atributos como el último mensaje, los participantes, la hora y el número de mensajes sin leer por cada conversación entre dos actores de la aplicación) y los chats en sí (conteniendo cada mensaje de una conversación con la hora en que fue mandado).

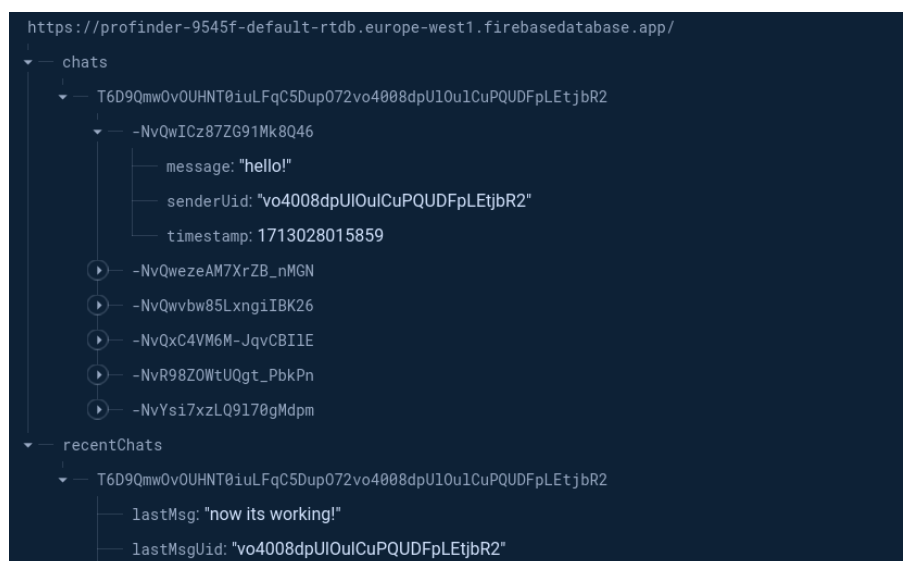


Figura 5.10: Captura de pantalla de la estructura de Realtime Database

5.2.4. Storage

Firebase storage es un servicio de almacenamiento en la nube. Se ha utilizado para almacenar las fotos de perfil de los actores de la aplicación. Cada foto de perfil se guarda en una carpeta cuyo nombre es el id del actor y se accede a ella a través de una uri autogenerada que se procesa en la aplicación usando Coil.

5.3. Datastore

Data store es un sistema de almacenamiento persistente que permite guardar pares clave valor en la aplicación para un acceso rápido a través de corrutinas. En el caso de Profinder se ha utilizado para guardar 2 cosas:

- **Tema de la aplicación:** de tal forma que cuando se cambia su valor, devuelve un *flow* que indica a la Main Activity el tema que debe utilizar. Este tema se mantiene en caché por lo que se podría cerrar la aplicación y al abrirla se recordaría el tema establecido.
- **El id del usuario iniciado:** De tal forma que cuando se necesiten datos del usuario desde cualquier parte de la app se puede conseguir el id (que es único en base de datos) según necesidad, esto servirá para un acceso más rápido en Firestore debido a que es una base de datos que se indexa por este campo.

5.4. Uso del patrón Singleton como modelo de persistencia

En la programación Android, la persistencia de datos es muy importante ya que al contrario que en otro tipo de programas, cada poco tiempo se borran todos los datos no persistidos para evitar que la aplicación ocupe demasiado espacio en el dispositivo. En este contexto es donde se presenta el problema.

A lo largo del desarrollo de otras aplicaciones (en preparación a esta) se encontró que el uso de una base de datos local suponía una carga de espacio en la aplicación y en tiempo de compilación que la ralentizaba, más allá de esto, las necesidades de Profinder implicaban que los datos fueran actualizados cada no demasiado tiempo puesto que tanto servicios; como favoritos; como los datos de usuario están diseñados para cambiar con frecuencia entre los distintos actores de la aplicación. Por estos motivos se decidió buscar un sistema alternativo que se adecuara a las necesidades de la aplicación: actualización frecuente y poca carga en memoria sin tener que estar accediendo constantemente al servicio remoto (Firestore) ya que esto supondría un coste económico.

Esta solución se ha implementado siguiendo el patrón *Singleton*. Los datos a persistir se almacenan en una clase que los guarda de la siguiente forma: si los datos están cacheados los devuelve, si no, los saca del servicio remoto, los cachea y los devuelve. De esta forma la próxima vez que se accedan los datos, estarán cacheados y no hará falta llamar al servicio remoto.

```
suspend fun getData(uid: String = ""): Result<ActorModel, FirebaseError.Firestore> {  
    return if (cachedUser == null) {  
        | fetchDataFromFirestore(uid) //get user from firestore  
    } else {  
        | Result.Success(cachedUser!!) //user cached locally  
    }  
}
```

Figura 5.11: Ejemplo de función getData()

Para acceder a esta función es donde creamos el Singleton que será accedido desde cualquier lugar donde se necesiten los datos:

```
companion object { //Singleton  
    private var instance: UserDataSingleton? = null  
  
    👤 Jaimevzkz  
    fun getUserInstance(repository: Repository): UserDataSingleton {  
        | if (instance == null) {  
        |     | instance = UserDataSingleton(repository)  
        | }  
        | return instance!!  
    }  
}
```

Figura 5.12: Ejemplo de Singleton con datos de usuario

Capítulo 6

Conclusiones y Trabajo Futuro

En este capítulo han sido plasmadas las conclusiones, así como las posibles líneas de trabajo futuro del proyecto.

6.1. Conclusiones

Desde que se planteó este trabajo de fin de grado en Julio de 2023, hasta la fecha en que se están redactando estas conclusiones han pasado aproximadamente 11 meses. Ha sido un viaje en el que se partía prácticamente de cero en cuanto a concimientos específicos de Android y en el que no se han parado de aprender conceptos nuevos mes a mes, casi se ha convertido en un hábito el dedicar todo el tiempo libre posible al proyecto y como consecuencia ha quedado una aplicación completa a pesar de tener muchas cosas que mejorar y diversas líneas de trabajo a seguir para hacerlo más completo, al fin y al cabo, este aprendizaje constante es de lo que trata el mundo de la programación y el desarrollo de software. Y es esto precisamente lo que lo hace tan apasionante.

A nivel del código desarrollado, a medida que se han ido añadiendo funcionalidades se han encontrado multitud de cosas a mejorar, y se ha hecho en la medida de lo posible teniendo en cuenta los ajustados tiempos, a esto ha ayudado una base estructural robusta que ha hecho posible que el proceso de refactorización haya sido siempre cómodo y eficiente. Este proyecto ha servido también para entender lo importante que es hacer las cosas bien, en los proyectos iniciales, cuando era necesario hacer un cambio, el proceso se convertía en un infierno, cada cambio hacía que otras cosas no funcionaran y acababa siendo más fácil deshechar el cambio. Por eso se tomó la determinación de que se evitaría por todos los medios que esto pasara con Profinder. La elección de las tecnologías (como se ha descrito en el capítulo 3) también ha sido de vital importancia, esto se ha notado por ejemplo con el uso de Kotlin y Jeptack Compose.

Profinder ha conseguido cumplir los objetivos propuestos inicialmente: Una aplicación capaz de actuar como intermediario entre distintos tipos de expertos profesionales que busquen ofrecer un servicio y clientes dispuestos a consumirlo. Ofreciendo también una serie de funcionalidades que hagan del proceso anterior algo fluido, descentralizado y sin comisiones.

6.2. Líneas de trabajo futuro

Como se ha mencionado anteriormente, Profinder es una aplicación completa, sin embargo, hay multitud de puntos en los que se podría mejorar y/o completar. A continuación se han mencionado algunos:

- **Notificaciones:** añadiría un toque extra de calidad y personalización a la aplicación el uso de notificaciones, especialmente para funcionalidades como el chat (notificaciones cuando se envían y reciben mensajes), la solicitud de servicios o los trabajos (cuando emipecen y terminen).
- **Animaciones:** el uso de las animaciones en Android es otra de las cosas que añade calidad a las aplicaciones, en Profinder no ha dado tiempo ha implementarlas en su totalidad ya que son un campo muy amplio, sí que se han usado en algunos casos puntuales (como el efecto Shimmer que es una animación, aunque para ello se ha utilizado una biblioteca externa¹) pero la intención a futuro sería implementarlas en muchas otras partes de la aplicación.
- **Testing:** Como se mencionó en el apartado 3.2.3, el testing de la aplicación ha sido el reto más difícil a superar y por falta de tiempo se ha quedado un poco a medias, la idea sería hacer un amplio sistema de tests que incluya test unitarios, de integración y de UI para aportar una mayor robustez y conseguir eliminar los *bugs* que puedan ir surgiendo.
- **Refactor MVI:** a pesar de que el patrón MVI se ha aplicado de una forma correcta en la aplicación, de cara al final del proyecto se ha descubierto que se han cometido algunos errores (también denominados antipatronos) que podrían afectar a la escalabilidad en el futuro, debido a esto una refactorización que arregle estos problemas sería una de las líneas de trabajo futuro.
- **Playstore:** En un inicio, se planteó la posibilidad de publicar la aplicación en la Playstore pero al final no se llevó a cabo debido a que antes de hacerlo entra en juego la seguridad del código, habría que seguir procesos de ofuscamiento, control de entornos y CI/CD. No ha dado tiempo a estudiar con la profundidad necesaria estos temas por lo que se ha clasificado como línea de trabajo futuro.
- **Rol de administrador:** En la especificación de requisitos inicial se declaró el administrador como un actor que gestionara la aplicación desde dentro de la misma, a lo largo del proyecto no se consideró de suficiente importancia dentro de los plazos y se decidió hacer más incapié en otras partes de la aplicación. Sin embargo, podría ser interesante darle una vuelta a este concepto para establecer un mejor control de usuarios dentro de la aplicación.
- **Hacerla multiplataforma:** Kotlin multiplatform es una nueva forma de hacer aplicaciones usando Kotlin y Jeptack Compose, a día de hoy no es una tecnología suficientemente madura como para ser viable pero permitiría hacer aplicaciones para distintos sistemas operativos (IOS, Windows...) con la

¹Repositorio de la biblioteca Shimmer realizada por el usuario valentinilk: [compose-shimmer](#).

misma base de código, esto permitiría aumentar considerablemente el alcance de la aplicación y la haría más completa. No se descarta migrarla a Kotlin multiplatform en el futuro.

Introduction

Motivation

Profinder was born from the concept of streamlining and simplifying the relationships between professionals who offer a service and users who are willing to consume it.

It's a mobile application, initially developed for the Android operating system (though the possibility of making it cross-platform in the future is not ruled out, see Chapter 6.2) where both users and professionals can register, create an account, and interact with each other. The functionalities for each type of actor vary in different aspects, while maintaining some common functionalities.

Below, the idea of the publication, hiring, and rating flow of services is explained:

1. Upon registration, professionals will select the professional category to which they belong, this category can be modified at any time from the 'Edit Profile' screen.
2. Once registered, they will have the possibility to create services, which will be classified into categories and can be public or private (active or inactive).
3. Active services will appear to users, who can request them either from the 'Service List' screen or through the professional's profile.
4. When a user creates a request, it appears to the professional who can accept or reject it. If accepted, the job starts and will be shown as an active job until it is completed.
5. Once the job is finished, the professional will mark it as such and rate the user with stars (from 1 to 5). For the professional, the job will be considered complete.
6. Finally, the user will have the option to rate the professional in the same manner. Once this is done, the job is marked as completed, ending the service flow.

Besides this flow in which each actor has a defined role, there is some common functionality: users and professionals can chat with each other using the chat screen,

where they can discuss additional details, including specific dates and times. All users and professionals can be added to favorite lists for easy access to their profiles. Each actor's profile in the application can be completed with attributes such as a description or a profile photo.

Goals

The world of Android development is vast, the way of designing interfaces compared to web programming is quite different, and it has been constantly changing in recent years, with new technologies emerging each year that make previous ones obsolete, including a recent change in the programming language used.

The main goal of this project is to learn many of the necessary things to become a competent Android developer, starting from scratch to being able to create a robust, beautiful, and scalable application over time.

The application-related goals set from the beginning are shown below:

- Definition of an initial requirement specification, subject to changes throughout the development process, which defines the actors and use cases of the application.
- Once the requirement specification is defined, a series of milestones for the delivery of functionalities will be set.

Technical goals to be carried out in alongside the application goals have also been defined:

- Learn Kotlin in order to achieve a level of competence in the language suitable for development.
- Start making small projects using the Android Views system, the traditional way of developing interfaces in Android, which serves as an initial approach even though it will later be replaced by Jetpack Compose.
- Approach architectures, design patterns, and best practices most commonly used with the aim of starting to make more robust projects that approximate the final project structure.
- Learn the fundamentals of Jetpack Compose and gradually acquire new knowledge by making various test projects.
- Get in touch with the different Firebase services that will serve as the backend of the final application.
- By the end of the development process, it is expected to have obtained a solid knowledge base that will serve as a starting point for a possible future professional career.

Work Plan

To achieve the goals described in the previous section, a work plan represented with a Gantt chart has been established at the beginning of the project. This goals are divided —as in the previous section— into technical and application goals since the initial idea is to meet both type of goals in parallel over time.

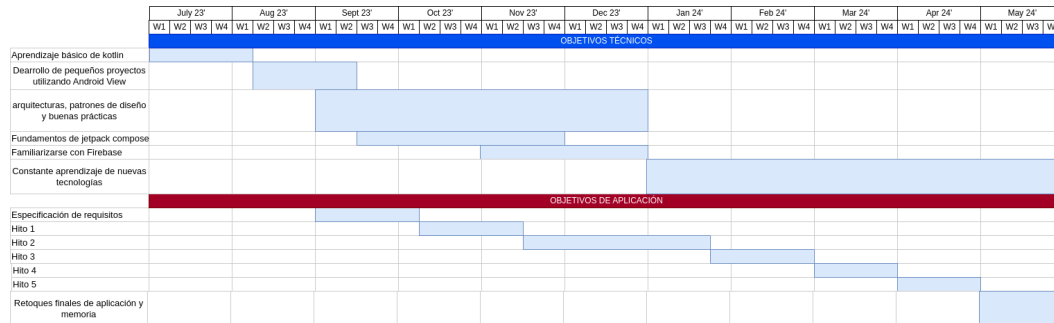


Figure 6.1: Gantt chart of the project

Regarding the project milestones, a list of the use cases to be developed in each milestone is shown below:

- Milestone 1
 - User registration/deletion.
 - Modify user data.
 - User login/logout.
 - Configure app.
- Milestone 2
 - Change professional status.
 - Register/delete service.
 - Modify service.
 - List registered services.
 - Add/delete/modify categories of offered services.
- Milestone 3
 - Search service.
 - Configure search.
 - View professional.
 - View client.
 - Add/remove professional to/from favorites list.
 - Add/modify/remove client from favorites list.

- View favorites list.
- Milestone 4
 - Hire service.
 - Chat with professional/client.
 - Rate client/professional.
 - Respond to hiring request.
- Milestone 5
 - View professional/client data/statistics.
 - Search clients/professionals.
 - Modify client/professional/offered services data.
 - Delete users.

Since the use cases were an initial approximation, as the development process progressed, some use cases were implemented earlier, later, or discarded because they didn't fit well into the application. Despite this, the initial planning has been very useful as a guide and has served to correctly mark delivery deadlines, assimilating the process to what it would be like in a real world environment.

Conclusions and Future Work

In this chapter, conclusions have been drawn and possible future lines of work for the project have been outlined.

Conclusions

Since this final degree project was proposed in July 2023, approximately 11 months have passed from these conclusions are being written. It has been a journey that started from having no specific knowledge of Android, where new concepts have been continuously learnt month by month. Dedicating all possible free time to the project almost became a habit, resulting in a well-rounded application despite having many aspects to improve and various lines of work to pursue to make it complete. Ultimately, this constant learning is what programming and software development are all about, and it is precisely this that makes it so exciting.

Regarding the developed code, as functionalities were added, many areas for improvement were found and addressed in the best way possible given the tight timelines. A robust structural base has made the refactoring process always comfortable and efficient. This project has also highlighted the importance of doing things right; in the initial projects, making a change often turned into a nightmare, with each change causing other things to break, making it easier to abandon the change. Therefore, the decision was made to avoid this at all costs with Profinder. The choice of technologies has also been crucial, noticeable, for example, with the use of Kotlin and Jetpack Compose.

Profinder has managed to meet the initially proposed objectives: an application capable of acting as an intermediary between different types of professional experts looking to offer a service and clients willing to consume it, offering functionalities that make the previous process smooth, decentralized, and commission-free.

Future Work Lines

As mentioned earlier, Profinder is a complete application; however, there are many areas where it could be improved and/or completed. Below are some of them:

- **Notifications:** Adding notifications would provide an extra touch of quality

and personalization to the application, especially for functionalities such as the chat (notifications when messages are sent and received), service requests, or jobs (when they start and finish).

- **Animations:** The use of animations in Android is another aspect that adds quality to applications. In Profinder, there has not been enough time to implement them fully as it is a broad field. Animations have been used in some specific cases (like the Shimmer effect, which is an animation, although an external library was used for this²), but the intention for the future would be to implement them in many other parts of the application.
- **Testing:** As mentioned in section 3.2.3, testing the application has been the most challenging task to overcome, and due to lack of time, it has remained somewhat incomplete. The idea would be to create a comprehensive testing system that includes unit tests, integration tests, and UI tests to provide greater robustness and eliminate potential bugs.
- **Refactor MVI:** Although the MVI pattern has been correctly applied in the application, towards the end of the project, some mistakes (also known as anti-patterns) were discovered that could affect scalability in the future. Therefore, a refactoring to address these issues would be one of the future work lines.
- **Playstore:** Initially, the possibility of publishing the app on the Playstore was considered, but it was ultimately not done because of security related issues. Code obfuscation, environment control, and CI/CD processes would need to be followed. There was not enough time to study these topics in the necessary depth, so it is classified as a future work line.
- **Administrator Role:** In the initial requirements specification, the administrator was declared as an actor who would manage the application from within. Throughout the project, it was not considered sufficiently important within the deadlines, and more emphasis was placed on other parts of the application. However, it could be interesting to revisit this concept to establish better user control within the application.
- **Making it Cross-Platform:** Kotlin Multiplatform is a new way of making applications using Kotlin and Jetpack Compose. Today, it is not a sufficiently mature technology to be viable, but it would allow making applications for different operating systems (IOS, Windows, etc.) with the same codebase. This would significantly increase the application's reach and make it more complete. Migrating to Kotlin Multiplatform in the future is not ruled out.

²Shimmer library repository by user valentinilk: [compose-shimmer](#).

Bibliografía

Nobody actually creates perfect code the first time around, except me. But there's only one of me.

Linus Torvalds

FUENTES, R. ¿qué es y cómo funciona la arquitectura mvi? ejemplos —desarrollo android— kotlin. *Medium*, 2020. Disponible en <https://medium.com/@robercoding/qu%C3%A9-es-y-c%C3%B3mo-funciona-la-arquitectura-mvi-desarrollo-android-kotlin-e6a161e1b2db> (último acceso, Mayo, 2024).

ISAACSON, W. *The Innovators: How a Group of Hackers, Geniuses and Geeks Created the Digital Revolution*. Simon And Schuster, 2015.

KHAN, M. H. Recyclerview vs. lazyrow/lazycolumn: Choosing the right layout for dynamic lists in android. *Medium*, 2023. Disponible en [https://medium.com/@humzakhalid94/recyclerview-vs-lazyrow-lazycolumn-choosing-the-right-layout-for-dynamic-lists-in](https://medium.com/@humzakhalid94/recyclerview-vs-lazyrow-lazycolumn-choosing-the-right-layout-for-dynamic-lists-in-android-1e1b2db) (último acceso, Mayo, 2024).

MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson, 2008.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

Especificación de requisitos

En este apéndice se especifican los actores y casos de uso de la aplicación Profinder. Se trata de una especificación inicial por lo que tanto actores como casos de uso han sido cambiados, ampliados o recortados a lo largo del proceso de desarrollo.

A.1. Actores

A continuación se describen los distintos actores de la aplicación:

- **Usuario:** Un usuario es cualquier persona que se dé de alta como tal, debe haberse creado una cuenta e iniciado sesión para ser identificado, estos son los que contratan los servicios que los profesionales ofrecen, entre otras funciones los usuarios pueden ver el mapa de profesionales en tiempo real y filtrar los datos que le aparecen según la categoría de trabajo seleccionada.
- **Profesional:** Un profesional es cualquier persona que se dé de alta como tal, su proceso de registro es ligeramente diferente al de un usuario normal ya que deben especificar la categoría en la que son profesionales, así como otros detalles que sean importantes para su profesión.
- **Administrador:** El administrador de la aplicación es el que se encarga de la organización y el correcto funcionamiento de la misma, podrá añadir o eliminar categorías, bloquear usuarios y/o profesionales, así como responder a mensajes dirigidos a soporte, los administradores deben ser asignados una vez creada la cuenta desde fuera de la aplicación.

A.2. Casos de Uso

Los casos de uso se han clasificado en cuatro categorías según el actor principal del mismo.

A.2.1. Casos de uso generales

Requisito	Registro/baja	
Identificador	1.1	
Prioridad	Alta	
Precondición	En caso de baja: tener una cuenta creada.	
Descripción	Los actores de la aplicación tendrán que crear una cuenta para poder interactuar con la misma, asimismo tendrán la capacidad de dar de baja su cuenta cuando lo deseen.	
Entrada	Nombre de usuario, correo electrónico, contraseña.	
Salida	Actor registrado/dado de baja	
Secuencia normal	Paso	Acción
	1	El actor abre la aplicación y se le muestra la pantalla para iniciar sesión con un botón específico para registrarse.
	2	se selecciona la opción 'Registrarse'.
	3	El actor rellena el formulario con los datos de entrada y pulsa 'Enviar'.
	4	El sistema procesa el formulario y crea la cuenta.
Postcondición	Se ha creado la cuenta.	
Excepciones	Paso	Acción
	4	Los datos introducidos son incorrectos o no cumplen los requisitos. No se crea la cuenta y se avisa al usuario.
Comentarios	Este caso de uso permite registrar o dar de baja de la aplicación a actores de la misma.	
Actores	Usuario, Profesional	

Tabla A.1: Registro/baja

Requisito	Modificar datos	
Identificador	1.2	
Prioridad	Media	
Precondición	Haber iniciado sesión.	
Descripción	Una vez creada una cuenta, tanto profesionales como usuarios podrán modificar sus datos de perfil.	
Entrada	Datos a cambiar.	
Salida	Nuevos datos.	
Secuencia normal	Paso	Acción
	1	El actor se dirigirá al apartado de ‘Mi perfil’ y seleccionará la opción ‘Modificar perfil’.
	2	Dentro de esta pantalla se modificarán todos los datos deseados.
Postcondición	Se han modificado los datos de perfil deseados.	
Excepciones	Paso	Acción
	2	Los nuevos datos no son válidos. No se guarda la modificación.
Comentarios	Este caso de uso permite modificar datos de su perfil a los actores.	
Actores	Usuario, Profesional	

Tabla A.2: Modificar datos

Requisito	Login/Logout	
Identificador	1.3	
Prioridad	Alta	
Precondición	Haber creado una cuenta previamente, en caso de logout tener sesión iniciada.	
Descripción	Para poder interactuar con la aplicación, tanto usuarios como profesionales tendrán que iniciar sesión en la aplicación. Los usuarios que ya hayan iniciado sesión y quieran cerrarla tendrán la opción de hacerlo.	
Entrada	En caso de login: correo electrónico/nombre de usuario, contraseña.	
Salida	N.A.	
Secuencia normal de login	Paso	Acción
	1	El actor abre la aplicación y se le muestra la pantalla de 'Iniciar sesión'.
	2	Se muestra una pantalla con el formulario de inicio de sesión. El usuario introduce los datos de entrada y pulsa el botón 'Iniciar sesión'.
Secuencia normal de logout	Paso	Acción
	1	El actor se dirige a la sección 'Mi perfil' donde se le mostrarán múltiples opciones de gestión de su cuenta.
	2	Dentro de estas opciones se encontrará la opción 'Cerrar sesión'. El actor pulsará el botón.
Postcondición	Se ha iniciado sesión/se ha cerrado sesión.	
Excepciones	Paso	Acción
	2 (login)	Los campos rellenados no concuerdan con ningún actor del sistema. No se completa el login.
Comentarios	Este caso de uso permite iniciar sesión en la aplicación, así como cerrar la misma.	
Actores	Usuario, Profesional, Administrador	

Tabla A.3: Login/Logout

Requisito	Configurar app	
Identificador	1.4	
Prioridad	Baja	
Precondición	Haber iniciado sesión.	
Descripción	Dentro de la app se podrán configurar aspectos como el tema de la aplicación (claro/oscuro).	
Entrada	Nuevos datos de configuración.	
Salida	Configuración modificada.	
Secuencia normal	Paso	Acción
	1	El actor se dirigirá a la sección de ‘Mi Perfil’ y entre las opciones seleccionará ‘Configuración de la aplicación’.
	2	Dentro de esta pantalla el usuario cambiará los valores deseados.
Postcondición	Se han cambiado los parámetros de la aplicación deseados.	
Excepciones	Paso	Acción
	2	Los nuevos parámetros de configuración no son correctos. Se mantiene la configuración anterior.
Comentarios	Este caso de uso permite modificar la configuración de la aplicación.	
Actores	Usuario, Profesional	

Tabla A.4: Configurar app

Requisito	Ver lista de favoritos	
Identificador	1.5	
Prioridad	Media	
Precondición	El Usuario o Profesional debe estar autenticado en la aplicación y tener al menos un elemento en su lista de favoritos.	
Descripción	Permite al Usuario o Profesional ver la lista de elementos marcados como favoritos (por ejemplo, Clientes o Profesionales) para un acceso rápido y conveniente.	
Entrada	Selección de la opción para ver la lista de favoritos.	
Salida	Lista de elementos marcados como favoritos con detalles relevantes.	
Secuencia normal	Paso	Acción
	1	El Usuario o Profesional accede a la sección de lista de favoritos.
	2	Selecciona la opción para ver la lista de favoritos.
	3	Visualiza la lista de elementos marcados como favoritos con sus detalles.
Postcondición	El Usuario o Profesional ve la lista de elementos marcados como favoritos.	
Excepciones	Paso	Acción
	2	Si la lista de favoritos está vacía, se muestra un mensaje indicando que no hay elementos en la lista.
Comentarios	N.A.	
Actores	Usuario, profesional	

Tabla A.5: Ver lista de favoritos

Requisito	Valorar cliente/profesional	
Identificador	1.6	
Prioridad	Alta	
Precondición	Haber iniciado sesión, haber recibido el trabajo de un profesional o haber realizado un trabajo a un usuario.	
Descripción	Una vez realizado/recibido un trabajo, los usuarios/profesionales podrán valorar a ese profesional/usuario así como denunciarlo si lo vieran necesario.	
Entrada	Datos de valoración, mensaje con más detalles (opcional)	
Salida	N.A.	
Secuencia normal	Paso	Acción
	1	Una vez recibido/realizado el trabajo, se mostrará la opción de valorar al profesional/usuario.
	2	El usuario/profesional rellena el formulario de valoración.
Postcondición	Se ha valorado al profesional/usuario.	
Excepciones	Paso	Acción
	2	Los datos de entrada no son correctos. No se completa la valoración.
Comentarios	N.A.	
Actores	Usuario, profesional	

Tabla A.6: Valorar cliente/profesional

Requisito	Chatear con cliente/profesional	
Identificador	1.7	
Prioridad	Media	
Precondición	Haber iniciado sesión, estar en el perfil de un profesional/-cliente.	
Descripción	En el perfil de los profesionales/clientes habrá un botón para iniciar un chat privado que servirá para aclarar dudas, pedir/dar presupuestos para cosas concretas, etc. . .	
Entrada	botón de chat, mensaje(s) a enviar.	
Salida	N.A.	
Secuencia normal	Paso	Acción
	1	Una vez en el perfil del profesional/usuario se pulsa el botón ‘Chat privado’.
	2	Al hacerlo, se abre un chat entre usuario y profesional donde se podrán enviar mensajes.
	3	El usuario/profesional envía los mensajes deseados.
Postcondición	Se ha creado un chat privado entre usuario y profesional.	
Excepciones	N.A.	
Comentarios	Este caso de uso permite establecer una conversación entre usuario y profesional.	
Actores	Usuario, profesional	

Tabla A.7: Chatear con profesional

A.2.2. Casos de uso de usuarios

Requisito	Configurar búsqueda	
Identificador	2.1	
Prioridad	Media	
Precondición	Haber iniciado sesión.	
Descripción	A la hora de buscar el servicio de un profesional, los usuarios tendrán la opción de configurar los parámetros de la búsqueda como por ejemplo, seleccionar una categoría u ordenar por valoración.	
Entrada	Datos de configuración de búsqueda.	
Salida	N.A.	
Secuencia normal	Paso	Acción
	1	Dentro de la aplicación, el usuario seleccionará la opción ‘Buscar profesional’.
	2	Se configurarán todos los parámetros de la búsqueda y se pulsará el botón ‘Buscar’.
Postcondición	Se han configurado los parámetros de la búsqueda.	
Excepciones	N.A.	
Comentarios	Este caso de uso permite establecer unos parámetros determinados de búsqueda.	
Actores	Usuario	

Tabla A.8: Configurar búsqueda

Requisito	Buscar servicio	
Identificador	2.2	
Prioridad	Media	
Precondición	Haber iniciado sesión, haber configurado la búsqueda.	
Descripción	Una vez configurados los parámetros de búsqueda, se mostrarán los servicios que cumplen los parámetros de la búsqueda.	
Entrada	Servicio a buscar.	
Salida	Resultados de servicios que coinciden con la búsqueda.	
Secuencia normal	Paso	Acción
	1	El usuario podrá recorrer una lista con todos los servicios ofrecidos que cumplen los filtros establecidos.
Postcondición	Se ha realizado una búsqueda de servicio.	
Excepciones	N.A.	
Comentarios	Este caso de uso sirve para que los usuarios puedan buscar servicios de acuerdo con unos parámetros	
Actores	Usuario	

Tabla A.9: Buscar servicio

Requisito	Consultar profesional	
Identificador	2.3	
Prioridad	Media	
Precondición	Haber iniciado sesión, hay profesionales disponibles.	
Descripción	Los usuarios tendrán la posibilidad de consultar el perfil de los distintos profesionales: sus datos (categoría, honorarios, etc. . .), sus valoraciones y otras estadísticas.	
Entrada	Profesional a consultar.	
Salida	Detalles del profesional consultado	
Secuencia normal	Paso	Acción
	1	El usuario realiza una búsqueda de profesional o escoge uno de los que se muestra en el mapa.
	2	Al pulsar en el profesional se abrirá su perfil con todos los datos de ese profesional.
Postcondición	Se ha consultado el profesional seleccionado.	
Excepciones	N.A.	
Comentarios	Este caso de uso permite obtener más detalles acerca de un profesional determinado.	
Actores	Usuario	

Tabla A.10: Chatear con profesional

Requisito	Contratar servicio	
Identificador	2.4	
Prioridad	Alta	
Precondición	Haber iniciado sesión.	
Descripción	Cuando un usuario encuentre el servicio que necesita tendrá la posibilidad de contratarlo y el profesional decidirá si tomar o no el trabajo.	
Entrada	Servicio elegido, respuesta del profesional.	
Salida	N.A.	
Secuencia normal	Paso	Acción
	1	Una vez seleccionado el profesional, en la pantalla del perfil del mismo habrá una opción llamada ‘Contratar servicio’, el usuario pulsará el botón.
	2	El sistema procesa y valida la contratación.
	3	La propuesta de trabajo llega al profesional que decide si la acepta o rechaza.
Postcondición	Se ha contratado el servicio.	
Excepciones	Paso	Acción
	2	Fallo en el proceso y validación de la petición. Se avisa al usuario y no se envía la propuesta.
Comentarios	N.A.	
Actores	Usuario	

Tabla A.11: Contratar servicio

Requisito	Añadir/Quitar Profesional de lista de favoritos	
Identificador	2.5	
Prioridad	Media	
Precondición	Haber iniciado sesión, estar en el perfil de un profesional.	
Descripción	Los usuarios tendrán la posibilidad de añadir o quitar a profesionales de su lista de favoritos, lista en la que podrán tener un acceso rápido a los perfiles de dichos profesionales.	
Entrada	Botón de añadir/quitar de favoritos.	
Salida	Lista de favoritos modificada.	
Secuencia normal	Paso	Acción
	1	Una vez dentro del perfil de un profesional, el usuario dispondrá de un botón de favoritos.
	2	Dependiendo de si quiere añadir o quitar al profesional de favorito lo activará o desactivará.
Postcondición	Se ha añadido/quitado un profesional en la lista de favoritos.	
Excepciones	N.A.	
Comentarios	N.A.	
Actores	Usuario	

Tabla A.12: Contratar servicio

A.2.3. Casos de uso de profesionales

Requisito	Cambiar estado de profesional	
Identificador	3.1	
Prioridad	Alta	
Precondición	Haber iniciado sesión como profesional.	
Descripción	El profesional puede cambiar su estado entre activo/inactivo/trabajando para indicar a los usuarios su disponibilidad actual.	
Entrada	Nuevo estado.	
Salida	N.A.	
Secuencia normal	Paso	Acción
	1	El profesional se dirige al apartado de ‘Mi Perfil’.
	2	Dentro del mismo selecciona la opción ‘Cambiar estado’.
	3	El profesional selecciona el nuevo estado que figura en su perfil.
Postcondición	Se ha cambiado el estado de profesional.	
Excepciones	N.A.	
Comentarios	N.A.	
Actores	Profesional	

Tabla A.13: Cambiar estado de profesional

Requisito	Dar de alta/baja servicio	
Identificador	3.2	
Prioridad	Alta	
Precondición	Haber iniciado sesión como profesional.	
Descripción	Los profesionales tendrán la opción de dar de alta y baja los servicios, en el primer caso esto significa que sube un servicio y en el segundo que lo retira de la oferta de servicios.	
Entrada	Servicio a dar de baja/alta.	
Salida	Confirmación de la baja/alta del servicio.	
Secuencia normal: alta de servicio	Paso	Acción
	1	Cuando un usuario contrata el servicio de un profesional, a este le llega una solicitud de servicio.
	2	El profesional abre la oferta, donde verá los detalles del servicio.
	3	Pulsa el botón 'Aceptar servicio'.
Secuencia normal: baja de servicio	Paso	Acción
	1	El profesional abre el servicio que había aceptado con anterioridad.
	2	Entre los detalles del servicio se muestra la opción 'Dar de baja servicio' el profesional pulsa el botón.
	3	Se muestra un mensaje de confirmación de baja de servicio.
Postcondición	Se ha dado de alta/baja un servicio.	
Excepciones	N.A.	
Comentarios	N.A.	
Actores	Profesional	

Tabla A.14: Dar de alta/baja servicio

Requisito	Modificar servicio	
Identificador	3.3	
Prioridad	Alta	
Precondición	Haber iniciado sesión como profesional, tener al menos un servicio dado de alta.	
Descripción	Permite al Profesional modificar la información de un servicio que ofrece.	
Entrada	Detalles actualizados del servicio.	
Salida	Confirmación de la modificación del servicio.	
Secuencia normal	Paso	Acción
	1	El profesional navega hasta la sección de gestión de servicios.
	2	Selecciona el servicio que desea modificar.
	3	Realiza las modificaciones necesarias en los detalles del servicio.
	4	Confirma la acción de modificación.
Postcondición	El servicio se actualiza con la nueva información en el perfil del Profesional.	
Excepciones	Paso	Acción
	3	Si no se proporciona la información necesaria, se muestra un mensaje de error.
	4	Si la operación falla por algún motivo, se notifica al Profesional.
Comentarios	Este caso de uso permite a los Profesionales mantener actualizada la información de sus servicios.	
Actores	Profesional	

Tabla A.15: Modificar servicio

Requisito	Listar servicios dados de alta	
Identificador	3.4	
Prioridad	Baja	
Precondición	Haber iniciado sesión como profesional, tener al menos un servicio dado de alta.	
Descripción	Permite al Profesional ver una lista de todos los servicios que ha dado de alta.	
Entrada	Selección de la opción para listar servicios.	
Salida	Lista de servicios con detalles.	
Secuencia normal	Paso	Acción
	1	El profesional navega hasta la sección de gestión de servicios.
	2	Selecciona la opción para listar sus servicios.
Postcondición	El Profesional puede ver una lista de los servicios que ha dado de alta.	
Excepciones	Paso	Acción
	2	Si no tiene servicios dados de alta, se muestra un mensaje indicando que no tiene servicios registrados.
Comentarios	N.A.	
Actores	Profesional	

Tabla A.16: Listar servicios dados de alta

Requisito	Contestar solicitud de contratación.	
Identificador	3.5	
Prioridad	Alta	
Precondición	El Profesional debe estar autenticado en la aplicación y haber recibido una solicitud de contratación.	
Descripción	Permite al Profesional aceptar o rechazar una solicitud de contratación de un Usuario.	
Entrada	Solicitud de contratación.	
Salida	Confirmación de la respuesta a la solicitud.	
Secuencia normal	Paso	Acción
	1	El profesional recibe una notificación de solicitud de contratación.
	2	Accede a la solicitud y selecciona aceptar o rechazar.
	3	Confirma la respuesta.
Postcondición	La solicitud de contratación se responde y se notifica al Usuario.	
Excepciones	Paso	Acción
	3	Si la solicitud ha caducado, se informa al Profesional.
Comentarios	N.A.	
Actores	Usuario, Profesional	

Tabla A.17: Contestar solicitud de contratación.

Requisito	Consultar cliente	
Identificador	3.6	
Prioridad	Media	
Precondición	Haber iniciado sesión como profesional y estar trabajando o haber trabajado con el cliente.	
Descripción	Permite al Profesional consultar información sobre el cliente, incluyendo sus datos, las valoraciones que ha recibido y otras estadísticas relevantes.	
Entrada	Selección del Cliente a consultar.	
Salida	Información detallada del Cliente, incluyendo sus datos personales, valoraciones recibidas y estadísticas.	
Secuencia normal	Paso	Acción
	1	El profesional accede a la sección de consulta de Clientes.
	2	Selecciona el Cliente cuya información desea consultar.
	3	Visualiza los datos y estadísticas del cliente.
Postcondición	El Profesional obtiene información sobre el cliente.	
Excepciones	N.A.	
Comentarios	Este caso de uso brinda al Profesional acceso a información relevante sobre los Clientes con los que ha interactuado.	
Actores	Usuario, Profesional	

Tabla A.18: Consultar cliente.

Requisito	Añadir/Modificar/Quitar cliente de lista de favoritos.	
Identificador	3.7	
Prioridad	Media	
Precondición	Haber iniciado sesión como profesional.	
Descripción	Permite al Profesional agregar, modificar o quitar Clientes de su lista de favoritos para un acceso más rápido y conveniente.	
Entrada	Selección de la acción (añadir, modificar o quitar) y Cliente seleccionado.	
Salida	N.A.	
Secuencia normal	Paso	Acción
	1	El profesional navega hasta la sección de gestión de favoritos.
	2	Selecciona la acción deseada (añadir, modificar o quitar) y el cliente correspondiente.
	3	Confirma la acción.
Postcondición	La lista de favoritos se actualiza según la acción realizada.	
Excepciones	Paso	Acción
	2	Si el Cliente ya está en la lista de favoritos y se selecciona 'añadir', se muestra un mensaje informativo.
Comentarios	Este caso de uso permite al Profesional gestionar su lista de favoritos para un acceso más rápido a los Clientes preferidos.	
Actores	Usuario, Profesional	

Tabla A.19: Añadir/Modificar/Quitar cliente de lista de favoritos.

A.2.4. Casos de uso de administrador

Requisito	Añadir/Eliminar/Modificar categorías de servicios ofrecidos.	
Identificador	4.1	
Prioridad	Alta	
Precondición	Estar registrado en la aplicación como administrador.	
Descripción	Permite al Administrador gestionar las categorías de servicios ofrecidos, incluyendo la adición, eliminación o modificación de categorías existentes.	
Entrada	Selección de la acción (añadir, eliminar o modificar) y detalles de la categoría.	
Salida	Confirmación de la acción realizada en las categorías.	
Secuencia normal	Paso	Acción
	1	El administrador accede a la sección de gestión de categorías de servicios.
	2	Selecciona la acción deseada (añadir, eliminar o modificar) y proporciona los detalles necesarios.
	3	Confirma la acción.
Postcondición	Las categorías se actualizan según la acción realizada.	
Excepciones	Paso	Acción
	3	Si la categoría ya existe y se selecciona 'añadir', se muestra un mensaje informativo.
	3	Si la categoría no existe y se selecciona 'eliminar' o 'modificar', se muestra un mensaje informativo.
Comentarios	Este caso de uso permite al Administrador gestionar las categorías de servicios para mantener la organización de la plataforma.	
Actores	Administrador	

Tabla A.20: Añadir/eliminar/Modificar categorías de servicios ofrecidos.

Requisito	Consultar datos/estadísticas de profesionales/clientes.	
Identificador	4.2	
Prioridad	Media	
Precondición	Estar registrado en la aplicación como administrador.	
Descripción	Permite al Administrador acceder a datos y estadísticas relacionadas con Profesionales y Clientes, lo que le permite realizar análisis y tomar decisiones informadas.	
Entrada	Selección de Profesional o Cliente a consultar.	
Salida	Información detallada y estadísticas del Profesional o Cliente seleccionado.	
Secuencia normal	Paso	Acción
	1	El administrador accede a la sección de consulta de datos/estadísticas.
	2	Selecciona el Profesional o Cliente cuya información desea consultar.
	3	Visualiza los datos y estadísticas.
Postcondición	El Administrador obtiene información detallada sobre el Profesional o Cliente seleccionado.	
Excepciones	Paso	Acción
	2	Si no se encuentra información para el Profesional o Cliente seleccionado, se muestra un mensaje informativo.
Comentarios	Este caso de uso proporciona al Administrador acceso a datos relevantes para la toma de decisiones y la gestión de la plataforma.	
Actores	Administrador	

Tabla A.21: Consultar datos/estadísticas de profesionales/clientes.

Requisito	Buscar clientes/profesionales.	
Identificador	4.3	
Prioridad	Media	
Precondición	Estar registrado en la aplicación como administrador.	
Descripción	Permite al Administrador buscar Clientes o Profesionales dentro de la aplicación según diversos criterios.	
Entrada	Criterios de búsqueda.	
Salida	Lista de Clientes o Profesionales que coinciden con los criterios de búsqueda.	
Secuencia normal	Paso	Acción
	1	El administrador accede a la sección de búsqueda de Clientes o Profesionales.
	2	Ingresa los criterios de búsqueda.
	3	Realiza la búsqueda.
	4	Visualiza la lista de resultados.
Postcondición	El Administrador obtiene una lista de Clientes o Profesionales que coinciden con los criterios de búsqueda.	
Excepciones	Paso	Acción
	3	Si no se encuentran resultados que coincidan con los criterios, se muestra un mensaje informativo.
Comentarios	Este caso de uso permite al Administrador buscar y acceder a perfiles de Clientes o Profesionales de manera eficiente.	
Actores	Administrador	

Tabla A.22: Buscar clientes/profesionales.

Requisito	Modificar datos de clientes/profesionales/servicios ofrecidos.	
Identificador	4.4	
Prioridad	Alta	
Precondición	Estar registrado en la aplicación como administrador.	
Descripción	Permite al administrador realizar modificaciones en los datos de clientes, profesionales o servicios ofrecidos cuando sea necesario.	
Entrada	Selección del tipo de modificación (Cliente, Profesional o Servicio) y detalles de la modificación.	
Salida	Confirmación de la modificación realizada.	
Secuencia normal	Paso	Acción
	1	El administrador accede a la sección de modificación de datos.
	2	Selecciona el tipo de modificación deseada (Cliente, Profesional o Servicio) y proporciona los detalles necesarios.
	3	Confirma la modificación.
Postcondición	Los datos se actualizan según la modificación realizada.	
Excepciones	N.A.	
Comentarios	Este caso de uso permite al Administrador gestionar y mantener actualizados los datos de la plataforma.	
Actores	Administrador	

Tabla A.23: Modificar datos de clientes/profesionales/servicios ofrecidos.

Requisito	Dar de baja usuarios.	
Identificador	4.5	
Prioridad	Alta	
Precondición	Estar registrado en la aplicación como administrador.	
Descripción	Permite al Administrador dar de baja a Usuarios de la aplicación en casos de incumplimiento de términos y condiciones u otras razones legítimas.	
Entrada	Selección del Usuario a dar de baja y motivo de la baja.	
Salida	Confirmación de la baja del Usuario.	
Secuencia normal	Paso	Acción
	1	El administrador accede a la sección de gestión de bajas de Usuarios.
	2	Selecciona el Usuario a dar de baja y especifica el motivo.
	3	Confirma la baja del Usuario.
Postcondición	El Usuario queda dado de baja de la aplicación.	
Excepciones	N.A.	
Comentarios	Este caso de uso permite al Administrador mantener la integridad de la plataforma al dar de baja a Usuarios que incumplen las reglas o políticas.	
Actores	Administrador	

Tabla A.24: Dar de baja usuarios.

Apéndice B

Interfaces de Profinder diseñadas en figma

En este apéndice se han adjuntado capturas de los diseños de pantallas en Figma. El proyecto se puede ver aquí: [Link al proyecto](#).

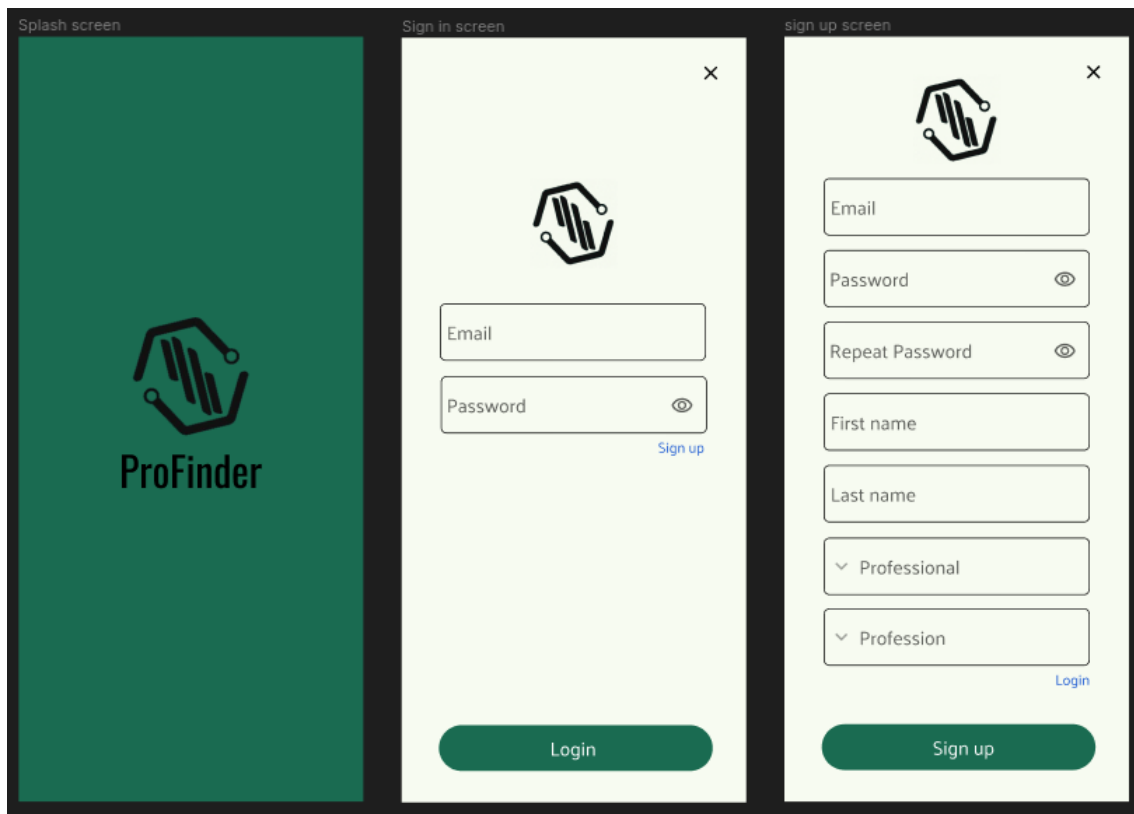


Figura B.1: Splash Screen, Login Screen y Sing Up Screen.

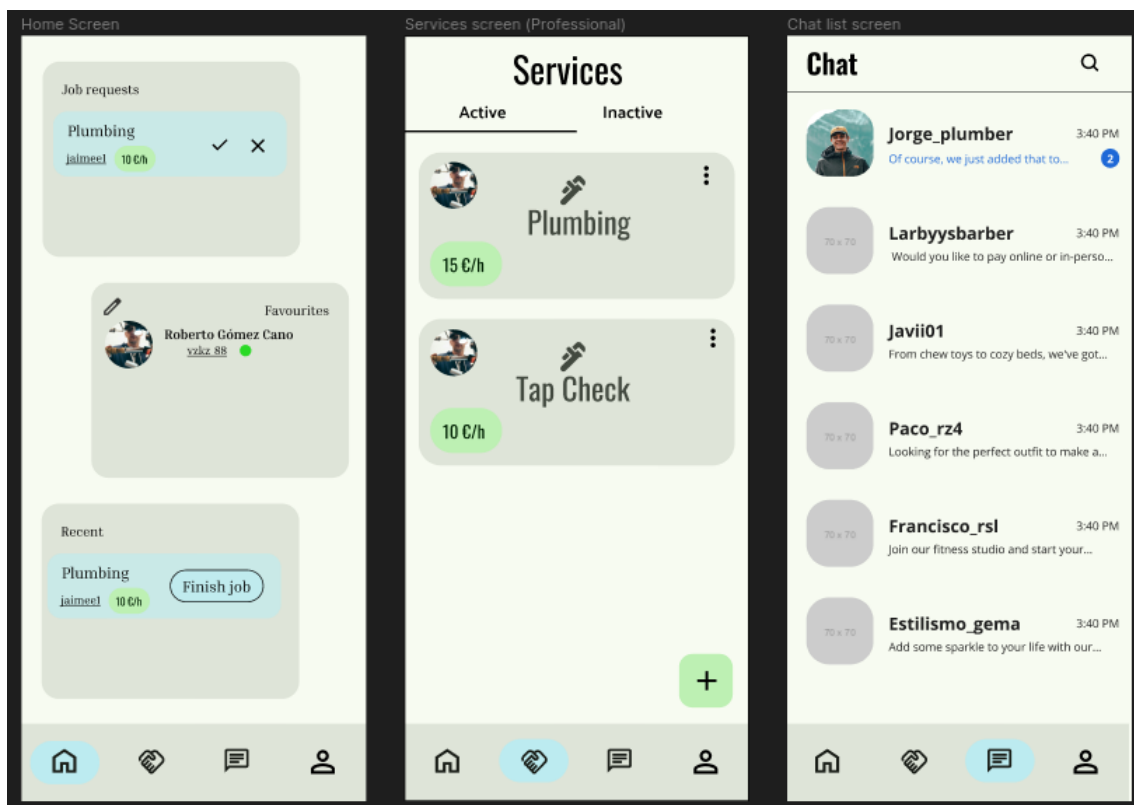


Figura B.2: Home Screen, Services Screen (professional) y Chat Screen.

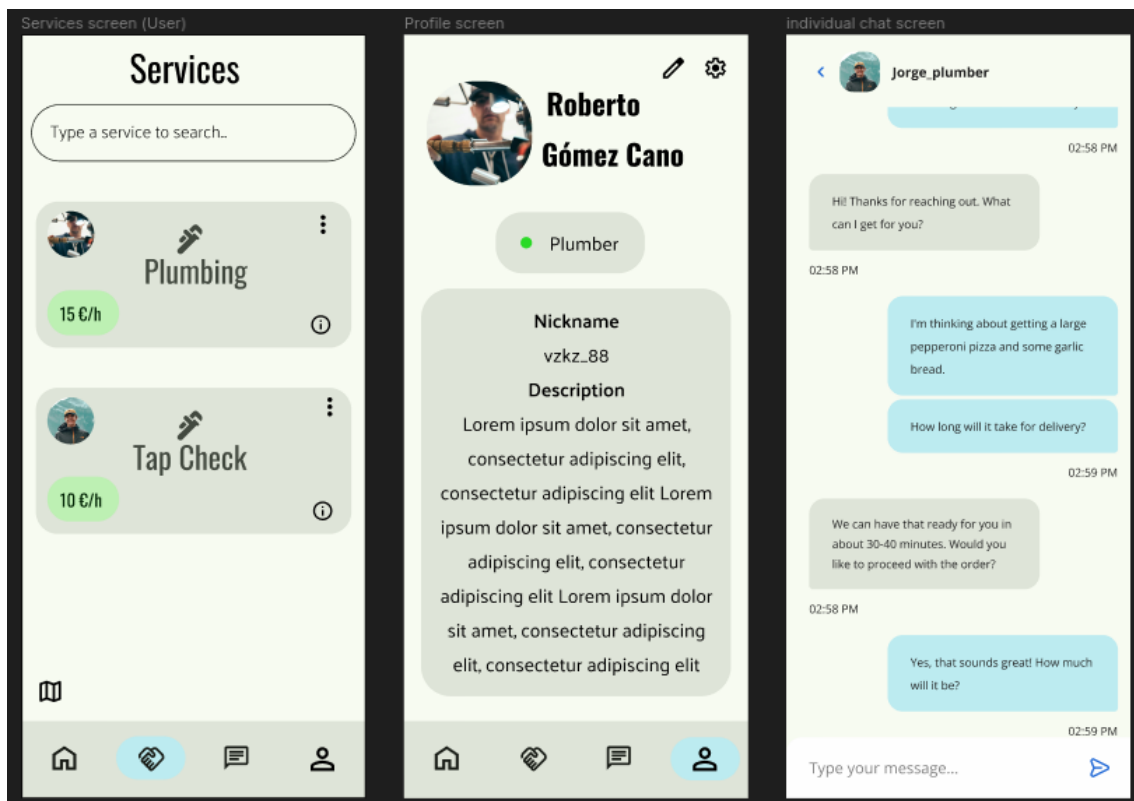


Figura B.3: Services Screen (user), Profile Screen y Individual Chat Screen.

Diseño de las bases de datos

En este apéndice se han adjuntado capturas de pantalla de los diseños usando la herramienta Drawio de las dos bases de datos utilizadas en Profinder, en primer lugar la base de datos Realtime y en segundo lugar la base de de datos Firestore (explicadas en el apartado 5.2)

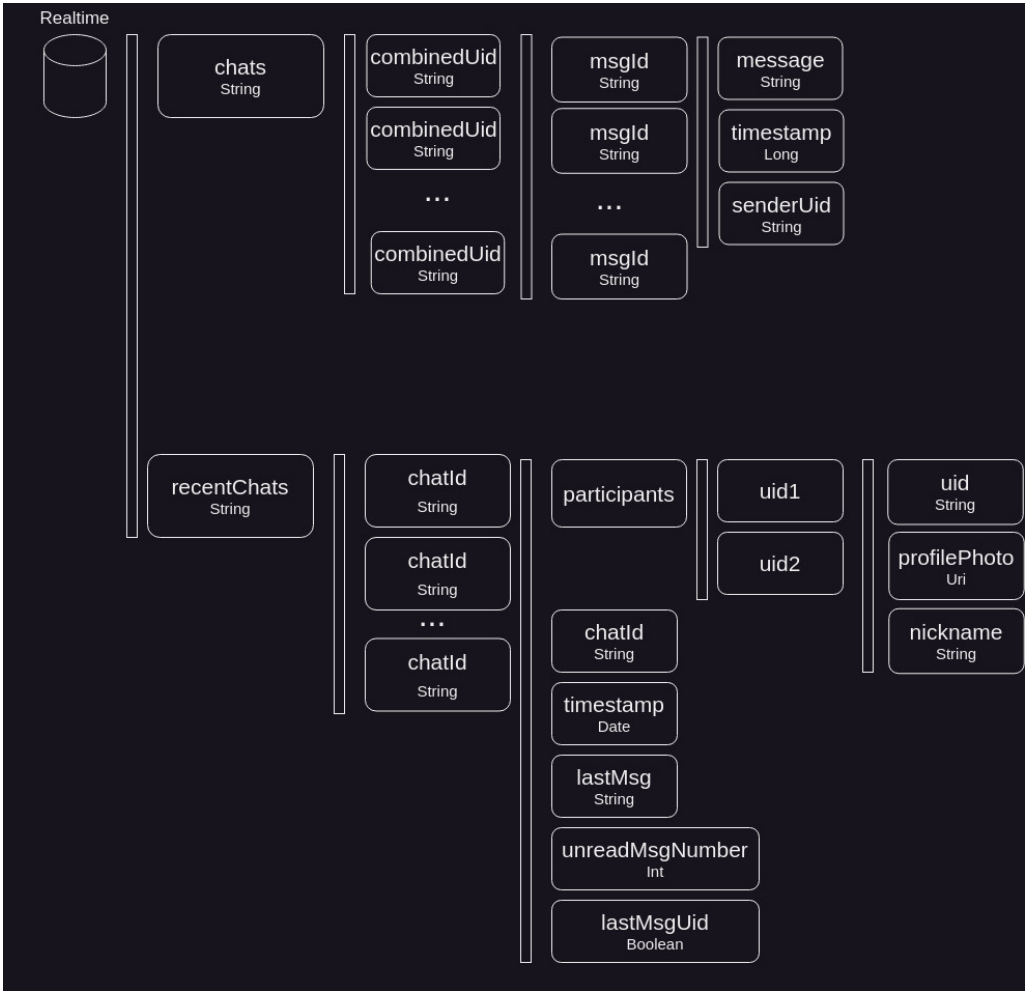


Figura C.1: Diseño de la base de datos Realtime usando Drawio.

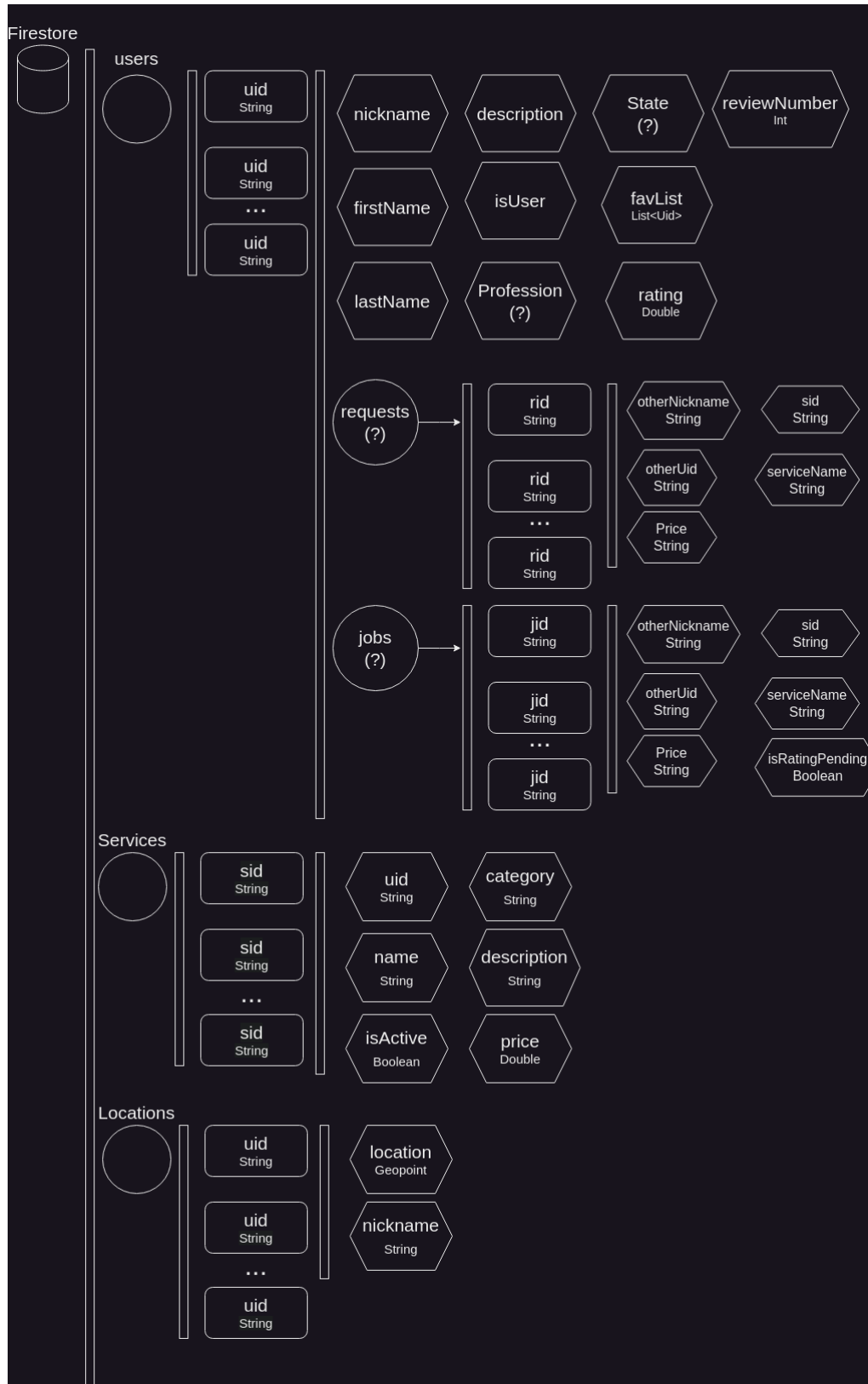


Figura C.2: Diseño de la base de datos Firestore usando Drawio.

URLs referenciadas en la memoria

- Uber: <https://www.uber.com>
- Habitissimo: <https://www.habitissimo.es>
- Booksy: <https://booksy.com/en-us>
- Upwork: <https://www.upwork.com>
- Introducción a Android Views: <https://www.studytonight.com/android/introduction-to-views>
- Android Studio: <https://developer.android.com/studio/intro>
- IntelliJ Idea: <https://www.jetbrains.com/idea>
- Java: <https://www.java.com>
- Scala: <https://www.scala-lang.org>
- Obsidian: <https://obsidian.md>
- Figma: <https://www.figma.com>
- Material Design: <https://m3.material.io>
- Drawio: <https://www.drawio.com/>
- Mockk Android: <https://mockk.io/ANDROID.html>
- Git: <https://git-scm.com/>
- Github: <https://github.com/>
- Lazygit: <https://github.com/jesseduffield/lazygit>
- Kotlin: <https://kotlinlang.org/>
- What is Kotlin: <https://www.plainconcepts.com/es/kotlin-android/>

- Jetpack Compose: <https://developer.android.com/develop/ui/compose>
- Lambdas en Kotlin: <https://kotlinlang.org/docs/lambdas.html>
- Dagger Hilt: <https://dagger.dev/hilt/>
- Android SDK for Maps: <https://developers.google.com/maps/documentation/android-sdk>
- Android Maps Compose: <https://github.com/googlemaps/android-maps-compose>
- Compose Destinations: <https://github.com/raamcosta/compose-destinations>
- Coil: <https://coil-kt.github.io/coil/compose/>
- Glide: <https://github.com/bumptech/glide>
- Picasso: <https://square.github.io/picasso/>
- Free Software Foundation: <https://www.fsf.org/>
- GNU: <https://www.gnu.org/>
- Firefox: <https://www.mozilla.org/es-ES/firefox/>
- Neovim: <https://neovim.io/>
- Linux: <https://www.linux.org/pages/>
- Viewmodel: <https://developer.android.com/topic/libraries/architecture/viewmodel>
- Patrón MVVM: <https://builtin.com/software-engineering-perspectives/mvvm-architecture>
- Android Intent: <https://developer.android.com/guide/components/intents-filters>
- Vídeo Phillip Lackner sobre gestión de errores: <https://www.youtube.com/watch?v=MILN2vs20e0>
- Expresiones When en Kotlin: <https://kotlinlang.org/docs/control-flow.html#when-expression>
- Biblioteca Compose Shimmer: <https://github.com/valentinilk/compose-shimmer>
- Kotlin multiplatform: <https://kotlinlang.org/docs/multiplatform.html>